MARCH 1994

WRL
Technical Note TN-42

Speculative Execution and Instruction-Level Parallelism

David W. Wall

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 is a second research laboratory located in Palo Alto, 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 technical note. We use this form for rapid distribution of technical material. Usually this represents research in progress. Research reports are normally accounts of completed research and may include material from earlier technical notes.

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@decwrl.dec.com`

UUCP: `decwrl!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.

Speculative Execution
and
Instruction-Level Parallelism

David W. Wall

March 1994

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

Abstract

Full exploitation of instruction-level parallelism by superscalar and similar architectures requires *speculative execution*, **in which we are willing to issue a potential future instruction early even though an intervening branch may send us in another direction entirely. Speculative execution can be based either on branch prediction, where we explore the most likely path away from the branch, or on branch fan-out, in which we explore both paths and sacrifice some hardware parallelism for the sake of not being entirely wrong. Recent techniques for branch prediction have greatly improved its potential success rate; we measure the effect this improvement has on parallelism. We also measure the effect of fan-out, alone and also in combination with a predictor. Finally, we consider the effect of** *fallible* **instructions, those that might lead to spurious program failure if we execute them speculatively; simply refusing to do so can drastically reduce the parallelism.**

1 Introduction

Recent years have seen a great deal of interest in *multiple-issue machines* [1, 6, 9], machines that can issue several mutually independent instructions in the same cycle. These machines exploit the parallelism that programs exhibit at the instruction level.

It is important to know how much parallelism is available in typical applications. Machines providing a high degree of multiple-issue would be of little use if applications did not display that much parallelism. The available parallelism depends strongly on how hard we are willing to work to find it. Recent studies studies [4, 5, 6, 13, 14, 15, 16, 17] have led to a growing consensus that high levels of parallelism are available only by doing *speculative execution*, in which we can issue an instruction whose data dependencies are satisfied even though its control dependencies are not. That is, we issue a potential future instruction early even though an intervening branch may send us in another direction entirely.

There are two approaches to the selection of instructions to execute speculatively. We can do *branch prediction,* trying to guess whether a conditional branch will be taken so we know which of the two possible paths to follow in selecting instructions. Or we can *fan out* and select instructions from both possible paths, spending some of our machine parallelism for the assurance that at least some of the instructions we speculatively execute will be useful. It is possible to use a combination of the two, fanning out part of the time and predicting the rest of the time.

This paper presents results concerning three questions. First, recent work in branch prediction [8, 10, 18, 19] has shown how to use very large predictors to improve the performance of hardware predictors from around 92% success to around 98%. What effect does this have on instruction-level parallelism? Second, how useful is a fan-out capability, both by itself and in combination with a predictor? Third, on some architectures certain instructions must not be executed speculatively because they can cause run-time exceptions. Does this cripple a multiple-issue machine, or can it be tolerated?

We start with an overview of branch prediction and fan-out techniques. We then describe our experimental environment, based like many others on trace-based simulation. Finally we present our results, and some conclusions.

2 Branch prediction

Branch prediction can be done statically or dynamically. Static prediction based on the direction of the branch or other heuristics is only somewhat effective, but prediction based on a profile of a previous run of the application is successful around 90% of the time. Dynamic prediction is normally done in hardware, with the prediction for a given branch based on recent events in the execution. A common hardware branch predictor [7, 12] maintains a table of saturating two-bit counters. Low-order bits of a branch's address provide an index into this table, associating a counter with each branch; if the table is small then the program space wraps around, possibly associating the same counter to several branches across the program. We predict that a branch will be taken if the associated counter is 2 or 3, and otherwise predict not taken. Later, when the branch is resolved, we increment the counter if it *was* taken, and otherwise decrement it. A predictor of 512 counters is successful about as often as a profile, but unfortunately increasing the size of the table does not help much; the success rate levels off at 92% or 93% regardless of the table size.

Recent studies have explored more sophisticated hardware prediction using *branch histories* [10, 18, 19]. 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 [18]. 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 with a small period.

Sometimes the behavior of one branch is correlated with the behavior of another. A *global-history* predictor [18] 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 total predictor size.

An interesting variation is the *gshare* predictor [8], 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 [8] 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 total predictor size was small, but did quite well indeed when large.

3 Branch fan-out

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.

In some respects fan-out duplicates the benefits of branch prediction, but they can also work together. We explore both paths up to the fan-out limit, and then explore only the predicted path beyond that point.

4 Fallible instructions

In most architectures, some instructions can fail, causing an exception. Examples are memory references, which can cause segmentation violations, and floating-point operations, which can cause several kinds of traps. Speculatively executing a fallible instruction is dangerous, because it might make a correct program fail; to avoid this, the hardware must somehow make the exception itself speculative, so that the failure does not occur until we are sure that the instruction should have been executed.

The easy way out is simply to refuse to speculatively execute a fallible instruction. This is likely to degrade the parallelism, since it will also delay safe instructions that depend on the fallible instruction, but it eliminates the need for hardware trickiness.

5 Simulation environment

To study the effects of these issues on instruction-level parallelism, we used the trace-based simulator described in detail in an earlier report [17]. An instruction trace of the application is passed, one instruction at a time, to the scheduler. The scheduler places each instruction into some cycle of a sequence of pending cycles, subject to dependencies with previously scheduled instructions. Whether there is a dependency is determined by the parallelism model we use. If the model does not include branch prediction, for example, then each instruction appearing after

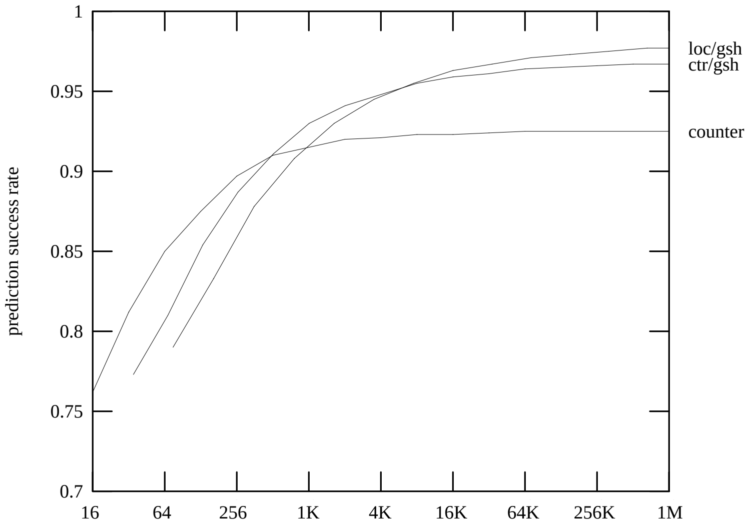


Figure 1: Fraction of branches predicted correctly by three different prediction schemes, as a function of the total number of bits in the predictor

a branch in the trace must be scheduled after that branch in the pending cycles. If the model does include branch prediction, in contrast, we can schedule later instructions into cycles before the branch, if the predictor is successful; otherwise we must assume that a real machine would have speculatively executed instructions from the wrong path, and would only start looking down the correct path when execution of the branch instruction reveals the misprediction.[1]

The simulator uses a greedy scheduling algorithm, placing each instruction as early as possible in the pending cycles, given the instructions that preceded it in the trace. Each cycle can hold a maximum of 64 instructions, and the entire sequence of pending cycles can hold 2048 instructions. When the number of pending instructions exceeds that number, we "issue" the first cycle, which prevents us from scheduling any more instructions in it.

For the purposes of this paper, the parallelism model simulated is specified by four parameters: branch prediction and fan-out, fallibility, register renaming, and memory disambiguation. The full system is somewhat more flexible than this.

In this paper we are interested in the effect of varying the size of the branch predictor. Different predictors do the best in different regions of this spectrum of size. Figure 1 shows the harmonic

---

[1] This approach to a missed prediction ignores the possibility that code could be moved from after the point where the paths rejoin to a position before the paths split apart. Recognizing such opportunities is difficult in hardware but feasible in a software scheduler [4, 14].

mean of the success rates of three predictors for twelve SPEC92 benchmarks, as the predictor size varies. The two-bit-counter predictor does best for predictor sizes up to 512 bits. A predictor built by combining a counter predictor and a gshare predictor does best in the middle range up through 4K bits, and a combination of a local predictor and a gshare predictor works best above 4K bits. Throughout this paper, when we speak of a predictor of a particular size, the range that includes this size will determine the prediction technique used.

Branch fan-out is a little trickier to model. We want to explore both paths away from a branch, but the simulator has only the correct instruction trace to work from and therefore cannot actually schedule instructions from paths not taken. Exploring these false paths on a real machine would use up hardware parallelism, however, especially since we will likely have to schedule another branch before the first is issued and resolved. We model this approximately by assuming that there is a *fan-out limit* on the number of branches we can look past. If our model has a non-zero fan-out limit, we can handle branches beyond that limit either by giving up or by conventional branch prediction.

There are two kinds of instructions we can consider fallible: floating-point binary operations and memory-reference instructions. In this paper we arbitrarily allowed only heap references to fail, on the perhaps generous assumption that program analysis or language semantics could preclude the failure of references to stack or static data.

This paper is not directly concerned with the effects of register or memory dependencies. To provide a small selection of contexts for our exploration of branch analysis and fallibility, however, we assumed four different base models. The *alpha* model assumes perfect memory disambiguation, so that a store conflicts with a load or store only if the two actually reference the same word in memory, and assumes an infinite number of registers with a perfect renaming scheme, so that we never have output dependencies or antidependencies between registers. The *beta* model also assumes perfect memory disambiguation, but assumes 64 CPU registers and 64 FPU registers, managed dynamically by a hardware renaming scheme using an LRU discipline (relative, of course, to the position in the scheduled cycles rather than in the instruction trace). The *gamma* model assumes perfect memory disambiguation and *no* register renaming, so that register conflicts are determined by the registers actually allocated by the DECstation compiler. The *delta* model assumes no register renaming and simple but very conservative memory disambiguation by *instruction inspection*, a common technique used in compile-time instruction-level pipeline schedulers: two instructions do not conflict if (a) they use they use the same base register but

	register renaming	*memory disambiguation*
alpha	infinite	perfect
beta	64 int, 64 fp	perfect
gamma	no renaming	perfect
delta	no renaming	inspection

Figure 2: The four base models of register renaming and memory disambiguation

different displacements, or (b) one uses a register known to point to the stack and the other one known to point to the global data area. Figure 2 summarizes these four models.

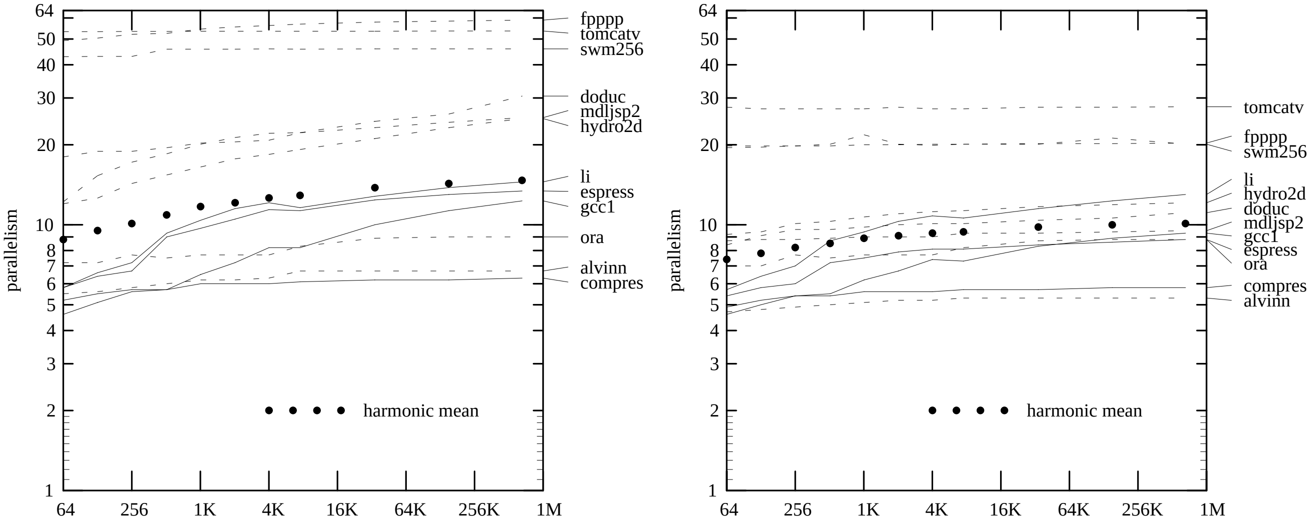
In all four of these models we assume that all indirect jumps (chiefly procedure returns, calls to procedure variables, and case-statement indexed jumps) are predicted perfectly. Procedure returns are easy to predict with simple hardware, but other jumps are less so. Assuming perfect jump prediction is therefore generous but probably not consequential; indirect jumps are rare enough in the programs we tested that jump prediction has a significant effect on parallelism only when branch prediction is also perfect.

All of our simulations were done with a set of twelve programs from the SPEC92 suite. (The rest of them run too long for our simulation to be feasible.) We usually gave them the official "small" data sets where possible, and in the case of tomcatv and alvinn we modified the value of a constant to reduce the number of iterations of the outer loop.

6 Results

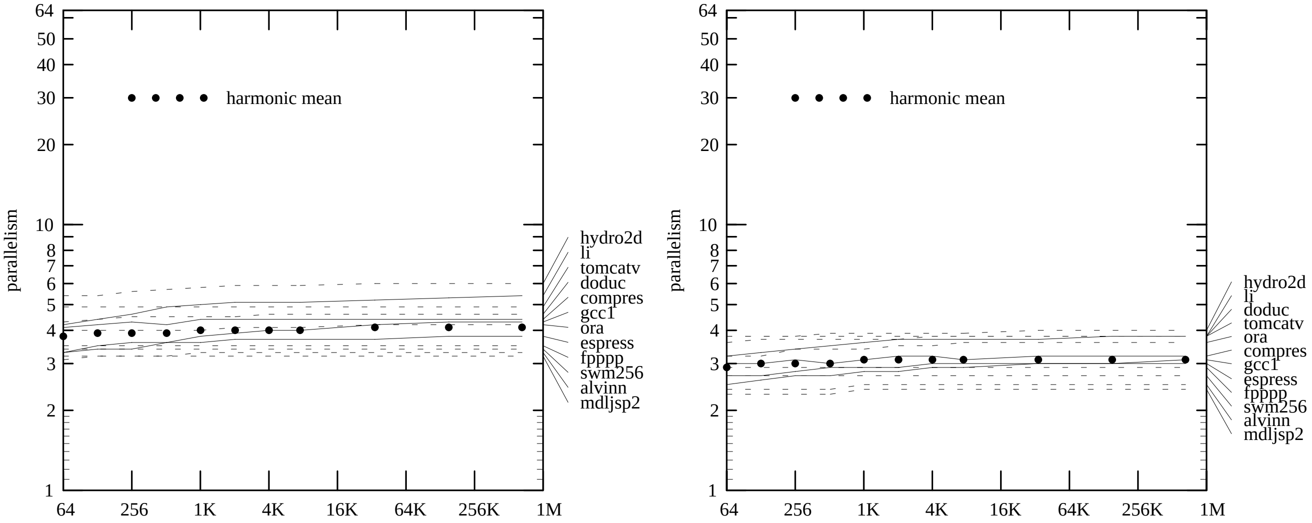
Our first experiment measured the parallelism as the total size of the branch predictor increased. As described earlier, different predictors have better success rates in different size ranges, so we use different predictors for the small, intermediate, and large predictor sizes. (This is why some benchmarks show a sudden change around 512 bits or 4K bits.) Figure 3 shows the results for each of our four base models. The solid curves are integer benchmarks; the dotted curves are floating-point benchmarks.

Under the *alpha* model, with infinite registers and perfect memory disambiguation, we see



(a) infinite registers and perfect memory disambiguation

(b) 64 renamed registers and perfect memory disambiguation



(c) no register renaming and perfect memory disambiguation

(d) no register renaming and memory disambiguation by inspection

Figure 3: Effect of branch predictor size on parallelism

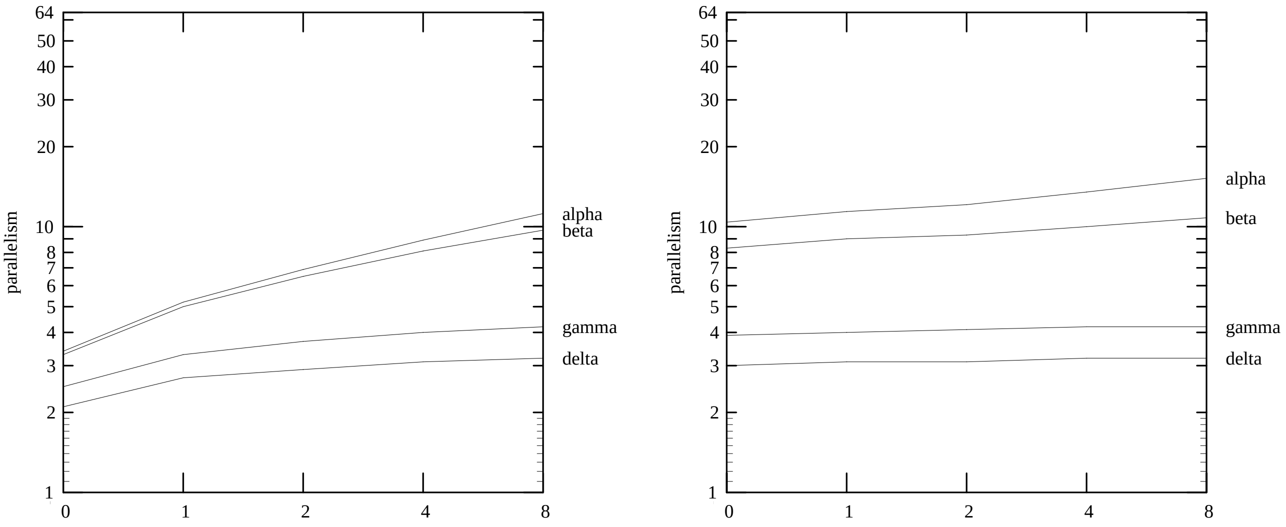


Figure 4: Effect of fan-out on parallelism without branch prediction (left) and with a 0.5-kilobit branch predictor (right)

that a few programs benefit considerably from large predictors. Gcc1 continues to improve even as we reach a predictor of three-quarters of a megabit. Both li and espresso improve 40% between 1 kilobit and 1 megabit; the harmonic mean of the improvements over that range is 25%. It is interesting that the programs least sensitive to the size of the predictor are those most parallel and those least parallel.

Under the *beta* model we see roughly the same behavior, though it is not as pronounced. Now the mean payoff of the biggest predictor over the 1-kilobit predictor is about 14%. When we eliminate first register renaming and then perfect memory disambiguation, we see that the advantage of a very large predictor evaporates almost completely. The largest improvement from 1Kb is 13%, but few programs do even close to this well; the mean is more like 2%.

Unsurprisingly, we see that a very large branch predictor can be helpful, but only if we get everything else right.

Next we consider the effects of fanning out at branches. Figure 4 shows the mean parallelism over the 12 programs as the fan-out limit increases, for each of the four base models. The left-hand graph assumes that the fan-out capability is working alone, without subsequent branch prediction: when the fan-out limit is reached we can look beyond no more branches for instructions to issue.

The right-hand graph assumes that fan-out is followed by branch prediction: when the fan-out limit we can continue to look past branches for instructions, but only along the predicted path. The predictor used is a modest one, a simple counter-based predictor of 256 entries. [2]

Without branch prediction, a little fan-out helps a lot, even in the poorer base models. Fanning out past just 1 level of branching improves the parallelism of *gamma* and *delta* by around 30%, and of *alpha* and *beta* by around 50%. Increasing the fan-out limit continues to improve things significantly, but the effects are not as dramatic.

Interestingly, fanning out even to a level of 8 branches gives us a parallelism in each model that is nearly the same as the parallelism from using the half-kilobit predictor with no fan-out at all. Adding eight levels of fan-out to this predictor improves the parallelism somewhat, by 30-45% in *alpha* and *beta*, and by about 7% in *gamma* and *delta*.

Thus an ambitious fan-out capability could be an adequate substitute for branch prediction, though it is hard to imagine the circumstances in which it would be easier to implement. Adding branch prediction to even a modest predictor does not buy us much unless (again) we do a very good job of handling register and memory dependencies.

We assumed that two kinds of instructions could fail: binary floating-point operations, and heap memory references. In the actual traces, of course, these operations never fail; since we could not know that in advance, we model their fallibility by insisting that they always be scheduled later than any previous branch. We also experimented with models in which only one of these two classes of instructions are fallible. These proved uninteresting, because the behavior of the twelve SPEC92 programs is bimodal: the integer programs do essentially no floating-point operations, and the floating-point programs make few or no heap references. In either case, assuming that only one could fail gave results essentially identical to assuming that neither or both could fail.

Figure 5 shows the results, for the four base models and two different predictor sizes. We have separated out the integer from the floating-point programs, and present the harmonic mean parallelism for each. The upper curve in each pair is the parallelism without fallible instructions; the lower is with fallible instructions.

Fallibility has a larger effect on the integer programs than on the floating-point programs. It

---

[2] The simulator's viewpoint is the reverse of the hypothetical hardware's. The simulator schedules each successive instruction into one of the pending cycles, so to implement a fan-out of n without prediction it allows the instruction to be scheduled earlier than the previous n branches, but not the $n + 1$st. To implement fan-out followed by branch prediction, we tentatively predict every branch, and allow an instruction to be scheduled before any number of successfully predicted branches, preceded by n more branches whether predicted successfully or not. In other words, the instruction must be scheduled after the nth branch before the last incorrectly predicted branch.

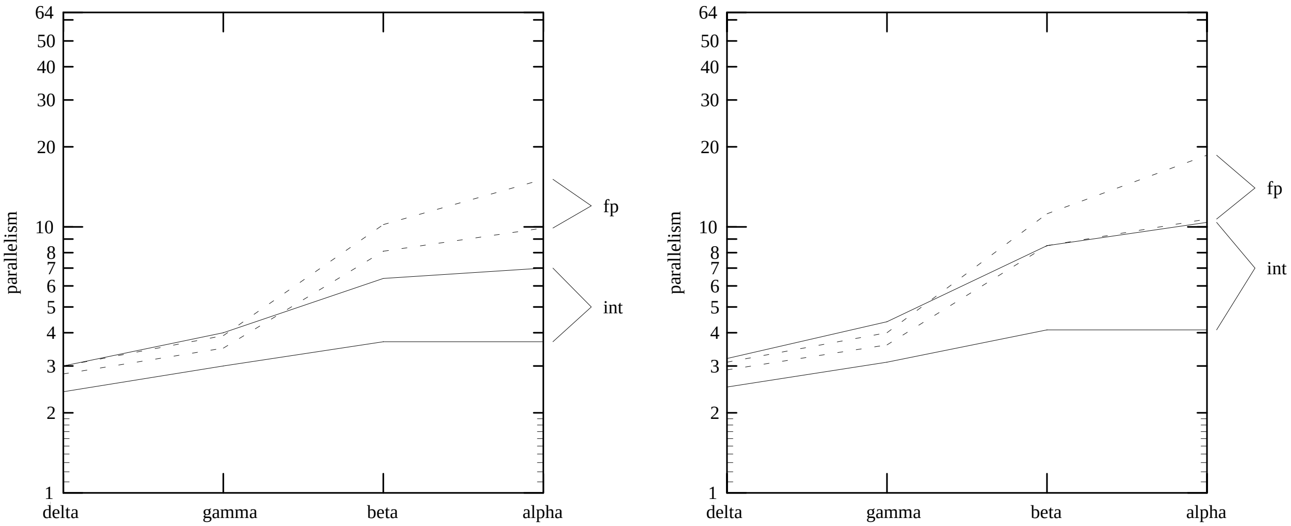


Figure 5: Effect of fallible instructions on parallelism with 0.5-kilobit branch predictor (left) and with 0.7-megabit branch predictor (right)

reduces the parallelism of integer programs so much that the most ambitious model has barely half again the parallelism of the poorest.

Fallibility has its greatest effect on the more ambitious models. It can cut the parallelism of a good model in half but rarely reduces the smaller parallelism of a poorer model by more than a fifth. Evidently (and perhaps obviously) the more different kinds of bottlenecks to scheduling you have, the less another one matters.

7 Conclusions

The qualitative conclusions of this study should come as no great surprise, though we hope the quantitative results will serve as useful hints to the architecture and compiler communities.

Very good branch prediction from megabit history-based predictors can significantly improve parallelism, though the magnitude of this improvement was not as great as we had hoped to see. The payoff of a large predictor is probably negligible unless we also take strong action to reduce false register dependencies and disambiguate memory references.

Fanning out across many levels of branches can in principle be a substitute for modest branch prediction, though a large predictor has no trouble beating it. Since fan-out is likely to be harder to implement than ordinary prediction, it is probably more interesting to note that adding fan-out to prediction can improve it. As before, however, the improvement is significant only if we have false register and memory conflicts well under control.

These results confirm that we really need to work with a combination of very good techniques if we want to achieve high levels of parallelism. It is therefore important to note that refusing to execute fallible instructions speculatively can halve the parallelism of the more ambitious models. Techniques that allow failures to be postponed until we are sure they were supposed to happen [2, 3, 11] are essential to the full exploitation of instruction-level parallelism.

References

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

[2] Roger A. Bringmann, Scott A. Mahlke, Richard E. Hank, John C. Gyllenhaal, and Wen-mei W. Hwu. Speculative execution exception recovery using write-back suppression. *26th Annual International Symposium on Microarchitecture*, 214–223, December 1993. Published as *SIG MICRO Newsletter 24*.

[3] Harry Dwyer and H. C. Torng An out-of-order superscalar processor with speculative execution and fast, precise interrupts. *25th Annual International Symposium on Microarchitecture*, 272–281, December 1992. Published as *SIG MICRO Newsletter 23*,(1&2).

[4] Joseph A. Fisher. Global code generation for instruction-level parallelism: trace scheduling-2. Technical Report HPL-93-43, Hewlett-Packard Laboratories, June 1993.

[5] Joseph A. Fisher and Stefan M. Freudenberger Predicting conditional branch directions from previous runs of a program. *Fifth International Symposium on Architectural Support for Programming Languages and Operating Systems*, pp. 85–95, October 1992. Published as *Computer Architecture News 20* (special issue), *Operating Systems Review 26* (special issue), *SIGPLAN Notices 27* (9).

[6] 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).

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

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

[9] 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.

[10] 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).

[11] Anne Rogers and Kai Li. Software support for speculative loads. *Fifth International Symposium on Architectural Support for Programming Languages and Operating Systems*, pp. 38–50, October 1992. Published as *Computer Architecture News 20* (special issue), *Operating Systems Review 26* (special issue), *SIGPLAN Notices 27* (9).

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

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

[14] Michael D. Smith, Mark Horowitz, and Monica Lam. Efficient superscalar performance through boosting. *Fifth International Symposium on Architectural Support for Programming Languages and Operating Systems*, pp. 248–259, October 1992. Published as *Computer Architecture News 20* (special issue), *Operating Systems Review 26* (special issue), *SIGPLAN Notices 27* (9).

[15] Kevin B. Theobald, Guang R. Gao, and Laurie Hendren. On the limits of program parallelism and its smoothability.. *25th Annual International Symposium on Microarchitecture*, 10–19, December 1992. Published as *SIG MICRO Newsletter 23,*(1&2).

[16] 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.

[17] David W. Wall. Limits of instruction-level parallelism. Research Report 93/6, Digital Western Research Laboratory, November 1993. An earlier version appeared in *Fourth International Symposium on Architectural Support for Programming Languages and Operating Systems*, 176–188, April 1991.

[18] 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).

[19] 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.

‘‘Tradeoffs in Two-Level On-Chip Caching.’’
Norman P. Jouppi & Steven J.E. Wilton.
WRL Research Report 93/3, October 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.

‘‘Fluoroelastomer Pressure Pad Design for Microelectronic Applications.’’
Alberto Makino, William R. Hamburgen, John S. Fitch.
WRL Research Report 93/7, November 1993.

‘‘Link-Time Optimization of Address Calculation on a 64-bit Architecture.’’
Amitabh Srivastava, David W. Wall.
WRL Research Report 94/1, February 1994.

‘‘ATOM: A System for Building Customized Program Analysis Tools.’’
Amitabh Srivastava, Alan Eustace.
WRL Research Report 94/2, March 1994.

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.