References

[AS92] Todd M. Austin and Gurindar S. Sohi. Dynamic dependency analysis of ordinary programs. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 342–351, 1992.

[AS95] Todd M. Austin and Gurindar S. Sohi. Zero-cycle loads: Microarchitecture support for reducing load latency. In *Proceedings of the 28th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 82–92, December 1995.

[ASU86] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers principles, techniques, and tools*. Addison-Wesley, Reading, MA, 1986.

[BK95] Peter Bannon and Jim Keller. Internal architecture of Alpha 21164 microprocessor. *COMPCON 95*, 1995.

[Cag96] Andrew Cagney. PSIM User Guide. Available as ftp://cambridge.cygnus.com/pub/psim/index.html, August 1996.

[CMMP95] Thomas M. Conte, Kishore N. Menezes, Patrick M. Mills, and Burzin A. Patel. Optimization of instruction fetch mechanisms for high issue rates. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 333–344, Santa Margherita Ligure, Italy, June 22–24, 1995.

[CS95] Robert P. Colwell and R. Steck. A 0.6um BiCMOS process with Dynamic Eexecution. In *Proceedings of ISSCC*, 1995.

[DSB86] M. Dubois, C. Scheurich, and F.A. Briggs. Memory access buffering in multiprocessors. In *Proceedings of the 13th International Symposium on Computer Architecture*, pages 434–443, June 1986.

[GCM+94] David M. Gallagher, William Y. Chen, Scott A. Mahlke, John C. Gyllenhaal, and Wen mei W. Hwu. Dynamic memory disambiguation using the memory conflict buffer. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 183–193, San Jose, California, October 4–7, 1994.

[HSS94] Andrew S. Huang, Gert Slavenburg, and John P. Shen. Speculative disambiguation: A compilation technique for dynamic memory disambiguation. In *Proceedings of the 21st International Symposium on Computer Architecture*, pages 200–210, Chicago, IL, April 1994.

[Joh91] M. Johnson. *Superscalar Microprocessor Design*. Prentice Hall, Englewood Cliffs, NJ, 1991.

[Jou88] N. P. Jouppi. Architectural and organizational tradeoffs in the design of the MultiTitan CPU. Technical Report TN-8, DEC-wrl, December 1988.

[Kel96] Jim Keller. The 21264: A superscalar Alpha microprocessor with out-of-order execution. In *Proceedings of the Microprocessor Forum*, October 1996.

[LS96] Mikko H. Lipasti and John Paul Shen. Exceeding the dataflow limit via value prediction. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, December 1996.

[LW92] M.S. Lam and R.P. Wilson. Limits of control flow on parallelism. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 46–57, 1992.

[LWS96] Mikko H. Lipasti, Christopher B. Wilkerson, and John P. Shen. Value locality and load value prediction. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, October 1996.

[McF93] Scott McFarling. Combining branch predictors. Technical report, Digital Equipment Corp, June 1993.

[MDO94] Ann Marie Grizzaffi Maynard, Colette M. Donnelly, and Bret R. Olszewski. Contrasting characteristics and cache performance of technical and multi-user commercial workloads. In *Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 145–156, San Jose, October 1994.

[ONH+96] Kunle Olukotun, Basem A. Nayfeh, Lance Hammond, Ken Wilson, and Kunyung Chang. The case for a single-chip multiprocessor. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, October 1996.

[Pat96] Yale N. Patt. A 10 IPC microprocessor, the next challenge. ICCD '96 Tutorial, October 1996.

[Per96] Paul Perez. The PA-8200: A high-performance follow-on to the PA-8000. In *Proceedings of the Microprocessor Forum*, October 1996.

[RBS96] Eric Rotenberg, Steve Bennett, and Jim Smith. Trace cache: a low latency approach to high bandwidth instruction fetching. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, December 1996.

[RF72] Edward M. Riseman and Caxton C. Foster. The inhibition of potential parallelism by conditional jumps. *IEEE Transactions on Computers*, pages 1405–1411, December 1972.

[US95] Augustus K. Uht and Vijay Sindagi. Disjoint eager execution: An optimal form of speculative execution. In *Proceedings of the 28th Annual ACM/IEEE International Symposium on Microarchitecture*, 1995.

[Wal91] D.W. Wall. Limits of instruction-level parallelism. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 176–188, Santa Clara, CA, 1991.

[YP91] T. Y. Yeh and Y. N. Patt. Two-level adaptive training branch prediction. In *Proceedings of the 24th Annual International Symposium on Microarchitecture*, pages 51–61, November 1991.

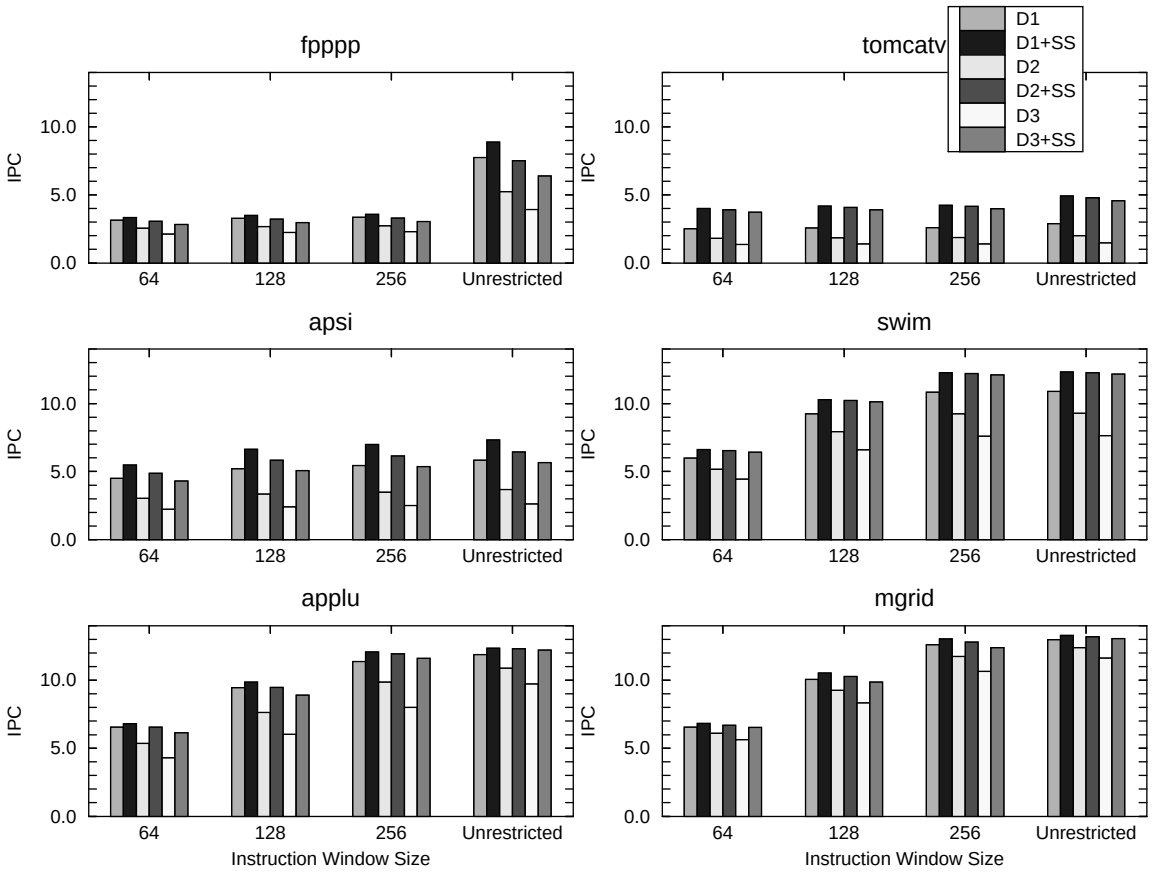

Figure 9-6 SPECFP95 Superspeculation with Restricted Machine Model. The sustained IPC for a dispatch width of 16 and various instruction window sizes is shown for single-cycle dispatch (D1), 2-cycle pipelined dispatch (D2), and 3-cycle pipelined dispatch (D3), all with and without *superspeculation* (+SS). Results for the *unrestricted* model are also included for reference.

out *superspeculation* are summarized in Table 10-1, and show that *superspeculation* is a robust, promising approach that delivers significant performance increases across a broad spectrum of benchmarks and machine models.

Table 10-1: Harmonic Mean Speedup Summary

Machine Model	Dispatch Width	SPECInt '95			SPECFP '95		
		Disp lat 1	Disp lat 2	Disp lat 3	Disp lat 1	Disp lat 2	Disp lat 3
Unrestricted	64	1.58	2.04	2.66	1.19	1.40	1.61
Perfect	64	1.65	2.62	2.76	1.11	1.27	1.39
Restricted	16	1.40	1.80	2.20	1.17	1.39	1.59

Acknowledgments

This work was supported in part by ONR grant N00014-96-1-0928. We gratefully acknowledge the generosity of the Intel Corporation for donating numerous fast Pentium Pro-based workstations for our use. We also wish to thank the authors of the PSIM functional emulator for their generosity in making this tool publicly available.

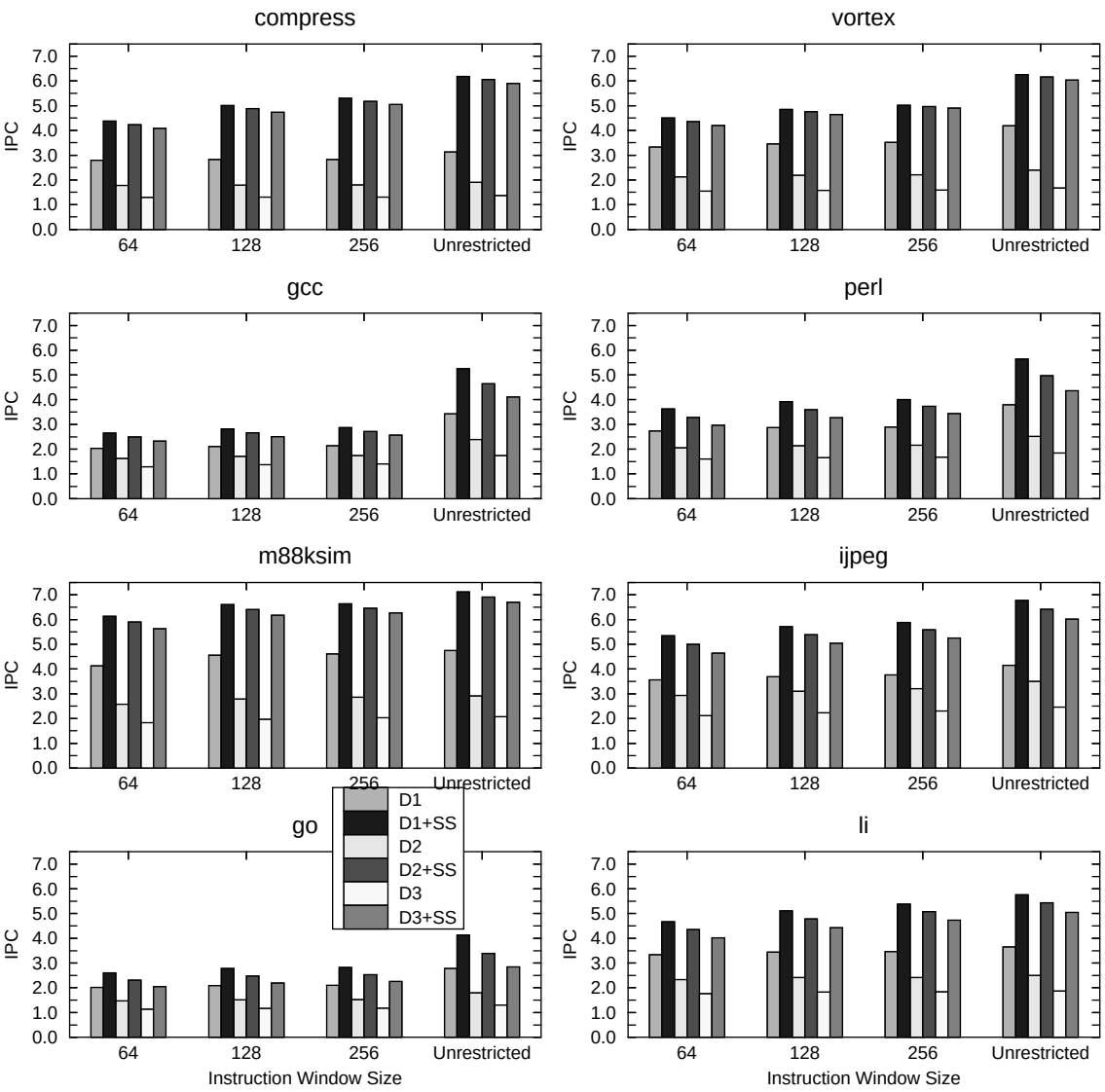

Figure 9-5 SPECInt95 Superspeculation with Restricted Machine Model. The sustained IPC for an dispatch width of 16 and various instruction window sizes is shown for single-cycle dispatch (D1), 2-cycle pipelined dispatch (D2), and 3-cycle pipelined dispatch (D3), all with and without *superspeculation* (+SS). Results for the *unrestricted* model are also included for reference.

paper. Second, we propose a *pipelined dispatch structure* that eases the implementation of wide-dispatch microarchitectures. Third, we propose *dependence prediction*, a speculative technique for alleviating the performance penalty of pipelined dispatch. Fourth, we propose *source operand value prediction*, which is a modified approach to *value prediction* that decouples instruction execution from dependence checking by predicting source operands rather than destination operands. Finally, we propose limited forms of *eager execution* and *control equivalence detection*, and show that these techniques, in conjunction with *value* and *dependence prediction*, can *superspeculate* beyond control flow, data-flow, and dependence detection bottlenecks to reach unprecedented levels of uniprocessor performance, and can in most cases handily exceed the performance achievable via perfect branch prediction or perfect caches, particularly with pipelined dispatch. Harmonic mean speedups for *superspeculation* over identical machine models with-

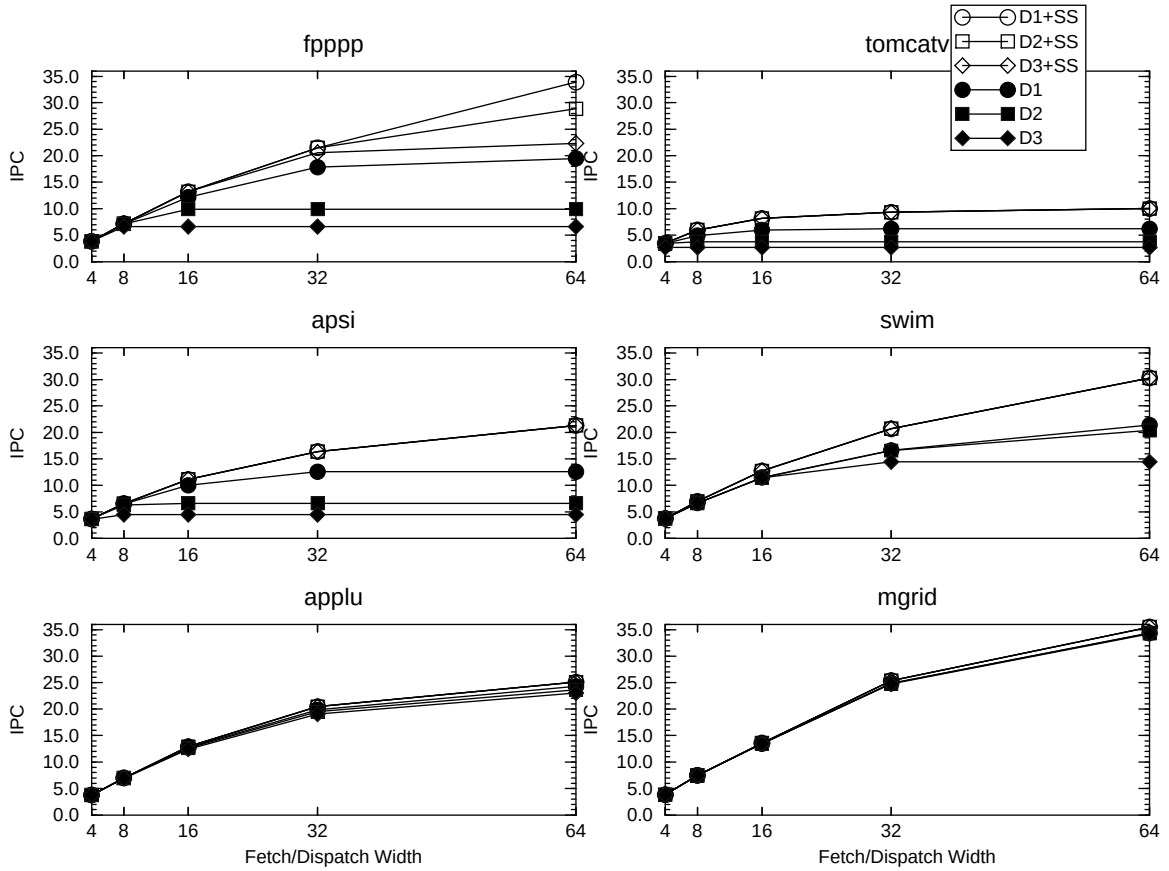

Figure 9-4 SPECFP95 Performance of Superspeculation with Perfect Branching. The sustained IPC for various dispatch widths and perfect branching is shown for single-cycle dispatch (D1), 2-cycle pipelined dispatch (D2), and 3-cycle pipelined dispatch (D3), all with and without *superspeculation* (+SS).

Furthermore, in all cases but one (*gcc* with single-cycle dispatch), *superspeculation* under the restricted model (which accounts for memory hierarchy latencies) handily outperforms the unrestricted model without *superspeculation* (which assumes a perfect memory system). This indicates that *superspeculation* is clearly a more promising route towards increased performance than building increasingly aggressive memory hierarchies or pursuing instruction or data prefetching techniques.

The results for the floating-point benchmarks, shown in Figure 9-6, are less interesting. While *superspeculation* achieves significant speedups on several benchmarks (*tomcatv*, *apsi*, *swim*, *applu*), particularly with multi-cycle dispatch, most floating-point benchmarks already perform quite well even without it. We attribute this to two factors: control-flow in these benchmarks is relatively predictable, hence our control-speculative techniques provide little benefit; and an abundance of program parallelism, which tends to already saturate available resources (for example, note the significant increase in IPC obtained by increasing window size for *swim*, *applu* and *mgrid*).

10.0 Conclusions

We make five major contributions in this paper. First of all, we introduce the *weak dependence model* for semantically correct execution, which lays a theoretical foundation for the speculative techniques proposed in this

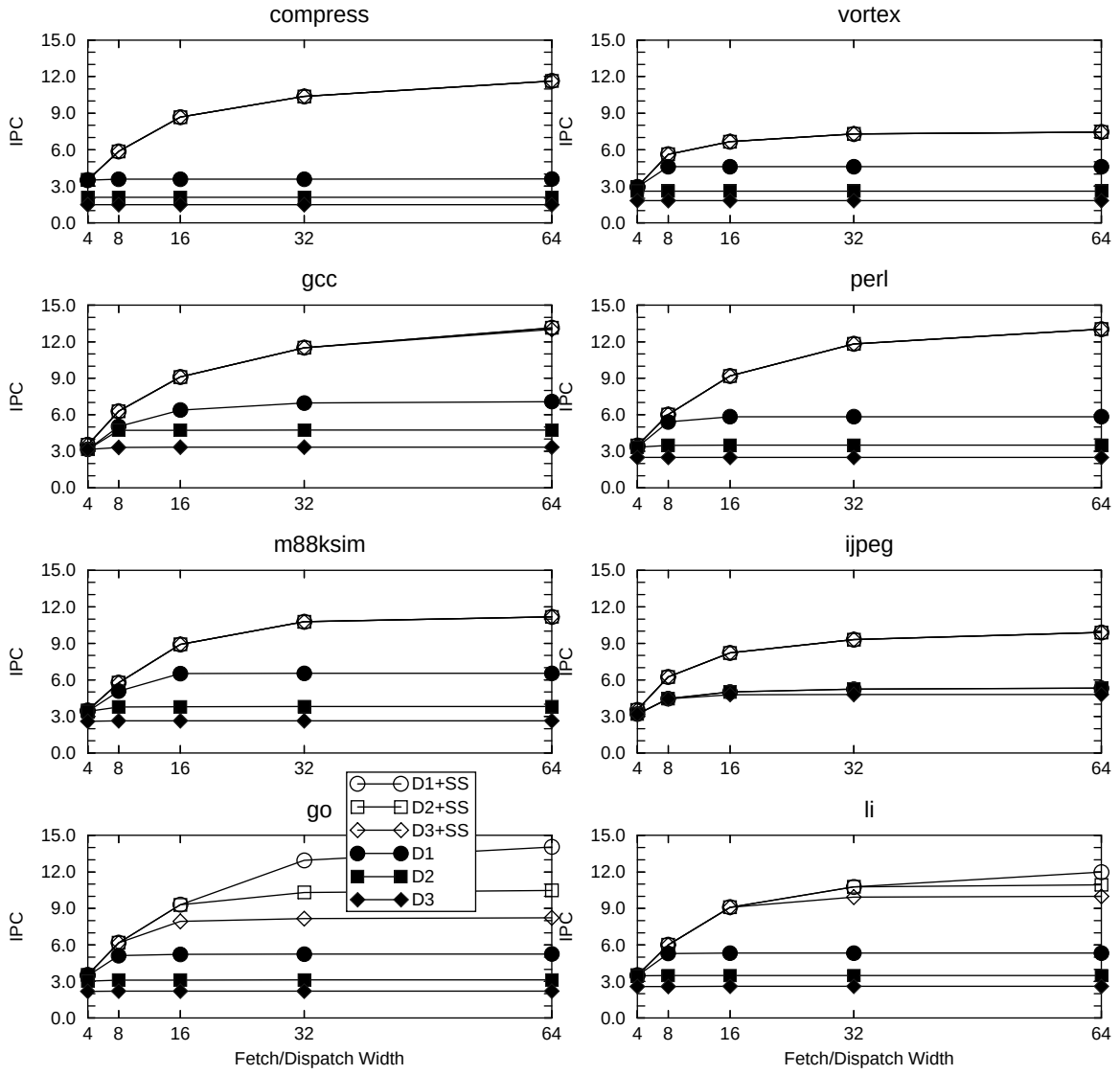

Figure 9-3 SPECInt95 Performance of Superspeculation with Perfect Branching. The sustained IPC for various dispatch widths and perfect branching is shown for single-cycle dispatch (D1), 2-cycle pipelined dispatch (D2), and 3-cycle pipelined dispatch (D3), all with and without *superspeculation* (+SS).

Figure 9-5 summarizes the performance of *superspeculation* under the *restricted* machine model for the integer benchmarks. There are two obvious trends. First of all, as with the unrestricted model, superspeculation is rather insensitive to multi-cycle dispatch; nearly all of the performance of single-cycle dispatch (D1+SS) can be obtained even with three-cycle dispatch (D3+SS). This is in stark contrast to the sharp reductions in performance that multi-cycle dispatch causes without *superspeculation* (D1 vs. D3). Second, *superspeculation* is able to achieve most of its available performance even with a relatively small instruction window. Having a larger instruction window (*128* or *256*) does improve performance, but only marginally. This is encouraging, since next-generation microprocessors already have instruction windows of size close to or greater than 64 (e.g. HP PA-8200 [Per96], DEC Alpha AXP 21264 [Kel96]).

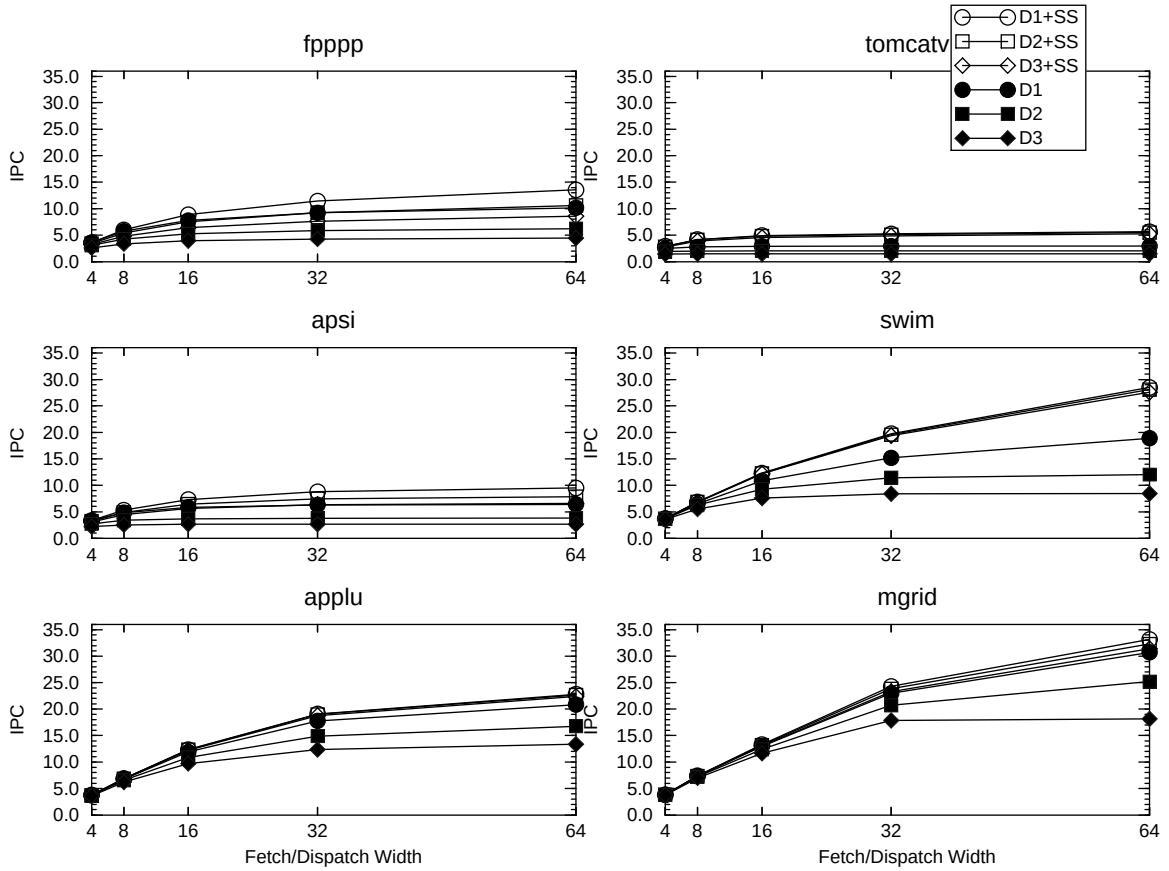

Figure 9-2 SPECFP95 Performance of Superspeculation. The sustained IPC for various dispatch widths is shown for single-cycle dispatch (D1), 2-cycle pipelined dispatch (D2), and 3-cycle pipelined dispatch (D3), all with and without *superspeculation* (+SS).

even when the effects of branch mispredictions are completely eliminated. Furthermore, by comparing Figure 9-1 with Figure 9-3, we see that *superspeculation* with an imperfect branch predictor consistently outperforms conventional issuing with a perfect branch predictor, even with single-cycle dispatch. With multi-cycle dispatch, *superspeculation* handily outperforms perfect branching, indicating that *superspeculation* is clearly a more promising route and provides more leverage towards increased performance than building ever-more-accurate branch predictors, particularly since branch predictors are already highly optimized. The only two exceptions to this trend are *applu* and *mgrid*, which are almost completely insensitive to dispatch latency with a perfect branch predictor. However, we suspect this behavior will break down quickly with any deviation from perfect branch prediction, since even with the very low branch misprediction rates shown for these benchmarks in Table 4-1, they show significant sensitivity to dispatch latency in Figure 9-2

9.3 Restricted Model

Finally, we want to examine the performance of *superspeculation* with a more realistic machine model. We include three additional structural limitations into our model: latencies due to the memory hierarchy, overall instruction window size, and completion bandwidth. These parameters are summarizes in Table 4-3 for the *restricted* machine model.

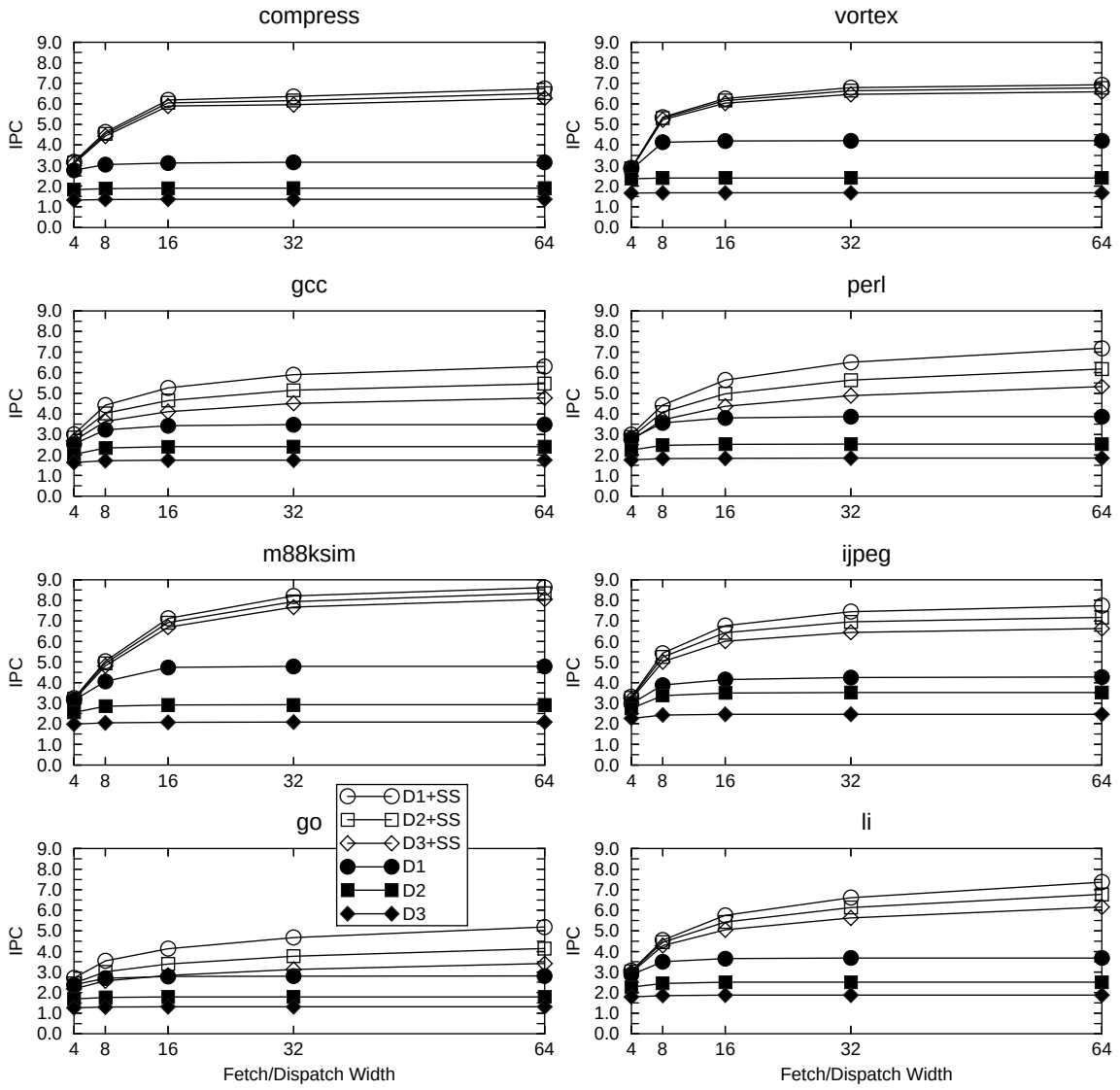

Figure 9-1 SPECInt95 Performance of Superspeculation. The sustained IPC for various dispatch widths is shown for single-cycle dispatch (D1), 2-cycle pipelined dispatch (D2), and 3-cycle pipelined dispatch (D3), all with and without *superspeculation* (+SS).

point benchmarks. Naturally, in the presence of a perfect branch predictor, limited *eager execution* will provide no benefit. However, we see that *dependence prediction* (+DP), source operand *value prediction* (+VP), and limited *control equivalence detection* (+CE) combine to provide significant performance increases, taking us to 10 sustained IPC and beyond for all but two of the integer benchmarks (*vortex* only reaches 7.5 IPC, while *go* reaches 14.0 IPC with single-cycle dispatch, but drops to 8.2 with three-cycle dispatch).

For the floating-point benchmarks, performance improves significantly for three of them with perfect branch prediction (*fpppp*, *tomcatv*, and *apsi*), while for the other three (*swim*, *applu*, and *mgrid*), perfect branch prediction primarily ameliorates the performance penalty of multi-cycle dispatch, and narrows the gap between conventional issuing and *superspeculation*.

In summary, we have shown that *superspeculation* continues to provide significant performance increases,

We see the beneficial effects of this interaction in Figure 8-3. Two excellent examples of this behavior are *compress* and *vortex*, which both derive virtually no benefit from +EE+CE without value prediction (see Figure 8-1 and Figure 8-2), but do derive significant benefit from it in conjunction with value prediction. For *compress*, single-cycle asymptotic IPC jumps from 5.2 with just value prediction to 6.2 when +EE+CE are added, while for *vortex*, single-cycle asymptotic IPC jumps from 6.2 with just value prediction to 6.9 when +EE+CE are added.

9.0 Putting It All Together: Superspeculation

The combination of these four techniques--*dependence prediction* (+DP), source operand *value prediction* (+VP), limited *eager execution* (+EE), and limited *control equivalence detection* (+CE)--leads to a microarchitectural paradigm we loosely term *superspeculation* (+SS). In this paradigm, we attempt to break through restrictions that were previously assumed to be hard limits by exploiting the *weak dependence model* to aggressively speculate beyond them. We do so by taking advantage of value locality in the data flow of a program (to do value prediction) as well as the control logic of the processor (to do dependence prediction). We also find the performance benefits of each of the techniques proposed to be mutually synergistic (i.e. they tend to be magnified in the presence of the others). We collect performance results for three different machine models: *superspeculation* with infinite execution resources and a realistic branch predictor (*unrestricted*), *superspeculation* with infinite execution resources and a perfect branch predictor (*perfect*), and *superspeculation* with a finite instruction window and memory hierarchy with a realistic branch predictor (*restricted*). The parameters for these machine models are summarized in Table 4-3.

9.1 Unrestricted Machine Model

The *unrestricted* machine model is similar to the one used in the previous sections, but combines all four of the *superspeculative* techniques to maximize sustainable IPC. In Figure 9-1 and Figure 9-2 we show the effect of *superspeculation* for various dispatch widths and dispatch latencies for the integer and floating-point benchmarks. The best possible performance is obtained with single-cycle dispatch and *superspeculation*, where the asymptotic IPC for the integer benchmarks is 8.6 in the best case (*m88ksim*) and 5.2 in the worst case (*go*). However, in all cases, the asymptotic IPC for even three-cycle dispatch with *superspeculation* exceeds that of single-cycle dispatch without *superspeculation*, and is only slightly worse than single-cycle dispatch with *superspeculation* in many cases (*compress*, *m88ksim*, *li*, *ijpeg*, *vortex*).

For the floating-point benchmarks, asymptotic IPC levels off around 14 for *fpppp*, around 10 for *apsi*, and around 6 for *tomcatv*. For the other three benchmarks (*applu*, *swim*, and *mgrid*), performance does not appear to be leveling off, even at 64-wide dispatch. As with the integer benchmarks, *superspeculation* for even three-cycle dispatch always outperforms the baseline model's conventional issuing policy. However, the margins are less significant in many cases.

In summary, we have shown *superspeculation* to be a microarchitectural approach that not only provides significant performance increases in the presence of imperfect branch prediction (frequently 200%-300% faster than the baseline), but one that is also relatively insensitive to deeper pipelining, hence greatly facilitating the implementation of very-wide-dispatch and deeply-pipelined superscalars.

9.2 Perfect Machine Model

In anticipation of future improvements in branch predictor technology, and to further explore the performance potential of *superspeculation*, we simulate *superspeculation* in conjunction with a perfect fetch and branch predictor. Results for the *perfect* machine model are shown in Figure 9-3 and Figure 9-4 for the integer and floating-

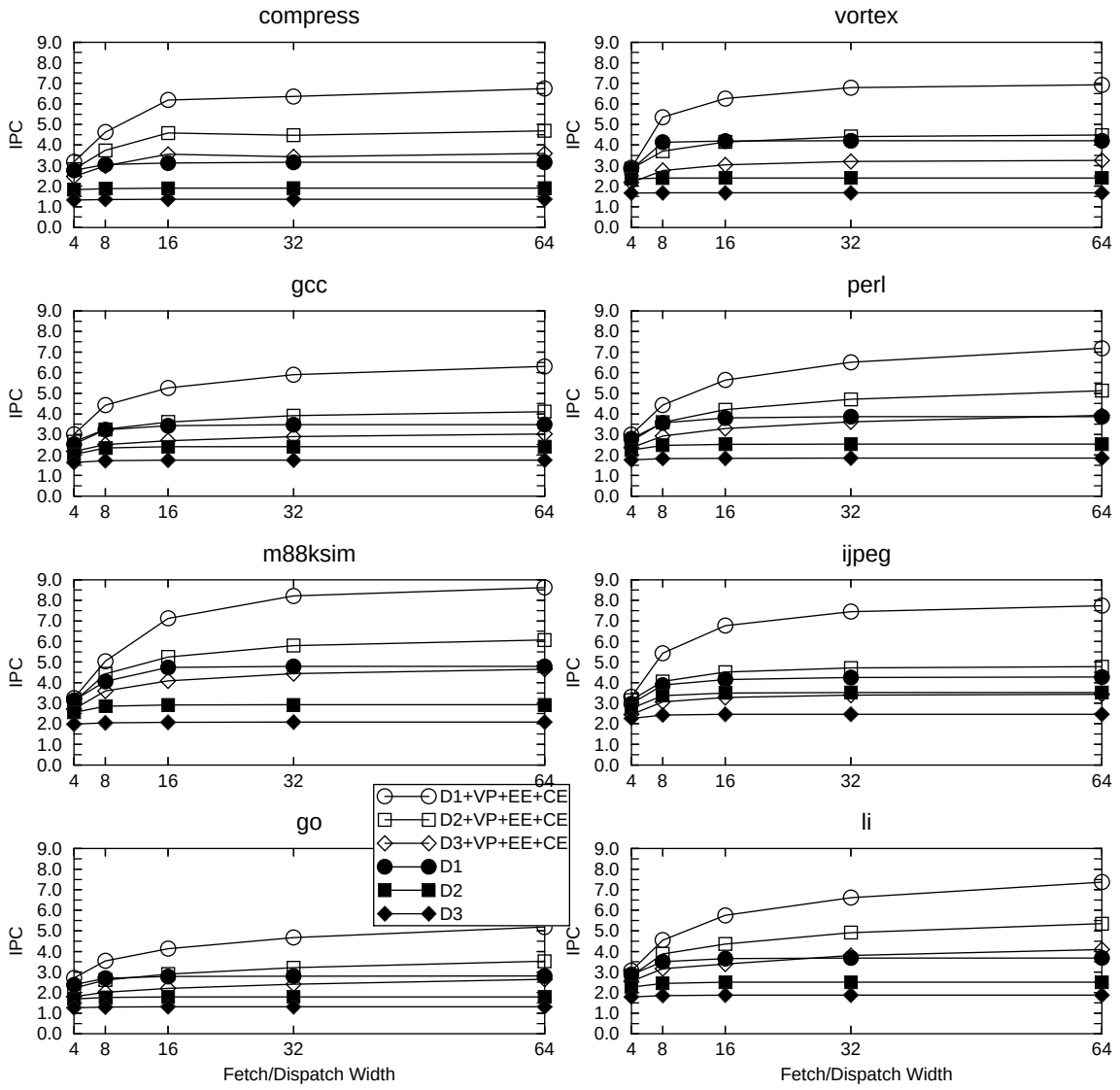

Figure 8-3 Cumulative Effect of +EE+CE and Value Prediction. The sustained IPC for various dispatch widths is shown for single-cycle dispatch (D1), 2-cycle pipelined dispatch (D2), and 3-cycle pipelined dispatch (D3), all with and without both +EE+CE and value prediction (+VP+EE+CE).

without +EE+CE (*m88ksim*, *gcc*, *ijpeg*).

Our limited scheme for *control equivalence detection* interacts in an interesting way with source operand *value prediction*. Since *control equivalence detection* introduces the potential of multiple definitions reaching a single dependent use at control flow joins (i.e. the targets of the short forward branches), the hardware must speculatively choose (through branch prediction) one or the other reaching definition to satisfy the dependent use. However, this choice cannot be verified as correct until the conditional branch is resolved. In effect, this increases the result latency of the defining instruction to match that of the conditional branch. Fortunately, source operand *value prediction*, in conjunction with the *weak dependence model*, is able to remedy this situation. In cases where the source operand value has been predicted, the only dependent *use* that is delayed until after the conditional branch resolves is the value comparison that verifies the predicted value. Hence, subsequent dependent instructions are free to issue and execute.

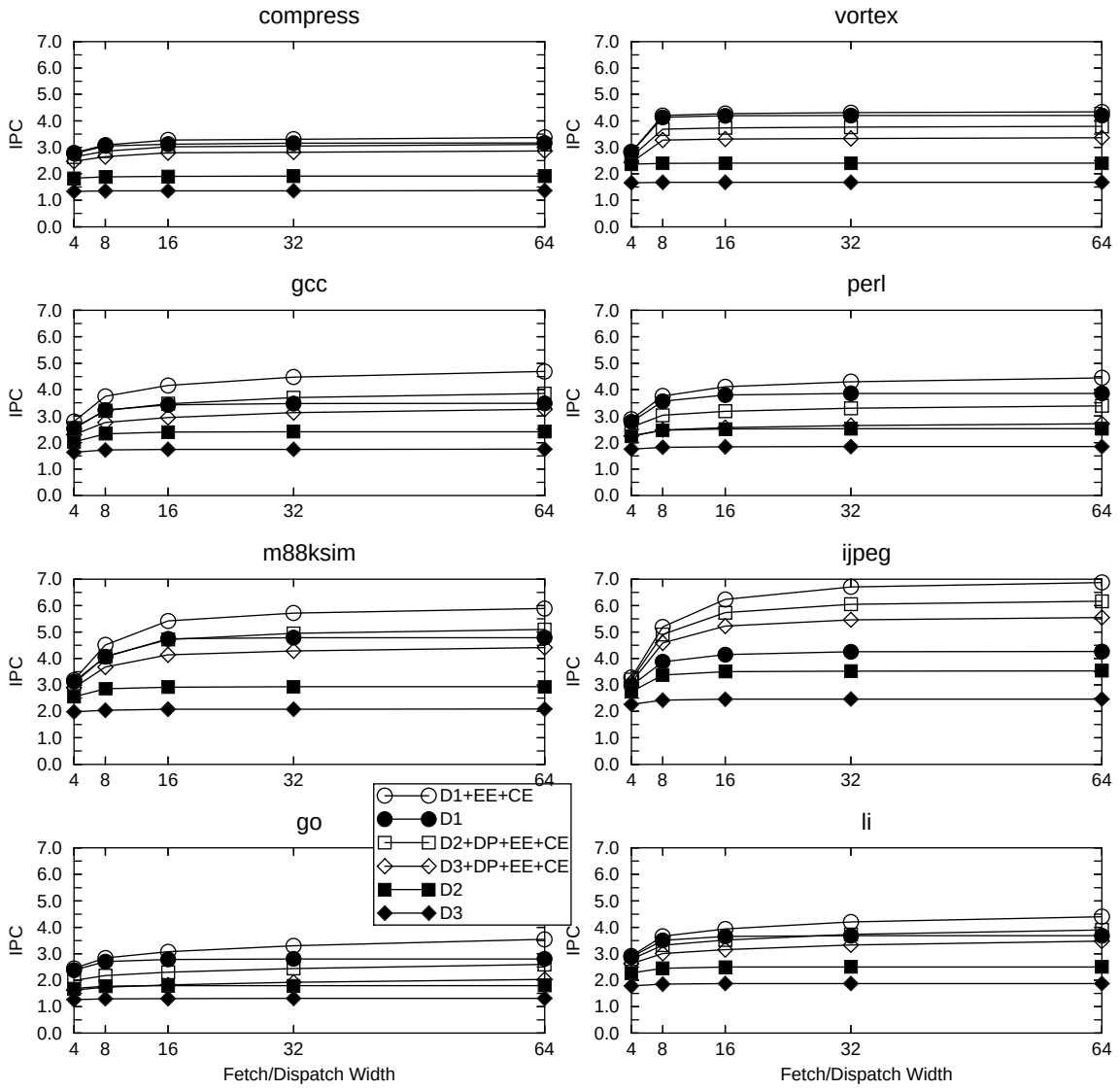

Figure 8-2 Cumulative Effect of +EE+CE and Dependence Prediction. The sustained IPC for various dispatch widths is shown for single-cycle dispatch (D1) with and without +EE+CE, 2-cycle pipelined dispatch (D2) with and without both +EE+CE and dependence prediction (+DP+EE+CE), and 3-cycle pipelined dispatch (D3) with and without both +EE+CE and dependence prediction (+DP+EE+CE).

cases (*compress* and *vortex*), limited *eager execution* and *control equivalence detection* provide virtually no performance gain at all. Also, in all cases, increasing the dispatch latency tends to reduce the magnitude of the speedup obtained with limited *eager execution* and *control equivalence detection*. However, an encouraging trend shows up with *m88ksim*, *gcc*, and *ijpeg*; namely, that noticeable improvements in IPC are available beyond dispatch widths of 16 and even 32.

Further good news, however, shows up when we combine +EE+CE with *dependence prediction*. These results are shown in Figure 8-2, where we see that much of the performance penalty associated with pipelined dispatch can be eliminated even in conjunction with +EE+CE, allowing two- and three-cycle dispatch to come much closer to the performance of single-cycle dispatch with +EE+CE, and occasionally even beating single-cycle dispatch

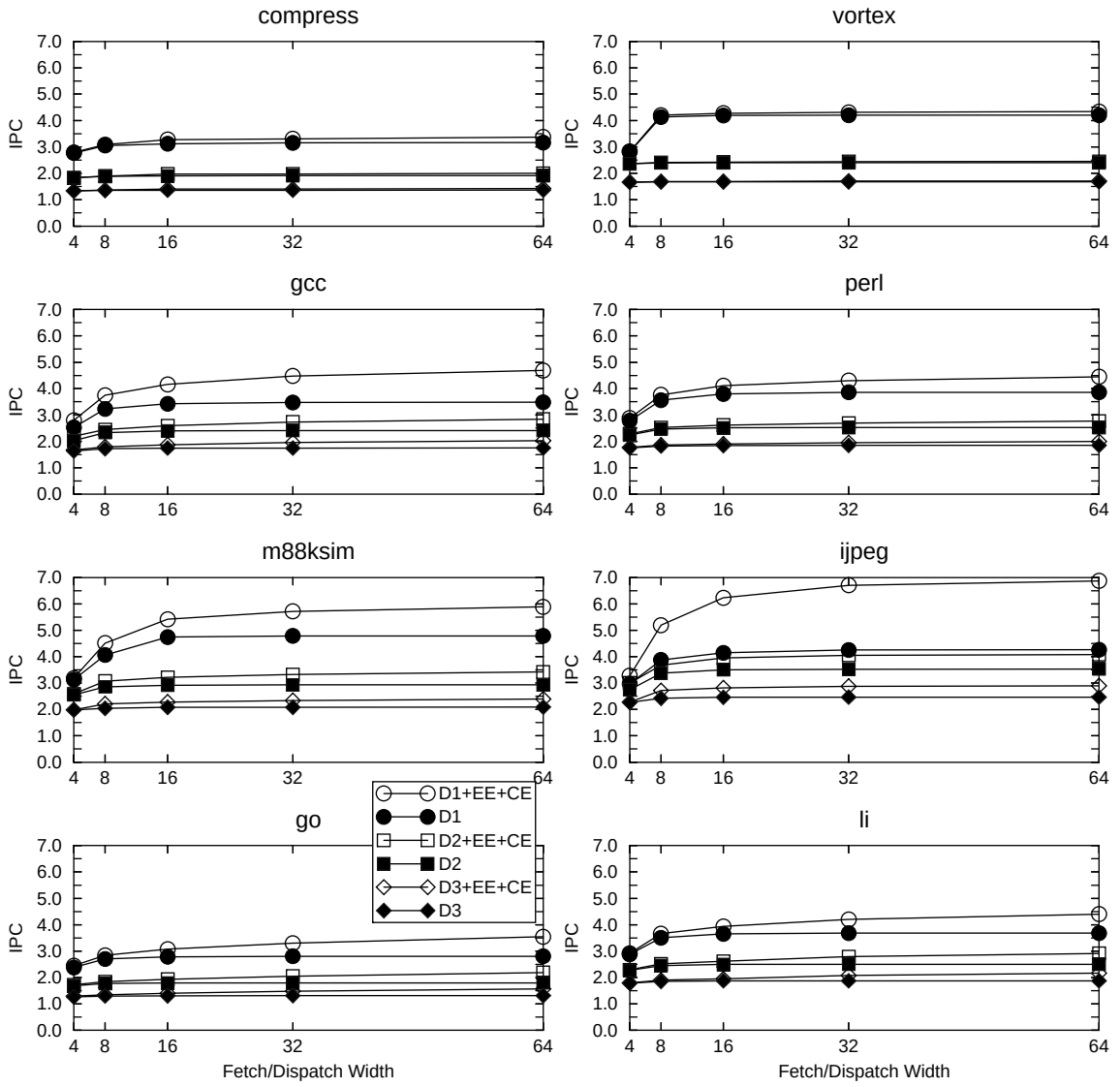

Figure 8-1 Effect of Limited Eager Execution (+EE) and Control Equivalence Detection (+CE). The sustained IPC for various dispatch widths is shown for single-cycle dispatch (D1), 2-cycle pipelined dispatch (D2), and 3-cycle pipelined dispatch (D3), all with and without +EE+CE.

is predicted taken, instructions between the branch and its target are prohibited from writing dependence information into the DRB or MF structures (just as in the +EE scheme described in Section 8.1). Once dispatch reaches the branch target, this prohibition is no longer in effect, and instructions are dispatched as usual. If the short forward branch that bypassed these instructions is later determined to have been mispredicted, the MF is restored, as always, from the branch recovery stack, and dispatch proceeds as usual for all subsequent instructions. However, since they may have already executed, their results will be immediately available in the value silo, resulting in a significantly reduced branch misprediction penalty.

In Figure 8-1 we show the effect of limited *eager execution* and *control equivalence detection* on IPC for various dispatch widths and dispatch latencies for the eight integer benchmarks. The best performance, obviously, is obtained with single-cycle dispatch, where asymptotic IPC jumps from 4.2 to 6.8 in the best case (*ijpeg*). In the worst

These results are shown in Figure 7-3, where we see that virtually all of the performance penalty associated with pipelined dispatch has been eliminated, allowing even three-cycle dispatch to nearly match the performance of single-cycle dispatch. Even the worst case benchmark (*go*) only degrades by 17% from single-cycle to two-cycle dispatch, while the best case (*vortex*) degrades by only 2% for two-cycle dispatch and 4% for three-cycle dispatch. Furthermore, three-cycle dispatch with *value* and *dependence prediction* can usually at least match, and frequently clearly outperform (*compress*, *vortex*, *m88ksim*, *li*), single-cycle dispatch without *value* or *dependence prediction*.

8.0 Limited Eager Execution and Control Equivalence Detection

To further enhance the performance of wide-dispatch superscalars, we propose two additional techniques that exploit the *weak dependence model* by relaxing the issue constraints that control dependences impose on conventional processors. These techniques are *limited eager execution* (+EE), which seeks to alleviate the branch misprediction penalty, and *limited control equivalence detection* (+CE), which exploits control-flow equivalence between the sites and targets of short conditional branches to expose additional instruction-level parallelism.

8.1 Limited Eager Execution

Eager execution has been proposed as one remedy for the branch misprediction penalty. By executing all paths, and then selecting the correct outcome once conditional branches resolve, the latency penalty of misprediction can be eliminated. However, this occurs at tremendous hardware costs, since the number of paths to be executed can grow exponentially. Recent work on Disjoint Eager Execution (DEE) attempts to remedy this problem by pruning the number of paths based on cumulative branch probabilities [US95]. In this paper, we propose a very limited form of eager execution that exploits instruction fetch and dispatch bandwidth that has already been consumed by instructions on the non-predicted path. In limited *eager execution* (hereafter referred to as +EE), we simply allow instructions that follow a predicted-taken branch within a fetch group to dispatch speculatively up to the next taken or predicted-taken branch or the end of the fetch group, whichever occurs first. By doing so, we may avoid the cost of eventually re-executing these instructions if the branch ends up being not-taken (i.e. mispredicted as taken). Naturally, we also impact usage of other structural resources, but since we are measuring the potential performance of this approach, we ignore that effect for now. The impact on our dispatch logic is minimal, and we avoid the “hardware explosion” usually associated with eager execution. Instructions dispatched on a non-predicted path are simply prohibited from writing their dependence information into the DRB or MF structures described in Section 5. If the branch that bypassed these instructions is later determined to have been mispredicted, the MF is restored, as always, from the branch recovery stack, and dispatch proceeds as usual for these instructions. However, since they may have already executed, their results will be immediately available in the value silo, often resulting in a reduced branch misprediction penalty.

8.2 Limited Control Equivalence Detection

Exploiting control equivalence has been shown to be very important to sustaining high IPC in a number of theoretical studies (e.g. [LW92]). However, dynamically detecting and exploiting control equivalence in the general case can be very difficult and lead to complex hardware (one possible implementation is discussed in [US95]). Rather than attacking the general case, we propose a simple, limited approach called limited *control equivalence detection* (+CE) that captures dynamic control equivalence between the site and target of short forward branches within a fetch group (note that this is not full control equivalence in the strict static sense, but only within our limited dynamic instruction window). This is a simple extension to the limited *eager execution* scheme described in Section 8.1. When the dispatch hardware detects a short forward branch, it continues dispatching subsequent instructions. If the branch

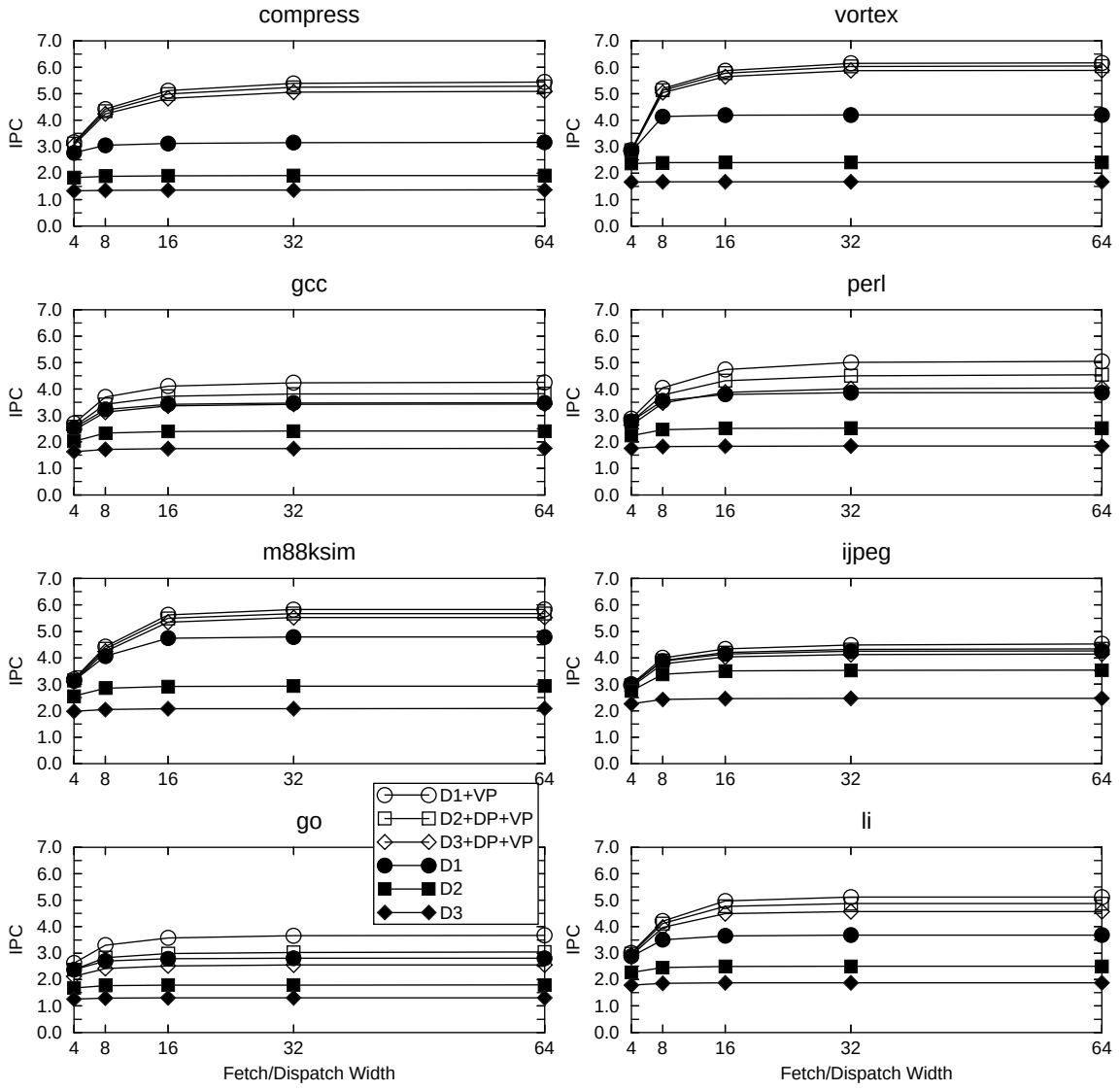

Figure 7-3 Cumulative Effect of Value and Dependence Prediction. The sustained IPC for various dispatch widths is shown for single-cycle dispatch (D1) with and without value prediction (+VP), 2-cycle pipelined dispatch (D2) with and without both dependence and value prediction (+DP+VP), and 3-cycle pipelined dispatch (D3) with and without both dependence and value prediction (+DP+VP).

In Figure 7-2 we show the effect of *value prediction* on IPC for various dispatch widths and dispatch latencies for the eight integer benchmarks. The best performance, obviously, is obtained with *value prediction* and single-cycle dispatch, which levels off around a dispatch width of sixteen to 3.7 IPC in the worst case (*go*), and 6.2 IPC in the best case (*vortex*). Lengthening dispatch to two and three cycles degrades *go* to asymptotic IPC of 2.6 and 2.0, respectively, while reducing *m88ksim* (which is now the best performer) to 4.5 and 3.4 IPC. *Value prediction* improves performance significantly in all cases over the baseline cases without value prediction, and the knees of the curves shifts further to the right. Also, for *ijpeg*, value prediction is able to overcome nearly all of the performance penalty of pipelined dispatch.

The real good news, however, shows up when we combine *value prediction* with *dependence prediction*.

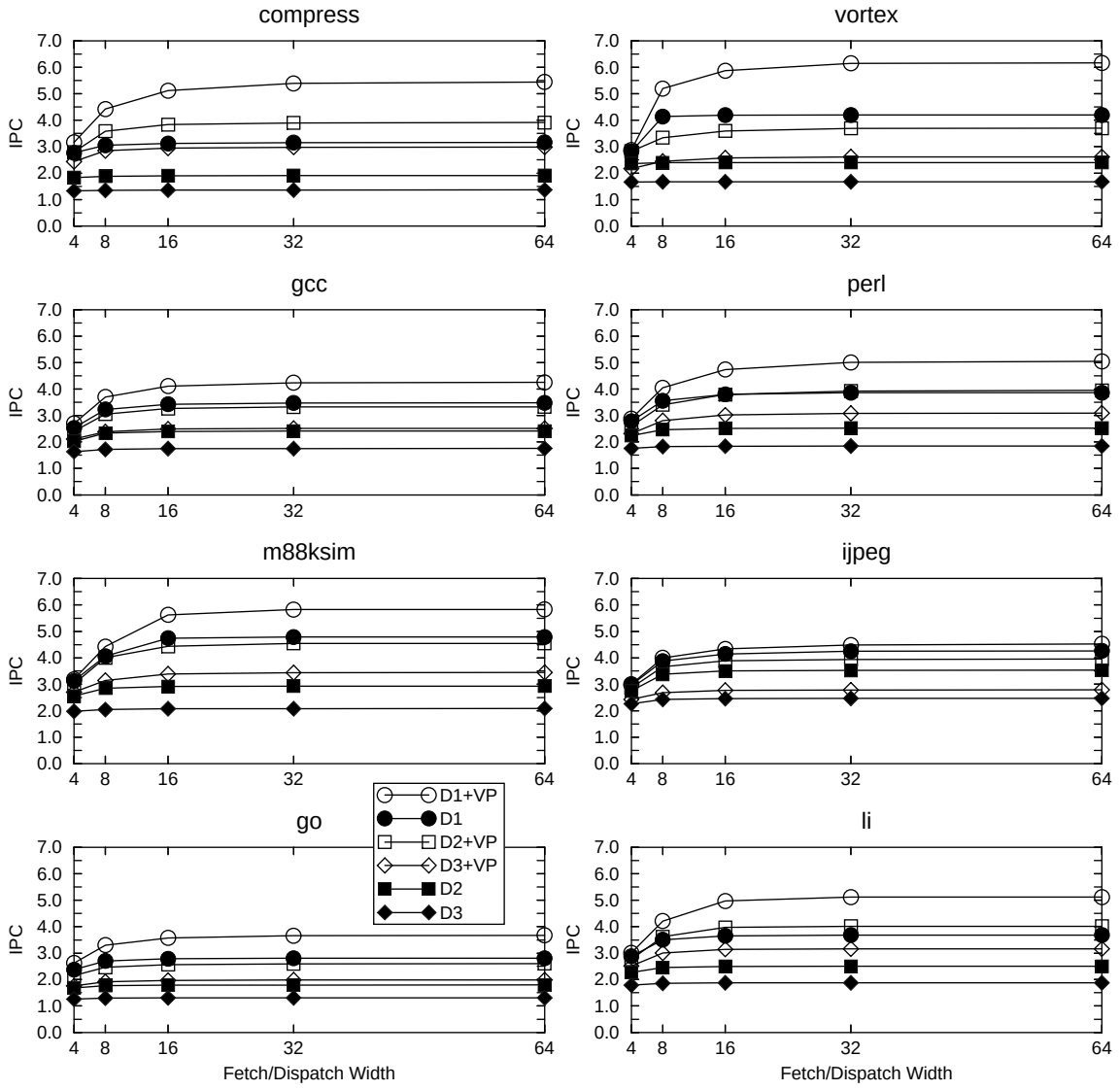

Figure 7-2 Effect of Value Prediction. The sustained IPC for various dispatch widths is shown for single-cycle dispatch (D1) with and without value prediction (+VP), 2-cycle pipelined dispatch (D2) with and without value prediction (+VP), and 3-cycle pipelined dispatch (D3) with and without value prediction (+VP).

the number of unpredictable source operands that were identified as such by the CT and the total number of unpredictable source operands. The *dependence prediction hit rate* (column five) is included to show the interaction between *value prediction* and *dependence prediction*. When both types of prediction are used, operands that are deemed unpredictable by the CT are relegated to *dependence prediction*. We see that the *dependence prediction* hit rates are better across the board than the ones shown in Table 6-1, indicating that the techniques are mutually synergistic. We also note that the *value locality* numbers are similar to those reported earlier [LS96], while the CT hit rates are somewhat better. The former is not surprising, since source operands should be no more or less predictable than destination operands, while we attribute the latter improvement to the *gshare*-like CT lookup index used in these experiments.

value prediction also uses a direct-mapped classification table (CT) similar to the one proposed in [LS96] for classifying the predictability of source operands and deciding whether or not the operands should be predicted. In our experiments, the CT is direct-mapped, has 8K entries with a 2-bit saturating counter at each entry, and is indexed by hashing together the instruction address bits, the *gshare* branch predictor's branch history register (BHR), and the relative position of the operand being looked up. As with *dependence prediction*, some initial experiments which are not reported in detail in this paper indicate that the classification hit rates improve noticeably when we use a *gshare*-like lookup index. Hence, we use the gshare-like lookup index in all experiments reported herein.

Table 7-1: Source Operand Value Prediction Results

Bench mark	Value Locality	CT Predictable Hit Rate	CT Unpredictable Hit Rate	Dependence Prediction Hit Rate
go	45.3%	77.0%	83.7%	42.1%
m88ksim	56.1%	92.8%	89.6%	89.6%
gcc	40.9%	78.0%	89.6%	63.0%
compress	42.4%	97.5%	98.8%	94.6%
li	33.7%	76.9%	92.9%	75.4%
ijpeg	35.2%	91.6%	95.9%	81.2%
perl	44.5%	76.4%	84.3%	54.4%
vortex	32.9%	83.3%	93.9%	82.5%
applu	25.5%	93.9%	95.9%	60.2%
apsi	35.0%	92.7%	95.2%	66.2%
fpppp	26.8%	75.7%	94.7%	33.0%
mgrid	12.1%	99.5%	99.9%	70.3%
swim	19.8%	94.8%	99.6%	89.3%
tomcatv	58.0%	93.8%	91.7%	84.8%

When all of the input operands of an instruction are classified as predictable, the instruction is permitted to dispatch early, after the first dispatch cycle (instructions with unpredictable source operands may still end up executing sooner than without value prediction, in cases where an operand that is predicted is on a critical path). Once dispatch finishes and exact dependence information becomes available, the instruction waits for its *verified* operands to become available in the value silo (operands in the value silo become *verified* when the instructions that generate them have validated all of their input operands) and then compares them against its predicted operands. If they match, the result operands of the instruction are marked *verified*, and the instruction is allowed to complete in program order. If they don't match, the instruction re-executes with the correct operands. Just as in [LS96], this results in a one-cycle misprediction penalty, since the instruction in question as well as all of its dependents do not execute with their correct inputs until one cycle later than if there had been no prediction. However, due to the lack of a global misprediction broadcast mechanism like the one used in [LS96], the one-cycle penalty can occur at every level in a dependence chain, rather than only at the top of the dependence chain.

Table 7-1 summarizes the value locality, classification hit rates, and dependence prediction hit rates for each of our benchmarks. Value locality (column two), as defined in [LWS96], is the ratio of the dynamic count of source operands that are predictable with the VPT mechanism and the dynamic count of all source operands. The *predictable hit rate* (column three) is the ratio of the number of predictable source operands that were identified as such by the CT and the total number of predictable source operands. Similarly, the *unpredictable hit rate* (column four) is the ratio of

latencies for the eight integer benchmarks. Without dependence prediction, the best performance, obviously, is obtained with single-cycle dispatch, which levels off at a dispatch width of sixteen to 2.8 IPC in the worst case (*go*), and 4.8 IPC in the best case (*m88ksim*). Lengthening dispatch to two and three cycles degrades *go* to asymptotic IPC of 1.7 and 1.3, respectively, while reducing *ijpeg* (which is now the best performer) to 3.7 and 2.5 IPC. Furthermore, the knee of the dispatch-width curves has now shifted left to between four and eight, rather than sixteen, hence eroding incentive for building processors with dispatch widths exceeding four, which is the width that many current-generation microprocessors implement. Fortunately, dependence prediction is able to alleviate these depressing trends by reducing the average dispatch latency. For both two- and three-cycle dispatch, dependence prediction significantly elevates the IPC curves and brings them much closer to the single-cycle case. Furthermore, the knee of these dispatch-width curves shifts back towards sixteen, restoring incentive for building wider superscalar processors. Three benchmarks--*compress*, *li*, and *ijpeg*--behave particularly well, eliminating nearly all of the performance penalty induced by two- and three-cycle dispatch.

7.0 Source Operand Value Prediction and Recovery

A complementary approach for reducing the adverse performance impact of pipelined dispatch involves a variation on previous work on value prediction [LS96]. In earlier work, the destination operands (i.e. results) of instructions were predicted via table-lookup at fetch/dispatch, and then forwarded directly to dependent instructions. The shortcoming of this approach is that dependence relationships must be detected before values can be forwarded to dependent instructions. To overcome this problem, we propose predicting the values of **source** operands, rather than **destination** operands, hence decoupling value-speculative instruction dispatch entirely from dependence detection. As in the earlier work, we predict only floating-point and general-purpose register operands, and not condition registers or special-purpose registers.

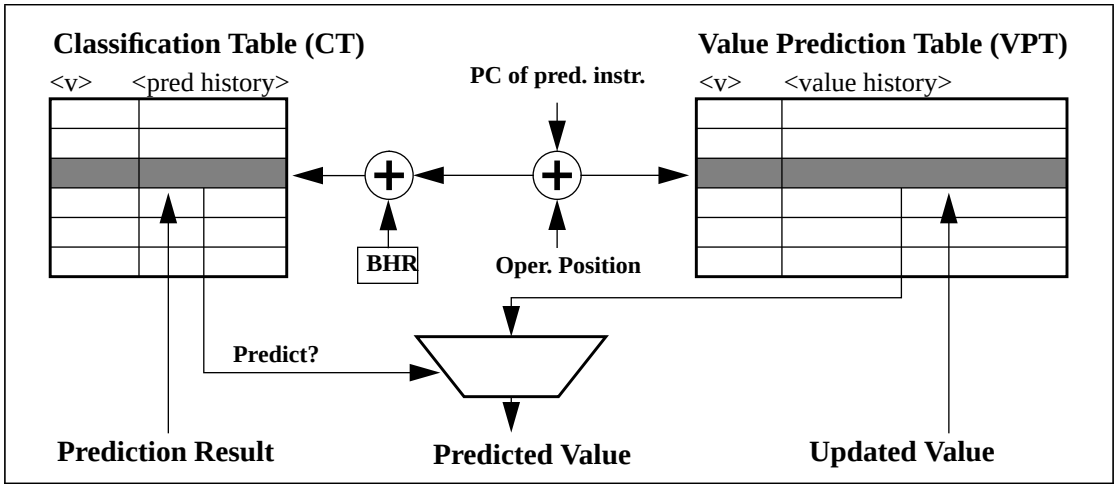

Figure 7-1 Source Operand Value Prediction Mechanism. The source operand position and PC are hashed together to index the VPT, while the operand position, PC, and BHR are hashed together to index the CT. The prediction and value histories are updated at completion.

Source operand *value prediction* (+VP) is illustrated in Figure 7-1. As in [LS96], we use a value prediction table (VPT) to keep track of past operand values, and exploit the value locality [LWS96] of operands to predict future values. In our experiments, the VPT is direct-mapped, 32KB in size, and is indexed by hashing together the instruction address bits and the relative position of the operand (i.e. first, second, or third) being looked up. Source operand

correct predictions are shown in Table 6-1. We find that for most benchmarks, the DPT achieves a respectable hit rate. Some initial experiments, which are not reported in detail in this paper, indicated that using a *gshare*-like lookup index improves the DPT hit rate considerably over using just the instruction address. Hence, we use a *gshare*-like lookup index for all the experiments reported herein. For three benchmarks--*go*, *perl*, and *fpppp*--the dependence prediction hit rates were rather low. This behavior can be attributed to the unpredictable branch behavior of these three benchmarks, since unpredictable branches can lead to unpredictable dependence distances when there are multiple definitions reaching a use. As seen in Figure 2-1 and Table 4-1, both *go* and *perl* have high BTB misprediction rates, while *go* and *fpppp* have high BHT misprediction rates.

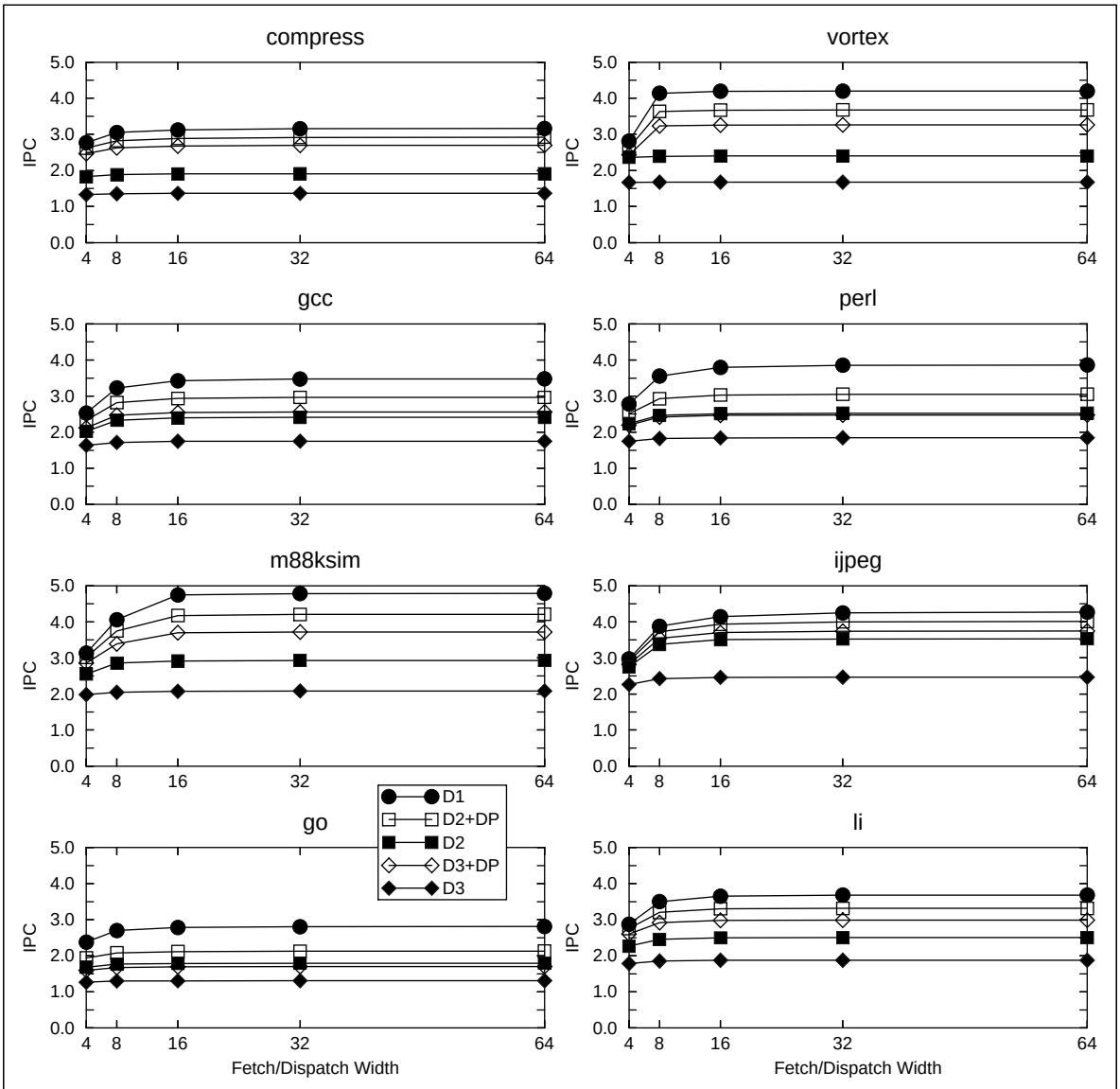

Figure 6-2 Effect of Dependence Prediction. The sustained IPC for various dispatch widths is shown for single-cycle dispatch (D1), 2-cycle pipelined dispatch (D2) with and without dependence prediction (+DP), and 3-cycle pipelined dispatch (D3) with and without dependence prediction (+DP).

In Figure 6-2 we show the effect of dependence prediction on IPC for various dispatch widths and dispatch

effects, we propose a mechanism called *dependence prediction* (+DP) that can frequently short-circuit multi-cycle dispatch by predicting the dependence relationships between instructions in flight and speculatively allowing instructions that are *predicted to be data ready* to execute in parallel with exact dependence checking.

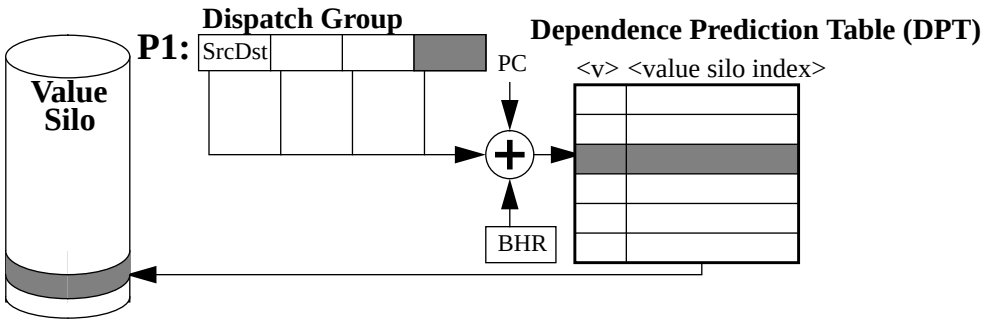

Figure 6-1 Dependence Prediction Mechanism. During stage P1, the source operand position, PC, and branch history register (BHR) are hashed together to index the DPT, which predicts the value silo entry that contains the source operand. During P2, the prediction is verified.

As shown in Figure 6-1, dependence prediction is implemented with a dependence prediction table (DPT) with 8K entries, which is direct-mapped and indexed by hashing together the instruction address bits, the *gshare* branch predictor's branch history register (BHR), and the relative position of the operand (i.e. first, second, or third) being looked up. Each DPT entry contains a numeric value which reflects the relative index of that input operand's source in the value silo. This relative index is used to check the value silo to see if the operand is already available. If all of the instruction's predicted input operands are available, the instruction is permitted to dispatch early, after the first dispatch cycle. In the second (or third, in the three-cycle dispatch pipeline) dispatch cycle, exact dependence information becomes available, and the earlier prediction is verified against the actual information. In case of a mismatch, the DPT entry is replaced with the correct relative position, and the early dispatch is cancelled.

Table 6-1: Dependence Prediction Results

Bench mark	Operands Predicted	Per Instruction	Correct Predictions
go	89.6M	1.126	38.0%
m88ksim	113.4M	1.060	77.4%
gcc	174.2M	0.958	59.7%
compress	40.6M	1.023	87.3%
li	49.8M	0.878	72.4%
ijpeg	92.3M	1.001	71.9%
perl	47.3M	0.944	48.2%
vortex	120.7M	0.788	71.3%
applu	53.1M	1.380	55.5%
apsi	215.8M	1.352	58.9%
fpppp	71.9M	1.435	32.1%
mgrid	148.5M	1.338	69.9%
swim	49.7M	1.282	87.3%
tomcatv	40.8M	0.864	77.8%

The total number of operands predicted, average number of predictions per instruction, and percentage of

mentation, it can be partitioned by register type and even register number.

As shown in Figure 5-1, during the first pipeline stage P1 of pipelined dispatch, all instructions in a fetch group allocate *value silo* slots for their destination operands, and then write the <register number, *value silo* slot> mapping tuples into their *dependence-resolution buffer* (DRB) entries. At the same time, the *value silo* slot numbers are written into the *mapping file* (MF), a table indexed by the register number. If a dispatch group contains more than one write to the same architected register, arbitration logic selects the last write before a taken or predicted-taken branch. During the second pipeline stage P2, all instructions in a fetch group search ahead in the DRB for a register number matching each of their input registers (the DRB is multi-ported and content-addressable). If multiple matching entries are found, the closest one (i.e. the most recent definition) is selected. If no matching entry is found, the *shadow mapping file* (MF') entry for the register is used instead. MF' summarizes the register-to-value silo mappings for all previous fetch groups, and is a one-cycle-delayed copy of the mapping file MF. If no register-to-value-silo mapping exists, the appropriate MF' entry will instead point to the architected register file. At the end of P2, all the instructions in the fetch group know where in the value silo they can find their input operands, and can check the scoreboarded valid bits to see if they are available.

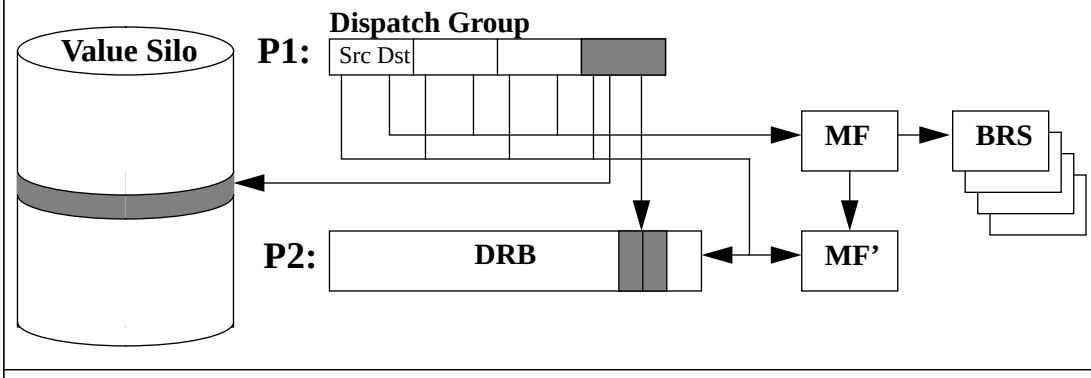

Figure 5-1 Pipelined Dispatch Structure. During stage P1, all instructions in a fetch group write destination register rename mappings into the DRB and the MF. During stage P2, the instructions search the DRB and MF' for source register rename mappings.

Whenever a predicted branch occurs within a dispatch group, a snapshot of the mapping file MF that includes all register writes through the branch is pushed onto a branch recovery stack (BRS). Any instruction following a taken or predicted-taken branch within a fetch group is discarded and prevented from writing into either the DRB or the MF. When a branch misprediction is resolved, any instructions that are newer than the branch are discarded along with their value silo slots, and fetching starts over from the actual destination of the mispredicted branch, while the MF snapshot corresponding to that branch is retrieved from the branch recovery stack.

As described here, instruction dispatch is pipelined into two stages. However, it is easy to envision even deeper pipelining of this process. Hence, we simulate the performance effects of and present results for one-, two-, and three-stage dispatch pipelines.

6.0 Dependence Prediction and Recovery

Figure 2-1 illustrates the detrimental performance effects of a pipelined dispatch structure. In short, a pipelined dispatch structure increases the number of cycles between a branch misprediction and the detection of that misprediction, hence aggravating the misprediction penalty and severely limiting performance. To alleviate these

arbitrarily, while the execute stage has unlimited width. The latency of the dispatch stage can also be varied from one to three cycles, while the latency of the execute stage is instruction-dependent and is summarized in Table 4-2. All functional units are fully pipelined, all architected registers are dynamically renamed, and instructions are allowed to execute out-of-order subject to the total instruction window size. However, all branches are executed in program order, and all loads are prevented from accessing memory until the addresses of all previous stores are known. If an alias to an earlier store exists, a forwarding mechanism exists that delays the load until the store's data becomes available, and then forwards that value directly to the load (in effect, the processor dynamically renames memory).

Our model uses a very aggressive *gshare* branch predictor [McF93] with a 256K entry branch history table (BHT) with 2 bits per entry that is indexed by the exclusive-or of a 18-bit branch history register and the branch instruction address. The branch target buffer (BTB) is direct-mapped, is not tagged, and has 1024 entries, while the return address stack (RAS) also has 1024 entries. Fetch and branch prediction both occur during the *fetch* stage of the pipeline, while instructions are fetched from a dual-banked instruction cache with line size equal to the specified fetch width (this configuration is described as *interleaved sequential* in [CMMP95]). Up to three conditional branches can be predicted per cycle, with a simple *gshare* extension to the scheme described in [RBS96]. Our model also supports a perfect branching mode, where all of these structures are assumed to be perfect, and hence introduce no delays into program execution.

We use three basic configurations of this machine model in our simulations. Key parameters for the three configurations are summarized in Table 4-3. The *unrestricted* configuration is used to generate the intermediate results presented in Section 6, Section 7, and Section 8. In Section 9, all three configurations--*unrestricted*, *perfect*, and *restricted*--are used to further explore the performance potential of *superspeculation*.

Table 4-3: Machine Model Configurations

Parameter	Unrestricted	Perfect	Restricted
Branch Predictor	Imperfect	Perfect	Imperfect
Fetch and Dispatch Width	{4,8,16,32,64}	{4,8,16,32,64}	16
Completion Width	Unrestricted	Unrestricted	16
Instruction Window	Infinite	Infinite	{64,128,256}
Instruction Cache	Perfect	Perfect	64K, 4-way, 64B lines
I-cache Miss Latency	N/A	N/A	10 CPU cycles
Data Cache	Perfect	Perfect	64K, 4-way, 32B lines
D-cache Miss Latency	N/A	N/A	10 CPU cycles

5.0 Pipelined Dispatch Structure

In this section, we describe a pipelined dispatch structure that facilitates the implementation of wide-dispatch microarchitectures by reducing the circuit complexity and cycle-time demands imposed by simultaneous cross-checking of data dependences within a large dispatch group. In this scheme, dependence checking is divided into two pipeline stages. During the first stage, all destination registers within a dispatch group are identified and renamed, and the rename mappings are written into the *dependence-resolution buffer* (DRB) and the *mapping file* (MF). During the second stage, all source registers are identified and their rename mappings are looked up in the DRB and the *shadow mapping file* (MF'). In our microarchitecture, all register writes are renamed to slots in a *value silo*. The *value silo* is used to scoreboard, hold, and forward the results of instructions until they are ready to complete and write back into the architected register file. Conceptually, the *value silo* is a monolithic structure, but in an actual hardware imple-

support libraries, but does not account for supervisor-state execution. All of the benchmarks are run to completion, albeit with reduced input sets and/or fewer iterations than in the SPEC95 reference runs.

Table 4-1: Benchmark set

Bench mark	Description	Input Set	Run Length	BHT Mispred	BTB Mispred	RAS Mispred
go	SPEC95 game	2stone9.in, 9x9, lev. 5	79.6M	12.0%	8.5%	0.0%
m88ksim	SPEC95 88K simulator	100 iter . of dhrystone	107.0M	2.7%	4.3%	0.0%
gcc	SPEC95 gcc compiler	genoutput.i	181.8M	5.1%	8.7%	3.9%
compress	SPEC95 data compression	small input (10K)	39.7M	5.9%	0.0%	0.0%
li	SPEC95 lisp emulator	six queens problem	56.8M	3.0%	5.3%	12.2%
ijpeg	SPEC95 jpeg encoder	tinyrose.ppm	92.1M	2.7%	0.6%	18.7%
perl	SPEC95 perl interpreter	train scrabbl.pl	50.1M	2.4%	11.1%	4.1%
vortex	SPEC95 database program	reduced version of train	153.1M	0.6%	1.6%	11.4%
applu	SPEC95 PDE Solver	5 iter, 12x12x12	38.5M	1.8%	0.2%	25.5%
apsi	SPEC95 Atmospheric model	10 steps, 128x1x32	159.0M	8.0%	0.8%	9.2%
fpppp	SPEC95 Quantum chemistry model	3 atoms (ref uses 30)	50.0M	10.9%	0.3%	0.8%
mgrid	SPEC95 Multigrid solver	10 iterations (test has 40)	111.0M	2.4%	0.5%	57.9%
swim	SPEC95 Shallow water model	10 iter 128x128	38.8M	1.9%	0.5%	0.4%
tomcatv	SPEC95 Mesh Generation	3 iter w/ array size 43	47.2M	1.3%	2.4%	47.4%
Total/HM			1,204.6M			

4.2 Simulation Environment

Our simulation environment is built around the PSIM PowerPC functional emulator that is distributed as part of the Free Software Foundation's GDB debugger [Cag96]. This functional emulator gives us complete access to the processor's internal register state at each instruction boundary. This is a requirement for us, since we use value-based dynamic prediction mechanisms to improve performance, and we need access to actual values, not just addresses, to accurately simulate these mechanisms. Full functional emulation also gives us the ability to accurately model the impact of mispredicted paths on internal processor structures. Currently, we only exploit this feature in a limited fashion (i.e. only certain mispredicted paths are fully emulated), but we plan to expand the functionality of our framework to include full simulation of all execution paths that the actual processor would exercise.

Table 4-2: Instruction Latencies

Instruction Class	Issue Latency	Result Latency
Integer Arithmetic and Logical	1	1
Integer Multiply	1	3
Integer Divide	1	10
Load/Store	1	1 for address, 2 for load data
FP Add/Subtract/Normalize/Negate	1	3
FP Multiply/Multiply-Add	1	3
FP Divide	1	11
Branch(pred/mispred)	1	0-1

4.3 Machine Model

The machine model used in our simulations has a canonical four-stage pipeline: *fetch*, *dispatch*, *execute*, and *complete*. The width of the fetch, dispatch, and completion stages and the total instruction window size can be varied

- Dependences can be temporarily violated during instruction execution, and
- Dependences need not be determined exactly or assumed pessimistically, but can instead be optimistically ignored or approximated.

Of course, the minimal requirement of semantic correctness must still be maintained before instructions are allowed to modify the processor's architected state. This can be achieved by verifying that the following conditions are met before an instruction is allowed to complete:

- The current instruction must be the immediate successor of the most recently completed instruction in the program's flow of control, and
- The current instruction must have received input values that match the current architected values, without regard for how those values were generated.

Note that these conditions eliminate the need for enforcing rigid source-to-sink relationships between data-dependent instructions and allow them to execute in parallel. It is this property that enabled the performance gains reported in [LWS96] and [LS96]. However, this property introduces an additional and previously unexploited degree of freedom for the microarchitect; namely, that source-to-sink data flow relationships between instructions needn't even be detected before these instructions can begin execution. In effect, it pushes dependence checking and dependence detection further down into the pipeline and eliminates it as an unnecessary front-end bottleneck. We use the term *superspeculation* to informally represent aggressive microarchitectural techniques that employ the *weak dependence model* to perform speculation beyond control and data dependences.

4.0 Experimental Framework

To evaluate the performance potential of the *weak dependence model* and *superspeculation*, we implement a flexible emulation-based simulation framework for the PowerPC instruction set. Currently, the simulation framework accurately models branch and fetch prediction, dispatch and completion width constraints, instruction window size, latency in the memory hierarchy, and all branch misprediction and data dependence delays for realistic instruction latencies. We intend to enhance the simulation framework to accurately model further structural resource limitations (e.g. rename buffers, functional units, forwarding paths, memory bandwidth, etc.). However, since our intent is to measure the performance potential of *superspeculation*, rather than the actual performance of a fixed implementation, we feel it is reasonable to refrain from modeling further structural constraints within the context of this paper.

4.1 Benchmarks

We selected the SPEC95 integer and floating-point benchmark suites for our study, since they are easily available, widely used, and well-understood. We are aware that these may not be representative of commercial workloads [MDO94]. Table 4-1 summarizes the benchmarks and input sets used, and also shows run length (dynamic instruction count) and BHT (*branch history table*), BTB (*branch target buffer*), and RAS (*return address stack*) branch misprediction rates. The BHT misprediction rate is the number of mispredicted conditional branches divided by the total number of conditional branches, the BTB misprediction rate is the number of taken branches with mispredicted targets divided by the total number of taken branches, and the RAS misprediction rate is the total number of subroutine returns with mispredicted targets divided by the total number of subroutine returns.

The integer benchmarks are compiled for the PowerPC instruction set with GCC version 2.7.2 at full optimization, while the floating-point benchmarks are compiled for the PowerPC instruction set with G77 version 0.5.18, also at full optimization (we only use six out of ten SPECFP95 benchmarks due to time and space constraints). The emulation environment captures the behavior of all user-state instructions, including those in the NetBSD runtime

weak dependence model.

3.0 The Strong and Weak Dependence Models

To satisfy the *sequential execution model* inherent in most instruction set architectures, any implementation of an architecture must maintain an in-order architectural state as if the instructions in an instruction stream were being executed sequentially without any overlap. This model is useful, and necessary in almost any computing system, because it enables a thread of control to be interrupted and later restarted at any instruction boundary (i.e. it enables precise exceptions). Specifically, all implementations must ensure that two conditions are met before an instruction is allowed to complete according to the sequential execution model:

- instructions must complete in the original program order (control-flow correctness), and
- instructions must produce semantically correct results (data-flow correctness).

That is, based on the in-order processor state that exists before the current instruction, the implementation must ensure that the current instruction is indeed the next one in the flow of control, and also that the current instruction has received the correct operands in the program's data flow. Current microprocessors expend significant amounts of hardware to ensure that both of these conditions are met throughout the lifetime of an instruction being executed. That is, there is an in-order front end that fetches and decodes instructions in order to determine control and data dependences, an out-of-order execution core that tolerates varying latencies but still rigorously or strongly enforces control and data dependences, and an in-order back end that completes instructions in program order once all dependences have been resolved and execution has finished.

The major shortcoming of this approach is that it is overly rigorous in enforcing both of the above conditions, and hence unnecessarily restricts available parallelism. When enforcing control flow correctness, modern processors discard all subsequent instructions when a fetch or branch misprediction is detected, and refetch from the new branch destination. All subsequent instructions already in flight are discarded due to the perceived difficulty of detecting which ones were useful and/or received correct data operands. However, doing so discards potentially large numbers of useful instructions, whenever control equivalence exists between basic blocks. Theoretical studies (e.g. [LW92]) have found the performance impact of this approach to be significant. Furthermore, when enforcing data flow correctness, modern processors detect and enforce rigid source-to-sink data-flow relationships before instruction execution is allowed to begin, which is more than the sequential execution model requires.

We classify such traditional and conservative processors as adhering to the *strong dependence model*. The *strong dependence model*, analogous to the *strong ordering* of memory references [DSB86], requires that all control and data dependences between instruction pairs--whether real or perceived--impose a serial ordering on the execution of the dependent instructions. In fact, at its most extreme, the strong dependence model requires that:

- All dependences must be adhered to throughout the execution of an instruction, and
- All dependences must be determined in an exact fashion (or, when in doubt, dependences are pessimistically assumed to exist).

Ad hoc attempts to relax the serialization requirements of the *strong dependence model* have appeared sporadically as modern computer architecture has evolved. For example, techniques like *speculative disambiguation* [HSS94, GCM+94] can temporarily violate data dependences before those dependences are known. Similarly, branch prediction enables speculative execution of instructions beyond conditional branches, hence violating control dependences temporarily, until the branches resolve. However, conventional single-path speculative execution also induces false control dependences, by assuming that all instructions following a conditional branch are control dependent on the branch. These are false control dependences, just as antidependences (WAR) and output dependences (WAW) are false data dependences [Joh91].

In order to circumscribe these and other aggressively speculative techniques, we propose the *weak dependence model*, which specifies that:

detected until a potentially large number of preceding branches have resolved. This complicates the implementation and reduces the benefit of multiple-path (eager) execution [US95]. Furthermore, wide (i.e. greater than four) dispatch is difficult to implement and has adverse impact on cycle time because all instructions in a dispatch group must be simultaneously cross-checked. Even current microprocessor implementations with dispatch windows of four or less (e.g. Alpha AXP 21164 [BK95], Pentium Pro [CS95]) require multiple instruction decode and dependence-checking pipeline stages.

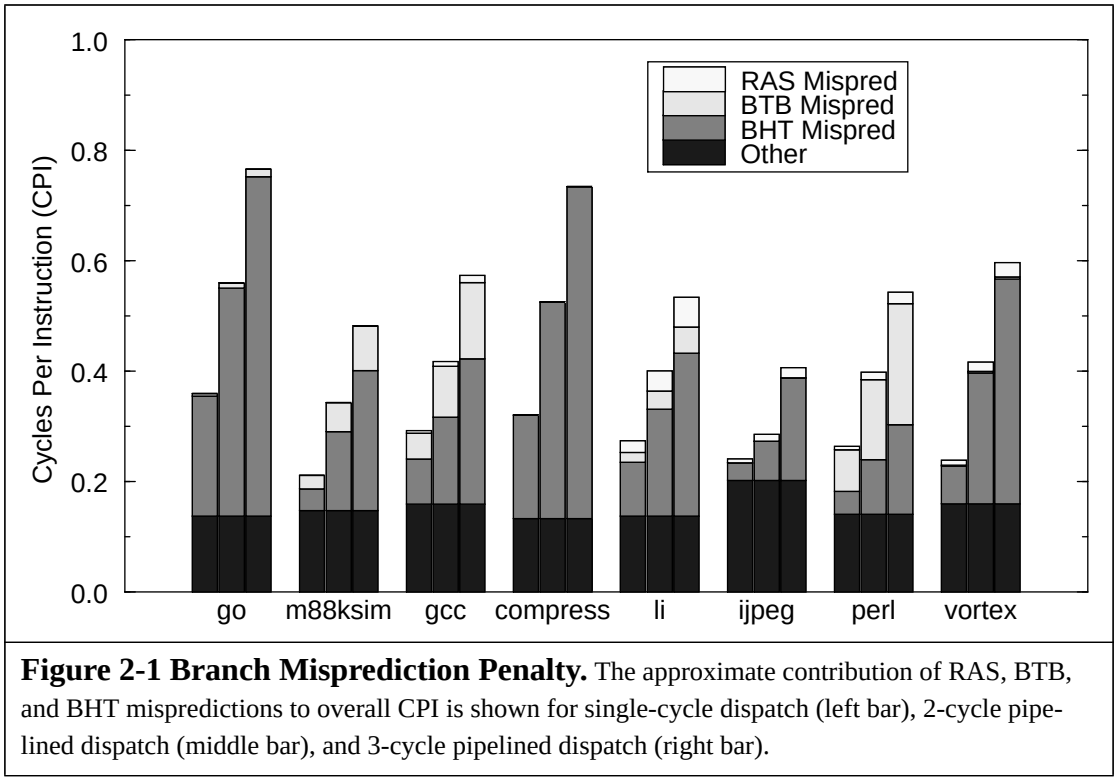

Figure 2-1 Branch Misprediction Penalty. The approximate contribution of RAS, BTB, and BHT mispredictions to overall CPI is shown for single-cycle dispatch (left bar), 2-cycle pipelined dispatch (middle bar), and 3-cycle pipelined dispatch (right bar).

One obvious solution to the problem of the complexity of dependence detection is to pipeline it into two or more stages to minimize impact on cycle time. In Section 5 we propose a pipelined approach to dependence detection that facilitates the implementation of wide-dispatch microarchitectures. However, pipelined dependence checking aggravates the cost of branch mispredictions by delaying resolution of mispredicted branches. In Figure 2-1, we see the IPC impact of pipelining dependence checking on a 16-dispatch machine with an advanced branch predictor and no other structural resource limitations (refer to Section 4.1 and Section 4.3 for further details on the benchmarks and machine model). We see that lengthening dispatch to two or three pipeline stages (vs. the baseline case of one) severely increases the number of cycles during which no useful instructions are dispatched and increases CPI (decreases IPC) dramatically, to the point where sustaining even 2-3 IPC becomes very difficult.

We propose to alleviate these problems in two ways: by introducing a scalable, pipelined, and speculative approach to dependence detection called *dependence prediction* and also by exploiting a modified approach to *value prediction* [LS96]. Fundamental to these is the notion that maintaining semantic correctness does not require that we rigorously enforce source-to-sink data-flow relationships or that we even exactly detect these relationships before we start executing. Rather, we use dynamically adaptive techniques for predicting values as well as dependences and speculatively issue instructions early, before their dependences are resolved or even known. In fact, we are exploiting a more relaxed enforcement of control and data dependences without violating semantic correctness. We call this the

Approaching 10 IPC via Superspeculation

1.0 Motivation and Related Work

Recently, some attention has been focused on the microarchitectural challenges involved in breaking the *10 IPC barrier* (i.e. sustaining 10 instructions per cycle) [Pat96]. Fundamentally, there are two restrictions that limit the degree of *IPC* that can be achieved with sequential programs: *control flow* and *data flow*. *Control flow* limits *IPC* by imposing serialization constraints at forks and joins in a program's control flow graph [ASU86]. *Data flow* limits *IPC* by imposing serialization constraints on pairs of instructions that are data dependent. Examining the extent and effect of these limits has been a popular and important area of research, particularly in the case of control flow [RF72,Wal91,LW92]. Continuing advances in the development of accurate branch predictors (e.g. [YP91]) have led to aggressive control-speculative microarchitectures (e.g. the Intel Pentium Pro [CS95]), which undertake aggressive measures to overcome control-flow restrictions by using branch prediction and speculative execution to bypass control dependences and expose additional instruction-level parallelism to the microarchitecture. Meanwhile, numerous mechanisms have been proposed and implemented to eliminate false data dependences and tolerate the latencies induced by true data dependences by allowing instructions to execute out of program order [Joh91].

Surprisingly, in light of the extensive energies focused on eliminating control-flow restrictions on parallel instruction issue, less attention has been paid to eliminating data-flow restrictions on parallel issue. Recent work has focused primarily on reducing the latency of specific types of instructions (usually loads from memory) by rearranging pipeline stages [Jou88], initiating memory accesses earlier [AS95], or speculating that dependences to earlier stores do not exist [HSS94, GCM+94].

In [LWS96], Lipasti et al. introduce the notion of *value locality*--defined as the recurrence of previously-seen values--and demonstrate a technique--*Load Value Prediction*, or *LVP*--for predicting the results of load instructions at dispatch by exploiting the affinity between load instruction addresses and the values the loads produce. In [LS96], they extend the LVP approach for predicting the results of load instructions to generalized value prediction for all instructions that write an integer or floating-point register and show that a significant proportion of such writes are trivially predictable. In the same vein, we propose exploiting *value locality* not only in the data-flow portion of a microarchitecture, but also in the control logic. We find that the dependence relationships between dynamic instructions contain a great deal of value locality, and propose a mechanism--*dependence prediction*--for capturing and exploiting this value locality to allow early dispatch of instructions in wide-dispatch machines. Furthermore, we find that combining both *value* and *dependence prediction* with limited forms of *eager execution* and *control equivalence detection* leads to significant performance increases, with harmonic mean speedups ranging, depending on machine model, from 40% to 176% for integer programs and 11% to 61% for floating-point programs.

2.0 Detecting Control and Data Dependences

Detecting control and data dependences among multiple instructions in flight is an inherently sequential task that becomes very expensive combinatorially as the number of concurrent in-flight instructions increases. Olukotun et al. argue convincingly against wide-dispatch superscalars because of this very fact [ONH+96]. Meanwhile, studies have shown that to achieve high *IPC* (sustained instructions per cycle), the number of in-flight instructions must be very large (e.g. [AS92]). Unfortunately, such studies have largely ignored the complexity of pair-wise dependence checking between instructions in flight. Dependence checking is further complicated by the presence of control-flow joins. Whenever multiple definitions reach a control-flow join, a data dependence relationship cannot even be

January 28, 1997. CMU Technical Report CMUCSC-97-1.

Approaching 10 IPC via Superspeculation

Mikko H. Lipasti and John Paul Shen

Technical Report CMUCSC-97-1

Department of Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh PA, 15213

(412) 268-3601
(412) 268-3204 (FAX)

{mhl,shen}@ece.cmu.edu

Abstract

Until recently, the serialization constraints induced by true data dependences have been regarded as an absolute limit--the data-flow limit--on the parallel execution of serial programs. Likewise, the exact detection and enforcement of these dependences has been assumed to be a requirement for semantically correct execution. This paper introduces the *weak dependence model*, which relaxes these restrictions without violating program correctness, and proposes a new microarchitectural paradigm called *superspeculation* that exploits this new dependence model to achieve uniprocessor IPC approaching ten. *Superspeculation* consists of a set of control- and data-speculative techniques that can be used to relax the serialization constraints induced by true dependences to allow instructions to issue and execute before their control or data-flow dependences are resolved or even detected. We find that dependence relationships between instructions are easily predictable, and that source operand values are frequently predictable as well. These discoveries minimize the IPC penalty of deeper pipelining of instruction decode and dispatch and enable relatively simple implementations of limited forms of eager execution and control equivalence detection that result in average integer program speedups ranging from 40% to 176% (harmonic mean), depending on machine model.

KEYWORDS: Microarchitecture, superscalar processors, speculative execution, value prediction, eager execution, control equivalence detection