The Performance Potential of Value and Dependence Prediction

Mikko H. Lipasti and John P. Shen

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

Abstract. The serialization constraints induced by the detection and enforcement of true data dependences have always been regarded as requirements for correct execution. We propose two data-speculative techniques–source operand value prediction and dependence prediction–that can be used to relax these constraints to allow instructions to execute before their data dependences are resolved or even detected. We find that inter-instruction dependences and source operand values are easily predictable. These discoveries minimize the per-cycle instruction throughput (or IPC) penalty of deeper pipelining of instruction dispatch and result in average integer program speedups ranging from 22% to 106%, depending on machine issue width and pipeline depth.

1 Introduction

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. Data flow limits IPC by forcing data-dependent pairs of instructions to serialize. Examining the extent of these limits has been a popular and important area of research (e.g. [1][2][3]). However, in light of the energies focused on eliminating control-flow restrictions on parallel instruction issue, surprisingly little attention has been paid to eliminating data-flow restrictions on parallel issue.

In [5], Lipasti et al. introduce the notion of value locality and demonstrate Load Value Prediction, or LVP, for predicting the results of load instructions at dispatch. In [5], they generalize the LVP approach to all instructions to allow one to exceed the classical dataflow limit.

Detecting data dependences between 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 [7]. Wide dispatch is difficult to implement and has adverse impact on cycle time because all instructions in a dispatch group must be simultaneously cross-checked.

One obvious solution to the problem of the complexity of dependence detection is to pipeline it to minimize impact on cycle time. In Section 3 we propose a pipelined approach to dependence detection that facilitates implementation of wide instruction dispatch. However, pipelined dependence checking aggravates the cost of branch mispredictions by delaying resolution of mispredicted

Table 1. Benchmark set

Benchmark	Run Length	BHT Mispred	BTB Mispred	RAS Mispred
go	79.6M	12.0 % (1.7%)	8.5% (0.8%)	0.0% (0.0%)
m88ksim	107.0M	2.7 % (0.4%)	4.3% (0.5%)	0.0% (0.0%)
gcc	181.8M	5.1 % (0.6%)	8.7% (1.1%)	3.9% (0.0%)
compress	39.7M	5.9 % (0.7%)	0.0% (0.0%)	0.0% (0.0%)
li	56.8M	3.0 % (0.4%)	5.3% (0.5%)	12.1% (0.3%)
ijpeg	92.1M	2.7 % (0.4%)	0.6% (0.1%)	18.7% (0.1%)
perl	50.1M	2.4 % (0.3%)	11.1% (1.2%)	4.1% (0.1%)
vortex	153.1M	0.6 % (0.1%)	1.6% (0.2%)	11.4% (0.1%)

branches. In Figure 1, we see the IPC impact of pipelined dependence checking on a 16-dispatch machine with an advanced branch predictor and no other structural resource limitations (refer to Section 2 for further details). Lengthening dispatch to two or three pipeline stages 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 and speculative approach to dependence detection called dependence prediction and also by exploiting a modified approach to value prediction called source operand value prediction [6]. Fundamental to these is the notion that maintaining semantic correctness does not require rigorous enforcement of source-to-sink data-flow relationships or even exact detection of these relationships before we start execution. Rather, dynamically adaptive techniques for predicting values as well as dependences allow early issue of instructions, before their dependences are resolved or even known. We find that the dependence relationships between instructions are quite predictable, and propose dependence prediction for capturing and exploiting this value locality to allow early issue of instructions in wide-dispatch machines. Furthermore, we find that combining value and dependence prediction leads to significant performance increases.

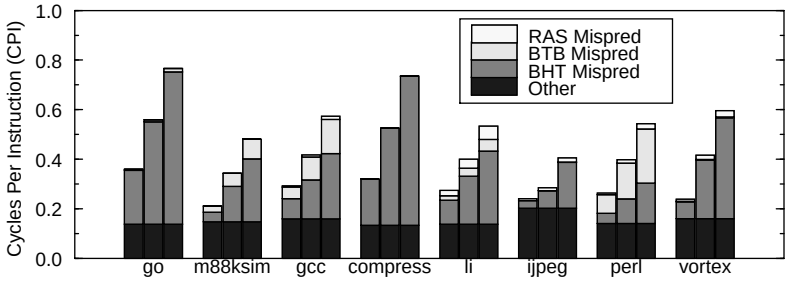

Fig. 1. Branch Misprediction Penalty. The approximate contribution of RAS, BTB, and BHT mispredictions to overall CPI is shown for single-cycle dispatch (left bar), 2-cycle (middle bar) and 3-cycle (right bar) pipelined dispatch.

2 Experimental Framework

To evaluate the performance potential of the dependence prediction and source operand value prediction, we implement a flexible emulation-based simulation framework for the PowerPC instruction set. The simulation framework is built around the PSIM PowerPC functional emulator that is distributed as part of the GDB debugger, and accurately models branch and fetch prediction, dispatch width constraints, and all branch misprediction and data dependence delays for realistic instruction latencies.

We selected the SPEC95 integer benchmark suite for our study, since it is readily available, widely used, and well-understood. Table 1 shows the run length and BHT (branch history table), BTB (branch target buffer), and RAS (return address stack) branch misprediction rates. The BHT, BTB, and RAS misprediction rates are shown with respect to total number of predictions and total number of completed instructions. The benchmarks are compiled for PowerPC GCC version 2.7.2 at full optimization. PSIM emulates user-state and NetBSD library code, but does not account for supervisor-state execution. All of the benchmarks are run to completion with reduced input sets and/or fewer iterations than in the SPEC95 reference runs.

The machine model used in our simulations has a canonical four-stage pipeline: fetch, dispatch, execute, and complete. The width of the fetch and dispatch stages can be varied 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 2. All functional units are pipelined, all architected registers are dynamically renamed, and instructions are allowed to execute out-of-order subject to data dependences. However, branches are executed in program order, and loads must wait until the addresses of all preceding stores are known. If an alias to a store exists, the load is delayed until the store's data becomes available and is forwarded directly to the load (in effect, the processor renames memory).

Our model uses a modern *gshare* branch predictor [4] with a 256K entry branch history table (BHT) with 2 bits per entry that is indexed by the XOR of the 18-bit branch history register and the branch instruction address. The RAS and the untagged, direct-mapped BTB both have 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

Table 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	2
Branch(pred/mispred)	1	0/1

Table 3. Machine Model Parameters

Parameter	Value
Branch Predictor	3-wide gshare(16)
Fetch and Dispatch Width	4,8,16
Completion Width and Instruction Window	Unrestricted
Instruction and Data Cache	Perfect

specified fetch width (this configuration is described as interleaved sequential in [8]). Up to three conditional branches can be predicted per cycle with the gshare predictor. The key parameters for the machine model used in our simulations are summarized in Table 3.

Additional simulation results for a variety of machine models as well as floating point benchmarks were omitted from this paper, but are available in [9].

3 Pipelined Dispatch Structure

In this section, we describe a pipelined dispatch structure that facilitates wide dispatch 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.

As shown in Figure 2, 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

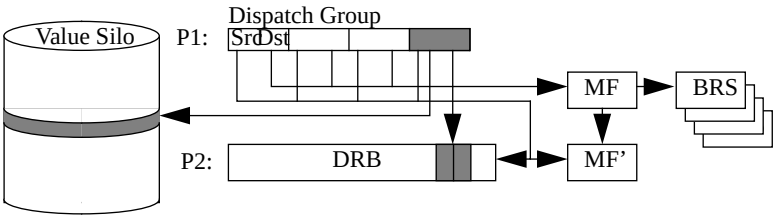

Fig. 2. Pipelined Dispatch Structure. During stage P1, all instructions in a fetch group write destination register rename mappings into the DRB and the MF. During stage P2, the instructions search the DRB and MF' for source register rename mappings.

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.

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.

4 Dependence Prediction and Recovery

Figure 1 illustrates the detrimental performance effects of a pipelined dispatch structure, which 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 effects, we propose

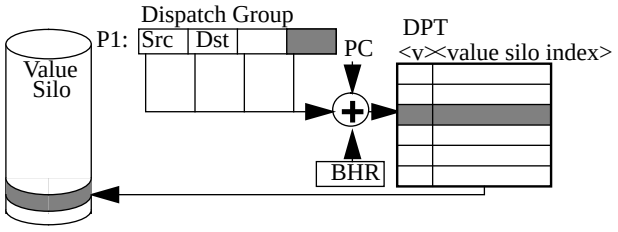

Fig. 3. Dependence Prediction Mechanism. During stage P1, the source operand position, PC, and branch history register (BHR) are hashed together to index the dependence prediction table (DPT), which predicts the value silo entry that contains the source operand. During P2 the prediction is verified.

dependence prediction that can frequently short-circuit multi-cycle dispatch by predicting the dependences between instructions in flight and speculatively allowing instructions that are predicted to be data ready to execute in parallel with exact dependence checking.

As shown in Figure 3, dependence prediction is implemented with a direct-mapped dependence prediction table (DPT) with 8K entries 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 issue 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 issue is cancelled.

The total number of operands predicted, average number of predictions per instruction, and percentage of correct predictions are shown in Table 4. We find that for most benchmarks, the DPT achieves a respectable hit rate. For two benchmarks–go and perl–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 1 and Table 1, both go and perl have high BTB misprediction rates, while go has a high BHT misprediction rates.

In Figure 5 we show the effect of dependence prediction on IPC for dispatch widths of four, eight, and sixteen, and dispatch latencies of one, two and three cycles. Without dependence prediction, the best performance is obtained with single-cycle dispatch, which sustains about 2.8 IPC in the worst case (go), and 4.8 IPC in the best case (m88ksim) with 16-wide dispatch. 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

Table 4. Dependence Prediction Results

Benchmark	Operands Pred	Per Instr	Correct
go	89.6M	1.126	38.0%
m88ksim	113.4M	1.060	77.4%
gcc	74.2M	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%

IPC. Furthermore, wider dispatch provides rapidly diminishing returns, hence eroding incentive for building processors with dispatch widths exceeding four. Fortunately, dependence prediction is able to alleviate these trends by reducing the average dispatch latency. For both two- and three-cycle dispatch, dependence prediction significantly elevates the sustainable IPC and brings it much closer to the single-cycle case. Furthermore, wider dispatch again harvests greater IPC, 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.

5 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 [6]. In earlier work, the destination operands (i.e. results) of instructions were predicted via table-lookup at fetch/dispatch, and then speculatively 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.

Source operand value prediction is illustrated in Figure 4. As in [6], we use a value prediction table (VPT) to keep track of past operand values, and exploit the value locality [5] 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 value prediction also uses a direct-mapped classification table (CT) similar to the one proposed in [6] 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

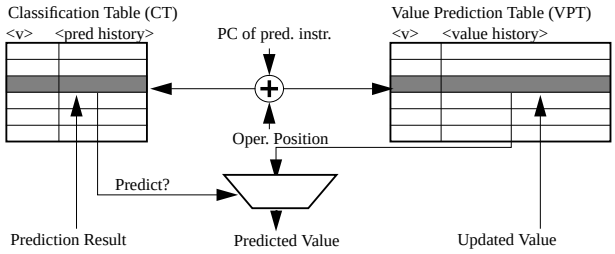

Fig. 4. Source Operand Value Prediction Mechanism. The source operand position and PC are hashed together to index the VPT and CT. The prediction and value histories are updated at completion.

Table 5. Source Operand Value Prediction Results

Benchmark	Value Locality	CT Pred Hit Rate	CT Unpred Hit Rate	Dep Pred 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%

the instruction address bits and the relative position of the operand being looked up.

When all of the input operands of an instruction are classified as predictable, the instruction is permitted to issue 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 [6], 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.

Table 5 summarizes the value locality, classification hit rates, and dependence prediction hit rates for each of our benchmarks. Value locality (column two), as defined in [5], 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 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 4, indicating that the techniques are mutually synergistic. We also note that the value locality numbers are similar to those reported earlier [6], while the CT hit rates are somewhat better.

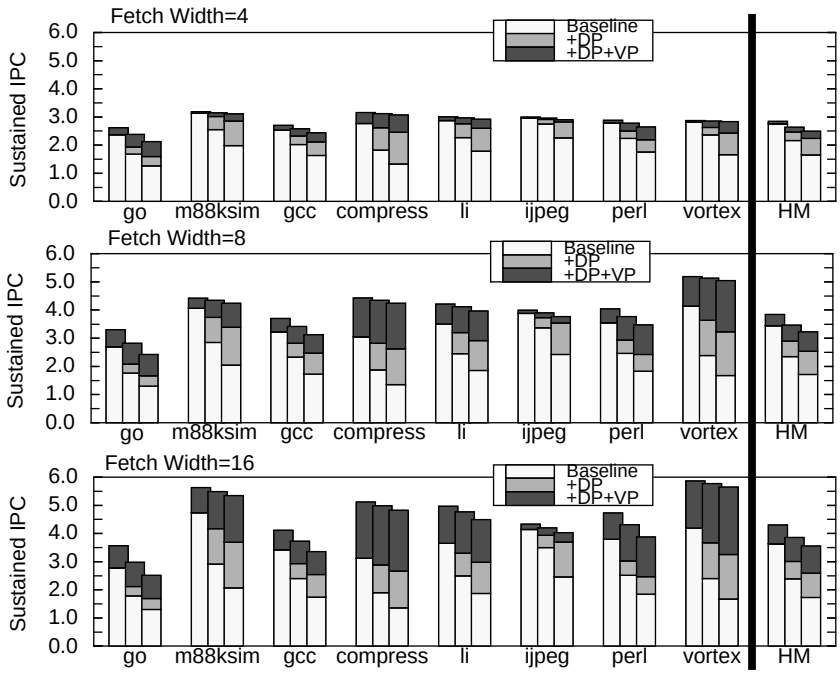

Fig. 5. Effect of Dependence and Value Prediction. The sustained IPC for dispatch widths of 4, 8 and 16 is shown for single-cycle dispatch (left bar), two-cycle dispatch (middle bar), and three-cycle dispatch (right bar). Each stacked bar shows cumulative IPC attainable with dependence prediction (+DP) and value prediction (+DP+VP).

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 larger CT size used in these experiments.

In Figure 5 we show the effect of dependence prediction and value prediction on IPC for various dispatch widths of four, eight, and sixteen, and dispatch latencies of one, two and three cycles. The best performance, obviously, is obtained with value prediction and single-cycle dispatch, which sustains 3.6 IPC in the worst case (go), and 5.9 IPC in the best case (vortex) with 16-wide dispatch. Lengthening dispatch latency to two and three cycles degrades go to 3.0 and 2.8 IPC, respectively, while reducing vortex to 5.8 and 5.6 IPC. Value and dependence prediction improve performance significantly over the baseline in all cases, and wider dispatch harvests even greater additional IPC, restoring incentive for building wide-dispatch processors.

We see that with dependence and value prediction, 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 dis-

patch 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.

6 Conclusions

We make three major contributions in this paper. First of all, we propose a pipelined dispatch structure that eases the implementation of wide-dispatch microarchitectures. Second, we propose dependence prediction, a speculative technique for alleviating the performance penalty of multi-cycle pipelined dispatch. Third, 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. We show that these techniques can speculate beyond data-flow and dependence detection bottlenecks to deliver significant improvements in uniprocessor performance, particularly for machines with wide and deeply pipelined instruction dispatch.

7 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.

References

1. Riseman, E., Foster, C.: The inhibition of potential parallelism by conditional jumps. IEEE Transactions on Computers, pages 1405-1411, December 1972.
2. Lam, M., Wilson, R.: Limits of control flow on parallelism. In Proceedings of ISCA-19, pages 46-57, June 1992.
3. Sazeides, Y., Vassiliadis, S., Smith, J.E.: The performance potential of data dependence speculation & collapsing. In Proceedings of MICRO-29, December 1996.
4. McFarling, S.: Combining branch predictors. Technical Report TN-36, Digital Equipment Corp, June 1993.
5. Lipasti, M.H., Wilkerson, C., Shen, J.P.: Value locality and load value prediction. In Proceedings of ASPLOS-VII, October 1996.
6. Lipasti., M.H., Shen, J.P.: Exceeding the dataflow limit via value prediction. In Proceedings of MICRO-29, December 1996.
7. Olukotun, K., Nayfeh, B.A., Hammond, L., Wilson, K., Chang,K.: The case for a single-chip multiprocessor. In Proceedings of ASPLOS-VII, October 1996.
8. Conte, T., Menezes, K., Mills, M., Patel, B.: Optimization of instruction fetch mechanisms for high issue rates. In Proceedings of ISCA-22, pages 333-344, June 1995.
9. Lipasti, M.H., Shen, J.P.: Approaching 10 IPC via superspeculation. Technical Report CMU-MIG-1, Carnegie Mellon University, 1997.