Speeding Up Processing with Approximation Circuits

Approximation can increase a microprocessor's clock frequency by replacing a complete logic function with a simplified circuit that mimics the function and uses rough calculations to speculate and predict results.

Shih-Lien Lu
Intel

Microprocessor performance has accelerated rapidly in recent years, primarily by achieving gains on two fronts. On one front, microarchitecture innovations have taken advantage of the increasing number of devices to process more useful instructions per cycle, predominantly through the superscalar[1] approach. On the other front, device miniaturization improves layout density and makes the circuits run faster because electrons travel a shorter distance.

A superscalar processor issues multiple instructions and executes them with multiple identical function units. It employs dynamic scheduling techniques and executes instructions outside the original program's order. Superscalar processing seeks mainly to exploit as much instruction-level parallelism as possible in the program.

Clever new circuit techniques further accelerate the logic as well. Together with finer pipe stages, its greatly accelerated clock frequency gives the modern microprocessor more cycles per unit time.

The combination of these two advances lets the system process more instructions per unit of time. However, most researchers believe that to continue exploitation of larger instruction-level parallelism, even more complexity will be necessary.[2]

This complexity increase tends to cause more circuit delay in the pipeline's critical path, thus limiting the clock frequency from rising further. The current approach lets logic structures with long delays spread over multiple pipe stages, which causes logic structures that completed the computation in single pipe stages previously to take more than one cycle time.

Using finer pipeline stages increases pipeline latencies and imposes higher penalties due to branch misprediction and other misspeculation. Moreover, other instructions that depend on the results of these multistaged functional blocks must wait until they finish to move forward in the pipeline.

Some researchers[3] have demonstrated the impact of data dependencies and branch penalties on pipeline performance. Since these two factors converge to undercut the performance gain of increasing pipeline stages, there is an optimal number of pipe stages with which a microprocessor will achieve maximum performance. This means that increasing pipe stage and frequency alone does not guarantee improved performance. Therefore, these long-delaying logic structures could become the microprocessor performance bottleneck as clock frequency continues to rise.

Thus, one challenge in achieving higher performance in future microprocessors is to increase instructions per cycle without compromising the increase in clock frequency. Current microprocessors use circuits based on Boolean functions as elementary units. The processor employs a global timing reference to synchronize data transfer between units.

For this type of synchronous system, it is essential to know the maximum time needed to compute a function, also known as the worst-case circuit delay. The depth of a circuit in gates, as well as each gate's delay, determines the worst-case delay's upper

0018-9162/04/$20.00 © 2004 IEEE Published by the IEEE Computer Society

Approximation uses rough calculations to speculate and predict results.

bound, which decides the frequency of the global timing reference. However, it is well known that a unit's delay depends strongly on its input. Usually, a circuit finishes the computation earlier than the worst-case delay—this interval is referred to as the *typical delay*. In these cases, the system still must wait for the maximum time bound to guarantee that the result is correct every time.

Faced with these constraints, my colleagues and I have turned to *approximation* to increase the clock's frequency. In this approach, instead of implementing the complete logic function necessary to realize a desired functionality, a simplified circuit mimics it. In contrast to traditional value prediction,[4] which relies on value history or value behavior, approximation uses rough calculations to speculate and predict results. The approximation circuit usually produces the correct result. If the approximate circuit fails—which usually occurs in worst-case delays—the machine employing approximation circuits must recover. Recovery degrades the overall performance. Thus, it is essential to ensure that the gain from reducing the worst-case delay outweighs the recovery overhead.

CRITICAL PIPELINE STAGES

We have applied the approximation concept to a few stages in a superscalar processor: execution, rename logic, and issue logic.

Execution

This stage consists of an approximate adder and a Booth's multiplier.

Approximate adder. Many instructions require addition. Load, store, and branch use the adder for address calculation. Arithmetic instructions use the adder for add, subtract, multiply, and divide calculations. The adder is a key performance structure in function units.

There are many different kinds of adders. Due to performance requirements, most current high-performance processors employ one of the known parallel adders. These parallel adders, such as carry-look-ahead, Brent-Kung, Kogge-Stone, and carry-select, all have comparable asymptotic performance when implemented in CMOS with either static or dynamic circuits. That is, their critical path delay is asymptotically proportional to log (N), where N is the number of bits used in the addition. The cost complexity of parallel adders approaches $N\log N$ when the fan-in and fan-out of the gates used are fixed.

The adder's full carry chain determines the critical path. To generate the correct final result, the calculation must consider all input bits to obtain the final carry out. However, in real programs, inputs to the adder are not completely random and the effective carry chain is much shorter for most cases. Thus, it is possible to build a faster adder with a much shorter carry chain to approximate the result. Since carry chains are usually much shorter, a design that considers only the previous k inputs (look-ahead k-bits) instead of all previous input bits for the current carry bit can approximate the result:

$c_i = f(a_{i-1}, b_{i-1}, a_{i-2}, b_{i-2}, \ldots, a_{i-k}, b_{i-k})$, where $0 < k < i + 1$ and $a_j, b_j = 0$ if $j < 0$.

Given that the delay cost of calculating the full carry chain length of N bits is proportional to log (N), if k equals the square root of N, the new approximation adder will only need on the order of half the delay. With random inputs, the probability of having a correct result considering only k previous inputs is:

$$P(N,k) = \left(1 - \frac{1}{2^{k+2}}\right)^{N-k-1}.$$

This is derived with the following steps. First consider why the prediction is incorrect. If we only consider k previous bits to generate the carry, the result will be wrong only if the carry propagation chain is greater than $k + 1$. Moreover, the previous bit must be in the carry-generate condition.

This can only happen with a probability of $1/2^{k+2}$ if we consider a k-segment. Thus, the probability of being correct is one minus the probability of being wrong. Second, there are a total of $N - (k + 1)$ segments in an N-bit addition. To produce the final correct result, the segment should not have an error condition. We multiply all the probabilities to produce the final product. This equation could determine the risk taken by selecting the value of k. For example, assuming random input data, a 64-bit approximation adder with 8-bit look-ahead ($k = 8$) produces a correct result 95 percent of the time.

Figure 1 shows a sample approximation adder design with $k = 4$. The top and bottom rows are the usual carry, propagate, and generate circuits. The figure also shows the sum circuits used in other parallel adders. However, the design implements the carry chain with 29 4-bit carry blocks and three boundary cells. These boundary cells are similar but smaller in size. A Manchester carry chain could implement 4-bit carry blocks. Thus, the critical path

delay is asymptotically proportional to constant with this design, and the cost complexity approaches N. In comparison with Kogge-Stone or Han-Carlson adders, this design is faster and smaller.

Since we know exactly what causes a result to be incorrect, the design could—and probably should—implement an error indication circuit. Whenever a carry propagation chain longer than k bits occurs, the approximation adder circuit will give an incorrect result. That is, for the ith carry bit, if the logic function (a_{i-1} XOR b_{i-1}) AND (a_{i-2} XOR b_{i-2}) AND ... AND (a_{i-k} XOR b_{i-k}) AND (a_{i-k-1} AND b_{i-k-1}) is true, the prediction will be wrong. The adder could implement this logic function for each carry bit and perform the logical OR of all these $n - 4$ outputs to signal if the approximation is incorrect. Instead of comparing the result of a fully implemented adder with this approximated adder, the error indication circuit provides a signal to select the correct result earlier in the process to help the pipeline recover.

Booth multiplier. Current microprocessors use Booth encoding and the Wallace tree to perform the multiply function. For radix-8 Booth encoding, generating $3x$ lies in the critical path. Unlike $2x$, $4x$—which can be generated by shifting—there is no easy way to generate $3x$ besides performing the actual addition. Again, the adder could approximate the multiplier to accelerate the frequency or reduce the number of cycles required. The straightforward approach uses the approximation adder to generate $3x$. However, its accuracy is unsatisfactory for the overall multiplication because errors accumulate. Remembering that an error occurs only when a carry propagation chain stretches longer than k, inspecting the differences between the correct and approximated result reveals that the discrepancy between them is always an N-bit vector sparsely populated with 1's separated by at least k 0's. Writing the equation using the propagate and generate (p,g) terms gives:

$$p_{i-1} \text{ AND } p_{i-2} \text{ AND } \ldots \text{ AND } p_{i-k} \text{ AND } g_{i-k-1}.$$

To obtain the correct multiplication result, these error vectors must be included in the Wallace tree. A simpler circuit also can approximate the sum of error vectors. For example, an OR gate can approximate the summation of all error vectors. This adds only one extra partial product for the Wallace tree to compress. The added delay to the tree is negligible, but the process has reduced the delay to generate $3x$. Developers also could use this method to design a nonapproximated multiplier.

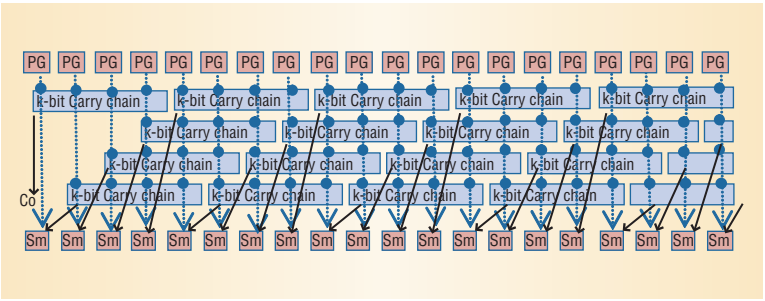

Figure 1. Sample 32-bit approximation adder. The adder has the usual carry, propagate, generate, and sum circuits but also implements a carry chain with 29 4-bit carry blocks and three boundary cells.

Rename logic

The register rename logic's critical path centers on the associative lookup delay and the priority logic when multiple matches are found. Experiments with benchmarks revealed that dependent instructions can have spatial locality: The instructions are most likely to be close to each other. Thus, the design can use a smaller *content-addressable memory* to implement the mapping table.

The CAM table basically contains a portion of the entire map. When a new instruction enters the rename logic, the CAM renames its destination binding and assigns it a new physical binding. The mapping table then updates if it is not full. Otherwise, the processor drops the oldest binding to leave room for the newly renamed destination binding.

At the same time, the processor uses the source bindings to look up the partial CAM. If it finds no physical mapping in the small CAM, but the mapping does exist in the full CAM, a misspeculation occurs. Since the number of inputs to the priority encoder equals the number of entries in the smaller CAM, the rename logic delay is also smaller.

Using a much smaller CAM table that contains only a number of instructions equal to the latest N-square-root register mapping, where N is the window size, doubles the speed. Given the locality property of register dependency, most of the reading operation from the rename logic should be correct. In addition to the faster approximation renaming logic, this design retains a regular CAM and the associated full-length priority encoder. This will recover the misspeculation and provide the correct renaming result in the next cycle.

Issue logic

This approach uses this same idea by targeting the issue logic on the earliest square-root-N entries so that the issue logic only needs to consider waking up, selecting, and bypassing data to instructions within square-root-N entries to the head of register update unit (RUU). Because resistance-capacitance dominates the wakeup and bypass delay and RC delay is more sensitive to window size, the speed increase will more than double in these two

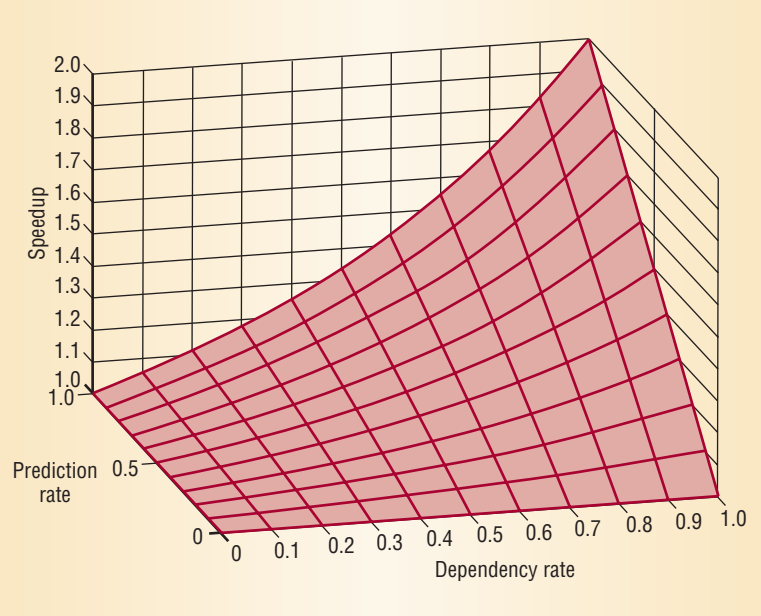

Figure 2. Speedup of speculative execution as a function of prediction rate and instruction dependency rate. The functional unit writeback-bandwidth-occupancy rate is 50 percent.

logics. Thus, the total speculative issue logic delay will be less than half the issue logic in the baseline microarchitecture if only square-root-*N* entries are considered. The approximated issue logic does not need a replay because this approach generates no false results. However, some bandwidth or functional units may be wasted because the square-root-*N* entries might not have enough ready instructions.

PERFORMANCE IMPACT

Using analytical modeling and simulation helps to assess the optimization's impact on performance.

Simple analytical modeling

Philip Emma and Edward Davidson[3] showed that as the length of any pipeline increases, data dependencies and branches monotonically degrade the pipeline performance in terms of clock cycles per instruction. The longer the pipeline, the more penalty cycles the data dependency and branch misprediction cause. However, increasing the pipeline length increases clock frequency monotonically. These two opposing factors will decide the optimal pipeline length based on specific technology.

A simple analytical model based on an in-order machine overcomes these data dependencies. The baseline model and speculative model run at the same frequency. In the baseline machine, the functional unit has a two-cycle execution time and the speculative unit has a one-cycle execution time with a replay penalty. Obviously, under the same frequency, the model with the shorter pipeline will suffer less from data dependency and branch mispredictions. However, the model will replay the wrongly speculated result, which will occupy more functional-unit-writeback bus bandwidth, reducing the performance gain.

The following are major factors in the performance comparison:

- prediction rate (PR) of the speculative logic,
- data-dependency rate (DR) for the instructions,
- functional unit writeback bus occupancy rate (FR), and
- overall branch miss rate (BR).

Notice that the overall branch miss rate is the product of the branch miss rate and branch frequency. Since the goal is to evaluate data dependency, the comparison simplifies the branch prediction factor to one term.

Since both machines modeled have the same frequency, the comparison performance metric is mainly cycle per instruction. The formula used for CPI with data dependency and branch penalty is

$$CPI = 1 + PD \times C_{Dstall} + PB \times C_{Bstall}$$

where PD is the probability that data dependency exists between two adjacent instructions, PB is the probability that an arbitrarily selected instruction is a branch and that the branch prediction is wrong. C_{Dstall} and C_{Bstall} are the corresponding stalled cycle when the dependency and branch misprediction takes place.

For simplicity, the comparison further assumes that C_{Bstall} is three cycles for both models. For the baseline pipe stage structure, PD is data dependency rate DR, and the stalled cycle is 1. Thus:

$$CPI = 1 + DR + PB \times C_{Bstall}.$$

A pipeline structure with speculative functions has four boundary cases. Either all instructions are

1. independent and the prediction rate is 100 percent,
2. independent and the prediction rate is 0,
3. dependent and the prediction rate is 100 percent, or
4. dependent and the prediction rate is 0.

For cases 1 and 2, when the prediction is perfect, there is no data-dependency penalty. For case 2, the verification logic will reissue the instruction in the next cycle. This requires an extra writeback slot for the reissued instruction. While the impact of the extra writeback slot on performance is complicated, the comparison can approximate the rela-

tionship by the following method: If the functional unit's FR is 100 percent for the original instructions, the extra writeback will always stall the pipeline by one cycle. If the original instructions have an occupancy rate of 50 percent or less, this extra writeback will not stall the pipeline. CPI will not be affected in this case.

A linear approximation interpolates the relationship between C_{Dstall} and FR. The linear equation needs to satisfy the two boundary conditions mentioned:

$$C_{Dstall}(FR) = 2 \times FR - 1.$$

For case 4, the analysis resembles case 2 but differs in that all instructions are dependent so that the pipeline will be stalled for one cycle even when no limitation on writeback bandwidth exists. This means that the pipeline will be one cycle when FR is 50 percent and two cycles when FR is 100 percent:

$$C_{Dstall}(FR) = 2 \times FR.$$

Combining all cases together with branch prediction provides the following:

$$CPI = 1 + (2 \times FR - 1) \times (1 - DR) \times (1 - PR) + 2 \times FR \times DR \times (1 - PR) + PB \times C_{Bstall}.$$

Let the speedup be the ratio of baseline CPI and speculative CPI. Figures 2, 3, and 4 show the speedup when FR is 0.5, 0.8, and 0.95, respectively. These figures assume either no branch misprediction impact or a perfect branch prediction rate. As the diagrams indicate, the speedup rate increases monotonically with dependency rate and prediction rate.

When the functional-unit occupation rate is high, the speculative method is more likely to sacrifice performance because replay instructions cause more penalties in writeback bandwidth. If FR is more than 50 percent, when the prediction rate or dependency rate is low enough, the speculative microarchitecture performance drops below the baseline. In an extreme case, when the dependency rate is 0, the speedup increases with prediction rate. However, the maximum speedup rate is 1, which means no speedup occurs even with perfect prediction and, with less than perfect prediction, performance actually decreases. The results show that the speculative method only works for the case in which the instruction-dependency rate is significant.

In another extreme case, when the prediction rate is 0, the speedup increases with the dependency rate but remains lower than 1. This means that the speculation method requires a minimum prediction rate to achieve any performance improvement.

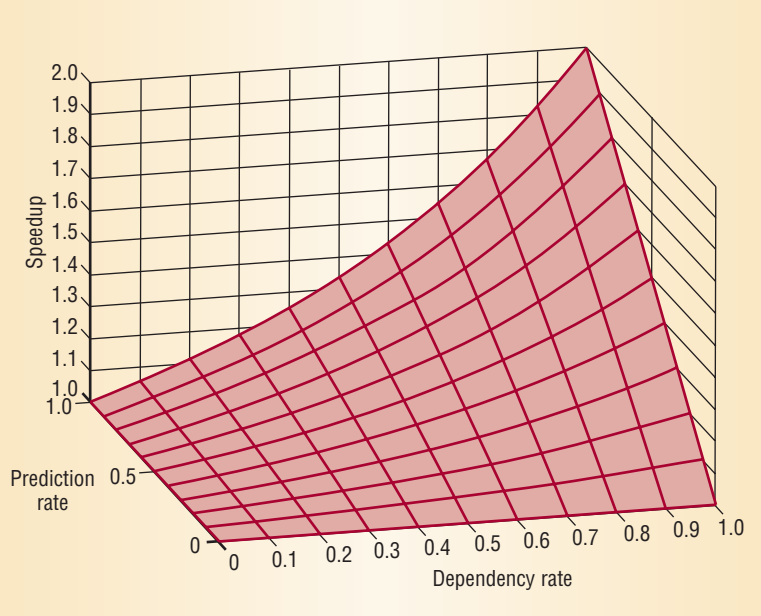

Figure 3. Speedup of speculative execution as a function of prediction rate and instruction-dependency rate. The functional unit writeback-bandwidth-occupancy rate is 80 percent.

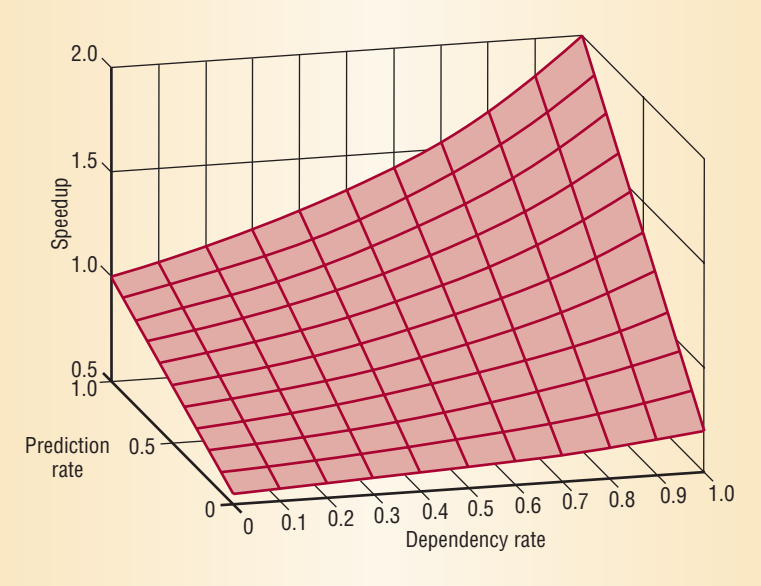

Figure 4. Speedup of speculative execution as a function of prediction rate and instruction-dependency rate. The functional unit writeback-bandwidth-occupancy rate is 95 percent.

Figures 5 and 6 show the impact of the overall branch misprediction rate and dependency rate on performance when the data prediction rate is high and the writeback-occupancy rate is medium. At the lower data-dependency rate, when speculative performance is low, the performance speedup increases as the PB increases; at the higher data-dependency rate, when speculative speedup is high, the speedup decreases as the PB increases.

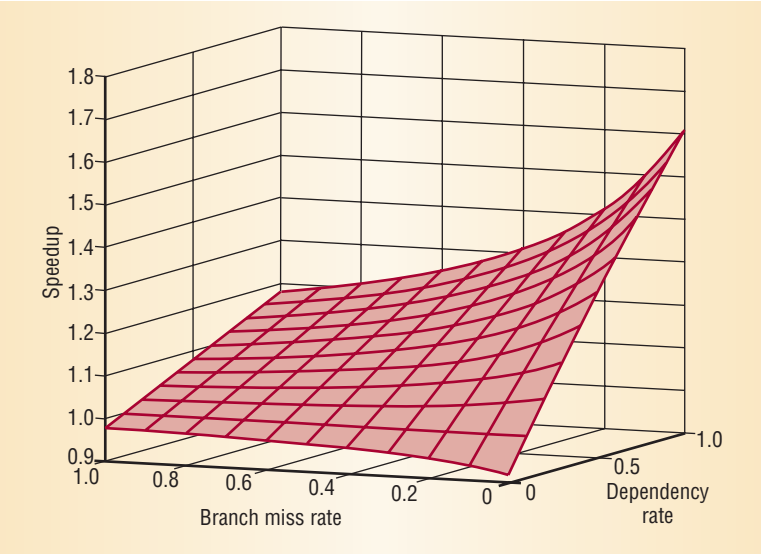

Figure 5. Speedup of speculative execution as a function of overall branch-misprediction rate and instruction-dependency rate. The functional unit writeback bandwidth occupancy rate is 80 percent and the data-prediction rate is 85 percent.

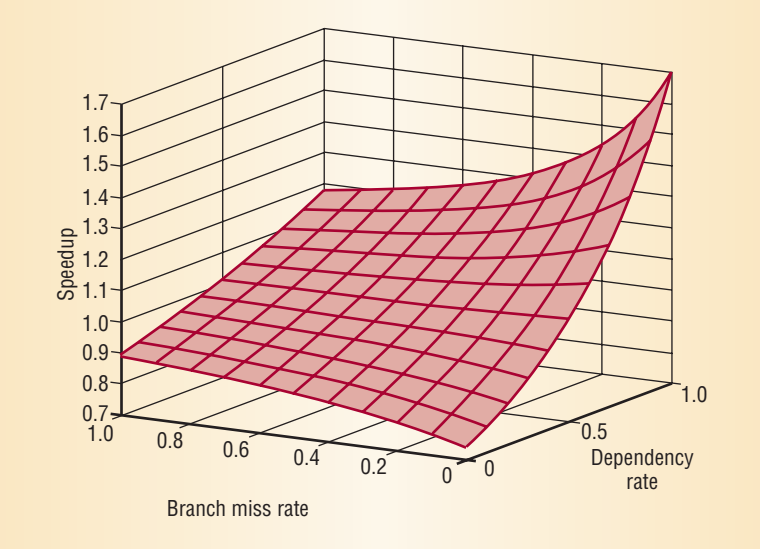

Figure 6. Speedup of speculative execution as a function of overall branch-misprediction rate and instruction-dependency rate. The functional unit writeback bandwidth occupancy rate is 80 percent and the data-dependency rate is 70 percent.

In the first case, data speculation suffers more in replay penalties than it gains in speculation performance because this case lacks dependent instructions. In the second case, the baseline model suffers more from a dependency stall penalty than it gains in performance from not replaying wrongly speculated instructions. The higher PB also causes more performance loss on both models at the same rate, thus it reduces the performance ratio.

The overall branch-misprediction rate favors the worst-case model because it compromises the data-dependency rate's performance impact. Figure 6 shows the relationship of speedup to PB and PR. For the same reason, the overall branch-misprediction rate will compromise the data-prediction rate's impact. This analysis suggests that a good branch-prediction rate is important for reaping the benefits of data speculation.

Simulation

The SimpleScalar tool set[5] provides a method for comparing the speculative microarchitecture's performance with the baseline machine.

Method. Assume both models run with the same frequency. In the baseline machine, to maintain the frequency, all cycle-limiting logic blocks take two cycles. In the speculative machine with approximation circuits, these same logic blocks take only one cycle. However, the speculative machine will need to replay when it incorrectly generates the result and incurs a misspeculation penalty. The independent simulation experiment uses the rename logic, issue logic, and adder, assuming that only one of these components provided the main performance limiter. The simulation, which ran several benchmarks from the SPEC suite, used the reference input.

Findings. These experiments revealed that the approximation adder's accuracy is much higher than the derived probability using random data inputs. With random data, the anticipated prediction accuracy was around 65 percent for 32-bit addition with a 4-bit carry chain. However, the simulation results show that close to 90 percent of the addition is correct using the approximation adder with a 4-bit carry and inputs from real applications.

The approximated rename logic produces close to 80 percent correct results. However, the approximated issue window logic has a low accuracy of only about 40 percent. The experiments evaluated the impact of approximation variance in two parameters: issue width and out-of-order window size.

After setting the RUU window size to 64, issue width to 4, integer adder number to 4, and integer multiplier number to 1, the experiment ran two billion instructions for each benchmark. Next, the study shuffled the parameters to obtain a window size of 16×32, an issue width of 8, an integer adder number of 8, and an integer multiplier number of 2, and ran each benchmark for 500 million instructions. Then the performance differences were compared with the baseline machine, normalized to one. The simulation showed that using approximation to speculate data as described does improve overall performance. For adder speculation, the performance improvement is less than the other two speculations.

The simulation achieved these results because addition completes close to the machine's back end. It is thus more likely to pollute the dependent instructions by false writebacks, thereby inducing more penalties. By reducing window size, the adder speculation performance relative to the baseline machine increases. A smaller issue window has fewer independent instructions, which produces a higher dependency between instructions and helps execution complete more quickly.

On the other hand, increasing issue width and the number of function units degrades relative performance. Wider issue width, larger window size, and more functional units potentially produce more instruction-level parallelism. More instructions execute per cycle. Every time a misspeculation occurs, the penalty becomes larger, outweighing the performance gain from resolving the dependency chain earlier.

For rename and issue logics, the simulation also reduces the speculative window size so that the worst-case delay is close to half the worst-case delay of the baseline window. This will compromise the relationship between relative performance and window size, issue width, and functional unit.

When the issue width is 8, with 64 entry-instruction windows and eight execution units—which represent a wide-issue machine with large window size and many more functional units—the relative performance of programs with inherent instruction-level parallelism suffers. For example, the performance of ijpeg degrades significantly with issue speculation.

With this application, the issue-speculation prediction accuracy of 24 percent—caused by the reduced issue window size—is low. This means that the number of ready instructions in the approximation window is much smaller than the real number of ready instructions. Even though approximating issue window size exacts no recovery penalty, the loss of parallelism from approximation degrades performance greatly. It also causes a huge waste of execution bandwidth, as the analytical model validates.

Programs with high instruction-level parallelism have a lower data-dependency rate. Programs with a lower data-dependency rate will not benefit much from shortening the pipeline length. Combined with the low prediction accuracy that a smaller window size causes, pipeline performance suffered when using approximation.[6]

Using an approximation circuit to reduce the number of cycles a function requires is the first step in achieving variable pipeline delays based on data values.[7] An asynchronous pipeline such as the micropipeline that Ivan Sutherland proposed[8] will provide the ultimate variable delay due to data values. Each stage of an asynchronous pipeline will take only as much time as necessary to process the data. Once it completes the evaluation, the pipeline can forward the result to the next stage for processing without waiting for the worst-case delay. Although a misspeculation penalty does not occur, signaling the completion of execution at each stage does cause handshaking overhead.

So far, there is still no efficient way to detect a stage's completion. Perhaps other means can be found to extend the approximation concept so that it can simplify more microprocessor stages to take advantage of the typical delay. ■

References

1. J.E. Smith and G.S. Sohi, "The Microarchitecture of Superscalar Processors," *Proc. IEEE*, vol. 83, no. 12, 1995, pp. 1609-1624.
2. S. Palacharla et al., "Complexity-Effective Superscalar Processors," *Proc. 24th ACM/IEEE Int'l Symp. Computer Architecture*, IEEE Press, 1997, pp. 206-218.
3. P.G. Emma and E.S. Davidson, "Characterization of Branch and Data Dependencies in Programs for Evaluating Pipeline Performance," *IEEE Trans. Computers*, vol. 36, no. 7, 1987, pp. 859-876.
4. M.H. Lipasti and J.P. Shen, "Exceeding the Dataflow Limit via Value Prediction," *Proc. 29th IEEE/ACM Int'l Symp. Microarchitecture*, IEEE Press, 1996, pp. 226 -237.
5. D.C. Burger and T.M. Austin, "The SimpleScalar Tool Set, v. 2.0," tech. report 1342, Computer Science Dept., Univ. of Wisconsin, Madison, 1997.
6. T. Liu and S-L. Lu, "Performance Improvement with Circuit-Level Speculation," *Proc. 33rd ACM/IEEE Int'l Symp. Microarchitecture*, IEEE Press, 2000, pp. 348-355.
7. S. Komori et al., "An Elastic Pipeline Mechanism by Self-Timed Circuits," *IEEE J. Solid-State Circuits*, vol. 23, no. 1, 1988, pp. 111-117.
8. I.E. Sutherland, "Micropipelines," *Comm. ACM*, vol. 32, no. 6, 1989, pp. 720-738.

Shih-Lien Lu *is a senior staff researcher at Intel's Hillsboro Microarchitecture Lab. His research interests include computer microarchitecture, embedded systems, self-timed circuits, and VLSI systems design. Lu received a PhD in computer science from the University of California, Los Angeles. He is a member of the IEEE Computer Society and the IEEE. Contact him at shih-lien.l.lu@intel.com.*