Theme Feature

One Billion Transistors, One Uniprocessor, One Chip

To achieve the highest performance possible, the billion transistors available on each chip should be utilized to support the highest performance uniprocessor, with the resulting chips interconnected to create a multiprocessor system.

Yale N. Patt

Sanjay J. Patel

Marius Evers

Daniel H. Friendly

Jared Stark
University of Michigan

A billion transistors on a single chip presents opportunities for new levels of computing capability. Depending on your design point, several distinct implementations are possible. In our view, if the design point is performance, the implementation of choice is a very high-performance uniprocessor on each chip, with chips interconnected to form a shared memory multiprocessor.

The basic design problem is partitioning: How should the functionality be partitioned to deliver the highest performance computing system? Because one billion transistors falls far short of an infinite number, the highest performance computing system will not fit onto a single chip. We must decide what functionality cannot tolerate the latency of interchip communication. To do otherwise precludes obtaining the highest performance.

Intraprocessor communication, to be effective, must keep latency to a minimum, locating on the same chip as many as possible of the structures necessary to support a high-performance uniprocessor. These structures include those necessary both for aggressive speculation, such as a very aggressive dynamic branch predictor, and for very-wide-issue superscalar processing, such as

- a large trace cache,
- a large number of reservation stations,
- a large number of pipelined functional units,
- sufficient on-chip data cache, and
- sufficient resolution and forwarding logic.

A reasonable on-chip specification would issue a maximum of 16 or 32 instructions per cycle (issue width), include reservation stations to accommodate 2,000 instructions, and include 24 to 48 highly optimized, pipelined functional units. We believe that the effectiveness of these structures continues to scale as the number of transistors on a chip increases. In our view, we will run out of transistors before we run out of functionality in support of a single instruction stream. Ergo, one billion transistors, one uniprocessor, one chip.

History argues that such an engine could never run at peak performance—that diminishing returns are inevitable. We disagree: Ingrained in the model is the flexibility of dynamic scheduling, coupled with the structures required to exploit it. True, this will require better algorithms to solve the application problems, better compiler optimizations, and better CAD tools to manage the great increase in design complexity. But one billion transistors on a chip is still a decade out, and the industry has talented people working on all these fronts.

Even if diminishing returns does prove to be the case, we suggest that it is still better to combine higher performance uniprocessor chips—where higher latency interchip communication is tolerable—than to put lower performance uniprocessors on the same chip—where lower latency communication is not essential.

Several alternatives have been suggested for obtaining highest performance with billion-transistor chips. One, a chip multiprocessor (CMP), does not correctly address the partitioning problem. The multiprocessor divides the available transistors among processors on the same chip. It would be better to put higher performing processors on separate chips and tolerate the resulting increase in interprocessor communication latency. Also, on-chip memory pin-bandwidth (that is, the number of bits of instructions and data that must cross the chip boundary via the available pins) to support a single instruction stream is a bottleneck; allocating multiple processors to one chip exacerbates that problem.

0018-9162/97/$10.00 © 1997 IEEE

Authorized licensed use limited to: Edinburgh University. Downloaded on January 26, 2010 at 10:30 from IEEE Xplore. Restrictions apply.

Multithreading will be a viable alternative when the multiprocessor pin-bandwidth problem is solved. Multithreading suffers from the same memory bandwidth requirements as a CMP, but it better utilizes the available transistors. Unlike a CMP, where the transistors are divided among all the processors, multithreading (by effective pipelining) provides the functionality of a multiple processor at the approximate transistor cost of (approximately) a single processor.[1]

IN PURSUIT OF HIGHEST PERFORMANCE

Highest performance execution of a single instruction stream involves

- delivering to the execution core (issuing) the maximum possible instruction bandwidth each cycle and
- consuming that delivered bandwidth.

Several things get in the way:

- Delivering optimal instruction bandwidth requires a minimal number of empty fetch cycles, a very wide (conservatively 16 instructions, aggressively 32), full issue each cycle, and a minimal number of cycles in which the instructions fetched are subsequently discarded.
- Consuming this instruction bandwidth requires sufficient data supply so that instructions are not unnecessarily inhibited from executing. It also requires sufficient processing resources to handle the instruction bandwidth.

A one-billion transistor chip like that shown in Figure 1 can help alleviate these problems. We suggested in 1985 that a high-performance microprocessor should contain hardware-intensive microarchitecture structures to support performance. We called our model HPS (High-Performance Substrate). The notion of aggressive hardware support for high performance will be equally valid in the year 2005.[2,3]

We view instruction delivery as the single most important problem—for example, 100 branch prediction is not possible. In this article we suggest an instruction cache system (the I cache) that provides for out-of-order fetch (fetch, decode, and issue in the presence of I cache misses). We also suggest a large sophisticated trace cache for providing a logically contiguous instruction stream, since the physical instruction stream is not contiguous. And we suggest an aggressive Multi-Hybrid branch predictor (multiple, separate branch predictors, each tuned to a different class of branches) with support for context switching, indirect jumps, and interference handling.

Memory bandwidth and latency are the second most

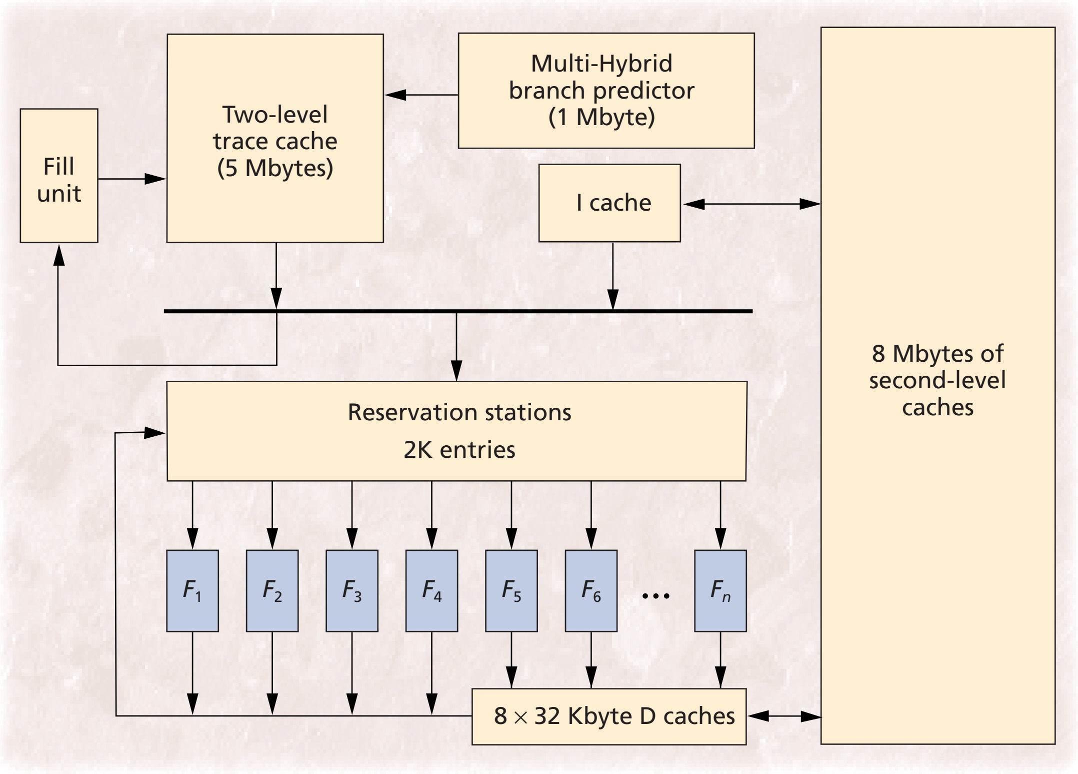
Fill unit
Two-level trace cache (5 Mbytes)
Multi-Hybrid branch predictor (1 Mbyte)
I cache
8 Mbytes of second-level caches
Reservation stations 2K entries
F_1
F_2
F_3
F_4
F_5
F_6
...
F_n
8 × 32 Kbyte D caches

Figure 1. With one billion transistors, 60 million transistors will be allocated to the execution core, 240 million to the trace cache, 48 million to the branch predictor, 32 million to the data caches, and 640 million to the second-level caches.

Authorized licensed use limited to: Edinburgh University. Downloaded on January 26, 2010 at 10:30 from IEEE Xplore. Restrictions apply.

important problem, and we allocate more than half of the billion transistors to its solution. We suggest a processing core that contains enough functional units to process the instructions, a mechanism to prevent artificial blocking, enough storage for instructions awaiting dependencies to be resolved, and no unnecessary delays between a functional unit producing a result and another functional unit requiring it.

THE TRACE CACHE

Fetching past a taken branch poses a serious problem for I cache designs—one that cannot be solved without complexity, which would increase access latency. Our design incorporates a *trace cache,* a new paradigm for caching instructions. Trace caches address the loss of instruction bandwidth due to incomplete fetches.[5,6] First suggested as an extension of the single block execution atomic unit,[4] the trace cache was developed into a viable entity by Alex Peleg and Uri Weiser.[7] Like the I cache, the trace cache is accessed using the starting address of the next block of instructions. Unlike the I cache, it stores logically contiguous instructions in physically contiguous storage. A trace cache line stores a segment of the dynamic instruction trace—up to an issue-width's worth of instructions—across multiple, potentially taken branches.

As a group of instructions is processed, it is latched into the fill unit. The fill unit attempts to maximize the size of the segment by coalescing newly arriving instructions with instructions latched from previous cycles. The fill unit finalizes a segment when the segment can be expanded no further—for example, when an issue width of instructions has been collected. Finalized segments are written into the trace cache.

Because the processing within the fill unit happens off the critical execution path, the latency of the fill unit does not directly impact overall performance.[6] The fill unit can spend several cycles analyzing and explicitly marking dependencies within a segment before writing the segment into the trace cache. With this framework, the amount of processing that must be performed when the instructions are refetched can be minimized; instructions can be sent from the trace cache into the reservation stations without having to undergo a large amount of processing and rerouting. Figure 1 includes an overview of the trace cache data path.

Figure 2 compares the performance trends of a conventional fetch mechanism with that of a trace cache mechanism, as the size of the cache increases. We simulated the three largest applications from the SPECint95 benchmarks—go, gcc, and vortex—on a 16-wide issue machine with perfect branch prediction. The data shows that

- the trace cache is effective in delivering more than one basic block per cycle, and
- the trace cache continues to add performance as the size of the storage structure is increased.

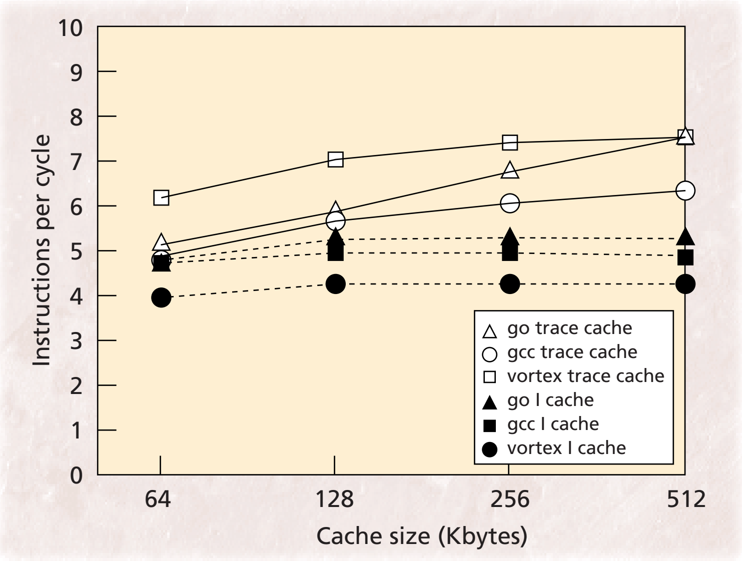

Figure 2. Performances of various fetch mechanisms as the size of the cache structure increases. At 512 Kbytes, the trace cache continues to gain performance.

Typically, the sizes of I caches are not determined by specific benchmarks, but can be as large as implementation budgets allow. However, as the I cache size grows beyond the working sets of a large class of typical applications, there is a diminishing growth in performance of single applications. The trace cache exploits this availability of storage by caching the program in an execution-oriented manner, rather than caching it for efficient storage. As a result, trace caches continue to increase performance with more space.

Even though large cache structures may fit into area budgets, they may not fit into timing budgets. Many of the techniques used to deal with the latency of accessing the I cache can be applied to the trace cache. The trace cache can be pipelined across several cycles, thus requiring a smaller one-cycle structure to cache next fetch addresses (similar to a branch target buffer in I cache designs). The trace cache can also be partitioned into two levels, with a smaller, frequently accessed, one-cycle component and a larger multicycle component. Set prediction and remapping schemes can deal with the additional latency of set-associative access.

Since its latency does not directly affect performance, the fill unit offers an intriguing framework for runtime manipulation of the program executable. First, it can boost fetch bandwidth by using runtime or compile-time information to more effectively create segments, which can contain instructions from multiple execution paths. Second, the fill unit can analyze the instructions in these segments and perform runtime code optimizations. Third, it can retarget the instruction set architecture of incoming blocks to one more efficiently supported by the hardware.

THE BRANCH PREDICTOR

To maximize the issue bandwidth, the penalties associated with branches must be avoided. Predictions must be made in a single cycle and must be very accurate.

Authorized licensed use limited to: Edinburgh University. Downloaded on January 26, 2010 at 10:30 from IEEE Xplore. Restrictions apply.

Although the proposed predictor can give very accurate predictions, accessing a predictor of this size in a single cycle is not reasonable.

Thus the branch predictor is an essential component in today's wide-issue processors, and its importance will be further emphasized in future processors.

Aggressive hybrid branch prediction

Many branches display different characteristics; some branches follow recurring execution patterns, while others depend on the outcomes of the preceding branches. A single-scheme predictor usually predicts a subset of the branches well. Our solution enlarges the subset of easily predictable branches. To do this, we use a hybrid predictor that comprises several predictors, each targeting different classes of branches. Scott McFarling first proposed combining two predictors.[8]

As predictors increase in size, they often take more time to react to changes in a program. This warm-up time can be detrimental when running a multiprogrammed workload, or even when running large programs that frequently move to different sections of the code. However, a hybrid predictor with several components can solve this problem by using component predictors with shorter warm-up times while the larger predictors are warming up. The most accurate conventional hybrid predictors consist of two large components with longer warm-up times—thus they lack this flexibility. Examples of predictors with shorter warm-up times are two-level predictors with shorter histories as well as smaller dynamic predictors.

The downside to using a hybrid predictor with many components is that some predictors only work well if they are large enough. This is generally due to the interference between different branches competing for the resources of that predictor. By increasing the number of predictor components, less hardware will be available for each of them. However, with the number of transistors likely available in the processor of the future, this detrimental effect will be overshadowed by the advantages of having several components. The interference effects on the small predictor components is further minimized by not updating the pattern history tables for mostly unidirectional branches.

One example of a multicomponent hybrid predictor is the Multi-Hybrid.[9] The Multi-Hybrid uses a set of selection counters for each entry in the branch target buffer, in the trace cache, or in a similar structure, keeping track of the predictor currently most accurate for each branch and then using the prediction from that predictor for that branch. Figure 3 shows that the Multi-Hybrid is able to take advantage of extra hardware even at fairly high implementation costs. The likely implementation cost for the processor we are proposing would be between 256 and 1,024 Kbytes. At 256 Kbytes, the average misprediction rate is still a little too high to take full advantage of a one-billion-transistor processor. However, the average misprediction rate is deceptively high due to the benchmark go, which is mispredicting almost 10 percent of the branches. Other techniques, such as predicated execution and program-based prediction, will be required for this benchmark to take advantage of the processor.

We suggest that the predictor used in a one-billion-transistor processor will be closely related to the Multi-Hybrid, but at this point very little research has been done on how to best combine multiple components in a single branch predictor. The Multi-Hybrid already performs better than regular hybrid predictors. With more research on how to best extend the Multi-Hybrid to predict several branches per cycle and more research on the interaction between component predictors, the Multi-Hybrid could evolve into an even better predictor, capable of fully meeting the demands of this aggressive processor design.

Implementation issues and indirect jumps

The need for a large predictor brings up several implementation issues. Although the proposed predictor can give very accurate predictions, accessing a predictor of this size in a single cycle is not reasonable. However, a single-cycle prediction is needed to allow the front end to work at peak capabilities.

Some of the component predictors in the Multi-Hybrid, such as the per-address two-level predictor, are capable of making their prediction ahead of time. This produces a temporary prediction that will be used to redirect the front end, being compared to the full prediction of the Multi-Hybrid a few cycles later. Such predictions have a slightly lower accuracy but can be accessed quickly. If the predictions disagree, the front end will be flushed, resulting in only a minor penalty that will be hidden as long as the reservation stations hold instructions waiting to be executed.

Indirect branches are another area where the additional hardware budget can help overcome a major problem. The indirect branch predictor uses the same concept applied in the two-level branch predictor. However, instead of a table of two-bit counters, the indirect branch predictor uses the branch history to index to a table of branch targets.[10] These targets are then used to predict the indirect jump, in much the same way as a two-level predictor predicts the direction of a branch. Even at fairly low implementation costs, this mechanism reduces by about half the number of mispredictions due to indirect jumps.

Finally, even though a dynamic predictor can do quite well on its own, a processor like this needs some help from the compiler and the application programmer. Certain branches will always remain inherently hard to predict, regardless of the amount of logic thrown at the problem. When possible, these branches should be predicated; or dynamic predictor hint instructions may be needed to improve the chance of correct predictions.

Authorized licensed use limited to: Edinburgh University. Downloaded on January 26, 2010 at 10:30 from IEEE Xplore. Restrictions apply.

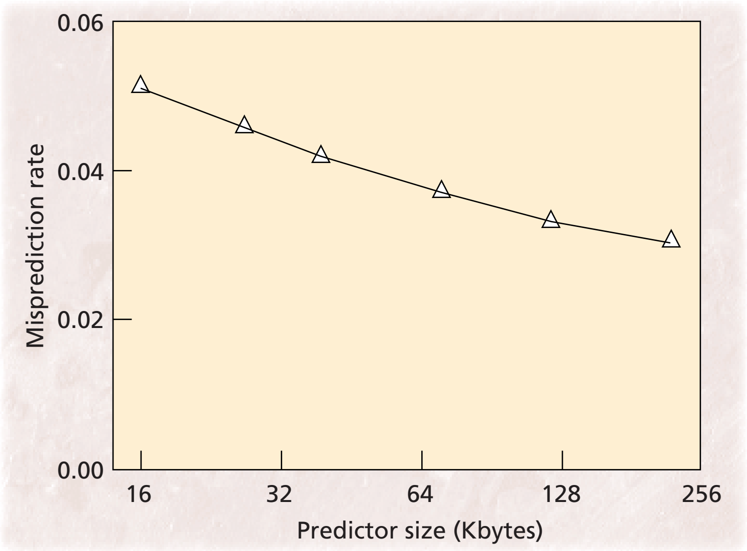

Figure 3. Misprediction rate of Multi-Hybrid for SPECint95.

THE MEMORY SYSTEM

Processor cycle time is decreasing faster than cache and memory access times. If this trend continues and a billion transistors are fabricated on a die, misses serviced by main memory will cause stalls of hundreds of cycles. Out-of-order execution tolerates data cache and memory latencies better than in-order execution, but out-of-order execution is not enough.

Instruction supply and out-of-order fetch

To deal with trace cache misses, high-performance processors will employ out-of-order fetch. An in-order fetch processor, upon encountering a trace cache miss, waits until the miss is serviced before fetching any new segments. But an out-of-order fetch processor temporarily ignores the segment associated with the miss, attempting to fetch, decode, and issue the segments that follow it. After the miss has been serviced, the processor decodes and issues the ignored segment.

Higher performance can be achieved by fetching instructions that—in terms of a dynamic instruction trace—appear after a mispredicted branch, but are not control-dependent upon that branch. In the event of a mispredict, only instructions control-dependent on the mispredicted branch are discarded.

Out-of-order fetch provides a way to fetch control-independent instructions. When a processor fetches a branch, it must predict which direction the branch takes in order to determine which instructions to fetch next. Because the branch condition is usually not known at the time of the branch's fetch, the processor must guess its direction and then speculatively fetch the instructions that follow. If the prediction is correct, no branch penalty incurs. However, if the prediction is incorrect, the processor pays the same penalty as if it had waited for the branch to be resolved. Current branch prediction technology is capable of prediction accuracies of 97 percent for conditional branches, but the remaining mispredictions still incur a large performance penalty.

An out-of-order fetch mechanism can reduce this penalty by not fetching the blocks that immediately follow a hard-to-predict branch until either an accurate prediction can be made for the branch or the branch is resolved. During this period, the processor fetches, decodes, and issues the instructions that begin at the merge point in the control-flow graph of the paths that follow the branch. Because these instructions are control-independent of the branch, they are guaranteed to be on the program's correct path regardless of which direction the branch takes. The processor will continue to fetch, decode, and issue along this path until either an accurate prediction can be made or the branch is resolved. At this point, it will return to the branch and begin fetching there. Upon reaching the merge point, the processor will jump past the instructions that it has already fetched, decoded, and issued and continue fetching, decoding, and issuing from the point at which it left off.

Data supply

A 16-wide-issue processor will need to execute about eight loads/stores per cycle. The primary design goal of the data-cache hierarchy is to provide the necessary bandwidth to support eight loads/stores per cycle. Cache latency does not become an issue unless this bandwidth requirement can be met. A small, multiported, first-level data cache will provide most of the bandwidth required to execute these loads and stores. The few loads and stores that miss in the first-level data cache will be routed to a bigger, second-level data cache. (Because few loads and stores are routed to the second-level data cache, the number of ports it requires is much less than that required for the first-level data cache.) The size of a single, monolithic, multiported, first-level data cache would likely be so large that it would jeopardize the cycle time. Because of this, we expect the first-level data cache to be replicated to provide the required ports. (The proposed Digital 21264 uses this technique for its register file: It replicates the register file to provide a large number of ports without jeopardizing the cycle time.[11])

"If you don't know, then predict" is quickly becoming a basic tenet of computer architecture. It is behind instruction and data prefetching, which are used in cache design to build set-associative caches with the access times of direct-mapped caches.[11] We expect prefetching and set prediction to become the norm in processor design. We also expect the other forms of prediction to become prevalent. To enhance performance, processors will predict the addresses of loads, allowing loads to be executed before the computation of operands needed for their address calculation. Also, processors will predict dependencies between loads and stores, allowing them to predict that a load is always, or almost always, dependent on some older store.[12] The store will forward its data directly to the

Authorized licensed use limited to: Edinburgh University. Downloaded on January 26, 2010 at 10:30 from IEEE Xplore. Restrictions apply.

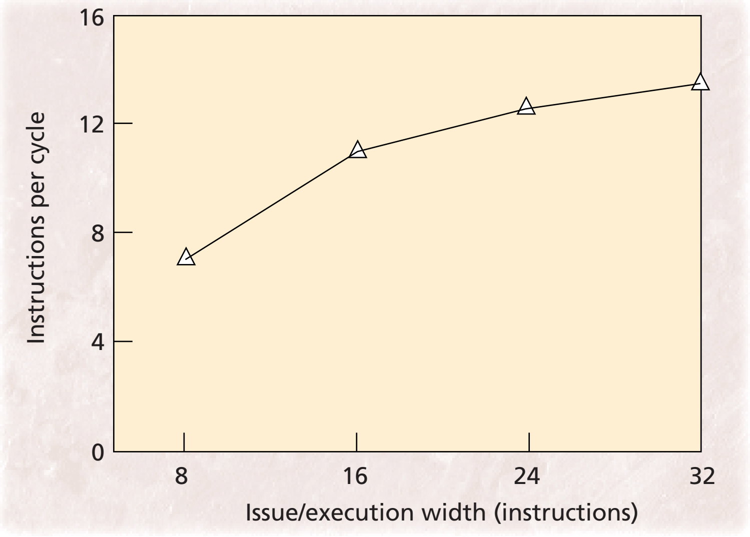

Figure 4. Available parallelism with a fixed instruction window.

load and the instructions that are dependent on the result of the load, even if the addresses of both the store and the load are unknown.

THE EXECUTION CORE

To achieve high performance, the execution core must accomplish a large amount of work every cycle. If the fetch engine is providing 16 to 32 instructions per cycle, then the execution core must consume instructions just as rapidly. To avoid unnecessary delays due to false dependencies, logical registers must be renamed. To compensate for the delays imposed by the true data dependencies, instructions must be executed out of order, allowing useful work to continue while earlier instructions are waiting for their source operands.

We envision an execution core comprising 24 to 48 functional units supplied with instructions from large reservation stations and having a total storage capacity of 2,000 or more instructions. This large execution window will ensure that there will be a steady supply of instructions ready for execution. Such a design does not come without difficulties, most of them involving communication. The communication traffic in the execution core falls into three categories: communicating instructions to the proper functional unit (routing), communicating the availability of functional units and operands to instructions (scheduling), and communicating the operand values to the functional units (data forwarding).

As the width of the execution core increases, so does the distance from the output of one functional unit to the input of another. To avoid long propagation delays, the functional units will be partitioned into clusters of three to five units. Each cluster will maintain an individual register file. Data forwarding within these clusters will require a single cycle, but communication between clusters will require multiple cycles. To ensure that the latency associated with cross-cluster communication is rarely incurred, the majority of values produced must be consumed by instructions executed within the same cluster.

The routing of instructions to the proper functional unit is done as the instructions enter the instruction window. Each functional unit has its own reservation station. Although other reservation station designs are possible, this technique offers an advantage: The fill unit has the time and opportunity to analyze the instructions and preroute them to the appropriate functional units. Instructions issued from the trace cache will be ordered to reduce the cross-cluster communication penalty.

The difficulties of scheduling instructions from a large instruction window arise from the need to examine each instruction in the pool to determine if it is ready for execution. To mitigate this, scheduling will be done in stages. The reservation stations will be partitioned so that the final stage of scheduling is done from only a subsection of the entire window. This technique effectively creates an on-deck portion of the reservation stations. If an instruction is nearly ready to execute, it will be placed on deck. The final stage of the scheduling logic will consider only instructions from this portion of the reservation stations.

Since it will be possible to design such an aggressive execution core, the question remains whether the applications have enough instruction-level parallelism (ILP) to warrant such a design. Figure 4 shows the average performance of the SPECint95 benchmarks achieved with a fixed window size of 2K instructions while varying the issue and execution widths. Because we are interested in the ILP available within the program, perfect caches and perfect branch prediction have been simulated, although all memory dependencies are honored. The figure shows that wider issue and execution widths allow the parallelism within the program to be exploited. However, the curve flattens out toward the high end of the spectrum. Aggressive speculation techniques, such as address prediction, should push the knee of the curve higher. Furthermore, both programming and compiler techniques will improve, exposing more parallelism to the hardware.

We have argued that the highest performance computing system (when one billion transistor chips are available) will contain on each processor chip a single processor. We have identified structures that will be necessary to make that uniprocessor perform, and showed the performance obtainable from those structures. All this makes sense, however, only if CAD tools can be improved to design such chips and only if algorithms and compilers can be redesigned to take advantage of such powerful dynamically scheduled engines.

The highest performance computing system will be a multiprocessor made up of very powerful single-chip uniprocessors. They will issue and execute 16 (or 32) instructions practically every cycle, with nearly 100

Authorized licensed use limited to: Edinburgh University. Downloaded on January 26, 2010 at 10:30 from IEEE Xplore. Restrictions apply.

percent branch prediction accuracy and a very fine cycle time made possible by very deep pipelines. ❖

Acknowledgments

The HPS execution model—aggressive microarchitecture in support of high-performance single-instruction streams—started in 1984 and has benefited enormously from the students who have worked on it since its inception. We particularly acknowledge the contributions of Wen-mei Hwu, Steve Melvin, Mike Shebanow, Tse-Yu Yeh, Mike Butler, Eric Hao, and Po-Yung Chang, whose PhD research greatly strengthened our understanding of the HPS paradigm. Also, we gratefully acknowledge the financial support of our industrial sponsors—in particular, Intel, NCR, and Motorola.

References

1. B.J. Smith, "A Pipelined Shared Resource MIMD Computer," *Proc. 1978 Int'l Conf. Parallel Processing,* IEEE Press, New York, 1978, pp. 6–8.
2. Y.N. Patt, W. Hwu, and M. Shebanow, "HPS, a New Microarchitecture: Rational and Introduction," *Proc. 18th Ann. ACM/IEEE Int'l Symp. Microarchitecture,* IEEE CS Press, Los Alamitos, Calif., 1985, pp. 103–107.
3. Y.N. Patt et al., "Critical Issues Regarding HPS, a High-Performance Microarchitecture," *Proc. 18th Ann. ACM/IEEE Int'l Symp. on Microarchitecture,* IEEE CS Press, Los Alamitos, Calif., 1985, pp. 109–116.
4. S.W. Melvin and Y.N. Patt, "Performance Benefits of Large Execution Atomic Units in Dynamically Scheduled Machines," *Proc. 1989 Int'l Conf. on Supercomputing,* ACM Press, New York, 1989, pp. 427–432.
5. E. Rotenberg, S. Bennett, and J.E. Smith, "Trace Cache: A Low-Latency Approach to High-Bandwidth Instruction Fetching," *Proc. 29th Ann. ACM/IEEE Int'l Symp. Microarchitecture,* IEEE CS Press, Los Alamitos, Calif., 1996, pp. 24–34.
6. S.J. Patel, D.H. Friendly, and Y.N. Patt, *Critical Issues Regarding the Trace Cache Fetch Mechanism,* Tech. Report CSE-TR-335-97, Univ. of Michigan, Ann Arbor, 1997.
7. A. Peleg and U. Weiser, *Dynamic Flow Instruction Cache Memory Organized Around Trace Segments Independent of Virtual Address Line,* US Patent 5,381,533, US Patent and Trademark Office, Washington, D.C., 1994.
8. S. McFarling, *Combining Branch Predictors,* Tech. Report TN-36, Digital Western Research Laboratory, Digital Equipment Corp., Palo Alto, Calif., 1993.
9. M. Evers, P.-Y. Chang, and Y.N. Patt, "Using Hybrid Branch Predictors to Improve Branch Prediction Accuracy in the Presence of Context Switches," *Proc. 23rd Ann. Int'l Symp. Computer Architecture,* ACM Press, New York, 1996, pp. 3–11.
10. P.-Y. Chang, E. Hao, and Y.N. Patt, "Predicting Indirect Jumps Using a Target Cache," *Proc. 24th Ann. Int'l Symp. Computer Architecture,* ACM Press, New York, 1997, pp. 274–283.
11. B.A. Gieseke et al., "A 600 MHz Superscalar RISC Microprocessor with Out-of-Order Execution," *1997 IEEE Int'l Solid-State Circuits Conf. Digest of Technical Papers,* IEEE Press, New York, 1997, pp. 176–178.
12. A. Moshovos et al., "Dynamic Speculation and Synchronization of Data Dependencies," *Proc. 24th Ann. Int'l Symp. Computer Architecture,* ACM Press, New York, 1997, pp. 181–193.

***Yale N. Patt** is a professor of electrical engineering and computer science at the University of Michigan. His research interests include high-performance computer architecture, processor design, and computer systems implementation. Patt received a BS in electrical engineering from Northeastern University and an MS and a PhD in electrical engineering from Stanford University. He is the recipient of the 1995 IEEE Emanuel R. Piore Award and the 1996 ACM/IEEE Eckert-Mauchly Award and a fellow of the IEEE.*

***Sanjay J. Patel** is a PhD candidate in computer science and engineering at the University of Michigan. He is investigating techniques for high-bandwidth instruction supply for processors in the era of 100 million and one billion transistors. His research interests include trace caches, branch prediction, memory bandwidth issues, and the performance simulation of microarchitectures. He received an MS in computer engineering from the University of Michigan.*

***Marius Evers** is a PhD candidate in computer architecture at the University of Michigan. His research focus is branch prediction and instruction delivery. He received a BS and an MS in computer engineering from the University of Michigan.*

***Daniel H. Friendly** is a PhD candidate in computer architecture at the University of Michigan. His research interests include high-bandwidth fetch mechanisms, superscalar architectures, and speculative execution techniques. He received a BA in American social issues from Macalester College and an MS in computer science from the University of Michigan.*

***Jared Stark** is a PhD candidate in computer science and engineering at the University of Michigan. His research interest is computer architecture. He received a BS in electrical engineering and an MS in computer science and engineering from the University of Michigan.*

Contact Patt at patt@eecs.umich.edu; Patel at sanjayp@eecs.umich.edu; Evers at olaf@eecs.umich.edu; Friendly at ites@eecs.umich.edu; and Stark at starkj@eecs.umich.edu.

Authorized licensed use limited to: Edinburgh University. Downloaded on January 26, 2010 at 10:30 from IEEE Xplore. Restrictions apply.