Superspeculative Microarchitecture for Beyond AD 2000

Employing a broad spectrum of superspeculative techniques can achieve significant performance increases over today's top-of-the-line microprocessors. The experimental, superspeculative microarchitecture Superflow has a potential performance of 9.0 instructions per cycle and realizable performance of 7.3 IPC for the SPEC95 integer suite, without requiring recompilation or changes to the instruction set architecture.

Mikko H. Lipasti

John Paul Shen

Carnegie
Mellon
University

In its brief lifetime of 26 years, the microprocessor has achieved a total performance growth of 10,000 times thanks to technology improvements and microarchitecture innovations. Transistor count and clock frequency have increased by an order of magnitude in each of the first two decades of microprocessors; transistor count increased from 10,000 to 100,000 in the 1970s and up to 1 million in the 1980s, while clock frequency increased from 200 KHz to 2 MHz in the 1970s and up to 20 MHz in the 1980s. This incredible technology trend has continued: Since 1990, both transistor count and clock frequency have already achieved an increase of 20 to 30 times. During the 1980s, sustained instructions per cycle also increased by almost an order of magnitude, from roughly 0.1 to 0.9. IPC is a measure of the instruction-level parallelism or instruction throughput achieved by the concurrent processing of multiple machine instructions. In the 1990s, IPC improvement is struggling and may not triple by 1999. New microarchitecture innovations are needed.

Current top-of-the-line microprocessors are four-instruction-wide superscalar machines; that is, they can fetch and complete up to four instructions in a single machine cycle. Such machines use pipelined functional units, aggressive branch prediction, dynamic register renaming, and out-of-order execution of instructions to maximize parallelism and tolerate memory latency. State-of-the-art processors include the Digital Equipment Alpha 21264, Silicon Graphics MIPS/R10000, IBM/Motorola PowerPC 604, and Intel Pentium Pro. Even with such elaborate microarchitectures, against a potential 4 IPC, these machines typically achieve only about 0.5 to 1.5 sustained IPC for real-world programs.

Worse yet, most studies indicate that machine efficiency drops even lower as we extrapolate to wider machines. One recent study indicated that although a hypothetical 2-instruction-wide machine achieves IPC in the range of 0.65 to 1.40, a similar, hypothetical, 6-instruction-wide machine will achieve only 1.2 to 2.3 IPC.[1] Such data imply that the current superscalar paradigm is running into rapidly diminishing returns on performance.

POTENTIAL NEW PARADIGMS

Future billion-transistor chips will inevitably implement machines that are much wider (issue more than four instructions at once) and deeper (have longer pipelines). The question is, how do we harvest additional parallelism proportional to increased machine resources? Several approaches have vocal advocates, each with valid reasons; they are

- reconfigurable parallel computing engines;
- specialized, very long instruction word (VLIW) machines;
- wide, simultaneous multithreaded (SMT) uniprocessors;
- single-chip multiprocessors (CMP);
- memory-centric computing engines (such as IRAM);
- very wide conventional superscalars; and
- wide superspeculative processors.

Two overriding concerns will determine which possibility might have the most commercial impact: performance scalability and software migration complexity. The approach that succeeds will have to deliver enough performance improvement to justify the hardware resources expended. It must also effectively deal with the tremendous effort and expense required to create or migrate a critical mass of useful applications. Computing history's two most economically successful instruction set architectures—the IBM S/360 and the Intel x86—have reaped rewards for paying meticulous attention to software cost and code compatibility.

Reconfigurable computers

Researchers have proposed reconfigurable computers that employ large arrays of highly programmable build-

Super-speculative architectures have the potential of sustaining IO IPC for nonnumeric applications.

ing blocks. Typical examples are complex programmable logic devices and field-programmable gate arrays. These devices rely on powerful software tools to map applications to the reconfigurable hardware. The intent is to construct powerful, specialized computing engines at a relatively low cost with very short turnaround time. This approach's performance scalability has yet to be demonstrated beyond a few specialized applications. Moreover, the huge investment required to leverage the latest and best fabrication technology may not be economically justified by the niche-market volume for reconfigurable systems. Finally, software tools for automating application-to-hardware mapping must incorporate the latest code compilation and optimization techniques as well as yet-to-be-developed hardware synthesis technologies. This is a very tall order for the software.

VLIW machines

Specialized VLIW machines already exist for multimedia applications. These statically controlled, wide machines contain numerous functional units with highly deterministic behavior, which permits tremendous computing throughput on specialized applications. They provide performance scalability by packing more into a single instruction. VLIW machines rely on powerful compilers to detect and resolve inter-instruction dependences in software. This keeps the hardware design clean and fast. Such machines usually require application recompilation, so retargetable compilers are essential. Such compilers have been long in coming and are still not widely available. Furthermore, the static nature of VLIWs makes them inherently incompatible with dynamic variations in parallelism or latency, both of which are caused by aggressive memory subsystems and speculative-execution techniques.

SMT uniprocessors

Simultaneous multithreaded processors are superscalar uniprocessors that support multiple machine contexts and execute multiple instruction streams simultaneously. They do so to increase a multiprogrammed workload's throughput or reduce a multithreaded program's latency. Performance scalability depends on finding enough thread parallelism, a task left to software. Developing multithreaded applications is challenging due to the extreme difficulty of debugging multithreaded programs and the lack of automatic thread-partitioning compilers. Therefore, we view SMT primarily as a technique for improving throughput in multiprogrammed workloads.

Single-chip multiprocessors

Our view of the single-chip multiprocessor approach is similar. CMPs will inevitably be used for improving throughput under multiprogrammed workloads.

However, their utility in improving single-program performance has thus far been restricted to numerical applications that contain easily parallelized loops. Limited processor interconnect and synchronization overhead will throttle performance scalability, particularly for more generalized applications with interthread dependences. Significant development is required before parallelizing compilers will provide performance gains for generalized applications on CMPs. Unless such software technology becomes widely available, CMP, like SMT, is destined to remain a technique for improving throughput but not latency. Furthermore, without a significant multithreaded-application base there may not be a mainstream demand for single-chip SMT processor and CMP implementations. Without mainstream market demand, these implementations are not economically justifiable.

Memory-centric engines

Proposing a memory-centric view (as opposed to the traditional CPU-centric view) to computer system design has become quite popular. The potential integration of dense DRAM technology with fast logic technology on the same chip (intelligent RAM) is certainly of technological interest. However, it is unclear that this integration inspires any truly new architecture paradigms. We see it as more of a technology/implementation issue that enables shorter latency and more density and bandwidth at the upper levels of the memory hierarchy. There is also the issue of software. Compilation tools for array-structured, message-passing multicomputers have been in development for over a decade and are still not widely available.

Wide conventional superscalars

On the other hand, scaling up current superscalars to process more instructions at a time, though attractive from a software cost perspective, doesn't seem promising either. Performance scalability is limited because current microarchitectural techniques for extracting instruction-level parallelism are returning significantly less performance improvement on wider machines. Incremental improvements will result in only marginal performance gains. Limit studies have shown that even when all control and structural hazards are removed, single-thread performance is still severely limited by true data dependences between instructions that produce results and instructions that use these results as their source operands. Due to such data dependences, the producer and consumer instructions must necessarily be serialized. Enforcing these serializations leads to the classical dataflow limit for program performance: Given unlimited machine resources, a program cannot execute any faster than the execution of the longest dependence chain induced by the program's true data dependences.

Figure 1. Superspeculation's performance potential chart showing Sustained IPC for various SPEC95 integer benchmarks (go, m88ksim, gcc, compress, li, ijpeg, perl, vortex, Harmonic mean). Legend: Conventional superscalar (baseline performance); Additional performance provided by superspeculation techniques.
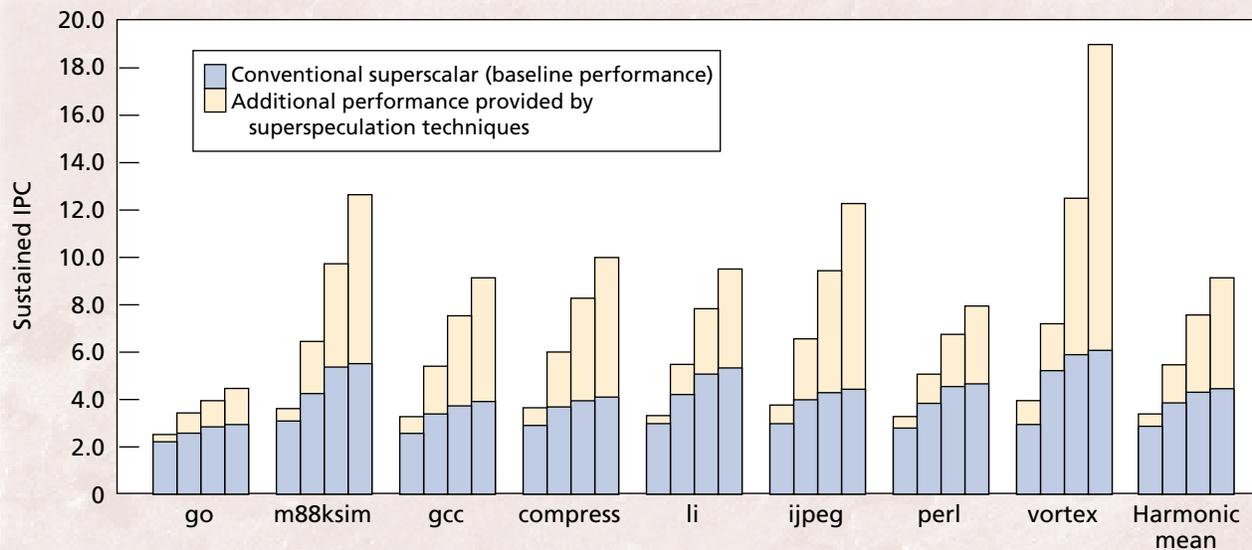
Superspeculative processors

We believe it is due to the seeming inability to get beyond the dataflow limit to harvest more instruction-level parallelism that many researchers are advocating one of the approaches just discussed, but all these approaches require a dramatic, expensive paradigm shift from sequential programming to an explicitly parallel model. Superspeculative processors, on the other hand, overcome the dataflow limit without sacrificing code compatibility. They do so by aggressively speculating past true data dependences and harvesting additional parallelism in places where none was believed to exist. The basis for the superspeculative approach is that producer instructions generate many highly predictable values in real programs. Consumer instructions can thus frequently and successfully speculate on their source operand values and begin execution without results from the producer instructions. Consequently, a superspeculative processor can remove the serialization constraints between producer and consumer instructions, enabling program performance to potentially exceed the classical dataflow limit.

Figure 1 summarizes the performance obtainable by scaling a conventional superscalar design—one that employs all the latest techniques in branch prediction and out-of-order execution—from today's issue width of four instructions per cycle up to eight, 16, and 32 instructions per cycle. The sustained IPC attainable for the SPEC95 integer benchmarks shown levels off quickly around width eight or sixteen.

In contrast, the stacked bars show the additional performance attainable by a processor that employs superspeculative techniques. These techniques frequently more than double the attainable performance and provide ample justification for continuing to devote processor implementation resources to microarchitectures that are compatible with current ISAs and do not require a massive software investment.

Superspeculative microarchitectures have the potential of sustaining close to 10 IPC for nonnumerical pro-

grams without requiring advanced compilation support. Superspeculation aggressively continues the statistical approach that emerged during the 1980s: designers optimizing machine performance for statistically common cases instead of the less likely worst case. We generalize the current control speculation (in the form of branch prediction) to include various forms of data speculation.[2] With this generalization, the processor can circumvent both the control flow and dataflow constraints of a program via aggressive speculation. Such speculation often pays off because of the predictable behavior of real programs. The theoretical basis for superspeculative microarchitectures rests on the weak dependence model,[3] which defers the detection of and relaxes the enforcement of control flow and dataflow dependences between machine instructions.

Strong-dependence model. The implied total instruction ordering of a sequential program is an overspecification and need not be rigorously enforced to meet the requirements of semantically correct execution. The actual program semantics and inter-instruction dependences are specified by the control flow graph, which specifies the possible paths for traversing the basic blocks of instructions in the program, and the dataflow graph, which specifies the true data dependences between producer and consumer instructions. As long as the processor does not violate the serialization constraints imposed by these graphs, it can overlap and reorder the execution of instructions. This achieves better performance by avoiding the enforcement of implied but unnecessary precedences. However, the processor must still enforce true inter-instruction dependences. To date, most machines enforce such dependences in a rigorous fashion that adheres to two requirements:

- Dependences are determined in an absolute and exact way; that is, two instructions are identified as either dependent or independent, and when in doubt, dependences are pessimistically assumed to exist.

- Dependences are enforced throughout instruction execution; that is, the dependences are never allowed to be violated and are enforced during instruction processing.

Such a traditional and conservative approach is called the strong-dependence model for program execution. This traditional model is overly rigorous and unnecessarily restricts available parallelism.

Weak-dependence model. Instead, we propose the weak-dependence model for superspeculative processors. This model specifies that

- Dependences need not be determined exactly or assumed pessimistically, but can instead be optimistically approximated or even temporarily ignored.
- Dependences can be temporarily violated during instruction execution as long as recovery can be performed prior to affecting the permanent machine state.

The weak-dependence model's advantage is that the machine can process instructions without having the program semantics completely determined, as specified by control flow and dataflow graphs. Furthermore, the machine can now speculate aggressively and temporarily violate the dependences as long as corrective measures are in place to recover from misspeculation. If a significant percentage of speculations are correct, the machine can effectively exceed the performance limit imposed by the traditional, strong-dependence model.

Similar in concept to branch prediction's implementation in current processors, superspeculation uses two interacting engines. The front-end engine assumes the weak-dependence model and is highly speculative, predicting instructions to aggressively speculate past them. When predictions are correct, these speculative instructions will effectively have skipped over certain stages of instruction execution. The back-end engine still uses the strong-dependence model to validate the speculations, recover from misspeculation, and provide history and guidance information to the speculative engine. By taking this approach, processors can harvest an unprecedented level of instruction-level parallelism. The edges of the dataflow graph that represent inter-instruction dependences are now enforced and become part of the critical dataflow path only when misspeculations occur.

A superspeculative machine. As Figure 2 shows, instruction execution can be divided into four logical stages, each taking one or more machine cycles.

- *Fetch*. The processor retrieves instructions from cache or main memory.
- *Decode*. The processor decodes instructions,

renames their operands, and detects inter-instruction dependences.
- *Execute*. Instructions wait until their operands are available, then allocate functional units, execute according to their prescribed semantics, and forward their results to subsequent dependent instructions.
- *Commit*. Instructions are allowed to write back their results into the architected registers in program order.

Figure 2 also identifies the three key parameters that a superspeculative microarchitecture must maximize:

- instruction flow, the rate at which useful instructions are fetched, decoded, and dispatched to the execution core;
- register dataflow, the rate at which results are produced and register values become available; and
- memory dataflow, the rate at which data values are stored and retrieved from data memory.

These three flows roughly correspond to the processing of branch, ALU, and load/store instructions. In a superspeculative microarchitecture, aggressive speculative techniques are employed to accelerate the processing of all these instruction types.

SUPERFLOW TECHNIQUES

Our proposed approach to superspeculative microarchitecture is called Superflow. We've implemented a prototype to collect the data reported here, which indicates its potential, but the detailed design effort is ongoing. Superflow employs a wide range of speculative techniques to improve the throughput of instruction flow, register dataflow, and memory dataflow beyond traditional limits. Hence the name Superflow.

Instruction flow techniques

With these techniques, the processor attempts to supply as many useful instructions as it can to the execution core. Instruction flow is a threefold problem: conditional-branch throughput, taken-branch throughput, and misprediction latency.

Conditional branches. To provide adequate conditional-branch throughput, the processor must accurately predict the outcomes and targets of multiple conditional branches in every cycle. We combined earlier work on predicting multiple branches in every cycle[4,5] and predicting branches accurately[6] in a two-phase branch predictor. This predictor employs a local (per static branch) and a global branch history to predict multiple branches in each cycle. During the first stage (fetch), global knowledge is used to predict multiple branches and generate a fetch address for

Figure 2. Superspeculative machine overview. A Superflow microarchitecture employs a broad spectrum of speculative techniques to maximize overall instruction throughput beyond traditional limits by maximizing the throughputs of instruction flow, register dataflow, and memory dataflow.
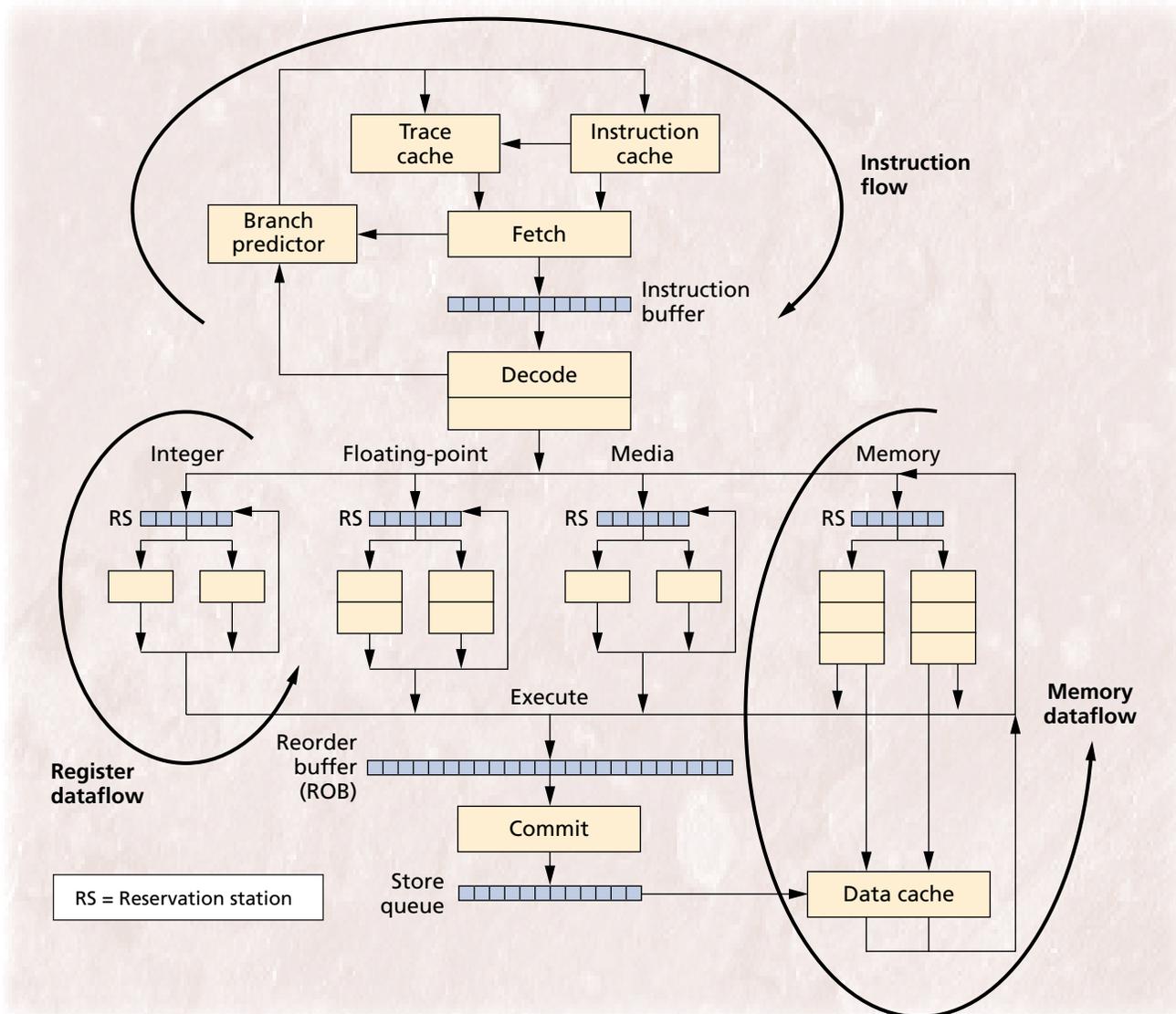
the next cycle.[5] During the second stage (decode), the earlier predictions are checked via an advanced gshare predictor,[6] which combines local and global knowledge to generate an accurate prediction. Hence, many mispredictions made in the first phase are corrected with a latency of only one cycle.

Taken branches. The fetch unit must be able to correctly process more than one taken branch per cycle, which involves predicting each branch's direction and target, and also fetching, merging, and aligning instructions from each branch target.

To reduce complexity, the Superflow fetch engine uses an interesting, recently proposed approach called a trace cache.[5] A trace cache is a history-based fetch mechanism that stores dynamic-instruction traces in a cache indexed by fetch address and branch outcome. Whenever it finds a suitable trace, it dispatches instructions from the trace cache rather than sequential instructions from the instruction cache. Since a dynamic sequence of instructions in the trace cache can contain multiple taken branches but is stored sequentially, there is no need to fetch from multiple

targets. This eliminates the need for a multiported instruction cache or complex merge/align logic in the critical path.

Misprediction latency. Finally, the processor must minimize the latency of correctly resolving mispredicted branches. To do so, we employ a set of aggressive data-speculative techniques in the execution core; the register and memory dataflow sections describe them. These techniques enable branches to execute earlier than they would otherwise and produced significant performance gains by significantly reducing misprediction penalties.

Register dataflow techniques

These techniques facilitate fast and efficient processing of ALU instructions. The execution core strives for two fundamental goals to increase instruction throughput. It must

- efficiently detect and resolve inter-instruction dependences and
- eliminate or bypass as many dependences as pos-

Table 1. Estimated transistor cost of Superflow.

Resource	Description	Transistor count (millions)
CPU core logic	(32 instructions issued per cycle / 4 instructions issued per cycle)^2 × 2 million transistors in a PowerPC 604, which issues four instructions per cycle	128.0
Value prediction table	32 Kbytes × 8 bits/byte × 6 transistors/SRAM bit	1.6
Classification table	8K entries × 2 bits/entry × 6 transistors/SRAM bit	0.1
Dependence prediction table	8K entries × 7 bits/entry × 64 ports × 6 transistors/SRAM bit	22.0
Alias prediction table	8K entries × 7 bits/entry × 32 ports × 6 transistors/SRAM bit	11.0
Pattern history tables	64K entries × 2 bits/entry × 2 tables × 6 transistors/SRAM bit	1.6
Trace cache	64 Kbytes (estimated size) × 8 bits/byte × 6 transistors/SRAM bit	3.1
Level 1 instruction cache	64 Kbytes × 8 bits/byte x 6 transistors/SRAM bit	3.1
Level 1 data cache	64 Kbytes × 4 ports × 8 bits/byte × 6 transistors/SRAM bit	12.6
Processor core total		**183.1**
Level 2 unified cache	16 Mbytes × 8 bits/byte × 6 transistors/SRAM bit (approximately)	805.3
Grand total		**988.4**

sible to expose more parallelism between instructions.

Detecting control and data dependences among multiple active instructions is an inherently sequential task that becomes expensive combinatorially as the number of concurrently active instructions increases. Furthermore, multiple-instruction dispatch is difficult to implement and has an adverse impact on cycle time because all instructions in a dispatch group must be simultaneously cross-checked. To avoid impacting cycle time, dependence checking and dispatch can be pipelined into multiple stages. Unfortunately, a deeper pipeline, especially in the decode portion, results in a greater performance penalty. However, this penalty can be overcome with dependence prediction, a speculative technique that can frequently short-circuit pipelined multicycle decode. It does so by predicting the dependence relationships between instructions and speculatively allowing instructions that are predicted to be data ready to execute in parallel with exact dependence checking.[7] We discovered that such dependence relationships between instructions are highly predictable in real programs. The Superflow execution core adopts this technique to help overcome the latency cost of pipelined decode/dispatch and facilitate the implementation of wide-dispatch processors.

Superflow employs conventional, well-understood techniques, such as register and memory renaming, to eliminate false dependences between instructions. However, it advances well beyond the limits imposed by true dependences by employing source operand value prediction[7] to eliminate true data dependences between instructions. This technique uses dynamic-value history information, stored per static program instruction, to predict future values of that instruction's source operands. Superflow improves the accuracy of source operand value prediction beyond previously reported levels by extending it to include *value stride prediction*. In value stride prediction, a dynamic hardware mechanism detects constant, incremental increases in operand values (strides) and uses them to predict future values.

Memory dataflow techniques

These techniques minimize average memory latency and provide adequate memory bandwidth for supporting a high-performance superspeculative processor core. To reduce average memory latency, we incorporate the prediction of load values,[8,9] addresses, and aliases into the execution core. For adequate load instruction throughput, we introduce load stream partitioning, a divide-and-conquer strategy for reducing the cost and complexity of a high-bandwidth memory system.

Memory latency is a severe bottleneck to processor performance, and three factors cause it:

- address generation interlocks, which delay the initiation of a fetch from memory because the address is unknown;
- the latency of accessing the storage device; and
- queuing delays caused by contention for shared resources in the memory subsystem.

Speculative prediction of load addresses can eliminate many address generation interlocks. Previous research on address prediction is largely subsumed by source operand value prediction, which incorporates stride prediction for detecting and predicting constant-stride memory addresses. Address generation interlocks also cause a secondary problem. When load addresses are unknown, it is impossible to detect the existence of an alias with an earlier outstanding store. To alleviate this problem, we propose alias prediction, a logical extension to the register dependence prediction technique described earlier.[7]

Rather than predict the dependence distance to a preceding register write, we predict the dependence distance to a preceding store. (Andreas Moshovos and his colleagues described a similar technique.[10]) The predicted distance is then used to obtain the load value from that offset in the processor's store queue, which holds outstanding stores. For this speculative forwarding to occur, neither the load nor the store need to have their addresses available yet, allowing the bypassing of address generation entirely.

The storage device's latency can be folded away (effectively eliminated) by performing load value prediction, which uses a per-static-load-value history to predict future values. This technique can frequently predict the value to be loaded when the load instruction is dispatched, in effect implementing a zero-cycle load.[8]

At first glance, providing adequate memory bandwidth to support Superflow's performance goals appears daunting. Since roughly 40 percent of the 10-instruction-per-cycle throughput consists of loads and stores, the memory subsystem must provide an average bandwidth of four references per cycle. Furthermore, the peak bandwidth required to prevent excessive queuing delays is even higher. However, several factors and techniques relieve this difficult problem.

First, recent discoveries indicate that the reuse distances of many stored values (measured in execution cycles) are very short.[11] In fact, our simulations show that many stores do not even make it out of the store queue before their values are needed again. Hence, the loads retrieving these values do not require a cache port. Second, a technique called constant promotion[8] can be used to eliminate the actual memory accesses normally performed by load instructions. Constant promotion provides a hardware mechanism that guarantees that the value generated via load value prediction is correct and need not be checked against an actual memory reference. It does so by keeping the load value prediction table coherent with main memory. This technique monitors all intervening stores between an update and a lookup of the load value prediction table, invalidating entries modified by such stores.

These two observations lead us to introduce the notion of *load stream partitioning*, which simply partitions loads into multiple streams based on their behavior and sends them to disjoint, specialized functional units for processing. This approach facilitates the implementation of high-bandwidth memory systems by eliminating the need for large, centrally located, and extremely multiported data caches. Experimental evidence suggests that a significant portion of the load stream can be diverted to these specialized units for processing.[3,8]

PERFORMANCE POTENTIAL

To evaluate superspeculation's performance potential, we simulated the performance of a Superflow processor with a

- fetch width of 32;
- a 128-entry reorder buffer;
- 64-Kbyte, 4-way, set-associative data and instruction caches with 10-cycle miss delay to a perfect, pipelined second-level cache; and
- a 128-entry, fully associative store queue.

Table 1 shows a crude estimate of the transistors needed for a naive implementation of such a processor. We estimate that CPU core logic will increase as the square of the issue width, consuming 128 million transistors relative to the two million core logic transistors in the PowerPC 604, which issues four instructions per cycle. To reduce design complexity, we expect that the functional units in the execution core will be clustered in some manner similar to earlier proposals.[12] Relative to the CPU core, the superspeculative prediction structures, including those for branch prediction, consume roughly 36 million transistors, while level-1 caches consume about 19 million transistors. A superspeculative processor core (including the level-1 caches) requires less than 200 million transistors. Assuming a total budget of one billion transistors, the chip still has room for a fast, 16-Mbyte, level-2 cache, which should be more than adequate to capture the working sets of most

Figure 3. Superflow performance; sustained IPC for a fetch width of 32, a reorder buffer size of 128, and a 128-entry store queue for various memory configurations. The leftmost bar shows IPC for a perfect cache with unlimited ports. The second bar shows IPC for a 64-Kbyte data cache with unlimited ports. The third, fourth, and fifth bars show the previous configuration, but with eight, four, and two cache ports. Each stacked bar shows cumulative IPC attainable beyond the conventional superscalar with instruction flow, memory dataflow, and register dataflow superspeculative techniques.

anticipated applications. If necessary, designers can reduce the size of the level-2 cache to make room for various I/O coprocessors and network interfaces. These estimates are approximate.

Figure 3 on page 65 shows performance results for an unlimited number of cache ports, and eight, four, and two cache ports. Results for a model with a perfect cache (one that incurs no cache misses) are also included for reference. Each stacked bar reflects the additional IPC harvested by adding superspeculative instruction flow techniques, memory dataflow techniques, and register dataflow techniques to the machine model. We have published additional detailed performance results.[3] These results conclusively demonstrate not only that superspeculative techniques provide impressive performance, but also that without them, very wide superscalars (such as the baseline case shown) do not scale to significantly improved levels of performance.

Our results demonstrate Superflow's dramatic performance potential, placing superspeculative microarchitecture in the forefront of competing approaches for billion-transistor computing. We are currently exploring these techniques in greater depth with more extensive and accurate simulations. The next phase of our effort will be to move these ideas toward implementation and produce a Superflow prototype after extensive implementation trade-off studies. We plan to develop simulation models of the prototype that can provide performance accuracy down to the machine cycle level. Trial circuit designs for the timing-critical pieces of the prototype will help us estimate chip complexity and machine cycle time. ❖

....

....

References

1. K. Olukotun et al., "The Case For a Single-Chip Multiprocessor," *Proc. Seventh Int'l Conf. Architectural Support for Programming Languages and Operating Systems*, ACM Press, New York, 1996, pp. 2-11.
2. M.H. Lipasti and J.P. Shen, "Exceeding the Data-Flow Limit Via Value Prediction," *Proc. 29th Ann. ACM/IEEE Int'l Symp. on Microarchitecture*, IEEE CS Press, Los Alamitos, Calif., 1996, pp. 226-237.
3. M.H. Lipasti, *Value Locality and Speculative Execution*, doctoral dissertation, Carnegie Mellon Univ., Dept. Electrical and Computer Eng., May 1997.
4. T.M. Conte et al., "Optimization of Instruction Fetch Mechanisms for High Issue Rates," *Proc. 22nd Int'l Symp. on Computer Architecture*, IEEE CS Press, Los Alamitos, Calif., 1995, pp. 333-344.
5. E. Rotenberg, S. Bennett, and J. Smith, "Trace Cache: A Low Latency Approach to High Bandwidth Instruction Fetching," *Proc. 29th Ann. ACM/IEEE Int'l Symp. on Microarchitecture*, IEEE CS Press, Los Alamitos, Calif., 1996, pp. 24-34.
6. S. McFarling, *Combining Branch Predictors*, Tech. Report TN-36, Digital Equipment Corp., Maynard, Mass., 1993, http://www.research.digital.com/wrl/home.html.
7. M.H. Lipasti and J.P. Shen, "The Performance Potential of Value and Dependence Prediction," *Proc. EURO-PAR '97*, Springer-Verlag, Passau, Germany, 1997.
8. M.H. Lipasti, C.B. Wilkerson, and J.P. Shen, "Value Locality and Load Value Prediction," *Proc. Seventh Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, ACM Press, New York, 1996, pp. 138-147.
9. L. Widigen, E. Sowadsky, and K. McGrath, "Eliminating Operand Read Latency," *Computer Architecture News*, Dec. 1996, pp. 18-22.
10. A. Moshovos et al., "Dynamic Speculation and Synchronization of Data Dependences," *Proc. 24th Int'l Symp. on Computer Architecture*, IEEE CS Press, Los Alamitos, Calif., 1997, pp.181-193.
11. A.S. Huang and J.P. Shen, "The Intrinsic Bandwidth Requirements of Ordinary Programs," *Proc. Seventh Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, ACM Press, New York, 1996, pp. 105-114.
12. S. Vajapeyam and T. Mitra, "Improving Superscalar Instruction Dispatch and Issue by Exploiting Dynamic Code Sequences, *Proc. 24th Int'l Symp. Computer Architecture*, ACM Press, New York, 1997, pp. 1-12.

Mikko H. Lipasti is an advisory engineer with IBM. His research interests include superspeculative computer architecture. Lipasti has a BS in computer engineering from Valparaiso University and an MS and PhD in electrical and computer engineering from Carnegie Mellon. He is a member of the IEEE, ACM, and Tau Beta Pi.

John Paul Shen is a professor in CMU's Electrical and Computer Engineering Department and heads up the Microarchitecture Innovation Group. Shen received a BS from the University of Michigan and an MS and PhD from the University of Southern California, all in electrical engineering. He is an IEEE fellow.

Contact Shen at the Microarchitecture Innovation Group, Dept. of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, PA 15213; shen@ece.cmu.edu.