KILO-INSTRUCTION PROCESSORS: OVERCOMING THE MEMORY WALL

KILO-INSTRUCTION PROCESSORS ARE A NEW TYPE OF OUT-OF-ORDER SUPERSCALAR PROCESSOR THAT OVERLAPS LONG MEMORY ACCESS DELAYS BY MAINTAINING THOUSANDS OF IN-FLIGHT INSTRUCTIONS, IN A SCALABLE, EFFICIENT MANNER.

Adrián Cristal, Oliverio J. Santana, Francisco Cazorla, Marco Galluzzi, Tanausú Ramírez, Miquel Pericàs, and Mateo Valero
Universitat Politècnica de Catalunya and Barcelona Supercomputing Center

Historically, advances in integrated circuit technology have driven improvements in processor microarchitecture and led to today's microprocessors with sophisticated pipelines operating at very high clock frequencies. However, performance improvements achievable by high-frequency microprocessors have become seriously limited by main-memory access latencies because main-memory speeds have improved at a much slower pace than microprocessor speeds. It's crucial to deal with this performance disparity, commonly known as the *memory wall*,[1] to enable future high-frequency microprocessors to achieve their performance potential.

To overcome the memory wall, we propose kilo-instruction processors—superscalar processors that can maintain a thousand or more simultaneous in-flight instructions. Doing so means designing key hardware structures so that the processor can satisfy the high resource requirements without significantly decreasing processor efficiency or increasing energy consumption.

Nature of the memory wall

One of the first approaches to the memory wall problem was the development of cache memory hierarchies. Cache memories exploit program locality and can dramatically reduce the number of long-latency accesses to main memory. The first level, or L1 cache, is built into the processor core and typically takes one to three processor clock cycles to access. If there is a miss in the L1 cache, the on-chip L2 cache takes on the order of 10 processor cycles. Accessing main memory, on the other hand, takes at least an order of magnitude longer, and in the future this will become two orders of magnitude, that is, several hundred clock cycles. (In general, the cache hierarchy can have more than two levels, but to simplify our discussion here, we assume two levels with the understanding that the same principles apply to systems with deeper cache hierarchies.)

Modern superscalar processors employ out-of-order execution as a way of smoothing out disruptions caused by data cache misses (see the "Hiding latency in superscalar processors" sidebar). If a load instruction should experience a data cache miss, then instructions that depend on the miss data must wait in the issue queue(s). Meanwhile, independent instructions are free to execute; they issue from the issue queue(s) and essentially "pass" the blocked load instruction and its dependent instructions. For an L1 cache miss, these out-of-order instructions can often completely hide the L2 access latency, so the miss causes little or no performance loss.

This approach is much less effective for the long L2 cache misses, however. For example, along the top of Figure 1 is a sequence of instructions in program order. Following a

Published by the IEEE Computer Society

Hiding latency in superscalar processors

At the beginning of a superscalar pipeline like that shown in Figure A, the instruction fetch unit relies on branch prediction to fetch a stream of instructions following the most likely execution path. The processor decodes fetched instructions and assigns physical registers to hold their results. This register renaming process maps the architected registers into a larger set of physical registers and assures that any interinstruction dependences are true data dependences. After renaming, instructions go into an instruction issue queue (or one of a small number of issue queues, depending on the implementation), and the processor assigns an entry for the instruction in the reorder buffer (ROB). It also assigns entries in load-store queues (LSQs) to load and store instructions .

Instructions can issue from an issue queue as soon as all their input operand values are ready (subject to constraints such as available functional units and register read ports). So, strictly speaking, the issue queue is not first-in first-out (FIFO); instructions can issue *out-of-order.* On the other hand, the ROB follows a strict FIFO discipline, and the processor does not remove (or commit) an instruction from the ROB until it and all prior instructions have completed. When an instruction commits, its result becomes part of the architected program state. Hence, the ROB contains a record of all the pending or *in-flight* instructions. Among other things, the ROB assures that the processor can construct the correct architected state at the time of a trap or interrupt.

When an instruction commits, its physical register holds the value of the architected register that is mapped to it; hence the rename register can release the physical register previously assigned to the same architected register, permitting its reassignment. Also, at the time a load or store instruction commits, the processor can release its LSQ entry and physically write data values held in the store queue to the memory hierarchy.

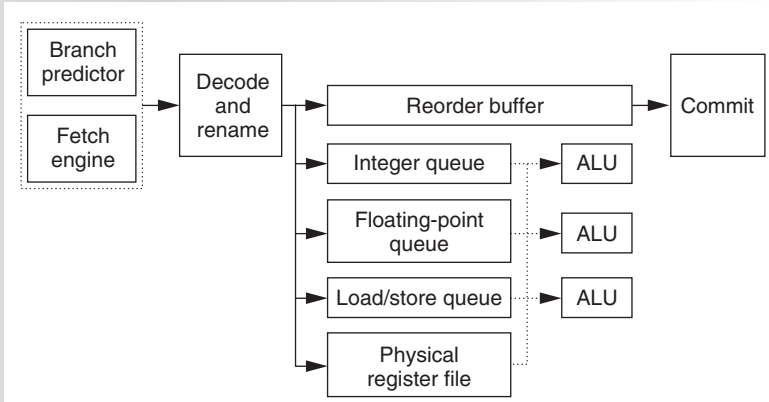

Figure A. Block diagram of an out-of-order superscalar processor.

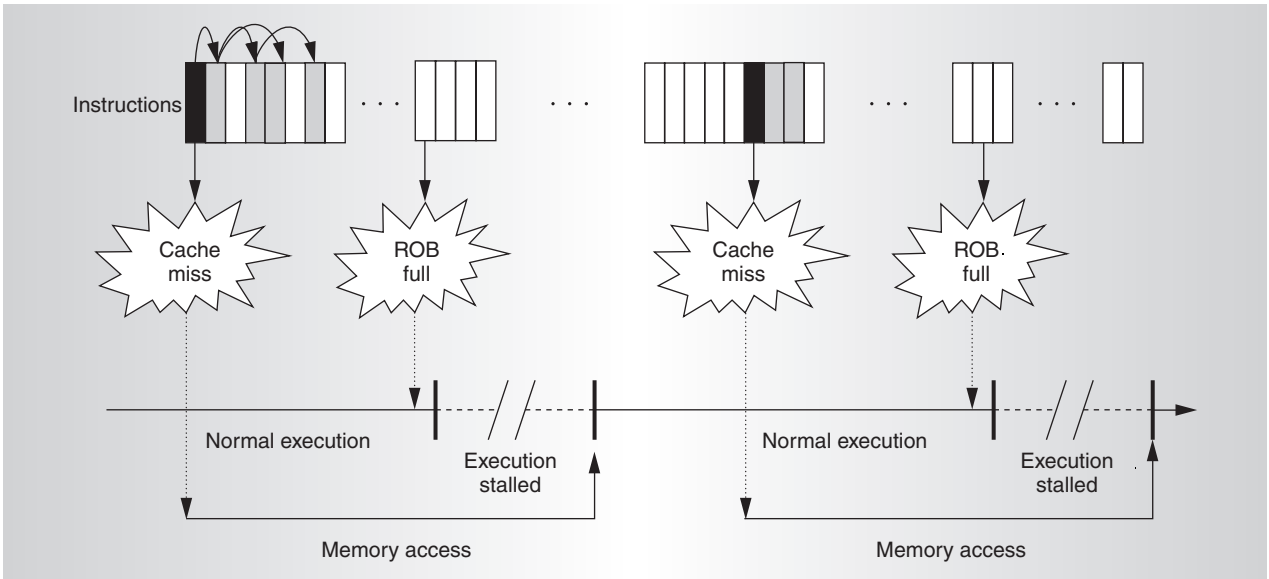

Figure 1. Sequence of instructions containing data cache misses. When the reorder buffer fills up, instruction execution stalls.

load instruction that misses in the cache (black boxes in the figure) are several dependent instructions (gray boxes). However, there are also independent instructions (white boxes).

When an L2 miss occurs, it triggers a memory access for the miss data, and normal instruction execution continues. However, when the load that misses reaches the head of the reorder buffer (ROB), it blocks, and no instructions can commit until the load

completes. Eventually, the ROB will fill up completely; no more instructions can go into the issue queue(s) and instruction execution stalls, regardless of whether or not there are further independent instructions.[2] If the miss latency is several hundred clock cycles, then the processor will spend most of this time idle, with the ROB blocked, waiting for the load miss instruction to complete.

To make matters worse, cache misses often happen in bursts because of program working set changes. As soon as the ROB becomes unblocked, it often blocks again due to another L2 miss. This produces a series of long stall periods, waiting for main memory to respond to a series of L2 misses. The memory wall strongly asserts itself.

Overcoming the memory wall

One solution to the memory wall problem is to reduce the number of L2 misses in the first place by prefetching data from the memory to the cache. Conventional prefetch techniques use heuristics to identify addressing patterns and predict memory addresses that load instructions will likely request in the future. For long-latency memory, prefetching is of limited value, however. One problem is that addressing patterns must be highly predictable, and this is not always the case. A second problem is that prefetches must occur far in advance to overcome very long memory latencies, and this makes prefetch addresses more difficult to predict. Finally, any unused prefetches create wasteful memory traffic, which may slow down other processors in multiprocessor systems, degrading system performance.

Assisted threads[3] is a much more elaborate technique that uses a higher amount of semantic information to improve prefetch effectiveness. It relies on pre-executing future parts of the program, selected at compile time or generated dynamically at runtime. Then, it takes advantage of idle processor resources to pre-execute the selected parts of the program. For example, in simultaneous multithreaded processors, idle thread contexts can perform the pre-execution. This pre-execution by idle threads generates smart memory prefetches based on actual program flow, thereby boosting the cache performance of primary-thread execution.

Runahead execution also targets enhanced prefetching behavior.[4] This technique pre-executes future instructions while an L2 cache miss is in progress. To avoid blocking the ROB, runahead execution temporarily assigns a bogus value to the load miss and speculatively executes all the following instructions, releasing ROB entries. Later, when the load miss finishes, the processor recovers the architected state at this point and restarts execution using the correct value. It discards all the results obtained during the speculative execution because they might be based on the bogus value. The speculative execution provides accurate prefetches when memory operations are re-executed.

Assisted threads and runahead execution can improve prefetch accuracy, but still suffer from some of the disadvantages of conventional prefetching. In addition, these techniques execute many extra instructions with throw-away results. As a consequence, these speculative techniques not only consume available resources, but also significantly increase energy consumption.

Kilo-instruction processors

Kilo-instruction processors overcome the memory wall by dramatically increasing the number of in-flight instructions; doing so hides L2 data cache misses in a manner similar to ways for hiding L1 misses. Consider Figure 2, which is similar to Figure 1, except that the much larger ROB in a kilo-instruction processor allows many more in-flight instructions before it blocks. This means that not only can more independent instructions overlap the L2 cache miss, but program execution reaches other L2 misses more quickly and can overlap the misses with each other.

Figure 3 illustrates the impact of increasing the number of in-flight instructions supported by a four-instruction-wide out-of-order superscalar processor when the main-memory access latency ranges from 100 to 1,000 cycles. We provide data for both the SPEC2000 integer and floating-point applications, using a 512-Kbyte 4-way-associative second-level cache with 10-cycle latency. These graphs show performance measured as average instructions executed per cycle versus the number of in-flight instructions. The dashed line shows performance with an ideal memory system (no cache misses). For conventional super-

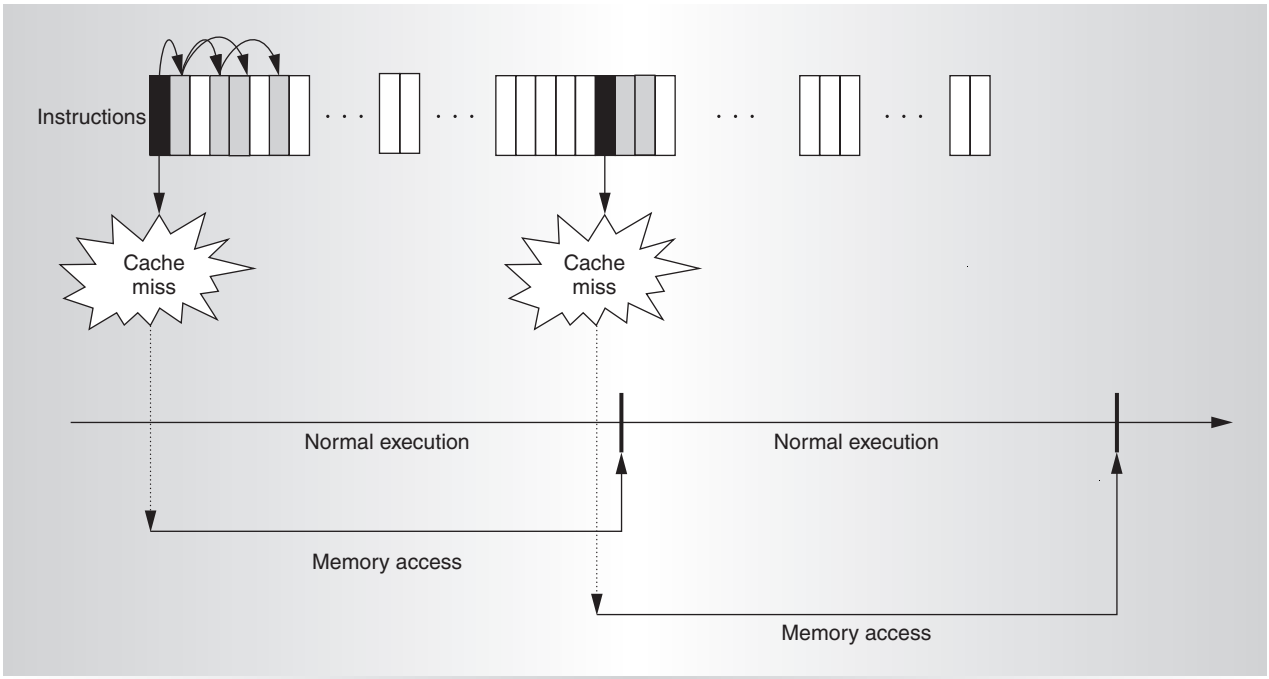

Figure 2. With a kilo-instruction processor, hardware resources support large number of in-flight instructions. Doing so greatly reduces stalls due to ROB fills and permits the overlap of long-latency memory accesses.

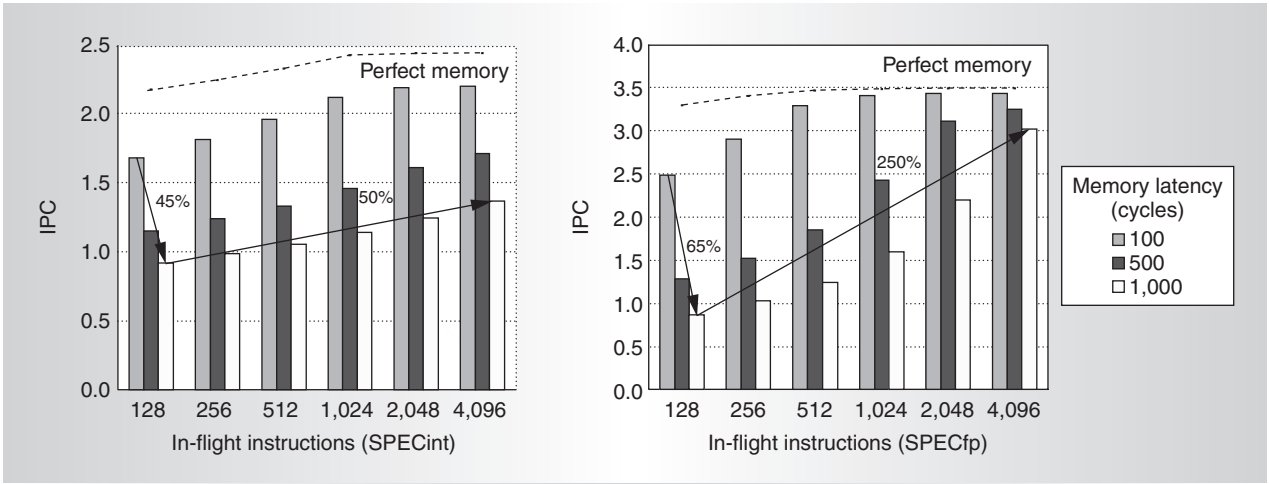

Figure 3. Average performance of a four-instruction-wide out-of-order superscalar processor executing the SPEC2000 integer and floating-point programs for different memory latencies and number of supported in-flight instructions.

scalar processors (those with 128 in-flight instructions), increasing the main-memory latency causes a huge performance degradation: 45 percent for integer applications, 65 percent for floating-point. It is possible to largely mitigate this performance degradation by increasing the number of in-flight instructions. In a processor with 4,096 in-flight instructions, increasing the main-memory latency results in a lower degradation: 40 percent for the integer programs and only 15 percent for floating-point programs. In other words, when the main memory access latency is 1,000 cycles, a processor with 4,096 in-flight instructions improves the performance of a processor with 128 in-flight instructions by 50 percent for integer programs, which are often limited by the presence of pointer chasing

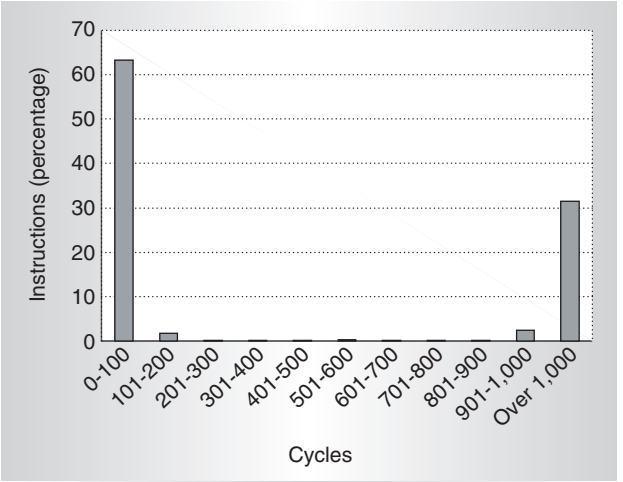

Figure 4. Distribution of instruction flight times measured in cycles. This time begins when an instruction goes into the ROB and ends when it commits and leaves the ROB.

(multiple levels of indirection through memory) and hard-to-predict branches. Because these limitations are much less severe for floating-point programs, their performance is improved by 250 percent.

These results show that increasing the number of in-flight instructions is an effective way of tolerating large memory access latencies. Therefore, at least conceptually, kilo-instruction processors provide a promising approach for future processors, making them capable of tolerating large memory access latencies.

Efficient kilo-instruction processor design

Kilo-instruction processors originated in a proposal by Cristal et al.[5,6] Although the idea is conceptually simple, one cannot design kilo-instruction processors simply by increasing the sizes of the major processor structures: the ROB, issue queue(s), physical registers, and load-store queues (LSQs). Naively scaling up the number of entries in these structures to support thousands of in-flight instructions is impractical because of area, temperature, and energy consumption constraints, and because the sizes of these structures often determine the processor cycle time. Cristal et al. proposed overcoming this problem by using an efficient mechanism that changes the management of these processor structures, making the design of kilo-instruction processors practical.

Figure 4 shows a key property that makes kilo-instruction processors feasible. The figure contains the distribution of instruction *flight times*; that is, the lengths of time instructions are in the ROB. This number is important because the reservation of many of the critical hardware resources occurs when an instruction goes into the ROB; those resources are not released until the instruction commits and leaves the ROB.

The Figure 4 data is an average over the SPECfp2000 benchmarks for a processor that can maintain 128 in-flight instructions; the main-memory latency is 1,000 cycles. The key aspect of this data is the bimodal nature of instruction behavior. Most instructions either have a short flight-time (that is, they hold resources for a short time) or they hold resources for a very long time. These latter instructions are the ones blocked in the ROB because of long-latency memory operations.

Clearly, the hardware resources reserved by instructions in the second category are highly underutilized.[7] Kilo-instruction processors exploit the bimodal flight time distribution by attempting to give critical resources to the short-flight-time instructions (at the left of the distribution) and early releasing resources that the long-flight-time instructions use (reserving them again, later). In some cases, the processor temporarily off-loads the long-flight-time instructions to lower-cost hardware structures. Kilo-instruction processors apply targeted implementations for each type of hardware structure and integrate these structures in a cohesive way. We discuss implementations for each of the four major hardware structures in the following subsections.

Early release of ROB resources via checkpointing

In a conventional superscalar processor, the ROB keeps a copy of all in-flight instructions so the processor can restore the correct architectural state after branch mispredictions or exceptions (traps and interrupts). Thus, maintaining thousands of in-flight instructions would require a ROB having thousands of entries. Kilo-instruction processors reduce the ROB's size requirement by using selective checkpointing.

A *checkpoint* is a snapshot of the processor state taken at a specific instruction of the program being executed. Using this snapshot, the processor can restore state to that point if an

exception or misprediction should occur. The ROB allows the restoration of the correct state at *any* instruction. This is not strictly necessary for correct operation, however. An alternative is to checkpoint processor state for a subset of instructions; then, if there is an exception or mispredicted branch, the processor can roll the state back to the closest checkpoint prior to the instruction causing the exception. Execution can then proceed forward until it reaches the correct, precise state. Using a relatively small set of checkpoints for long-flight-time instructions assures safe points of return and considerably reduces ROB requirements. The cost is longer recovery time when a long-flight-time instruction suffers an exception or (more importantly) a branch misprediction.

The implementation of a checkpointing mechanism involves several design decisions. First, the processor can use a few or many checkpoints. The latter reduces the recovery penalty, but increases the implementation cost. Second, the designer should determine the instruction types for checkpointing. Finally, the designer should define the state information to be captured at each checkpoint. In general, it is only necessary to keep the information strictly required to recover the correct processor state. However, it can be beneficial to store additional information to reduce recovery time.

Depending on the alternatives, researchers have proposed several checkpointing techniques for implementing processors supporting a large number of in-flight instructions. Cristal et al.[6] propose selective checkpoints taken only at certain load instructions. In particular, this work recommends taking a checkpoint when a load that misses in the L2 cache reaches the head of the ROB. After taking such a checkpoint, the processor can *early release* the ROB resources. In addition, this also releases the physical registers and LSQ slots used by instructions in the ROB. Instructions independent of the L2-missing load can then use these resources.

The Cherry architecture,[8] an independently developed proposal, uses a single checkpoint outside the ROB, which makes it possible to divide the ROB into two regions: speculative and nonspeculative. Cherry can then early release physical registers and LSQ entries for instructions in the nonspeculative ROB region.

The Checkpoint Processing and Recovery (CPR) architecture[9] is a proposal that uses a multi-checkpointing mechanism similar to that proposed by Cristal et al.[6] CPR removes the ROB entirely and checkpoints hard-to-predict branches at the decode stage.

The conventional ROB can also be replaced with a structure called the *pseudo-ROB.*[10] The pseudo-ROB has the same functionality as a ROB, but the processor removes instructions that reach the pseudo-ROB's head at a fixed rate, independent of the instructions' completion state. Because the processor state is recoverable for any instruction in the pseudo-ROB, generating a checkpoint is only necessary when an incomplete instruction leaves the pseudo-ROB. Delaying checkpoint generation in this manner reduces the performance impact of branch misprediction recovery. On average, 97.5 percent of mispredictions in SPECint2000 programs arise from branches that are still inside the pseudo-ROB. The percentage is even higher for SPECfp2000 programs: near 99 percent. This means that most branch mispredictions do not need to roll back to the previous checkpoint to recover the correct state, minimizing the misprediction penalty that a non-ROB design, like the CPR architecture, would suffer.

Bi-level issue queues

Unlike a conventional ROB, the issue queue holds an instruction only while the instruction is waiting for its input data operands and execution resources. After an instruction issues, its issue queue slot becomes available for another instruction. Consequently, an instruction may spend much less time in the issue queue than in the ROB. On the other hand, the issue queue is a very expensive resource: It is on the critical path for instruction execution and is one of the major energy consumers in the processor core. This means that a practical superscalar design has relatively few issue queue slots. Furthermore, when a load instruction suffers a long-latency data cache miss, any queued instructions dependent on the load continue to occupy their issue queue slots, as do any instructions that depend on those instructions. In general, some issue queue slots may not be tied up in this way[2] so instruction execution can progress. However, this progress will be restricted, and any additional L2 data

cache misses will tie up more (possibly all) issue queue slots.

A kilo-instruction processor overcomes the issue queue limitation by using bi-level issue queues, based on the bimodal distribution of issue queue occupancy just noted. First, the processor detects instructions that will hold an issue queue slot for a long time. It then removes these instructions from the primary issue queue, off-loading them to larger, but slower and less complex, structures. Later, when the long-latency load instruction completes and its dependent instructions become ready, the processor moves the off-loaded instructions back into the primary issue queue.

The Waiting Instruction Buffer (WIB)[11] follows the principles just outlined. It is a structure that holds all the instructions dependent on an L2 cache miss until the miss is resolved and the corresponding data returns from memory. The Slow-Lane Instruction Queue (SLIQ)[10] is similar in concept to the WIB, but it is an integral component of an overall kilo-instruction microarchitecture. For example, the microarchitecture also contains a pseudo-ROB, which reduces the complexity required to accurately detect the instructions with long issue queue occupancies. The pseudo-ROB allows long-occupancy detection to be deferred until it becomes critical, thus increasing the accuracy of long-occupancy instruction detection, while at the same time reducing the complexity of the logic required. The recently proposed Continual Flow Pipelines (CFP) architecture[12] is an efficient implementation of a bi-level issue queue, based on the Pentium 4 pipeline. It contains a Slice Data Buffer (SDB), which is similar to the WIB and the SLIQ. And, as with the SLIQ, the SDB is an integral part of an overall design with a complete set of scalable structures.

Physical register file

A conventional superscalar processor assigns physical registers to architected registers when an instruction enters the issue queue and ROB. A physical register remains assigned until a later instruction writing to the same architected register commits. Hence, an instruction reserves a physical register for its entire flight time (and then some). Because a large fraction of in-flight instructions have assigned physical registers, the number of physical registers required in a kilo-instruction processor would be extremely large, on the order of many hundreds.

Although assigned to an instruction early in the process, a physical register often is not written with a value until much later. In the meantime, its primary function is tracking data dependences. That is, the actual physical resource goes unused for a long time; its *name* is the only thing that is really important. Based on this observation, we can greatly reduce physical register requirements by using techniques for late register allocation, such as *virtual registers*. Instead of assigning a physical register to each renamed instruction, the renaming unit assigns only a virtual tag. These virtual tags keep track of the data dependences, making it unnecessary to assign a physical register to an instruction until it produces result.

In addition, the conventional rule for releasing physical registers is often very conservative. In theory, a processor can release a physical register as soon as it is known that no later instruction will read the register's value; for example, this could happen while the instruction holding the register is still in the ROB. This leads to techniques for the early (non-conservative) release of physical registers. One possibility is to associate a counter with each physical register for tracking the unexecuted instructions that read the register's contents. An instruction that will read the register increments the counter after register renaming; the instruction will later decrement the counter after reading the register. The processor can release the physical register when the counter decrements to zero. A problem remains if the processor should need the register's value to reconstruct the precise architected state in the event of an exception, but checkpoints solve this problem.

To greatly reduce physical register requirements in kilo-instruction processors, we combine techniques for early register release and late register allocation with the checkpointing mechanism, leading to an aggressive register-recycling technique that we call *ephemeral registers*.[7] This technique dissociates register release from the instruction commit process, and register allocation from instruction renaming.

The CFP architecture uses a similar

approach for releasing physical registers assigned to instructions dependent on a cache miss.[12] At the time these instructions enter the SDB, the CFP architecture uses physical register numbers as virtual tags. The instructions subsequently reacquire physical registers when they reenter the pipeline. In addition, this technique reads out register contents (similar to the original Tomasulo algorithm), which typically causes source registers to free up sooner than if they were tied up by load-miss-dependent instructions.

Scalability of load-store queues

The processor inserts load and store instructions into the LSQs at the same time it inserts them into the ROB. The primary function of these queues is to guarantee that all load and store instructions reach the memory system and update the architected state in correct program order.

An instruction reserves an LSQ slot for its entire flight time, so the number of LSQ slots should be proportional to the total number of in-flight instructions. As with issue queue slots, the LSQ slots are relatively complex structures that do not readily scale, so maintaining thousands of in-flight instructions can lead to an LSQ bottleneck. Using techniques for early release of load instructions can greatly reduce load queue resource requirements.[6,8] Several recent proposals deal with the scalability problems of store queues; these include bi-level[9] and partitioned[13] queues. Any of these mechanisms can be implemented in a kilo-instruction processor.

When integrated into an overall microarchitecture, the selective checkpoint technique combined with a pseudo-ROB, the SLIQ, ephemeral registers, and advanced LSQ techniques can support thousands of in-flight instructions while avoiding scalability problems found in conventional ROBs, issue queues, physical register files, and LSQs.

Future research in kilo-instruction processors

Kilo-instruction processors perform best for floating-point applications. Performance for integer programs is improved, but often not at the same level as floating-point. The achieved performance of integer applications is sometimes limited by hard-to-predict branches and pointer chasing. To overcome these limitations, kilo-instruction processors invite the reconsideration of techniques that might not be considered feasible or worthwhile in the context of conventional small-window superscalar processors. Such techniques might make good sense, however, in using thousands of in-flight instructions.

Hard-to-predict branches are especially harmful because a branch misprediction causes a pipeline flush, eliminating the advantage of a large number of in-flight instructions. Mispredicted branch instructions that depend on long-latency memory operations are even more detrimental because of the very long delay in discovering the misprediction. This provides a strong incentive for ongoing research efforts directed at accurate branch predictors. It is also possible to apply additional techniques. For example, combining kilo-instruction processors with multipath execution would allow multiple flows of control on a much larger scale than previously considered. It is also possible to combine kilo-instruction processors with mechanisms for detecting control-independent instructions that follow branches. There is no need to flush control-independent instructions after a branch misprediction, saving resources, energy, and execution bandwidth.

In some programs, pointer chasing is even more harmful than hard-to-predict branches, because it creates a serialization in the execution of long-latency memory operations, inhibiting the overlap illustrated in Figure 2. Although just 4 percent of load instructions chase pointers in floating-point applications, this percentage rises to 23 percent in integer programs. To solve this problem, value prediction techniques might be useful for predicting the addresses along a pointer chain, thereby allowing the overlap of these accesses.

Kilo-instruction processors can also be incorporated in multiprocessor designs. For example, the kilo-instruction multiprocessor proposed by Galluzzi et al.[14] uses kilo-instruction processors as computing nodes for building small-scale non-uniform memory access (NUMA) multiprocessors. In general, multiprocessor systems aggravate the memory latency problem, and in NUMA processors, the latency for accessing non-local data may be very long. Therefore, kilo-instruction processors and

multiprocessors seem to be a natural match; very long memory latencies can be hidden, boosting overall system performance.

Even more interesting is the use of the kilo-instruction processor checkpoint capability to implement multiprocessor transaction coherence and consistency[15] in a completely transparent manner. Checkpoints allow a processor to combine a series of memory load and store operations into a bundle that is committed to memory as a single transaction. If a memory operation in one processor's memory transaction would cause a coherence or consistency violation with a second processor's uncommitted loads and stores, then this violation can trigger the second processor to rollback and resume execution from a checkpoint preceding the violation. In contrast to the previous proposal,[15] the kilo-instruction processor checkpoint mechanism does not require any new instructions or re-writing of parallel software for correct operation. Moreover, there are a number of hardware implemented enhancements that can improve performance by adaptively selecting checkpoints based on dynamically observed memory behaviour.

In summary, kilo-instruction processors provide a flexible paradigm for constructing computer systems. Kilo-instruction processors are able to cross-pollinate readily with other system-level architecture techniques, thereby improving their capabilities and boosting overall processor performance. The possibilities are almost endless, creating a large number of new and appealing topics for future research. MICRO

Acknowledgments

This work has been supported by the Ministry of Education of Spain under contract TIN-2004-07739-C02-01, the HiPEAC European Network of Excellence, and the Barcelona Supercomputing Center. We would like to thank Srikanth Srinivasan, Ravi Rajwar, Haitham Akkary, and Konrad Lai for their worthwhile comments on this work. Thanks also go to Ayose Falcón, Rubén González, Daniel Jimenez, Josep Llosa, José F. Martínez, Daniel Ortega, and Alex Pajuelo for their contributions to the kilo-instruction processors. We would like to give special thanks to Jim Smith for his encouragement and valuable help during the writing of this paper.

References

1. W.A. Wulf and S.A. McKee, "Hitting the Memory Wall: Implications of the Obvious," *ACM SIGARCH Computer Architecture News*, vol. 23, no. 1, Mar. 1995, pp. 20-24.
2. T. Karkhanis and J.E. Smith, "A Day in the Life of a Data Cache Miss," *Proc. 2nd Ann. Workshop on Memory Performance Issues, 2002. http://www.ece.wisc.edu/~jes/papers/wmpi02.tejas.pdf*.
3. M. Dubois and Y. Song, *Assisted Execution*, tech. report CENG 98-25, Dept. EE-Systems, Univ. Southern California, 1998.
4. O. Mutlu et al., "Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-Order Processors," *Proc. 9th Int'l Symp. High-Performance Computer Architecture* (HPCA 03), IEEE CS Press, 2003, pp. 129-140.
5. A. Cristal et al., White paper: grant proposal to Intel-MRL in January 2002, Departament d'Arquitectura de Computadors, Universitat Politècnica de Catalunya, 2002.
6. A. Cristal et al., *Large Virtual ROBs by Processor Checkpointing*, tech. report UPC-DAC-2002-39, Dept. de Computadors, Universitat Politècnica de Catalunya, 2002. Submitted to 35th Int'l Sump. Microarchitecture (MICRO-35).
7. A. Cristal et al., "Toward Kilo-instruction Processors," *ACM Transactions on Architecture and Code Optimization*, vol. 1, no. 4, Dec. 2004, pp. 389-417.
8. J.F. Martínez et al., "Checkpointed Early Resource Recycling in Out-of-Order Microprocessors," *Proc. 35th Int'l Symp. Microarchitecture* (Micro-35), IEEE CS Press, 2002, pp. 3-14.
9. H. Akkary, et al., "Checkpoint Processing and Recovery: Towards Scalable Large Instruction Window Processors," *Proc. 36th Int'l Symp. Microarchitecture* (Micro-36), IEEE CS Press, 2003, pp. 423-434.
10. A. Cristal et al., "Out-of-Order Commit Processors," *Proc. 10th Int'l Symp. High-Performance Computer Architecture* (HPCA 04), IEEE CS Press, 2004, pp. 48-59.
11. A. Lebeck et al., "A Large, Fast Instruction Window for Tolerating Cache Misses," *Proc. 29th Int'l Symp. Computer Architecture* (ISCA 02), IEEE CS Press, 2002, pp. 59-70.
12. S.T. Srinivasan et al., "Continual Flow Pipelines," *Proc. 11th Int'l Conf. Architec-*

tural Support for Programming Languages and Operating Systems (ASPLOS 04), ACM Press, 2004, pp. 107-119.
13. S. Sethumadhavan et al., "Scalable Hardware Memory Disambiguation for High ILP Processors," *Proc. 36th Int'l Symp. Microarchitecture* (Micro-36), IEEE CS Press, 2003, pp. 399-410.
14. M. Galluzzi et al., "A First Glance at Kilo-Instruction Based Multiprocessors," *Proc. 1st Conf. Computer Frontiers*, ACM Press, 2004, pp. 212-221.
15. L. Hammond et al, "Transactional Coherence and Consistency: Simplifying Parallel Hardware and Software," *IEEE Micro*, vol 24, no. 6, Nov/Dec 2004, pp. 92-103.

Adrián Cristal is an assistant professor and a doctoral candidate in the Computer Architecture Department at the Polytechnic University of Catalonia (UPC), Spain. His research interests include processor microarchitecture and, in particular, kilo-instruction processors. Cristal has a BS and an MS in computer science from the University of Buenos Aires, Argentina.

Oliverio J. Santana is a collaborator professor in the Computer Architecture Department at UPC. His research interests include complexity-effective fetch and decoding architectures for high-performance processors. Santana has a PhD in computer science from UPC, as well as a BS and an MS in computer science from the University of Las Palmas de Gran Canaria (ULPGC), Spain.

Francisco Cazorla is a doctoral candidate in the Computer Architecture Department at UPC. His research interests include fetch policies for simultaneous multithreaded architectures. Cazorla has a BS and an MS in computer science from ULPGC.

Marco Galluzzi is a doctoral candidate in the Computer Architecture Department at UPC. His research interests include coherence and consistence multiprocessor protocols, and their relationships with techniques for tolerating the memory latency. Galluzzi has a BS and an MS in computer science from ULPGC.

Tanausú Ramírez is a doctoral candidate in the Computer Architecture Department at UPC. His research interests include techniques for overcoming the memory wall problem. Ramírez has a BS and an MS in computer science from ULPGC.

Miquel Pericàs is a researcher at the Barcelona Supercomputing Center and a doctoral candidate in the Computer Architecture Department at UPC. His research interests include efficient kilo-instruction processor implementations. Pericàs has an MS in telecommunications from UPC.

Mateo Valero is a professor in the Computer Architecture Department at UPC. His research interests include high-performance architectures. Valero has a PhD in telecommunications from UPC. He is an IEEE Fellow, an Intel Distinguished Research Fellow, and an ACM Fellow. Since 1994, he is a foundational member of the Royal Spanish Academy of Engineering.

Direct questions and comments about this article to Mateo Valero, D6-201 Campus Nord UPC, C/ Jordi Girona 1-3, 08034 Barcelona, Spain; mateo@ac.upc.edu.