A Case for Resource-conscious Out-of-order Processors

Adrián Cristal[†], José F. Martínez[‡], Josep Llosa[†], and Mateo Valero[†]

† Departament d'Arquitectura de Computadors
Universitat Politécnica de Catalunya
08034 Barcelona, Spain
`{adrian,josepll,mateo}@ac.upc.es`

‡ Computer Systems Laboratory
Cornell University
Ithaca, NY 14853 USA
`martinez@csl.cornell.edu`

***Abstract*—Modern out-of-order processors tolerate long-latency memory operations by supporting a large number of in-flight instructions. This is achieved in part through proper sizing of critical resources, such as register files or instruction queues. In light of the increasing gap between processor speed and memory latency, tolerating upcoming latencies in this way would require impractical sizes of such critical resources.**

To tackle this scalability problem, we make a case for *resource-conscious* out-of-order processors. We present quantitative evidence that critical resources are increasingly underutilized in these processors. We advocate that better use of such resources should be a priority in future research in processor architectures.

***Index Terms*—Out-of-order processor, memory latency, instruction-level parallelism, resource utilization, checkpointing.**

I. Introduction

THE gap between processor speed and memory latency is continuously widening. In order to tolerate long-latency memory operations, modern out-of-order processors maintain a large number of in-flight instructions that effectively hide such latencies. At current trends, however, processors cannot keep up with this growing disparity, and as a result, long-latency operations are increasingly more taxing on performance.

Figure 1 is a motivating example of this problem. It shows average IPCs attained by SpecFP and SpecInt applications in simulations using variations of the processor and memory models summarized in Table I. The three leftmost bars show the effect of increasing memory latency on a processor with 64 entries in reorder buffer (ROB), instruction queues, and register files. As memory latency increases, the limited processor resources are unable to keep enough instructions in flight and, as a result, the IPC drops dramatically.

By increasing the size of such resources, the processor can conceivably support enough in-flight instructions to tolerate long-latency operations. (Similar observations have been made elsewhere [7][8].) In Figure 1, the remaining bars show the effect that such idealized size increase has on IPC when memory is set to a futuristic latency of 500 processor cycles. *Limited ROB*, *Regs*, and *Queue* represent configurations with a limited number of entries (as indicated on the X axis) in ROB, register files, or instruction queues, respectively, and an unlimited size of the other resources in each case.

In the case of SpecFP applications, performance suffers if *any one* of these three resources is too small in size. However, as the size of the limited resource increases, so does the IPC.

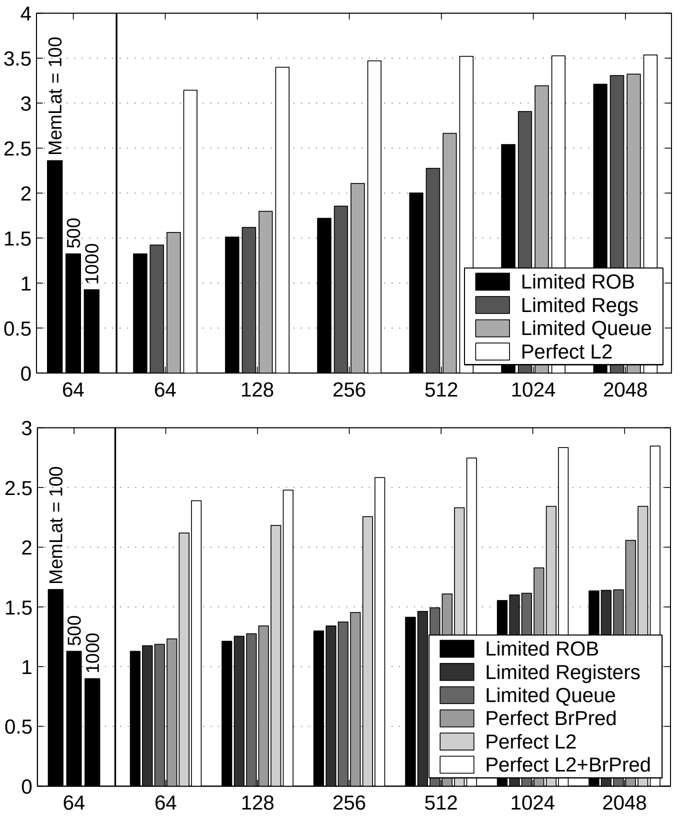


Figure 1. Average IPC of SpecFP and SpecInt applications varying size of critical resources and L2 miss penalty.

Only when all three resources are scaled up adequately can we maintain enough number of in-flight instructions to bring the IPC closer to that of a memory-insensitive configuration (represented by *Perfect L2* in each group).

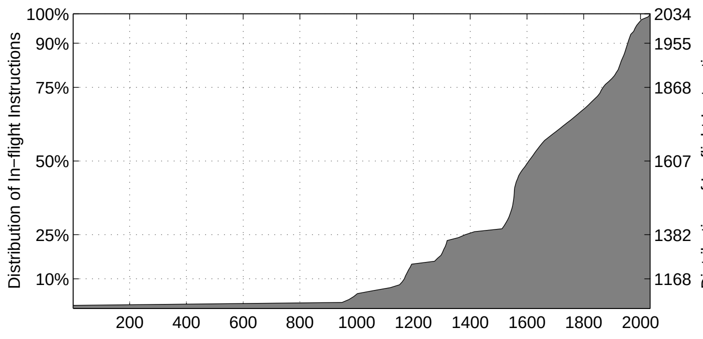
In the case of integer applications, larger resources are still insufficient to overcome long-latency operations. One reason for this is that many in-flight instructions are not profitable, as relatively frequent branch mispredictions squash a large number of them. However, once hard-to-predict branches are overcome (*Perfect BrPred*), increasingly higher IPCs are obtained as resources scale up in size. (Still, the observed IPC is far from that of *Perfect (L2+BrPred)*, in part because of pointer-chasing memory access patterns.)

In general, however, it is impractical to design processors with thousands of entries in resources such as the ROB, the register file, and the various queues, since this could adversely affect clock cycle, pipeline depth, or both [13].

In this paper, we analyze resource utilization in out-of-order processors, and quantitatively show that, behind this *apparent* need to grow critical resources to support many in-flight instructions, a significant fraction of such resources are in fact

Appears in *IEEE TCCA Computer Architecture Letters*, Oct. 2003.

This work first appeared as Cornell Computer Systems Lab Tech. Rep. CSL-TR-2003-1034, May 2003, and was presented in the MEDEA workshop, concurrently with Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT), Sep. 2003.



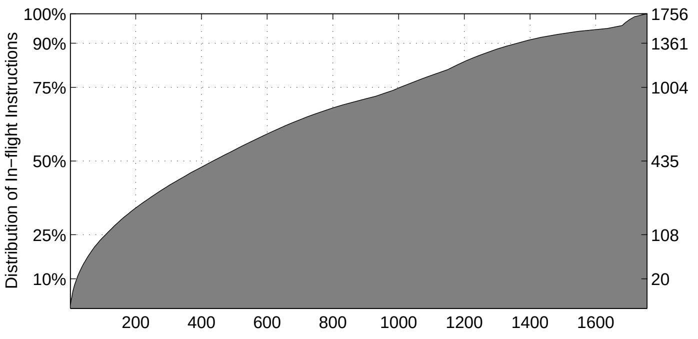


Figure 2. Distribution function of in-flight instructions in SpecFP (left) and SpecInt (right) applications.

being wasted by blocked or executed instructions. We identify the different causes of resource wasting, and motivate research on *resource-conscious* processors to overcome upcoming memory latencies. We provide some examples of existing work in this direction.

II. Resource Utilization

Figure 2 shows the average cumulative distribution of in-flight instructions for SpecFP and SpecInt applications, obtained through simulations using the parameters of Table I. An idealistic 2,048-entry ROB and enough entries in all other resources are assumed. (This corresponds to the first bar of the 2,048-entry group in Figure 1.) Floating-point applications have, on average, more than 1,500 instructions in flight 75% of the time, while integer applications exhibit a relatively moderate number of instructions in flight (under 500 50% of the time).

Since each in-flight instruction is assigned an entry in the ROB until it is retired, the ROB should be rather large in order to support high memory latencies. Techniques to cope with small ROBs [12] or to construct large *virtual* ROBs that allow early release of uncommitted instructions [4] have been proposed.

We now discuss the allocation of various other critical resources, namely register file, instruction queues, and load and store queues. For each resource, we plot the average number of allocated entries against the number of in-flight instructions. The X axis is adjusted according to the distribution function shown in Figure 2.

A. *Register files*

Figure 3 shows the average number of allocated registers against the distribution of in-flight instructions. Registers are classified in four categories as follows:

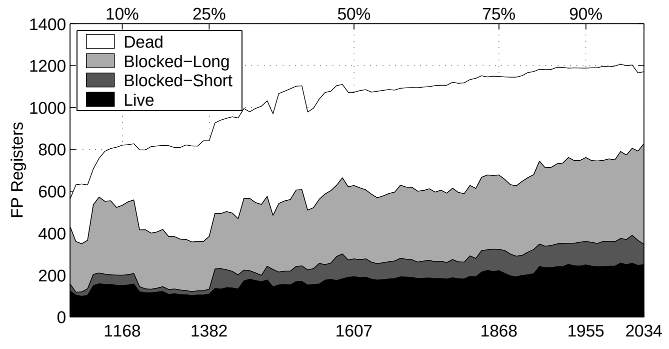
Live registers contain values currently in use. Notice that this class constitutes only ten to twenty percent of the total number of allocated registers.

TABLE I: Base Processor Configuration

Element	*Entries*
ROB	64-2k entries
Ld/St Queue	64-2k entries
Int Queue	64-2k entries
FP Queue	64-2k entries
Fetch/Decode/Commit	4/4/4
FUs	4 int/addr ALU, 2 int mult/div, 4+2 fp adders+mult/div, 2 memory ports
Branch predictor	16k-entry GShare
Branch penalty	8 cycles
L1 Data cache	32 kB, 4 way, 32B/line, 3 cycles
L1 Inst. cache	32 kB, 4 way, 32B/line, 3 cycles
L2 cache	256 kB, 4 way, 64B/line, 15 cycles
TLB	64 entries, 4 way, 8kB page, 30 cycles
Memory	500 cycles
Int/FP Register file	64-2k/64-2k entries

Blocked-Short registers have been allocated during rename, but are blocked at the instruction queue waiting for the execution of predecessor instructions that will issue shortly. This class is shortlived by definition, and represents a relatively small fraction of the allocated registers; therefore, attacking this type of registers would be of limited impact.

On the other hand, *Blocked-Long* registers are still empty because the producer instruction is blocked waiting for the execution of some long-latency predecessor instruction (e.g., a load miss). Conversely, *Dead* registers are no longer in use, but they are still allocated because the superseding producer instructions have not retired. Together, these two categories constitute the largest fraction of allocated registers, and should be the target of register management techniques. In Figure 3, results for SpecFP applications show that, at the median number of in-flight instructions (about 1,600), as many as 80 percent of the approximately 1,000 allocated floating-point registers fall in one of these two categories. Techniques for late physical register allocation [5], early register recycling



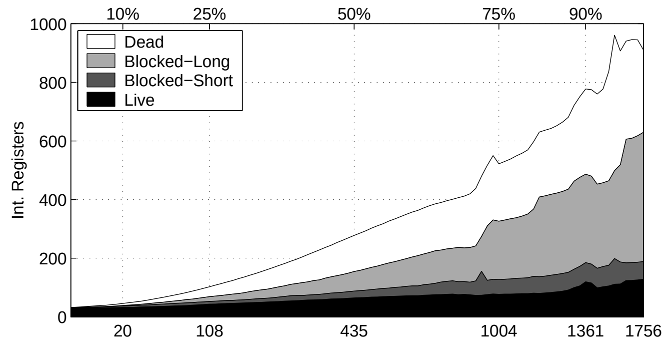
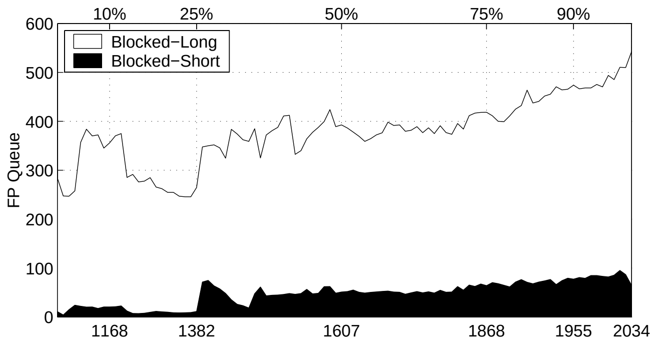


Figure 3. Breakdown of allocated floating-point and integer registers vs. no. of in-flight instructions in SpecFP (left) and SpecInt (right) applications, respectively. The horizontal axis is adjusted following the distribution function of in-flight instructions in each case (Figure 2).



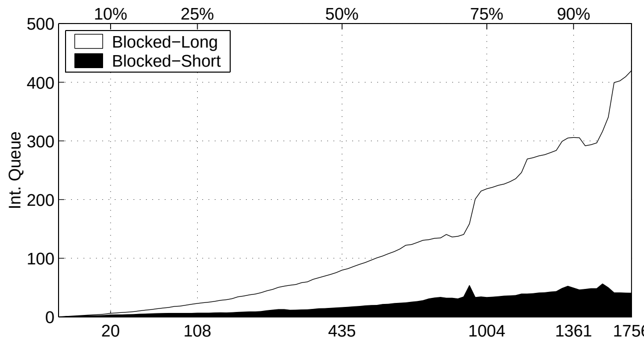


Figure 4. Breakdown of allocated floating-point and integer instruction queue entries vs. no. of in-flight instructions for SpecFP (left) and SpecInt (right) applications, respectively. The horizontal axis is adjusted following the distribution function of in-flight instructions in each case (Figure 2).

[4][10][11], or both [9], can make these registers available to other instructions.

B. Instruction queues

Figure 4 shows the average number of allocated entries in the floating-point queue (for SpecFP applications) and in the integer queue (for SpecInt applications). Unless explicitly noted, our comments are applicable to both.

Blocked-Short entries pertain to instructions that are waiting for a functional unit or for results from short-latency operations. This group represents a relatively small fraction of the allocated entries.

Blocked-Long entries, however, correspond to instructions that are blocked waiting for some long-latency instruction to complete. This group represents by far the largest fraction of entries allocated in the instruction queue: Figure 4 shows that, in SpecFP applications, at about 1,600 in-flight instructions, only 15 percent of the approximately 400 floating-point instruction queue entries are not in this category. Multilevel queues can be used to track this type of instructions, delegating their handling to slower, but larger and less complex structures [1][8][14][2].

C. Load queue

Figure 5 shows the average number of allocated load queue entries. They are broken down in the following categories:

Live entries correspond to loads that are being executed. This type abounds in floating-point applications, in which cache miss rates are higher than in their integer counterparts. It also includes loads that have executed out of program order with respect to some store whose address remains unresolved. These loads and their dependent operations must be *replayed* if the store is later found to overlap with the load.

Blocked-Short entries pertain to loads waiting for its address to be produced by a short-latency operation.

Blocked-Long entries belong to load instructions whose address depends on a long-latency operation. In the case of integer applications, this represents a significant fraction of the allocated load queue entries. This is because pointer chains are quite common in this type of applications.

Finally, *Dead* entries correspond to load instructions that have been executed and are not subject to replay traps, i.e., the addresses of all previous stores in program order have been resolved. For SpecFP applications, Figure 5 shows that, at about 1,600 in-flight instructions, as many as 75 percent of the approximately 400 allocated load queue entries are Dead. Traditional out-of-order processors keep these entries until their load retires, but aggressive implementations that recycle them earlier been proposed [4][10].

D. Store queue

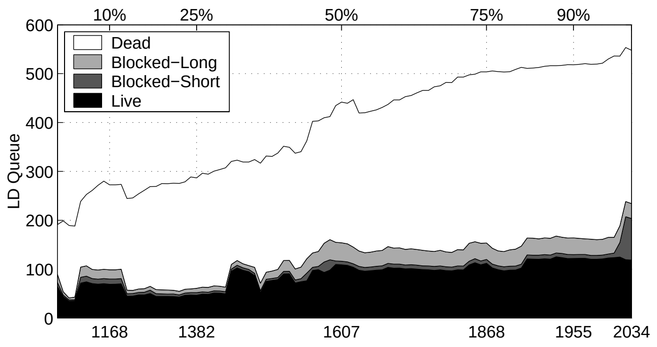
Figure 6 shows the average number of allocated store queue entries. They are classified as follows:

Ready entries represent store instructions whose address and source operand are available, and are only waiting to reach the ROB head to execute. Under the right conditions, these entries could be recycled before their store executes [10]. In general, however, exception handling and other issues still mandate in-order execution of stores.

Address Ready entries correspond to stores whose address is ready, but are still waiting for the data. These represent a significant part of all in-flight stores. In general, these entries could also be recycled if disambiguation with earlier memory operations is no longer necessary [10].

Blocked-Long entries relate to store instructions whose address depends on a long-latency operation. Notice that this category is nearly inexistent in floating point applications, since it is mostly data and not addresses that depend on long-latency operations.

Finally, *Blocked-Short* entries correspond to stores waiting for its address to be produced by short-latency operations. At about 1,600 in-flight instructions, Figure 6 shows that, in



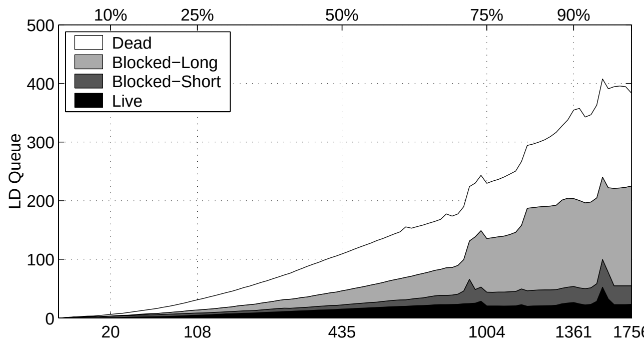


Figure 5. Breakdown of no. of allocated load queue entries vs. no. of in-flight instructions in SpecFP (left) and SpecInt (right). The horizontal axis is adjusted following the distribution function of in-flight instructions in each case (Figure 2).

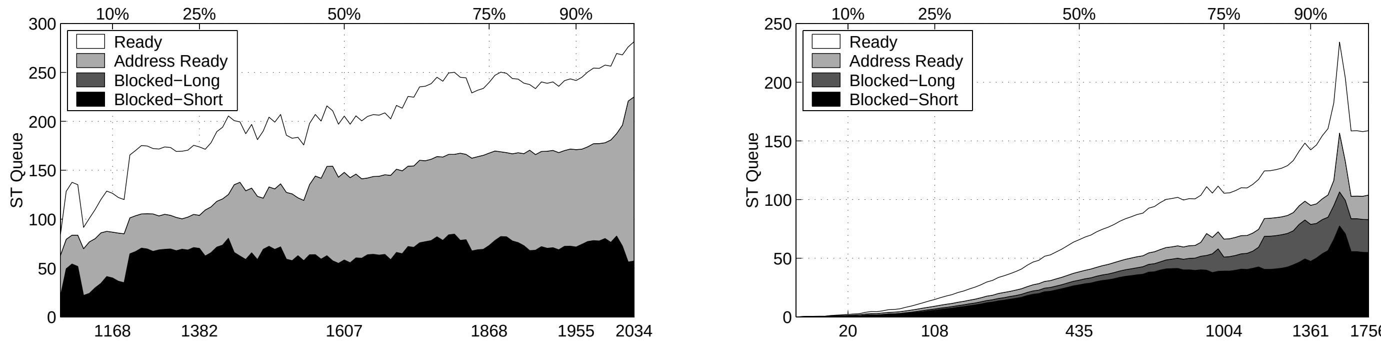


Figure 6. Breakdown of allocated store queue entries in SpecFP and SpecInt applications. The horizontal axis is adjusted following the distribution function of in-flight instructions in each case (Figure 2).

SpecFP applications, only 25 percent of approximately 200 allocated store queue entries fall under this category.

III. Conclusion

This paper provides quantitative evidence that, in the pursuit of large numbers of in-flight instructions to tolerate long memory latencies in current out-of-order processors, resources identified as critical to attaining high performance are in fact increasingly underutilized. Specifically, we have looked at register files, instruction queues, and load and store queues. We have broken down allocated resources into several categories according to how they are used, and found that a significant fraction of allocated resources actually does not contribute to execution.

These results point to future research that considers new, more efficient ways of supporting a large number of in-flight instructions, by reducing the amount and the complexity of hardware resources.

Designing processors that support thousands of in-flight instructions reopens classical research topics. Among those:

— Methods that allow the reuse of very large blocks of instructions (potentially hundreds) that are re-executed several times during the service of a very long-latency load.

— Continued research in branch prediction. Considering Figure 1, for integer applications, more accurate branch prediction is essential for a large number of in-flight instructions to be meaningful.

— Research that allows multiple flows of control, such as predicated execution and/or multipath execution, but on a larger scale than previously considered. These techniques do not provide impressive results in the context of short ROBs, but they may in the context of thousands of in-flight instructions.

— Methods that allow the design of very deep load-store queues. Load-store queues must be made to scale with the other structures.

Moreover, we believe that it is crucial to reconsider the way current processors commit instructions in program order to support precise exceptions. The results in this paper suggest that the majority of the resources the processor needs to support in-order commit are in fact severely underutilized most of the time. We believe that much inefficiency can be eliminated if processors support some form of coarse-grain checkpointing, to reduce the size and overhead of critical resources in the architecture such as ROB, register file, instruction queues, or load/store queues. It is possible to support precise exceptions with much fewer resources, at the expense of some performance penalty when exceptions do occur. One possible way is to apply checkpointing on a few, very specific instructions, e.g., long-latency loads or hard-to-predict branches, and allow some form of *out-of-order commit* of the other instructions (and the release of their resources) [2][3][4]. If checkpointing is applied strategically in this way, traditional ROBs can become virtually unnecessary [4]. Another option is to checkpoint periodically, and allow eager release of dead or likely-dead resources in all or part of the in-flight instructions [4][9][10][11].

Acknowledgments

We thank Yale Patt, Jim Smith, Guri Sohi, and Mark Hill for useful feedback. This work was supported in part by the Spanish Ministry of Science and Technology under Contract TIC-2001-0956-C04-01.

References

[1] E. Brekelbaum, J. Rupley II, C. Wilkerson, and B. Black. Hierarchical instruction windows. In *Intl. Symp. on Microarchitecture*, Nov. 2002

[2] A. Cristal, D. Ortega, J. Llosa and M. Valero. Out-of-order commit processors. In *Intl. Symp. on High-Performance Computer Architecture*, Feb. 2004

[3] A. Cristal, D. Ortega, J. Llosa, M. Valero. Kilo-instruction processors. Invited paper. ISHPC-V. In *Intl. Symp. on High Performance Computers*, October 2003. *Lecture Notes in Computer Science (LNCS) 2858*, 2003

[4] A. Cristal, M. Valero, J. Llosa, and A. González. Large virtual ROBs by processor checkpointing. Tech. Rep. UPC-DAC-2002-39, Universitat Politécnica de Catalunya, July 2002

[5] A. González, J. González, and M. Valero. Virtual-physical registers. In *Intl. Symp. on High-Performance Computer Architecture*, Jan.–Feb. 1998

[6] W. W. Hwu and Y. N. Patt. Checkpoint repair for out-of-order execution machines. In *Intl. Symp. on Computer Architecture*, June 1987

[7] T. Karkhanis and J. E. Smith. A day in the life of a data cache miss. In *Wkshp. on Memory Performance Issues*, in conjunction with *Intl. Symp. on Computer Architecture*, July 2002

[8] A. R. Lebeck, J. Koppanalil, T. Li, and J. Patwardhan, and Eric Rotenberg. A large, fast instruction window for tolerating cache misses. In *Intl. Symp. on Computer Architecture*, June 2002

[9] J. F. Martínez, A. Cristal, M. Valero, and J. Llosa. Ephemeral registers. Tech. Rep. CSL-TR-2003-1035, Computer Systems Lab, Cornell University, June 2003

[10] J. F. Martínez, J. Renau, M. C. Huang, M. Prvulovic, and J. Torrellas. Cherry: Checkpointed early resource recycling in out-of-order microprocessors. In *Intl. Symp. on Microarchitecture*, Nov. 2002

[11] M. Moudgill, K. Pingali, and S. Vassiliadis. Register renaming and dynamic speculation: An alternative approach. In *Intl. Symp. on Microarchitecture*, Dec. 1993

[12] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt. Runahead execution: An alternative to very large instruction windows for out-of-order processors. In *Intl. Symp. on High-Performance Computer Architecture*, Feb. 2003

[13] S. Palacharla, N. P. Jouppi, and J. E. Smith. Complexity-effective superscalar processors. In *Intl. Symp. on Computer Architecture*, June 1997

[14] S. E. Raasch, N. L. Binkert, and S. K. Reinhardt. A scalable instruction queue design using dependence chains. In *Intl. Symp. on Computer Architecture*, June 2002