A Comprehensive Description of Kilo-Instruction Processors

Adrian Cristal, Oliverio J. Santana, and Mateo Valero

Departament d'Arquitectura de Computadors
Universitat Politècnica de Catalunya
Barcelona, Spain
{adrian,osantana,mateo}@ac.upc.es

Abstract. Although the memory access latency can be tolerated by maintaining a high number of in-flight instructions, the continuous increase in the gap between processor and memory speed increases the number of in-flight instructions required, causing scalability problems in the design of the critical structures of the processor. Our approach to support thousands of in-flight instructions, while avoiding scalability problems, is the kilo-instruction processor. This affordable architecture relies on an intelligent use of the available resources instead of simply up-sizing the processor structures. The high number of in-flight instructions maintained by our architecture allows it to achieve a high performance, even in the presence of large memory latencies, which makes the kilo-instruction processor an efficient architecture for dealing with future memory latencies.

1 Introduction

A lot of research effort is devoted to design new architectural techniques able to take advantage of the continuous improvement in microprocessor technology. The current trend leads to processors with longer pipelines, which combines with the faster technology to allow an important increase in the processor clock frequency every year.

However, the main memory access latency has become an important limiting factor for the performance of high-frequency microprocessors. The DRAM technology improves at a speed much lower than the microprocessor technology. Due to this fact, each increase in the processor clock frequency causes that a higher number of processor cycles are required to access the main memory, degrading the potential performance achievable with the clock frequency improvement.

If the main memory access latency increase continues, it will be a harmful problem for future microprocessor technologies. Therefore, dealing with the gap between the processor and the memory speed is vital in order to allow high-frequency microprocessors to achieve all their potential performance. A plethora of well-known techniques has been proposed to overcome the main memory latency, like cache hierarchies or data prefetching, but they do not completely

A. Gelbukh, G. Sidorov, W. Olán, J. Vera (Eds.) Recientes avances en la ciencia de la computación en México, pp.144-154.© Centro de Investigación Computación, IPN, México

solve the problem. A different approach to tolerate the main memory access latency is to dramatically increase the number of in-flight instructions that can be maintained by the processor.

2 Increasing the Number of In-Flight Instructions

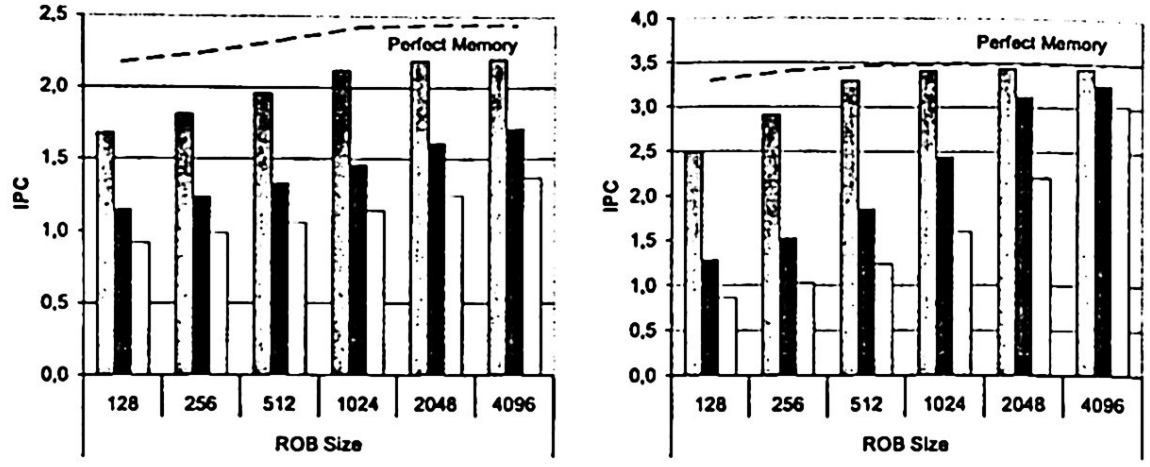
If the processor is able to maintain many in-flight instructions, the latency of a load instruction that access to the main memory can be overlapped with the execution of subsequent independent instructions, that is, the processor can hide the main memory access latency by executing useful work. Figure 1 shows an example of the impact of increasing the maximum number of in-flight instructions supported by an eight instruction wide out-of-order superscalar processor. The main memory access latency is varied from 100 to 1000 cycles. Data is provided for both the SPECint2000 integer applications and the SPECfp2000 floating point applications.

A first observation from this figure is that increasing the main memory latency from 100 to 1000 cycles causes enormous performance degradation. In a processor able to support 128 in-flight instructions, the integer applications suffer from an average 45% performance reduction. The degradation is even higher for floating point applications, whose average performance is reduced by 65%.

It is also clear that a higher number of in-flight instructions improves the processor performance. Increasing the number of in-flight instructions from 128 to 4096 in a processor having 100-cycle memory latency, the integer programs achieve an average 30% performance speedup and the floating point programs achieve 40% speedup. Nevertheless, it is important to note that the improvement is higher for larger memory access latencies. Increasing the number of in-flight instructions in a processor having 1000-cycle memory latency causes an average 50% performance improvement for the integer programs, while the floating point programs achieve a much higher 250% improvement. Such a high speedup is due to the larger amount of instruction-level parallelism available in floating point applications. This fact, along with a better branch prediction accuracy, allows floating point programs to take more advantage of a higher number of in-flight instructions.

These results show that increasing the number of in-flight instructions is an effective way of tolerating large memory access latencies. Although increasing the main memory latency from 100 to 1000 cycles causes big performance degradation for a processor able to maintain up to 128 in-flight instructions, a higher number of in-flight instructions alleviates this degradation, especially for floating point applications. If a processor is able to maintain up to 4096 in-flight instructions, the performance degradation caused by increasing the memory access latency from 100 to 1000 cycles is reduced by 10% in integer programs, and by 50% in floating point programs.

On average, executing integer programs, a processor able to maintain up to 4096 in-flight instructions having 1000-cycle memory latency is only 18% slower than a processor having 100-cycle latency but only being able to maintain up to



(a) SPECint2000 (b) SPECfp2000

Fig. 1. Average performance of an 8-wide out-of-order superscalar processor executing both the SPEC2000 integer and floating point programs. The maximum number of in-flight instructions supported is varied from 128 to 4096, and the main memory access latency is varied from 100 to 1000 cycles.

128 in-flight instructions. Moreover, when executing floating point applications, the processor supporting 4096 in-flight instructions with a 1000-cycle memory latency performs 22% better than the processor supporting 128 in-flight instructions, even when this processor has a much lower 100-cycle memory latency.

Therefore, future microprocessors will be able to tolerate large memory access latencies by maintaining thousands of in-flight instructions. The simplest way of supporting so much in-flight instructions is to scale all the processor resources involved, that is, the reorder buffer, the physical register file, the general purpose instruction queues (integer and floating point ones), and the load/store queue. However, scaling-up the number of entries in these structures is impractical, not only due to area and power consumption constraints, but also because these structures often determine the processor cycle time [14].

This is an exciting challenge. On the one hand, a higher number of in-flight instructions allows to tolerate large memory access latencies and thus provide a high performance. On the other hand, supporting such a high number of in-flight instructions involves a difficult scalability problem for the processor design. Our approach to overcome this scalability problem, while supporting thousands of in-flight instructions, is the kilo-instruction processor.

3 The Kilo-Instruction Processor

In essence, the kilo-instruction processor [5] is an out-of-order processor that keeps thousands of in-flight instructions. The main feature of our architecture is that its implementation is affordable. In order to support thousands of in-flight instructions, the kilo-instruction architecture relies on an intelligent use of the processor resources, avoiding the scalability problems caused by an ex-

cessive increase in the size of the main processor structures. Our design deals with the problems of each of these structures in an orthogonal way, that is, we apply particular solutions for each structure. These solutions are described in the following sections.

3.1 Multi-Checkpointing the Reorder Buffer

In a superscalar out-of-order processor, all instructions are inserted in the reorder buffer (ROB) after they are fetched and decoded. Therefore, the ROB is a microarchitectural mechanism that keeps a history window of all in-flight instructions, allowing for the precise recovery of the program state at any of those instructions. Instructions are removed from the ROB when they commit, that is, when they finish executing and update the architectural state of the processor.

However, for implementing precise recovery, instructions should be committed in-order, which is a serious problem in the presence of large memory access latencies. Let us suppose that a processor has a 128-entry reorder buffer and 500-cycle memory access latency. If a load instruction does not find a data in the cache hierarchy, it accesses the main memory, and thus it cannot be committed until its execution finishes 500 cycles later. When the load arrives to the head of the ROB, it blocks the in-order commit, and no later instruction will commit until the load finishes. Part of these cycles can be devoted to do useful work, but the ROB will become full soon, stalling the processor during several hundreds cycles.

To avoid this, a larger ROB is required, that is, the processor requires a higher number of in-flight instructions to overlap the load access latency with the execution of following instructions. Since each in-flight instruction requires an entry in the ROB, it should contain a high number of entries. However, scaling-up the number of ROB entries is impractical, mainly due to cycle time limitations.

The problem here is the presence of a centralized ROB structure devoted to provide precise recovery of the processor state. The kilo-instruction architecture solves this problem by replacing the ROB with a multi-checkpointing mechanism which also allows precise state recovery. Checkpointing is a well established and used technique [7]. The main idea is to create a checkpoint at specific instructions of the program being executed. This checkpoint can be though of as a snapshot of the state of the processor at that point, which contains all the information required to recover the architectural state and restart execution at that point.

The novelty of our mechanism is that the kilo-instruction architecture uses checkpointing to allow an early release of resources. Figure 2 shows an example of our checkpointing process [5]. First of all, it is important to state that there always exists at least one checkpoint in the processor (timeline A). The processor will fetch and issue instructions, taking new checkpoints at particular ones. If an instruction is miss-speculated or an exception occurs (timeline B), the processor rolls back to the previous checkpoint and resumes execution from there. When all instructions between two checkpoints are executed (timeline C), the last checkpoint is removed and its resources are freed (timeline D).

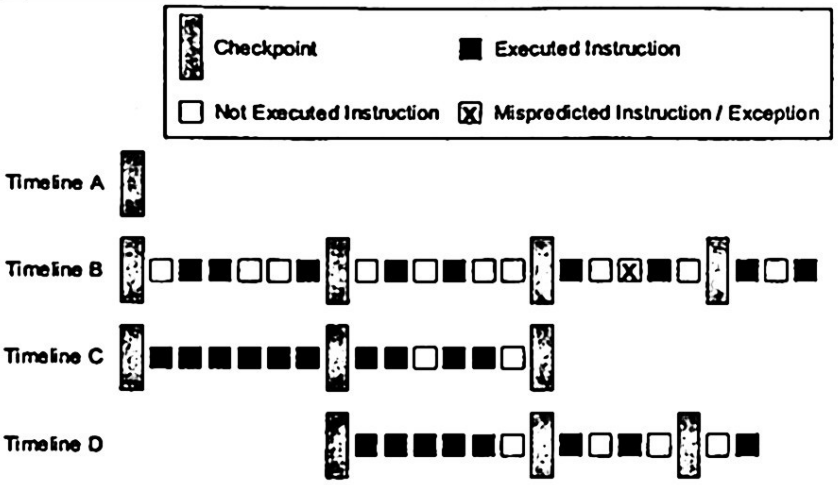


Fig. 2. The checkpointing process performed by the kilo-instruction processor.

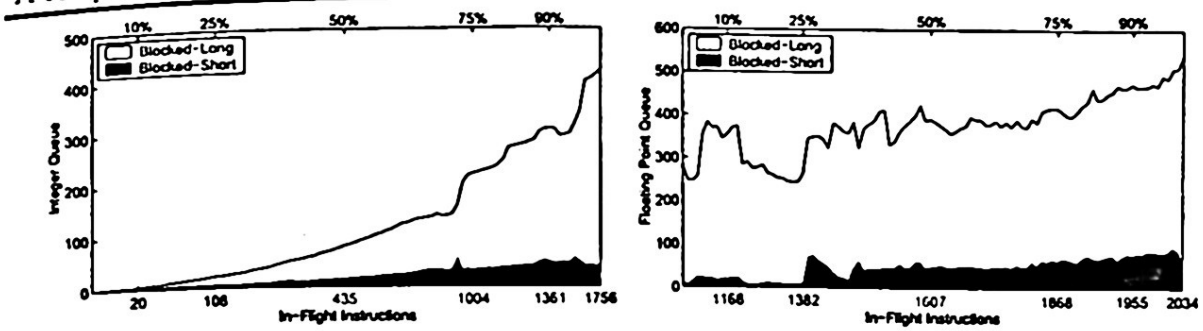
In case of a long-latency load, which accesses the main memory because its data is not in the cache hierarchy, the presence of a previous checkpoint allows that all the following instructions independent of the load result can finalize their execution and commit out-of-order, that is, they release their associated resources without having to wait until the load commits several hundreds cycles later. As a consequence, the multi-checkpointing mechanism makes it possible for the kilo-instruction processor to overlap large memory access latencies with the execution of thousands of following independent instructions without requiring an unimplementable centralized ROB structure with thousands of entries.

3.2 Instruction Queues

At the same time that instructions are inserted in the ROB, they are also inserted in their corresponding instruction queues. Each instruction should wait in an instruction queue until its execution finishes. All the instructions following a long-latency load can finalize their execution and be removed from the instruction queues due to the presence of a previous checkpoint. However, all the dependent instructions should be kept in the instruction queues until they finish their execution several hundreds cycles later.

This means that, in order to hide the load latency with the execution of thousands of following instructions, a typical instruction queue design should contain a high amount of entries, which makes it unpractical. The kilo-instruction processor solves this problem by taking advantage of the different waiting times of the instructions in the queues. These instructions can be divided in two types: blocked-short instructions when they are waiting for a functional unit or for results from short-latency operations, and blocked-long instructions when they are waiting for some long-latency instruction to complete.

Figure 3 shows the accumulative distribution of allocated entries in the integer queue (for SPECint2000 programs) and in the floating point queue (for SPECfp2000 programs) with respect to the amount of total in-flight instructions.



(a) Integer Queue (SPECint2000) (b) Floating Point Queue (SPECfp2000)

Fig. 3. Accumulative distribution of allocated entries in the integer queue (using the SPECint2000 programs) and in the floating point queue (using the SPECfp2000 programs) with respect to the amount of total in-flight instructions for a processor able to maintain up to 2048 in-flight instructions.

This data corresponds to a processor able to maintain up to 2048 in-flight instructions and having 500-cycle memory access latency. For example, in floating point applications, 50% of the time there are 1600 or less in-flight instructions, requiring 400 floating point queue entries. On average, the amount of entries allocated in the instruction queues is much smaller than the amount of in-flight instructions. However, to cope with over 90% of the scenarios the processor is going to face, the integer queue requires 300 entries and the floating point queue requires 500 entries, which is definitely going to affect the cycle time [14].

Fortunately, not all instructions behave in the same way. Blocked-long instructions represent by far the largest fraction of entries allocated in the instruction queues. These instructions are dependent on long-latency loads or on their dependents. Since these instructions take a very long time to even get issued for execution, maintaining them in the instruction queues just takes away issue slots from other instructions that will be executed more quickly. Multilevel queues can be used to track this type of instructions, delegating their handling to slower, but larger and less complex structures. Some previous studies have proposed such multilevel queues [8, 2], but they require a wake-up and select logic which might be on the critical path, thus potentially affecting the cycle time.

The kilo-instruction processor deals with this problem by using a simple secondary buffer called Slow Lane Instruction Queue (SLIQ). This queue is a FIFO-like structure that enables a simple but efficient wakeup and select process [5]. All the instructions dependent on a long-latency load are removed from the general purpose instruction queues and stored in-order in the SLIQ, freeing entries from the instruction queues that can be used by short-latency operations. Once the long-latency load finishes its execution, the dependent instructions are removed from the SLIQ and inserted back into their corresponding instruction queue, where they can start their execution. This mechanism allows to effectively implement the functionality of a large instruction queue while requiring a reduced number of entries, and thus it makes it possible to support a high number of in-flight instructions without scaling-up the instruction queues.

3.3 Load/Store Queue

Load and store instructions are inserted in the load/store queue at the same time they are inserted in the ROB. This queue takes care of memory disambiguation, that is, it guarantees that load and stores arrive to the memory in the correct order. Increasing the number of in-flight instructions also increases the number of loads and stores that should be taken into account, which can make the memory disambiguation logic a true bottleneck both in latency and power.

As for the instruction queues, the solution for this problem is using multilevel structures. Some recent works [1, 15, 16] describe such multilevel structures for performing memory disambiguation in a load/store queue containing a great amount of instructions. These works propose different filtering schemes that use two-level structures for storing most or all instructions in a big structure, while a smaller structure is used to easily check the dependencies.

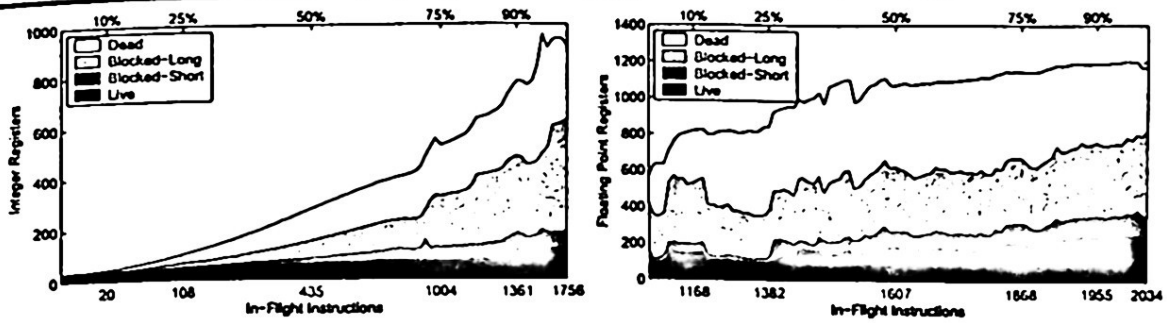
3.4 Physical Register File

Each instruction that generates a result uses a physical register to store it. Therefore, maintaining thousands of in-flight instructions involves that a high amount of physical registers is required. This high amount of registers increases the register file access time, especially taking into account the large number of access ports needed by this structure to implement an efficient issue mechanism. Nevertheless, since the physical register file is a critical component of superscalar processors, increasing its access time will surely involve an increase in the processor cycle time.

In order to reduce the number of physical register needed, the kilo-instruction processor relies on early register release and late register allocation. Figure 4 shows the accumulative distribution of allocated integer registers (SPECint2000 programs) and floating point registers (SPECfp2000 programs) with respect to the amount of total in-flight instructions. This data is provided for a machine able to maintain up to 2048 in-flight instructions and having 500-cycle memory access latency.

Registers are classified in four categories. Live registers contain values currently in use. Blocked-short and blocked-long registers have been allocated during rename, but are blocked because the corresponding instructions are waiting for the execution of predecessor instructions. Blocked-short registers are waiting for instructions that will issue shortly, while blocked-long registers are waiting for long-latency instructions. Finally, dead registers are no longer in use, but they are still allocated because the corresponding instructions have not yet committed.

It is clear that blocked-long and dead registers constitute the largest fraction of allocated registers. Some previous proposals describe how to made these registers available to other instructions, reducing the total amount of physical registers needed. In order to avoid blocked-long registers, the assignment of physical registers can be delayed using virtual tags [11]. These virtual register mapping keeps track of the rename dependencies, making unnecessary the assignment of



(a) Integer Registers (SPECint2000) (b) Floating Point Registers (SPECfp2000)

Fig. 4. Accumulative distribution of allocated integer registers (using the SPECint2000 programs) and floating point registers (using the SPECfp2000 programs) with respect to the amount of total in-flight instructions for a processor able to maintain up to 2048 in-flight instructions.

a physical register to an instruction until it starts execution. Dead registers can also be eliminated by using mechanisms for early register recycling [12]. These mechanisms release a physical register when it is possible to guarantee that it will not be used again, regardless the corresponding instruction has committed or not.

The kilo-instruction architecture combines these two techniques with checkpointing, leading to an aggressive register recycling mechanism that we call ephemeral registers [4, 10]. This is the first proposal that integrates both a mechanism for delayed register allocation and early register release and analyzes the synergy between them. The combination of these two techniques with checkpointing allows the processor to non-conservatively deallocate registers, making it possible to support thousands of in-flight instructions without requiring an excessive number of registers.

4 Real Performance

Figure 5 provides some insight about the performance achievable by the kilo-instruction processor. It shows the average performance of a kilo-instruction processor executing the SPECint2000 floating point applications. The kilo-instruction processor modeled is able to support up to 2048 in-flight instructions, but it uses just 128-entry instruction queues. It also uses 32KB separate instruction and data caches as well as an unified 1MB second level cache. The figure is divided into three zones, each of them comprising the results for 100, 500, and 1000 cycles of main memory access latency. Each zone is composed of three groups of two bars, corresponding to 512, 1024, and 2048 virtual registers or tags [11]. The two bars of each group represent the performance using 256 or 512 physical registers.

In addition, each zone of the figure has two lines which represent the performance obtained by a baseline superscalar processor, able to support up to 128 in-flight instructions, and a limit unfeasible microarchitecture where all

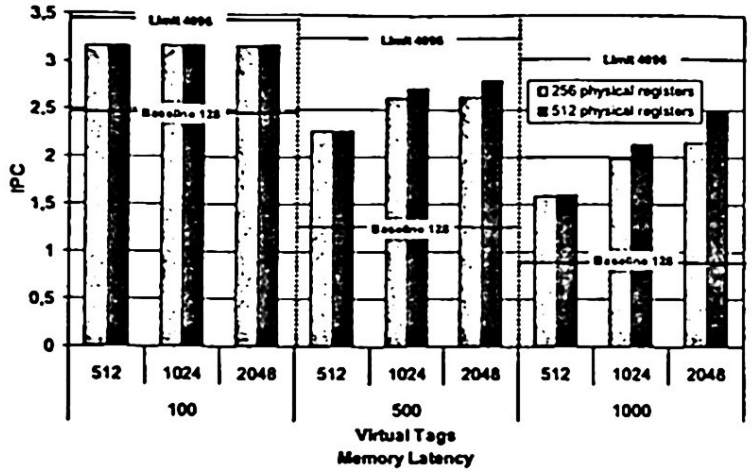


Fig. 5. Average performance results of the kilo-instruction processor executing the SPECfp2000 programs with respect to the amount of virtual registers, the memory latency, and the amount of physical registers.

the resources have been up-sized to allow up to 4096 in-flight instructions. The main observation is that the kilo-instruction processor provides important performance improvements over the baseline superscalar processor. Using 2048 virtual tags, the kilo-instruction processor is more than twice faster than the baseline when the memory access latency is 500 cycles or higher. Moreover, a kilo-instruction processor having 1000 cycles memory access latency is only a 5% slower than the baseline processor having a memory access latency 10 times lower.

These results show that the kilo-instruction processor is an effective way of approaching the unimplementable limit machine in an affordable way. However, there is still room for improvement. The distance between the kilo-instruction processor performance and the limit machine is higher for larger memory access latencies. This causes that, although the performance results for more aggressive setups nearly saturate for a memory access latency of 100 or 500 cycles, the growing trend is far from saturating when the memory access latency is 1000 cycles. This makes us believe that a more aggressive machine, able to support a higher number of in-flight instructions, will still provide a better performance.

5 Related Work

The first step in the design of our kilo-instruction architecture [3] was using checkpointing as an efficient way to control and manage the use of critical resources inside the processor. We propose to checkpoint critical long-latency instructions, which allows to create a very large virtual ROB, while actually using a small physical one. This multi-checkpointing mechanisms allows to release physical registers early and to remove load instructions early from the load/store queue. In addition, the multi-checkpointing mechanism is used to release instructions from the ROB early, which leads to an architecture where the classical ROB is essentially unnecessary [5].

Cherry [9] is another checkpointing scheme that was developed in parallel with the kilo-instruction processor. Instead of using a multi-checkpointing mechanism, Cherry is based on a single checkpoint outside the ROB. The ROB is divided in two regions: the region occupied by speculative instructions and the region occupied by non-speculative instructions. Cherry is able to release registers and load/store queue entries early in the ROB area not subject to misspeculation, providing precise exception handling using the checkpoint. On the other hand, the instructions belonging to the region subject to misspeculation (like speculative instructions after a non-resolved branch prediction) still depend on the reorder buffer to recover the correct state in case of misspecualtion, and so they are not able to release their corresponding resources.

A later proposal based in checkpointing is runahead execution [13], which follows the conceptual path of [9]. This technique creates a checkpoint of the architectural state when the head of the reorder buffer is reached by a load that has missed in the second level cache. In addition, the processor start executing instructions in a special mode using a bogus result for the load. When the load instruction actually completes, the processor returns to the normal mode, restoring the checkpoint. The first execution provides useful knowledge, like accurate data and instruction prefetches, that improves the performance during the second execution.

6 Conclusions

Tolerating large memory access latencies is a key topic in the design of future processors. Maintaining a high amount of in-flight instructions is an effective mean for overcoming this problem. However, increasing the number of in-flight instructions requires up-sizing several processor structures, which is is impractical due to power consumption, area, and cycle time limitations. The kilo-instruction processor is an affordable architecture able to support thousands of in-flight instructions. Our architecture relies on an intelligent use of the processor resources, avoiding the scalability problems caused by an excessive increase in the size of the critical processor structures. The ability of maintaining a high number of in-flight instructions makes the kilo-instruction processor an efficient architecture for dealing with future memory latencies, being able to achieve a high performance even in the presence of large memory access latencies.

Acknowledgements

This research has been supported by CICYT grant TIC-2001-0995-C02-01, the European Network of Excellence on High-Performance Embedded Architecture and Compilation (HIPEAC), and CEPBA. O. J. Santana is also supported by Generalitat de Catalunya grant 2001FI-00724-APTIND. Special thanks go to Francisco Cazorla, Ayose Falcón, Marco Galluzzi, Josep Llosa, José F. Martínez, Daniel Ortega, and Tanausú Ramírez for their contribution to the kilo-instruction processors.

References

1. H. Akkary, R. Rajwar, and S. T. Srinivasan. Checkpoint processing and recovery: towards scalable large instruction window processors. *Proceedings of the 36th International Symposium on Microarchitecture*, 2003.
2. E. Brekelbaum, J. Rupley, C. Wilkerson, and B. Black. Hierarchical scheduling windows. *Proceedings of the 35th International Symposium on Microarchitecture*, 2002.
3. A. Cristal, M. Valero, A. Gonzalez, and J. Llosa. Large virtual ROBs by processor checkpointing. *Technical Report UPC-DAC-2002-39*, Departament d'Arquitectura de Computadors, Universitat Politècnica de Catalunya, 2002.
4. A. Cristal, J. F. Martinez, J. Llosa, and M. Valero. Ephemeral registers with multicheckpointing. *Technical Report UPC-DAC-2003-51*, Departament d'Arquitectura de Computadors, Universitat Politècnica de Catalunya, 2003.
5. A. Cristal, D. Ortega, J. Llosa, and M. Valero. Out-of-order commit processors. *Proceedings of the 10th International Symposium on High-Performance Computer Architecture*, 2004.
6. M. Galluzzi, V. Puente, A. Cristal, R. Beivide, J. A. Gregorio, and M. Valero. A first glance at kilo-instruction based multiprocessors. *Proceedings of Computing Frontiers*, 2004.
7. W. M. Hwu and Y. N. Patt. Checkpoint repair for out-of-order execution machines. *Proceedings of the 14th International Symposium on Computer Architecture*, 1987.
8. A. Lebeck, J. Koppanalil, T. Li, J. Patwardhan, and E. Rotenberg. A large, fast instruction window for tolerating cache misses. *Proceedings of the 29th International Symposium on Computer Architecture*, 2002.
9. J. F. Martinez, J. Renau, M. Huang, M. Prvulovic, and J. Torrellas. Cherry: checkpointed early resource recycling in out-of-order microprocessors. *Proceedings of the 35th International Symposium on Microarchitecture*, 2002.
10. J. F. Martinez, A. Cristal, M. Valero, and J. Llosa. Ephemeral registers. *Technical Report CSL-TR-2003-1035*, Cornell Computer Systems Lab, 2003.
11. T. Monreal, A. Gonzalez, M. Valero, J. Gonzalez, and V. Viñals. Delaying physical register allocation through virtual-physical registers. *Proceedings of the 32nd International Symposium on Microarchitecture*, 1999.
12. M. Moudgill, K. Pingali, and S. Vassiliadis. Register renaming and dynamic speculation: an alternative approach. *Proceedings of the 26th International Symposium on Microarchitecture*, 1993.
13. O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt. Runahead execution: an alternative to very large instruction windows for out-of-order processors. *Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, 2003.
14. S. Palacharla, N. P. Jouppi, and J. E. Smith. Complexity-effective superscalar processors. *Proceedings of the 24th International Symposium on Computer Architecture*, 1997.
15. I. Park, C. Ooi, and T. Vijaykumar. Reducing design complexity of the load/store queue. *Proceedings of the 36th International Symposium on Microarchitecture*, 2003.
16. S. Sethumadhavan, R. Desikan, D. Burger, C. Moore, and S. Keckler. Scalable hardware memory disambiguation for high ILP processors. *Proceedings of the 36th International Symposium on Microarchitecture*, 2003.