Maintaining Thousands of In-Flight Instructions

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. Superscalar processors tolerate long-latency memory operations by maintaining a high number of in-flight instructions. Since the gap between processor and memory speed continues increasing every year, the number of in-flight instructions needed to support the large memory access latencies expected in the future should be higher and higher. However, scaling-up the structures required by current processors to support such a high number of in-flight instructions is impractical due to area, power consumption, and cycle time constraints.
The kilo-instruction processor is an affordable architecture able to tolerate the memory access latency by supporting thousands of in-flight instructions. Instead of simply up-sizing the processor structures, the kilo-instruction architecture relies on an efficient multi-checkpointing mechanism. Multi-checkpointing leverages a set of techniques like multi-level instruction queues, late register allocation, and early register release. These techniques emphasize the intelligent use of the available resources, avoiding scalability problems in the design of the critical processor structures. Furthermore, the kilo-instruction architecture is orthogonal to other architectures, like multi-processors and vector processors, which can be combined to boost the overall processor performance.

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, this trend generates two important problems that processor designers should face up. On the one hand, long pipelines increase the distance between branch prediction and branch resolution. The longer a pipeline is, the higher amount of speculative work that should be discarded in case of a branch misprediction. In this context, an accurate branch predictor becomes a critical processor component. On the other hand, the higher clock frequency increases the main memory access latency. Since the DRAM technology improves at a speed much lower than the microprocessor technology, each increase in the processor clock frequency causes that a higher number of processor cycles are required to access

M. Danelutto, D. Laforenza, M. Vanneschi (Eds.): Euro-Par 2004, LNCS 3149, pp. 9–20, 2004.

the main memory, degrading the potential performance achievable with the clock frequency improvement.

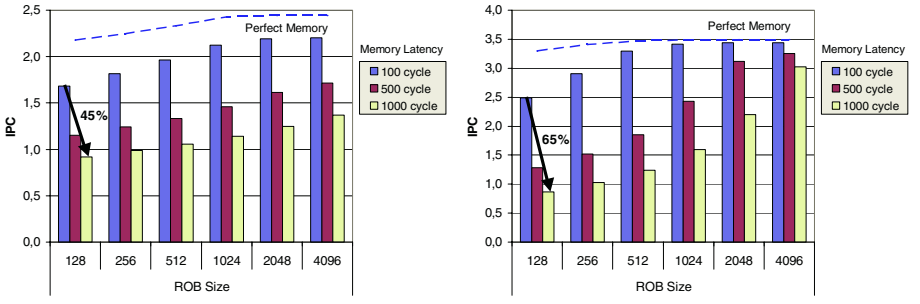
If the main memory access latency increase continues, it will be a harmful problem for future microprocessor technologies. Therefore, dealing with the gap between the processor and the memory speed is vital in order to allow high-frequency microprocessors to achieve all their potential performance. A 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 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.

A processor able to maintain thousands in-flight instructions can overlap the latency of a load instruction that access to the main memory with the execution of subsequent independent instructions, that is, the processor can hide the main memory access latency by executing useful work. However, maintaining a high number of in-flight instructions requires scaling-up critical processor structures like the reorder buffer, the physical register file, the instruction queues, and the load/store queue. This is not affordable due to cycle time limitations. The challenge is to develop an architecture able to support a high number of in-flight instructions while avoiding the scalability problems involved by such a big amount of in-flight instructions.

2 Increasing the Number of In-Flight Instructions

Figure 1 shows an example of the impact of increasing the maximum number of in-flight instructions supported by a four 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 the increase in the main memory latency 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%. Nevertheless, a higher number of in-flight instructions mitigates this effect. 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.

These results show that increasing the number of in-flight instructions is an effective way of tolerating large memory access latencies. On average, executing floating point programs, a processor able to maintain up to 4096 in-flight instructions having 1000-cycle memory latency performs 22% better than a processor having 100-cycle latency but only being able to maintain up to 128 in-flight instructions. Executing integer programs, the processor supporting 4096 in-flight instructions cannot achieve a better performance than the processor support-



(a) SPECint2000 (b) SPECfp2000

Fig. 1. Average performance of a 4-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.

ing 128 in-flight instructions. However, it is only 18% slower despite having a memory latency 10 times larger. The improvement achieved in floating point applications is due to the higher instruction-level parallelism available in these programs. The amount of correct-path instructions that can be used to overlap a long-latency memory operation in integer programs is limited by the presence of chains of instructions dependent on long-latency loads, especially other long-latency loads (i.e. chasing pointers), as well as the higher branch misprediction rate. Nevertheless, the continuous research effort devoted to improve branch prediction suggests that integer programs can achieve an improvement as high as floating point programs in a near future. Moreover, the higher number of in-flight instructions also enables different optimizations, like instruction reuse and value prediction, which can provide even better performance.

Therefore, high-frequency 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 (ROB), the physical register file, the general purpose instruction queues (integer and floating point ones), and the load/store queue. Every decoded instruction requires an entry in the ROB, which is not released until the instruction commits. Every decoded instruction also requires an entry in the corresponding instruction queue, but it is released when the instruction issues for execution. The load/store queue has a different behavior. An entry in this queue is not released until it can be assured that no previous instruction will modify the contents of the memory address that will be accessed by the load or store instruction. In addition, store instructions cannot release their entries until commit because the memory hierarchy must not be updated until the execution correctness is guaranteed. Finally, every renamed instruction that generates a result requires a physical register, which is not released until it can be assured that the register value will not be used by a later instruction, that is, when the commit stage is reached by a new instruction that overwrites the contents of the registers.

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 [15]. 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 [4] 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 excessive 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

Checkpointing is a well established and used technique [8]. The main idea is to create a checkpoint at specific instructions of the program being executed. A checkpoint can be though of as a snapshot of the state of the processor, which contains all the information required to recover the architectural state and restart execution at that point. Several recent proposals are based on checkpointing. Cherry [10] uses a single checkpoint outside the ROB. The ROB is divided into a region occupied by speculative instructions, which can use the checkpoint for recovery, and a region occupied by non-speculative instructions, which depend on the ROB for recovery. Runahead execution [14] also creates a single checkpoint when the head of the ROB is reached by a load that has missed in the second level cache, allowing to speculatively execute the following instructions in order to perform data and instruction prefetches.

Instead of using a single checkpoint, the kilo-instruction architecture relies on a multi-checkpointing mechanism. Figure 2 shows an example of our checkpointing process [6]. 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 older checkpoint among the two is removed (timeline D).

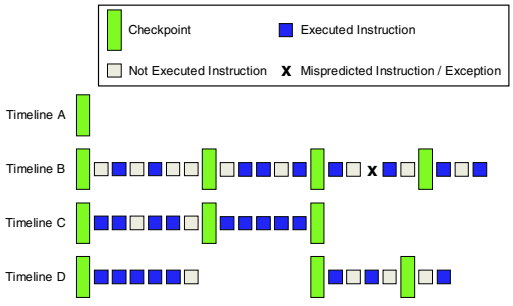


Fig. 2. The checkpointing process performed by the kilo-instruction processor.

Our multi-checkpointing mechanism has been designed as an efficient way to control and manage the use of critical resources inside the processor [3]. The novelty of this mechanism is that the kilo-instruction architecture uses checkpointing to allow an early release of resources. The multi-checkpointing mechanism enables the possibility of committing instructions out-of-order, allowing to early release ROB entries, which can lead to an architecture where the classical ROB is essentially unnecessary [6]. Early release can also be applied to physical registers, reducing the number of required registers. In addition, multi-checkpointing makes it possible to early manage the entries of the instruction queues, implementing a two-level structure. In summary, the multi-checkpointing mechanism is the key technique that makes our kilo-instruction architecture affordable.

3.2 Out-of-Order Commit

In a superscalar out-of-order processor, all instructions are inserted in the reorder buffer (ROB) after they are fetched and decoded. The ROB keeps a history window of all in-flight instructions, allowing for the precise recovery of the program state at any point. Instructions are removed from the ROB when they commit, that is, when they finish executing and update the architectural state of the processor. However, to assure that precise recovery is possible, the instructions should be committed in the program order.

In-order commit is a serious problem in the presence of large memory access latencies. Let us suppose that a processor has a 128-entry ROB 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 becomes the older instruction in 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. However, scaling-up the number of ROB entries is impractical, mainly due to cycle time limitations.

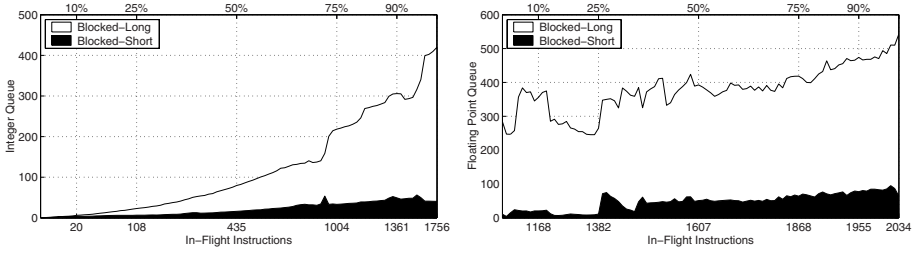
The kilo-instruction architecture solves this problem by using the multi-checkpointing mechanism for enabling out-of-order commit [3, 6]. The presence of a previous checkpoint causes that in-order commit is not required for preserving correctness and exception preciseness. If an exception occurs, the processor returns to the previous checkpoint and restarts execution. Indeed, the ROB itself becomes unnecessary. In spite of this, we keep a ROB-like structure in our kilo-instruction processor design. This structure, which we call pseudo-ROB [6], has the same functionality of a ROB. However, the instructions that reach the head of the pseudo-ROB are removed at a fixed rate, not depending on their state. Since the processor state can be recovered from the pseudo-ROB, generating a checkpoint is only necessary when the instructions leave the pseudo-ROB. Delaying checkpoint generation is beneficial to reduce the impact of branch mispredictions. Over 90% of mispredictions are caused by branches that are still inside the pseudo-ROB. This means that most branch mispredictions do not need to roll back to the previous checkpoint for recovering the correct state, minimizing the misprediction penalty. In this way, the combination of the multi-checkpointing and the pseudo-ROB allows to implement the functionality of a large ROB without requiring an unimplementable centralized structure with thousands of entries.

3.3 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 it is issued for execution. 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. The main observation is that, in a processor able to support up to 2048 in-flight instructions, the instruction queues need a high number of entries. 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 [15].

Fortunately, not all instructions behave in the same way. Instructions are divided in two groups: 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, like a load instruction that misses in the second level cache. Blocked-long instructions represent by far the largest fraction of entries allocated in the instruction queues. 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 [9, 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 also takes advantage of the different waiting times of the instructions in the queues, but doing it in an affordable way. First



(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 example, in floating point applications, 50% of the time there are 1600 or less in-flight instructions, requiring 400 floating point queue entries. Data presented corresponds to a processor able to maintain up to 2048 in-flight instructions and having 500-cycle memory access latency.

of all, our mechanism detects those instructions that will take a very long time to get issued for execution. The presence of the pseudo-ROB is beneficial for this detection process, since it allows to delay the decision of which instructions will require a long execution time until it can be effectively known. This not only increases the accuracy of long-latency instruction detection, but also greatly reduces the complexity of the logic required.

Long-latency instructions are removed from the general purpose instruction queues and stored in-order in a secondary buffer, which allows to free entries from the instruction queues that can be used by short-latency operations. The secondary buffer is a simple FIFO-like structure that we call Slow Lane Instruction Queue (SLIQ) [6]. The instructions stored in the SLIQ will wait until there is any need for them to return to the respective instruction queue. When the long-latency operation that blocked the instructions finishes, they are removed from the SLIQ and inserted back into their corresponding instruction queue, where they can be issued for 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.

3.4 Load/Store Queue

Load and store instructions are inserted in the load/store queue at the same time they are inserted in the ROB. The main objective of this queue is to guarantee that all load and store instructions reach the memory system in the correct program order. For each load, it should be checked if an earlier store has been issued to the same physical address, and thus use the value produced by the store. For each store, it should be checked if a later load to the same physical address has been previously issued, and thus take corrective actions. Maintaining a high number of in-flight instructions involves an increase in the number of loads and

stores that should be taken into account, which can make the load/store queue a true bottleneck both in latency and power.

Some solutions have been proposed for this problem. In [17], the load/store queue scalability is improved by applying approximate hardware hashing. A Bloom filter predictor is used to avoid unnecessary associative searches for memory operations that do not match other memory operations. This predictor is also used to separate load/store queue partitions, reducing the number of partitions that should be looked up when a queue search is necessary. Another approach is using multilevel structures [1, 16]. 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.

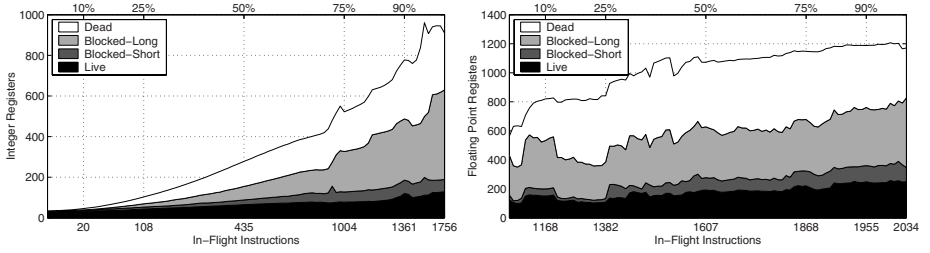
3.5 Physical Register File

A great amount of physical registers is required to maintain thousands of in-flight instructions. Figure 4 shows the accumulative distribution of allocated integer registers (for SPECint2000 programs) and floating point registers (for SPECfp2000 programs) with respect to the amount of total in-flight instructions. To cope with over 90% of the scenarios the processor is going to face, almost 800 registers are required for the integer programs and almost 1200 are required for the floating point programs. Such a large register file will be impractical not only due to area and power limitations, but also because it requires a high access time, which will surely involve an increase in the processor cycle time.

In order to reduce the number of physical registers needed, the kilo-instruction processor relies on the different behaviors observed in the instructions that use a physical register. 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. In order to avoid blocked-long registers, the assignment of physical registers can be delayed using virtual tags [12]. These virtual register mapping keeps track of the rename dependencies, making unnecessary the assignment of a physical register to an instruction until it starts execution. Dead registers can also be eliminated by using mechanisms for early register recycling [13]. 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 the multi-checkpointing mechanism, leading to an aggressive register recycling mechanism that we call ephemeral registers [5, 11]. This is the first proposal that integrates both a mechanism for delayed register allocation and early register



(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. Data presented corresponds to a processor able to maintain up to 2048 in-flight instructions and having 500-cycle memory access latency.

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 4-wide processor executing the SPEC2000 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 [12]. 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 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,

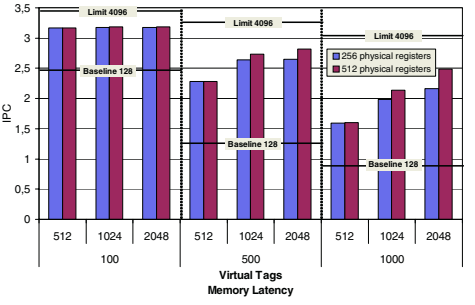


Fig. 5. Average performance results of the kilo-instruction processor executing the SPECfp2000 programs with respect to the amount of virtual registers, the memory latency, and the amount of physical registers.

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 trend suggests that a more aggressive machine, able to support a higher number of in-flight instructions, will provide even a better performance.

5 Current Research Lines

The kilo-instruction architecture is not only an efficient technique for tolerating the memory access latency. It is also a flexible paradigm able to cross-pollinate with other techniques, that is, it can be combined with other mechanisms to improve its capabilities and boost the processor performance.

The kilo-instruction architecture can be combined with multiprocessors or simultaneous multithreaded processors. In particular, the kilo-instruction multiprocessor architecture [7] uses kilo-instruction processors as computing nodes for building small-scale CC-NUMA multiprocessors. This kind of multiprocessor system provide good performance results, showing two interesting behaviors. First, the great amount of in-flight instructions makes it possible for the system not just to hide the latencies coming from local and remote memory accesses but also the inherent communication latencies involved in preserving data coherence. Second, the significant pressure imposed by many in-flight instructions translates into a very high contention for the interconnection network. This fact remarks the need for better routers capable of managing high traffic levels, dictating a possible way for building future shared-memory multiprocessor systems.

Other interesting research line is the combination of the kilo-instruction architecture with vector processors. Vector architectures require high bandwidth memory systems to feed the vector functional units. The ability of bringing a high amount of data from memory reduces the total number of memory accesses, also reducing the impact of large memory access latencies. However, this

improvement is limited to the vector part of a program. The performance of the scalar part will still be degraded due to the presence of a large memory access latency. Our objective is to use a kilo-instruction processor to execute the scalar part of vector programs, removing this limitation and thus providing a great improvement in the performance of vector processors.

We are also analyzing the combination of kilo-instruction processors with other techniques, which were previously proposed for processors with small instruction windows. Combining value prediction with kilo-instruction processors makes it possible to predict the results of long-latency memory operations. This technique allows to break the dependencies between long-latency loads and other instructions, increasing the available instruction-level parallelism, and thus improving the kilo-instruction processor ability of tolerating the memory access latency. Kilo-instruction processors can also be combined with mechanisms for detecting the reconvergence point after branch instructions. Correctly identifying the control independent instructions will enable the possibility of reusing independent instructions after branch mispredictions, alleviating the misprediction penalty. A different approach for avoiding the branch misprediction penalty is combining the kilo-instruction processor with techniques for multi-path execution. In general, the possibility of combining the kilo-instruction architecture with other techniques in an orthogonal way creates a great amount of new appealing ideas for future research.

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 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. *Procs. of the 36th Intl. Symp. on Microarchitecture*, 2003.
2. E. Brekelbaum, J. Rupley, C. Wilkerson, and B. Black. Hierarchical scheduling windows. *Procs. of the 35th Intl. Symp. 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, D. Ortega, J. Llosa, and M. Valero. Kilo-instruction processors. *Procs. of the 5th Intl. Symp. on High Performance Computing*, 2003.
5. 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.
6. A. Cristal, D. Ortega, J. Llosa, and M. Valero. Out-of-order commit processors. *Procs. of the 10th Intl. Symp. on High-Performance Computer Architecture*, 2004.
7. M. Galluzzi, V. Puente, A. Cristal, R. Beivide, J. A. Gregorio, and M. Valero. A first glance at kilo-instruction based multiprocessors. *Procs. of the 1st Conf. on Computing Frontiers*, 2004.
8. W. M. Hwu and Y. N. Patt. Checkpoint repair for out-of-order execution machines. *Procs. of the 14th Intl. Symp. on Computer Architecture*, 1987.
9. A. Lebeck, J. Koppanalil, T. Li, J. Patwardhan, and E. Rotenberg. A large, fast instruction window for tolerating cache misses. *Procs. of the 29th Intl. Symp. on Computer Architecture*, 2002.
10. J. F. Martinez, J. Renau, M. Huang, M. Prvulovic, and J. Torrellas. Cherry: checkpointed early resource recycling in out-of-order microprocessors. *Procs. of the 35th Intl. Symp. on Microarchitecture*, 2002.
11. J. F. Martinez, A. Cristal, M. Valero, and J. Llosa. Ephemeral registers. *Technical Report CSL-TR-2003-1035*, Cornell Computer Systems Lab, 2003.
12. T. Monreal, A. Gonzalez, M. Valero, J. Gonzalez, and V. Viñals. Delaying physical register allocation through virtual-physical registers. *Procs. of the 32nd Intl. Symp. on Microarchitecture*, 1999.
13. M. Moudgill, K. Pingali, and S. Vassiliadis. Register renaming and dynamic speculation: an alternative approach. *Procs. of the 26th Intl. Symp. on Microarchitecture*, 1993.
14. O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt. Runahead execution: an alternative to very large instruction windows for out-of-order processors. *Procs. of the 9th Intl. Symp. on High-Performance Computer Architecture*, 2003.
15. S. Palacharla, N. P. Jouppi, and J. E. Smith. Complexity-effective superscalar processors. *Procs. of the 24th Intl. Symp. on Computer Architecture*, 1997.
16. I. Park, C. Ooi, and T. Vijaykumar. Reducing design complexity of the load/store queue. *Procs. of the 36th Intl. Symp. on Microarchitecture*, 2003.
17. S. Sethumadhavan, R. Desikan, D. Burger, C. Moore, and S. Keckler. Scalable hardware memory disambiguation for high ILP processors. *Procs. of the 36th Intl. Symp. on Microarchitecture*, 2003.