A Decoupled KILO–Instruction Processor

Miquel Pericàs[†⋆], Adrian Cristal[†⋆], Ruben González[†], Daniel A. Jiménez[‡] and Mateo Valero[†⋆]

[†]Departament d'Arquitectura de Computadors, Universitat Politècnica de Catalunya
{mpericas,adrian,gonzalez,mateo}@ac.upc.edu

[⋆]Barcelona Supercomputing Center

[‡]Department of Computer Science, Rutgers University
djimenez@acm.org

Abstract

Building processors with large instruction windows has been proposed as a mechanism for overcoming the memory wall, but finding a feasible and implementable design has been an elusive goal. Traditional processors are composed of structures that do not scale to large instruction windows because of timing and power constraints. However, the behavior of programs executed with large instruction windows gives rise to a natural and simple alternative to scaling. We characterize this phenomenon of execution locality *and propose a microarchitecture to exploit it to achieve the benefit of a large instruction window processor with low implementation cost. Execution locality is the tendency of instructions to exhibit high or low latency based on their dependence on memory operations. In this paper we propose a decoupled microarchitecture that executes low latency instructions on a* Cache Processor *and high latency instructions on a* Memory Processor. *We demonstrate that such a design, using small structures and many in-order components, can achieve the same performance as much more aggressive proposals while minimizing design complexity.*

1 Introduction

The most important impediment to improving single-threaded processor performance is the memory wall [1]. Due to improvements in process technology and microarchitecture, modern microprocessors are clocked in the gigahertz range and can reach peak performances of several billion instructions per second. Unfortunately, improvements in memory systems have not kept pace with improvements in microprocessors. As these rates of improvement continue to diverge, we reach a point where some instructions can execute in just a few cycles while others may take hundreds or even thousands of cycles because they depend on uncached data. Critical high-latency instructions caused by cache misses can slow a processor well below its peak potential.

We present an efficient and practical proposal to address this problem. We observe that programs exhibit *execution locality*. That is, instructions tend to exhibit regular behavior in terms of their ability to be classified as either high or low latency depending on their dependence on uncached data. Just as caches were proposed to exploit data locality, we propose a decoupled Cache/Memory microarchitecture to exploit execution locality. One pipeline, the out-of-order Cache Processor (CP), handles low latency instructions and exploits as much traditional instruction-level parallelism (ILP) as possible. A second pipeline, the in-order Memory Processor (MP), handles high latency instructions and enables memory-level parallelism (MLP). The two pipelines are connected by a simple queue that gives the effect of a large instruction window without the need for a large content-addressable memory (CAM). Our decoupled approach fully exploits the parallelism available in the instruction stream by maintaining an effective window of thousands of in-flight instructions with much less complexity than competing proposals.

1.1 Our Contribution

Rather than address the limitations of each processor structure, we exploit the behavior of programs with respect to long memory latencies to produce a holistic approach to building a large instruction window processor.

This analysis allows establishing an important relationship between the memory hierarchy and the number of cycles instructions must wait for issue. This relationship gives rise to the new concept of execution locality, enabling us to build a new decoupled architecture that has many design advantages over a centralized architecture.

Config	L1 access time	L1 size	L2 access time	L2 size	memory access time
L1-2	2	∞	-	-	-
L2-11	2	32KB	11	∞	-
L2-21	2	32KB	21	∞	-
MEM-100	2	32KB	11	512KB	100
MEM-400	2	32KB	11	512KB	400
MEM-1000	2	32KB	11	512KB	1000

Table 1. Configurations for quantifying the effect of memory wall

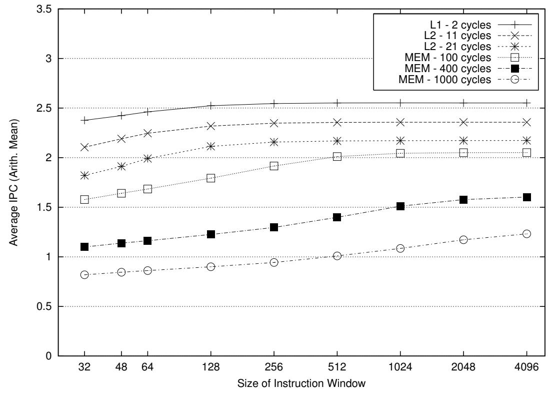

Figure 1. Effects of memory subsystem on SpecINT

2 Execution limits and Execution Locality

The memory wall itself is not necessarily a limitation to the ability to exploit instruction-level parallelism. Processor characteristics and program characteristics play an important role in the effect of the memory wall on execution.

We quantify this effect by observing the impact of several memory subsystems on a range of out-of-order cores. Out-of-order execution is necessary to evaluate the ability of these cores to hide latencies of independent instructions, usually loads. The resources of all out-of-order cores evaluated are sized such that stalls can only occur due to shortage of entries in the ROB. Thus, providing the number of ROB entries is enough to describe these 4-way speculative processors. Using SPEC2000 as the workload, six different memory subsystems are evaluated for IPC. Table 1 details their configurations. In this table, memory access times are given in processor clock cycles.

Figures 1 and 2 show the effects on IPC of using these six memory subsystems. In these figures *Size of Instruction Window* is the same size as the Reorder Buffer.

An analysis of SpecFP benchmarks shows that even for the slowest memory subsystems it is possible to recover the lost IPC simply by scaling the processor to support thousands of in-flight instructions. With an ROB of 4K entries

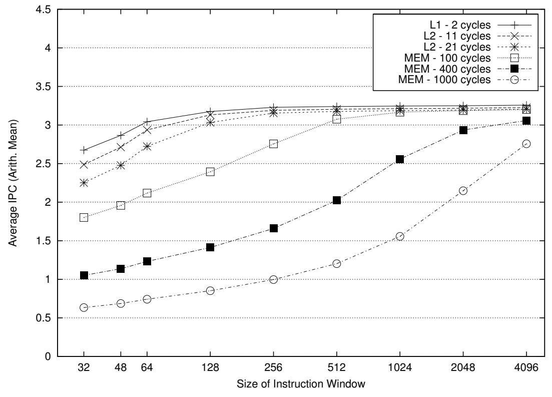

Figure 2. Effects of memory subsystem on SpecFP

almost all architectures perform similarly to the perfect L1 cache configuration. The reason behind this IPC recovery is that load misses are not on the critical path on SpecFP when enough instructions can be kept in–flight.

For SpecINT benchmarks the analysis is somewhat different. These workloads sometimes misbehave in two ways that can put a high latency load on the critical path: pointer chasing behavior and branch mispredictions depending on uncached data. The latter will force a complete squash of the instructions of the processor with a devastating effect on performance. Note that branch mispredictions depending on short latency events can be recovered from quickly and thus have little impact on IPC. Figure 1 shows how in this case recovering IPC by using large instruction windows is not an effective solution. Thus, different techniques need to be researched for these cases [2]. In any case, large-instruction windows are not detrimental to integer codes. They simply do not help as much as they do for floating point workloads.

2.1 Exploiting Execution Locality

Clearly, one way to build an architecture capable of overcoming the memory wall is to produce a chip with resources to handle thousands of in-flight instructions. This is analogous to the design methodology used for current out-of-order chips with respect to handling L1 cache miss latencies. These latencies are quite small. An L2 hit normally takes around 5-20 cycles. As new technologies have been used to implement chips, the cache distance has slowly increased. To hide this new latency, the resources on the chip (instruction queue, register file, load/store queue and reorder buffer) must be increased commensurately to maintain the previous IPC rates. This approach is feasible for dealing with increasing L1 miss latencies. However, increasing the structures more than 1000% to support thousands of in-flight instructions is totally impractical due to

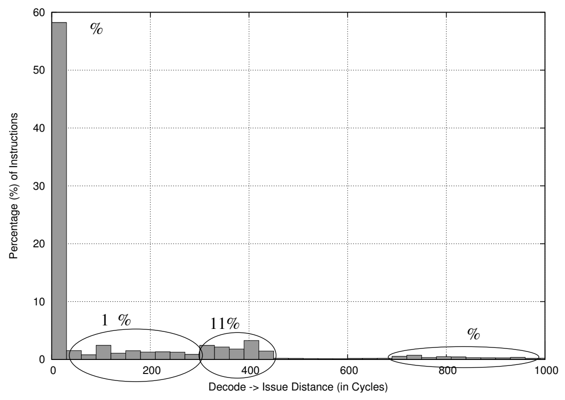

Figure 3. Average Distance between Decode and Issue/Writeback

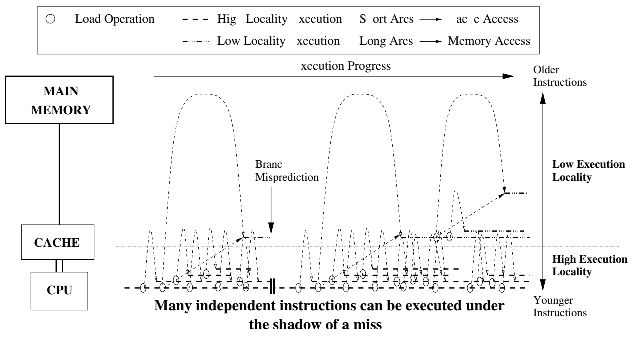

Figure 4. Execution Progress and Execution Locality

power and timing issues. Thus, most research on large-window processors has focused on replacing non-scalable resources with new resources that scale much better using a variety of techniques ranging from virtualization to hierarchical implementations, etc. Unfortunately, the design complexity of such approaches is still high.

To propose innovative designs with reduced complexity we wanted to collect more information about the execution of programs. To this end, the following instruction-centric analysis was performed. Using an out-of-order architecture with a memory latency of 400 cycles and an unlimited processor we focused on SpecFP and analyzed the average number of cycles a correct–path instruction waits in the instruction queue until it is issued. The results, shown in Figure 3, indicate that there is considerable regularity in the *issue latency* of instructions. This regularity is highly correlated with the parameters of the memory subsystem.

Several groups/peaks can be seen in the figure. Most instructions (about 70%) execute in fewer than 300 cycles while about 11% of all instructions execute around 400 cycles and about 4% execute around 800 cycles. This distribution is highly correlated with the memory access type of loads present in the instruction slices. The front-end normally advances at full speed, fetching 4 instructions per cycle. Thus the whole instruction slice is fetched in a relatively small number of cycles. We can conclude that the 70% of instructions that execute in fewer than 300 cycles are instructions that depend on a cache hit or are instructions whose source registers will all be computed in a short amount of time. The small peak around 400 cycles corresponds to instructions that depend on a single cache miss. The same applies to the small peak around 800 cycles which is made of instructions that depend on a chain of 2 cache misses. The provided numbers add up to 85%. The remaining 15% belongs to instructions where it is not clear if they depend on 1 or 2 misses, or are instructions that depend on more than 2 misses. For SpecINT applications, almost all correct–path instructions are issued shortly after decode. The reason is that long latency events are often on a mispredicted path. After recovery, long latency loads have been turned into prefetches and most of correct-path code ends up having short issue latency.

This phenomenon can be given an interesting interpretation in terms of register availability. Once an instruction enters the instruction window, its source registers can be in 1 of 4 states: 1) READY, ie, with computed value; 2) NOT READY, depending on cache events; 3) NOT READY, waiting for other instructions to writeback; and 4) NOT READY, depending on long-latency memory events. In terms of execution, cases 2 and 3 have a very similar behavior. We are now ready to establish a new classification of instructions: Those instructions that have at least one long–latency register (ie, in state 4) are classified as having *low execution locality*. The remaining instructions, which will be issued quickly, have *high execution locality*. Execution locality is a property that describes instructions as a function of the number of cycles they wait in the queues until they issue. This distance is called the *Issue Latency*. In general, instructions depending on cache misses will have low execution locality while instructions that depend only on cache hits will have high execution locality. This concept is exemplified in Figure 4 where it is shown that different execution groups are clearly disjoint. This figure also represents one of they key properties of programs which is Karkhanis' observation that *many independent instructions can be executed under the shadow of a load miss* [3]. Large window processors, such as KILO-Instruction processors [2], profit from this characteristic to hide the latencies of long-latency misses. Figure 4 also shows the detrimental effects of mispredictions depending on cache misses. Therefore, to establish a relationship between execution locality and performance it is necessary to take into account the criticality of loads. In general, loads get very critical when they drive a low-confidence branch. These cache accesses will strongly determine performance [4].

3 A Decoupled KILO-Instruction Processor

It is interesting to take a second look at Figure 4. As the processor advances execution, the gap between the youngest and the oldest instruction tends to increase. Some events, like mispredictions or stores that end long-latency slices, will reduce this gap. Two features provide the most benefit to an architecture with an unlimited window:

1. The Fetch Unit never stalls. Thus, high locality code continues to execute in the presence of several high latency loads and loads that miss can be executed early.

2. Low locality instructions can be executed in parallel with recently fetched high locality instructions.

Execution of low locality code deserves one more look. As Karkhanis *et al.* point out [3], most instructions that are fetched under the shadow of a miss are independent of it. Thus, *The amount of low locality code is small when compared to high locality code*. Most of the execution bandwidth is consumed by high locality code. Nevertheless, current architectures have to stall every time they encounter a memory access. Thus, the small amount of low locality code present in the instruction stream causes stalls that significantly reduce performance.

The following guidelines can be derived from the analysis of execution locality:

- *Never Stall in Fetch*: It is important to continue fetching instructions and executing them because a large part of the execution bandwidth will be used for short issue latency instructions. Also, this permits executing `load` operations as soon as possible. This is the same motivation behind Continual Flow Pipelines [5].

- *Large Storage is Important but a Large CAM is Not*: We will need to store many non-executable instructions during a cache miss. However, there will be plenty of time to execute them and they do not require high execution bandwidth. Therefore it is not necessary to have all of these instructions in an issuable queue based on CAM logic. This concept has been exploited before in the Waiting Instruction Buffer [6].

- *Distributed Execution*: Low execution locality code is very decoupled from high execution locality code. There is no need to communicate back values as low locality code feeds only low locality code.

3.1 A Decoupled KILO-Instruction Processor

Using these insights we will now introduce the main contribution of this paper: the *Decoupled KILO-Instruction Processor* (D-KIP). The D-KIP is the result of exercising

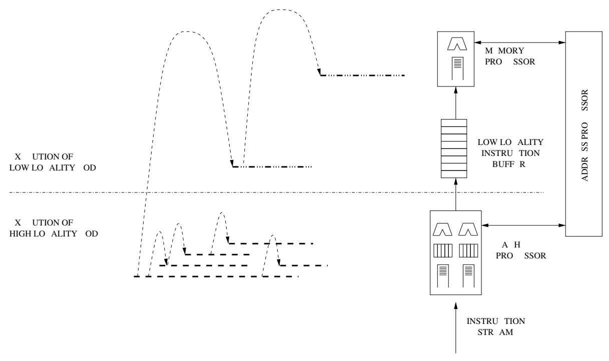
M MORY PRO SSOR
X UTION OF LOW LO ALITY OD
LOW LO ALITY INSTRU TION BUFF R
ADDR SS PRO SSOR
X UTION OF HIGH LO ALITY OD
A H PRO SSOR
INSTRU TION STR AM

Figure 5. 2-Level Decoupled Processor

the simplest implementation of the presented guidelines; we have 1) two different execution points, 2) unidirectional communication from high locality to low locality code and 3) high latency tolerance in the low locality code. The direct implementation of these ideas is to use one processor for each locality type linked by an unidirectional instruction buffer. This structure, shown in Figure 5, complies with all design guidelines.

We will now describe the details of this microarchitecture before analyzing its performance and complexity characteristics.

3.2 Implementation of a D-KIP processor

The two-level D-KIP processor is based mostly on structures for which implementations already exist, allowing it to be built reusing standard modules and focusing on the interfaces. Thus most of the work consists in adapting the structures to comply with the interfaces. In Section 5, we analyze D-KIP from a design perspective.

Figure 5 shows that the microarchitecture consists of two processors, a simple, non-issue-capable instruction buffer and an address processor that handles all memory operations. The small and fast *Cache Processor* can be efficiently implemented using the MIPS R10000 as starting point [7]. It is useful to use a register-mapped architecture because it is important to provide fast branch recovery in the front-end. Because this processor assumes a perfect L2 Cache it can be smaller than current generation processors. The *Memory Processor* can be even simpler since it does not require much execution bandwidth. Thus in this study we have modeled it using a simple Future File architecture [8].

We target the following execution model: Instructions are fetched by the Cache Processor (CP). They stay there, waiting to be executed until they are issued to the functional units or they are determined to have long issue latency, i.e., they belong to a low execution locality slice. In this case they are moved from the CP into a Low Locality Instruction Buffer (LLIB) and wait until all long-latency events they depend on have finished. There is one LLIB for floating point and another LLIB for integer instructions.

When the long-latency load completes it simply keeps the value in the address processor. When the depending instructions arrive at the head of the LLIB and the load value is available, both the instructions and the value are inserted into the corresponding Memory Processor (MP). Execution can now proceed in the MP.

The LLIB Queues do not provide global issue capabilities. In addition, they use a FIFO policy which greatly simplifies the register management.

We will now discuss some of the modifications necessary to implement the decoupled processor.

Aging-ROB. The Cache Processor contains a ROB like other processors but in addition to allowing recovery from mispredictions and exceptions, the CP's ROB is also used to determine if instructions belong to a low execution locality group or not. For this task the ROB cannot wait until writeback because this is too late.

The D-KIP proposes using a scheme known as the *Aging-ROB*, which improves and supersedes the pseudo-ROB scheme introduced in [9]. The Aging-ROB is a ROB structure in which instructions progress at a constant pace. This allows checking whether instructions are short latency or not using a timer. In general the size of the ROB will be the number of aging cycles multiplied by the commit width, which in our study is 4. The Aging-ROB is implemented as a circular FIFO with a `head` and a `tail` pointer that is moved forward at the same speed as decode but with a constant delay.

LLIB Insertion and Wake-up. The Aging-ROB will force analysis of an instruction after a certain number of cycles. This operation is called *Analyze* in our pipeline and it determines if the instruction is long–latency. There are two instruction types that behave differently. Loads are special and they are analyzed first here. When a load arrives at *Analyze* it must be determined if it has missed; thus, the ROB timer should be large enough that it can detect a miss. To detect whether a load has missed in the L2 cache it is necessary to wait until the tag array has been accessed and the hit/miss information returns. This imposes a minimal size on the ROB timer. If the load missed, then this information is recorded in a bit vector that identifies long–latency registers (the Low Locality Bit Vector, LLBV). Otherwise the register is marked as short latency. In any case, the analysis of instructions does not stall unless the current situation of the load is still unknown. When a generic (non-load) instruction arrives at *Analyze* it is first checked whether it has already executed (this information is stored in the ROB). In that case the destination register is marked as short latency. Otherwise the sources are analyzed. If one of the sources is long-latency then the instruction is also classified as long-latency and is therefore inserted in the LLIB. If none of the two situations applies, then the instruction is still in-flight, but will be executed soon. In this case the architecture stalls in the *Analyze* stage until the instruction writeback. The reason to stall and not continue at this point is due to the way checkpoints work, which we explain shortly. The impact of these stalls is minimal, averaging 0.7% IPC loss.

As instructions source long-latency registers, new registers are marked as long-latency in the LLBV. It is theoretically possible that, after a while, all registers are marked as long–latency, an undesirable situation that would not improve performance as all instructions would be processed by the potentially slower memory processor. However, it has been measured that this does not happen during steady state. There are various reasons for this:

- Checkpoint recovery restores the full state to the cache processor. This operations clears the LLBV completely.
- Short–latency operations, which represent more than 65% of all executed code (see section 2.1), will redefine registers that were marked as long–latency. After completion, the corresponding bit in the LLBV will be cleared.

Long-latency loads are executed in the address processor, where the LSQ is located. Upon completion, the load value is stored in a FIFO buffer, one per LLIB. Each entry in this FIFO is associated to a long latency load. When the first depending instruction is about to enter the Memory Processor, and the load value is available, the value is first inserted from this buffer into the Future File of the Memory Processor from where the operation will then obtain the value.

Registers. Register management is a critical issue. We want to keep a minimum number of registers while having a simple and implementable management algorithm. The D-KIP architecture provides a solution for both goals using a distributed organization of registers allowing for distributed and independent register management.

The Cache Processor does not need any modification. The traditional algorithm of freeing registers once the renaming instruction is *analyzed* can be used here. The Memory Processor uses a Future File architecture. It requires a logical register file in the front-end plus the associated space in the reservation stations. The low requirements of the MP enable it to have a very small number of reservation stations, so register management is very efficient in this part of the architecture.

The only structure that requires more attention is the LLIB since it may need to store many registers. However, the LLIB has some helpful properties. First, this

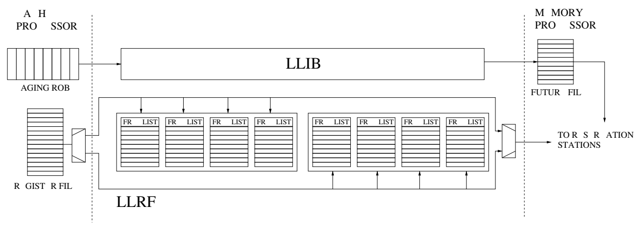

Figure 6. Schematic of the LLRF

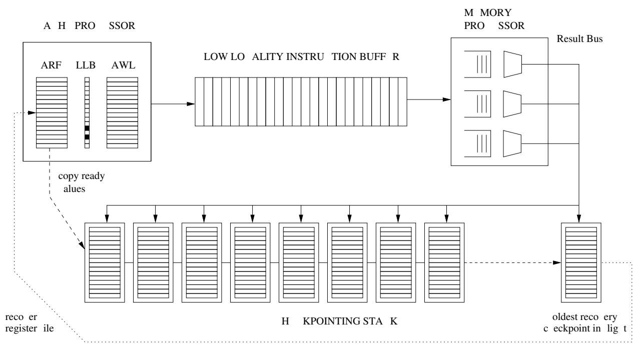

Figure 7. Schematic of checkpoints

structure is in-order, so the order in which registers are inserted/extracted is known before-hand. The LLIB register storage (called LLRF, *Low Locality Register File*) works as follows. During *Analyze* it is determined if a long-latency instruction has READY operands. These operands are then inserted into the LLRF. In the Alpha ISA, which we are using, there will never be more than one READY operand per instruction. Thus we need to preallocate at most one register per instruction. The commit width of the processor we are modeling is 4. This will require an insertion rate of 4 registers per cycle in the worst case. The same applies to the instruction extraction rate. However, the serial FIFO nature of the LLIB allows storing each register in a different bank. We model our LLRF as a banked register file with 8 banks. LLIB Insertion and LLIB Extraction always operates on a disjoint group of 4 banks. If a value to be read happens to be in a bank that is being written, the reading instruction is simply stalled for one cycle. This avoids conflicts in their access and allows to implement these banks as single-ported structures. The result is that each bank occupies minimal area and has minimal size. We calculated that the data array would be 6.6 times smaller than a centralized register file with 4 read ports and 4 write ports and the same number of entries [10]. Each bank has a free list that works independently of the other banks. The instruction in the LLIB records the position of the READY operand during insertion. Actually, not all instructions have a READY register. There are many integer instructions that have a single operand and there are instructions with two long-latency sources. These will not require an additional operand. We will analyze how to exploit this property to further reduce the size of the LLRF. A schematic showing the LLRF and the associated machinery is shown in Figure 6.

Checkpoints. The processor can recover mispredictions in the Cache Processor using the ROB structure there or a rename stack. In the memory processor, these events, although less likely, also occur. Recovery in the MP is supported by using selective checkpointing [11]. Full state checkpoints are taken at specific points during the analyze stage in the CP. At this point the instruction sees a register file composed of READY registers and some long-latency registers. Taking a checkpoint involves copying the ready values from the architectural register file (ARF) at *Analyze* into a free entry of the Checkpointing stack. In addition, all operations that generate a long-latency register must be informed that they should writeback their destination values into this entry of the stack. This implementation is aided by keeping a small RAM parallel to the long-latency bit vector (LLBV). For each active bit in this vector, the RAM contains the position in the LLIB of the instruction that generates the value or a pointer to a previous checkpoint from which to copy the value. Having at least one checkpoint inflight in the LLIB before wakeup assures that no inconsistencies occur. This scheme is shown in Figure 7 where the small RAM structure is referred to as the *Architectural Writers Log* (AWL). The number of ports of the Checkpointing Stack is not a problem as this structure is not frequently accessed.

In the LLIB paragraph it was mentioned that *Analyze* needs to wait for short latency instructions that have not written back their values. This simplifies checkpoint management as it makes sure that all short latency values have written into the ARF when a checkpoint is taken.

3.3 Load/Store Queues

We do not directly address a very important component of the microarchitecture: the load/store queues. A large window processor requires a scalable structure capable of supporting hundreds of in–flight loads and stores. In the D-KIP, the LSQ is decoupled from the remaining structures of the processor and it requires only small modifications to comply with the new interfaces. The reader can assume that D-KIP integrates the hierarchical queue designs presented in [12] or one of the several scalable LSQ designs that have been recently proposed [13] [14].

The LSQ is decoupled from the D-KIP in the same sense as a Decoupled Access-Execute Architecture [15]. In the D-KIP, the Address Processor needs to interface the Cache Processor and the Memory Processor. Load and Store ports can be asymmetrically partitioned – with more capacity for

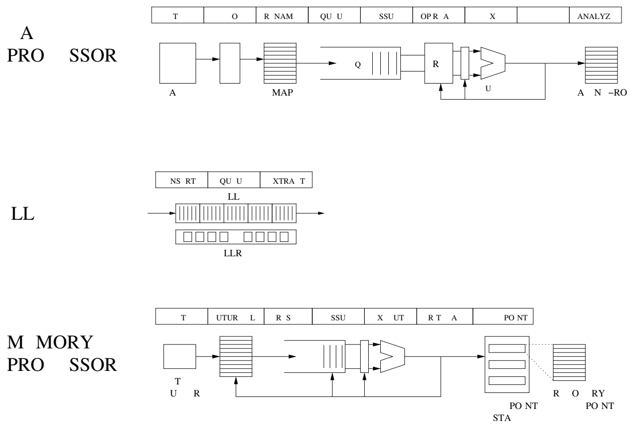

Figure 8. Full Pipeline of the D-KIP

the CP – to support both cores. As was already mentioned, the address processor also needs to keep a FIFO buffer, one per LLIB, to store the results of long latency loads.

3.4 Pipeline

To conclude this section we show the full pipeline of the architecture in Figure 8. The three pipelines (Cache Processor, LLIB, Memory Processor) are chained. Most instructions only traverse the CP pipeline. Instructions will enter the LLIB when the *Analyze* stage determines that they belong to a low locality slice. Finally, insertion into the Memory Processor happens when the oldest instruction in the LLIB depends on a long-latency load that has completed. For other instructions insertion is performed without additional checks.

4 Performance Evaluation

The evaluation of the D-KIP is oriented toward verifying the introduced concept of execution locality and analyzing the efficiency of the architecture itself.

4.1 Simulation Infrastructure

Our simulation infrastructure is designed to execute Alpha binaries and traces. We rely on Simplescalar-3.0 [16] for the loading and execution of these programs. The simplescalar cycle accurate simulator is replaced by a KILO-Instruction Processor simulator capable of simulating a decoupled KILO-Instruction processor. The workbench consists of all benchmarks of the SPEC2000 benchmarking suite. We simulate 200 million committed instructions selected using the SimPoint methodology [17].

We will be evaluating several sizes for structures in the architecture. Table 2 summarizes architectural parameters

Cache Processor	
Architecture	Merged Register File [7]
Fetch/Decode/Analyze Width	4
Branch Predictor	Perceptron [18]
ROB Timer	16 cycles
ROB Capacity	64 entries
ALU Units	4
Integer Multipliers	1
FP Adders	4
FP Multipliers/Divisors	1
LLIB	
Architecture	FIFO Queue
Number of Entries	2048 each
Insertion/Extraction Rate	4
Register Storage	8 banks, 256 regs each (max)
Integer Memory Processor	
Architecture	Future File [8]
Decode Width	4
ALU Units	4
Integer Multipliers	1
FP Memory Processor	
Architecture	Future File [8]
Decode Width	4
FP Adders	4
FP Multipliers/Divisors	1
Address Processor	
Architecture	Hierarchical [12]
Load/Store Queue Size	512 entries
Number of Memory Ports	2 R/W ports (global)
L1 Cache Size	32 KB
L1 Cache Hit Latency	2 (1+1) cycles
L2 Cache Hit Latency	11 (1+10) cycles
Memory Access Latency	400 cycles

Table 2. Parameters of the architecture

that are invariant throughout this evaluation. Table 3 summarizes parameters that are going to be analyzed throughout the paper. The provided values are the defaults and are used when not specified otherwise.

L2 Cache Size	512 KB
CP Integer Queue Size	40
CP FP Queue Size	40
CP Scheduler	Out-of-Order
MP Integer Queue Size	20
MP FP Queue Size	20
MP Scheduler	In-Order

Table 3. Default values for variable parameters

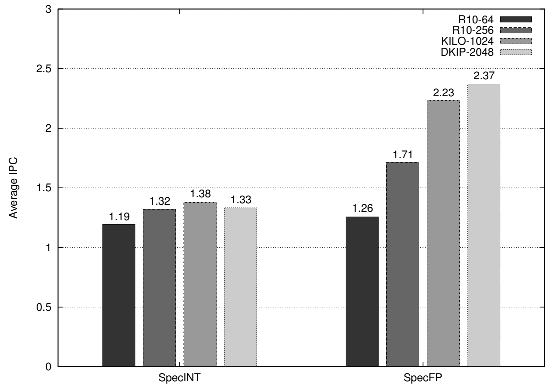

Figure 9. Performance of the D-KIP compared to baselines and a traditional KILO processor

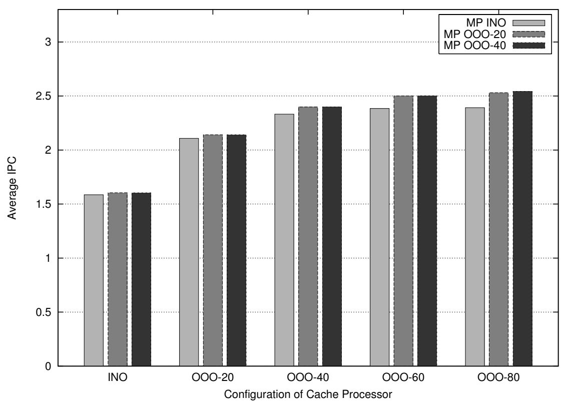

Figure 10. Impact of Scheduling Policy and Queue Sizes in SpecFP

4.2 Performance Comparison

First of all we want to test the performance of our architecture against other existing and experimental architectures. For this test we will use all the default values shown in Table 3. All LSQs are identical and have 512 entries. We will compare against three architectures:

R10-64 An Out-of-Order processor that models a MIPS R10000 processor. It has a ROB size of 64 entries and 40-entry issue queues. It is thus identical to the default Cache Processor.

R10-256 Another R10000-style processor, but with futuristic ROB and Queue sizes. The ROB here has 256 entries and the queues have 160 entries.

KILO-1024 This is an implementation of [9]. The pseudo–ROB has 64 entries and the Slow Lane Instruction Queue is Out-of-Order with 1024 entries. The issue queues have 72 entries.

D-KIP-2048 This is the implementation described here. It features two LLIBs (integer + FP) of 2048 entries each. Note that these queues are FIFOs, which allows them to be larger than the SLIQ. However, as will be shown, this has no impact on performance.

Figure 9 shows the IPC that these configurations yield. The figure shows dramatic speed-ups achievable from the two large window processors. The floating point benchmarks in particular achieve a considerable performance benefit. The reason is simple: Branch prediction in these architectures is highly accurate. Thus, long latency instructions are almost never discarded and are simply processed later after being reinserted from the long latency buffering system. Note that for integer benchmarks the performance of the D-KIP is less than that of the traditional KILO processor. The reason is that integer codes feature a lot of chasing pointers which will profit from an out-of-order instruction buffer such as the SLIQ [9]. Therefore it achieves better performance. It does so, however, at the cost of high complexity and requires a very complex mechanism for register storage [19]. The performance advantage of the D-KIP in SpecFP compared to the KILO stems from the fact that the D-KIPs simple implementation supports trivially to implement two LLIBs and two memory processors which allows it to exploit more parallelism and adds a minimal out-of-order capability to the LLIBs which now can progress out-of-order, but only with respect to each other.

Looking at Figure 2 we see that the D-KIP-2048 achieves a SpecFP performance similar to that of the R10-768, with the difference that the D-KIP-2048 processor has no out-of-order structure larger than 40 entries.

We will analyze now which parameters are most important in these speed–ups.

4.3 Impact of Scheduler Policies and Queue Sizes

In this section we will evaluate the impact of the instruction queue sizes and impose a more severe restriction by forcing the queues to be in-order.

We find that, for integer benchmarks, the D-KIP configuration is only sensitive to the scheduling policy in the Cache Processor. Being out-of-order instead of in-order in this part of the pipeline increases the IPC by 29%. The D-KIP is insensitive to the configuration of the MP. This is reasonable as the MP processes only about 5% of all instructions during integer codes. The speed-ups are a sign that integer benchmarks profit from the D-KIP prefetching capabilities, but not from the additional processing capacity.

An analysis of SpecFP benchmarks shows that there is more potential here. Figure 10 shows the impact of the processor configurations on the execution speed for SpecFP. In this figure, *INO* means ”In-Order”, while *OOO-XX* means ”Out-of-Order” and *XX* refers to the size of the queue.

First, the difference of in-order execution versus out-of-order execution in the Cache Processor again produces a speedup of 32%. However, in this case there are still performance increases as we go to larger processors. With an in-order MP, there is a 13% speed–up when going from an OOO CP with 20 entries to an OOO CP with 80 entries. In addition, as we go to larger CP processors, the configuration of the MP also has more impact. Going from an in-order MP to an out-of-order MP with 40 entries in the queues gives a speed-up of 1% when the Cache Processor is in-order. The same variation produces a speed-up of 6.3% when the Cache Processor is out-of-order with 80 queue entries. Figure 10 also shows that the OOO MP with 20 entries achieves almost the same IPC as the OOO MP with 40 entries. Thus, while OOO in the MP can be useful for aggressive configurations, the number of entries required can be very small in general. The most aggressive configuration achieves an IPC of 2.54, up from the 2.37 achieved by our baseline D-KIP in Figure 9.

4.4 Impact of Cache Sizes

Our next analysis focuses on the memory subsystem. We want to see how the D-KIP behaves under a subset of different sized caches. Smaller caches result in higher miss rates which in the context of the D-KIP means that more instructions are going to be executed in the memory processor. If more instructions are executed in the MP, the scheduling policy there could be of higher importance.

Based on the previous section we select a subset of configurations: *Config–CacheProc/Config–MemProc*. The configurations are: INO/INO (as the worst behaving), OOO-20/INO, OOO-80/INO and OOO-80/OOO-40. We will modify the size of the L2 Cache from 64KB to 4MB, maintaining all other parameters, and analyze the behavior of the architecture under different cache sizes. We also add the R10-256 processor to show the differences with traditional OOO–based processors. The average IPC is shown in Figure 11 and Figure 12 for SpecINT and SpecFP.

The behavior of the D-KIP under cache variations in integer benchmarks is quite common. Each duplication of size in the L2 cache produces more or less a linear speed-up in the IPC. This is very similar to the single-core out-of-order processor. The interesting properties of the D-KIP do show itself in the SpecFP figure. IPC variations are much smaller here. The capacity of the D-KIP to process correct-path long-latency instructions without stalls allows it to be more cache insensitive. The difference between using a 64KB cache and a 2MB cache is less than 15%. It is only when a 4MB Cache is added that a considerable speed-up can be perceived. In any case, the maximum speed–up is still only about 24% (INO–INO configuration).

From the figure it seems that the scheduling policy in the memory processor does not have that much influence on the IPC variations. Thus we expect that even for the smallest cache of 64KB there is still enough execution locality so that the cache processor still processes most of the instruction stream. Our simulations confirm this hypothesis. Even for the 64KB cache, the OOO-80/OOO-40 CP still processes 67% of all committed instructions in the cache processor (for SpecFP). When a 4MB cache is in use, the total number of high locality instructions increases to 77%. This difference is not so large considering that the cache size differs by about two orders of magnitude. It also explains why the D-KIP configuration is so tolerant to variations of the cache size. Compare this with the single-core R10-256 configuration. For the range of caches observed the R10-256 configuration sees a total speed-up of 1.55 while the most aggressive D-KIP configuration sees only a speed–up of 1.18. This shows the tolerance of the decoupled architecture to different cache sizes when executing numerical codes.

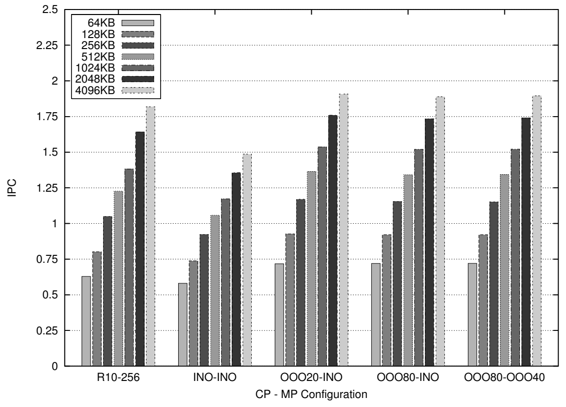

Figure 11. Impact of L2 Cache Size on SpecINT)

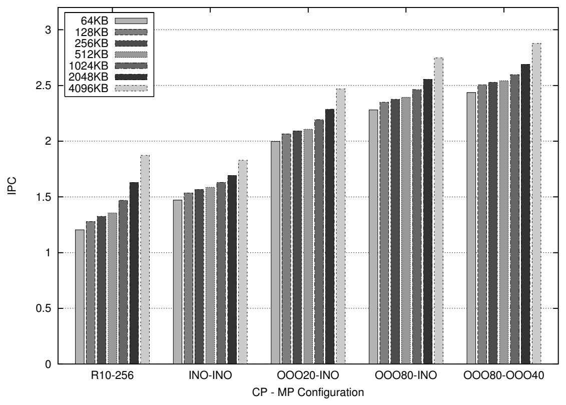

Figure 12. Impact of L2 Cache Size on SpecFP

4.5 Storage Requirements

The LLIB requires an associated register buffer that can be very large. However, not all instructions in the LLIB

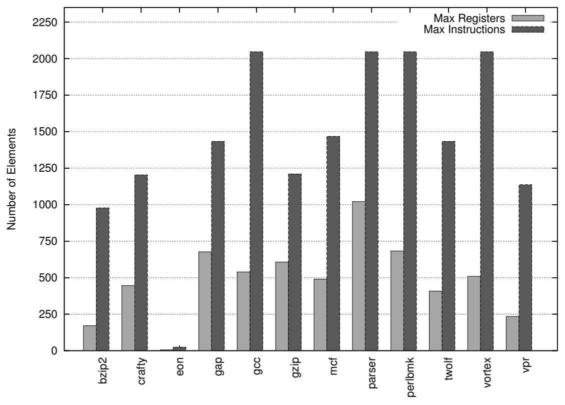

Figure 13. Maximum number of registers and instructions in the LLIB for SpecINT

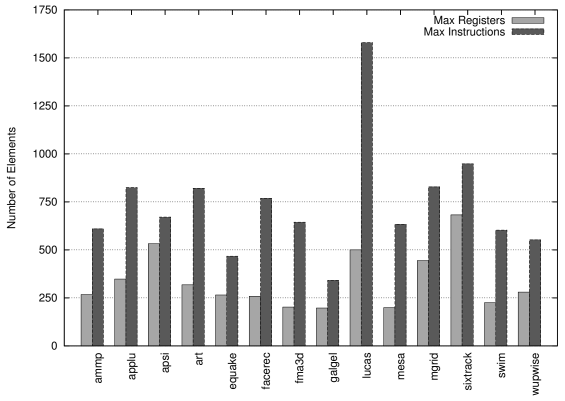

Figure 14. Maximum number of registers and instructions in the LLIB for SpecFP

have an associated READY register. They can be single-source instructions or both sources may be long latency. If a large number of instructions do not require additional storage it may be possible to reduce the size of the register storage by not allocating a register.

Figures 13 and 14 show the maximum number of simultaneous instructions and registers in the LLIB during execution in our D-KIP architecture. Each LLIB can accommodate up to 2048 instructions and an equal number of registers.

The figures show that the real number of necessary registers can be much smaller. The worst case corresponds to integer registers/instructions during SpecINT benchmarks. Many of these benchmarks contain large and irregular load chains. This results in LLIB stalls due to fill-ups for four integer benchmarks. On the other side, none of the SpecFP benchmarks required to fill the LLIB. Note that in Figure 13 we are considering the integer LLIB while in Figure 14 only the floating point LLIB is being considered.

These results suggest that the LLIB can probably be reduced considerably without significantly degrading IPC. For the code regions executed, an LLRF with only 1000 entries would have been enough. This number is large, however there is only a single benchmark that required more than 750 registers. The average number is much smaller, fewer than 500 registers. In any case, it must not be forgotten that the LLRF is a very regular structure with 8 single–ported banks. This makes it clear that a structure such as the LLRF would not be a bottleneck, for neither area nor energy reasons.

5 Design Issues

The D-KIP processor is an attempt to provide the benefits of KILO–Instruction Processing at moderate cost. This section will focus on design issues and comment on the complexity of the approach.

The main technique that we have focused on using to reduce the complexity is decoupling [15]. While *decoupling* does not reduce the amount of hardware that has to be designed, it does limit the interaction between the two modules. The idea is to maintain only very narrow interfaces. Exploiting this property allows designer groups to work almost in isolation, with only little efforts to verify cross-module interaction correctness.

The following interfaces must be considered between the four structures:

CP→LLIB During the *Analyze* stage instructions may be sent to the LLIB as if it were a functional unit. The LLIB must synchronize with the CP and provide entries for the instruction and possibly an associated register. Moreover, when a checkpoint is taken there needs to be a path into the LLIB to inform instructions that create checkpointed registers that they have to writeback into the checkpoint stack.

LLIB→MP When an instruction slice is ready it must be sent to the MP. This is simple considering that the LLIB is a FIFO. In addition, some registers may need to be fetched from the LLRF.

LSQ→MP When long latency loads complete their values are temporarily stored in a FIFO buffer. When the depending instruction arrives at the head of the LLIB it checks if the value is available. In that case, the value needs to be written into the Future File of the MP.

CHPT→CP When the architecture returns to a checkpoint the CP's register file has to be recovered and the LLBV cleared.

CP→CHPT Checkpointed registers must be copied from the ARF in the CP to the Checkpoint Stack when a Checkpoint is taken.

MP→CHPT Checkpointed instructions must write their results into the Checkpointing Stack. Note that MP→CHPT→CP is the only way back–communication can happen in the D-KIP.

6 Related Work

Recently, there has been much research conducted to design microarchitectures able to overcome the memory wall by introducing techniques for the ROB, register files, instruction queues and load/store queues. The basic difference between these techniques and the proposal introduced here is that the D-KIP approaches the memory wall problem from the point of view of the execution locality concept, while previous efforts have concentrated on individually overcoming the scalability problem of complex processor structures.

Several modern techniques try to improve the accuracy of prefetching by actually pre–executing the program but without committing the results. *Assisted threads* [20] [21] [22] rely on pre–executing future parts of the program, selected at compile time or generated dynamically at run time. *Runahead Execution* [23] [24] pre–executes future instructions while an L2 cache miss is blocking the ROB.

Processor behavior in the event of L2 Cache misses has been studied in detail in [3]. Karkhanis *et al.* showed that many independent instructions can be fetched and executed in the shadow of a cache miss. This observation has fueled the development of microarchitectures to support thousands of in–flight instructions.

Many suggestions have been proposed for overcoming the ROB size and management problem. Cristal *et al.* propose virtualizing the ROB by using a small sequential ROB combined with multicheckpointing [25] [9] [26]. Akkary *et al.* have also introduced a checkpointing approach [12] which consists in taking checkpoints on low–confidence branches. *Cherry* [27] uses a single checkpoint outside the ROB to divide the ROB into two regions: a speculative region and a non–speculative region. *Cherry* is then able to early release physical registers and LSQ entries for instructions in the non–speculative ROB section.

Instruction queues have also received attention. The *Waiting Instruction Buffer* (WIB) [6] is a structure that holds all the instructions dependent on a cache miss until it is resolved. The *Slow Lane Instruction Queue* (SLIQ) [9] is similar in concept to the WIB but is designed as an integral component of an overall KILO–instruction microarchitecture. Recently, Akkary *et al.* have proposed the *Continual Flow Pipelines* (CFP) architecture [5] in which they propose the efficient implementation of a two–level instruction queue. It contains a Slice Data Buffer (SDB) which is similar in concept with the SLIQ. As with the SLIQ, the SDB is tightly integrated in a complete microarchitecture designed to overcome the memory wall.

Register Management has also been studied extensively. Several techniques have been developed in the context of out-of-order processors with centralized register storage. *Virtual Registers* [28] is a technique to delay the allocation of physical registers until the issue stage. On the other hand, *Early Release* [29] tries to release register earlier by keeping track of the number of consumers. An aggressive technique consists in combining both approaches. This technique is known as *Ephemeral Registers* [19]. The CFP architecture [5] stores long-lived registers along with the instructions in the Slice Data Buffer. Thus, each entry in the SDB is increased with the space to hold a register value. If the instruction has no READY registers, then the space is wasted. The D-KIP stores long-lived registers through an additional level of indirection and is thus able to save register storage.

7 Conclusions

Our main conclusion is that traditional out–of–order (OOO) execution is not a cost–effective way to handle code that depends on long–latency events. We showed that this technique is effective only to handle code dependent on cache hits, where it provides around 30% IPC improvement (see Figure 10).

Studying program behavior, we observed that over 70% of all instructions are executed a short time after they are fetched. Making an analogy with memory subsystems we described program execution using the concept of execution locality. Instructions depending on short latency events are said to have high execution locality while instructions which depend on off-chip memory accesses are said to have low execution locality.

Exploiting this idea we propose building a decoupled KILO–Instruction processor at moderate cost. We showed that high locality instructions are best processed by an out–of–order *Cache Processor* while low locality instructions can be efficiently processed by a simple in–order *Memory Processor*. Our basic implementation of the architecture featuring out–of–order queues in the Cache Processor with 40 entries and an in–order Memory Processor obtains a speed–up for SpecFP of 40% compared to a futuristic out–of-order processor with 256 entries in the issue queues and an 88% speed–up when compared to a smaller, Cache Processor–like, out–of–order processor. For SpecINT the gains are limited by the irregular branch behavior and by the presence of load chains. In future work we plan to address the impact of these load chains as well as to investigate a decentralized load/store queue organization.

The nature of the decoupled design offers the promise for reduced design complexity. Both the Cache Processor and the Memory Processor are based on well known designs such as the R10000 [7] or the Future File [8]. In addition the Low Locality Instruction Buffer uses a FIFO architecture with a simple register management algorithm.

Acknowlegements

This work has been supported by the Ministerio de Educación y Ciencia of Spain under contract TIN–2004–07739–C02–01, the HiPEAC European Network of Excellence under contract IST-004408, and by the Barcelona Supercomputing Center (BSC-CNS). Daniel A. Jiménez is supported by NSF Grant CCR-0311091 as well as a grant from the Ministerio de Educación y Ciencia of Spain, SB2003-0357. In addition, we would like to thank James C. Hoe, Francisco J. Cazorla, Oliverio J. Santana and the anonymous reviewers for the extensive comments that were helpful to improve the quality of this paper.

References

[1] W. A. Wulf and S. A. McKee, "Hitting the memory wall: Implications of the obvious," *Computer Architecture News*, March 1995.

[2] A. Cristal, O. J. Santana, F. Cazorla, M. Galluzzi, T. Ramirez, M. Pericas, and M. Valero, "Kilo-Instruction Processors: Overcoming the Memory Wall," *IEEE Micro*, vol. 25(3), pp. 48–57, May/June 2005.

[3] T. Karkhanis and J. E. Smith, "A day in the life of a data cache miss," in *Proc. of the Workshop on Memory Performance Issues*, 2002.

[4] S. T. Srinivasan and A. R. Lebeck, "Load latency tolerance in dynamically scheduled processors," *Journal of Instruction Level Parallelism*, vol. 1, pp. 1–24, 1999.

[5] S. T. Srinivasan, R. Rajwar, H. Akkary, A. Gandhi, and M. Upton, "Continual flow pipelines," in *Proc. of the 11th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2004.

[6] A. R. Lebeck, J. Koppanalil, T. Li, J. Patwardhan, and E. Rotenberg, "A large, fast instruction window for tolerating cache misses," in *Proc. of the 29th Intl. Symp. on Computer Architecture*, 2002.

[7] K. C. Yeager, "The MIPS R10000 superscalar microprocessor," *IEEE Micro*, vol. 16, pp. 28–41, Apr. 1996.

[8] J. E. Smith and A. R. Pleszkun, "Implementation of precise interrupts in pipelined proccessors," *Proc. of the 12th Intl. Symp. on Computer Architecture*, pp. 34–44, 1985.

[9] A. Cristal, D. Ortega, J. Llosa, and M. Valero, "Out-of-order commit processors," in *Proc. of the 10th Intl. Symp. on High-Performance Computer Architecture*, 2004.

[10] S. Rixner, W. J. Dally, B. Khailany, P. R. Mattson, U. J. Kapasi, and J. D. Owens, "Register organization for media processing," in *Proc. of the 6th Intl. Symp. on High Performance Computer Architecture*, 2000, pp. 375–386.

[11] W. mei W. Hwu and Y. N. Patt, "Checkpoint repair for out-of-order execution machines," in *Proc. of the 14th Intl. Symp. on Computer Architecture*, 1987, pp. 18–26.

[12] H. Akkary, R. Rajwar, and S. T. Srinivasan, "Checkpoint processing and recovery: Towards scalable large instruction window processors," 2003.

[13] S. Sethumadhavan, R. Desikan, D. Burger, C. R. Moore, and S. W. Keckler, "Scalable hardware memory disambiguation for high ILP processors," in *Proc. of the 36th Intl. Symp. on Microarchitecture*, 2003.

[14] I. Park, C. L. Ooi, and T. N. Vijaykumar, "Reducing design complexity of the load/store queue," in *Proc. of the 36th Intl. Symp. on Microarchitecture*, 2003.

[15] J. E. Smith, "Decoupled state/access computer architectures," in *Proc. of the 9th annual Intl. Symp. on Computer Architecture*, 1982.

[16] T. Austin, E. Larson, and D. Ernst, "Simplescalar: an infrastructure for computer system modeling," *IEEE Computer*, 2002.

[17] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," in *Proc of the 10th Intl. Conference on Architectural Support for Programming Languages and Operating Systems*, 2002.

[18] D. A. Jimenez and C. Lin, "Dynamic branch prediction with perceptrons," in *Proc. of the 7th Intl. Symp. on High Performance Computer Architecture*, January 2001, pp. 197–206.

[19] A. Cristal, J. Martinez, J. LLosa, and M. Valero, "Ephemeral registers with multicheckpointing," Tech. Rep., 2003, technical Report number UPC-DAC-2003-51, Departament d'Arquitectura de Computadors, Universitat Politecnica de Catalunya.

[20] Y. H. Song and M. Dubois, "Assisted execution," Tech. Rep., 1998, technical Report #CENG 98-25, Department of EE-Systems, University of Southern California.

[21] R. S. Chappell, J. Stark, S. P. Kim, S. K. Reinhardt, and Y. N. Patt, "Simultaneous subordinate microthreading (SSMT)," in *Proc. of the 26th. Intl. Symp. on Computer Architecture*, 1999.

[22] A. Roth and G. Sohi, "Speculative data-driven multithreading," in *Proc.of the 7th Intl. Symp. on High-Performance Computer Architecture*, 2001.

[23] J. Dundas and T. Mudge, "Improving data cache performance by pre-executing instructions under a cache miss," in *Proc. of the 11th Intl. Conf. on Supercomputing*, 1997, pp. 68–75.

[24] 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 *Proc. of the 9th Intl. Symp. on High Performance Computer Architecture*, 2003, pp. 129–140.

[25] A. Cristal, M. Valero, A. Gonzalez, and J. LLosa, "Large virtual ROBs by processor checkpointing," Tech. Rep., 2002, technical Report number UPC-DAC-2002-39. [Online]. Available: http://www.ac.upc.edu/recerca/reports/DAC/2002/index,en.html

[26] A. Cristal, O. J. Santana, J. F. Martinez, and M. Valero, "Toward kilo-instruction processors," *ACM Transactions on Architecture and Code Optimization (TACO)*, pp. 389 – 417, December 2004.

[27] J. Martinez, J. Renau, M. Huang, M. Prvulovic, and J. Torrellas, "Cherry: Checkpointed early resource recycling in out-of-order microprocessors," in *Proc. of the 35th Intl. Symp. on Microarchitecture*, 2002, pp. 3–14.

[28] A. Gonzalez, M. Valero, J. Gonzalez, and T. Monreal, "Virtual registers," in *Proc. of the 4th Intl. Conf. on High-Performance Computing*, 1997.

[29] M. Moudgill, K. Pingali, and S. Vassiliadis, "Register renaming and dynamic speculation: an alternative approach," in *Proc. of the 26th. Intl. Symp. on Microarchitecture*, 1993, pp. 202–213.