Kilo-instruction Processors, Runahead and Prefetching

Tanausú Ramírez
DAC - UPC
D6-113 Campus Nord
Barcelona, Spain
tramirez@ac.upc.edu

Alex Pajuelo
DAC - UPC
C6-205 Campus Nord
Barcelona, Spain
mpajuelo@ac.upc.edu

Oliverio J. Santana
DIS - ULPGC
s5 - Campus Tafira
Las Palmas de GC, Spain
ojsantana@dis.ulpgc.es

Mateo Valero*
DAC - UPC
D6-201 Campus Nord
Barcelona, Spain
mateo@ac.upc.edu

ABSTRACT

There is a continuous research effort devoted to overcome the memory wall problem. Prefetching is one of the most frequently used techniques. A prefetch mechanism anticipates the processor requests by moving data into the lower levels of the memory hierarchy. Runahead mechanism is another form of prefetching based on speculative execution. This mechanism executes speculative instructions under an L2 miss, preventing the processor from being stalled when the reorder buffer completely fills, and thus allowing the generation of useful prefetches. Another technique to alleviate the memory wall problem provides processors with large instruction windows, avoiding window stalls due to in-order commit and long latency loads. This approach, known as "Kilo-instruction processors", relies on exploiting more instruction level parallelism allowing thousands of in-flight instructions while long latency loads are outstanding in memory.

In this work, we present a comparative study of the three above-mentioned approaches, showing their key issues and performance tradeoffs. We show that Runahead execution achieves better performance speedups (30% on average) than traditional prefetch techniques (21% on average). Nevertheless, the Kilo-instruction processor performs best (68% on average). Kilo-instruction processors are not only faster but also generate a lower number of speculative instructions than Runahead. When combining the prefetching mechanism evaluated with Runahead and Kilo-instruction processor, the performance is improved even more in each case (49,5% and 88,9% respectively), although Kilo-instruction with prefetch achieves better performance and executes less speculative instructions than Runahead.

Categories and Subject Descriptors

B.8 [**Hardware**]: Performance and Reliability—*General, Performance Analysis and Design Aids*; C.1.3 [**Processor Architecures**]: Other Architecture Styles—*Pipeline processors, out-of-order processors*

General Terms

Measurement, Performance, Experimentation

Keywords

Memory wall, Speculative execution, Prefetching, Runahead, Kilo-instruction processors

*Professor Mateo Valero is also member of the Barcelona Supercomputing Center (BSC - CNS).

CF'06, May 3–5, 2006, Ischia, Italy.

1. INTRODUCTION

The difference between the processor and the memory speed becomes higher and higher every year. This gap between memory and processor speed is well-known in the computer architecture area as the *memory wall* problem [35]. A plethora of techniques have been proposed to alleviate this problem, such as cache memories [25, 34] and out-of-order execution [2, 30]. However, as processor frequency continues increasing and DRAM latencies do not keep up with this improvement, these traditional techniques are not enough to hide the main memory latency, severely limiting the potential performance achievable by the processor. As a consequence, new and different approaches have been appeared to narrow this gap.

The objective of our work is to analyze state-of-the-art mechanisms aiming to overcome the memory wall problem. Because of the large number of proposals, it is not possible to analyse each particular technique in a single paper. Therefore, we have chosen to focus on three well-known techniques: prefetching, Runahead, and Kilo-instruction processors. Aggressive hardware prefetchers are commonly implemented in current processors [12, 29]. Prefetching does an attempt to anticipate the needs of the program being executed, bringing data near the processor before the program requires them, and thus reducing the latency of cache accesses. The efficiency of prefetch depends on data predictability, that is, on the regularity of program access patterns. If future data accesses are correctly predicted, data prefetches will improve the processor performance. On the contrary, wrong prefetches could cause bus contention and pollution in the cache hierarchy.

Runahead execution [11, 20] is an advanced mechanism that relies on improving prefetch efficiency. Runahead prevents the reorder buffer from stalling on long-latency memory operations by executing speculative instructions. To do this, when a memory operation that misses in the L2 cache

gets to the ROB head, it takes a checkpoint of the architectural state. After taking the checkpoint, the processor assigns an invalid value to the destination register of the memory instruction that caused the L2 miss and enters in runahead mode. During runahead mode, the processor speculatively executes instructions relying on the invalid value. All the instructions that operate over the invalid value will also produce invalid results. However, the instructions that do not depend on the invalid value will be pre-executed. When the memory operation that started runahead mode is resolved, the processor rolls back to the initial checkpoint and resumes normal execution. As a consequence, all the speculative work done by the processor is discarded. Nevertheless, this previous execution is not completely useless. The main advantage of Runahead is that the speculative execution would have generated useful data and instructions prefetches, improving the behaviour of the memory hierarchy during the real execution. The drawback of this technique is that it generates a great number of speculative instructions, increasing the overall energy consumption, and leading to the need for research effort focused on reducing this problem [19].

A different approach to overcome the memory wall problem is not relying just on data prefetching, but also on increasing the instruction level parallelism. Several new designs have been recently proposed to increase the amount of instructions available for execution by enlarging the instruction window. When having a larger instruction window, it is possible to execute more independent instructions while long latency loads are outstanding in memory. Thus, while the memory access is being resolved, the processor is able to overlap it with the execution of useful work. Moreover, this useful work includes memory accesses that would not be executed using smaller instruction windows, effectively prefetching data from memory.

Since increasing the size of the instruction window would involve an important increase of the processor complexity, it is necessary to do a smart design of the main processor structures. This trend has lead to the design of *Kilo-instruction processors* [7, 8, 9, 10], a complexity-effective architecture that virtually enlarges the instruction window, by using an efficient checkpoint mechanism, leading to an affordable design that is able to maintain thousands of in-flight instructions.

This paper presents an overall comparison of a stride-based prefetching mechanism, Runahead execution and Kilo-instruction processor in a joint framework. We analyze and evaluate important parameters such as performance, number of executed instructions and the distribution of memory access instructions. This analysis shows the ability of each technique to reduce the memory wall problem, as well as their main advantages and disadvantages. We show what are the limitations that prevent each technique from achieving better performance. Finally, we combine the prefetch mechanism with Runahead and Kilo-instruction processors in order to evaluate the benefits of applying two orthogonal techniques.

The reminder of this paper is organized as follows. We discuss related work and detail background in Section 2. In Section 3 we describe our experimental framework. In Section 4, we present a comprehensive study of the three techniques, identifying key performance issues and research trends. Finally, we conclude in Section 5.

2. BACKGROUND AND RELATED WORK

Prefetch is one of the most used techniques to alleviate the memory wall problem. It is based on predicting future memory accesses to bring, in advance, data to the faster levels of the memory hierarchy. Unfortunately, prefetching has two major problems. Firstly, the extra memory accesses increase the pressure in the memory hierarchy. Secondly, wrong prefetches would pollute the caches, causing unnecessary misses.

Software prefetching techniques [4, 17, 21] rely on the compiler to reduce cache misses by inserting prefetch instructions into the code. This is not a trivial task, since the compiler has limited knowledge of the actual memory behavior of an application. Software prefetching has as major drawback the increment in the size of the application code, as well as the need to devote front-end bandwidth to fetch these instructions.

Hardware prefetching techniques [3, 13, 14] try to dynamically predict the effective memory address of future memory instructions in order to anticipate the data that will be required. These techniques do not enlarge programs by inserting prefetch instructions, but they increase the processor complexity with the tables needed to store the memory access patterns and the logic required to use these data and generate a prediction. There are two important parameters that should be considered when implementing a hardware prefetch mechanism. One is the *degree of prefetching*[33], which indicates the number of prefetches that will be generated for a given instruction. The second parameter is the *distance of prefetching*[33], which sets when the first prefetch starts for a given instruction.

There also exist hybrid prefetching techniques that combine both software and hardware schemes [32]. Another prefetch technique is thread-based prefetching [5, 6, 26, 16]. This technique takes benefit from idle thread contexts in a multithreaded processor to prefetch data for the main thread. Helping threads and assisted threads are two of the most important techniques in this point.

Runahead execution is another mechanism to perform speculative prefetch. It was first proposed for in-order processors [11] and later extended for out-of-order processor as a simple alternative to large instruction windows [20]. A processor with Runahead achieves performance improvement. However, it considerably increases the number of executed instructions, and thus the overall energy consumption of the processor. To reduce this problem, there are some proposals [19] oriented to make Runahead a more energy-efficient technique.

A different approach to overcome the memory wall problem is using complexity-effective strategies to virtually enlarge the instruction window. A simple proposal is the *Waiting Instruction Buffer* (WIB) [15]. Those instructions that depend on an L2 miss are stored in this structure and removed from the instruction window to allow the commit of those instructions that are independent of the L2 miss. Once the data is brought from memory, the instructions in the WIB are reinserted into the instruction window.

Kilo-instruction processors [7, 8, 9, 10] are an architectural proposal that prevents the processor from stalling due to the lack of entries in the ROB under L2 misses. A Kilo-instruction processor consists in a set of techniques to allow thousands of in-flight instructions in the processor, such as multi-checkpointing mechanism, late-allocation and early-

Table 1: Baseline processor configuration

Processor core	
Fetch/issue/commit width	4/4/6
Reorder buffer size	256
INT/FP registers	224 / 224
INT/FP/LS issue queues	128 / 128 / 128
INT/FP/LdSt units	4 / 4 / 2
Branch predictor	Perceptron
RAS	64
Memory subsystem	
Icache	64 KB, 4-way, 1 cyc latency
Dcache	64 KB, 4-way, 3 cyc latency
L2 Cache	1 MB, 8-way, 16 cyc latency
Caches line size	64 bytes
MSHRs	256
Main memory latency	500 cycles

release of registers, and complexity-effective designs of the instruction queues. With a similar philosophy, Akkary *et al.* [1] proposed the *Checkpoint Processing and Recovery* (CPR) mechanism, in which the ROB is completely removed from the processor. This approach incorporates a set of microarchitectural schemes to overcome the ROB limitations, such as selective checkpoint mechanisms, a hierarchical store queue organization and an algorithm for aggressive physical register de-allocation. The *Continual Flow Pipeline* (CFP) architecture [28] is an evolution of CPR, in which an efficient implementation of a bi-level issue queue is provided. To further improve this design, it uses a Slice Data Buffer (SDB), which is a structure with the same philosophy of the above-mentioned WIB.

3. EXPERIMENTAL FRAMEWORK

Data presented in this paper have been obtained using an improved version of the SMTSIM simulator [31] that contains an enhanced memory model. This simulator models an aggressive single-thread superscalar processor that we use as baseline. The main configuration parameters of our baseline are shown in Table 1.

Our simulator also models the three techniques evaluated in this paper. We have implemented a two-delta stride prefetcher as an state-of-the-art prefetching mechanism [22, 23]. The two-delta stride prefetcher includes a 256-entry table located between the shared L2 cache and main memory. The predictor is updated in any first level data cache miss. When an L2 miss is detected, the prefetcher issues a number of prefetch accesses to main memory depending on the prefetch degree. We evaluate this mechanism with distance 1 and prefetch degrees ranging from 1 to 4. Our simulator also models Runahead execution according to the implementation described by Mutlu et al. [20], including some enhancement to reduce the number of extra speculative instructions that are executed [19]. Finally, we model a Kilo-instruction processor by scaling-up the main processor structures (the number of entries in the ROB, L/S queue and instruction queue are 1K, 512 and 512 entries respectively). This strategy provides a good approach to the performance results achievable by an actual Kilo-instruction processor designed to allow resource scalability [10, 8].

Our execution-driven simulator emulates Alpha standard binaries. All experiments were performed using the SPEC 2000 Integer (SpecInt) and Floating Point (SpecFP) benchmark suite [27] with the exception of *sixtrack* and *facerec* benchmarks due to problems with the Fortran compiler. All benchmarks were compiled with the Compaq C V5.8-015 compiler on Compaq UNIX V4.0 with the -O3 optimization level. In order to reduce simulation time, we simulate 300 million representative instructions of each benchmark using the reference input set. To identify the most representative simulation segments we have analyzed the distribution of basic blocks as described in [24]. Table 2 provides important figures about the benchmarks we use in this analysis with our framework. For every benchmark, we show the IPC of our baseline (IPC base), the first level cache miss rate (L1 miss rate, percentage of memory instructions that miss the L1 Dcache) the second level cache miss rate (L2 miss rate, percentage of L1 misses that miss the L2 cache), the global miss rate [1] (percentage of memory instructions that access the main memory), and the branch prediction rate.

4. EVALUATION OF THE MECHANISMS

In this section, we present a comprehensive analysis and evaluation of prefetching, Runahead execution and Kilo-instruction processors. We compare the three approaches, showing their key issues and performance tradeoffs. In order to provide an overall view of their behavior, we also examine other important parameters, such as the number of speculative instructions and the distribution of memory accesses. All these data would enable processor designers to select the best approach to overcome the memory wall problem, both in terms of performance and energy consumption.

4.1 Performance Evaluation

This section provides insight into the performance of the evaluated techniques. Figures 1 and 2 show IPC results for the SpecInt and SpecFP benchmarks. Each Figure has five performance bars: the baseline processor (base), the baseline processor using stride prefetching with a degree of 1 (PF1) and 4 (PF4), Runahead execution (RA), and the Kilo-instruction processor model simulated (kilo).

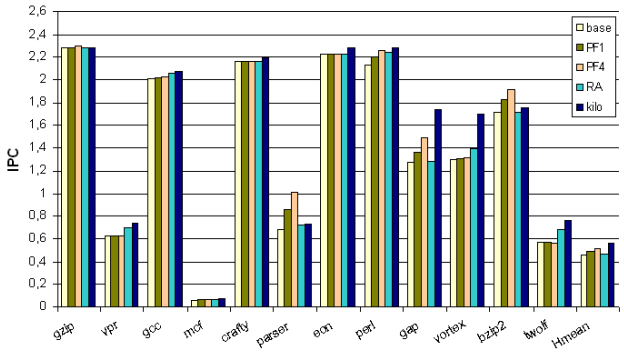

Figure 1: IPC for SpecInt.

In general, all techniques perform better for SpecFP than for SpecInt. This happens because the instruction-level parallelism available for SpecInt is limited by hard-to-predict

[1]The Global miss rate column is computed as $(L2misses/Total\_processor\_accesses)$

Table 2: SPEC2000 Benchmarks details

SPEC INT	IPC base	L1 miss rate (%)	L2 miss rate (%)	Global miss rate (%)	BR prediction rate (%)
gzip	2,28	2,09	3,93	0,08	93,47
vpr	0,63	3,31	18,68	0,63	90,91
gcc	2,01	4,91	1,46	0,11	99,57
mcf	0,06	24,41	87,61	21,41	95,13
crafty	2,16	0,47	4,50	0,02	93,23
parser	0,68	1,97	27,14	0,55	94,84
eon	2,23	0,05	20,92	0,01	99,72
perl	2,13	0,11	31,01	0,04	99,74
gap	1,27	0,34	96,10	0,33	99,32
vortex	1,30	0,97	11,06	0,24	99,77
bzip2	1,71	0,55	18,35	0,10	96,73
twolf	0,57	4,98	19,12	0,95	94,90
SPEC FP					
wupwise	1,46	1,09	84,48	0,93	99,93
swim	0,55	12,05	59,53	7,88	99,92
mgrid	0,77	2,42	61,19	1,51	99,00
applu	0,92	3,56	99,60	3,72	99,96
mesa	2,53	0,30	47,67	0,15	98,20
galgel	1,92	3,85	19,36	0,78	99,50
art	0,43	19,96	73,89	14,90	99,95
equake	0,32	7,57	61,65	4,70	96,00
ammp	0,98	4,41	27,81	1,23	99,38
lucas	0,62	7,04	99,83	7,48	100
fma3d	2,68	0,42	2,31	0,01	98,68
apsi	2,27	1,93	17,61	0,34	99,60

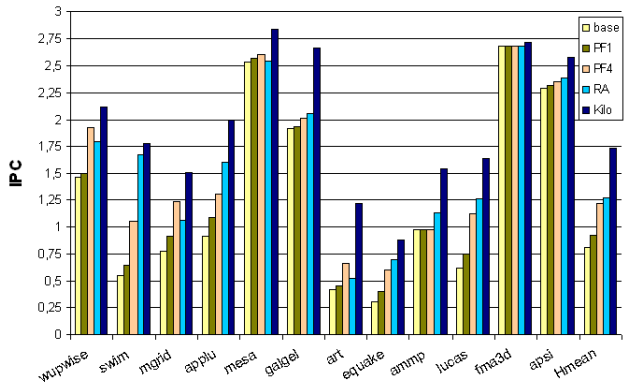

Figure 2: IPC for SpecFP.

branches and chasing pointers [8]. The PF1 prefetcher achieves averaged speedups of 6% for SpecInt and 14% for SpecFP. The more aggressive PF4 prefetcher improves performance by 12% for SpecInt and 50% for SpecFP. The lower instruction level parallelism available in SpecInt is more harmful for Runahead execution, which just achieves 5% performance speedup, although it performs better for SpecFp, achieving 57% speedup. Finally, the Kilo-instruction processor provides the best performance on average for both SpecInt (22%) and SpecFP (115%) programs.

Although the Kilo-instruction processor provides the best performance for both SpecInt and SpecFP programs, it is not the best approach for all individual programs. The aggressive PF4 stride prefetcher achieves better performance for the benchmarks *parser* and *bzip2*. This is due to the fact that the stride prefetcher is able to predict some addresses of the memory operations involved in pointer chains that limit the ability of Runahead and Kilo-instruction processors of exploiting instruction-level parallelism.

As shown in Figure 2, the PF4 prefetcher is not able to outperform Runahead and Kilo-instruction processors for the SpecFP benchmarks, since there is more instruction-level parallelism available. Even so, the PF4 prefetcher is still able to provide performance close to Runahead due to the high predictability of data access patterns in these programs. However, the prefetcher alone is still far from the Kilo-instruction processor.

Like Runahead execution, the Kilo-instruction processor is able to go ahead executing instructions beyond the point where a processor with prefetch alone is forced to stall due to the lack of entries in the ROB. Moreover, the Kilo-instruction processor has an important advantage over Runahead: it does not need to discard the work done under an L2 cache miss. There are also certain long-latency floating-point instructions that Kilo-instruction processor can tolerate well whereas Runahead processor in normal mode cannot do it.

4.2 Executed Instructions

An important parameter to take into account when a speculative mechanism is studied is the amount of extra instructions executed apart from those belonging to the normal program execution.

In the case of prefetching, the extra work performed comes from the additional memory accesses generated by the pre-

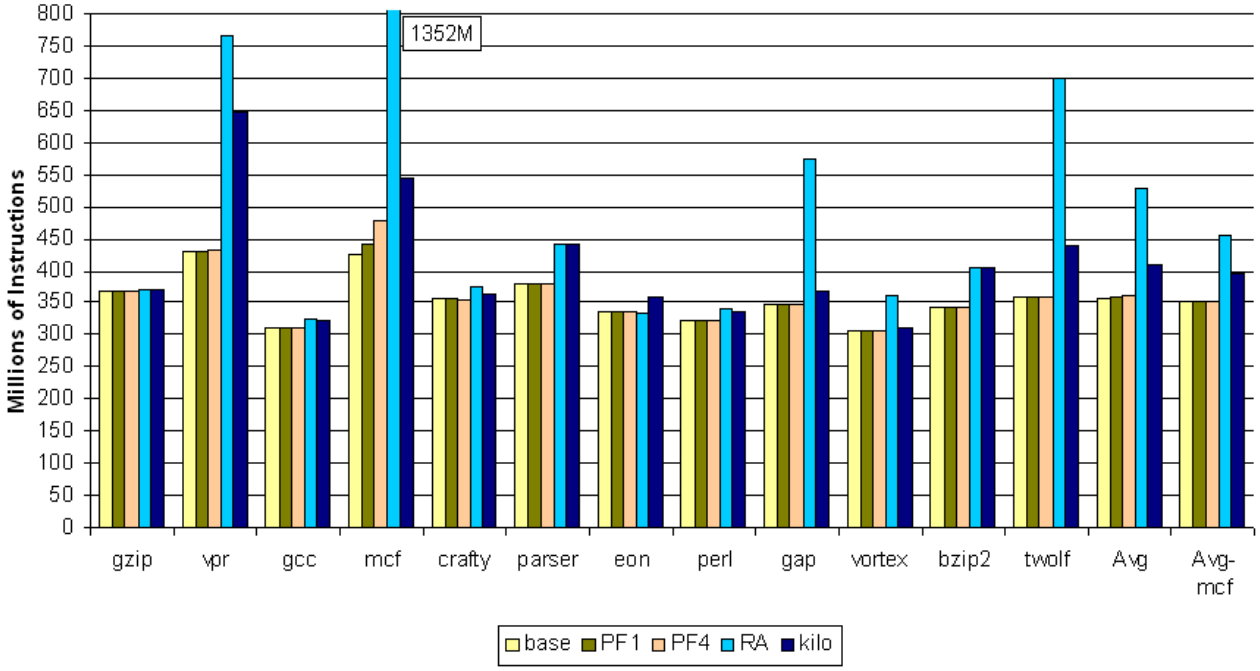

Figure 3: Total executed instructions SpecInt.

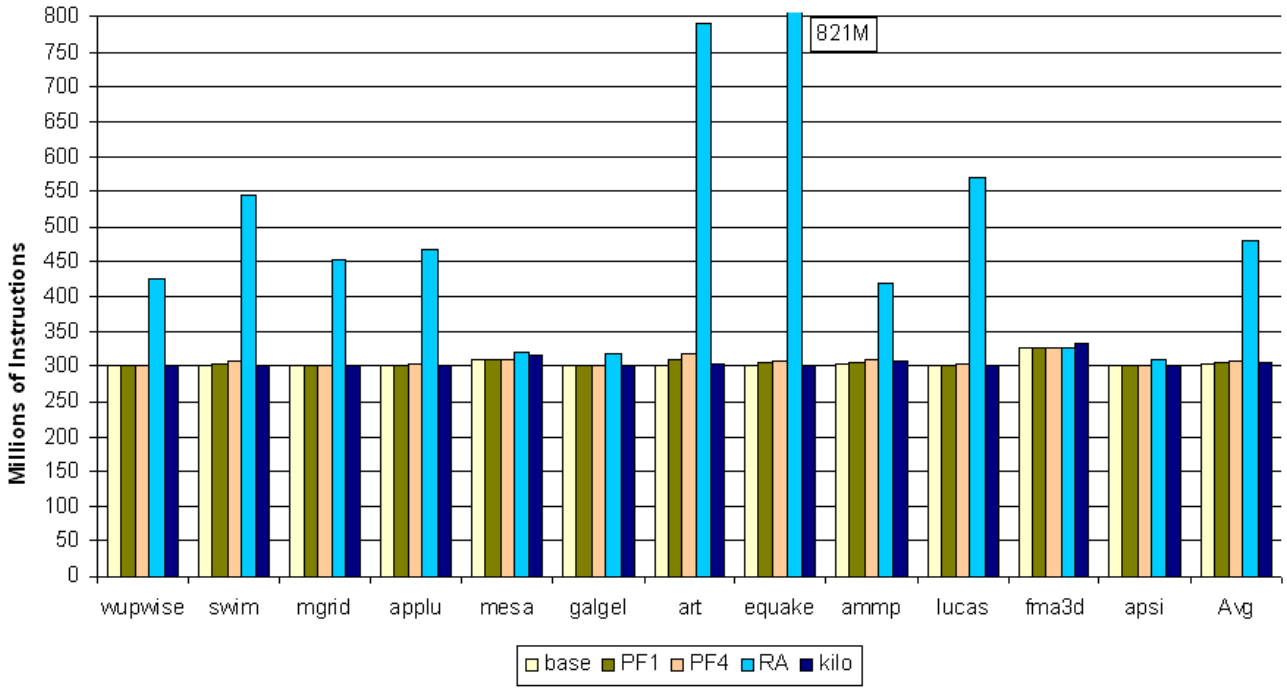

Figure 4: Total executed instructions SpecFP.

fetch mechanism. The Runahead mechanism executes some instructions in the program stream more than once, since a large amount of speculative instructions are executed during runahead mode. Finally, a Kilo-instruction processor can execute more extra instructions down the wrong path due to a larger instruction window.

To compare the evaluated techniques and summarize this effect, we show in Figures 3 and 4 the total number of executed instructions for every mechanism. We have to note that we have implemented the Runahead mechanism with the combination of best dynamic enhancements described in [19] to avoid short (dynamic threshold), overlapped (half threshold policy) and useless runahead periods (Runahead Cause Status Table -RCST).

Figures 3 and 4 show that, in spite of these enhancements, Runahead produces the largest amount of speculative instructions (175 millions of instructions more than the baseline on average). In certain programs, this number is even more than twice the baseline count, such as *twolf* in SpecInt or *art* and *equake* in SpecFP. *Mcf* is the benchmark that, by far, executes the largest amount of speculative instructions (1351 millions) in Runahead mode. Even discarding *mcf*, (Avg-mcf), Runahead increases the number of extra instructions in 104 millions and 58 millions compared to the baseline processor and the Kilo-instruction processor respectively.

On the other hand, the stride prefetcher is the most conservative technique in terms of extra instructions. In this case, the memory accesses issued by the prefetcher are the extra operations when compared to the baseline. One important point to have into account is that, in some cases, the stride prefetcher reduces the number of instructions in some benchmarks respect to the baseline configuration (595.829 instructions less for *bzip2*, 434.414 instructions less for *parser* and 200.983 instructions less for *crafty*). It is due to the earlier resolution of branch instructions that depend on prefetched long-latency loads, thus reducing the number of missfetched instructions down the wrong path. The opposite effect occurs both in Runahead and Kilo-instruction processors, where an eager capacity to execute instructions makes the number of wrong path executed instructions increase (34 millions and 23,5 millions respectively for Runahead and the Kilo-instruction processor).

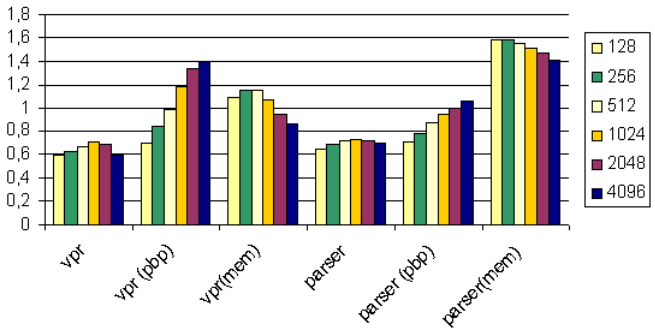

Figure 5: IPC study for *vpr* and *parser*.

Finally, it is interesting to note that large instruction windows may be not beneficial for processor performance in a few particular cases. Figure 5 shows the impact in performance when passing from a 256 to a 4096-entry instruction window for *vpr* and *parser*. Furthermore, this Figure shows the performance benefits of perfect branch prediction (vpr-pbp and parser-bpb) and a memory latency of 1 cycle (vpr-mem and parser-mem). An enlargement of the instruction window in the Kilo-instruction processor from 1K entries to 4K entries impacts negatively in the performance of these benchmarks, *vpr* and *parser*, reducing their IPC from 0,71 to 0,59 (20%) and from 0,73 to 0,69 (6%) respectively. This is mainly due to the larger number of instructions executed down the misspredicted path of branch instructions, which increases the pressure in the execution engine of the processor. In the case of *vpr* and *parser*, passing from a 1K entry instruction window to a 4K entry instruction window means a net increment of 61 and 412 millions of instructions for those benchmarks respectively. Even reducing the memory latency, the performance degradation effect is still present being only alleviated by perfect branch prediction.

4.3 Distribution of Memory Access Instructions

To complete the study of the previous section, here we focus on memory access instructions. All the studied techniques create extra memory accesses to reduce the latency of critical load instructions, improving the performance of applications. However, an excess of extra memory accesses could be harmful because it increases the pressure in the memory hierarchy, delaying non-speculative accesses.

Figures 6 and 7 shows the distribution for SpecInt and SpecFP of executed loads down the correct (light portion of bars) and wrong path (dark portions of bars) for each mechanism. As in the previous section (see Section 4.2), Runahead execution, in spite of the additional techniques devoted to increase its efficiency, presents the largest number of total loads executed (140 millions for SpecInt and 143 millions for SpecFP).

Figures 6 and 7 also show a well-known result: mispredictions are more harmful for SpecInt than for SpecFP, since a larger amount of memory accesses are performed down the wrong path of a conditional branch. This effect is caused by hard-to-predict branches that depend on long latency memory instruction. Current superscalar processors with a bounded instruction window hinder a more harmful effect of this problem, since, sooner or later, the lack of entries in the ROB stalls the fetch and decode of new instructions down the wrong path of a misspredicted branch. In larger instructions windows, since the processor does not stall due to the lack of entries in the ROB, more instructions are executed down the wrong path when the branch predictor provides a wrong prediction.

Vpr, *twolf* and *parser* are examples of this effect (see Figure 6). These benchmarks present a misprediction ratio of 90,9%, 94,9% and 94,8% respectively, which produces 67 millions, 22 millions and 15 millions of extra wrong memory accesses when a larger instruction window is provided.

On average, RA and Kilo-instruction processors present an increment in the number of memory accesses down the wrong path of 7,5 and 11 millions respectively when compared to the baseline superscalar processor for SpecInt. Therefore, a larger window allows more wrong-path references to be executed. Nevertheless, these wrong-path references could have positive effects in the performance due to the prefetching of control-independent memory instructions. Con-

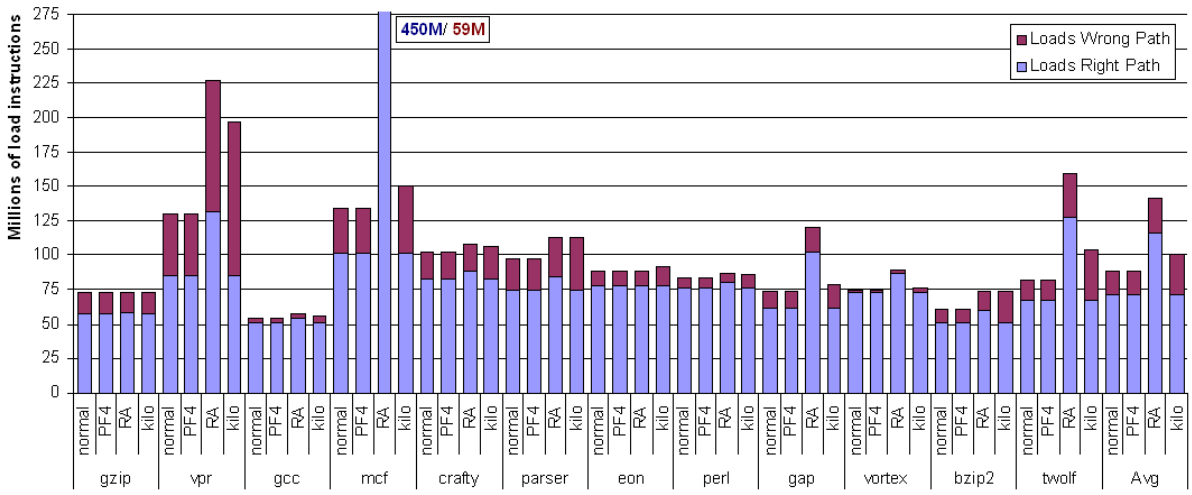

Figure 6: Load instructions distribution SpecInt.

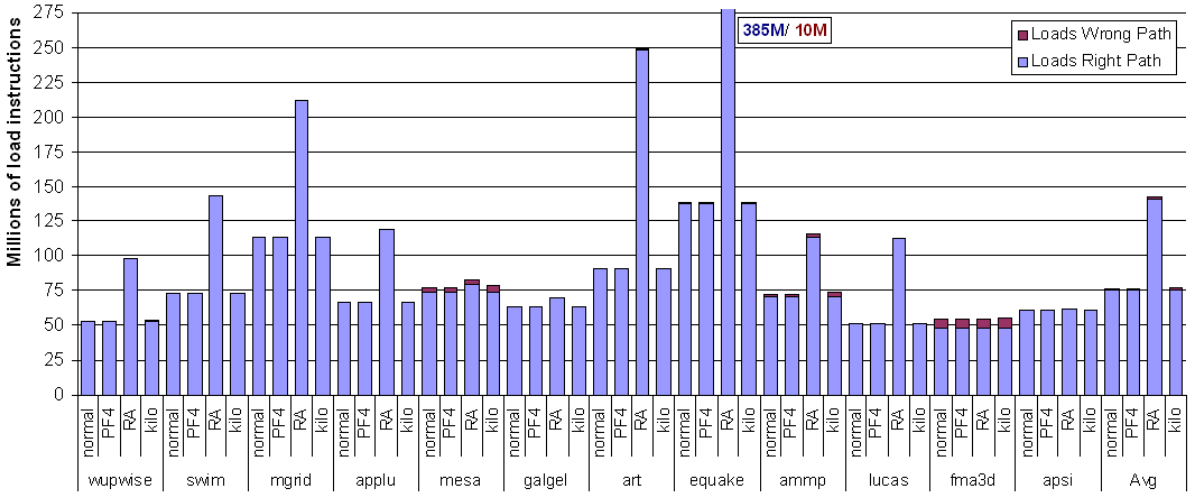

Figure 7: Load instructions distribution SpecFP.

trol independent memory instructions are instructions present in every possible path of a hard-to-predict branch, since they do not depend on the data generated down those paths. Once the processor resolves the misspredicted branch, some of the memory accesses done on the wrong-path can be reused in the correct path of a branch.

4.4 Combining Prefetch with Runahead and Kilo-instructions Processors

Up to this point we have analyzed every technique individually. Now, we show the performance when both Runahead execution and the Kilo-instruction processor are enhanced with a stride-based prefetcher. We choose the 2-delta stride prefetching with an aggressive degree of 4.

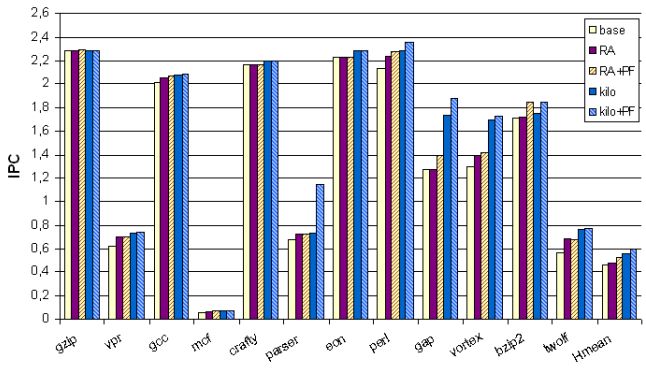

Figure 8: IPC for SpecInt for the mechanisms when prefetching is provided.

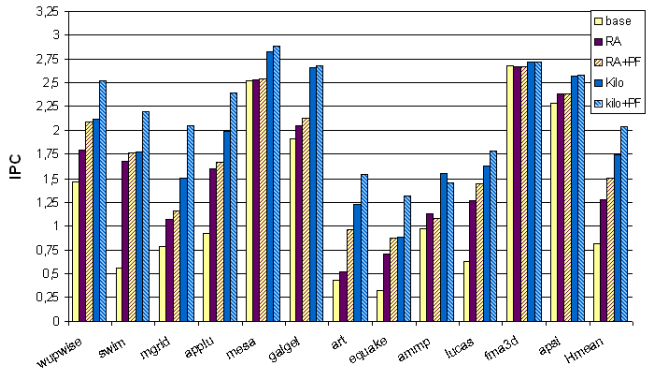

Figure 9: IPC for SpecFP for the mechanisms when prefetching is provided.

Figures 8 and 9 present the performance obtained for the baseline processor (base), the Runahead mechanism (RA), the Runahead mechanism with prefetch (RA+PF), the Kilo-instruction processor (Kilo) and the Kilo-instruction processor with prefetch (Kilo+RA) for every benchmark as well as the harmonic mean for the whole Spec2K. Overall, these Figures show that combining a prefetch with Runahead and the Kilo-instruction processor is in most cases beneficial for both of them. The interaction between prefetch and Runahead achieves 10,6% speedup for SpecInt and 17,8% speedup for SpecFp over Runahead alone. Regarding the Kilo-instruction processor, the interaction with prefetch results in a performance improvement of 8,4% for SpecInt and 17,3% for SpecFP over the Kilo-instruction processor without prefetch.

The benchmark *parser* is an interesting case to remark. As show in Figure 8, the performance improvement of *parser* is higher for the Kilo-instruction processor with prefetch than without it, about 57%. This is mainly due to the fact that, as previously commented, prefetch reduces the resolution time of branch instructions that depend on long latency memory operations. Since prefetch effectively reduces the latency of L2 miss instructions, dependent branches are resolved quicker, what allows to trigger a misprediction recovery sooner, reducing the number of instructions executed down the wrong path. In the case of *parser*, the average branch latency passes from 12,35 cycles to 7,89 cycles when the mechanism of prefetch is provided, reducing the number of accesses down the mispredicted path by nearly 16 millions. On the other hand, Runahead with prefetch is not able to improve the performance of parser, since stride prediction coverage is low (3,65%) for this benchmark in this case.

There are two cases where the addition of a prefetch mechanism in both Runahead and Kilo results in lower performance improvement that the obtained in isolation: *vpr* and *ammp*. This is mainly due to the low accuracy of the stride predictor for these benchmarks (49% for vpr and 42% for ammp).

Finally, Runahead execution and the Kilo-instruction processor achieve 49,5% and 88,9% average speedups for both SpecInt and SpecFP with regard to the baseline processors when the stride-based prefetcher is included. It is remarkable that the Kilo-instruction processor model obtains a better performance improvement than Runahead. Even if both techniques face the same problems (long-latency loads and hard-to-predict branches) Runahead has a clear disadvantage: all the instructions executed speculatively in runahead mode are discarded, only obtaining benefits from prefetches. This makes us think that any technique focused to alleviate the memory wall problem would perform better combined with a Kilo-instruction processor than with Runahead. For example, whatever technique that tries to resolve the dependent long-latency loads problem in Runahead [18] is completely orthogonal and can be applied in a Kilo-instruction processor for the same problem. Moreover, while in Runahead this enhancement will affect only the speculative mode, in the Kilo-instruction processor the enhancement will affect all correct-path executed instructions.

5. CONCLUSIONS

In this paper we present a detailed analysis of three well-known techniques to alleviate the memory wall problem: prefetch, Runahead and Kilo-Instruction processors. We show that Kilo-instruction processors provide the best performance compared to prefetching and Runahead execution. The prefetcher mechanism is limited by the data predictability in the program, which reduces its potential coverage. Moreover, an aggressive prefetch is necessary to achieve good performance, which sometimes could increase the pressure over memory.

We analyze other important factors to consider, such as the number of executed instructions and wrong-path memory references. We show that Runahead execution obtains

an acceptable performance improvement but, unfortunately, it is the mechanism that executes the largest amount of speculative extra instructions. Then, the Runahead mechanism creates a large amount of speculative instructions that consume dynamic energy to lately discard them. Besides, both Runahead and Kilo-instruction processors execute a large amount of accesses down the mispredicted path, which becomes a factor with a high impact on performance for the latter.

We also show that the combination of stride-based prefetching with Runahead or Kilo-processor improves the average performance in both cases. This is basically due to the fact that prefetches reduce the average branch resolution time, reducing the number of executed instructions down the misspredicted path of a hard-to-predict branch that depends on a long-latency memory operation. Finally, we show that applying a technique focused on alleviating the memory wall problem to the Kilo-instruction processor is more beneficial than in the Runahead mechanism since the latter executes more instructions and is not able to reuse the speculatively computed data.

6. ACKNOWLEDGMENTS

This work has been supported by the Ministry of Education of Spain under contract TIN-2004-07739-C02-01 and grant AP2003-3682, the HiPEAC European Network of Excellence, and the Barcelona Supercomputing Center (BSC-CNS).

7. REFERENCES

[1] H. Akkary, R. Rajwar, and S. T. Srinivasan. Checkpoint processing and recovery: Towards scalable large instruction window processors. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture (MICRO 36)*, pages 423–434, Washington, DC, USA, 2003. IEEE Computer Society.

[2] D. Anderson, F. Sparacio, and R. Tomasulo. The ibm system/360 model 91: Machine philosophy and instruction-handling. *IBM J. Research and Development*, pages 8–24, 1967.

[3] J.-L. Baer and T.-F. Chen. An effective on-chip preloading scheme to reduce data access penalty. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 176–186, New York, NY, USA, 1991. ACM Press.

[4] D. Callahan, K. Kennedy, and A. Porterfield. Software prefetching. In *Proceedings of the fourth international conference on Architectural support for programming languages and operating systems (ASPLOS-IV)*, pages 40–52, New York, NY, USA, 1991. ACM Press.

[5] R. S. Chappell, J. Stark, S. P. Kim, S. K. Reinhardt, and Y. N. Patt. Simultaneous subordinate microthreading (ssmt). In *Proceedings of the 26th annual international symposium on Computer architecture (ISCA '99)*, pages 186–195, Washington, DC, USA, 1999. IEEE Computer Society.

[6] J. D. Collins, D. M. Tullsen, H. Wang, and J. P. Shen. Dynamic speculative precomputation. In *Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture (MICRO 34)*, pages 306–317, Washington, DC, USA, 2001. IEEE Computer Society.

[7] A. Cristal, D. Ortega, J. Llosa, and M. Valero. Out-of-order commit processors. In *Proceedings of the 10th International Symposium on High Performance Computer Architecture (HPCA '04)*, Madrid, Spain, 2004. IEEE Computer Society.

[8] A. Cristal, O. J. Santana, F. Cazorla, M. Galluzzi, T. Ramírez, M. Pericas, and M. Valero. Kilo-instruction processors: Overcoming the memory wall. *IEEE Micro*, 25(3):48–57, 2005.

[9] A. Cristal, O. J. Santana, M. Valero, and J. F. Martínez. Toward kilo-instruction processors. *ACM Transactions on Architecture and Code Optimization.*, 1(4):389–417, 2004.

[10] 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*, July 2002.

[11] J. Dundas and T. Mudge. Improving data cache performance by pre-executing instructions under a cache miss. In *Proceedings of the 11th international conference on Supercomputing (ICS '97)*, pages 68–75, New York, NY, USA, 1997. ACM Press.

[12] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. The microarchitecture of the Pentium 4 processor. *Intel Technology Journal Q1*, page 13, Feb. 2001.

[13] D. Joseph and D. Grunwald. Prefetching using markov predictors. In *Proceedings of the 24th annual international symposium on Computer architecture (ISCA '97)*, pages 252–263, New York, NY, USA, 1997. ACM Press.

[14] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the 17th annual international symposium on Computer Architecture (ISCA '90)*, pages 364–373, New York, NY, USA, 1990. ACM Press.

[15] A. R. Lebeck, J. Koppanalil, T. Li, J. Patwardhan, and E. Rotenberg. A large, fast instruction window for tolerating cache misses. In *Proceedings of the 29th annual international symposium on Computer architecture (ISCA '02)*, pages 59–70, Washington, DC, USA, 2002. IEEE Computer Society.

[16] C.-K. Luk. Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors. In *Proceedings of the 28th annual international symposium on Computer architecture (ISCA '01)*, pages 40–51, Göteborg, Sweden, 2001. ACM Press.

[17] T. C. Mowry, M. S. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of the fifth international conference on Architectural support for programming languages and operating systems (ASPLOS-V)*, pages 62–73, oston, Massachusetts, USA, 1992. ACM Press.

[18] O. Mutlu, H. Kim, and Y. N. Patt. Address-value delta (avd) prediction: Increasing the effectiveness of runahead execution by exploiting regular memory allocation patterns. In *Proceedings of the 38th annual IEEE/ACM International Symposium on*

Microarchitecture (MICRO 38), pages 233–244, Washington, DC, USA, 2005. IEEE Computer Society.

[19] O. Mutlu, H. Kim, and Y. N. Patt. Techniques for efficient processing in runahead execution engines. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA '05)*, pages 370–381, Washington, DC, USA, 2005. IEEE Computer Society.

[20] 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 *Proceedings of the 9th International Symposium on High-Performance Computer Architecture (HPCA '03)*, page 129, Washington, DC, USA, 2003. IEEE Computer Society.

[21] D. Ortega, M. Valero, and E. Ayguadé. A novel renaming mechanism that boosts software prefetching. In *Proceedings of the 15th international conference on Supercomputing (ICS '01)*, pages 501–510, Sorrento, Italy, 2001. ACM Press.

[22] D. G. Perez, G. Mouchard, and O. Temam. Microlib: A case for the quantitative comparison of micro-architecture mechanisms. In *Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture (MICRO 37)*, pages 43–54, Washington, DC, USA, 2004. IEEE Computer Society.

[23] Y. Sazeides and J. E. Smith. The predictability of data values. In *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture (MICRO 30)*, pages 248–258, Washington, DC, USA, 1997. IEEE Computer Society.

[24] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques (PACT '01)*, pages 3–14, Washington, DC, USA, 2001. IEEE Computer Society.

[25] A. J. Smith. Cache memories. *ACM Computing Surveys*, 14(3):473–530, September 1982.

[26] Y. Song and M. Dubois. Assisted execution. *Technical Report #CENG 98-25*, Department of EE-Systems, University of Southern California, October 1998.

[27] SPEC. Standard performance evaluation corporation (spec) 2000 benchmark suite. *http://www.spec.org*

[28] S. T. Srinivasan, R. Rajwar, H. Akkary, A. Gandhi, and M. Upton. Continual flow pipelines. In *Proceedings of the 11th international conference on Architectural support for programming languages and operating systems (ASPLOS-XI)*, pages 107–119, New York, NY, USA, 2004. ACM Press.

[29] J. M. Tendler, J. S. Dodson, J. S. Fields(Jr.), H. Le, and B. Sinharoy. POWER4 system microarchitecture. *IBM Journal of Research and Development*, 46(1):5–26, 2002.

[30] Thornton, J. E. Parallel operation in the CONTROL DATA 6600. In *Proceedings Fall Joint Computer Conference*, pages 26, 33–40, 1964.

[31] D. M. Tullsen. Simulation and modeling of a simultaneous multithreading processor. In *International Annual Computer Measurement Group Conference*, pages 819–828, 1996.

[32] S. VanderWiel and D. Lilja. A compiler-assisted data prefetch controller. In *International Conference on Computer Design (ICCD '99)*, pages 372–377, Washington - Brussels - Tokyo, Oct. 1999. IEEE.

[33] S. Vanderwiel and D. J. Lilja. A survey of data prefetching techniques. *Technical Report HPPC-96-05*, Departament of Computer Science, University of Minnesota, October 1996.

[34] Wilkes, M. V. Slave memories and dynamic storage allocation. *IEEE Transaction on Electronic Computers*, EC-14(2):270–271, 1965.

[35] W. A. Wulf and S. A. McKee. Hitting the memory wall: implications of the obvious. *SIGARCH Computer Architecture News*, 23(1):20–24, 1995.