Finding and solving difficult predictable branches

A. Gellert - A. Florea

University of Sibiu, Computer Science Department, Sibiu, Romania

Abstract. – *A solution to overcome the increasing gap between processor performance and memory speed is the Kilo-Instruction Processor (KIP) architecture. KIP performance achieved on integer applications is sometimes limited by hard to predict branches. Hard-to-predict branches are especially harmful because a branch misprediction causes a pipeline flush, eliminating the advantage of a large number of in-flight instructions. The recovery process in this case will restart execution of all instructions between the last checkpointed instruction and the mispredicted branch, causing additional power consumption. Thus, it would be more convenient if the distance between loads and hard-to-predict branches would be as short as possible. Mispredicted branch instructions that depend on long-latency memory operations are even more detrimental because of the very long delay in discovering the misprediction. In this work we performed some statistics on SPEC2000 benchmarks related to critical load instructions and the distance between them and the dependent branches. In average 11.10% of dependent branch instructions follow next to critical loads. Also, in average there are 28.68% of branch instructions that are dependent on critical loads, 5.61% of them being unbiased.*

Limits of Instruction Level Parallelism: Memory Wall and Unbiased Branches

In the last decade processor cycle time has been decreasing at a rate faster than the memory access time. Additionally, the architectural design of processors has improved with the development of deeper pipelined and multiple instruction issue architectures. These factors have influenced the increasing technological gap – named *memory wall* [2] – between processor speed and the speed of the underlying memory hierarchy. Thus, in order to improve the performance of memory-intensive application programs, there are needed innovative techniques to tolerate long-latency main memory accesses. Recently, different approaches were proposed to design microarchitectures able to overcome the memory wall by introducing techniques for efficient management of architectural resources (reorder buffer, register files, instruction queues and load/store queues).

Cristal et al. [2] proposed the *Kilo-Instruction Processors* – a solution to overcome the increasing gap between processor performance and memory speed. KIP is a new type of out-of-order superscalar processor that overlaps long memory access delays by maintaining thousand of in-flight instructions. The traditional approach of scaling up critical processor structures to provide such support is impractical at these levels, due

to area, power, and cycle time constraints. In order to overcome this resource-scalability problem it was proposed a smarter use of the available resources, supported by a selective checkpointing mechanism. This mechanism allows instructions to commit out of order, and makes a reorder buffer unnecessary. KIP is based on a set of techniques such as multilevel instruction queues, late allocation and early release of registers, and early release of load/store queue entries.

However, KIP performance achieved on integer applications is sometimes limited by *hard to predict branches* and *pointer chasing*. Hard-to-predict branches are especially harmful because a branch misprediction causes a pipeline flush, eliminating the advantage of a large number of in-flight instructions. The recovery process in this case will restart execution of all instructions between the last checkpointed instruction and the mispredicted branch, causing additional power consumption. Thus, it would be more convenient if the distance between loads and hard-to-predict branches would be as short as possible. On the other side, if the unbiased branch is enough far away from the load and there are sufficient resources, multi-path execution (multiple flows of control) could be applied. Mispredicted branch instructions that depend on long-latency memory operations are even more detrimental because of the very long delay in discovering the misprediction. Another challenge comes from the opportunity to combine kilo-instruction processors with mechanisms for detecting control-independent instructions that follow branches. There is no need to flush control-independent instructions after a branch misprediction, saving resources, energy, and execution bandwidth.

The Simulated Architecture and Results

Starting from these challenges we made some analysis based on Simplescalar-3.0 tool set [1]. The simulated infrastructure is designed to execute Alpha binaries and traces. The baseline cycle accurate simulator used was the Decoupled Kilo Instruction Processor (DKIP) simulator developed by UPC team for HPCA-2006 [3]. The Figure 1

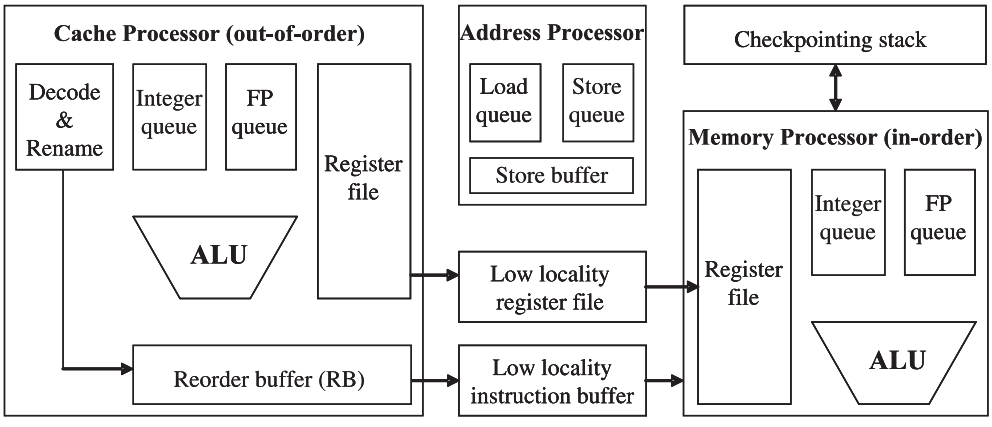

FIGURE 1. – *The Decoupled Kilo-Instruction Processor.*

illustrates the architecture of DKIP. We developed on this structure some own modules in order to obtain different statistics related to critical load instructions (loads with miss in the L2 data cache that reach the head of the ROB). The workbench consists of some computer intensive integer benchmarks – *bzip2*, *gcc*, *gzip* – and some memory-intensive integer benchmarks – *mcf*, *parser*, *twolf* and *vpr* –, selected from SPEC2000 [5]. To perform this job we ran the simulators on SUN machines with Sparc processors under UNIX operating systems.

Pericàs et al. [3] introduced the execution locality concept, a property that describes instructions as a function of the number of cycles they wait in the queues until they issue. Thus, instructions depending on cache misses have low execution locality, while the remaining instructions, including those that depend only on cache hits, have high execution locality. The small amount of low execution locality code causes stalls that significantly reduce performance.

Based on the observation that low execution locality code is decoupled from high execution locality code, Pericàs et al. proposed a decoupled microarchitecture (Figure 1) that executes low latency instructions on a Cache Processor and high latency instructions on a Memory Processor. Thus, one pipeline, the out-of-order Cache Processor, exploits instruction-level parallelism, while a second pipeline, the in-order Memory Processor, exploits memory-level parallelism. The instructions are fetched by the Cache Processor where they are waiting to be issued to the functional units. If an instruction turns out – based on a timer – to have long issue latency, it is moved from the Cache Processor into a Low Locality Instruction Buffer (LLIB), where it is waiting

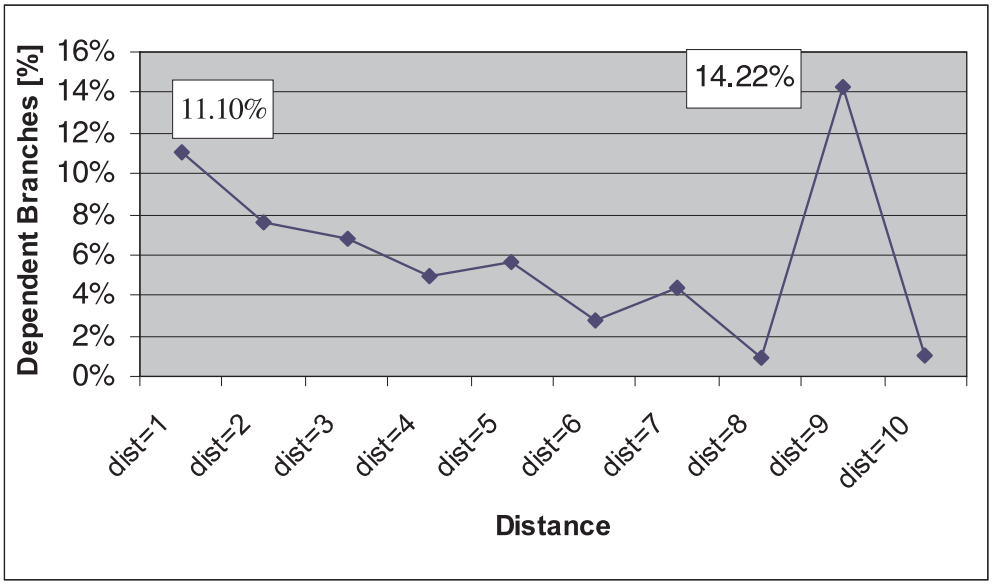

FIGURE 2. – *Average percentage of branches that depend on critical loads. Fine-grained statistics regarding the number of instructions that are committed between critical loads and dependent branches.*

until all long-latency load operations it depends on have finished. When the operands of a long latency instruction are available, they are inserted into the Low Locality Register File (LLRF). Long-latency loads are executed in the address processor by the LSQ. After a long latency load completes, the value is kept in the address processor. When the dependent instructions arrive to the head of the LLIB and the load value is available, they are moved to the Memory Processor to be executed. For the recovery after mispredictions in the Cache Processor an RB structure is used. In the Memory Processor the recovery is assured through selective checkpointing. Taking a checkpoint involves copying the ready values from the architectural register file into a free entry of the Checkpointing Stack. The state of the Memory Processor is restored from the Checkpointing Stack if an exception occurs.

We continued our research trying to determine how many branches depend on a critical load (loads that accessed with miss the L2 data cache and reached the head of the ROB). We determined how many instructions are committed between a critical load and the first dependent branch instruction (see Figure 2). We have found that 11.10% of dependent branch instructions follow next to critical loads. If all mispredicted branch instructions (supposing those found unbiased) belong to this category, the waste in performance and energy is not too bad. A very interesting aspect is that the dependent branch is the ninth instruction after a critical load in 14.22% of cases.

We computed the percentage of dependent branches reported to the total number of branch instructions that are committed (branches from the correct paths). The percentage of dependent branches reaches 60% – maximum value obtained on *mcf* – with an average of 28.68% (see Figure 3).

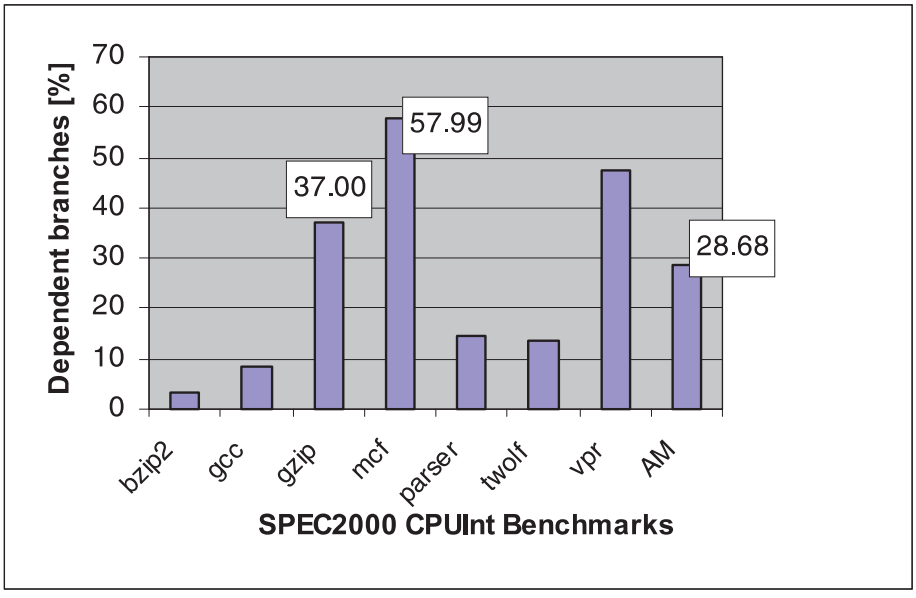

FIGURE 3. – *The percentage of committed dependent branches.*

This dependence chain occurs in the polymorphic call for computing the target of an indirect call. The polymorphism is translated in accessing the Virtual Method Table (VMT) using a sequence of three dependent load instructions followed by an indirect function call [4]. A Virtual Method Table contains the addresses (at different offsets) of all methods accessible by a given object class. Every object is initialized with a pointer to the VMT corresponding to its class and accesses its function via this pointer. The initial load accesses the base of the object, the second uses the object's base address to access its VMT and the third retrieves the function address which is used by the indirect call.

Further, we want to find if these *low execution locality* branches [3] are unbiased branches detected with our methodology. In a previous work [6] we considered that a dynamic branch in a certain attached dynamic context (local history, global history, path information, etc.) is difficult predictable if it is unbiased – meaning that the behavior of the branch is not sufficiently polarized for that certain context – and its Taken respectively Not Taken outcomes are shuffled. We identified and reduced the number of unbiased branches by passing them through successive cascades of different contexts, increasing history information (from 16- to 28-bits). The simulation results show that for context lengths of 16 bits the average percentage of unbiased dynamic branches is 17.47%, and for context lengths of 28 bits it remains still high (6.19%).

In [6] we showed that branches which are unbiased in a certain dynamic context are hard to predict using that context information. We obtained with the PAg, predicting only the branches that were unbiased on their 16-bit local history context, a prediction accuracy of 75% at average. With the GAg, predicting only the branches that were unbiased on their 16-bit global history context, we measured an average prediction accuracy of only 72%, confirming the fact that unbiased branches are highly unpredictable.

TABLE 1. – Reducing the number of unbiased branches by increasing history register lengths.

SPEC2000	**16-bits**	**20-bits**	**24-bits**	**28-bits**
mcf	3.28%	2.58%	2.17%	1.81%
parser	12.95%	8.39%	5.46%	3.56%
bzip	23.40%	14.62%	8.92%	5.35%
gzip	28.89%	24.07%	18.85%	14.55%
twolf	32.41%	22.99%	7.28%	5.67%
gcc	3.91%	1.94%	1.20%	0.85%
Average	17.47%	12.43%	7.31%	6.19%

From Table 1 and Figure 3 it can be observed that despite the unbiased branch percentage on *mcf* benchmark is quite low (1.81%), the percentage of dependent branches is very high (60%). Also, 31% of dependent branches from the *mcf* benchmark

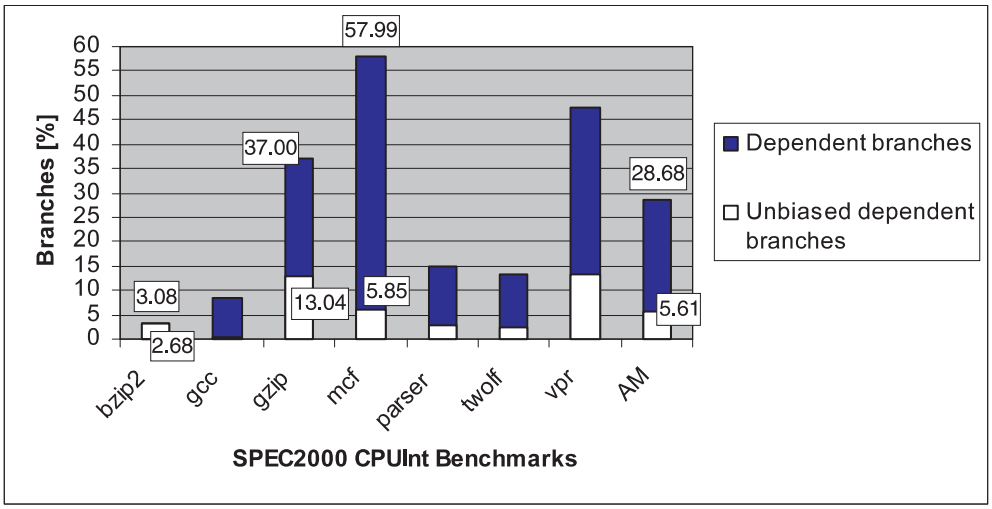

FIGURE 4. – *The percentage of dependent branches and unbiased dependent branches from total number of committed branches.*

are committed next to a critical load, dramatically decreasing the overall performance. A lower performance can be obtained on *gzip* where the percentage of unbiased branches is the highest from all tested benchmarks (14.55%), and the percentage of dependent branches is also very high (37%), 43.68% of them following at a distance of at most three instructions after critical loads. Our evaluations show that 5.61% of the branches are unbiased on their 16-bit global history context and in the same time they depend on a previously committed critical load instruction (Figure 4). Figure 5 shows

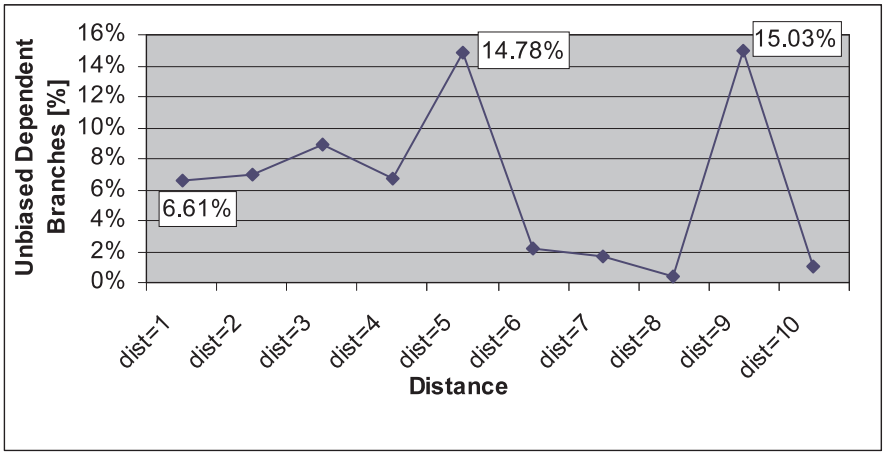

FIGURE 5. – *Average percentage of unbiased branches that depend on critical loads. Fine-grained statistics regarding the number of instructions that are committed between critical loads and unbiased dependent branches.*

that 6.61% of unbiased dependent branches are committed next to the critical load they depend on. The unbiased dependent branch is the fifth committed instruction after the critical load in 14.78% of cases and, respectively, the ninth in 15.03% of cases.

Future Work

Another analyze that we would like to make refers to the performance gain obtained if the critical loads, that provide the source register in branch condition, would be correctly predicted. We will also study the correlation between the percentage of unbiased branches and total number of instructions executed from programs (very important in analyze of additional power consumption caused by unbiased branches). Further, we will continue our previous work in finding and solving difficult predictable branches using different predictors (i.e. perceptron) and larger context information. Also we will study the feasibility of a branch predictor dedicated for unbiased branches integrated into KIP.

Acknowledgements. The work has been performed under the Project HPC-EUROPA (RII3-CT-2003-506079), with the support of the European Community – Research Infrastructure Action under the FP6 "Structuring the European Research Area" Programme.

Publications

[1] BURGER D. and AUSTIN T., *The SimpleScalar tool set*, version 2.0, Technical Report 1342, University of Wisconsin - Madison Computer Sciences Department, 1997.

[2] CRISTAL A., SANTANA O., CAZORLA F., GALLUZI M., RAMIREZ T., PERICAS M. and VALERO M., *Kilo-Instruction Processors: Overcoming the Memory Wall*, IEEE Micro, Vol. **25**, No. **3**, 2005.

[3] PERICAS M., GONZALEZ R., CRISTAL A., JIMENEZ D. and VALERO M., *A Decoupled KILO-Instruction Processor*, Proceedings of the 12th International Symposium on High Performance Computer Architecture, Austin, Texas, 2006.

[4] ROTH A., MOSHOVOS A. and SOHI G., *Improving virtual function call target prediction via dependence-based pre-computation*, Proceedings of the 13th International Conference on Supercomputing, Greece, 1999.

[5] SPEC, *The SPEC benchmark programs*, http://www.spec.org.

[6] VINTAN L., GELLERT A., FLOREA A., OANCEA M. and EGAN C., *Understanding Prediction Limits Through Unbiased Branches*, submitted to the 11th ACSAC2006, China, 2006.