Memory wall – a critical factor in current high-performance microprocessors

A. Florea - A. Gellert

University of Sibiu, Computer Science Department, Romania

Abstract. – *The memory access time is a critical factor that limits the performance of current microprocessors. Critical high-latency instructions caused by cache misses can slow a processor well below its peak potential. Pointer chasing from applications are responsible for reducing the overall processor performance because it creates a serialization in the execution of long-latency memory operations, inhibiting the overlap. In this paper we show some statistics on SPEC2000 benchmarks related to critical load instructions (loads that access with miss the L2 data cache and reached the head of the ROB). We obtained that 29.13% of critical loads appear at a distance at most 2 instructions. We also determined how many critical load instructions are not overlapping their access to main memory. In average 10.91% of critical loads are isolated or in front of a group of loads.*

Different approaches in overcoming the memory wall

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* [3] – 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 conducted to design microarchitectures able to overcome the memory wall by introducing techniques for efficient management of architectural resources (Reorder Buffer – ROB, register files, instruction queues and load/store queues). Akkary et al. [1] introduced the *Checkpoint Processing and Recovery* microarchitecture – ROB-free and requiring only a small number of rename map checkpoints selectively created at low-confidence branches, while capable of supporting a large instruction window of the order of thousands of instructions. In [5] the authors propose *runahead execution* as an effective way to increase memory latency tolerance in an out-of-order processor, without requiring an unreasonably large instruction window. Runahead execution unlocks the instruction window blocked by long latency operations allowing the processor to execute far ahead in the program path. In [4] is proposed a hybrid mode of execution based on ROB and checkpointing that decouples resource recycling and instruction retirement – *Checkpointed Early Resource Recycling* (*Cherry*). In this approach it is used a single checkpoint outside the ROB to divide it 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.

Cristal et al. [3] proposed the *Kilo-Instruction Processors* (*KIP*) – 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 *pointer chasing* and *hard to predict branches*. *Pointer chasing* is even more harmful than hard-to-predict branches, because it creates a serialization in the execution of long-latency memory operations, inhibiting the overlap. To solve this problem, value prediction techniques might be useful for predicting the addresses along a pointer chain, thereby allowing the overlap of these accesses. For memory intensive workloads with heavy pointer chasing, sequential cache-misses resulting from pointer chasing code structures dominate the overall execution time. These cache-misses form a memory dependence chain since the address of a missing load is dependent on the value of the previous missing load. Taking as an example the frequently executed code segment from *refresh_potential* function (see below) belonging to *mcf* benchmark, the profile information shows that the pointer chasing codes '*node->child*', '*node->basic_arc->cost*' and '*node->pred->potential*' result in many cache misses [10].

```
while( node )
{
  if( node->orientation == UP )
        node->potential = node->basic_arc->cost + node->pred->potential; // (Nodes 1, 2, 3, 4)
  else /* == DOWN */
  {
        node->potential = node->pred->potential - node->basic_arc->cost;
        checksum++;
  }
  tmp = node;
  node = node->child; // (Nodes 0, 5)
}
```

The memory dependence chains formed by these missing loads are shown below (see Figure 1).

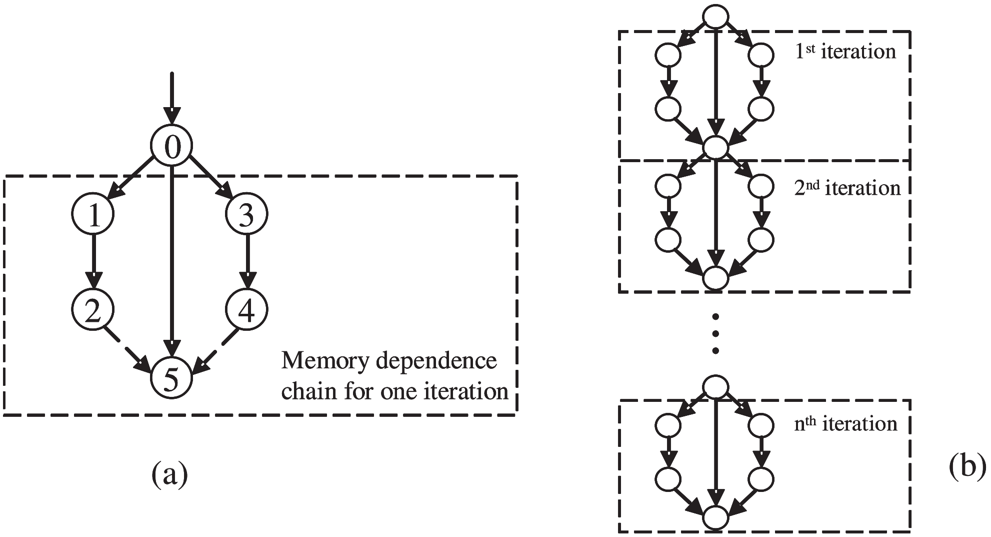


FIGURE 1. – *The memory dependence chain based on the above code. (a) The dependence chain for a single iteration. (b) The dependence chain for multiple iterations.*

In Figure 1(a), the dependence chain is based on a single iteration of the *while* loop, where nodes 1 and 2 correspond to two dependent missing loads from '*node->basic_arc->cost*'. Nodes 3 and 4 correspond to '*node->pred->potential*'. Node 5 corresponds to '*node->child*' and node 0 is the same load '*node->child*' from the previous iteration. Figure 1(b) shows the dependence chain when the loop is unrolled multiple times. The solid arrows in Figure 1 represent the true data dependences and the dashed arrows represent the alias dependences between missing loads.

The Simulated Architecture and Results

Starting from these challenges we made some analysis based on Simplescalar-3.0 tool set [2]. 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 [7]. Figure 2 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 that access with miss the L2 data cache and reached 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 [8]. To perform this job we ran the simulators on SUN machines with Sparc processors under UNIX operating systems.

We first determined the number of instructions *decoded* between two load instructions that are dispatched (loads are moved from the Cache Processor

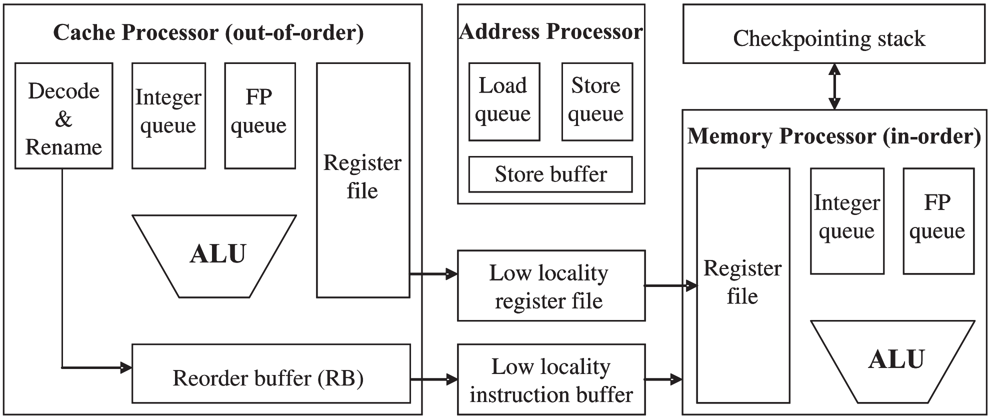


FIGURE 2. – *The Decoupled Kilo-Instruction Processor.*

during the writeback stage into the execution units LQ/SQ of the address processor or into LLIB) [7]. The simulation results show that 2 loads follows one after another in 27.28% of cases, at average. The operations on dynamic data structures met in applications with link list, trees or hash tables are responsible for the previous result.

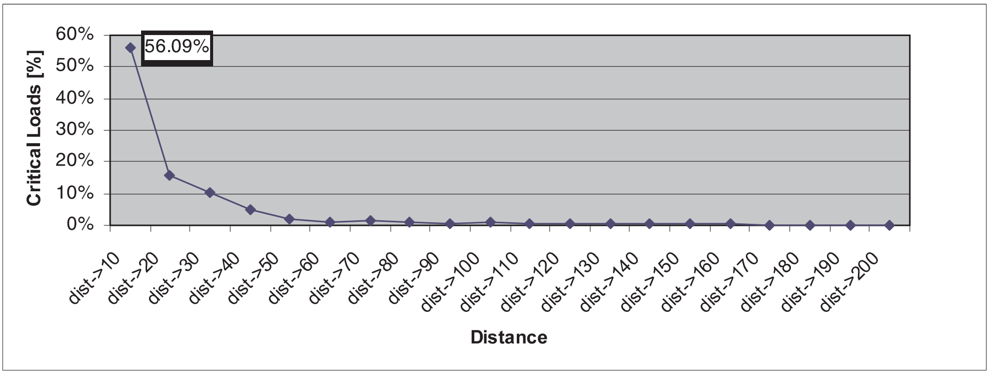


FIGURE 3. – *Average percentage of consecutive critical loads occurred at different distances. Coarse-grained statistics on SPEC*2000 *regarding the number of instructions that are committed between consecutive critical loads.*

Next, we take statistics between 2 load instructions that access with miss the Data Cache L2 and they are situated in the head of ROB (*critical*). The critical loads could be used for a more selective checkpoint mechanism. We determined the number of instructions that make commit phase between such two consecutive memory accesses. We obtained that 29.13% of critical loads appear at a distance at most 2

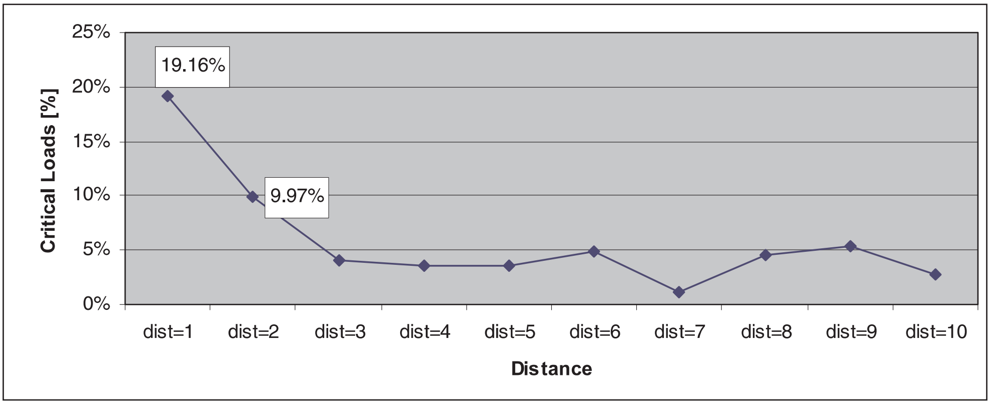
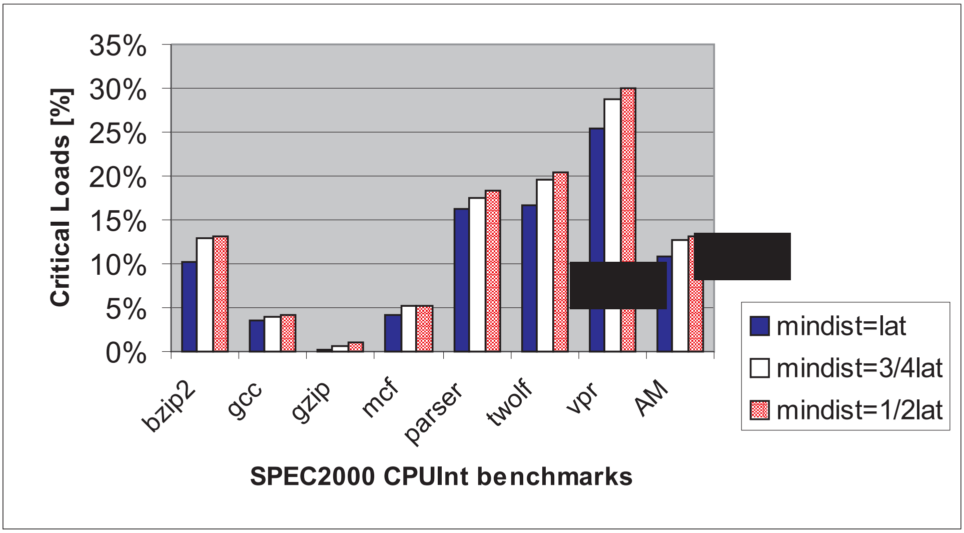


FIGURE 4. – *Average percentage of consecutive critical loads occurred at different distances. Fine-grained statistics on SPEC*2000 *regarding the number of instructions that are committed between consecutive critical loads.*

instructions (see Figure 4). We also found how many critical load instructions are not overlapping their access to main memory. We obtained that in average 10.91% of critical loads are isolated or in front of a group of loads (see Figure 5).



[redacted]

[redacted]

FIGURE 5. – *Percentage of critical loads without overlapping. Statistics regarding overlapped accesses to main memory of two critical loads.*

In the next step we determined how many instructions are committed between a critical load and the first dependent load instruction. The simulation results (see

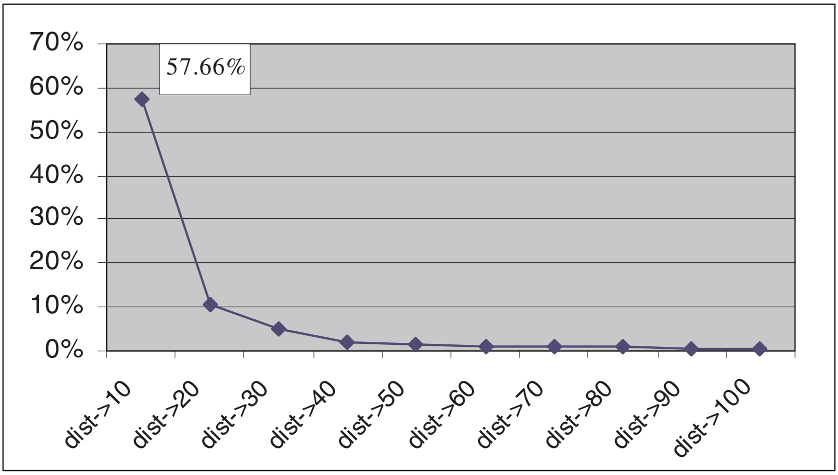


FIGURE 6. – *Average percentage of loads that depend on critical loads. Coarse-grained statistics regarding the number of instructions that are committed between a critical load and the first dependent load.*

Figure 7) show that dependent loads follow frequently immediately after a critical load (the dependent instruction is the first in 10.46% of cases, respectively the second in 11.22% of cases, after a critical load).

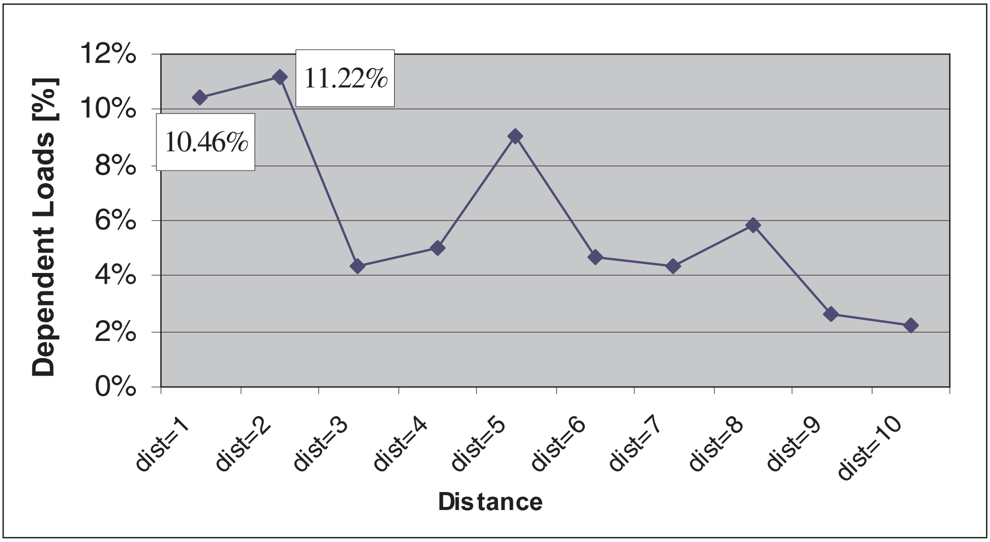


FIGURE 7. – *Average percentage of loads that depend on critical loads. Fine-grained statistics regarding the number of instructions that are committed between a critical load and the first dependent load.*

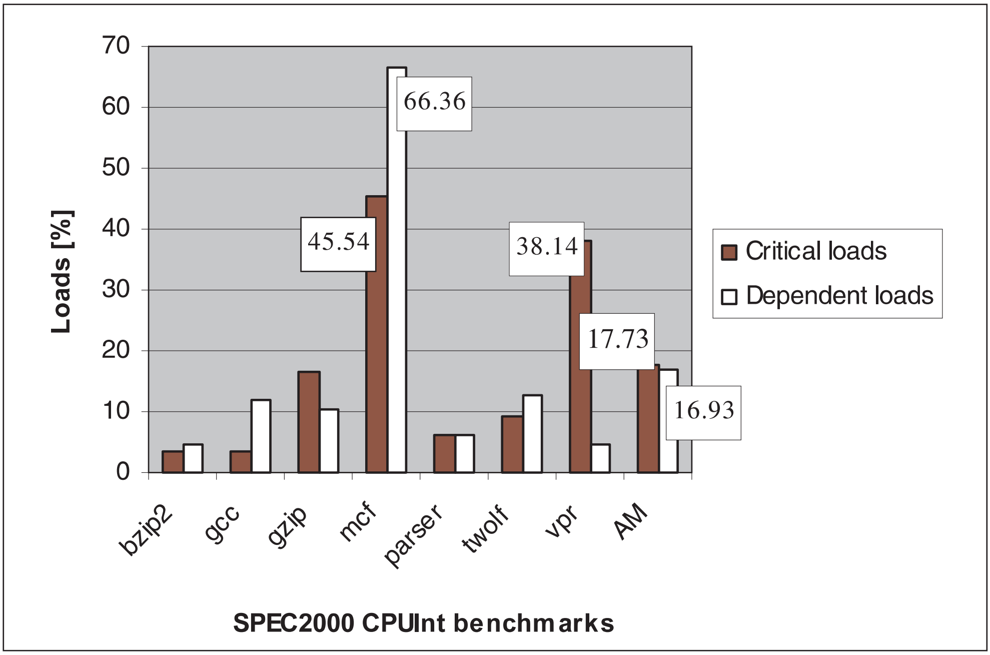


FIGURE 8. – *Comparative statistics regarding committed critical loads and dependent loads.*

Further, we compute the percentage of committed critical load instructions and also the percentage of dependent loads. The frequency of critical loads varies between 3.46% (*bzip2*) and 45.54% (*mcf* – natural, taking into account the Figure 1) with an average of 17.73%. The highest percentage of dependent loads on *mcf* and *vpr* proves that respective benchmarks are characterized by a heavy pointer chasing, and also, their overall performance is quite limited [5]. Also, it can be observed that, in average, one of six (≈ 16.93%) load instructions depends on a critical load. In the case of *mcf* benchmark, at least 11.90% of dependent loads are also critical. Except *gzip* and *vpr* benchmarks, the percent of dependent loads is greater than that of critical loads. Since every dependent load has attached only a critical load, likely more than one load instruction depend on the same critical instruction. Also, it is possible that some of the dependent loads to be even critical. However, the rest of dependent loads generate hit in the cache hierarchy (L1 or L2) and they would be quickly executed if the critical loads (address and /or value) would be correctly predicted. Mutlu et al. [6] proposes a solution for this kind of situations. Exploiting regular memory allocation patterns his value and address-based prediction technique enables the pre-execution of dependent instructions, including load instructions that incur long-latency cache misses.

Future Work

Further we'll continue our research trying to determine how many branches depend on a critical load. We want to find if these *low execution locality* branches [7] are unbiased [9] and how we can model branch predictors dedicated for these branches in KIP. Another idea is based on the feasibility of register/instruction centered value predictor 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.

References

[1] Akkary H., Rajwar R. and Srinivasan S.T., *Checkpoint Processing and Recovery: Towards Scalable Large Instruction Window Processors*, Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture, ACM Press, 2003.

[2] Burger D. and Austin T., *The Simple Scalar tool set*, version 2.0, Technical Report 1342, University of Wisconsin - Madison Computer Sciences Department, 1997.

[3] 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**, n. **3**, 2005.

[4] Martinez J., Renau J., Huang M., Prvulovic M. and J. Torrellas, *Cherry: Checkpointed early resource recycling in out-of-order microprocessors*, in *Proc. of the 35th Intl. Symposion On Microarchitecture*, 2002.

[5] Mutlu O., Stark J., Wilkerson C. and Patt Y.N., *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, 2003.

[6] Mutlu O., Kim H. and Patt Y., *Address-Value Delta (AVD) Prediction: Increasing the Effectiveness of Runahead Execution by Exploiting Regular Memory Allocation Patterns*, Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture, Barcelona, 2005.

[7] 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.

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

[9] Vintan L., Gellert A., Florea A., Oancea M. and Egan C., *Understanding Prediction Limits Through Unbiased Branches*, submitted to the 11th ACSAC, 2006, China, 2006.

[10] Zhou Y. and Conte T., *Enhancing Memory Level Parallelism via Recovery-Free Value Prediction*, Proceedings of the 17th International Conference on Supercomputing, USA, 2003.