Modeling Superscalar Processor Memory-Level Parallelism

Sam Van den Steen and Lieven Eeckhout

Abstract—This paper proposes an analytical model to predict Memory-Level Parallelism (MLP) in a superscalar processor. We profile the workload once and measure a set of distributions to characterize the workload's inherent memory behavior. We subsequently generate a virtual instruction stream, over which we then process an abstract MLP model to predict MLP for a particular micro-architecture with a given ROB size, LLC size, MSHR size and stride-based prefetcher. Experimental evaluation reports an improvement in modeling error from 16.9 percent for previous work to 3.6 percent on average for the proposed model.

Index Terms—Modeling, memory level parallelism (MLP), micro-architecture

1 Introduction

Analytical performance models are useful to speed up design space exploration and provide a mental model for architects to reason about application-architecture interactions. One of the important and challenging components to model is the total time an application spends waiting for main memory. This is a challenge in particular because current superscalar processors hide part of the memory access latency by exploiting *Memory-Level Parallelism (MLP)*, i.e., by servicing multiple memory requests in parallel [1].

The key contribution in this paper is an analytical model that requires profiling an application of interest only once, after which MLP can be accurately estimated for a range of superscalar processor architectures while varying reorder buffer (ROB) size, last-level cache (LLC) size, number of Miss Status Handling Registers (MSHR) entries, stride-based hardware prefetching, etc. This is a non-trivial endeavor as MLP is a result of complex interactions between an application's inherent memory access patterns and the available hardware resources. More specifically, characteristics such as burstiness of misses, inter-load dependences, locality, memory access patterns, etc. have an (in)direct effect on the exploitable MLP. Existing models fall short in various dimensions, as we will discuss in the next section.

The MLP model proposed consists of three steps. We first collect a number of distributions to characterize the relative positions of memory references in the instruction stream (to model burstiness), their dependences (dependent loads cannot be processed in parallel), their reuse distances (to model temporal locality), and their strided access patterns (to model spatial locality and their prefetchability). In the second step, we generate a virtual instruction stream with characteristics following these distributions. Finally, in the third step, we estimate MLP by processing the virtual instruction stream using an abstract MLP model; we take burstiness, inter-load dependences, locality and strided access patterns into account to estimate the amount of MLP.

Our experimental evaluation reports significant improvements over prior work. Compared to detailed simulation, the proposed MLP model achieves an average absolute error of 3.6 percent for predicting the total time waiting for main memory for a superscalar processor (with a stride-based prefetcher).[1] This is a significant improvement over prior work with a 16.9 percent average error [2].

• *The authors are with Ghent University, Belgium.*
E-mail: {sam.vandensteen, lieven.eeckhout}@ugent.be.

Manuscript received 24 Jan. 2017; revised 19 Apr. 2017; accepted 30 Apr. 2017. Date of publication 3 May 2017; date of current version 19 Mar. 2018.
(Corresponding author: Sam Van den Steen)
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/LCA.2017.2701370

2 Prior Work

There exist a number of models to predict MLP in superscalar processors. Karkhanis and Smith [3] consider the number of independent cache misses within an ROB-sized sequence of instructions from the dynamic instruction stream as a measure for MLP. Chen et al. [4] refine that model and consider pending cache hits, prefetching and MSHR registers for estimating the time spent waiting for main memory. A key limitation of these works is the reliance on a cache simulator to generate a stream of main memory accesses. This implies that the memory access stream needs to be re-generated whenever a change in the cache hierarchy is considered. Our model on the other hand requires profiling the application only once. Miftakhutdinov et al. [5] describe hardware extensions to accurately measure the time waiting for memory on real hardware with the goal of dynamically steering voltage and frequency for optimum energy efficiency.

The model proposed in this paper is most closely related to our own previous work in which we model superscalar processor performance from a micro-architecture independent profile [2]. This prior model includes a fairly simple MLP model, which basically assumes that conflict and capacity misses are uniformly distributed across the instruction stream, and that the burstiness in cache miss behavior results from cold misses. This paper presents a significant improvement over this prior work.

Wang et al. [6] propose a stochastic DRAM access model which assumes that a memory access stream from the processor side is given. In contrast, our work focuses on the processor side; we plan to combine our model with a model that captures the memory side as part of our future work.

Our approach also bears some similarity with prior work in statistical simulation, see for example [7] for a paper specifically focusing on modeling the memory data flow in superscalar processors. In contrast to statistical simulation which generates a *synthetic* trace that is then simulated using a detailed processor timing model, we generate a *virtual* instruction stream which is then processed by an abstract analytical MLP model.

3 MLP Model

Our MLP model relies on a number of statistics that we capture on a per micro-trace basis. A micro-trace is a short sequence of instructions (e.g., 1,000 instructions);[2] we collect 100 micro-traces per 100 M instructions (one micro-trace per 1M instructions). The reason for considering micro-traces is to reduce profiling time, and more importantly, to be able to capture MLP burstiness—an average profile across a number of micro-traces would average out the statistics which would compromise model accuracy.

Within each micro-trace, we measure a load-spacing distribution, inter-load dependence distribution, reuse distance distribution and stride distribution. Fig. 1 serves as an illustrative example: It shows a trace of 32 instructions consisting of 16 loads with the oldest instruction appearing on the left. Loads are indicated as L_x with x indicating recurrences of the same *static* load instruction. Dependences between loads are shown through arrows; the addresses accessed are shown below the loads. We collect these distributions for each static load in each micro-trace.[3]

1. The model is available at https://github.com/samvandensteen.
2. We find a micro-trace of 1,000 instructions to strike a good balance between profiling speed and model accuracy.
3. Collecting distributions requires on average 25× less disk space than recording micro-traces.

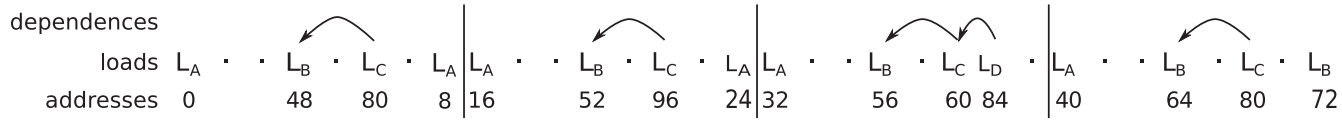

Fig. 1. Illustrative virtual memory access stream.

3.1 Load Spacing Distribution

We profile the positions of recurrences of each static load in the micro-trace using a load spacing distribution that records a load's first position in the micro-trace along with the number of instructions in-between recurrences of the same static load. For load L_C in Fig. 1, the load spacing distribution equals '$5;(8,3)$' meaning that the first occurrence appears at position 5 and there are 8 instructions between the next three recurrences. The rationale behind the load spacing distribution is to capture the burstiness of loads, i.e., load instructions that miss in the on-chip caches and that occur within the same ROB is a necessary condition to expose MLP.

3.2 Inter-Load Dependence Distribution

The inter-load dependence distribution quantifies inter-load data dependences in a statistical way. Inter-load dependences have an important impact on MLP, i.e., loads that depend upon each other (either directly or indirectly) cannot be issued simultaneously, hence they cannot expose MLP. The inter-load dependence distribution quantifies the probability that a load depends on any of the n previous loads in the instruction stream. For example, in Fig. 1, load L_C (always) depends on load L_B. Because of this dependence, loads L_B and L_C will serialize their execution, and hence no MLP can be exploited.

3.3 Reuse Distance Distribution

The reuse distance distribution quantifies temporal locality by quantifying the number of (not necessarily unique) memory accesses between two accesses to the same memory location. This reuse distribution is then transformed using StatStack [8] into a stack distance distribution, which quantifies the number of *unique* accesses between two accesses to the same memory location. Once the stack distance distribution is known, it is trivial to derive the miss rate assuming a fully associative LRU cache of arbitrary size, i.e., if there are more unique accesses between two accesses to the same memory address than there are sets in the cache, the last access to the same memory address will be a miss. Note that the reuse distance distribution is measured per static load, hence it enables estimating the miss rate per static load for any cache size. Moreover, we can use the reuse distance distribution for predicting hits and misses at all levels of cache, from the L1 cache to the LLC.

3.4 Stride Distribution

The last distribution we consider is the stride distribution. A stride is defined as the relative memory address difference between two subsequent recurrences of the same static load. The stride distribution collects this stride information. Whereas the reuse distance distribution quantifies temporal locality in a statistical way, the stride distribution is a measure for spatial locality. For example, a load that follows a strided access pattern with stride equal to 4, i.e., it accesses the following stream of memory locations: $0, 4, 8, 12, \ldots$ will result in a cache miss for every other load assuming a cache line size of 8 bytes. The stride distribution is also critical to model stride-based prefetching, as we will describe in Section 5.

Memory accesses do not always follow (in fact, they rarely follow) a neat stride pattern, i.e., some patterns can be a mixture of several strides, other memory accesses may appear to be random. We classify loads into three categories based on their access patterns. The first category includes loads that follow *some* stride pattern. The second category includes loads that occur only once in our micro-trace. The third category includes loads that do not fit in either of the above two categories; we refer to this category as random-strided loads.

For the strided-load category, we search for up to four distinct strides per load, and we use a cutoff percentage to filter out accesses that are not part of a real stride pattern. To categorize a load as an instruction with a single stride, one element in the stride distribution needs to have a percentage of occurrence of at least 60 percent. For a two-strided load, their cumulative percentage needs to exceed 70 percent, for a three-strided load 80 percent, and for a four-strided load 90 percent. We always choose the simplest stride pattern; this means that if the cumulative percentage of occurrence exceeds a threshold, we stop searching for additional strides, such that we can easily filter out random strides.

In Fig. 1, L_A recurs six times and exhibits a single-strided pattern with stride 8. Load L_B recurs five times with memory addresses: $48, 52, 56, 64, 72$. There are two strides of 4 and two strides of 8. Each stride thus has an occurrence equal to 50 percent, hence this load is classified as a two-strided load.

3.5 Putting It All Together

The distributions as just described need to be collected only once per application, from which we can predict MLP for a range of architecture configurations. We first generate a virtual instruction stream from these distributions; this virtual instruction stream is built up as a data structure by the MLP modeling software. We then hover over this virtual instruction stream with an abstract MLP model to estimate the amount of MLP for a particular architecture. This is done for each micro-trace.

3.5.1 Virtual Instruction Stream Generation

The load spacing distribution is first used to build up a skeleton virtual instruction stream. We position loads in the instruction stream using the load spacing distributions which determine the first position of each static load in the stream as well as the subsequent recurrences of the load; this is done for all static loads in the micro-trace. We then use the stride distribution to assign (relative) memory addresses for each load occurrence of the same static load. The stride distribution points out hits and misses in the cache, at least for those loads that exhibit a strided access pattern. We predict hits and misses at all levels in the cache hierarchy. More in particular, we mark the first access of a stride pattern as a miss and we mark the following accesses that fit the same cache line as hits. We use the reuse distance distribution and StatStack [8] to predict whether an address has been used before and the respective load will turn into a hit or a miss. We leverage the inter-load dependence distribution to impose dependences between loads.

3.5.2 Abstract MLP Model

The abstract MLP model then hovers over this virtual instruction stream to estimate MLP for a particular architecture with a specific ROB size. MLP is defined as the number of outstanding memory requests (LLC misses) if at least one is outstanding. The abstract model breaks up the virtual instruction stream into ROB-sized instruction sequences over which it estimates the available MLP.[4]

4. We considered two possibilities: An ROB that slides versus steps over the instruction stream; both gave similar results according to our preliminary results, hence we opt for the stepping approach which is slightly simpler to implement.

MLP is affected by various factors including the number and burstiness of cache misses; inter-load dependences, i.e., two loads that depend on each other cannot be serviced simultaneously; and the ROB size, i.e., independent loads need to reside in the same ROB is a necessary condition for MLP. For a given ROB-size sequence of instructions, MLP is computed as the number of independent main memory accesses in the ROB. MLP for the micro-trace is computed as the average MLP across all ROB-sized instruction sequences.

4 Modeling MSHRs

The MLP model discussed so far makes a number of simplifying assumptions. It assumes that all independent memory references access main memory simultaneously; in addition, it does not consider hardware prefetching. This section and the next discuss extensions to the MLP model to overcome these assumptions.

Modern processors typically feature Miss Status Handling Registers (MSHR) to coalesce multiple requests to the same cache line. An MSHR entry is allocated upon an access to a cache line that is not yet outstanding. Subsequent requests to an already outstanding cache line are then coalesced, avoiding yet another request being sent to the next level in the memory hierarchy. The size of the Miss Status Handling Register is (obviously) limited, and hence it may limit MLP, i.e., a memory access to a not yet outstanding cache line may be stalled if the MSHR runs out of available entries.

In this work we consider an MSHR table at the L1 data cache level, however, the approach can be trivially generalized to MSHRs at other levels of cache. We predict whether the number of outstanding L1 data cache misses in the virtual instruction stream exceeds the number of MSHR entries. If it does, we compute a scaling factor that accounts for the extra latency added to the loads waiting for an available MSHR entry. This model differs from the one proposed by Chen et al. [4] in which the MLP is simply capped to an upper bound; our model puts a 'soft' cap on the MLP and models partially overlapping memory accesses.

We estimate the impact of a limited number of MSHR entries as follows. The micro-trace is split up into ROB-size sequences of instructions of which the first instruction is a (predicted) access to main memory and the last instruction the one that still fits within the ROB. The first few memory accesses that miss in L1 all fit in the MSHR table and are hence considered to execute in parallel. All subsequent main memory accesses that would overflow the MSHR table have to wait until one of the outstanding accesses is resolved. Hence, they only partially overlap with the previous accesses. We model this phenomenon by considering the time it has to wait for a free MSHR slot. Intuitively, this means that the first part of the latency is serialized and the remaining part is hidden underneath another access. This results in the following formula which puts a 'soft' cap on the exploitable MLP

$$MLP = DRAM_{MSHR} + DRAM_{wait} \cdot \frac{T_{DRAM} - T_{MSHR\,free}}{T_{DRAM}}$$

with $DRAM_{MSHR}$ the number of main memory accesses in the MSHR table, i.e., this is the number of parallel main memory accesses; $DRAM_{wait}$ is the number of main memory accesses that have to wait; T_{DRAM} equals the main memory access latency and $T_{MSHR\,free}$ is the average time before an MSHR slot becomes available, which is computed as the weighted average access latency across all allocated MSHR entries.

5 Modeling Stride-Based Hardware Prefetching

A key feature of the proposed MLP model is that it enables estimating the performance impact of stride-based prefetching. In this work, we consider a stride prefetcher that is able to track the stride patterns of a number of static loads (per-PC stride prefetching) [9].

TABLE 1
Reference Architecture, Based on Intel Nehalem

Core frequency	2.66 GHz
Dispatch width	4
ROB	64, 96, 128, 160, 192, 224, 256 entries
L1I and L1D	32 KB, latency = 1 and 4 cycles, respectively
L2	256 KB, latency = 8 cycles
LLC	1, 2, 4, 8, 16:MB, latency = 30 cycles
MSHR	Between L1D and L2, entries = 10
Prefetcher	stride prefetcher, streams = 16
Memory bus	Bandwidth = 7.6 GB/s
DRAM	latency = 45 ns

If a load exhibits a stride pattern, we mark it as a prefetchable load, subject to a number of constraints as discussed below.

A stride prefetcher needs to keep track of previously executed loads and their addresses to compute a load's stride pattern. There is obviously a limit to the number of static loads the prefetcher is able to track. If there are more static loads than the maximum tractable loads in the prefetch table, the load is marked non-prefetchable.

Second, prefetchers often only prefetch within a DRAM page, meaning that if two subsequent accesses are not part of the same DRAM page, the second one will not be prefetched. We also model this by considering the stride between two subsequent accesses by the same load; if the stride exceeds a DRAM page, we mark the load as non-prefetchable.

The third component relates to timeliness. If the prefetcher starts fetching new data just before the data is requested, the prefetch will not be timely, and the latency of the load will be hidden only partially. We model this by assuming that a prefetch for a load that is ROB-size instructions away in the dynamic instruction stream, is timely. If the load appears in the same ROB-size instruction window as the prefetch, we then subtract the fraction of the latency equal to the time it would take for the latter load to hit and stall the ROB head.

6 Evaluation

6.1 Experimental Setup

We consider the 29 SPEC CPU2006 benchmarks[5] which we simulate using Sniper v6.0 [10]. We use a periodic sampling strategy to limit experimentation time while still covering the entire benchmark execution. To compute the ground truth to evaluate the model against, we fast-forward 800 M instructions, warm up the memory hierarchy for 100 M instructions, and then simulate 100 M instructions in detailed mode; this is repeated till the end of the execution. We consider a similar sampling strategy for collecting our profile: We fast-forward 900 M instructions and collect our profile during the next 100 M instructions; this procedure guarantees that the profile corresponds to the detailed simulation region. The simulated processor is based on the Intel Nehalem architecture; see Table 1 with our reference architecture shown underlined. We assume a fixed memory access latency.

6.2 Accuracy W/O Prefetching

We evaluate the MLP model's accuracy by quantifying the total time spent waiting for DRAM. In Sniper, the DRAM cycle component is measured as the number of cycles between a load miss accessing main memory and blocking the head of the ROB [11]. In our model, we estimate the DRAM component by multiplying the estimated number of LLC misses times DRAM access latency divided by the predicted MLP. Fig. 2 reports the model's accuracy

5. We use train inputs as we run the benchmarks to completion — using the reference inputs would be infeasible, even with the employed sampling strategy.

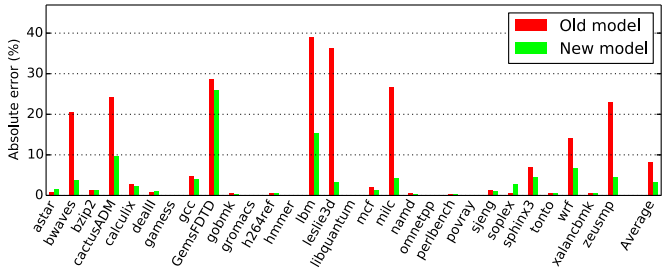

Fig. 2. Absolute error for predicting total time waiting for DRAM for the new and old MLP models, assuming no hardware prefetching.

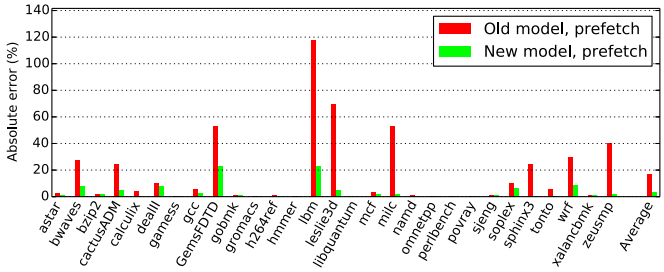

Fig. 3. Absolute error for predicting total time waiting for DRAM for the new and old MLP models, assuming hardware stride prefetching.

against simulation. The average absolute error equals 3.3 percent. The highest error for the new model is observed for gemsFDTD (26.0 percent).

The new model is substantially more accurate than our previous model [2], which essentially assumes that conflict and capacity misses are uniformly distributed across the execution whereas cold misses incur bursty cache misses. The old model achieves an average absolute error of 8.2 percent and a maximum error of 39.1 percent. Modeling the relative spacing of memory references, their dependences and strides clearly leads to a more accurate model.[6]

6.3 Accuracy W/ Prefetching

The results reported so far did not consider hardware prefetching. Fig. 3 reports the absolute prediction error assuming a stride-based prefetcher. The new model achieves an average absolute prediction error of 3.6 percent and at most 22.8 percent. The old model, which does not model stride-based prefetching, leads to an absolute average prediction of 16.9 percent and absolute errors up to 118 percent. This re-emphasizes the importance of incorporating the impact of hardware prefetching in an analytical MLP model.

6.4 Design Space Exploration

The most obvious use case for the model is to drive design space exploration. We consider 35 processor designs in total while varying two micro-architecture parameters that do have an immediate effect on MLP, namely ROB size (7 sizes) and LLC size (5 sizes), see also Table 1. This rather limited design space already takes considerable simulation time—more than 5 years of single core simulation time as some of the benchmarks take over one week. Profiling the benchmarks is a one-time cost, and the proposed MLP model (including the cost of profiling) evaluates the same design space 160× faster.

More than 90 percent of the designs have an absolute error below 15 percent for the new MLP model, whereas for the old model less than 80 percent of the designs have an absolute error below 15 percent. The largest errors are typically observed for unbalanced processor designs (e.g., a big ROB with 256 entries along with a relatively small 1 MB LLC). If we plug the new MLP model into the complete performance prediction model, we see an average improvement of 2.2 percent.

7 Conclusions and Future Work

This paper proposed a novel model for estimating MLP in a superscalar processor by considering a set of distributions regarding the relative position of memory references in the instruction stream, their dependences, reuse distances and stride behavior. Generating a virtual instruction stream using these distributions enables modeling the impact of the processor's ROB size, number of MSHR entries, LLC size and stride-based prefetching on MLP, from a single profile. The model was shown to improve accuracy for predicting the total time waiting for DRAM from 16.9 to 3.6 percent on average.

This paper made a number of simplifying assumptions. In particular, we focused on the processor side only and did not consider the impact DRAM may have on the exploitable MLP. We essentially assumed that all memory requests that are sent out by the processor are serviced simultaneously by the DRAM subsystem. We will extend the proposed model to consider a more realistic DRAM subsystem as part of our future work. In addition, we plan to incorporate the impact of multi-core processing and multi-threaded workloads in the model (including interference in shared resources, coherence, synchronization), as well as more advanced hardware prefetchers.

6. As a side note, it is interesting to note that the new model is also much faster to profile while consuming less memory. A typical profiling run takes approximately 40 percent less time and consumes up to 0.5 GB less memory.

Acknowledgments

We thank the anonymous reviewers for their constructive and insightful feedback. Sam Van den Steen is supported through a doctoral fellowship by the Agency for Innovation by Science and Technology (IWT).

References

[1] Y. Chou, B. Fahs, and S. Abraham, "Microarchitecture optimizations for exploiting memory-level parallelism," in *Proc. 31st Annu. Int. Symp. Comput. Archit.*, 2004, pp. 76.

[2] S. Van den Steen, S. De Pestel, M. Mechri, S. Eyerman, T. Carlson, D. Black-Schaffer, E. Hagersten, and L. Eeckhout, "Micro-architecture independent analytical processor performance and power modeling," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw.*, 2015, pp. 32–41.

[3] T. S. Karkhanis and J. E. Smith, "A first-order superscalar processor model," in *Proc. 31st Annu. Int. Symp. Comput. Archit.*, 2004, pp. 338.

[4] X. E. Chen and T. M. Aamodt, "Hybrid analytical modeling of pending cache hits, data prefetching, and MSHRs," *Proc. 41st Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2011, pp. 59–70.

[5] R. Miftakhutdinov, E. Ebrahimi, and Y. N. Patt, "Predicting performance impact of DVFS for realistic memory systems," in *Proc. 45th Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2012, pp. 155–165.

[6] Y. Wang, G. Balakrishnan, and Y. Solihin, "MeToo: Stochastic modeling of memory traffic timing behavior," in *Proc. Int. Conf. Parallel Archit. Compilation*, 2015, pp. 457–467.

[7] D. Genbrugge and L. Eeckhout, "Chip multiprocessor design space exploration through statistical simulation," *IEEE Trans. Comput.*, vol. 58, no. 12, pp. 1668–1681, Dec. 2009.

[8] D. Eklov and E. Hagersten, "Statstack: Efficient modeling of LRU caches," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw.*, 2010, pp. 55–65.

[9] J. W. Fu, J. H. Patel, and B. L. Janssens, "Stride directed prefetching in scalar processors," *Proc. 25th Annu. Int. Symp. Microarchitecture*, 1992, pp. 102–110.

[10] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2011, Art. no. 52.

[11] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith, "A performance counter architecture for computing accurate CPI components," in *Proc. 12th Int. Conf. Archit. Support Program. Languages Oper. Syst.*, 2006, pp. 175–184.