AFEC: An Analytical Framework for Evaluating Cache Performance in Out-of-Order Processors

Kecheng Ji*, Ming Ling*, Qin Wang*, Longxing Shi*, Jianping Pan†
*National ASIC System Engineering Technology Research Center, Southeast University, Nanjing 210096, China
Email: {jikecheng, trio, 220153670, lxshi}@seu.edu.cn
†Department of Computer Science, University of Victoria, Victoria, British Columbia, Canada
Email: pan@uvic.ca

***Abstract*—Evaluating cache performance is becoming critically important to predict the overall performance of out-of-order processors. Non-blocking caches, which are very common in out-of-order CPUs, can reduce the average cache miss penalty by overlapping multiple outstanding memory requests and merging different cache misses with the same cacheline address into one memory request. Normally, memory-level-parallelism (MLP) has been used as a metric to describe the concurrency of memory access. Unfortunately, due to the extremely dynamic dependences among the program memory references, it is very difficult to quantify MLP without time-consuming simulations. Moreover, the merging of multiple cache misses, which makes the average cache miss service time less than the physical DDR access latency, is seldom considered in the existing researches. In this paper, we propose a cache performance evaluation framework based on program trace analysis and analytical models to fast estimate MLP and the effective cache miss service time without simulations.**

Comparing with the results by Gem5 simulations of MobyBench 2.0, Mibench 1.0 and Mediabench II, the average accuracy of the modeled MLP and the average cache miss service time is higher than 91% and 92%, respectively. Combined with cache misses calculated by the stack distance theory, the average absolute error of CPU stall time (due to cache misses) is lower than 10%, while the evaluation time can be sped up by 35 times relative to the Gem5 full simulations.

I. Introduction

Undoubtedly, cache performance influences the performance of CPUs significantly. How to evaluate the impacts of different cache architectures on the system performance, precisely and quickly, has become an attractive topic. While simulations, either in RTL or cycle-accurate ones, can give relatively accurate and detailed results, they are normally unacceptably time consuming when used as the design space exploration tools, especially in the era of complexity explosion of software and hardware. On the other hand, analytical models generally use the information collected from trace profiling as an input, and calculate the performance metrics from equations based on mechanism analysis or empirical fittings [1] [2]. Unfortunately, due to the extremely dynamic and complex nature of out-of-order processors, it is very difficult to precisely model the cache behaviour analytically in these architectures. This paper proposes a fast and precise framework based on analytical models to evaluate the memory-level-parallelism (MLP) and effective cache miss service time, when considering the influences of overlapping outstanding memory requests and the request merging mechanism in out-of-order processors [3].

Because the services of cache misses are never overlapped within in-order processors [4], the cache miss penalties in these architectures can be simply described as $Cache\_Misses * DDR\_Access\_Latency$. Thus, past studies merely adopted the stack distance theory to estimate the cost of cache misses, if the caches are equipped with the LRU cache replacement policy [5]. However, the non-blocking cache technology, which is very commonly seen in out-of-order processors, makes things complicated. In this case, miss status handle registers, or MSHRs, are introduced to support multiple outstanding memory requests. Unfortunately, traditional memory metrics, such as Cache Miss Rate (MR), Average Miss Penalty (AMP), and Average Memory Access Time (AMAT), are all measured based on a single access activity, and cannot reflect the reality of cache/memory concurrency [6]. That is why the memory-level-parallelism (MLP), which means the number of outstanding memory requests concurrently handled by the cache, must be considered as a factor to model the non-blocking cache performance [7]. According to recent researches, the CPU stall time caused by cache misses could be modeled as Eq. (1) [8]. However, the MLP is difficult to be modeled precisely without simulations [7].

$$Stall\_Time_{misses} = \frac{Misses * Average\_Service\_Time}{MLP} \quad (1)$$

By definition, the service time for each cache miss is the duration when the cache miss occurs until the service is completed [6]. Generally, most researches regarded the average cache miss service time as the DDR access latency, which is often quantified as a constant value [8]. However, we argue that the average service time for each cache miss is not simply equal to the DDR access latency due to the influence of some memory requests being merged by MSHRs. To the best of our knowledge, there is hardly any previous research considering this effect.

Our work in this paper improves upon the related works in the following aspects:

1) Proposing a fast method to estimate the MLP without time-consuming cycle-accurate simulations.
2) Revising the traditional cache miss penalty equation by introducing the effective average service time which considers the influence of memory requests that merged by MSHRs.

The rest of this paper is structured as follows: Section II introduces the related works. Section III describes the principle of our method to model the MLP. Section IV gives the effective service time model. Section V shows the experimentation setup and evaluations. Section VI concludes this paper.

II. Related Works

The increasing complexities of embedded systems make the methods based on simulations less appropriate in forecasting the cache performance due to their long running time [9]. In early stages, most researches combined binary instrumentation tools with constant timing information to imitate cycle-accurate simulators with faster speed [10]. However, the changes of timing information in out-of-order (OoO) CPUs render these methods less accurate [11] .

Reducing simulation times is another optimization direction in design space explorations (DSE) [12]. James E. Smith and Eyerman

978-3-9815370-8-6/17/$31.00 ©2017 IEEE

proposed a framework based on an empirical MLP to fast calculate cache misses penalties, in which the MLP was measured by a simulation running a special benchmark [2]. Roeland J. Douma gave a fast cache performance estimation method based on the simulation result of a "perfect cache" [11]. Although the framework considered the influences by OoO processors, the hypothesis that "if two memory requests overlap, the overlapping has to finish before another request can start" is not completely tenable [13]. Moreover, the framework still did not achieve the goal of being decoupled from simulations.

Analytical models, which are frequently employed in DSEs most recently [14], have the feature of fast speed but relatively low accuracy [9]. By estimating MLP as the total number of memory instructions in the current instruction window divided by the maximum number of memory instructions in a dependency chain, Jian Chen put forward an analytical model [15]. However, he did not consider the fact that each load/store may have more than one dependence path, which influences the MLP estimation. As our experiments in Section V show, their estimation results are often higher than the simulated MLP. In this paper, we propose a faster and more accurate framework based on analytical models to evaluate the cache performance in out-of-order processors.

III. MLP Modeling

With the purpose to simplify the model's complexity, we assume that the cache organization in this paper has independent instruction and data caches that are connected to the main memory (DDR SDRAM) via a crossbar. The caches are configured with the Least Recently Used (LRU) replacement policy. Meanwhile, the CPU is an ARMv7-a ISA single-core at 1.7 GHz with the traditional 7-stage out-of-order pipeline.

After being decoded and renamed, instructions will be dispatched into the instruction queue (for out-of-order scheduling) and the reorder buffer (for orderly committing). The reorder buffer (ROB) has an alias named "instruction window" in previous works [9]. Once back-end long latency miss events detected, such as D-Cache misses in this paper, the instruction window will be fully filled with following instructions immediately. Only the cache misses caused by load/store instructions in the same instruction window can be serviced concurrently. Based on these two facts, we construct an instruction window tracer, which is proposed by Eyerman in [16], to analyze instruction information and evaluate cache performance,. Interested readers can find the implementation details of instruction window tracer in that paper.

Essentially, the tracer can be implementated based on a binary instrumentation tool, such as PIN [17] or QEMU [18], which merely emulates software functions without hardware simulations. PIN is designed for x86 instruction set architecture while QEMU only does one-time decoding for each instruction which cannot reflect the behaviors of an instruction stream [18]. In this paper, we revise the Gem5 [19] source code and utilize its AtomicSimpleCPU mode to play the role of a binary instrumentation tool. The AtomicSimpleCPU mode simulates hardware behaviors without timing information so its speed is much faster than that of cycle-accurate full simulations.

A. Structure of the Framework

Fig. 1 gives the framework for cache performance modeling in this paper. In summary, the framework contains three abstract layers. The "Instruction Window Tracer and Analyzer" layer focuses on tracing and analyzing software information such as load/store accessing addresses, source-destination registers' index numbers and the number of cachelines that loads/stores would visit. Utilizing the information collected in the upper layer, the number of cache misses, memory-level-parallelism and the effective cache miss service time could be calculated by the "Analytical Modeling" layer. Lastly, based on Eq. (1), the whole cache performance can be completely estimated in the "Cache Performance Estimation" layer.

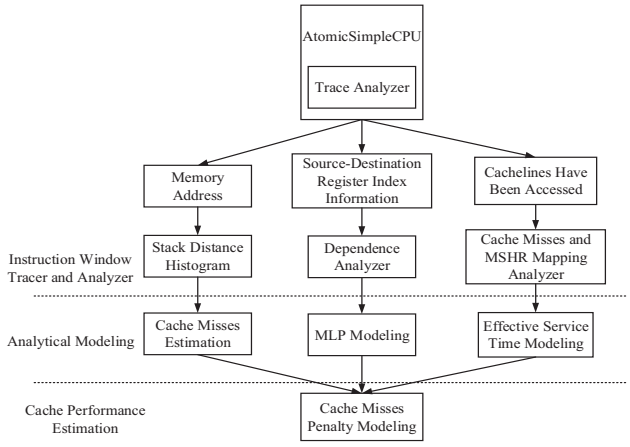

Fig. 1. Cache Performance Evaluation Framework in This Paper

B. Stack Distance Theory and Cache Misses

Equipped with the fully-associative LRU caches, the number of conflict cache misses in a single-core CPU can be calculated by using the stack distance theory [4]. Meanwhile, it can be easily extended to evaluate cache misses in set-associative caches by filtering addresses with the cache set domain [20], which is utilized in this paper. Furthermore, cold misses are estimated by recording memory footprints [1]. In a word, the whole cache misses can be described as Eq. (2).

$$Cache\_Misses = \sum_{assoc+1}^{\infty} H_s(d) + footprints \qquad (2)$$

$H_s(d)$ is the histogram of stack distances, while the $footprints$ mean the number of requests whose cacheline addresses are never accessed in the cache before. The $assoc$ gives the associativity of the cache.

C. Dependences and MLP Modeling

In cache performance evaluations, MLP means the number of cache misses that can be serviced concurrently in the system [6]. Hardware hazards, such as those caused by the limited MSHR capacity (entries or targets) or cache ports, influence MLP significantly. In order to simplify analytical model derivations, we do not consider the hardware effects by assuming infinite hardware resources within Section III and Section IV. The hardware limitation influences will be considered in Section V.

Fig. 2 gives an instance of dependences among different cache misses within an instruction window. Each circle represents a cache miss (CM), while the arrows express dependence directions. For example, the memory instruction of cache miss A relies on the execution result of another instruction that causes the cache miss B (indirectly or directly). In this case, the intruction of cache miss A cannot be serviced until B is completed. In summary, all cache misses along the same dependence path are non-overlapped, or in other words, must be processed in serial. Although MSHRs make cache misses be processed in parallel, the existence of non-overlapped cache misses separates this process into several phases. In this

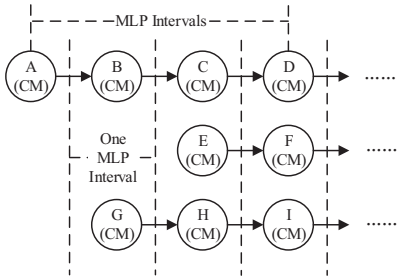

Fig. 2. MLP Intervals and Average MLP

paper, we define these non-overlapped phases as "MLP intervals". Previous works pointed out that the number of MLP intervals in an instruction window is equal to the maxium of non-overlapped cache misses among all dependence paths [15]. The average MLP should be estimated by using the total number of cache misses within an instruction window divided by the number of MLP intervals. In this case, the average MLP within a given instruction window is estimated as Eq. (3).

$$MLP_{avg\_per\_window} = \frac{Cache\_Misses_{per\_window}}{MLP\_Intervals_{per\_window}} \tag{3}$$

Assuming that each load/store along any dependence path has the same possibility to be a cache miss [15], the MLP intervals can be approximately calculated as $R_w * K_{max}$. For each instruction window w, R_w represents the cache miss rate in this window, while K_{max} is the number of loads/stores along the dependence path that contains the maximum loads/stores. Consequently, Eq. (3) can be manipulated as Eq. (4). N_{ls-w} means the total number of memory instructions in the current instruction window.

$$MLP_{avg\_per\_window} = \frac{N_{ls-w} * R_w}{K_{max} * R_w} = \frac{N_{ls-w}}{K_{max}} \tag{4}$$

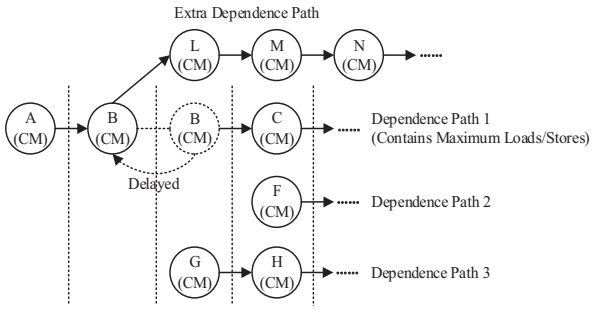

Fig. 3. Instances of Dependences and MLP

However, the number of MLP intervals is not always equal to $R_w * K_{max}$. In fact, a load/store instruction may have one or two source registers [21] and can be executed merely when all registers are ready. In Fig. 3, the memory instruction B would cause a cache miss and is supposed to be serviced with another memory miss G concurrently. Formally, the number of MLP intervals should be equal to the cache misses along the dependence path 1, which contains the maximum loads/stores. However, one of B's source registers is not ready due to cache misses on the extra dependence path. In this case, instruction B is very likely delayed into the next MLP interval. For each load/store on the dependence path 1, we consider that each extra dependence path will increase the number of MLP intervals by one. Consequently, Eq. (4) should be revised into Eq. (5), while P_{e-dep} is the possibility of having an extra dependence path for each load/store along the dependence path 1.

$$\begin{aligned} MLP_{avg\_per\_window} &= \frac{N_{ls-w}}{K_{max} + K_{max} * P_{e-dep}} \\ &= \frac{N_{ls-w}}{K_{max} * (1 + P_{e-dep})} \end{aligned} \tag{5}$$

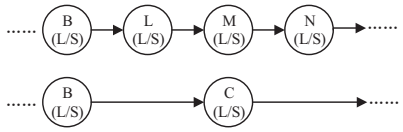

Fig. 4. Dependence Path Splitting

To achieve the value of $1 + P_{e-dep}$, we allocate a dependence path for each of the instruction's dependent source registers. Such as the instruction B in Fig. 3, which has an extra dependence path, is allocated two dependence paths as shown in Fig. 4. Same as previous researches [15], the trace analyzer in this paper uses comparisons between destination and source registers among different instructions to establish dependence relationships. Ultimately, for each instruction window, we maintain a dependence graph to express the memory dependence paths.

It is not difficult to find out that the physical meaning of $(1 + P_{e-dep})$ is actually the average number of dependence paths that each load/store instruction is on. We define the total number of load/store nodes in all dependence paths of an instruction window as $\sum K_{dep}$. The average dependence paths for a load/store instruction is calculated as $\frac{\sum K_{dep}}{N_{ls-w}}$. Based on Eq. (5), the final formula for estimating MLP should be shown as Eq. (6).

$$\begin{aligned} MLP_{avg\_per\_window} &= \frac{N_{ls-w}}{K_{max} * \frac{\sum K_{dep}}{N_{ls-w}}} \\ &= \frac{N_{ls-w}^2}{K_{max} * \sum K_{dep}} \end{aligned} \tag{6}$$

IV. Effective Service Time

A. MSHR Technology

Most current MSHR implementations support multiple address targets in one MSHR entry [19], [22], which merges memory requests with the same target address into one request. As a result, the average cache miss service time in this architecture is less than a single DDR access latency.

As shown in Fig. 5, the modern MSHR system is composed of several MSHR entries and in each MSHR entry, there are several request targets. Each MSHR entry accesses the DDR with the cacheline aligned address. Once a cache miss occurs, the generated memory request address will be compared with those in all valid MSHR entries. A new MSHR entry will be assigned if no existing MSHR entry stores the same cacheline aligned address as that of the cache miss. Otherwise, the request will be a target that can be merged into the matched MSHR entry.

B. Effective Service Time Modeling

Fig. 6 shows a normal case for merging cache misses merging. T_n represents the arrival time of the n_{th} cache miss within a DDR access interval (time from the initialization of the MSHR entry to service

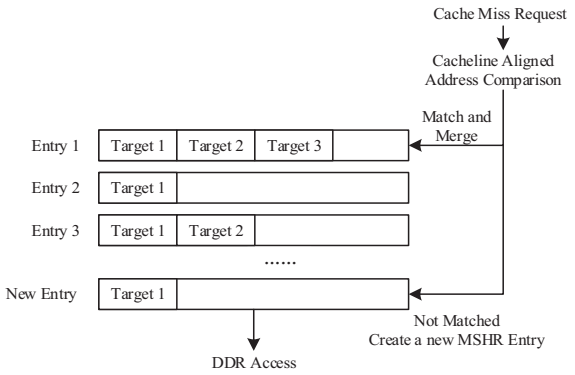

Fig. 5. Merging Memory Requests with The Same Cacheline Address into One MSHR Entry

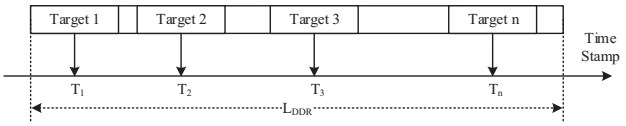

Fig. 6. A Normal Case for Merging Cache Misses

completion). Once the DDR access has been completely serviced, all merged cache misses will be satisfied simultaneously. Thus, the average service time (ST_{avg}) for each cache miss can be described as Eq. (7), in which L_{DDR} is the single DDR access latency.

$$\begin{aligned} ST_{avg} =& \{L_{DDR} + [L_{DDR} - (T_2 - T_1)] + [L_{DDR} - (T_3 - T_1)] \\ & \cdots + [L_{DDR} - (T_n - T_1)]\}/n \end{aligned} \tag{7}$$

Qualitatively, in each DDR access interval, we assume all merged cache misses are evenly distributed and the duration between two successive misses could be estimated as L_{DDR}/n (n is the number of merged cache misses in one MSHR entry, $n \geq 1$. T_2-T_1=L_{DDR}/n, T_3-T_1=$2L_{DDR}/n \cdots$). Thus, we can re-write Eq. (7) as Eq. (8).

$$\begin{aligned} ST_{avg} &= \frac{L_{DDR} * n - \frac{n(n-1)}{2} * \frac{L_{DDR}}{n}}{n} \\ &= \frac{L_{DDR}}{2}(1 + \frac{1}{n}) \end{aligned} \tag{8}$$

We define N_{mpc} as the average memory access times to each cacheline as shown in Eq. (9), in which N_{cl} denotes the number of different cachelines accessed in this instruction window. Hence, the average cache misses on every cacheline can be described as $N_{mpc} * R_w$. In each MLP interval, the number of cache misses per cacheline address is described as Eq. (10), which is the average number of targets per MSHR entry named as $N_{targets}$.

$$N_{mpc} = \frac{N_{ls-w}}{N_{cl}} \tag{9}$$

$$N_{targets} = \frac{N_{mpc} * R_w}{K_{max} * R_w} = \frac{N_{ls-w}}{K_{max} * N_{cl}} \tag{10}$$

Considering the calculation of MLP intervals in Eq. (6), Eq. (10) should be revised as shown in Eq. (11).

$$\begin{aligned} N_{targets} &= \frac{N_{ls-w}}{K_{max} * N_{cl} * \frac{\sum K_{dep}}{N_{ls-w}}} \\ &= \frac{N_{ls-w}^2}{K_{max} * N_{cl} * \sum K_{dep}} \end{aligned} \tag{11}$$

Combined with Eq. (8), Eq. (11) can be written as Eq. (12).

$$\begin{aligned} ST_{avg} &= \frac{L_{DDR}}{2}(1 + \frac{K_{max} * N_{cl} * \sum K_{dep}}{N_{ls-w}^2}) \\ &= \frac{L_{DDR}}{2}(1 + \frac{N_{cl}}{MLP}) \end{aligned} \tag{12}$$

V. Evaluations

A. Experiments Setup

In this paper, 9 benchmarks from MobyBench 2.0 [23] and 7 benchmarks from Mibench 1.0 [24] along with Mediabench II [25] are adopted in our model evaluations. The Gem5 in $arm\_detailed$ mode (cycle-accurate) works as the validation hardware platform while hardware configurations are shown in Table I. All evaluations are compared with Gem5 simulation results every 100,000 cycles. Taking the impacts of hardware MSHR entries (HME) and hardware targets per MSHR entry (HTPE) into account, the analytical models in Eq. (6) and Eq. (12) should be revised as Eq. (13) and Eq. (14), respectively.

TABLE I
Hardware Configurations

Parameters	CONF-1	CONF-2	CONF-3
Instruction Set Architecture	ARMv7-a		
Instruction Window Size	40	80	120
Cache Replacement Policy	Least Recently Used		
Cache Organization	16KB-2Way	32KB-4Way	64KB-4Way
MSHR Organization	Entries: 8 Targets: 4	Entries: 12 Targets: 8	Entries: 16 Targets: 8
DDR Latency	235 Cycles		
CPU Frequency	1.7 GHz		
Statistic Dump Cycles	100,000 Cycles		

B. Cache Misses Evaluation

Fig. 7 gives the cache misses evaluation results for all the benchmarks we have tested, while the hardware configuration is marked on the top of the figure. According to the results from our experiments, the average error of the cache misses model based on the stack distance theory in OoO processors is about 13%. These relatively large errors could be caused by the changes of memory access locality due to the non-blocking issues, out-of-order executions and other technologies [26] used in the modern out-of-order processors.

C. MLP Evaluation

Fig. 8 gives an instance(running the BBench) of the comparisons among the simulation results, our modeled MLP and the model proposed by Jian Chen [15]. The y-axis represents the MLP value and the x-axis gives the group number of the data. The hardware configuration is shown at the bottom of this figure. According to the

$$MLP = \begin{cases} \dfrac{N_{ls-w}^2}{K_{max} \times \sum K_{dep}}, & (\dfrac{MLP}{N_{targets}} < HME) \\ HME \times N_{targets}, & (others) \end{cases} \tag{13}$$

$$ST_{avg} = \begin{cases} \dfrac{L_{DDR}}{2}(1 + \dfrac{K_{max} \times N_{cl} \times \sum K_{dep}}{N_{ls-w}^2}), & (1 \le N_{targets} < HTPE) \\ \dfrac{L_{DDR}}{2}(1 + \dfrac{1}{HTPE}), & (N_{targets} \ge HTPE) \\ L_{DDR}, & (N_{targets} < 1) \end{cases} \tag{14}$$

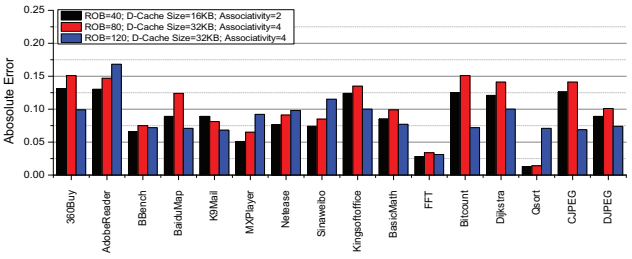

Fig. 7. Absolute Errors of Cache Misses Forecasting

results, our model can forecast the dynamic MLP accurately while Jian Chen's modeling results are higher than simulated ones. Fig. 9 gives all the absolute errors of our tested benchmarks in different hardware configurations. The average absolute error of our work is lower than 9%, while that of Jian Chen's work is around 18%.

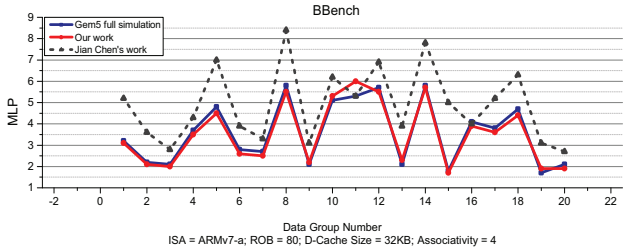

Fig. 8. MLP from Gem5 Simulations, Our Model and Jian Chen's Model

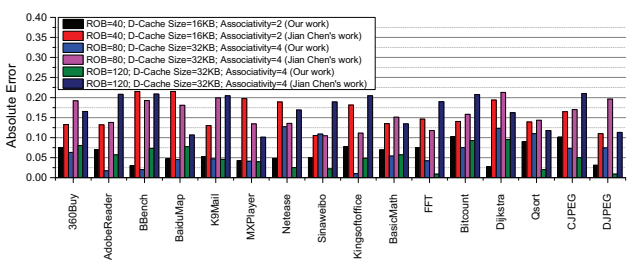

Fig. 9. Absolute Errors of Our Modeled MLP and that of Jian Chen's Work

D. Service Time Evaluation

Fig. 10 gives an example(running the MXPlayer) of the comparisons among the simulated average service time, our model's results and the constant DDR access latency. In most cases, our model reflects the real effective service time. Fig. 11 gives all the absolute errors for the benchmarks we have tested in different hardware configurations, and the lowest error is less than 2%, while the average error is around 8%.

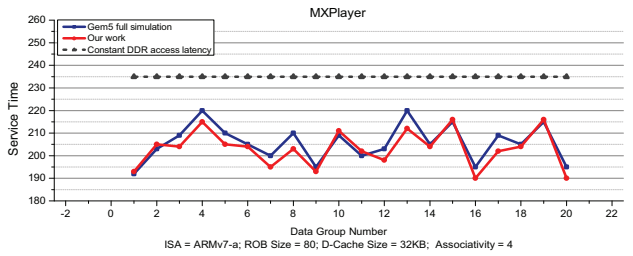

Fig. 10. Service Time from Gem5 Simulations, Our Model and Constant DDR Access Latency

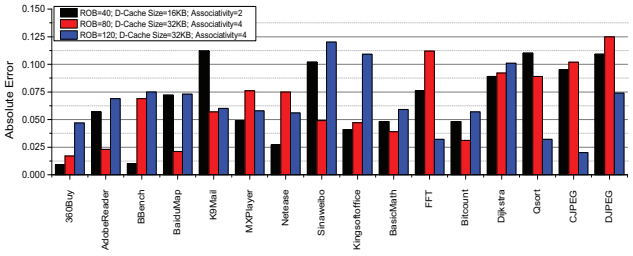

Fig. 11. Absolute Errors of Our Modeled Service Time

E. Framework Evaluation

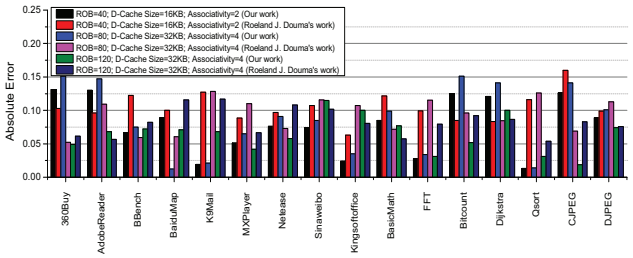

Fig. 12. Absolute Errors of Our Framework and Roeland J. Douma's Work

Combined with the predicted cache misses, CPU stall time caused by cache misses can be computed using Eq. (1). The absolute errors of different benchmarks are shown in Fig. 12 with the average error around 10%. In some benchmarks, such as 360Buy and Adobe Reader, the relatively high errors are, we believe, caused by the considerable errors of the stack distance cache misses model. Moreover, we compare our work with Roeland J. Douma's work in terms of

accuracy. Also shown in Fig. 12, the average error of Roeland J. Douma's work is around 12%, which is a little higher than ours.

F. Running Time Evaluation

We make the comparison of the running time among Gem5 full simulations, our framework, Roeland J. Doumas work [11] and Jian Chen's work [11]. All experiments are implementated on an Intel I7-4790 computer. The time cost of our framework contains three parts, instructions tracing, traces analysis and performance estimations. Because all estimations in our framework are based on formulas, its time cost is much lower than that of instructions tracing or traces analysis. Hence, it is ignored in the running time comparison. As for the time cost of instructions tracing, we evaluated it on the AtomicSimpleCPU [19] mode of Gem5 simulator. Furthermore, each benchmark only needs be traced once and the trace can be utilized repeatedly for different hardware configurations. Thus, the time of instructions tracing, in this paper, is shared by all three hardware configurations we have tested. The comparing results are shown in Fig. 13. Our framework evaluating time can be sped up around 35 times relative to Gem5 full simulations and 8 times compared with Roeland J. Douma's work on average [11]. Jian Chen's work focuses on design space explorations but not performance evaluations, so we merely evaluate the running time of his work on modeling MLP, which is slightly less than ours.

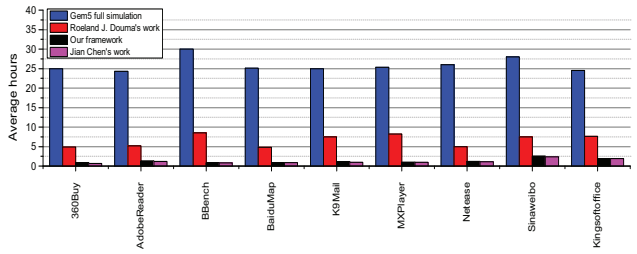

Fig. 13. Running Time Comparisons

VI. Conclusion

This paper proposed a framework to fast evaluate the whole cache performance for out-of-order processors without cycle-accurate simulations. The framework contains three components, cache misses modeling, MLP modeling and effective cache miss service time modeling. According to the results of our experiments, the average error of CPU stall time caused by cache misses is around 10%, while the evaluation time is x35 faster than cycle-accurate simulations. In future, we will refine the model by considering the deviation effects of the stack distance distribution caused by out-of-order mechanisms, which significantly increases the error of cache misses prediction.

References

[1] D. Eklov and E. Hagersten. Statstack: Efficient modeling of LRU caches. In *IEEE International Symposium on PERFORMANCE Analysis of Systems & Software*, pages 55–65, 2010.

[2] Stijn Eyerman, Kenneth Hoste, and Lieven Eeckhout. Mechanistic-empirical processor performance modeling for constructing CPI stacks on real hardware. In *IEEE International Symposium on PERFORMANCE Analysis of Systems and Software*, pages 216–226, 2011.

[3] Norman P Jouppi and Ramsey W Haddad. Destination indexed miss status holding registers, July 28 1998. US Patent 5,787,465.

[4] Alm, George Si, C caval, and David A. Padua. Calculating stack distances efficiently. *ACM Sigplan Notices*, 38(2 supplement):37–43, 2002.

[5] Chi Xu, Xi Chen, R. P. Dick, and Z. M. Mao. Cache contention and application performance prediction for multi-core systems. In *IEEE International Symposium on PERFORMANCE Analysis of Systems & Software*, pages 76–86, 2010.

[6] Xian He Sun and Dawei Wang. Concurrent average memory access time. *Computer*, 47(5):74–80, 2014.

[7] Chou Yuan, Brian Fahs, and Santosh Abraham. Microarchitecture optimizations for exploiting memory-level parallelism. *ACM Sigarch Computer Architecture News*, 32(2):76–87, 2004.

[8] Stijn Eyerman, Lieven Eeckhout, Tejas Karkhanis, and James E. Smith. A mechanistic performance model for superscalar out-of-order processors. *ACM Transactions on Computer Systems*, 27(2):824–833, 2009.

[9] Tejas S. Karkhanis and James E. Smith. Automated design of application specific superscalar processors: an analytical approach. *ACM Sigarch Computer Architecture News*, 35(2):402–411, 2007.

[10] Zhonglei Wang and Jrg Henkel. Fast and accurate cache modeling in source-level simulation of embedded software. pages 587–592, 2013.

[11] Roeland J. Douma, Sebastian Altmeyer, and Andy D. Pimentel. Fast and precise cache performance estimation for out-of-order execution. In *Design, Automation & Test in Europe Conference & Exhibition*, 2015.

[12] Stijn Eyerman, Lieven Eeckhout, and Koen De Bosschere. Efficient design space exploration of high performance embedded out-of-order processors. In *Conference on Design, Automation and Test in Europe, DATE 2006, Munich, Germany, March*, pages 351–356, 2006.

[13] Stijn Everman and Lieven Eeckhout. A memory-level parallelism aware fetch policy for SMT processors. In *IEEE International Symposium on High PERFORMANCE Computer Architecture*, pages 240–249, 2007.

[14] Rik Jongerius, Giovanni Mariani, Andreea Anghel, and Gero Dittmann. Analytic processor model for fast design-space exploration. In *IEEE International Conference on Computer Design*, pages 411–414, 2015.

[15] Jian Chen, Lizy Kurian John, and Dimitris Kaseridis. Modeling program resource demand using inherent program characteristics. In *SIGMETRICS 2011, Proceedings of the 2011 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, San Jose, CA, USA, 07-11 June*, pages 1–12, 2011.

[16] Stijn Eyerman, Lieven Eeckhout, Tejas Karkhanis, and James E. Smith. A mechanistic performance model for superscalar out-of-order processors. *ACM Transactions on Computer Systems*, 27(2):824–833, 2009.

[17] Chi Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *ACM Sigplan 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June*, pages 190–200, 2005.

[18] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *Freenix Track: 2005 Usenix Technical Conference, April 10-15, 2005, Anaheim, CA, USA*, pages 41–46, 2005.

[19] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, and Somayeh Sardashti. The Gem5 simulator. *Computer Architecture News*, 39(2):1–7, 2011.

[20] Calin Caval and David A. Padua. Estimating cache misses and locality using stack distances. In *International Conference on Supercomputing, ICS 2003, San Francisco, CA, USA, June*, pages 150–159, 2003.

[21] John Goodacre and Andrew N. Sloss. Parallelism and the ARM instruction set architecture. *Computer*, 38(7):42–50, 2005.

[22] Kazuhisa Ishizaka and Takashi Miyazaki. Cache memory, including miss status/information and a method using the same, October 8 2013. US Patent 8,555,001.

[23] Yongbing Huang, Zhongbin Zha, Mingyu Chen, and Lixin Zhang. Moby: A mobile benchmark suite for architectural simulators. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 45–54, 2014.

[24] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Workload Characterization, 2001. Wwc-4. 2001 IEEE International Workshop*, pages 3–14, 2001.

[25] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communicatons systems. In *ACM/IEEE International Symposium on Microarchitecture*, pages 330–335, 1997.

[26] Speculative load instructions with retry: US, us20030135722[p]. 2003.