A Methodology for OLTP Micro-architectural Analysis

Utku Sirin
EPFL
utku.sirin@epfl.ch

Ahmad Yasin
Intel Corporation
ahmad.yasin@intel.com

Anastasia Ailamaki
EPFL, RAW Labs SA
anastasia.ailamaki@epfl.ch

ABSTRACT

Micro-architectural analysis is critical to investigate the interaction between workloads and processors. While today's aggressive out-of-order processors provide a rich set of performance events for deep execution cycle analysis, OLTP characterization studies usually use a cache-miss-based method (CMBM). In this work, we investigate the validity and the functionality of CMBM by comparing it with Intel's state-of-the-art Top-down Micro-architecture Analysis Method (TMAM) for OLTP workloads. We show that, while CMBM and TMAM provide a similar high-level micro-architectural behavior, it is inadequate for a fine-grained micro-architectural analysis. We further show that TMAM underestimates memory stalls. We optimize TMAM's execution cycle breakdown, and improve its estimation of memory stalls up to 50%.

1 INTRODUCTION

Micro-architectural analysis plays a crucial role in understanding the interaction between workloads and processors. Modern processors provide a performance monitoring unit (PMU) exposing hundreds of performance events to examine this interaction. PMU performance events can be used to understand how efficiently a workload uses the processor resources, and how compatible the processor features are to the workload [3]. On the other hand, today's modern processors are aggressively optimized for speed featuring wide-issue, out-of-order execution engines, deep cache hierarchies, deep pipeline stages and large pipeline buffers. While these features allow processors to exploit instruction- and memory-level parallelism, the increasing complexity of the micro-architecture makes the micro-architectural analysis harder.

Intel has recently announced its Top-down Micro-architecture Analysis Method (TMAM) proposing a hierarchical execution cycles breakdown based on a set of new performance events [19]. TMAM examines every instruction issue slot independently, and therefore, it is able to provide an accurate slot-level breakdown. Although TMAM is an ambitious method for analyzing the execution cycles, it has not been adopted by the studies characterizing the micro-architectural behavior of online transaction processing (OLTP) workloads. The OLTP characterization studies usually use a cache-miss-based method (CMBM) to analyze the execution cycles [13]. While it is conventional knowledge that OLTP workloads spend the majority of their time on the stalls due to cache misses, it is not clear whether this assumption still holds, and if not what the other reasons for the stalls are when running the OLTP workloads.

In this paper, we compare CMBM with TMAM for OLTP workloads. We then optimize TMAM's execution cycle breakdown, and improve its estimation of memory stalls. To compare the two methods, we use one traditional disk-based system, Shore-MT [12], and two in-memory systems, VoltDB [18] and HyPer [4], running the standard TPC-C benchmark. Our first goal is to estimate the validity and the functionality of CMBM on today's aggressive out-of-order processors compared to TMAM. Our second goal is to examine TMAM in detail, and improve its breakdown using the rich set of performance events available on today's modern processors. Our study demonstrates the following:

- CMBM and TMAM provide similar high-level micro-architectural behavior. This is because the majority of the stalls are due to accessing the instruction and/or data memory for which cache misses partially account.
- CMBM is only able to present memory stalls at the front-end, and cache-miss-related stalls at the back-end. This renders CMBM inadequate for representing detailed micro-architectural behavior as that includes other stalls. On the other hand, TMAM significantly underestimates memory stalls.
- We modify the way TMAM calculates stalls, and optimize its execution cycles breakdown by improving its estimation of memory stalls up to 50%.

The rest of the paper is organized as follows. Section 2 summarizes the micro-architectural analysis methods used by existing OLTP characterization studies. Section 3 presents an example micro-architecture of today's out-of-order processors. Section 4 describes the micro-architectural analysis methods we examine. While Section 5 presents the experimental setup and methodology, Section 6 evaluates the analysis methods we discuss. Finally, Section 7 concludes.

2 BACKGROUND

Micro-architectural analysis of OLTP workloads has received a significant attention over the last two decades. Barroso et al. and Stets et al. investigate the memory system behavior of OLTP- and DSS-like workloads based on performance events counting the number of cache misses, TLB misses and branch mispredictions similar to CMBM [2, 14]. Keeton et al. also use performance events to profile OLTP workloads while separating the stalls due to the cache misses and the stalls due to resource/data dependencies [8]. Although taking resource/data dependencies into account provides a comprehensive understanding of the micro-architectural behavior, the study does not provide a complete cycle/slot-level breakdown. On the other hand, Ranganathan et al. examine the micro-architectural

DaMoN'17, Chicago, IL, USA

DOI: http://dx.doi.org/10.1145/3076113.3076116

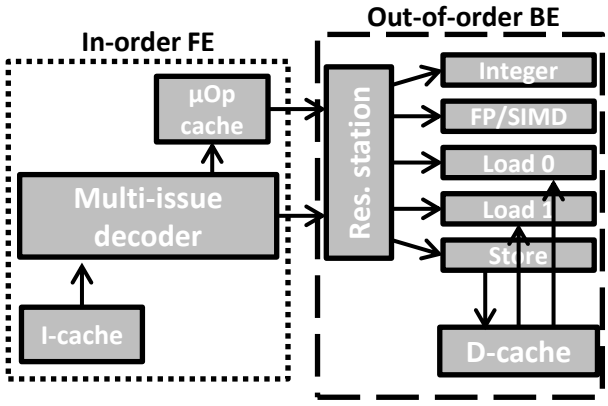

Figure 1: An out-of-order processor model based on Intel Sandy/Ivy Bridge micro-architecture [7]. FE: front-end, BE: back-end.

behavior of OLTP and DSS workloads in a cycle-accurate simulation environment, which allows monitoring the entire micro-architecture of the processor [11]. Our work focuses on analysis methods when profiling real hardware.

Ailamaki et al. provide a formula for full-cycle breakdown accounting for all the stall cycle components [1]. While the presented formula provides a consistent way to analyze the micro-architectural behavior, today's processors provide a rich set of performance events enabling to enhance the presented method. Lastly, Ferdman et al., Tözün et al., and Sirin et al. use the instruction-retired-per-cycle (IPC) metric to understand the overall micro-architectural behavior of OLTP workloads on modern out-of-order processors [3, 13, 16]. They elaborate this overall behavior by examining the cache miss behavior. In our work, we investigate the validity and the functionality of cache miss behavior by comparing CMBM with TMAM.

3 AN EXAMPLE MICRO-ARCHITECTURE

In this section, we outline the major micro-architectural structures of today's out-of-order processors. Figure 1 shows the simplified block diagram of the Intel Sandy/Ivy Bridge micro-architecture [7]. It contains two major building blocks, the front-end (FE) and the back-end (BE). FE contains the micro-architectural structures to fetch, decode and issue instructions, which are an I-cache, a multi-issue instruction-to-micro-operation(μOp) decoder and a small (1.5KB [7]) μOp cache. The I-cache is responsible for keeping the instructions that the processor will execute next. While an instruction cache miss stalls the entire pipeline, many of these stalls are hidden by buffering instructions in the pipeline. The decoder decodes the fetched instructions into μOps. As Intel uses a complex instruction set architecture, i.e., CISC, it needs to convert complex instructions into simpler μOps. The decoder is multi-issue since it is able to process and deliver multiple instructions in one cycle. Delivering multiple instructions in one cycle enables exploiting instruction-level parallelism, and using the resources of the BE more efficiently. Lastly, the μOp cache is responsible for keeping the most recently decoded μOps to avoid re-decoding them.

BE contains the micro-architectural structures to execute the issued μOps, which are a reservation station (RS), several execution units and a D-cache. When a μOp arrives to BE, BE registers the μOp to the RS. RS is then responsible for tracing the operands and dependencies of the μOp, and delivering the μOp to the relevant execution unit. The execution units are responsible for actually realizing the operations such as loading a value from memory to register. All execution units operate in parallel unless there is a dependency. Parallel execution enables exploiting instruction-level parallelism, and therefore, executing multiple instructions in one cycle. An out-of-order processor can theoretically execute as many instructions as issued in one cycle. Furthermore, every execution unit has its own private buffer. For example, the Sandy/Ivy Bridge micro-architecture contains a 64-entry load and a 36-entry store buffer, which enables buffering up to 64/36 load/store operations. Buffering allows to overlap the same kinds of μOps when some are not ready to complete. For example, if a load operation misses from D-cache, another load operation can start its execution without waiting for the former load operation to finish. Lastly, D-cache is the L1 data cache. It is incorporated to BE and works cooperatively with the load and store execution units to deliver data from/to memory to/from processor. The D-caches of today's processors contain a set of registers for tracking the outstanding cache misses, i.e., miss status handling registers (MSHRs). MSHRs allow processors to continue their execution when there is a cache miss, and further improve parallelism.

4 ANALYSIS METHODS

In this section, we explain the three methods we compare: CMBM, TMAM, and the optimized TMAM that we have derived.

4.1 Retiring

In this section, we describe the cycles used to retire μOps. TMAM identifies these cycles as the Retiring category. Since CMBM only accounts for the stall cycles, we explain how to estimate the Retiring category for the sake of completeness, and then ignore the Retiring category in the rest of the paper.

TMAM estimates the Retiring based on the performance event counting the number of retired μOps. TMAM counts the number retired μOps, and then normalizes it with respect to the number of issue slots times the number of clock cycles. This is because the out-of-order processors can retire more than one μOps in one cycle. Therefore, the number of μOps is a slot-level metric, rather than a cycle-level metric. Overall, TMAM estimates the Retiring as follows:

$$Retiring = \frac{NumberOfRetired\mu Ops}{IssueWidth \times NumberOfClockCycles}$$

The Retiring category presents the overall rate of the occupied issue slots during the execution of the program.

4.2 CMBM

This section describes CMBM. Table 1 presents the stall cycles breakdown. CMBM decomposes the stall cycles into three main components: the front-end (FE) stalls, the back-end (BE) stalls and the stalls due to the branch mispredictions. It approximates the FE

Component		Description
FE		Stalls from front-end
	L1I	Stalls due to L1I misses
	L2I	Stalls due to L2I misses
	L3I	Stalls due to L3I misses
BE		Stalls from back-end
	L1D	Stalls due to L1D misses
	L2D	Stalls due to L2D misses
	L3D	Stalls due to L3D misses
Branch		Stalls due to branch mispredictions

Table 1: Stall cycles breakdown for CMBM for a three-level of cache hierarchy.

stalls as the sum of the stalls due to L1, L2 and L3 instruction cache misses, and approximates the BE stalls as the sum of the stalls due to L1, L2 and L3 data cache misses[1]. CMBM uses the traditional performance events, which counts the number of instruction/data misses for every level of caches[2]. Then, it uses a pre-estimated cache miss penalty for every level of caches, and multiplies the number of misses with the estimated penalty to obtain the number of stall cycles for a particular level of cache. In other words, the L1I stalls identify the stall cycles spent between the L1 and L2 caches due to L1I misses, L2I stalls identify the stall cycles spent between the L2 and L3 caches due to L2I misses, and so on. The cache miss penalties are usually available at the optimization reference manual of the profiled processor. For example, Intel's cache miss penalties can be found in page 2-24 and B-42 of [7]. CMBM assumes L1I and L1D hits do not cause any stalls.

Lastly, CMBM estimates the branch misprediction stalls based on the traditional performance event counting the number of retired branch mispredictions. It then uses a pre-estimated penalty of a single branch misprediction to obtain the number of stall cycles. Branch misprediction penalty is usually available at the optimization reference manual of the profiled processor. For example, Intel's branch mispredicton penalty can be found in page B-47 of [7]. Please refer to Table 4 and 5 in Appendix A for CMBM formulas.

CMBM has been widely used by OLTP characterization studies that assumes OLTP workloads spend the majority of their time on cache misses. While this is a reasonable assumption, today's wide-issue, out-of-order processors use sophisticated techniques to extract instruction- and memory-level parallelism, and have large pipeline buffers to overlap the stall cycles. Moreover, the true cache miss penalties may vary based on the way that instruction stream interacts with the micro-architecture. For example, if the total number of outstanding misses is larger than the total number of outstanding miss status handling registers, a cache miss can cost more than the estimated penalty. Therefore, simply counting the number of misses per cache level, and multiplying it by a constant pre-estimated penalty can potentially over/underestimate the number of stall cycles. Moreover, today's in-memory OLTP systems are heavily-optimized for high transactional throughput [15]. Therefore, stall components other than cache misses, such as resource/data dependencies, might rise up for a different system, configuration and/or optimization. CMBM is unable to account for them.

[1]Based on a three level of cache hierarchy.
[2]Note that only L1 cache is split for instruction and data, whereas L2 and L3 caches are unified. However, Intel provides performance events counting the instruction and data misses separately for all the three levels of caches.

Component			Description
FE			Stalls from front-end
	FE Latency		Stalls due to fetch starvation
		iTLB	Stalls due to iTLB misses
		iCache Miss	Stalls due to L1, L2 or L3I misses
	FE BW		Stalls due to FE units like decoder/μOp cache
	FE Misc.		Stalls due to misc. FE issues
BE			Stalls from back-end
	Memory Bound		Stalls due to memory accesses
		DTLB	Stalls due to DTLB misses
		L1 Bound	Stalls due to L1D hits
		L2 Bound	Stalls due to L1D misses w/ L2 hit
		L3 Bound	Stalls due to L2D misses w/ L3 hit
		DRAM Bound	Stalls due to L3D misses
		Store Bound	Stalls due to store operations
	Core Bound		Stalls due to instruction dependencies and execution unit pressure
	BE Misc.		Stalls due to misc. BE issues
Bad Speculation			Stalls due to misspeculations

Table 2: Stall cycles breakdown for TMAM for a three-level of cache hierarchy.

4.3 TMAM

In this section, we describe TMAM [19]. Table 2 presents the stall cycles breakdown. TMAM decomposes the stall cycles into three main components at the top-level: the Front-End (FE) stalls, the Back-End (BE) stalls and the Bad Speculation stalls. TMAM uses Intel's new performance event that directly counts the FE, and derives the BE and the Bad Speculation stalls. At every cycle, TMAM categorizes every μOP issue slot as either a Retiring, or a FE stall, or a BE stall or a Bad Speculation slot, which provides an accurate slot-level breakdown of the execution cycles. For example, if the processor is 4-issue, then TMAM provides an accurate breakdown of $4 \times NumberOfClockCycles$ cycles.

TMAM further breakdowns the FE and the BE stalls into lower-level components. To do that, TMAM follows a hierarchical approach. It identifies the main micro-architectural bottlenecks at the top-level, and the user would analyze the lower-level nodes only if the parent nodes are significant. Moreover, TMAM only compares the weights of the sibling nodes among each other. Hence, the

hierarchical approach provides hierarchical safety guarantee preventing the lower-level nodes from over/underestimate the stalls.

TMAM breakdowns the FE stalls into two main components: FE Latency and FE Bandwidth (BW). FE Latency stalls include the instruction TLB stalls (iTLB) and the instruction cache miss stalls (iCache Miss), whereas FE BW stalls include the bubbles due to the inefficiencies in the FE units such as the decoder and the μOp cache. The iTLB stalls are due to iTLB misses, and the iCache Miss stalls are due to L1, L2 or L3 instruction cache misses. TMAM estimates the iTLB and the iCache Miss stalls by directly counting the number of stall cycles rather than counting the number of iTLB/iCache misses. Directly counting the stalls mitigates the problem of overcounting the iCache Miss stalls that are hidden by the buffered instructions. FE Latency category includes certain other stalls such as the stalls due to fetching from the correct target right after a branch misprediction, i.e., Branch Resteers, and the stalls due to switching between the decoder and the μOp cache. We combine these stall components under a new category, the FE Miscellaneous, as they usually are small for OLTP workloads.

TMAM decomposes the BE stalls into two main components: Memory Bound and Core Bound stalls. The Memory Bound stalls represent the stalls while accessing the memory hierarchy, whereas the Core Bound stalls represent the stalls due to data/resource dependencies. Memory Bound stalls include certain other stalls such as the stalls due to the load operations blocked by store forwarding, and the stalls due to false sharing. We combine these stall components under a new category, the BE Miscellaneous, as they usually are small for OLTP workloads. TMAM measures the Memory Bound stalls by using Intel's new performance event that directly counts the stalls while waiting data from the memory hierarchy. Directly counting the stalls enables accounting for the overlapped stalls as the execution units at BE work in parallel, and therefore can overlap stalls. To estimate the Core Bound stalls, TMAM identifies the number of sub-optimal execution cycles where BE executes less than the maximum number of μOps (e.g., less than 4 μOps for a 4-issue processor). TMAM then subtracts the number of Memory Bound stalls from the number of sub-optimal execution cycles to obtain core stalls. The idea is that the cycles that cannot execute the optimal number of μOps when there is no memory stalls should be suffering from resource/data dependencies.

TMAM further decomposes the Memory Bound stalls into the data TLB stalls (DTLB), L1, L2, L3 and the Store Bound stalls. The DTLB stalls represent the stalls due to DTLB misses. The L1 Bound stalls represent the stalls hitting the L1D cache. While L1D hits are largely overlapped, they can be problematic in certain cases such as having a large number of data accesses that requires higher bandwidth than the available L1D bandwidth, or store to load forwarding block where the load does not eventually miss the L1D[3]. The L2, L3 and DRAM Bound stalls represent the stalls due to L1D, L2D and L3D cache misses with L2D, L3D and DRAM hits. TMAM uses Intel's new performance events to estimate the DTLB, L1, L2, L3 and DRAM Bound stalls. These new events directly count the number of stall cycles rather counting the number of misses. Directly counting the stalls enables TMAM accounting for the overlapped stalls at the BE. Lastly, the Store Bound stalls contain the stalls due to the store operations. In today's processors, store operations are buffered as they mostly are not on the critical path of the program. The Store Bound stalls identify the stalls when the store buffer is full. TMAM uses a dedicated performance event to measure this.

Finally, TMAM accurately calculates the Bad Speculation stalls due to branch mispredictions as well as data speculations such as memory ordering violations. The speculation cost includes the wasted issue slots for bogus operations and the recovery cycles for the out-of-order engine to recover from the misspeculation. Please refer to Table 4, 6 and 7 in Appendix A for TMAM formulas.

[3]L1D hits could also be estimated based on the traditional performance event counting the number of L1D hits. We could multiply the number of L1D hits with a pre-estimated L1D hit latency, and obtain the complete L1D hit access-time. However, TMAM's L1 Bound stalls represent the L1D hit stall-time rather than the complete L1D hit access-time. As L1D hits are largely overlapped, the L1D hit access-time can be significantly larger than the L1D hit stall-time. Our numbers show that L1D hit access-time is 2.5-8x larger than L1D hit stall-time.

4.4 Optimized TMAM

In this section, we further investigate TMAM, and optimize TMAM's estimation of the Memory and Core Bound stalls. TMAM estimates the Core Bound stalls by subtracting the Memory Bound stalls from the sub-optimal execution cycles. A sub-optimal execution cycle is a cycle where back-end (BE) executes sub-optimal number of μOps. A critical step of estimating the sub-optimal execution cycles is to exclude the cycles where front-end (FE) already delivers a sub-optimal number of μOps. As processor pipelines the FE to the BE, when FE delivers a sub-optimal number of μOps, BE naturally executes a sub-optimal number of μOps disregarding any stalls at BE.

We observe that the original TMAM does not explicitly take care of the cycles where FE delivers a sub-optimal number of μOps, and therefore over-counts the sub-optimal execution cycles. Since TMAM obtains the core stalls by subtracting the memory stalls from the sub-optimal execution cycles, this over-counting causes overestimating the core stalls. Furthermore, TMAM only compares the number of stalls that are at the same level, i.e., the siblings. Since the Core and Memory Bound stalls are the siblings, overestimating the Core Bound stalls results in underestimating the Memory Bound stalls. We mitigate this problem based on Intel's new performance events counting the number of cycles where FE delivers a sub-optimal number of μOps. By using these events, we obtain the true number of sub-optimal execution cycles, and improve TMAM's estimation of the Core and Memory Bound stalls. Please refer to Table 8 in Appendix A for the formula of the optimized Core Bound stalls. This issue is also realized in the recent version 3 of TMAM [6].

5 SETUP AND METHODOLOGY

In this section, we present the experimental setup and methodology.

Hardware: We run our experiments on an Intel Xeon processor Ivy Bridge micro-architecture. Table 3 shows the details of the micro-architecture. To collect hardware counter numbers about performance events, we use Intel VTune Amplifier XE 2015 [5], which provides an API for lightweight hardware counter sampling. We disable hyper-threading to obtain more precise hardware sampling values and increase predictability in measurements.

Processor	Intel(R) Xeon(R) CPU E5-2640 v2 (Ivy Bridge)
#Sockets	2
#Cores per Socket	8
Hyper-Threading	Off
Clock Speed	2.00GHz
Memory	256GB
L1I / L1D (per core)	32KB / 32KB
L2 (per core)	256KB
L3 (shared)	20MB

Table 3: Server parameters.

OS: We run all the experiments using RHEL 6.5 with Linux kernel version 2.6.32.

Benchmarks: We run the standard TPC-C benchmark [17] with a database of size 20GB for all the experiments.

Analyzed systems: We analyze one traditional disk-based system, Shore-MT [12], and two in-memory OLTP systems, VoltDB [18] (Community Edition Version 4.8), and HyPer [4] (online demo-version). For all the systems, we use asynchronous logging. Therefore, there is no delay due to I/O in the critical path of the transaction execution. We choose these three systems as they are well-known in the community and their design characteristics represent a good variety of today's OLTP systems. Hence, we aim to increase the reliability our work, and show that our conclusions apply for OLTP systems using different architectures. Shore-MT is a disk-based system possessing large amount of overheads such as buffer pool and disk-oriented index structures. VoltDB and HyPer are in-memory systems eliminating most of the overheads the disk-based systems possess. Both VoltDB and HyPer rely on physical data partitioning. While VoltDB uses a traditional B-tree with node size tuned to the last-level cache line size [15], HyPer implements adaptive radix tree with adaptive compact node sizes [9]. Furthermore, while HyPer compiles transactions into the machine code [10], VoltDB uses stored procedures without compiling the stored procedures into the machine code.

Measurements: We populate the databases from scratch before each experiment and the data remains memory-resident throughout the experiments. We use memory-mapped I/O for log flushing. Both the worker threads executing the transactions and the client threads generating the transactions run on the same machine. We first start the server process, populate the database, and then start the experiment by launching the clients that generate and submit transactional requests to the database server. For a client submitting transactional requests there is one OLTP worker thread satisfying the requests. Less than 1% of the execution time is spent for the client threads, whereas the remaining 99% of the time is spent for the server threads. We profile the database server process by attaching VTune to it during a 120-second benchmark run following a 60-second warm-up period. We repeat every experiment three times and report the average result. We observe a standard deviation of less than 5% for the three repeats.

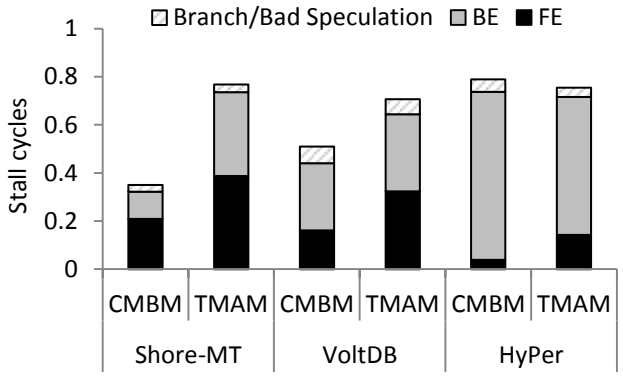

Figure 2: Stall cycles breakdown at top level for CMBM and TMAM. FE: front-end, BE: back-end, Branch: branch misprediction.

We run only single-threaded experiments as it has been shown that micro-architectural behavior of OLTP workloads does not change significantly across single- and multi-threaded runs [13].

6 EXPERIMENTAL RESULTS

This section presents the experimental evaluation. We firstly compare the high-level micro-architectural behavior CMBM and TMAM provide. We then compare CMBM and TMAM at their front-end (FE) and back-end (BE) breakdowns for a deeper understanding of the high-level micro-architectural behavior.

6.1 High-level analysis

In this section, we compare CMBM and TMAM at their top-level breakdowns. Our goal is to investigate how well CMBM represents the high-level micro-architectural behavior of OLTP workloads compared to TMAM.

Figure 2 compares the number of stall cycles of CMBM and TMAM. As TMAM provides a slot-level breakdown, the number of stall cycles of TMAM is normalized with respect to the number of issue slots times the number of clock cycles, i.e., $IssueWidth \times NumberOfClockCycles$. CMBM, however, provides a cycle-level breakdown. Therefore, we normalize the number of stall cycles of CMBM with respect to the number of clock cycles. We observe that, for Shore-MT and VoltDB, CMBM severely underestimates the number of stall cycles. This shows that CMBM is not sufficient to represent all the stalls for OLTP workloads. On the other hand, we observe that the ratio between the FE and the BE stalls are similar for CMBM and TMAM. CMBM and TMAM show that Shore-MT mostly suffers from the FE stalls. CMBM shows that the BE stalls are larger than the FE stalls for VoltDB, whereas TMAM shows that the FE stalls are slightly larger than the BE stalls. However, both CMBM and TMAM show that the difference between the FE and BE stalls are small. Lastly, both CMBM and TMAM show that HyPer mostly suffer from the BE stalls. Therefore, CMBM and TMAM provide similar high-level micro-architectural behavior for all the three systems.

VoltDB is an in-memory system that eliminates most of the overheads that the traditional disk-based systems possess. Despite being an in-memory system, VoltDB significantly suffers from the

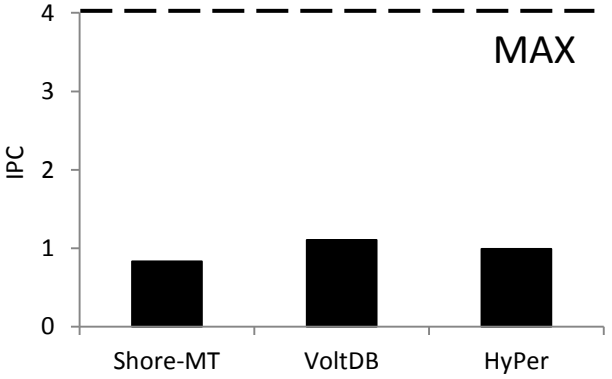

Figure 3: Instruction retired per cycle, i.e., IPC, values. This metric is not used by neither CMBM nor TMAM. We include it for the sake of completeness.

FE stalls similar to the disk-based Shore-MT. The large FE stalls of VoltDB shows that the instruction footprint of OLTP workloads is large and complex even for the optimized in-memory systems.

On the other hand, the other in-memory system we use, HyPer, suffers from FE stalls much less than Shore-MT and VoltDB. This is because HyPer compiles transactions into aggressively optimized machine code, further reducing sources of the overheads such as function calls and branches. The aggressively optimized machine code enables HyPer to eliminate FE stalls, and therefore to finish a larger number of transactions using the same number of instructions. Having eliminated the FE stalls, HyPer exposes the next bottleneck, BE stalls. HyPer suffers from the BE stalls more than the other systems. Therefore, we conclude that, in addition to the instruction footprint, the data footprint of OLTP workloads is also complex. This finding extends the discussion in [13], where we argued that reduced instruction stalls amplify the data stalls.

Lastly, we present the number of instructions retired per cycle (IPC) values for the sake of completeness of our work. Figure 3 shows the results. We observe that all the systems exhibit low IPC. Although the processor we use can retire up to four instructions per cycle, the IPC values barely reach one, showing that the OLTP systems severely underutilize the micro-architectural resources. These results corroborate with the previous research on the analysis of enterprise workloads [20].

The micro-architectural behavior of the OLTP systems are inadequate to compare the performance of the OLTP systems. This is because the ratio of stall cycles is unable to highlight how many stall cycles the systems incur to execute a single transaction. A system with a high ratio of stall cycles may have much higher performance than a system with a low ratio of stall cycles, as it is the case for Shore-MT and HyPer. Shore-MT and HyPer have similar ratios of stall cycles (~80%) based on the TMAM results, although HyPer delivers much higher throughput than Shore-MT. This is because the number of stall cycles per transaction is much higher for Shore-MT than it is for HyPer. However, when we normalize the stall cycles to the total number of cycles, we observe a similar ratio of the stall cycles for Shore-MT and HyPer. The similar ratio of the stall cycles only shows that even though an OLTP system is aggressively optimized and delivers orders of magnitude higher throughput, the

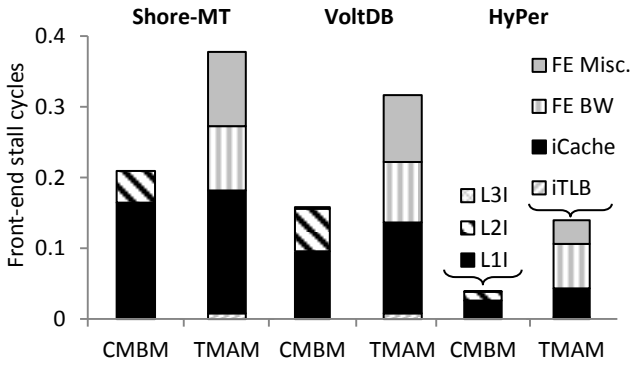

Figure 4: Stall cycles breakdown at FE for CMBM and TMAM. FE: front-end, BE: back-end, BW: bandwidth, misc: miscellaneous.

OLTP workload is still complex causing large number of stall cycles. On the other hand, the micro-architectural behavior is useful to estimate how efficiently the workload uses the micro-architectural resources, and highlight the opportunities to improve the system performance. We refer the reader to [13] for a detailed analysis of the micro-architectural behavior of the OLTP systems including the analysis of the number of stall cycles per transaction.

6.2 Front-end stalls

This section compares the FE breakdowns of CMBM and TMAM. Our goal is to identify the reasons for the similarities and differences between the two methods. TMAM only compares the nodes that are at the same level. To be able to compare the stalls that are at the different levels in the hierarchy of TMAM, we factorize the values of the lower-level nodes of the FE stalls with the values of the higher-level nodes that are at the path of the lower-level nodes. Figure 4 shows the results. Figure 4 contains two bars for each OLTP system we profile: a bar for the breakdown of CMBM and a bar for the breakdown of TMAM. As the two breakdowns have different stall components, the two bars have different legends. We present the legend only for a single system (HyPer) as it is the same for the other systems. Moreover, we use a coloring scheme where the similar/same components of the two breakdowns have the same colors/patterns (e.g., L1I and iCache).

We observe that TMAM's iCache stalls are close to the sum of the L1, L2 and L3 instruction stalls that CMBM estimates. This is consistent with the definition of iCache stalls (see Section 4.3). On the other hand, we observe that CMBM misses the half of the FE stalls as it is unable to account for the FE Bandwidth (FE BW) and the FE Miscellaneous (FE Misc.) stalls. FE BW and FE Misc. are the stalls due to certain inefficiencies at the FE engine such as legacy decoding of long-latency instructions and switching between the decoder and the μOp cache. The FE BW and the FE Misc. stalls account for ~50% of the FE stalls for Shore-MT and VoltDB. While for HyPer, the FE BW and the FE Misc. stalls are ~70% of the FE stalls, the overall FE stalls are not significant for HyPer.

Therefore, we conclude that CMBM closely approximates the stalls due to the instruction cache misses at FE, which enables CMBM to provide a similar high-level micro-architectural behavior to TMAM. On the other hand, CMBM is unable to represent the

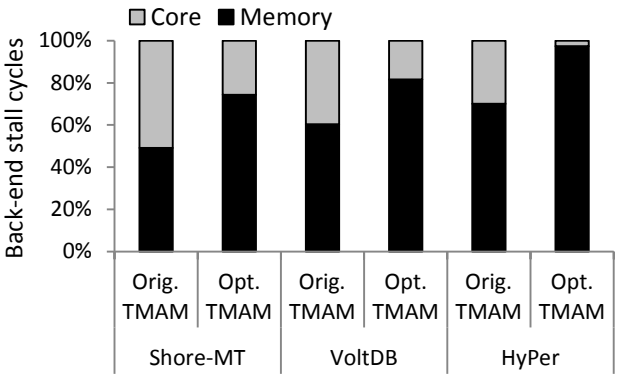

Figure 5: Intel's original TMAM (Orig. TMAM) vs. our optimized TMAM (Opt. TMAM) at back-end stalls breakdown.

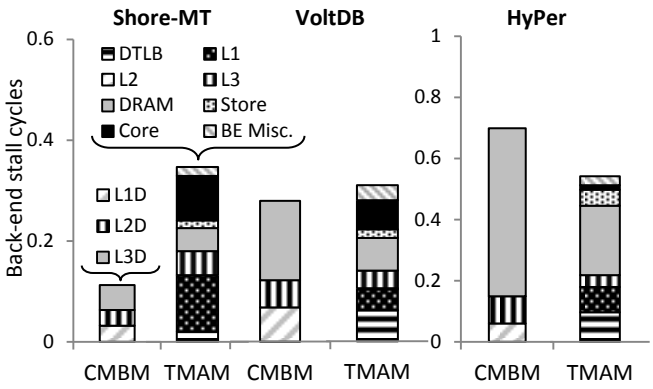

Figure 6: Stall cycles breakdown at BE for CMBM and the optimized TMAM. FE: front-end, BE: back-end, misc: miscellaneous.

half of the FE stalls due to ignoring the FE BW and the FE Misc. stalls, which renders it inadequate to provide a detailed micro-architectural behavior. Such FE stalls do matter for superscalar machines, and are properly handled by TMAM [19].

6.3 Back-end stalls

This section examines the BE stalls to further understand the differences and similarities between CMBM and TMAM. We firstly justify the optimization of TMAM, and then compare CMBM with the optimized TMAM at the BE breakdowns.

Original vs. Optimized TMAM: Figure 5 compares the Memory and Core Bound stalls of the original and the optimized TMAM. The optimized TMAM significantly improves the estimation of the Memory Bound stalls. For Shore-MT, the original TMAM estimates the Memory Bound stalls at 50% of the BE stalls, whereas the optimized TMAM estimates the Memory Bound stalls at 75% of the BE stalls. This means that, for Shore-MT, the optimized TMAM improves the estimation of the Memory Bound stalls by 50%. For VoltDB and HyPer, we observe that the Memory Bound stalls increase from 60 to 80% and 70 to 95% of the BE stalls, showing 30 and 40% improvements on the estimation of the Memory Bound stalls. Therefore, the original TMAM indeed underestimates the Memory Bound stalls, and the proposed optimization successfully mitigates this problem.

CMBM vs. Optimized TMAM: Figure 6 compares the BE stalls of CMBM and optimized TMAM. TMAM only compares the nodes that are at the same level. To be able to compare the stalls that are at the different levels in the hierarchy of TMAM, we factorize the values of the lower-level nodes of the BE stalls with the values of the higher-level nodes that are at the path of the lower-level nodes. Figure 6 follows the same format Figure 4 follows, including two bars per system, presenting the legend only once (Shore-MT), and using a coloring scheme where the similar/same components of the two breakdowns have the same colors/patterns. While L1D of CMBM corresponds to the L2 Bound of TMAM, L2D of CMBM corresponds to L3 Bound of TMAM, and L3D of CMBM corresponds to DRAM of TMAM. We firstly observe that CMBM's sum of the L1, L2 and L3D stalls is significantly higher than TMAM's sum of the L2, L3 and DRAM stalls. The sum of the L1, L2 and L3D stalls of CMBM is 1.2x higher when running Shore-MT, and 2.6x higher when running VoltDB and HyPer. This is because the out-of-order BE engine is able to overlap the stalls due to data cache misses for which CMBM is unable to account. TMAM, on the other hand, uses the new performance events directly counting the memory stalls, and therefore is able to account for the overlapped stalls.

Secondly, we observe that CMBM is lack of representing various important stall components. We observe that VoltDB and HyPer suffer from the DTLB misses at 6% and 9% of the execution cycles. Shore-MT suffers from the L1 Bound stalls at 11%, and HyPer suffers from the L1 Bound stalls at 8%. While the Store Bound stalls are not significant for Shore-MT and VoltDB, for HyPer, the Store Bound stalls cost 5% of the execution cycles. Lastly, the Core stalls cost 8% for Shore-MT.

On the other hand, we observe that the majority of the BE stalls are due to accessing the memory hierarchy for all the OLTP systems. Since the number of memory accesses is correlated to the number of data cache misses, having the majority of the stalls to the memory hierarchy enables CMBM to provide a similar high-level micro-architectural behavior to TMAM. The memory stalls are ~70% of the BE stalls for Shore-MT and VoltDB, and ~90% of the BE stalls for HyPer.

Therefore, we conclude that CMBM is inadequate to represent BE stalls thoroughly. This is firstly because it is not able to account for the overlapped data stalls. Secondly, BE stalls are highly fragmented among numerous non-cache-miss stalls, such as L1D hits and store stalls, for which CMBM is unable to represent. On the other hand, as the majority of the BE stalls are due to accessing the memory hierarchy, CMBM and TMAM provide a similar high-level micro-architectural behavior.

7 CONCLUSIONS

In this work, we compare CMBM with TMAM. We show that, CMBM and TMAM provide similar high-level micro-architectural behavior. This is because the majority of the stalls are due to instruction and/or data memory accesses for which CMBM accounts. On the other hand, CMBM is inadequate for representing a detailed micro-architectural behavior as it is unable to address non-memory

stalls in the front-end and non-cache-miss stalls in the back-end. We furthermore show that TMAM underestimates memory stalls. We optimize TMAM's formulation of execution cycles, and improve TMAM's estimation of memory stalls up to 50%.

ACKNOWLEDGEMENTS

We would like to thank the DaMoN reviewers and the members of the DIAS laboratory for their constructive feedback and support throughout this work. We would like to thank Edouard Bugnion, David Levinthal and Miguel Castro for their useful comments. This project has received funding from the Swiss National Science Foundation, Project No.: 200021_146407/1 (Workload- and hardware-aware transaction processing).

REFERENCES

[1] Anastasia Ailamaki, David J. DeWitt, Mark D. Hill, and David A. Wood. 1999. DBMSs on a Modern Processor: Where Does Time Go?. In *VLDB*. 266–277.
[2] Luiz André Barroso, Kourosh Gharachorloo, and Edouard Bugnion. 1998. Memory System Characterization of Commercial Workloads. In *ISCA*. 3–14.
[3] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. 2012. Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware. In *ASPLOS*. 37–48.
[4] HyPer. 2015. HyPer. (2015). http://hyper-db.de/.
[5] Intel. 2015. Intel VTune Amplifier XE Performance Profiler. (2015). http://software.intel.com/en-us/articles/intel-vtune-amplifier-xe/.
[6] Intel. 2016. Intel TMAM Metrics Version 3. (Sept. 2016). https://download.01.org/perfmon/TMAM_Metrics.xlsx.
[7] Intel. 2016. Intel(R) 64 and IA-32 Architectures Optimization Reference Manual. (June 2016).
[8] Kimberly Keeton, David A. Patterson, Yong Qian He, Raphael C. Raphael, and Walter E. Baker. 1998. Performance Characterization of a Quad Pentium Pro SMP Using OLTP Workloads. In *ISCA*. 15–26.
[9] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The Adaptive Radix Tree: ARTful Indexing for Main-memory Databases. In *ICDE*. 38–49.
[10] Thomas Neumann and Viktor Leis. 2014. Compiling Database Queries into Machine Code. *IEEE DEBull* 37, 1 (2014), 3–11.
[11] Parthasarathy Ranganathan, Kourosh Gharachorloo, Sarita V. Adve, and Luiz André Barroso. 1998. Performance of Database Workloads on Shared-memory Systems with Out-of-Order Processors. In *ASPLOS-VIII*. 307–318.
[12] Shore-MT. 2017. Shore-MT Official Website. (2017). http://diaswww.epfl.ch/shore-mt/.
[13] Utku Sirin, Pinar Tozun, Danica Porobic, and Anastasia Ailamaki. 2016. Micro-architectural Analysis of In-memory OLTP. In *SIGMOD*. 387–402.
[14] R. Stets, K. Gharachorloo, and L.A. Barroso. 2002. A Detailed Comparison of Two Transaction Processing Workloads. In *WWC*. 37–48.
[15] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. 2007. The End of An Architectural Era: (It's Time for A Complete Rewrite). In *VLDB*. 1150–1160.
[16] Pınar Tözün, Ippokratis Pandis, Cansu Kaynak, Djordje Jevdjic, and Anastasia Ailamaki. 2013. From A to E: Analyzing TPC's OLTP Benchmarks – The Obsolete, The Ubiquitous, The Unexplored. In *EDBT*. 17–28.
[17] TPC. 2017. TPC Transcation Processing Performance Council. (2017). http://www.tpc.org/.
[18] VoltDB. 2015. VoltDB. (2015). http://www.voltdb.com/.
[19] Ahmad Yasin. 2014. A Top-Down Method for Performance Analysis and Counters Architecture. In *ISPASS*. 35–44.
[20] Ahmad Yasin, Yosi Ben-Asher, and Avi Mendelson. 2014. Deep-dive Analysis of the Data Analytics Workload in CloudSuite. In *IISWC*. 202–211.

A FORMULAS

In this section, we present the formulas CMBM, TMAM and the optimized TMAM use. We use the performance events available on the server we profile (see Table 3). While Table 4 shows the constant penalty values used by the formulas, Table 5 presents the formulas for CMBM, Table 6 and 7 present the formulas for TMAM, and Table 8 shows the formula for the optimized Core Bound stalls.

Type	Penalty (cycles)
L1_TO_L2_PENALTY	8
L2_TO_L3_PENALTY	17
L3_TO_DRAM_PENALTY	227
BRANCH_MISP_PENALTY	20
STLB_HIT_LATENCY	7
STORE_FORWARD_BLOCK_PENALTY	13
4K_ALIAS_CONFLICT_PENALTY	5
CONTESTED_ACCESS_PENALTY	60
DATA_SHARING_PENALTY	43
FALSE_SHARING_PENALTY_LOCAL	60
FALSE_SHARING_PENALTY_REMOTE	200
LLC_HIT_TO_MISS_PENALTY_RATIO	7

Table 4: Penalties based on Intel optimization reference manual [7], and the graphical user interface, i.e., GUI, of Intel VTune 2015 [5].

Component	Formula
FE	L1I + L2I + L3I
L1I	(ICACHE.MISSES * L1_TO_L2_PENALTY) / CPU_CLK_UNHALTED.THREAD
L2I	(L2_RQSTS.CODE_RD_MISS * L2_TO_L3_PENALTY) / CPU_CLK_UNHALTED.THREAD
L3I	(OFFCORE_RESPONSE.ALL_CODE_RD.LLC_MISS.ANY_RESPONSE_0 * L3_TO_DRAM_PENALTY) / CPU_CLK_UNHALTED.THREAD
BE	L1D + L2D + L3D
L1D	(MEM_LOAD_UOPS_RETIRED.L1_MISS * L1_TO_L2_PENALTY) / CPU_CLK_UNHALTED.THREAD
L2D	(MEM_LOAD_UOPS_RETIRED.L2_MISS * L2_TO_L3_PENALTY) / CPU_CLK_UNHALTED.THREAD
L3D	(MEM_LOAD_UOPS_RETIRED.LLC_MISS * L3_TO_DRAM_PENALTY) / CPU_CLK_UNHALTED.THREAD
Branch	(BR_MISP_RETIRED.ALL_BRANCHES * BRANCH_MISP_PENALTY) / CPU_CLK_UNHALTED.THREAD

Table 5: CMBM formulas.

Component	Formula
Retiring	UOPS_RETIRED.RETIRE_SLOTS / (4 * CPU_CLK_UNHALTED.THREAD)
FE	IDQ_UOPS_NOT_DELIVERED.CORE / (4 * CPU_CLK_UNHALTED.THREAD)
FE Latency	IDQ_UOPS_NOT_DELIVERED.CYCLES_0_UOPS_DELIV.CORE / CPU_CLK_UNHALTED.THREAD
iTLB	(ITLB_MISSES.STLB_HIT * STLB_HIT_LATENCY + ITLB_MISSES.WALK_DURATION) / CPU_CLK_UNHALTED.THREAD
iCache Miss	ICACHE.IFETCH_STALL / CPU_CLK_UNHALTED.THREAD
FE BW	FE - FE Latency
Branch restreers	((BR_MISP_RETIRED.ALL_BRANCHES + BACLEARS.ANY + MACHINE_CLEARS.COUNT) * ((RS_EVENTS.EMPTY_CYCLES - ICACHE.IFETCH_STALL) / RS_EVENTS.EMPTY_END)) / CPU_CLK_UNHALTED.THREAD
DSB switches	DSB2MITE_SWITCHES.PENALTY_CYCLES / CPU_CLK_UNHALTED.THREAD
Length changing prefixes	ILD_STALL.LCP / CPU_CLK_UNHALTED.THREAD
FE Misc.	Branch restreers + DSB switches + Length changing prefixes
BE	1 - Retiring - FE - Bad speculation
Memory Bound	(CYCLE_ACTIVITY.STALLS_LDM_PENDING + RESOURCE_STALLS.SB) / CPU_CLK_UNHALTED.THREAD
DTLB	(DTLB_LOAD_MISSES.STLB_HIT * STLB_HIT_LATENCY + DTLB_LOAD_MISSES.WALK_DURATION) / CPU_CLK_UNHALTED.THREAD
L1 Bound	(CYCLE_ACTIVITY.STALLS_LDM_PENDING - CYCLE_ACTIVITY.STALLS_L1D_PENDING) / CPU_CLK_UNHALTED.THREAD
L2 Bound	(CYCLE_ACTIVITY.STALLS_L1D_PENDING - CYCLE_ACTIVITY.STALLS_L2_PENDING) / CPU_CLK_UNHALTED.THREAD
L3 Bound	CYCLE_ACTIVITY.STALLS_L2_PENDING * (MEM_LOAD_UOPS_RETIRED.LLC_HIT / (MEM_LOAD_UOPS_RETIRED.LLC_HIT + LLC_HIT_TO_MISS_PENALTY_RATIO * MEM_LOAD_UOPS_RETIRED.LLC_MISS))
DRAM Bound	CYCLE_ACTIVITY.STALLS_L2_PENDING * ((LLC_HIT_TO_MISS_PENALTY_RATIO * MEM_LOAD_UOPS_RETIRED.LLC_MISS) / (MEM_LOAD_UOPS_RETIRED.LLC_HIT + LLC_HIT_TO_MISS_PENALTY_RATIO * MEM_LOAD_UOPS_RETIRED.LLC_MISS))
Store Bound	RESOURCE_STALLS.SB / CPU_CLK_UNHALTED.THREAD
Core Bound	(CYCLE_ACTIVITY.CYCLES_NO_EXECUTE + UOPS_EXECUTED.CYCLES_GE_1_UOP_EXEC - UOPS_EXECUTED.CYCLES_GE_3_UOPS_EXEC - RS_EVENTS.EMPTY_CYCLES) / CPU_CLK_UNHALTED.THREAD) - Memory Bound
Loads blocked by store forwarding	(LD_BLOCKS.STORE_FORWARD * STORE_FORWARD_BLOCK_PENALTY) / CPU_CLK_UNHALTED.THREAD
Split loads	((L1D_PEND_MISS.PENDING / MEM_LOAD_UOPS_RETIRED.L1_MISS) * LD_BLOCKS.NO_SR) / CPU_CLK_UNHALTED.THREAD

Table 6: TMAM formulas, Part I.

Component	Formula
4K aliasing	(LD_BLOCKS_PARTIAL.ADDRESS_ALIAS * 4K_ALIAS_CONFLICT_PENALTY) / CPU_CLK_UNHALTED.THREAD
Contested accesses	((MEM_LOAD_UOPS_LLC_HIT_RETIRED.XSNP_HITM_PS + MEM_LOAD_UOPS_LLC_HIT_RETIRED.XSNP_MISS_PS) * CONTESTED_ACCESS_PENALTY) / CPU_CLK_UNHALTED.THREAD
Data sharing	(MEM_LOAD_UOPS_LLC_HIT_RETIRED.XSNP_HIT_PS * DATA_SHARING_PENALTY) / CPU_CLK_UNHALTED.THREAD
False sharing	(FALSE_SHARING_PENALTY_LOCAL * OFFCORE_RESPONSE.DEMAND_RFO.LLC_HIT.HITM_OTHER_CORE + FALSE_SHARING_PENALTY_REMOTE * OFFCORE_RESPONSE.DEMAND_RFO.LLC_MISS.REMOTE_HITM) / CPU_CLK_UNHALTED.THREAD
BE Misc.	Loads blocked by store forwarding + Split loads + 4K aliasing + Contested accesses + Data sharing + False sharing
Bad Speculation	(UOPS_ISSUED.ANY - UOPS_RETIRED.RETIRE_SLOTS + 4 * INT_MISC.RECOVERY_CYCLES / (4 * CPU_CLK_UNHALTED.THREAD)

Table 7: TMAM formulas, Part II.

Component	Formula
Core Bound	(CYCLE_ACTIVITY.CYCLES_NO_EXECUTE + UOPS_EXECUTED.CYCLES_GE_1_UOP_EXEC - UOPS_EXECUTED.CYCLES_GE_4_UOP_EXEC - IDQ_UOPS_NOT_DELIVERED.CYCLES_LE_3_UOP_DELIV.CORE) / CPU_CLK_UNHALTED.THREAD) - Memory Bound

Table 8: The formula for the optimized Core Bound stalls.