Liu J, Espina P, Sun XH. A study on modeling and optimization of memory systems. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 36(1): 71–89 Jan. 2021. DOI 10.1007/s11390-021-0771-8

A Study on Modeling and Optimization of Memory Systems

Jason Liu[1], Pedro Espina[1], and Xian-He Sun[2], *Fellow*, *IEEE*

[1]*School of Computing and Information Sciences, Florida International University, Miami, FL 33199, U.S.A.*
[2]*Department of Computer Science, Illinois Institute of Technology, Chicago, IL 60616, U.S.A*

E-mail: liux@cis.fiu.edu; pespi004@fiu.edu; sun@iit.edu

Received July 2, 2020; accepted November 19, 2020.

Abstract Accesses Per Cycle (APC), Concurrent Average Memory Access Time (C-AMAT), and Layered Performance Matching (LPM) are three memory performance models that consider both data locality and memory assess concurrency. The APC model measures the throughput of a memory architecture and therefore reflects the quality of service (QoS) of a memory system. The C-AMAT model provides a recursive expression for the memory access delay and therefore can be used for identifying the potential bottlenecks in a memory hierarchy. The LPM method transforms a global memory system optimization into localized optimizations at each memory layer by matching the data access demands of the applications with the underlying memory system design. These three models have been proposed separately through prior efforts. This paper reexamines the three models under one coherent mathematical framework. More specifically, we present a new memory-centric view of data accesses. We divide the memory cycles at each memory layer into four distinct categories and use them to recursively define the memory access latency and concurrency along the memory hierarchy. This new perspective offers new insights with a clear formulation of the memory performance considering both locality and concurrency. Consequently, the performance model can be easily understood and applied in engineering practices. As such, the memory-centric approach helps establish a unified mathematical foundation for model-driven performance analysis and optimization of contemporary and future memory systems.

Keywords performance modeling, performance optimization, memory architecture, memory hierarchy, concurrent average memory access time

1 Introduction

The "memory wall" problem, first coined by Wulf and McKee in 1994 [1], refers to the growing disparity between CPU and memory speed that causes memory accesses to become a severe performance bottleneck in modern computer architectures. Although we have seen significant changes in the landscape of computing over the last two and half decades, e.g., with the introduction of multi-core and many-core design, deep pipelining, and multi-port, multi-banked, pipelined cache, non-blocking cache, as well as the advent of non-volatile memory technologies—the "memory wall" problem persists, and in fact has become more severe as we are now faced with a deluge of modern big-data and data-intensive applications.

There are three outstanding issues with the current "memory wall" problem. The first is a scale-up issue. As new memory technologies are introduced into the market, the memory hierarchy (albeit now somewhat blurs with the storage hierarchy, as illustrated in Fig.1) has become even deeper and the disparity between the CPU speed and memory access latency at both ends of the spectrum has grown wider. While increasing memory hardware performance is certainly expected, the history shows that the pace of advancement for different technologies is often unbalanced. Every time when memory hardware performance increases, CPU computing power grows by an even larger amount.

The second is a scale-out issue. With the increas-

Regular Paper
Special Section on Memory-Centric System Research for High-Performance Computing
This work is supported in part by the U.S. National Science Foundation under Grant Nos. CCF-2008000, CNS-1730488, and CCF-2008907, and the U.S. Department of Homeland Security under Grant No. 2017-ST-062-000002.

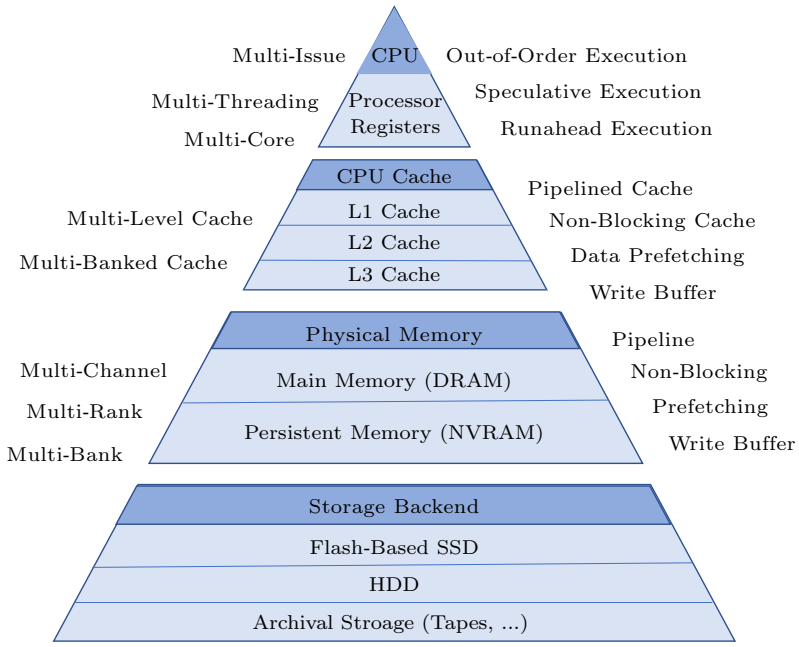


Fig.1. Memory and storage hierarchy.

ing core count in modern architecture design, the performance of applications running on these systems can be severely hindered by the data access latency with the increasing traffic at the threshold of the underlying memory system.

The third issue is that the memory architecture has become more heterogeneous, largely due to the shortened time-to-market for a variety of emerging memory/storage devices (the recently introduced 3D-XPoint memory devices being a case in point①②). Such heterogeneity inevitably introduces more diverging architectural designs. Many important applications, including various database, big data, data processing, and machine learning applications, are inherently data-intensive which come with complex data access patterns, variations in execution timing behaviors, and memory locality characteristics. Consequently, memory performance optimization has become increasingly more complex for modern computing architectures.

In addition to improvements in hardware devices, memory performance optimization generally falls into two categories, optimization to improve data locality, and optimization to improve data concurrency. Locality encompasses both temporal locality (the data previously accessed are more likely to be reused soon) and spatial locality (the data close to previously accessed data are more likely to be accessed). Locality-based optimization is a well-studied topic and is the focus of data access optimization for many years (e.g., [2, 3]). Techniques exploiting data concurrency or memory parallelism, such as out-of-order execution and non-blocking cache, have also been applied in modern systems to effectively overlap computation and memory accesses [4]. We posit the overall performance optimization of modern memory architectures has to combine both locality and concurrency and encompass all layers of the memory hierarchy.

Many memory utilization methods have been developed during the years, from hardware design to software configuration. They operate under different assumptions and with different limitations, and as such, are oftentimes entangled and may conflict with one another. Toward achieving the joint optimization of the memory architecture, we focus on the integration of three earlier proposed performance models: C-AMAT, APC, and LPM. We revisit these models from a memory-centric view to make it easier for them to be understood and used in engineering practices.

①Intel Optane™ DC Persistent Memory. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html, Nov. 2020.

②Intel Optane™ Memory. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-memory.html, Nov. 2020.

Concurrent Average Memory Access Time (C-AMAT) is a memory performance metric that accounts for both locality and concurrency [5]. C-AMAT extends the conventional AMAT (Average Memory Access Time) model [1] by including memory concurrency in the calculation of memory access time. Accesses Per Cycle (APC) is another performance metric for evaluating modern memory architectures, which measures the number of memory accesses per cycle [6]. APC can be applied at each level of the memory hierarchy using existing hardware counters, and consequently can capture the overall memory performance. Conducting performance optimization of modern memory systems is complicated. In a hierarchical memory system, the performance of a memory layer can influence and be influenced by other memory layers. It is difficult to translate the contribution from an optimization at one memory layer in the overall memory system performance. The Layered Performance Matching (LPM) method [7] extends the common idea of matching the data request rate and the data supply rate in memory systems, and makes a significant contribution in segmenting a complex whole memory system optimization into relatively simpler optimizations at each memory layer, and then combining them to achieve an overall performance objective.

While these three separately proposed models provide the necessary foundation for improving memory system performance for architectural design and applications, they have never been studied under one coherent mathematical framework. Consequently, the understanding of their practical implications on contemporary and future computing systems has been limited. Furthermore, since C-AMAT is an extension of AMAT, its original derivation closely follows the AMAT notations and its logical flow. AMAT is derived from a CPU-centric view where cache hits and cache misses are accounted for by individual memory accesses when instructions are being executed by CPU. This CPU-centric view makes it difficult to explain the overlap of memory accesses at different cache levels in the memory hierarchy. Both C-AMAT and LPM are not naturally supported by existing simulators and measurement tools. The CPU-centric view also makes it difficult to measure the memory access concurrency at different cache layers.

In this paper, we reexamine the prior work of the C-AMAT, APC, and LPM models [5–7] and cast them in the same mathematical framework with a memory-centric view. We redefine some model parameters to achieve better consistency across these models and make it easier to measure and use them in engineering practices. We focus on their derivations so as to provide more clarity and better insight of these models. More specifically, we revisit the three models using a top-down approach. We start with the run time, define the memory stall time (MST), and then examine on the memory performance metrics of AMAT and C-AMAT. We extend the performance analysis from a first-level cache to a multi-level memory hierarchy. We present a new perspective and thereby new proofs of C-AMAT, APC, and LPM using a memory-centric view. We divide memory cycles into four categories: inactive cycles, pure hit cycles, pure miss cycles, and mixed hit/miss cycles, and then use these four types to define the memory access concurrency and the hierarchical memory behavior.

Using the memory-centric approach, we establish a concrete mathematical foundation for model-driven performance analysis and optimization of memory systems. Such a performance model provides us with a better understanding of the memory system performance and thus paves the way for developing next-generation memory system simulators and measurement tools. In doing so, it will aid the design and development of more effective memory architectures for modern computer systems, potentially featuring deep and diverse memory system hierarchy, heterogeneous memory devices, and complex data-intensive applications, including big data, cloud and data centers, high-performance computing applications.

Our major contributions can be summarized as follows.

• We combine the C-AMAT, APC, and LPM models within a coherent mathematical framework with new and detailed derivations.

• We apply a memory-centric approach to modeling the hierarchical memory system by classifying the memory cycles into four distinct categories, and use them to define the memory access concurrency and the hierarchical memory behavior.

• We provide a much clearer recursive definition of C-AMAT using the memory-centric approach, and a formal proof on the correctness of the LPM optimization method.

For consistency, Table 1 shows the list of model parameters and their definitions of the hierarchical memory performance model described in this paper (in the order of their first appearances). Detailed explanation of these model parameters is located at the respec-

Table 1. Model Parameters: Definitions and Locations

Name	At Level l	Definition	Equation
n	n	Total number of CPU cycles	(4)
T	T	Run time of the program	(1)
T_{cycle}	T_{cycle}	CPU cycle time	
f_{mem}	f_{mem}	Average number of memory accesses per instruction	
α	$\alpha(l)$	Total number of memory accesses	(2), (9), (44), (45)
IC	IC	Instruction count	
CPI_{exe}	CPI_{exe}	Cycles per instruction (without memory stall)	
IPC_{exe}	IPC_{exe}	Instructions per cycle (without memory stall)	
MST	MST	Memory stall time per memory access	(18), (35)
Δ	Δ	Ratio of memory access time over compute time	(5)
MSE	MSE	Memory system efficiency	(6)
α_h	$\alpha_h(l)$	Number of hit memory accesses	(7)
α_m	$\alpha_m(l)$	Number of miss memory accesses	(8)
ρ_h	$\rho_h(l)$	Hit ratio	(7)
ρ_m	$\rho_m(l)$	Miss ratio	(8)
H	$H(l)$	Hit time	(20)
AMT	$AMT(l)$	Average miss time	(10)
AMP	$AMP(l)$	Average miss penalty	(21)
$\mathbb{E}$	$\mathbb{E}(l)$	Set of memory inactive cycles	
e	$e(l)$	Number of memory inactive cycles	
$\mathbb{H}$	$\mathbb{H}(l)$	Set of pure hit cycles	
h	$h(l)$	Number of pure hit cycles	
$\mathbb{M}$	$\mathbb{M}(l)$	Set of pure miss cycles	
m	$m(l)$	Number of pure miss cycles	
$\mathbb{X}$	$\mathbb{X}(l)$	Set of mixed hit/miss cycles	
x	$x(l)$	Number of mixed hit/miss cycles	
Ω	$\Omega(l)$	Set of memory active cycles	(11)
ω	$\omega(l)$	Number of memory active cycles	(12), (49), (50)
ϕ	$\phi(l)$	Ratio of hit cycles over memory active cycles	(13)
μ	$\mu(l)$	Ratio of miss cycles over memory active cycles	(14)
κ	$\kappa(l)$	Ratio of pure miss cycles over miss cycles	(15), (40)
α_M	$\alpha_M(l)$	Number of pure miss memory accesses	
ρ_M	$\rho_M(l)$	Pure miss ratio	(17)
c_i	$c_i(l)$	Memory access concurrency at CPU cycle i	(19)
$c_i^{(h)}$	$c_i^{(h)}(l)$	Hit concurrency at CPU cycle i	
$c_i^{(m)}$	$c_i^{(m)}(l)$	Miss concurrency at CPU cycle i	
C	$C(l)$	Average memory access concurrency	(22)
C_H	$C_H(l)$	Average hit concurrency	(23), (25)
C_m	$C_m(l)$	Average miss concurrency	(26)
C_M	$C_M(l)$	Average pure miss concurrency	(27)
$AMAT$	$AMAT(l)$	Average memory access time	(28), (29), (31), (39)
$C\text{-}AMAT$	$C\text{-}AMAT(l)$	Concurrent average memory access time	(30), (31), (33), (41), (42)
APC	$APC(l)$	Accesses per cycle	(32)
$pAMP$	$pAMP(l)$	Average pure miss penalty	(34)
L	L	Number of cache levels	
$LPMR$	$LPMR(l)$	Matching ratio	(43), (52), (53), (55)
λ	$\lambda(l)$	Request rate	(46)–(48)
ν	$\nu(l)$	Supply rate	(51)

tive part of the text, and we identify in the table the specific equations in which they are derived. The first column of the table lists the model parameters used by the special case developed only for the first-level cache. These model parameters are defined and used in Sections 2–5. The second column of the table lists the corresponding model parameters used in the generalized model to analyze multiple levels of cache. These model parameters are defined and used in Section 6 and onwards.

The rest of this paper is organized as follows. The first three sections to follow establish the necessary foundation for the hierarchical memory performance model. More specifically, in Section 2, we examine the overall program runtime and the memory system efficiency. In Section 3, we examine the memory accesses at the granularity of CPU cycles. In Section 4, we formally define the concurrency as the memory accesses overlap with the execution of the instructions by CPU. The definitions outlined in these three sections are instrumental to the derivation of the time-based and rate-based analyses described in the subsequent sections.

Section 5 focuses on the average memory access time, in particular, the concurrent average memory access time. For simple exposition, Sections 3–5 deal with the first-level cache, as its performance represents the overall memory system performance. In Section 6, we generalize the performance model to analyze multiple layers of cache. A recursive definition of the memory access time is derived, as it serves as the basis for the rate-based analysis of memory accesses and the concept of layered performance matching, which we describe in Section 7. We discuss the potential contributions and practical implications of the unified analysis in Section 8. Finally, we discuss related work in Section 9 and conclude the paper in Section 10.

2 Runtime

Run time is the ultimate concern of the performance model. We start out our performance analysis with runtime. A program's execution consists of n CPU cycles: $1, 2, 3, \cdots, n$, where n is the total number of CPU cycles. Let T be the run time of the program, which would be:

$$T = n \times T_{\text{cycle}}, \tag{1}$$

where T_{cycle} is the time of each CPU cycle. Since T_{cycle} is a constant for a given CPU, for the rest of this paper, we simply use the number of CPU cycles to represent the run time.

Let f_{mem} be the average number of memory accesses per instruction ($f_{\text{mem}} \geqslant 0$). Let α be the total number of memory accesses:

$$\alpha = IC \times f_{\text{mem}}, \tag{2}$$

where IC is the instruction count (i.e., the total number of instructions executed by the program).

CPI_{exe} is the number of cycles per instruction, i.e., the average number of CPU cycles needed to execute an instruction (averaged among all types of instructions), assuming there is no memory stall. The reciprocal of CPI_{exe} is IPC_{exe}, which is the number of instructions per cycle, again assuming no memory stall. We have:

$$CPI_{\text{exe}} \times IPC_{\text{exe}} = 1. \tag{3}$$

MST is the memory stall time. It is the average number of CPU cycles that the CPU is stalled waiting for a memory access. The run time of a program can be decomposed into two parts: the compute time ($IC \times CPI_{\text{exe}}$) and the memory time ($\alpha \times MST$). Thus, we have:

$$\begin{aligned} n &= IC \times CPI_{\text{exe}} + \alpha \times MST \\ &= IC \times (CPI_{\text{exe}} + f_{\text{mem}} \times MST)\,. \end{aligned} \tag{4}$$

The above equation reveals that the run time is ultimately determined by the CPU computing speed (related to CPI_{exe}), the program behavior (related to IC and f_{mem}), and the memory system performance (related to MST). One important feature of our study is the use of MST as the measure of performance of memory systems. In Section 3, we break down the memory accesses of a program along the CPU cycles to obtain a first-hand estimation of the memory stall time. We then develop a more comprehensive hierarchical memory performance model over the subsequent sections.

Note that memory stalls are different from pipeline hazards that prevent instructions from being executed at designated CPU cycles in the pipeline [8]. Pipeline hazards include structural hazards (caused by conflicts in the simultaneous use of CPU resources), data hazards (caused by data dependencies), and control hazards (caused by changes in the control flow). We do not consider pipeline hazards in our study as they are not related to the memory systems; their overall runtime effect should have been captured by CPI_{exe} in (4).

To improve performance, we want MST to be small. However, its value needs to be considered relative to the compute time. Let Δ be the ratio of memory access time over compute time. That is,

$$\begin{aligned} \Delta &= \frac{\text{memory access time}}{\text{compute time}} \\ &= \frac{\alpha \times MST}{IC \times CPI_{\text{exe}}} = \frac{MST \times f_{\text{mem}}}{CPI_{\text{exe}}}. \end{aligned} \tag{5}$$

The smaller MST is, the smaller Δ is, and vice versa, when the CPU computing speed and the program behavior stay unchanged. In fact, we can define memory

system efficiency (MSE) exactly as the ratio of the compute time over the entire run time:

$$\begin{aligned} MSE &= \frac{\text{compute time}}{\text{run time}} = \frac{IC \times CPI_{\text{exe}}}{IC \times CPI_{\text{exe}} + \alpha \times MST} \\ &= \frac{CPI_{\text{exe}}}{CPI_{\text{exe}} + MST \times f_{\text{mem}}} \\ &= \frac{1}{1+\Delta} \approx 1 - \Delta. \end{aligned} \tag{6}$$

The last approximation can be obtained using Taylor series expansion when $|\Delta| << 1$. Especially, when $MST \to 0$, $\Delta \to 0$ and $MSE \to 1$.

3 Memory Accesses and CPU Cycles

Memory speed is usually given in memory clock cycles. Since memory cycles are always a fixed multiple of the CPU cycles, for simplicity, we directly use the CPU cycles to measure and analyze the memory performance in this study. A memory access can be either a hit memory access or a miss memory access. A hit or a miss corresponds to whether the specific data of the memory access is present in the cache or not. For now, our analysis focuses on the first cache level. We extend our analysis to multi-level caches in subsequent sections.

Let α_h be the number of hit memory accesses. Let α_m be the number of miss memory accesses. Let ρ_h denote the hit ratio, and let ρ_m denote the miss ratio. They are simply the proportion of hit memory accesses and miss memory accesses, respectively:

$$\rho_h = \frac{\alpha_h}{\alpha}, \quad \alpha_h = \alpha \times \rho_h, \tag{7}$$

$$\rho_m = \frac{\alpha_m}{\alpha}, \quad \alpha_m = \alpha \times \rho_m. \tag{8}$$

Certainly,

$$\rho_h + \rho_m = 1, \quad \alpha_h + \alpha_m = \alpha. \tag{9}$$

Let H be the hit time, which is the duration of a hit memory access, in the number of CPU cycles. This hit time is an architecture-dependent constant. Let AMT be the average miss time, which is the average duration of a miss memory access, again in the number of CPU cycles. AMT is an indication of a program's memory performance as a function of the program's data access behavior (such as data locality) and the memory architecture on which the program is run. One can express AMT in two parts.

1) *Hit Portion.* It is the same as the hit time, H. It represents the time a memory access (regardless whether it is a hit or a miss) uses to "visit" the cache.

2) *Miss Portion.* It is defined as the average miss penalty, denoted by AMP. AMP is the additional time in the number of CPU cycles (which can be a fraction) for handling the cache miss (such as fetching the data from memory to cache).

H and AMP are two parameters used in AMAT [1]. Assuming there is no concurrent memory access, we have:

$$AMT = H + AMP. \tag{10}$$

The CPU cycles for memory accesses can therefore be classified into hit-access cycles and miss-access cycles. On average, a hit memory access consists of H hit-access cycles. A miss memory access consists of H hit-access cycles and AMP miss-access cycles. Accordingly, a program's run time in CPU cycles $(1, 2, 3, \cdots, n)$ on a memory layer can be divided into four types.

1) *Memory Inactive Cycles.* Here no memory activities occur at the memory layer during these cycles.

- Let $\mathbb{E}$ be the set of memory inactive cycles.
- Let $e = |\mathbb{E}|$ be the number of memory inactive cycles.

2) *Pure Hit Cycles.* They contain only the hit-access cycles, regardless whether they belong to hit memory accesses or miss memory accesses. Recall that a hit memory access contains only hit-access cycles, and a miss memory access contains both hit-access cycles and miss-access cycles. The hit-access cycles count for the time the memory access is "visiting" the cache.

- Let $\mathbb{H}$ be the set of pure hit cycles.
- Let $h = |\mathbb{H}|$ be the number of pure hit cycles.

3) *Pure Miss Cycles.* They contain only the miss-access cycles.

- Let $\mathbb{M}$ be the set of pure miss cycles.
- Let $m = |\mathbb{M}|$ be the number of pure miss cycles.

4) *Mixed Hit/Miss Cycles.* They contain both hit-access cycles and miss-access cycles. In other words, a mixed hit/miss cycle contains at least one hit-access cycle and at least one miss-access cycle.

- Let $\mathbb{X}$ be the set of mixed hit/miss cycles.
- Let $x = |\mathbb{X}|$ be the number of mixed hit/miss cycles.

The total number of CPU cycles is the sum of the cycles of the four types:

$$n = e + h + m + x.$$

Since pure hit cycles, pure miss cycles, and mixed hit/miss cycles are the three possible states of memory activity at the memory layer, they are collectively called memory active cycles.

- Let Ω be the set of memory active cycles.

$$\Omega = \mathbb{H} \cup \mathbb{M} \cup \mathbb{X}. \tag{11}$$

- Let ω be the number of memory active cycles.

$$\omega = h + m + x. \tag{12}$$

We simply call the pure hit cycles and the mixed hit/miss cycles collectively as hit cycles. Similarly, we call the pure miss cycles and the mixed hit/miss cycles collectively as miss cycles.

Note that we use the terms, hit-access cycles and miss-access cycles, to refer to the two parts of a particular memory access. We use the terms, memory inactive cycles, pure hit cycles, pure miss cycles, and mixed hit/miss cycles, as well as hit cycles, miss cycles, and memory active cycles, to categorize memory activities of a program's execution at each memory layer at the granularity of CPU cycles. We call them collectively as memory cycles. The memory cycles are aligned in time across the memory hierarchy (shown as the slotted time separated by vertical dash lines in Figs.2 and 3).

Let ϕ be the ratio of hit cycles over memory active cycles:

$$\phi = \frac{h + x}{\omega}. \tag{13}$$

The hit cycles are the cycles that contain at least a hit at the first-level cache. For that, we assume that they can be serviced without memory stall. Let μ be the ratio of miss cycles over memory active cycles and let κ be the ratio of pure miss cycles over miss cycles. That is,

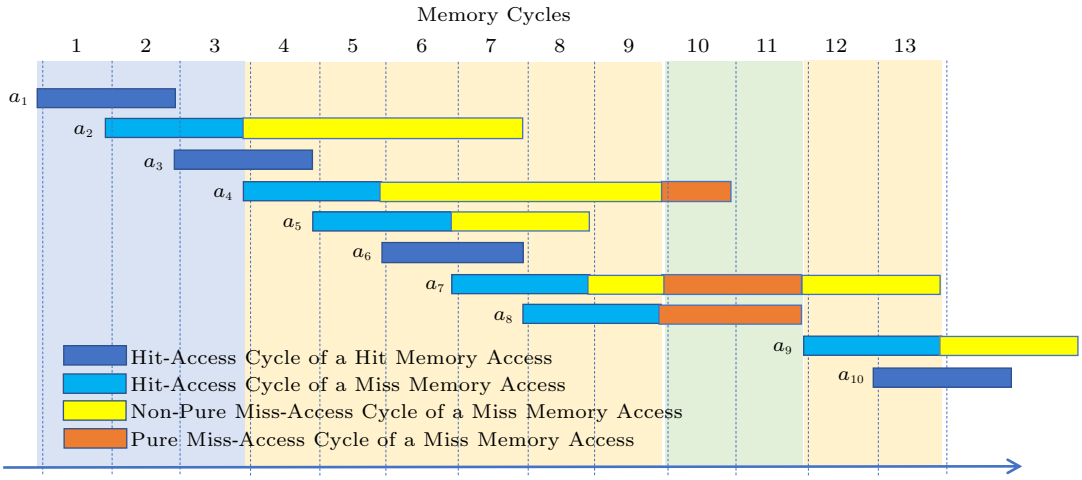
$$\mu = \frac{m + x}{\omega}, \tag{14}$$

$$\kappa = \frac{m}{m + x}. \tag{15}$$

We can derive the following equality (which we use later for deriving the memory stall time as a function of the average memory access time):

$$\begin{aligned} 1 - \phi &= 1 - \frac{h + x}{\omega} = \frac{m}{\omega} \\ &= \frac{m + x}{\omega} \times \frac{m}{m + x} = \mu \times \kappa. \end{aligned} \tag{16}$$

Fig.2 illustrates the memory accesses and CPU cycles. A hit memory access contains only hit-access cycles and therefore can straddle on pure hit cycles and mixed hit/miss cycles. For example, hit memory access a_1 contains only pure hit cycles (1 and 2); a_3 contains both pure hit cycles and mixed hit/miss cycles (3 and 4); a_6 contains only mixed hit/miss cycles (6 and 7).



c	1	2	2	3	3	4	5	4	3	3	2	2	3
$c^{(h)}$	1	2	2	2	2	2	2	2	1	0	0	1	2
$c^{(m)}$	0	0	0	1	1	2	3	2	2	3	2	1	1

Fig.2. CPU cycles and memory accesses. In this example, we set the hit time $H = 2$. It shows as the x-axis the memory active CPU cycles, which consist of three types: pure hit cycles (1–3), pure miss cycles (10, 11), and mixed hit/miss cycles (4–9 and 12–13). It shows 10 memory accesses: $a_1, a_2, \cdots, a_{10}$, among which a_1, a_3, a_6, a_{10} are hit memory accesses (colored in dark blue). The rest of them, $a_2, a_4, a_5, a_7, a_8, a_9$, are miss memory accesses. The hit-access cycles of the miss memory accesses are colored in light blue. For easy exposition, the hit-access cycles of a miss memory access are always placed at the beginning of the memory access. The miss-access cycles of the miss memory accesses are colored either in yellow (for non-pure miss-access cycles, i.e., if they are located in mixed hit/miss cycles) or in orange (for pure miss-access cycles, if they are located in pure miss cycles).

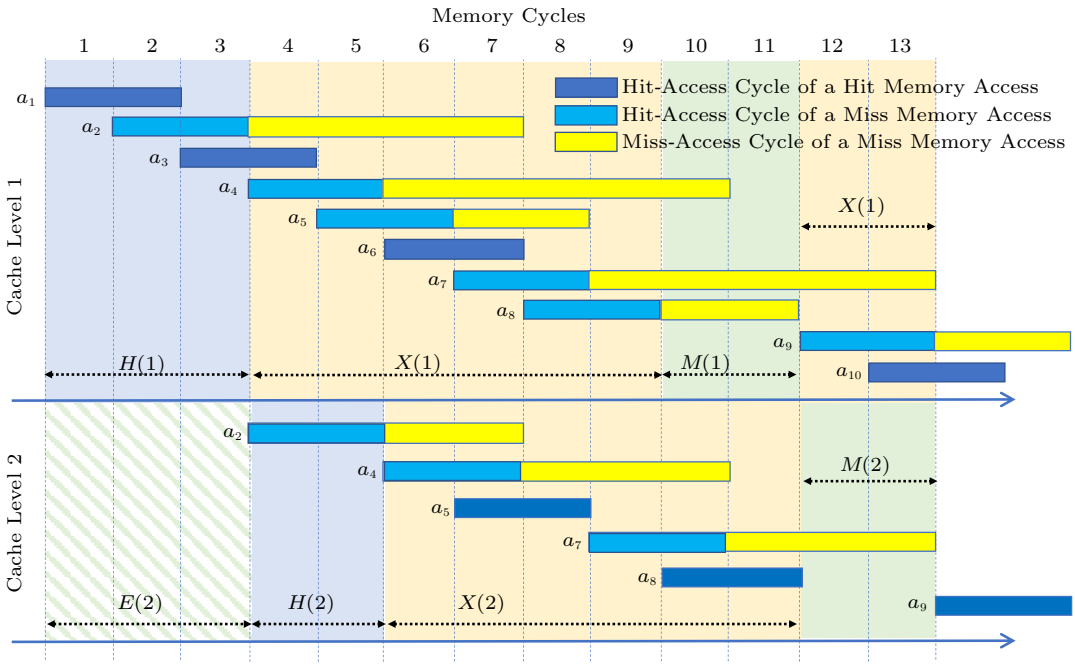


Fig.3. Memory accesses traversing two cache layers. In this example, we set the hit time $H(1) = H(2) = 2$. It shows 10 memory accesses at cache level 1: $a_1, a_2, \cdots, a_{10}$, among which a_1, a_3, a_6, a_{10} are hit memory accesses (colored in dark blue). The rest of them are miss memory accesses. The hit-access cycles of the miss memory accesses are colored in light blue. The miss-access cycles of the miss memory accesses are colored in yellow. Note that only the miss-access cycles at cache level 1 are present at cache level 2, which are also divided into hit-access cycles (for both hit and miss memory accesses) and miss-access cycles.

A miss memory access contains both hit-access cycles and miss-access cycles, and therefore can straddle on all types of memory active cycles (pure hit cycles, pure miss cycles, and mixed hit/miss cycles). However, the miss portion of a miss memory access contains only miss-access cycles and therefore can straddle only on pure miss cycles and mixed hit/miss cycles. For clarity, we call the miss-access cycles that fall in the pure miss cycles "pure miss-access cycles", and call the miss-access cycles that fall in the mixed hit/miss cycles "non-pure miss-access cycles". In the example, a_2, a_5, a_9 contain only non-pure miss-access cycles; a_8 contains only pure miss-access cycles; a_4 and a_7 contain both non-pure and pure miss-access cycles.

We differentiate two types of miss memory accesses. A "non-pure miss memory access" is a miss memory access that does not contain any pure miss cycles. That is, the miss portion of a non-pure miss memory access contains only non-pure miss-access cycles. A "pure miss memory access" is a memory access that contains at least one pure miss cycle. A pure miss memory access can have both non-pure and pure miss-access cycles, but with at least one pure miss-access cycle. We make such a distinction between a non-pure miss access and a pure miss memory access because we consider only the latter contributes to memory stalls.

Accordingly, we use α_M to denote the number of pure miss memory accesses, and use ρ_M to denote the pure miss ratio.

$$\rho_M = \frac{\alpha_M}{\alpha}, \quad \alpha_M = \alpha \times \rho_M. \tag{17}$$

Finally, we are ready to express the memory stall time, MST, with the definitions presented earlier in this section. MST is the average number of CPU cycles that the CPU is stalled waiting for each memory access. Since we assume that only pure miss cycles contribute to memory stalls, we can calculate MST as the average number of pure miss cycles among all memory accesses:

$$MST = \frac{m}{\alpha}. \tag{18}$$

While (18) is simple, it can be difficult to account for the pure miss cycles to calculate MST in practice. In Section 5, we associate the memory stall time with the concurrent average memory access time. Before that, we need to understand how memory accesses are overlapped at the cache at the granularity of CPU cycles, which we elaborate in Section 4.

4 Memory Access Concurrency

Let c_i be the memory access concurrency at CPU cycle i, where $1 \leqslant i \leqslant n$ and n is the total number of CPU cycles of the program's execution. c_i is the number of overlapped memory accesses (including both hit memory accesses and miss memory accesses) during the CPU cycle i. We have $c_i = 0$ if $i \in \mathbb{E}$ since memory inactive cycles do not overlap with any memory accesses. Also, we have $c_i \geqslant 1$ if $i \in \Omega$.

Let $c_i^{(h)}$ be the hit concurrency, which is the number of overlapped hit-access cycles at CPU cycle i. Similarly, let $c_i^{(m)}$ be the miss concurrency, which is the number of overlapped miss-access cycles at CPU cycle i. We have:

$$c_i = c_i^{(h)} + c_i^{(m)}, \quad (1 \leqslant i \leqslant n). \tag{19}$$

Note that the hit concurrency contains hit-access cycles that belong to either hit memory accesses or the hit portion of miss memory accesses; and the miss concurrency contains only the miss-access cycles. Thus, we can calculate the hit time by adding all the hit-access cycles divided by the total number of memory accesses:

$$H = \frac{1}{\alpha} \sum_{i=1}^{n} c_i^{(h)}. \tag{20}$$

Similarly, we can calculate the average miss penalty (i.e., the average length of the miss portion of the miss memory accesses) by adding all the miss-access cycles divided by the total number of miss memory accesses:

$$AMP = \frac{1}{\alpha_m} \sum_{i=1}^{n} c_i^{(m)}. \tag{21}$$

The table in Fig.2 shows the different memory access concurrency values corresponding to the CPU cycles. For example, at CPU cycle 7, there are two hit-access cycles (one from a hit memory access a_6 and the other from a miss memory access a_7), and three miss-access cycles (two from non-pure miss memory accesses a_2 and a_5, and one from a pure miss memory access a_4). We have $c_7 = 5$, $c_7^{(h)} = 2$, and $c_7^{(m)} = 3$.

Let C be the average memory access concurrency, which is the number of memory accesses (including both hit and miss memory accesses) divided by the total number of memory active CPU cycles:

$$C = \frac{1}{\omega} \sum_{i \in \Omega} c_i = \frac{1}{\omega} \sum_{i=1}^{n} c_i. \tag{22}$$

Let C_H be the average hit concurrency, which is the average number of overlapped hit-access cycles among the CPU cycles that contain at least one hit-access cycle. These would include all pure hit cycles and mixed hit/miss cycles.

$$C_H = \frac{1}{h+x} \sum_{i \in \mathbb{H} \cup \mathbb{X}} c_i^{(h)} = \frac{1}{h+x} \sum_{i=1}^{n} c_i^{(h)}. \tag{23}$$

Since each memory access (whether it is a hit or a miss memory access) contains H hit-access cycles, the total number of hit-access cycles should be:

$$\alpha \times H = \sum_{i=1}^{n} c_i^{(h)} = (h+x) \times C_H. \tag{24}$$

One can thus calculate the average hit concurrency as:

$$C_H = \frac{\alpha \times H}{h+x}. \tag{25}$$

Let C_m be the average miss concurrency, which is the average number of overlapped miss memory accesses during the miss cycles (including pure miss and mixed hit/miss cycles).

$$C_m = \frac{1}{m+x} \sum_{i \in \mathbb{M} \cup \mathbb{X}} c_i^{(m)} = \frac{1}{m+x} \sum_{i=1}^{n} c_i^{(m)}. \tag{26}$$

Let C_M be the average pure miss concurrency, which is the average number of overlapped pure miss memory accesses during the pure miss cycles.

$$C_M = \frac{1}{m} \sum_{i \in \mathbb{M}} c_i^{(m)}. \tag{27}$$

5 Memory Access Time

The AMAT model was originally developed based on a CPU-centric view. The C-AMAT model is an extension of AMAT with its original derivations closely following the AMAT notations and logical flow. In the CPU-centric view, cache hits and cache misses are accounted for by individual memory accesses as instructions are being executed by the CPU. Although seemingly intuitive, it turns out that it is difficult to explain the overlap of memory accesses that occur at different cache levels in the memory hierarchy using the CPU-centric approach.

Equipped with the definitions in Section 4, in this section we derive the average memory access time using the memory-centric approach. We first look at the traditional AMAT model and then extend it for the C-AMAT model, which incorporates both memory access locality and concurrency. The memory-centric view facilitates an easier understanding of the two models and

provides the foundation for a recursive definition of the memory performance model for all layers of a hierarchical memory system, which we discuss in Section 6.

Average memory access time contributes to the memory stall time for estimating the memory system performance. The relationship is highlighted in Subsection 5.2.

5.1 Average Memory Access Time

Let $AMAT$ be the average memory access time, which is average duration in the number of cycles for each memory access. Conventionally, this is calculated by first adding up the number of cycles of each memory access and then dividing the total by the number of memory accesses. In Fig.2, $AMAT$ is the average length of the horizontal bars. We can calculate $AMAT$ by counting the number of hit-access and miss-access cycles and then taking the average among memory accesses:

$$AMAT = \frac{1}{\alpha}\sum_{i\in\Omega} c_i = \frac{1}{\alpha}\sum_{i=1}^{n} c_i. \tag{28}$$

Alternatively, one can derive $AMAT$ by proportionally adding the memory access time of the hit memory accesses and that of the miss memory accesses:

$$\begin{aligned} AMAT &= \rho_h \times H + \rho_m \times AMT \\ &= \rho_h \times H + \rho_m \times \boxed{(H + AMP)}_{(10)} \\ &= H + \rho_m \times AMP. \end{aligned} \tag{29}$$

Here, the box with the number shows the equation used in the derivation.

One can show that (28) and (29) are actually equivalent.

Proof. $AMAT$ can be calculated using both (28) and (29).

$$\begin{aligned} AMAT &= H + \rho_m \times AMP \\ &= \boxed{\frac{1}{\alpha}\sum_{i=1}^{n} c_i^{(h)}}_{(24)} + \boxed{\frac{\alpha_m}{\alpha}}_{(8)} \times \boxed{\frac{1}{\alpha_m}\sum_{i=1}^{n} c_i^{(m)}}_{(21)} \\ &= \frac{1}{\alpha}\sum_{i=1}^{n}\left(c_i^{(h)} + c_i^{(m)}\right) = \frac{1}{\alpha}\sum_{i=1}^{n} c_i. \end{aligned}$$ □

5.2 Concurrent Average Memory Access Time

The AMAT model is based on the single data access viewpoint. At the time of its inception, memory-level concurrency was uncommon, thereby the sequential latency calculated by AMAT was sufficient. However, since then, memory concurrency technologies have become a norm for the modern memory architecture design. AMAT does not consider memory concurrency upon overlapping memory hits and misses. For example, for a processor supporting out-of-order execution, when a cache miss occurs that fails to provide the data to an instruction, other instructions can be executed while the memory system is serving the cache miss. Multiple outstanding reads and writes can coexist at a given time in the memory system. With wide instruction sets and multiple pipelines, multiple data instructions can be issued at the same time, in addition to outstanding data accesses. In contrast, the C-AMAT model [5] considers locality, concurrency, and overlapping, and thus is more accurate in describing the memory performance of modern computer architectures.

Let $C\text{-}AMAT$ be the concurrent average memory access time, which is the average real time spent for each memory access in terms of CPU cycles. $C\text{-}AMAT$ can be calculated as the number of memory active cycles divided by the number of memory accesses:

$$C\text{-}AMAT = \frac{\omega}{\alpha}. \tag{30}$$

The difference between $C\text{-}AMAT$ and $AMAT$ is that in $AMAT$, overlapped memory accesses each contribute to the total memory access time, while in $C\text{-}AMAT$, concurrency is discounted—overlapped memory accesses only contribute to the total memory access time once. In fact, using the memory-centric approach, one can easily show that:

$$C\text{-}AMAT = C^{-1} \times AMAT, \tag{31}$$

where C is the average memory access concurrency.

Proof. We prove the relationship between $C\text{-}AMAT$ and $AMAT$ as follows:

$$\begin{aligned} C\text{-}AMAT &= C^{-1} \times AMAT = \frac{AMAT}{C} \\ &= \frac{\boxed{\frac{1}{\alpha}\sum_{i=1}^{n} c_i}_{(28)}}{\boxed{\frac{1}{\omega}\sum_{i=1}^{n} c_i}_{(22)}} = \frac{\omega}{\alpha}. \end{aligned}$$ □

The reciprocal of $C\text{-}AMAT$ is called APC, or accesses per cycle [6], which is the average number of memory accesses per memory active cycle:

$$APC = \frac{\alpha}{\omega} = \frac{1}{C\text{-}AMAT}. \tag{32}$$

APC is a measurable quantity on architectures that

provide hardware performance counters. Thus, empirically, $C\text{-}AMAT$ can be measured by runtime profiling.

Alternatively, $C\text{-}AMAT$ can also be expressed using (33) by extending the AMAT formulation to consider concurrency [5]:

$$C\text{-}AMAT = \frac{H}{C_H} + \rho_M \times \frac{pAMP}{C_M}, \tag{33}$$

where $pAMP$ represents the average pure miss penalty, which is the average number of pure miss cycles per pure miss memory access. That is,

$$pAMP = \frac{\sum_{i\in\mathbb{M}} c_i^{(m)}}{\alpha_M}. \tag{34}$$

Proof. One can prove (33) as follows:

$$\begin{aligned}
&\frac{H}{C_H} + \rho_M \times \frac{pAMP}{C_M} \\
&= \frac{H}{\boxed{\frac{\alpha \times H}{h+x}}_{(25)}} + \\
&\frac{\boxed{\frac{\alpha_M}{\alpha}}_{(17)} \boxed{\frac{\sum_{i\in\mathbb{M}} c_i^{(m)}}{\alpha_M}}_{(34)}}{\boxed{\frac{1}{m}\sum_{i\in\mathbb{M}} c_i^{(m)}}_{(27)}} \\
&= \frac{h+x}{\alpha} + \frac{m}{\alpha} = \frac{w}{\alpha} = \boxed{C\text{-}AMAT}_{(30)}.
\end{aligned}$$

□

The concurrent average memory access time, $C\text{-}AMAT$, is directly related to the memory stall time, MST:

$$\begin{aligned}
MST &= \boxed{\frac{m}{\alpha}}_{(18)} = \frac{m}{\omega} \times \frac{\omega}{\alpha} \\
&= \boxed{\mu \times \kappa}_{(16)} \times \boxed{C\text{-}AMAT}_{(30)},
\end{aligned} \tag{35}$$

where μ is the ratio of miss cycles over memory active cycles and κ is the ratio of pure miss cycles over miss cycles, as defined in (14) and (15), respectively.

6 Memory Hierarchy and Recursion

The previous sections deal with memory performance focusing only on the first cache level. The model parameters and equations—for example, the hit ratio ρ_h, the hit time H, and the average hit concurrency C_H, are all defined with respect to the first-level cache.

The cycle-based performance model can naturally be extended for multi-level cache analysis. Let L be the number of cache levels in the memory system, where level 1 is the first-level cache, and level L is the last-level cache. If a memory access reaches the last-level cache and causes a cache miss, the memory access will visit the main memory.

Both $AMAT$ and $C\text{-}AMAT$ can be defined recursively with respect to the cache memory level. The recursive definitions are by-products of the hierarchical memory architecture, and the model parameters used in the recursive definitions are measurable both in simulation and on many real systems. The recursion allows us to capture the contribution of memory access at a cache layer in relation to the adjacent layers for the overall runtime performance. Such information can be crucial for hierarchical memory architecture design and optimization.

6.1 Extensions and Observations

We first extend the model parameters previously used by the special model for the first-level cache. In particular, we add a parameter l ($1 \leqslant l \leqslant L$) to indicate the specific cache level the model parameters are related to. For example, $\rho_h(l)$ is the hit ratio at cache level l, $H(l)$ is the hit time for accessing cache level l (in addition to the hit time already spent in the previous $l-1$ cache levels), and $C_H(l)$ is the hit concurrency at cache level l. The second column of Table 1 contains the model parameters defined for cache level l that correspond to the originally defined model parameters in the first column for the first-level cache. All equations described so far will remain true if we replace the model parameters in the first column of Table 1 with those in the second column.

In the remainder of this subsection, we provide a recursive definition for the memory access time, both $AMAT(l)$ and $C\text{-}AMAT(l)$, based on the cache level l. Before doing so, we make a few observations. Fig.3 is an example showing the same set of memory accesses as in Fig.2, but this time for two cache levels.

Observation 1. Only the miss memory accesses at level l reach level $l+1$. In other words, the memory accesses at the next level are the miss memory accesses

at the previous level[③].

$$\alpha(l+1) = \alpha_m(l) = \alpha(l) \times \rho_m(l), \tag{36}$$

where $1 \leqslant l < L$.

Observation 2. Only the miss portion of the miss memory accesses at level l are the memory accesses at level $l+1$. That is, the overlapped hit-access and miss-access cycles at the next level all come from the miss-access cycles of the previous level.

$$c_i(l+1) = c_i^{(m)}(l), \tag{37}$$

where $1 \leqslant i \leqslant n$ and $1 \leqslant l < L$.

Observation 3. The pure hit cycles at level l are the memory inactive cycles at level $l+1$, because the hit-access cycles of a memory access at a previous cache level (regardless of whether it is a hit memory access or a miss memory access) do not penetrate to the next level; only the miss-access cycles of a miss memory access will traverse to the next cache level.

$$\begin{aligned} \mathbb{E}(l+1) &= \mathbb{E}(l) \cup \mathbb{H}(l), \\ \Omega(l+1) &= \Omega(l) \setminus \mathbb{H}(l). \end{aligned}$$

Accordingly, we can recursively calculate the number of memory active cycles for a cache layer:

$$\begin{aligned} \omega(l+1) &= \omega(l) - h(l) = m(l) + x(l) \\ &= \left(\frac{m(l)+x(l)}{\omega(l)}\right) \times \omega(l) = \mu(l) \times \omega(l), \end{aligned} \tag{38}$$

where $1 \leqslant l < L$. In other words, the number of memory active cycles at the subsequent level $l+1$ shrinks at the rate of $\mu(l)$.

6.2 $AMAT$ Recursion

From observation 2, only the miss portion of the miss memory accesses at level l are present in level $l+1$. By definition, $AMP(l)$ is the average duration of the miss portion of the miss memory accesses at level l, and $AMAT(l+1)$ is the average memory access time at level $l+1$. We have:

$$AMP(l) = AMAT(l+1),$$

where $1 \leqslant l \leqslant L$ and $AMAT(L+1)$ is the memory access time of the main memory (in addition to the hit time already spent in all L cache levels, to be precise). Given that sequential data access is a special case of concurrent data access, one can also arrive at the same conclusion using the memory access concurrency:

$$\begin{aligned} AMP(l) &= \boxed{\frac{1}{\alpha_m(l)} \sum_{i=1}^{n} c_i^{(m)}(l)}_{(21)} \\ &= \frac{1}{\boxed{\alpha(l+1)}_{(36)}} \sum_{i=1}^{n} \boxed{c_i(l+1)}_{(37)} \\ &= \boxed{AMAT(l+1)}_{(28)}. \end{aligned}$$

Therefore, we can derive the well-known AMAT recursive relation using our notations:

$$\begin{aligned} AMAT(l) &= \boxed{H(l) + \rho_m(l) \times AMP(l)}_{(29)} \\ &= H(l) + \rho_m(l) \times AMAT(l+1). \end{aligned} \tag{39}$$

6.3 C-$AMAT$ Recursion

We can show that $\kappa(l)$, which is the fraction of pure miss cycles among the pure miss and mixed hit/miss cycles at cache level l, can be expressed as follows:

$$\kappa(l) = \frac{\rho_M(l)}{\rho_m(l)} \times \frac{pAMP(l)}{AMP(l)} \times \frac{C_m(l)}{C_M(l)}. \tag{40}$$

Proof.

$$\begin{aligned} &\frac{\rho_M(l)}{\rho_m(l)} \times \frac{pAMP(l)}{AMP(l)} \times \frac{C_m(l)}{C_M(l)} \\ &= \frac{\boxed{\frac{\alpha_M(l)}{\alpha(l)}}_{(17)}}{\boxed{\frac{\alpha_m(l)}{\alpha(l)}}_{(8)}} \times \frac{\boxed{\frac{\sum_{i \in \mathbb{M}(\mathbb{l})} c_i^{(m)}(l)}{\alpha_M(l)}}_{(34)}}{\boxed{\frac{1}{\alpha_m(l)} \sum_{i=1}^{n} c_i^{(m)}(l)}_{(21)}} \times \\ &\quad \frac{\boxed{\frac{1}{m(l)+x(l)} \sum_{i=1}^{n} c_i^{(m)}(l)}_{(26)}}{\boxed{\frac{1}{m(l)} \sum_{i \in \mathbb{M}(\mathbb{l})} c_i^{(m)}(l)}_{(27)}} \\ &= \frac{m(l)}{m(l)+x(l)} = \boxed{\kappa(l)}_{(15)}. \end{aligned}$$

□

We can define C-$AMAT$ recursively as follows:

[③] When applying the formulas given in this study, as same as with other existing performance models, such as AMAT, one needs to pay special attention to the underlying hardware/software environment. For example, modern memory systems manage the data in blocks (cache lines) and use the Miss Status Holding Register (MSHR) [9] to handle concurrent cache misses. Advanced MSHR design differentiates between a primary and a secondary miss depending on whether there exists a pending miss on the same cache line. Only the primary miss causes a memory access request issued to the next level. A secondary miss does not. In this case, we need to adjust the cache misses in the calculation (by discounting the secondary misses).

$$C\text{-}AMAT(l) = \frac{H(l)}{C_H(l)} + \rho_m(l) \times \kappa(l) \times C\text{-}AMAT(l+1). \quad (41)$$

Proof.

$$\begin{aligned}
&\frac{H(l)}{C_H(l)} + \rho_m(l) \times \kappa(l) \times C\text{-}AMAT(l+1) \\
&= \frac{H(l)}{\boxed{\frac{\alpha(l) \times H(l)}{h(l)+x(l)}}_{(25)}} + \\
&\quad \boxed{\frac{\alpha_m(l)}{\alpha(l)}}_{(8)} \boxed{\frac{m(l)}{m(l)+x(l)}}_{(15)} \times \boxed{\frac{\omega(l+1)}{\alpha(l+1)}}_{(30)} \\
&= \frac{h(l)+x(l)}{\alpha(l)} + \\
&\quad \frac{\alpha_m(l) \times m(l) \times \boxed{(m(l)+x(l))}_{(38)}}{\alpha(l) \times (m(l)+x(l)) \times \boxed{\alpha_m(l)}_{(36)}} \\
&= \frac{h(l)+x(l)}{\alpha(l)} + \frac{m(l)}{\alpha(l)} \\
&= \frac{w(l)}{\alpha(l)} = \boxed{C\text{-}AMAT(l)}_{(30)}.
\end{aligned}$$

□

One can obtain several important insights from the above derivation. We have:

$$\begin{aligned}
&C\text{-}AMAT(l) \\
&= \boxed{\frac{H(l)}{C_H(l)}}_{\text{hit}} + \\
&\quad \boxed{\rho_m(l) \times \kappa(l) \times C\text{-}AMAT(l+1)}_{\substack{\text{pure}\\\text{miss}}} \\
&= \boxed{\frac{h(l)+x(l)}{\alpha(l)}}_{\text{hit}} + \boxed{\frac{m(l)}{\alpha(l)}}_{\substack{\text{pure}\\\text{miss}}} \\
&= \boxed{\frac{\omega(l)}{\alpha(l)} \times \frac{h(l)+x(l)}{\omega(l)}}_{\text{hit}} + \\
&\quad \boxed{\frac{\omega(l)}{\alpha(l)} \times \frac{m(l)}{\omega(l)}}_{\substack{\text{pure}\\\text{miss}}} \\
&= \frac{\omega(l)}{\alpha(l)} \left(\boxed{\frac{h(l)+x(l)}{\omega(l)}}_{\text{hit}} + \boxed{\frac{m(l)}{\omega(l)}}_{\substack{\text{pure}\\\text{miss}}} \right) \\
&= C\text{-}AMAT(l) \times \\
&\quad \left(\boxed{\phi(l)}^{(13)}_{\text{hit}} + \boxed{\mu(l) \times \kappa(l)}^{(16)}_{\substack{\text{pure}\\\text{miss}}} \right).
\end{aligned}$$

We see that $C\text{-}AMAT(l)$ consists of two portions (as marked in the above derivation): the hit portion, $\phi(l) = \frac{h(l)+x(l)}{\omega(l)}$, which is overlapped with hit cycles, and the pure miss portion, $1-\phi(l) = \frac{m(l)}{\omega(l)} = \mu(l) \times \kappa(l)$, which consists of only the pure miss cycles. Furthermore,

$$\begin{aligned}
&\rho_m(l) \times C\text{-}AMAT(l+1) \\
&= \rho_\mu(l) \times \boxed{\frac{\omega(l+1)}{\alpha(l+1)}}_{(30)} = \rho_m(l) \times \frac{\boxed{\mu(l) \times \omega(l)}_{(38)}}{\boxed{\rho_m(l) \times \alpha(l)}_{(36)}} \\
&= \mu(l) \times \frac{\omega(l)}{\alpha(l)} = \boxed{\mu(l) \times C\text{-}AMAT(l)}_{(30)}.
\end{aligned}$$

Therefore,

$$C\text{-}AMAT(l) = \frac{\rho_m(l)}{\mu(l)} \times C\text{-}AMAT(l+1).$$

From the above equation, one can derive a recursive definition of $C\text{-}AMAT(l)$ at the cache level l:

$$C\text{-}AMAT(l) = C\text{-}AMAT(1) \times \prod_{i=1}^{l-1} \frac{\mu(i)}{\rho_m(i)}. \quad (42)$$

The above equation shows that $C\text{-}AMAT$ at layer l of the memory hierarchy expands or shrinks from that of the previous layer $l-1$ at a rate commensurate with the ratio between μ, the ratio of miss cycles over memory active cycles, and ρ_m, the miss ratio. This is not surprising as $C\text{-}AMAT$ captures the combined effect of data access locality (from ρ_m) and concurrency (from μ).

7 Layered Performance Matching

The average access time described in Section 5 and Section 6 provides a time-based performance model. One can also examine memory system performance based on rates. The Layered Performance Matching (LPM) [7] is a technique for examining the memory performance at each level of a hierarchical memory system based on data flow analysis. LPM allows to transfer a global performance optimization problem for the entire memory system to local optimization problems at each layer of the memory hierarchy. In this section, we establish a recursive definition of the matching ratios. Like in $C\text{-}AMAT$, the recursion allows us to capture the contribution of memory performance at a cache layer in relation to the adjacent layers for the overall runtime performance.

Let $LPMR(l)$ be the matching ratio at cache level l. Let $\lambda(l)$ be the request rate at cache level l, and let $\nu(l)$

be the supply rate at cache level l. The matching ratio is actually the utilization at the corresponding cache level, and is defined to be the ratio between the request rate and the supply rate:

$$LPMR(l) = \frac{\lambda(l)}{\nu(l)}. \tag{43}$$

Fig.4 shows an example of the request and supply rates for a three-level cache memory system.

The number of memory accesses at each cache level can be defined recursively.

$$\alpha(l) = \begin{cases} IC \times f_{\text{mem}}, & \text{if } l = 1, \\ \alpha(l-1) \times \rho_m(l-1), & \text{if } 1 < l \leqslant L+1, \end{cases} \tag{44}$$

where L is the number of cache levels. The base case $l = 1$ is from (2); the recursion is from (36). We use $\alpha(L+1)$ to represent the main memory accesses. More succinctly, we have:

$$\alpha(l) = IC \times f_{\text{mem}} \times \prod_{i=1}^{l-1} \rho_m(i), \tag{45}$$

where $1 \leqslant l \leqslant L+1$.

The request rate is the number of memory accesses divided by the total number of CPU cycles for running the program without memory stall (i.e., the ideal run time of the program assuming $MST = 0$):

$$\lambda(l) = \frac{\alpha(l)}{IC \times CPI_{\text{exe}}} \tag{46}$$

$$= \begin{cases} IPC_{\text{exe}} \times f_{\text{mem}}, & \text{if } l = 1, \\ \rho_m(l-1) \times \lambda(l-1), & \text{if } 1 < l \leqslant L+1 \end{cases} \tag{47}$$

$$= IPC_{\text{exe}} \times f_{\text{mem}} \times \prod_{i=1}^{l-1} \rho_m(i), \tag{48}$$

where $1 \leqslant l \leqslant L+1$. We use $\lambda(L+1)$ to represent the request rate at the main memory.

The number of memory active cycles can be derived from (30):

$$\begin{aligned} \omega(1) &= \alpha(1) \times C\text{-}AMAT(1) \\ &= IC \times f_{\text{mem}} \times C\text{-}AMAT(1). \end{aligned}$$

Combining it with the recursion from (38), we can have a recursive definition:

$$\omega(l) = \begin{cases} IC \times f_{\text{mem}} \times C\text{-}AMAT(1), & \text{if } l = 1, \\ \mu(l-1) \times \omega(l-1), & \text{if } 1 < l \leqslant L+1, \end{cases} \tag{49}$$

where $1 \leqslant l \leqslant L+1$. We use $\omega(L+1)$ to represent the number of main memory active cycles. We can also use the succinct form:

$$\omega(l) = IC \times f_{\text{mem}} \times C\text{-}AMAT(1) \times \prod_{i=1}^{l-1} \mu(i). \tag{50}$$

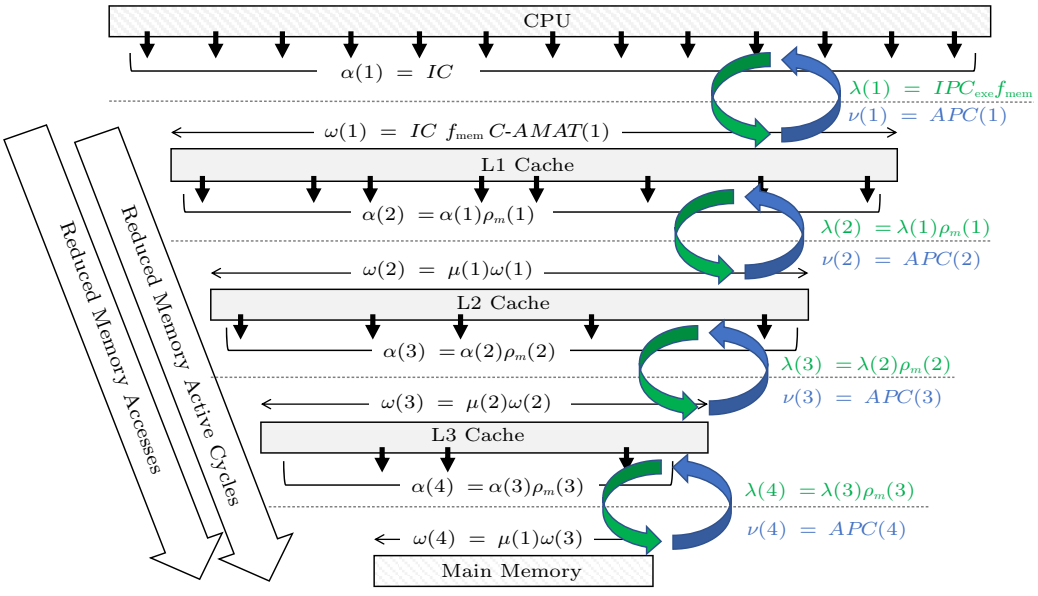


Fig.4. Request and supply rates at different cache levels. This example shows three cache levels: L1 (first-level cache), L2, and L3 (last-level cache). At each cache level l, we show the request rate, $\lambda(l)$, and the supply rate, $\nu(l)$, as well as the number of memory accesses (requests), $\alpha(l)$, and the number of memory active cycles, $\omega(l)$. As a special case, $\lambda(4)$, $\nu(4)$, $\alpha(4)$ and $\omega(4)$ are the request rate, the supply rate, the number of memory accesses, and the number of memory active cycles at the main memory, respectively.

Given that there is a one-to-one correspondence between request and supply at each memory layer, the supply rate can be calculated as the number of memory accesses divided by the number of memory active cycles:

$$\nu(l) = \frac{\alpha(l)}{\omega(l)} = APC(l) = C\text{-}AMAT(l)^{-1}. \tag{51}$$

The above equation is derived directly from (30) and (32). Note that the request rate is calculated over the ideal run time of the program without memory stall, and the supply rate is defined over the memory active cycles at a specific memory layer.

Finally, the matching ratio follows:

$$\begin{aligned} & LPMR(l) \\ &= \frac{\lambda(l)}{\nu(l)} = \frac{\alpha(l)/(IC \times CPI_{\text{exe}})}{\alpha(l)/\omega(l)} = \frac{\omega(l)}{IC \times CPI_{\text{exe}}} \\ &= \begin{cases} IPC_{\text{exe}} \times f_{\text{mem}} \times C\text{-}AMAT(1), & \text{if } l = 1, \\ \mu(l-1) \times LPMR(l-1), & \text{if } l > 1 \end{cases} \qquad (52) \\ &= IPC_{\text{exe}} \times f_{\text{mem}} \times C\text{-}AMAT(1) \times \prod_{i=1}^{l-1} \mu(i). \qquad (53) \end{aligned}$$

The following derivation establishes the relationship between the matching ratio, $LPMR(l)$, and the ratio of memory access time over compute time, Δ:

$$\begin{aligned} & LPMR(l) \\ &= IPC_{\text{exe}} \times f_{\text{mem}} \times C\text{-}AMAT(1) \times \prod_{i=1}^{l-1} \mu(i) \\ &= IPC_{\text{exe}} \times f_{\text{mem}} \times \boxed{\frac{MST}{\mu(1) \times \kappa(1)}}_{(35)} \times \prod_{i=1}^{l-1} \mu(i) \qquad (54) \\ &= \frac{IPC_{\text{exe}} \times \boxed{CPI_{\text{exe}} \times \Delta}_{(5)}}{\mu(1) \times \kappa(1)} \times \prod_{i=1}^{l-1} \mu(i) \\ &= \frac{\Delta}{\mu(1) \times \kappa(1)} \times \prod_{i=1}^{l-1} \mu(i). \qquad (55) \end{aligned}$$

The last step comes from (3). More specifically, for $l = 1$, 2, and 3, we have:

$$LPMR(1) = \frac{\Delta}{\mu(1) \times \kappa(1)}, \tag{56}$$

$$LPMR(2) = \frac{\Delta}{\kappa(1)}, \tag{57}$$

$$LPMR(3) = \frac{\mu(2) \times \Delta}{\kappa(1)}. \tag{58}$$

We know that Δ is related to the memory system efficiency (see (6)). A smaller $LPMR(l)$ indicates higher memory performance at cache level l. In fact, the LPMR value can be directly related to the memory stall time (MST). From (54), we have:

$$\begin{aligned} & MST \\ &= \frac{\mu(1) \times \kappa(1)}{IPC_{\text{exe}} \times f_{\text{mem}} \times \prod_{i=1}^{l-1} \mu(i)} \times LPMR(l). \qquad (59) \end{aligned}$$

The LPM method allows one to optimize memory performance by tuning the matching ratio at each memory layer to meet the target requirement [7]. The equations above provide a formal proof of the LPM method. In particular, if we want Δ to be no larger than a given threshold $x\%$ as the memory system optimization target, based on (56), we only need to have $LPMR(1) \leqslant \frac{x\%}{\mu(1)\kappa(1)}$. In general, as it follows from (55) and (59), if $LPMR(l) \leqslant \frac{x\%}{\mu(1) \times \kappa(1)} \times \prod_{i=1}^{l-1} \mu(i)$, we also have $MST \leqslant x\% \times \frac{CPI_{\text{exe}}}{f_{\text{mem}}}$.

Recall from (4), an instruction's execution time consists of two parts: CPI_{exe}, which is the ideal compute time in cycles per instruction without memory stall, and $(f_{\text{mem}} \times MST)$, which is the memory stall time per instruction. (Note that MST is the memory stall time per memory reference.) Therefore, $x\%$ is actually the relative memory stall time with respect to the ideal compute time in cycles per instruction.

The performance of the L1 cache is the overall memory system performance. In theory, we only need to match $LPMR(1)$ to optimize the overall system performance. However, L1's performance depends on L2. The value of $LPMR(1)$ can increase due to $LPMR(2)$ if L2's performance does not meet its match. The same argument can be applied to other memory layers. Therefore, memory performance optimization can be viewed as a recursive process in which one can tune the matching ratio at each memory layer to meet the target requirement.

For example, if the measurement shows that the matching ratio at the L1 cache is below the target value ((56)), it means that the memory system performance already meets the efficiency requirement. Otherwise, one needs to check the matching ratio at the L2 cache ((57)). If L2's matching ratio is below the target value, it means that L2 has sufficiently supplied data to L1, but L1 does not meet the requirement. Thus, one should focus only on the memory optimization at L1.

On the other hand, if L2's matching ratio is also above the target, one should continue to check the matching ratio at the L3 cache ((58)). If L3's matching ratio is below the target, one should focus on L1 and L2, but not L3. Otherwise, all three cache levels need to be optimized.

The above method shows a systematic and fast approach to matching the data access demands of a running application to the underlying memory architecture. The matching can be achieved by adjusting the configuration of the underlying memory architecture or through the scheduling of applications appropriately in a heterogeneous environment. Using the C-AMAT formulation described above, one can assess the data access locality and concurrency of an application in accordance with the underlying memory architecture. Layered matching leads to the improved measurement and analysis of data access delays, and as such, results in more effective and efficient methods for obtaining a balanced memory architecture design and optimization.

8 Discussions

Our unified memory performance model establishes a coherent and enhanced mathematical foundation for several recently proposed memory performance models, including C-AMAT, APC, and LPM. In this section, we discuss the practical implications of the proposed model.

Two of the key issues of memory system modeling are: 1) How can the theoretical results be used in actual architecture design? 2) How can the results be used by researchers in the field? That is, how to benefit the general architecture and system research community? Applying the memory performance models in practice not only requires an in-depth understanding of the models, the underlying hardware, and the target applications, but also in many times needs to tailor the models for the specific hardware configuration, the target application, or both.

One example is the Disaggregated Memory System (DMS). The DMS architecture has been recently introduced to tackle the memory capacity scaling problem of high-performance computing systems, by adding a remote global memory layer to expand the memory capacity of individual compute nodes [10]. DMS usually consists of a large pool of memory, a memory controller, and a network interface for the compute nodes to communicate with the disaggregated memory over a low-latency high-bandwidth interconnection network [11]. Earlier, we performed a preliminary measurement study using the C-AMAT analysis for a specific DMS machine, named Cooley, at the Argonne Leadership Computing Facility (ALCF) [12]. The memory-centric view of the unified model can be applied in this case to study the contribution of individual memory devices in the over-all system performance. We note that DMS not only adds an additional level into the existing memory hierarchy, but also complicates the architectural design: data accesses may split among multiple devices in the system and merge onto a particular disaggregated memory as it is accessible from different cores, processors, and machines; data accesses also need to travel through the interconnection network with substantial delay variations and interference. The unified model needs to be extended to handle these situations. As another example, graphics processing units (GPUs) have been widely used as accelerators for general purpose computing. However, the benefits brought by GPUs vary substantially among scientific applications, which mainly can be attributed to the variation of data access delay. We have previously extended the C-AMAT model to study the memory performance on GPUs [13]. In particular, warp-level data accesses are considered in aggregation for data locality and concurrency in C-AMAT. A limitation of the approach is that the GPU memory is considered as stand-alone and separate from the host memory. The proposed unified model can be applied in this situation where we can study individual contributions of the memory system components in the overall performance (with considerations of merging, splitting, and off-chip latencies).

The memory-centric view offered by the unified performance model provided in this study is essential in memory system simulator design. Modeling and simulation has been proven valuable for studying complex system behaviors, especially when the system under study involves a large number of variables in the design space, as in the case of memory systems. The task of memory architecture optimization must be evaluated with diverse system configurations and algorithmic choices that impact host systems, memory/storage devices, interconnection networks, and applications/workloads. Our cycle-level analyses in the unified memory model (regarding data locality, concurrency, latency, as well as their recursive relationships) can be used to reveal potential performance issues of complex memory architectures, including diverse system configurations and memory hierarchy, various memory devices (including both tra-

ditional and new technologies, including non-volatile memory), cache management policies (such as multi-banked, pipelined, and non-blocking caches), and interconnection networks (such as disaggregated memory). The model can enable the design of accurate memory system simulation, and in doing so will be instrumental to generating insight to architecture design, guiding design decisions, and evaluating performance optimizations. Further study on this issue is warranted.

9 Related Work

In 1990, Sun and Ni [14] proposed a memory-bounded parallel speedup model, also known as the Sun-Ni's Law. The memory-bounded parallel speedup model identifies that data access is the key factor influencing performance, and scalable computing is bounded by the memory capacity. In 1994, the term "memory wall" was formally introduced by Wulf and McKee, based on the Average Memory Access Time (AMAT) model [1]. For two and half decades, intensive research has been conducted to improve memory system performance. However, the "memory wall" problem persisted. Today, modern microprocessors spend a vast majority of their transistors for on-chip caches, rather than devote them for computing.

There are three types of memory performance models. The first type of models deals with data access locality. The study of locality started from Denning's working set theory in 1968 [2, 3]. Mattson *et al.* proposed the stack algorithm to calculate reuse distance in 1970 [15]. Several metrics have been developed based on the concept of working set and reuse distance. For example, Weinberg *et al.* [16] presented weighted stride for HPC applications. Berg and Hagersten [17] and Gu *et al.* [18] proposed to quantify locality based on measuring the change of miss rates or reuse distances. Anghel *et al.* [19] proposed to use a probability distribution of reuse distance to quantify the locality. These metrics are based on heuristics and lack formal mathematical definition. Ding *et al.* formally established the relationship among the five locality metrics: footprint, inter-miss time, volume fill time, miss ratio, and reuse distance [20, 21]. Jiang *et al.* [22] proposed "concurrent reuse distance" for multi-threaded programs. Gupta *et al.* [23] quantified the data access locality as a conditional probability. Liu and Sun [24] proposed a concurrency-aware data access locality metric, which can accurately reflect the combined impact of data access locality and concurrency in modern memory architectures.

The second type of models deals with cache performance based on hit/miss rate and timing. Miss rate (MR), miss per kilo instructions (MPKI), average miss penalty (AMP), and average memory access time (AMAT) are commonly used performance metrics [8]. MR is defined by the number of misses over the total number of memory accesses. Similarly, MPKI is defined as the number of misses in thousands divided by the number of total committed instructions. Both MR and MPKI reflect the proportion of the data in or out of the cache, but they do not reflect the penalty of the misses. AMP is the average latency for the cache miss; it is the sum of all single miss latency divided by the total number of misses. AMP catches the penalty of the cache miss access, but does not tie it up with the associated performance degradation of the program execution. The AMAT model considers the layered performance; it is recursive as it can be used in each memory layer to reflect the portion of latency consumed at each memory layer. AMAT is thus a more comprehensive memory metric, but it is based on the single data access viewpoint. In particular, AMAT does not consider memory concurrency upon overlapping memory hits and misses. In contrast, the C-AMAT model [5] considers locality, concurrency, and overlapping and can be recursively applied across the memory hierarchy for performance optimization.

The third type of models deals with cache performance based on data flows across the memory hierarchy. Memory Level Parallelism (MLP) has been proposed in recent years as a common memory metric [4, 25, 26]. MLP is the average number of outstanding long-latency main-memory accesses for each active memory access cycle. However, it only focuses on data concurrency, but not locality and overlapping. Amdahl's Balanced System Law [27], as a rule of thumb for the system design, says that a system would need one bit per second for I/O for each instruction per second, which basically matches the ratio of memory access speed to the computing speed. The roofline model by Williams *et al.* [28] is a more attainable system performance model that compares peak computing speed with the product of peak memory bandwidth and operational intensity. Although the specific values can be obtained through measurements, the roofline model is not intended to be accurate, but to provide insight to system performance limitations and bottlenecks. Zhu *et al.* [29] proposed a balanced design for supercomputers. The main idea of the balanced design is to provide the maximum bandwidth with the maximum

number of compute nodes that can concurrently access I/O systems. The Layer Performance Matching (LPM) method [7] essentially follows the data flow analysis. It is developed based on C-AMAT for memory system optimization.

Our study reexamines the APC, C-AMAT, and LPM models under a coherent mathematical framework using a new memory-centric view, where we classify the memory cycles at each memory layer into four distinct types and use them to define memory access concurrency and access time. Using the memory-centric approach, we are able to provide new insights on the performance of modern hierarchical memory systems. With the new proofs we build a memory system modeling framework for us to develop a cycle-accurate memory system simulation model.

10 Conclusions

This paper describes a CPU memory performance model and establishes a mathematical foundation combining three existing models (C-AMAT, APC, and LPM) in a coherent mathematical framework. In particular, new derivations were provided using a memory-centric approach, which can help achieve better understanding and gain new insights to the performance of memory systems. The resulting hierarchical memory performance model provides us with a better understanding of the memory system performance and paves the way for developing next-generation memory system simulators and measurement tools, and thereby aides the design and development of more efficient memory architectures for modern computer systems.

For future work, we plan to extend the concurrency-based hierarchical memory performance framework to analyze more complex scenarios in modern computer architectures using the memory-centric approach. For example, data access merging may occur in shared-memory scenarios such as those in a multi-processor multi-core environment. Split data access may occur in a memory system where multiple heterogeneous memory devices (such as DRAM, NVM, and disaggregated memory) coexist at the same memory layer. The memory performance model must be able to handle situations where both merging and splitting memory accesses may occur. We also plan to develop a cycle-based memory simulator based on this memory modeling study.

Acknowledgments The authors would like to thank the reviewers for their constructive comments and suggestions.

References

[1] Wulf W A, McKee S A. Hitting the memory wall: Implications of the obvious. *ACM SIGARCH Computer Architecture News*, 1995, 23(1): 20-24. DOI: 10.1145/216585.216588.

[2] Denning P J. The working set model for program behavior. In *Proc. the 1st ACM Symposium on Operating System Principles*, October 1967, Article No. 15. DOI: 10.1145/357980.357997.

[3] Denning P J. The locality principle. In *Communication Networks and Computer Systems: A Tribute to Professor Erol Gelenbe*, Barria G A (ed.), London, Imperial College Press, 2006, pp.43-67.

[4] Chou Y, Fahs B, Abraham S G. Microarchitecture optimizations for exploiting memory-level parallelism. In *Proc. the 31st Annual International Symposium on Computer Architecture*, June 2004, pp.76-87. DOI: 10.1109/ISCA.2004.1310765.

[5] Sun X H, Wang D W. Concurrent average memory access time. *Computer*, 2014, 47(5): 74-80. DOI: 10.1109/MC.2013.227.

[6] Wang D W, Sun X H. APC: A novel memory metric and measurement methodology for modern memory systems. *IEEE Transactions on Computers*, 2014, 63(7): 1626-1639. DOI: 10.1109/TC.2013.38.

[7] Liu Y, Sun X. LPM: A systematic methodology for concurrent data access pattern optimization from a matching perspective. *IEEE Transactions on Parallel and Distributed Systems*, 2019, 30(11): 2478-2493. DOI: 10.1109/TPDS.2019.2912573.

[8] Hennessy J L, Patterson D A. Computer Architecture: A Quantitative Approach (5th edition). Morgan Kaufmann, 2011.

[9] Tuck J, Ceze L, Torrellas J. Scalable cache miss handling for high memory-level parallelism. In *Proc. the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, December 2006, pp.409-422. DOI: 10.1109/MICRO.2006.44.

[10] Lim K, Turner Y, Santos J R, AuYoung A, Chang J, Ranganathan P, Wenisch T F. System-level implications of disaggregated memory. In *Proc. the 2012 IEEE International Symposium on High-Performance Comp Architecture*, Feb. 2012, pp.189-200. DOI: 10.1109/HPCA.2012.6168955.

[11] Gao P X, Narayan A, Karandikar S, Carreira J, Han S, Agarwal R, Ratnasamy S, Shenker S. Network requirements for resource disaggregation. In *Proc. the 12th USENIX Symposium on Operating Systems Design and Implementation*, Nov. 2016, pp.249-264. DOI: 10.5555/3026877.3026897.

[12] Zhang N, Toonen B, Sun X H, Allcock B. Performance modeling and evaluation of a production disaggregated memory system. In *Proc. the 2020 International Symposium on Memory Systems*, Sept. 28–Oct. 2. 2020.

[13] Zhang N, Jiang C T, Sun X H, Song S. Evaluating GPGPU memory performance through the C-AMAT model. In *Proc. the Workshop on Memory Centric Programming for HPC*, Nov. 2017, pp.35-39. DOI: 10.1145/3145617.3158214.

[14] Sun X H, Ni L M. Another view on parallel speedup. In *Proc. the 1990 ACM/IEEE Conference on Supercomputing*, November 1990, pp.324-333. DOI: 10.1109/SUPERC.1990.130037.

[15] Mattson R L, Gecsei J, Slutz D R, Traiger I L. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 1970, 9(2): 78-117. DOI: 10.1147/sj.92.0078.

[16] Weinberg J, McCracken M O, Strohmaier E, Snavely A. Quantifying locality in the memory access patterns of HPC applications. In *Proc. the 2005 ACM/IEEE Conference on Supercomputing*, November 2005, Article No. 50. DOI: 10.1109/SC.2005.59.

[17] Berg E, Hagersten E. Fast data-locality profiling of native execution. In *Proc. the International Conference on Measurements and Modeling of Computer Systems*, June 2005, pp.169-180. DOI: 10.1145/1071690.1064232.

[18] Gu X M, Christopher I, Bai T X, Zhang C L, Ding C. A component model of spatial locality. In *Proc. the 8th International Symposium on Memory Management*, June 2009, pp.99-108. DOI: 10.1145/1542431.1542446.

[19] Anghel A, Dittmann G, Jongerius R, Luijten R. Spatio-temporal locality characterization. In *Proc. the 1st Workshop on Near Data Processing*, December 2013.

[20] Ding C, Xiang X Y. A higher order theory of locality. In *Proc. the 2012 ACM SIGPLAN Workshop on Memory System Performance Correctness*, June 2012, pp.68-69. DOI: 10.1145/2247684.2247697.

[21] Ding C, Zhong Y T. Predicting whole-program locality through reuse distance analysis. In *Proc. the 2003 ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2003, pp.245-257. DOI: 10.1145/781131.781159.

[22] Jiang Y L, Zhang E Z, Tian K, Shen X P. Is reuse distance applicable to data locality analysis on chip multiprocessors? In *Proc. the 19th International Conference on Compiler Construction*, March 2010, pp.264-282. DOI: 10.1007/978-3-642-11970-5_15.

[23] Gupta S, Xiang P, Yang Y, Zhou H Y. Locality principle revisited: A probability-based quantitative approach. *Journal of Parallel and Distributed Computing*, 2013, 73(7): 1011-1027. DOI: 10.1016/j.jpdc.2013.01.010.

[24] Liu Y H, Sun X H. CaL: Extending data locality to consider concurrency for performance optimization. *IEEE Transactions on Big Data*, 2017, 4(2): 273-288. DOI: 10.1109/TBDATA.2017.2753825.

[25] Glew A. MLP yes! ILP no. In *Proc. the ASPLOS Wild and Crazy Idea Session*, October 1998.

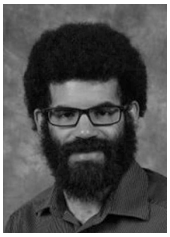
[26] Sorin D J, Pai V S, Adve S, Vernon M K, Wood D A. Analytic evaluation of shared-memory systems with ILP processors. In *Proc. the 25th Annual International Symposium on Computer Architecture*, June 1998, pp.380-391. DOI: 10.1109/ISCA.1998.694797.

[27] Gray J, Shenoy P. Rules of thumb in data engineering. In *Proc. the 16th International Conference on Data Engineering*, March 2000, pp.3-10. DOI: 10.1109/ICDE.2000.839382.

[28] Williams S, Waterman A, Patterson D. Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM*, 2009, 52(4): 65-76. DOI: 10.1145/1498765.1498785.

[29] Zhu M F, Xiao L M, Ruan L, Hao Q F. DeepComp: Towards a balanced system design for high performance computer systems. *Front. Comput. Sci. China*, 2010, 4(4): 475-479. DOI: 10.1007/s11704-010-0150-z.

Jason Liu is a University Eminent Scholar Chaired Professor at the School of Computing and Information Sciences, Florida International University (FIU) in Miami, Florida, USA. His research focuses on modeling and simulation, parallel discrete-event simulation, performance modeling and simulation of computer systems and computer networks. He currently serves on the Editorial Board of ACM Transactions on Modeling and Computer Simulation (TOMACS), SIMULATION, Transactions of the Society for Modeling and Simulation International, and IEEE Networking Letters. He is also on the Steering Committee of ACM SIGSIM Conference on Principles of Advanced Discrete Simulation (SIGSIM-PADS). Jason Liu is an NSF CAREER awardee in 2006 and an ACM Distinguished Scientist in 2014.



Pedro Espina is a Ph.D. student at the School of Computing and Information Sciences, Florida International University (FIU) in Miami, Florida, USA. His degrees include B.S. degrees in physics and mathmatics, and M.S. degree in mathematics. His research focuses on modeling and simulation of computer systems.

Xian-He Sun is a University Distinguished Professor and the Ron Hochsprung Endowed Chair of Computer Science at the Illinois Institute of Technology (IIT), Chicago, USA, and the director of the Scalable Computing Software Laboratory at IIT. Before joining IIT, he worked at DoE Ames National Laboratory, at ICASE, NASA Langley Research Center, at Louisiana State University, Baton Rouge, and was an ASEE Fellow at Navy Research Laboratories. Dr. Sun is an IEEE Fellow and is known for his memory-bounded speedup model, also called Sun-Ni's Law, for scalable computing. His research interests include data-intensive high-performance computing, memory and I/O systems, software system for big data applications, and performance evaluation and optimization. He has over 300 publications and six patents in these areas. He is the associate chief editor of IEEE Transactions on Parallel and Distributed Systems, a Golden Core member of the IEEE CS Society, a former chair of the Computer Science Department at IIT, and is serving and served on the editorial board of leading professional journals in the field of parallel processing.