LPM: A Systematic Methodology for Concurrent Data Access Pattern Optimization from a Matching Perspective

Yuhang Liu, *Member, IEEE* and Xian-He Sun, *Fellow, IEEE*

Abstract—As applications become increasingly data intensive, conventional computing systems become increasingly inefficient due to data access performance bottlenecks. While intensive efforts have been made in developing new memory technologies and in designing special purpose machines, there is a lack of solutions for evaluating and utilizing recent hardware advancements to address the memory-wall problem in a systematic way. In this study, we present the memory Layered Performance Matching (LPM) methodology to provide a systematic approach for data access performance optimization. LPM uniquely presents and utilizes the data access concurrency, in addition to data access locality, in a memory hierarchical system. The LPM methodology consists of models and algorithms, and is supported with a series of analytic results for its correctness. The rationale of LPM is to reduce the overall data access delay through the matching of data request rate and data supply rate at each layer of a memory hierarchy, with a balanced consideration of data locality, data concurrency, and latency hiding of data flow. Extensive experimentations on both physical platforms and software simulators confirm our theoretical findings, and they show that the LPM approach can be applied in diverse computing platforms and can effectively guide performance optimization of memory systems.

Index Terms—Memory wall, memory stall time, efficiency, performance optimization, layered performance matching (LPM), memory concurrency

1 Introduction

The memory wall problem is a long-standing issue facing the computing community.[1] It is on the top of the 10 ways to waste time and efficiency of parallel computers [7]. The term memory wall refers to the growing disparity of speed between CPU and memory outside the CPU chip [48]. During the last two decades, processors have advanced from in-order to out-of-order and from uni-core to multi-core; memory systems have advanced with many concurrency-oriented features such as multi-port, multi-banked, pipelined and non-blocking cache. Although these resources for concurrence are abundant, they are usually not well utilized. Meanwhile, memory-intensive applications have become increasingly common in diverse fields such as bioinformatics, computer aided designs, and complex social media interactions [15]. Compared with twenty years ago, the landscape of computing has changed, but the notorious memory wall still exists and "thicker". Memory stall time is the time CPU stalled waiting for data. Memory stall time often contribute 50 to 70 percent of the total execution time for many applications [20], [21], [25], [33]. Memory systems have become the most prominent performance bottleneck of computing systems.

Modern memory systems are equipped with many innovations developed over the years, where improving locality and concurrency are the two basic approaches. Hierarchical memory is a standard design of modern computers to ease the memory wall problem. The idea behind memory hierarchy is to explore locality. That is, data previously accessed may be used again soon (temporal locality), and data near the previously accessed data are likely to be used next (spatial locality). The locality of hierarchical memory systems has been well studied [17], [18]. However, a modern memory system is not only supported by memory hierarchy but also by various data access concurrency [16]. The overall performance of a memory system is a combined effort of memory hierarchy and concurrency, where the effectiveness of concurrency also influence latency hiding. Understanding the combined impact of locality and concurrency of memory accesses is a necessity to fully utilize the potential of a modern memory system.

In this study, we use two newly proposed performance models, Concurrent Average Memory Access Time (C-AMAT) [37] and Accesses Per memory active Cycle (APC) [36], [46] to capture the combined impact of locality and concurrency. The APC model reflects the quality of service (QoS) of modern memory systems. That is, due to concurrency, the

1. This paper is an extensively extended version of the authors' ICPP paper [31], with addition on formal presentation and evaluation of the time model, QoS goal, analytical definition, minimum requirement, measurement method, new experiments and open issues of LPM.

- *Y. Liu is with the State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, No.6 Kexueyuan South Road Zhongguancun, Haidian District, Beijing 100190, China. E-mail: liuyuhang@ict.ac.cn.*
- *X.-H. Sun is with the Department of Computer Science, Illinois Institute of Technology, Chicago, IL 60616. E-mail: sun@iit.edu.*

Manuscript received 24 Nov. 2018; revised 9 Apr. 2019; accepted 10 Apr. 2019. Date of publication 23 Apr. 2019; date of current version 11 Oct. 2019. (Corresponding author: Yuhang Liu.)
Recommended for acceptance by W. Yu.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2019.2912573

ultimate measurement is no longer the cache hit latency or cache hit rate, but how many accesses are finished in each memory active cycle. The C-AMAT model provides an analytical tool to understand the APC measurement. C-AMAT is an extension of the conventional AMAT [48] model by including memory concurrency into average memory access time. It can be expressed recursively to identify the bottleneck of a memory hierarchy and will be used as the major performance measurement in this study.

Designing a layered performance matching memory system is the goal of many computer architects [2], [3], [9], [10], [49]. While the concept of memory Layered Performance Matching (LPM) was presented in 2015 [31], the potential and practical implementation of the LPM method were not fully understood.

In this study, from a request-supply perspective, a quantatitive definition of LPM is first presented, based on which the necessity and feasibility of LPM are formally studied. The necessity is on the required matching degree of supply over request, while the feasibility concerns how to achieve the required matching degree.

One major contribution of concurrent data access for reducing memory stall time is latency-hiding. Specifically, latency-hiding can be achieved via overlapping, which includes the overlapping of hit versus miss and miss versus miss in concurrent data accesses. When upper layer can hide more latency, the matching requirement for lower layer is reduced. Many latency-hiding possibilities with diverse technologies are available in modern computing systems. In our study, we explicitly quantify the effectiveness of latency-hiding due to data access concurrency, and then we present the requirement of LPM under different degrees of latency-hiding.

Our study of LPM is based on the following four observations. First, each time a word is requested, the word is loaded in by a cache block (line) [30]. Therefore, if the cache block is reused multiple times, one physical data movement can support multiple data requests. Consequently, the supply rate can be increased with the improvement of locality.

Second, memory concurrency can mask data access delay. The hit-hit (all the data accesses are cache hits, and is termed as pure hits), hit-miss (or miss-hit, that is, at least a cache hit exists with cache misses), and miss-miss (all the data accesses are cache misses, and is termed as pure misses) [37] operations all have overlapping, thus all can mask data access delay. The hit-hit scenario increases data access bandwidth. In the hit-miss scenario, the cache hit provide data for processor to keep the pipeline go ahead and therefore masks the penalty of the cache miss. A miss-miss overlapping also will reduce data access delay, since two overlapping misses partly or completely share their delay.

Third, programs often have periodic behaviors, and their data access patterns often are predictable [32]. With a set of lightweight performance counters, we can deploy adaptive optimization techniques to meet the data requirements of an application.

Fourth, application-specific integrated circuit, reconfigurable hardware [47] and heterogeneous memory systems [15] have become prevalent and continue progress. These technologies provide the means to achieve LPM and, in the meantime, can utilize LPM to realize their full potential in reducing memory stall time.

The LPM methodology proposed in this study facilitates both hardware and algorithms to simultaneously consider data locality and concurrency, provides a systematic and automatic way to match data request to supply, and can reduce the memory stall time to achieve user-defined performance targets effectively.

We have made the following contributions in this study:

(1) Four theorems (Theorems 1, 2, 3 and 4) are derived to quantify the latency-hiding effect in modern memory system. With the quantifications, data access latency-hiding can be accurately measured. We have measured that up to 85.1 percent memory access latency can be hidden on commercial processors by cache hits, showing the importance of data access concurrence in reducing memory stall time.

(2) Three theorems (Theorems 5, 6 and 7) are derived to present the performance models and the associated measurements of LPM. With the theoretical results, we can understand what is an LPM-oriented design, what factors impact LPM, and how to optimize.

(3) Two theorems (Theorems 8 and 9) are derived to formally present the requirements of LPM. Knowing the requirements will facilitate fast optimization and avoid useless exploration and testing.

(4) The parameters of LPM can be measured online via performance counters of commercial processors. Compared to miss ratio (MR), LPM is more effective for performance tuning.

(5) Three case studies on either a physical platform or a simulator are conducted. Experiment results demonstrate that the LPM approach can be applied through code modification, reconfiguring system hardware, or scheduling on a heterogeneous platform. With only a few lines of code modification, the performance of the mcf benchmark that is a well-known cache buster has been improved by 13.8 percent. With a reconfigurable hardware, the LPM algorithm finds a perfect match for the $bewaves$ benchmark in five steps rather than exhaust searching. With LPM, a semi-optimal scheduling is achieved in polynomial time in a heterogeneous environment, which is impossible without LPM. These case studies confirm the practical value of the proposed theoretical results.

The remainder of this paper is organized as follows. Section 2 introduces the backgrounds. With two theorems, Section 3 proves that the time model is recursive and quantifies the latency-hiding effect of modern processors. Section 4 formalizes the goal of LPM. Using three theorems, Section 5 presents the definition and quantification of LPM. Section 6 proposes the requirement of LPM, for which two theorems are presented. Section 7 presents the algorithm of LPM. Section 8 presents a case study on commercial processors and two case studies on simulators. Section 9 reviews related works. Finally, Section 10 concludes this study and discusses potential future work.

2 Backgrounds

Memory systems can be characterized from space or time perspectives. From the space perspective, hierarchical structures

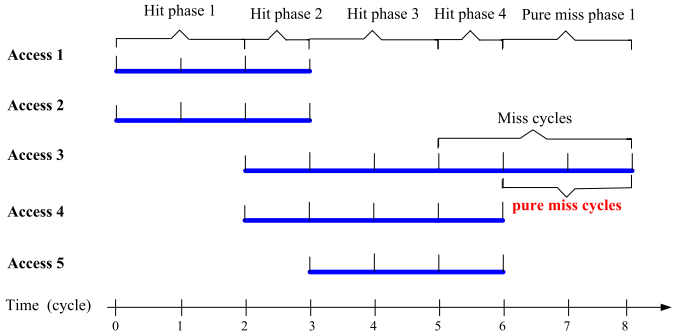

Fig. 1. Example of concurrent data accesses and their overlapping detail.

are developed to explore the data access locality. From the time perspective, in each clock cycle, there could be many accesses being issued or returned in each layer of a memory system, and these concurrent data accesses can increase data access bandwidth and hide data access delay. Therefore, memory stall time is the combined result of locality and concurrency.

The performance optimization of modern memory system needs a comprehensive and accurate model integrating the effects of both locality and concurrency. Note that, sixteen individual optimization methods are summarized in [22]. For instances, loop interchange and code vectorization can increase data access locality; non-blocking and multibank cache can increase data access concurrency. These techniques are useful on different occasions. However, these methods are entangled and sometimes even conflict with each other. How to use these techniques effectively to reach a global optimization state is still elusive.

There are four important measurements related to memory performance, the longest access time, the Average Memory Access Time (AMAT), the Concurrent Average Memory Access Time, and the memory stall time. Fig. 1 shows five individual memory accesses, where the longest one takes six cycles. However, not all the six cycles are memory stall time. Let n be the total number of memory layers. The conventional AMAT formulation of the ith (i = 1, 2, ..., n-1) memory layer is shown in Eq. (1) [48], where H is the hit time of memory accesses, MR is the miss rate and AMP is the average miss penalty. AMP is the sum of all miss access latencies divided by the total number of misses.

$$AMAT_i = H_i + MR_i \times AMP_i. \tag{1}$$

The average miss penalty of one layer is the average access time of its lower layer, that is, Eq. (2) holds.

$$AMP_i = AMAT_{i+1}. \tag{2}$$

Eqs. (3) can be derived by Eqs. (1) and (2), showing that AMAT is recursive.

$$AMAT_i = H_i + MR_i \times AMAT_{i+1}. \tag{3}$$

In Fig. 1, each access contains three cycles for cache hit operations. If it is a cache miss, additional miss penalty cycles will occur, depending where the miss ends. Access 1, 2, and 5 are cache hits; Access 3 and 4 are cache misses. Access 3 has a 3-cycle miss penalty; Access 4 has only a 1-cycle miss penalty. By Eq. (1), AMAT is 3 + 0.4 × 2 or 3.8 cycles per access. Compared to the longest access time, AMAT is 36.6 percent smaller.

TABLE 1
Symbol Abbreviations

Notation	*Meaning*
MST	Memory Stall Time
MSE	Memory System Efficiency
AMAT	Average Memory Access Time
C-AMAT	Concurrent AMAT
APC	Accesses Per (memory active) Cycle
H	Hit time
MR	conventional Miss Rate
pMR	pure Miss Rate
AMP	conventional Average Miss Penalty
pAMP	pure Average Miss Penalty
C_H	Hit Concurrency
C_M	pure Miss Concurrency
C_m	conventional miss Concurrency
μ_1	miss cycle proportion
κ_1	the impact indicator of pure cache misses
f_{mem}	memory access frequency
CPU time	the total application running time
IC	Instruction Count
cycle-time	the length of each clock cycle

Although locality has been considered, AMAT does not consider the concurrency of memory accesses. To cover the concurrent access properties of modern memory systems, the analytical expression of C-AMAT model is shown in Eq. (4) [37]. The first parameter H is the same as that in AMAT; the second parameter C_H represents hit concurrency; the third parameter C_M represents the pure miss concurrency while the conventional miss concurrency is referred to as C_m. C_H can be provided by caches with multi-port, multi-bank or/and pipelined structures. C_M can be provided by non-blocking cache structures. In addition, out-of-order execution, multi-issue pipeline, multithreading, chip multiprocessor (CMP), can all increase C_H and C_M. The pure miss rate pMR is the number of pure misses over the total number of accesses, which is different from the conventional miss rate. A pure miss here means that a miss contains at least one miss cycle that does not have any hit access activity [37]. pAMP is the average number of pure miss cycles per pure miss access. The notation abbreviations are summarized in Table 1.

$$C\text{-}AMAT_i = \frac{H_i}{C_{H_i}} + pMR_i \times \frac{pAMP_i}{C_{M_i}} \ (i = 1, 2, .., n.). \tag{4}$$

In Fig. 1, when considering the access concurrency, only Access 3 contains two pure miss cycles. Though Access 4 has one miss cycle, this cycle is not a pure miss cycle because it overlaps with the hit cycles of Access 5. Therefore according to the definition of concurrent pure miss rate, the (pure) miss rate of the five accesses is 0.2, instead of 0.4 as that of the conventional non-concurrent version. When miss cycles are overlapping with hit accesses, the processor will not stall; the processor can continue processing with the hit accesses. According to Eq. (4), C-AMAT is eight cycles out of 5 accesses or 1.6 cycles per access. Compared to AMAT, C-AMAT is 57.9 percent smaller. The difference between AMAT and C-AMAT is due to the contribution of concurrency of memory

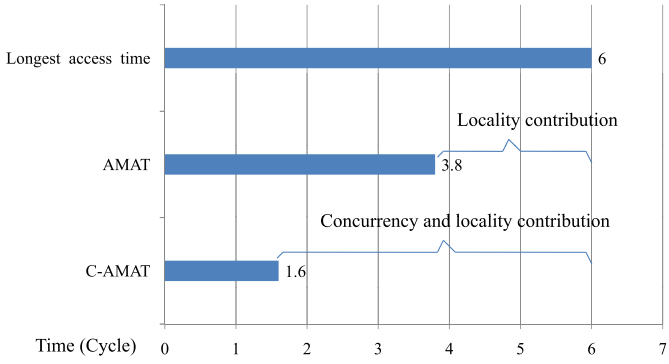

Fig. 2. The longest access latency, AMAT, and C-AMAT of Fig. 1.

access. Fig. 2 shows the difference among the longest access time, AMAT and C-AMAT.

A key contribution of C-AMAT is that it provides a unified formulation to capture the joint performance impact of locality and concurrency. Performance analysis and optimization can be conducted with the five parameters of C-AMAT. Access Per memory active Cycle is a newly proposed memory performance measurement to consider memory concurrency[36], [46]. The relation between C-AMAT and APC is given by Eq. (5) [36], [46]. The measurement of C-AMAT can be obtained directly through APC as shown in Eq. (5), where n is the total number of memory layers.

$$C\text{-}AMAT_i = \frac{1}{APC_i} \ (i = 1, 2, 3, \ldots, n.). \tag{5}$$

C-AMAT contains AMAT as a special case where memory concurrency does not exist. When memory concurrency does not exist, C_H = 1 and C_M = 1, pAMP = AMP, pMR = MR, therefore, C-AMAT = AMAT. C-AMAT provides an approach to evaluate and optimize the five performance parameters, individually or in combination. Note that C-AMAT can be used in multi-core environments, since each core can measure their individual C-AMAT separately. Due to these merits, we choose to use C-AMAT in the LPM study.

In an in-order processor, when a cache miss occurs, the processor waits for the fetched data before continuing, stalling processor pipeline for several cycles. The stalled cycles per instruction due to memory access are often referred to as memory stall time (MST). Eq. (6) is the conventional MST formula based on AMAT for in-order processor [48]. Here, f_{mem} is the portion of the instructions that access memory.

$$MST(in\text{-}order) = f_{mem} \times AMAT_1. \tag{6}$$

Unfortunately, Eq. (6) is no longer true for out-of-order (OoO) processors, because it does not reflect the concurrency in the modern complex memory systems. To address this issue, Eq. (7) presents the memory stall time expression of out-of-order processors, that is, the relationship between the memory stall time and the concurrency-aware data access time C-AMAT [27]. Here, $\text{overlapRatio}_{c\text{-}m}$ is the ratio of the computing and memory access overlapping time over the total memory access time. In modern processors, simultaneous multi-threading, out-of-order execution, and non-blocking cache contribute $\text{overlapRatio}_{c\text{-}m}$ by enabling computation to continue while memory access is being conducted. The expression of $\text{overlapRatio}_{c\text{-}m}$ will be provided in Section 3.

$$MST(OoO) = f_{mem} \times C\text{-}AMAT_1 \times (1 - overlapRatio_{c\text{-}m}). \tag{7}$$

Eq. (6) is a special case of Eq. (7), because in-order processor is a special case of out-of-order processor when $\text{overlapRatio}_{c\text{-}m}$ is zero and concurrency does not exist (i.e., C-AMAT=AMAT). In other words, Eq. (7) holds for any type of processors, regardless memory concurrency is involved or not. Therefore, in this study, we take Eq. (7) rather than Eq. (6) as the basic performance formula.

3 The Time Model of LPM

An accurate time model is of great importance for efficient performance optimization. The time model of LPM should include the effect of concurrency and locality, especially the hiding influence among cache hits and misses, and the overlapping impact between computing and data access.

Theorem 1 shows the relation between C-AMAT and AMAT. Therefore, C-AMAT concerns the effects of both locality and concurrency.

Theorem 1 (Relation between C-AMAT and AMAT). *C-AMAT and AMAT have the following relationship, where C is the average data access concurrency.*

$$C\text{-}AMAT = C^{-1} \times AMAT. \tag{8}$$

Proof. Assume the number of data accesses is N and the number of memory active cycles is M.

Each data access has its own length of duration time; thus each data access is corresponding to a time line (See Fig. 1 for instance). We have two different methods to count the total length of the time line of each data access. One method is to count access by access, and the result is $N \times AMAT$. The other method is to count cycle by cycle, and the result is $\sum_{i=1}^{M} C_i$.

The results of the two counting methods are same, that is, the sum of the concurrency C_i of each memory active cycle is the same as the sum of the sequential access cycles of each of the N data accesses. Therefore, we have

$$\sum_{i=1}^{M} C_i = N \times AMAT. \tag{9}$$

Recall Eq. (5), we have

$$C\text{-}AMAT = \frac{M}{N}. \tag{10}$$

Combining Eqs. (9) and (10), we get

$$\sum_{i=1}^{M} C_i = \frac{M}{C\text{-}AMAT} \times AMAT. \tag{11}$$

That is,

$$\sum_{i=1}^{M} C_i \times \frac{1}{M} = \frac{AMAT}{C\text{-}AMAT}. \tag{12}$$

TABLE 2
Concurrency Degree of Each Pure Miss Cycle

Clock cycle id	p_1	p_2	...	p_d
Pure miss concurrency degree	C_{1p}	C_{2p}	...	C_{dp}

TABLE 3
Concurrency Degree of Each Hit/Miss Mixed Cycle

Clock cycle id	h_1	h_2	...	h_e
Hit concurrency degree	C_{1h}	C_{2h}	...	C_{kh}
Miss concurrency degree	C_{1m}	C_{2m}	...	C_{km}

Recall the definition of memory concurrency, C, we have

$$C = \sum_{i=1}^{M} C_i \times \frac{1}{M}. \tag{13}$$

Combining Eqs. (12) and (13), we conclude the theorem. □

It is well known that the widely used average memory access time model, AMAT, can be extended recursively to the next layer of the memory hierarchy [22]. It is important to know whether this recursiveness is also true for C-AMAT. Theorem 2 shows the recursive relationships between C-AMAT$_1$ and C-AMAT$_2$. Therefore, C-AMAT is an effective cross-hierarchy memory model.

Theorem 2 (Recurrence Relation of C-AMAT). *Assume C-AMAT$_1$ is the L1 C-AMAT and C-AMAT$_2$ is the L2 C-AMAT. Eq. (14) gives the recurrence relation of C-AMAT.*

$$C\text{-}AMAT_1 = \frac{H_1}{C_{H_1}} + MR_1 \times \kappa_1 \times C\text{-}AMAT_2, \tag{14}$$

where

$$C\text{-}AMAT_1 = \frac{H_1}{C_{H_1}} + pMR_1 \times \frac{pAMP_1}{C_{M_1}}$$

$$C\text{-}AMAT_2 = \frac{H_2}{C_{H_2}} + pMR_2 \times \frac{pAMP_2}{C_{M_2}}$$

$$\kappa_1 = \frac{pMR_1}{MR_1} \times \frac{pAMP_1}{AMP_1} \times \frac{C_{m_1}}{C_{M_1}}. \tag{15}$$

Proof. According to Theorem 1, for the second memory layer, we have

$$C\text{-}AMAT_2 = \frac{AMAT_2}{C_{m_1}}, \tag{16}$$

and recall Eq. (2),

$$AMAT_2 = AMP_1, \tag{17}$$

we then have

$$C\text{-}AMAT_2 = \frac{AMP_1}{C_{m_1}}. \tag{18}$$

Combining Eqs. (15) and (18), we obtain

$$MR_1 \times \kappa_1 \times C\text{-}AMAT_2 = pMR \times \frac{pAMP_1}{C_{M_1}}. \tag{19}$$

We then conclude the theorem. □

By following the same arguments for Eq. (14), C-AMAT can be further extended to the next layer of the memory hierarchy as well. Generally, for memory layer $i = 1, 2, \ldots, n-1$, we have Eq. (20).

$$C\text{-}AMAT_i = \frac{H_i}{C_i} + MR_i \times \kappa_i \times C\text{-}AMAT_{(i+1)}. \tag{20}$$

By Theorem 2, Eq. (14) illustrates useful properties of concurrent data accesses: the impact of C-AMAT$_2$ on C-AMAT$_1$ can be mitigated by MR$_1$ and κ_1, where MR$_1$ indicates the locality contribution and κ_1 quantifies the hit-miss masking effect. This potential of penalty reduction is the theoretical foundation and motivation of the layered matching mechanism proposed in this research. The parameter κ_1 is measurable and has a physical representation. A part of cache miss penalty can be masked by cache hit activities. The parameter κ_1 indicates these contributions.

From Eq. (14), we can identify which layer incurs the memory stall time. Taking the typical values of the parameters for modern processors, H$_1$ = 1, C$_{H_1}$ = 4, MR = 1 percent, κ_1= 3/4. Let us consider two cases. If C-AMAT$_2$ = 100, then by Eq. (14), C-AMAT$_1$ = 1. If C-AMAT$_2$ = 20, then by Eq. (14), C-AMAT$_1$ = 0.4. It is seen that the 5-fold difference of C-AMAT$_2$ values in the two cases only bring in 2.5-fold difference to C-AMAT$_1$. That is, there exists a latency-hiding effect in the memory system. The latency-hiding effect is contributed by two factors, MR$_1$ and κ_1, where MR$_1$ is the contribution of locality and κ_1 is the result of hit-miss masking effect.

Theorem 3 (Hiding influence among cache hits and misses). *The value of κ_1 defined in Eq. (15) equals the ratio of the amount of pure miss cycles over that of conventional miss cycles, and its value is less than or equal to 1.*

Proof. As shown in Table 2, we assume the number of pure miss cycles is d, and the cycle id is $p_1, p_2, \ldots, p_d$, respectively. Meanwhile, as shown in Table 3, we assume the number of hit/miss mixed (i.e., miss but not pure miss) cycles is e, and the cycle id is $h_1, h_2, \ldots, h_e$, respectively. By definition, conventional miss cycles include the cycles of the above two cases, and therefore the conventional miss cycle id is $h_1, h_2, \ldots, h_e, p_1, p_2, \ldots, p_d$, respectively. That is, the number of conventional miss cycles is $(e + d)$.

Let us assume the number of pure misses and conventional misses are β and α, respectively. By Tables 2 and 3, we can calculate MR, pMR, C_m, C_M, AMP, and pAMP, respectively. The number of accesses is N. Therefore, MR=α/N, pMR=β/N.

$$C_m = \frac{1}{e+d}\left(\sum_{i=1}^{e} C_{im} + \sum_{i=1}^{d} C_{ip}\right) \tag{21}$$

$$C_M = \frac{1}{d}(\sum_{i=1}^{d} C_{ip}) \tag{22}$$

$$AMP = \frac{1}{\alpha}(\sum_{i=1}^{e} C_{im} + \sum_{i=1}^{d} C_{ip}) \tag{23}$$

$$pAMP = \frac{1}{\beta}(\sum_{i=1}^{d} C_{ip}). \tag{24}$$

Combining the above equations, we get

$$\kappa_1 = \frac{pMR_1}{MR_1} \times \frac{pAMP_1}{AMP_1} \times \frac{C_{m_1}}{C_{M_1}} = \frac{d}{e+d}. \tag{25}$$

As $e \geq 0$, we can conclude that $\kappa_1 \leq 1$ is always true. □

According to Eq. (14) in Theorem 2, the smaller the κ_1 is, the smaller the C-AMAT$_1$ is, thus κ_1 is a new latency reducer presented in this study.

Due to Theorem 2, the measurable parameter κ_1 has a physical representation, which reflects the difference between pure miss and conventional miss. Note that a part of the conventional miss penalty is masked by hit activities, and the parameter κ_1 indicates these contributions. In this meaning, κ_1 quantifies the latency-hiding effect in the first memory layer perspective. We have proved that $\kappa_1 \leq 1$ is always true. Taking Fig. 1 for example, $\kappa_1 = 2/3$. The impact of C-AMAT$_2$ toward the final C-AMAT$_1$ can be reduced by both MR$_1$ and κ_1. In this meaning, κ_1 can be taken as a lever to reduce C-AMAT$_1$.

Theorem 4 (Overlapping impact between computing execution time and data access time). *Assume the ratio of the amount of conventional miss cycles over that of memory active cycles is μ_1. The following relation holds.*

$$1 - overlapRatio_{c\text{-}m} = \mu_1 \times \kappa_1. \tag{26}$$

Proof. By definition, overlapRatio$_{c\text{-}m}$ is the ratio of the computing and memory access overlapping time over the total memory access time. Assume the amount of pure miss cycles is d, the amount of conventional miss cycles is $(e+d)$, and the number of pure hit cycles is c.

The computing and memory access overlapping time includes two exclusive parts, i.e., pure hit cycles, miss but not pure miss cycles, thus the overlapping time is $c+e$. The total access time, i.e., the number of memory active cycles, is $c+d+e$. Therefore,

$$overlapRatio_{c\text{-}m} = \frac{c+e}{c+d+e}. \tag{27}$$

By definition,

$$\mu_1 = \frac{e+d}{c+d+e}. \tag{28}$$

By Eq. (25), we have

$$\kappa_1 = \frac{d}{e+d}. \tag{29}$$

Then we have Eq. (26). □

In Fig. 1, there exist 8 memory active cycles, 5 pure hit cycles, 3 conventional miss cycles, and only 2 pure miss cycles which incur memory stall, so in 6 cycles processor computing can continue. The value of overlapRatio$_{c\text{-}m}$ is 6/8. The value μ_1 equals 3/8. The κ_1 is 2/3. All the five instructions issue memory access, so the f$_{mem}$ is 1. The MST (per instruction) is only 2/5 cycles.

4 The QoS Goal of LPM

The execution time of a computer processor consists of two parts [22]: processor computing time and memory stall time. Here the processor computing time is the time when the processor is occupied executing the user program. Eq. (30) is the classic formulation of the CPU-time in terms of these two components of time [22].

$$CPU\text{-}time = IC \times (CPI_{exe} + MST) \times cycle\text{-}time. \tag{30}$$

Here, IC is the number of instructions, cycle-time is the length of a clock cycle, and CPI$_{exe}$ is the processor computation cycles per instruction under perfect cache (i.e., no cache miss occurs).

As shown in Eq. (31), when the ratio of MST over CPI$_{exe}$ is less than a threshold, $\Delta\%$, we say the memory system is highly efficient, where Δ is a parameter determined by the user and is application dependent. We take the percentage approach, because the impact of memory stall time is relative to computing time. If MST is less than 1 percent of the computing time, for instance, we can claim there is no noticeable memory wall effect toward the overall performance.

$$\frac{MST}{CPI_{exe}} \leq \Delta\%. \tag{31}$$

The goal of memory system performance optimization is to make $\Delta\%$ as small as possible, with a reasonable tuning cost. We define Memory System Efficiency (MSE) as the ratio of pure computing time over the sum of pure computing time plus memory stall time as shown by Eq. (32). When memory stall time is zero, MSE is 100 percent; when $\Delta\%$ is close to zero, the value of MSE is close to 1 - $\Delta\%$. We can use 1 - $\Delta\%$ to approximate MSE. For examples, when $\Delta\%$ is 1 percent, the MSE is about 99 percent. When $\Delta\%$ is 10 percent, the MSE is about 90 percent.

$$MSE = \frac{CPI_{exe}}{CPI_{exe} + MST} \geq \frac{1}{1+\Delta\%} \approx 1 - \Delta\%. \tag{32}$$

Before presenting the LPM algorithm to achieve high efficiency, we first estimate its potential benefit. For many data-intensive applications, memory stall time can be up to 90 percent of the total application execution time [20], [21], [25], [33], whereas pure computing time is only 10 percent. At that time, MSE is only 10 percent, and MST is 9 times of the pure computing time. Therefore, when the optimization goal of "$\Delta\% = 10\%$" is achieved, MSE is improved from 10 to 90 percent; when the optimization goal "$\Delta\% = 1\%$" is achieved, MSE is improved from 10 to 99 percent. As shown in our case studies, the "$\Delta\% = 10\%$" condition is reachable on reconfigurable architectures where a huge design space exists.

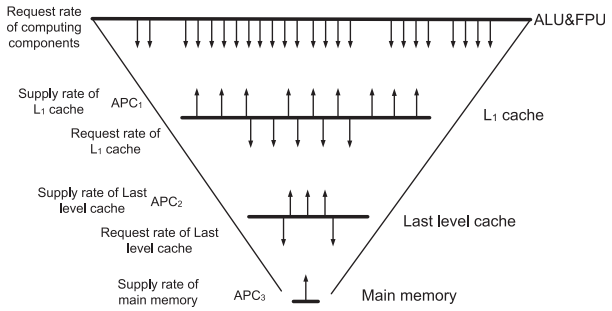

Fig. 3. The request-reply LPM model (The memory hierarchy is seen from the perspective of performance matching; note that the computing components, ALU and FPU, are taken as L0.).

5 The Analytical Definition of LPM

In this section, we quantify the effects of mismatch, and therefore provide a systematic way of performance optimization.

To optimize the performance of a memory system, we need to match the performance at each layer of a hierarchical memory system as closely as possible. Fig. 3 illustrates the Layered Performance Matching model of a memory hierarchy. For the sake of simplicity and brevity, L2 is taken as the LLC in Fig. 3 and in this study. The extension to additional cache levels is straightforward.

Each memory layer (except the top layer and the bottom layer) has data accesses at two sides, including the requests issued from the upper layer and the supplies provided by the lower layer. For example, the demand from computing components is the request rate of ALU and FPU, while the service is the supply rate of L1 cache.

The matching ratios in the LPM model are the ratios of request rate and supply rate between any two memory layers. Eqs. (33) and (34) are the definitions of the LPM Ratios (LPMR) that are abbreviated as $LPMR_1$ and $LPMR_i$, respectively.

$$LPMR_1 = \frac{Request\ rate\ from\ ALU\&FPU}{Supply\ rate\ by\ L1\ cache} \tag{33}$$

$$LPMR_i = \frac{Request\ rate\ from\ L_{i-1}}{Supply\ rate\ by\ L_i} \quad (i = 2, \ldots, n.). \tag{34}$$

Although supplies are activated by requests, the supply rate by a lower layer can be greater than the request rate from an upper layer, because one cache line may provide multiple cache hits. In counting the LPM ratio, we differentiate demand accesses and prefetches. When prefetching is used, the requests of prefetches are not counted into the request rate of a memory layer. Prefetchers are designed to fetch demanded data in advance. If the prefetches are successful, the prefetched data would be used as demanded data. Otherwise, prefetchers generate data traffic without any benefit. As we only concern useful work, prefetch requests are not counted into the request rate in our C-AMAT and APC calculations.

Theorem 5 (LPMR in terms of APC). *The LPM Ratios can be expressed as Eqs. (35) and (36), respectively, in terms of APC.*

$$LPMR_1 = \frac{IPC_{exe} \times f_{mem}}{APC_1} \tag{35}$$

$$LPMR_i = \frac{IPC_{exe} \times f_{mem} \times \prod_{j=1}^{i-1} MR_j}{APC_i} \ (i = 2, \ldots, n.). \tag{36}$$

Proof. The supply rates of different layers in the memory hierarchy can be denoted by APC values of corresponding layers [36], [46]. That is, APC supplied by L1, LLC, and main memory are referred to as APC_1, APC_2, and APC_3, respectively.

IPC_{exe} is the peak performance of a processor. IPC_{exe} multiplied with the memory access frequency equals the memory access intensity perceived by the L1 cache. IPC_{exe} is the reciprocal of CPI_{exe}. Therefore,

$$Request\ rate\ of\ ALU\ and\ FPU = IPC_{exe} \times f_{mem}. \tag{37}$$

The memory access intensity perceived by the L_i is the request rates from the L_{i-1}, which equals the memory intensity filtered by L_{i-1}.

$$Request\ rate\ from\ L_i = IPC_{exe} \times f_{mem} \times \prod_{j=1}^{i-1} MR_j \quad (i = 2, \ldots, n). \tag{38}$$

MR_j here refers to the local miss rate of the jth cache. Therefore, by Eqs. (33) and (34), Theorem 5 holds. □

Theorem 6 (LPMR in terms of total value of C-AMAT). *In terms of C-AMAT, the matching degree of different memory layers, measured by LPM Ratios, can be expressed as Eqs. (39) and (40).*

$$LPMR_1 = \frac{C\text{-}AMAT_1 \times f_{mem}}{CPI_{exe}} \tag{39}$$

$$LPMR_i = \frac{C\text{-}AMAT_i \times f_{mem} \times \prod_{j=1}^{i-1} MR_j}{CPI_{exe}} \quad (i = 2, \ldots, n.). \tag{40}$$

Proof. According to Eq. (5), the supply traffic in each memory layer can be transformed from APC to C-AMAT for performance analysis and optimization. Note that CPI_{exe} is the reciprocal of IPC_{exe}. Therefore, by Theorem 5, we have Theorem 6. □

It is important to notice that LPMR quantifies the mismatch degree in a memory layer. The significance of Theorem-6 is that it links C-AMAT with LPMR. Therefore, the five parameters of C-AMAT can be used to decrease C-AMAT and thus the LPMR. Theorem-6 tells us that the C-AMAT values are measured in individual layers of a memory hierarchy so that the LPM optimizations can be done in individual layers too.

Theorem 7 (LPMR in terms of analytical expression of C-AMAT). *The LPM Ratios can be expressed in terms of the five parameters of C-AMAT as Eqs. (41) and (42).*

$$LPMR_1 = \left(\frac{H_1}{C_{H_1}} + pMR_1 \times \frac{pAMP_1}{C_{M_1}}\right) \times \frac{f_{mem}}{CPI_{exe}} \tag{41}$$

$$LPMR_i = \left(\frac{H_i}{C_{H_i}} + pMR_i \times \frac{pAMP_i}{C_{M_i}}\right) \times \frac{f_{mem}}{CPI_{exe}} \times \prod_{j=1}^{i-1} MR_j \quad (i = 2, \ldots, n.). \tag{42}$$

Proof. Combining Theorem 6 and Eq. (4), we get Theorem 7.□

As we have defined the LPM, we are now ready to study the requirement for LPM to achieve an arbitrarily given memory system efficiency.

6 The Minimum Requirement for LPM

The smaller the value of LPMR is, the higher the memory performance is. However, when LPMR is already small, further reduction may only bring in minimum performance gain with a large effort. To make the tuning process effective, we only need to meet the minimum requirement for LPM.

Theorems 8 and 9 give the thresholds of $LPMR_1$ and $LPMR_2$ to achieve an arbitrarily given performance goal.

Theorem 8 (Minimum Requirement for $LPMR_1$). *If Eq. (43) holds for an arbitrarily given application, then the memory stall time of the application is less than or equal to $\Delta\%$ of its pure computing time.*

$$LPMR_1 \leq \frac{\Delta\%}{\mu_1 \times \kappa_1}. \tag{43}$$

Proof. With Eqs. (7) and (39), we get the relation between memory stall time and $LPMR_1$ as shown in Eq. (44).

$$MST = CPI_{exe} \times (1 - overlapRatio_{c\text{-}m}) \times LPMR_1. \tag{44}$$

By Eq. (26) in Theorem 4, we have

$$MST = CPI_{exe} \times (\mu_1 \times \kappa_1) \times LPMR_1. \tag{45}$$

To achieve high system efficiency (see Eq. (31)), Eq. (46) must be satisfied.

$$MST \leq CPI_{exe} \times \Delta\%. \tag{46}$$

Combining Eqs. (45) and (46), we obtain Eq. (43). □

Corollary 1. *The threshold value of $LPMR_1$ decreases with the increase of the desired memory system efficiency (i.e., $1 - \Delta\%$) and L1's latency-hiding effect (i.e., $\mu_1 \times \kappa_1$).*

Proof. Corollary 1 is a direct result of Theorem 8. □

From the perspective of the first memory layer, $\mu_1 \times \kappa_1$ indicates the latency-hiding effect. In Fig. 1, μ_1 is 3/8, and κ_1 equals 2/3, so $\mu_1 \times \kappa_1$ is 2/8. Therefore, the threshold of $LPMR_1$ can be four times of $\Delta\%$. In practices, the value of $\mu_1 \times \kappa_1$ can be very small. For instance, we measured that the $overlapRatio_{c\text{-}m}$ of *wordcount* in BigDataBench is 98.4 percent, and $\mu_1 \times \kappa_1$ is 1.6 percent. In this case, if the expected computing efficiency is 90 percent (i.e., $\Delta\%$ is 10 percent), the value of T_1 is 6.25. In this manner, the requirement for the T_1 has been lowered.

Theorem 9 (Minimum Requirement for $LPMR_2$): *For an arbitrarily given $\Delta\%$, to make an application's memory stall time be less than or equal to $\Delta\%$ of its pure execution time, the minimum requirement for $LPMR_2$ is shown as Eq. (47).*

$$LPMR_2 \leq \frac{\Delta\%}{\kappa_1}. \tag{47}$$

Proof. By Theorem 2 in [27],

$$C\text{-}AMAT_1 \times (1 - overlapRatio_{c\text{-}m}) = pMR_1 \times \frac{pAMP_1}{C_{M_1}}. \tag{48}$$

Combining Eqs. (19) and (48), we derive

$$MST = f_{mem} \times MR_1 \times \kappa_1 \times C\text{-}AMAT_2. \tag{49}$$

Then due to the expression of $LPMR_2$ in Eq. (36), we get Eq. (50).

$$MST = CPI_{exe} \times \kappa_1 \times LPMR_2. \tag{50}$$

To achieve high system efficiency (see Eq. (31)), Eq. (51) must be satisfied.

$$MST \leq CPI_{exe} \times \Delta\%. \tag{51}$$

Combining Eqs. (50) and (51), we obtain Eq. (47). □

Corollary 2. *The threshold value of $LPMR_2$ decreases with the increase of the desired MSE (i.e., $1 - \Delta\%$) and L2's latency-hiding effect (i.e., κ_1).*

Proof. Corollary 2 is a direct result of Theorem 9. □

From the perspective of the second memory layer, κ_1 indicates the latency-hiding effect. In Fig. 1, there exist 3 conventional miss cycles, but only two of them are pure miss cycles, which incur processor data stall, so κ_1 equals 2/3. In practices, κ_1 can be less than 50 percent. For instance, we measured that the κ_1 value of the *bewaves* benchmark is 14.9 percent (the experimental environment will be presented in Section 8). Thus, for the *bewaves* benchmark, up to 85.1 percent latency can be hidden. If the expected memory system efficiency is 90 percent (i.e., $\Delta\%$ is 10 percent), the threshold of $LPMR_2$, T_2, is 6.04, which means that the rate of data supply can be up to 6 time slower than that of the data request to achieve the desired memory system efficiency.

As shown in Fig. 3, from a hierarchy perspective, the computing components, ALU and FPU, can be taken as L0. Of general significance, $\mu_1 \times \kappa_1$ and κ_1 quantify the latency-hiding effects in the view of L0 and L1, respectively. Theorems 8 and 9 show the impact of latency-hiding on LPM. If the latency-hiding effect is more significant, the required LPM values would be larger and thus easier to achieve.

Once the required LPM values given by Theorems 8 and 9 are met, the optimizations of the corresponding layers are finished. In next section, we will give the LPM algorithm to improve the matching of the layered performance to achieve global optimizations of memory systems.

7 The Algorithm of LPM

Different optimization techniques mentioned in the Background section can be used as levers to conduct the performance optimization at each layer. Fig. 4 shows the pseudocode of the LPM algorithm. Let T_i be the threshold of $LPMR_i$ as given by Theorems 8 and 9. Initially, the information about mismatch is measured and collected. Optimizations are necessary only when the $LPMR_1$ value is large (larger than T_1). If $LPMR_1$ is larger than T_1, we need to decide to optimize L1 layer and L2 layer simultaneously, or only

LPM Algorithm

```
1:  // Initially measure the metrics
2:    For each application or thread, measure the LPMR values in a memory hierarchy
3:    Get the threshold T1 and T2 according to Theorem-8 and 9
4:  Begin Do
5:  // LPM optimization loop
6:    // Case I when both L1 and L2 layer need an optimization
7:    While (LPMR1 > T1 and LPMR2 > T2) Do
8:      Optimizing at L1 layer and L2 layer,
9:      Update all the metrics (LPMR1 , LPMR2, T1 and T2)
10:   Until (LPMR1 ≤ T1 or LPMR2 ≤ T2)
11:   // Case II when only L1 layer needs an optimization
12:   While (LPMR1 > T1 and LPMR2 ≤ T2) Do
13:     Optimizing at L1 layer
14:     Update all the metrics (LPMR1 , LPMR2, T1 and T2)
15:   Until (LPMR1 ≤ T1)
16:   // Case III when no layer needs to optimize and overprovision may need to reduce
17:   // δ is a positive value
18:   While (LPMR1 + δ < T1) Do
19:     Reduce hardware overprovision
20:     Update all the metrics (LPMR1, LPMR2, T1 and T2)
21:   Until (LPMR1 ≥ T1 − δ)
22:   // Case IV when no layer needs to optimize and no overprovision needs to reduce
23:   If (T1 ≥ LPMR1 ≥ T1 − δ)
24:     End the algorithm
25:   Endif
26: Until (End)
```

Fig. 4. Pseudo-code of the LPM algorithm (Let T_i be the threshold of $LPMR_i$ as given by Theorems 8 and 9.).

optimize the L1 layer. In both cases, the optimization of the L1 layer is required.

Only when $LPMR_1$ is larger than T_1 and $LPMR_2$ is larger than T_2, the optimizations of the L1 layer and L2 layer are needed at the same time.

After each optimization, all the metrics are updated to decide if it is necessary to continue optimizing. An interesting point is how to find the "minimum match" when we have $LPMR_1 < T_1$. If $LPMR_1$ is smaller than its threshold value T_1 by more than δ, we know that hardware has been over provided. The value of δ is positive, which can be set by users with a tradeoff between benefit and effort. This step is optional, but we include it in the algorithm for completeness.

Note that all the steps of the LPM algorithm can be conducted online to adapt to the dynamic behavior of the underlying application. The LPM algorithm is called periodically for each time interval. The time interval size can be set with a trade-off between performance improvement and optimization cost. The cost here is due to the implementation of a reconfiguration operation or a scheduling operation. The LPM algorithm also can be conducted offline for application specific design optimization. We conduct online optimizations in this study, since it is the harder part of the problem.

Based on the GEM5 simulator, we implemented a reconfigurable 16-core CMP, with four cycles cost for each reconfiguration operation and 40 cycles for each scheduling. In our experiment, for hardware approach, we found that when the interval size is set to 10 cycles for each scheduling, 96 percent of the burst data access patterns can be perceived and processed timely. When the interval size is set to 20 cycles, 89 percent of the burst data access patterns can be perceived and processed timely. For software approach, when the interval size is set to 40 cycles, 73 percent of the burst data access patterns can be perceived and processed timely.

An interesting point of the LPM algorithm is that the optimization presents minimal but enough hardware support to ease memory stall. It not only can be applied to a general architecture design, but also can be applied in a shared environment to improve overall performance via scheduling. Taking time intervals i_1 and i_2 for example, we assume application in the interval i_1 is data intensive, and in the interval i_2 is not data intensive. In the initial stage of i_1, the LPMR values are large and thus optimization is conducted. The optimization takes effect in most of the time in interval i_1 (deducing the startup cost). Then in interval i_2, the data access patterns are changed (i.e., become not data intensive), thus the thresholds of LPMRs are also different in comparison (less than) with the previous threshold. The hardware support is over provided, and thus should be reduced for the present application (and then can be used for those applications that need more hardware support).

The optimization of the LPM algorithm shown in Fig. 4 can be achieved via the hardware, software, or a mixed approach, respectively. If the hardware can be improved, an optimization can be conducted easily by improving the five parameters of C-AMAT to increase data access concurrency and locality. If we assume the underlying hardware configuration is fixed, the LPM-based optimization can be conducted via software by exploring and utilizing heterogeneity of the underlying hardware. However, the condition of T_1 and T_2 may not be satisfied via pure software optimization or pure hardware optimization.

Let us take the scheduling on heterogeneous system for example to consider the software approach. Assuming N different computing units exist in a heterogeneous computing system, the idea of the software approach is to sort the LPMRs of the N computing units to find a best hardware/application match (in terms of the capacity of the available computing units).

The purposes of the hardware approach and software approach are the same, that is, to mitigate the layered mismatch to achieve application-awareness and hierarchy-awareness to improve performance. If we only have very limited hardware choices in hardware design or hardware availability, the software approach is the way to go. The difference is that hardware approach *designs* an optimal hardware configuration for an application, whereas software approach *chooses* an available configuration from a set of existing configurations to best match hardware configuration for an application.

The optimization potentials of hardware approach and software approach are determined by their available design space. In most cases, the design space of the hardware approach is much larger than that of the software approach, thus the former has a much larger room for performance improvement, and the minimum requirements indicated by T_1 and T_2 may not be satisfied solely by software optimization.

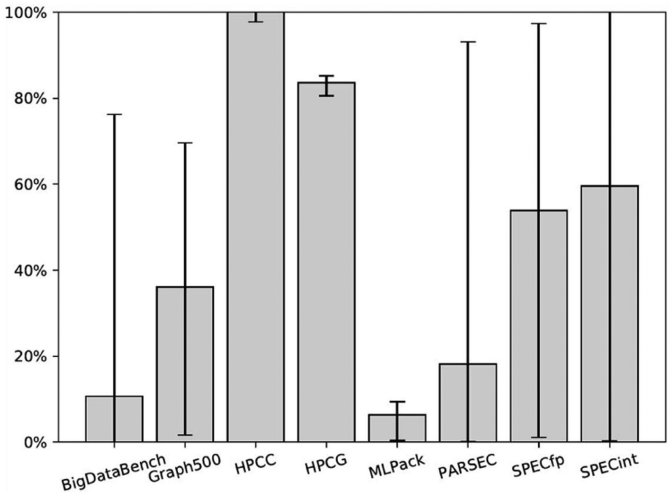

Fig. 5. The value of μ_1 for hotspots of diverse benchmarks.

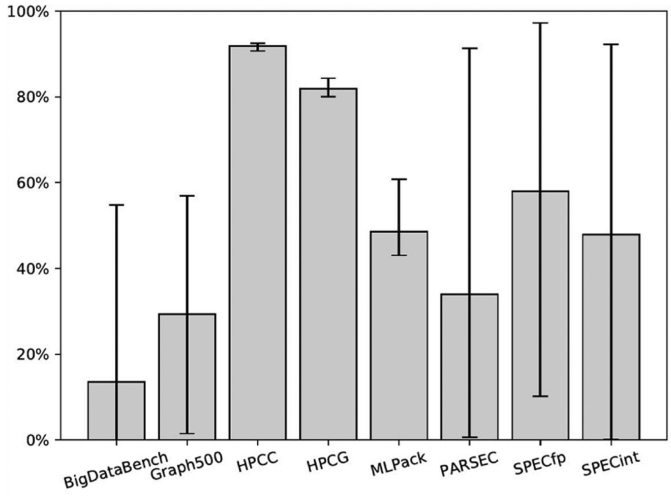

Fig. 6. The value of κ_1 for hotspots of diverse benchmarks.

As will be shown in next section, the LPM algorithm can facilitate software optimizing on commercial processors that are not reconfigurable, facilitate architecture design space exploration to avoid exhausting search for the best configuration, and facilitate scheduling to achieve application-awareness and heterogeneity-awareness.

8 Optimization Case Studies

In the following three sections, one case study for LPM optimization is conducted on commercial processors, and two representative case studies for LPM optimization are conducted on simulators. Case Study I uses software approach on commercial processors. Case Study II uses hardware approach on reconfigurable architectures. Case Study III deploys software approach to optimize LPMR in heterogeneous memory systems. The state-of-the-art cycle-accurate simulator GEM5 [1] is used to provide an appropriate full system performance simulation. A detailed out-of-order CPU model and DRAMSim2 module in the GEM5 simulator were adopted to achieve the most accurate simulation results. SPEC CPU2006 and CPU2017 benchmark suites with "ref" input data sets are used [34].

8.1 Case Study I: LPM Optimization on Commercial Processors

In this section, we conduct a case study on the mcf benchmark running on an Intel Xeon E5-2,630 processor. Experimental testing and case study can be conducted on either a real machine or a simulator. We conduct experiments on real machine in this section, because we want to verify that the LPM method can work on real machines only by software approach. We will conduct case studies on simulator in the other sections.

It is of great practical significance that the values of the parameters in LPM model can be measured in mainstream commercial processors. Intel Xeon E5 provides many performance monitor unites (PMU). It is worth noting that the number of memory active cycles are different at different memory layers. The number of active cycles in L1 is recorded by CYCLE_ACTIVE_MEM_ANY, while that in L2 is counted by L1D_PEND_MISS:PENDING_CYCLES. Meanwhile, OFFCORE_REQUESTS_OUTSTANDING:ALL_DATA_RD_CYCLES counts the number of active cycles in L3 for the detected core. Based on the PMUs in the processors, PAPI [11], HPCToolkits [12], Perfexpert [13], are used to identify the hotspots and their memory access patterns. Figs. 5 and 6 show the value of μ_1 and κ_1 for the hotspots of diverse benchmarks, respectively. The benchmarks are from eight suites that include SPECint and SPECfp from SPEC CPU2006 [34] [39], PARSEC [43], BigDataBench [45], MLbench [40], HPCC [42], HPCG [41] and Graph500 [44]. Each benchmark suite may include several different benchmarks and tests, and each benchmark may include multiple hotspots. For the convenience of presentation, Fig. 6 only shows the average and the range.

The mcf benchmark from SPEC CPU2006 is developed for application in the public transportation systems of Hamburg and Berlin, and has been integrated into the vehicle and personnel planning system MICROBUS [8]. We select mcf benchmark to demonstrate the optimization process, because mcf benchmark is known as a cache buster (see page 116 in [22]), which means the on-chip hierarchy cannot deal with the data requests of mcf effectively. The optimizing process will help people understand why it is a cache buster. Table 4 shows the configuration of the experimental system. The mcf benchmark is running on only one of the 10 cores, but possesses all the LLC without competitive applications. Note that, in the baseline, we do not disable the hyperthreading and turbo boost.

The hotspot code before optimization is shown in Fig. 7, which traverses the link structures. The link structure implies not only low locality, but also low concurrency of data accesses. Although the hotspot is very short, it takes 34.3 percent of total benchmark running time. Before optimization, the IPC is 0.452, $LPMR_2$ is 8.42, κ_1 is 0.86, memory stall time is 7.26-fold of pure computing time (i.e., $\Delta\%$ is 726 percent), and the computer's system efficiency is only 12.1 percent.

According to Corollary 2, for an arbitrarily given $\Delta\%$, the threshold value of $LPMR_2$ decreases with the decrease of the overlapping between cache hits and cache misses (i.e., 1 - κ_1). Because κ_1 is 0.86 which is close to one, the threshold value of $LPMR_2$ is 0.93 when the desired MSE is 80 percent. Note that the real value of $LPMR_2$ before optimization is 8.42, which is about 9-fold of the threshold value.

TABLE 4
The Experimental System Configuration

CPU chip	Intel Xeon E5-2630 v4
L1 dcache and L1 icache latency	4 cycles
L2 cache latency	about 12 cycles [1]
L3 cache latency	about 40 cycles [1]
DRAM memory latency	about 150 cycles [1]
CPU freqency	2.2 GHz
FP latency	2 cycles
FP slow latency	18 cycles
TLB latency	45 cycles
Number of Memory channels	4
Capacity per channel	16 GB

[1] *The latency value of each memory layer is variable, thus we measured them using Intel memory latency checker [14].*

To improve the system efficiency, according to Theorem 8, we need to reduce $LPMR_2$ or κ_1. The bottleneck in the mcf benchmark has a linked data access pattern and thus has a low concurrency. Prefetching is a way to increase concurrency, and prefetching will be effective if hardware concurrency is not a constraint. In this case study, the maximum available hardware concurrency in Intel Xeon processors is pretty high. For this situation, software prefetching can work well and can reduce $LPMR_2$.

In each iteration of the optimizing process, the MR and concurrency of each layer are changed simultaneously, thus their respective value and combined impact need to be considered. The C-AMAT takes the role of indicator to guide us to quickly find the optimal parameters of the prefetcher. The hotspot code after optimization is shown in Fig. 8.

Due to high cache miss rate of accessing the head pointer and tail pointer, software fetching is applied for these two pointers. Because head and tail pointers are the members of the arc structure, we load the arc structure into the on-chip caches before prefetching head and tail. Thus, the software prefetching strategy is to prefetch the arc structure required after 4 iterations and the head and tail required after 2 iterations.

After optimization, $LPMR_2$ has reduced by 50.2 percent. The total running time before optimization is 5.70×10^{11} cycles, while that after optimization is 5.01×10^{11} cycles, thus the improvement of performance is 13.8 percent. The memory stall time before optimization is 5.63×10^{11} cycles, and that after optimization is 4.60×10^{11} cycles.

Note that the original mcf program has 2766 different lines of code, and we have only added 9 lines of code in the optimization process, thus the small tuning effort pays off due to the effectiveness of the LPM mechanism.

```
/* price next group */
arc = arcs + group_pos;
for( ; arc < stop_arcs; arc += nr_group )
{
  if( arc->ident > BASIC )
  {
    red_cost = arc->cost - arc->tail->potential + arc->head->potential;
    …
  }
}
```

Fig. 7. The original hotspot code in mcf benchmark.

```
/* price next group */
arc = arcs + group_pos;
if (arc + (nr_group<<1) < stop_arcs)
  {
   _builtin_prefetch((arc+(nr_group<<1))->head,0,0);
   _builtin_prefetch((arc+(nr_group<<1))->tail,0,0);
  }
 if (arc + (nr_group<<1) + nr_group < stop_arcs)
 {
   _builtin_prefetch((arc+(nr_group<<1)+nr_group)->head,0,0);
   _builtin_prefetch((arc+(nr_group<<1)+nr_group)-tail,0,0);
  }
for ( ; arc < stop_arcs; arc += nr_group )
{
  _builtin_prefetch(arc + (nr_group<<4),0,0);
  if (arc + (nr_group<<2) < stop_arcs)
   {
   _builtin_prefetch((arc+(nr_group<<2))->head,0,0);
   _builtin_prefetch((arc+(nr_group<<2))->tail,0,0);
   }
  if ( arc->ident > BASIC )
   {
    red_cost = arc->cost - arc->tail->potential + arc->head->potential;
     ...
   }
}
```

Fig. 8. The optimized hotspot code in mcf benchmark.

This case study shows that the parameters of LPM methodology can be measured on commercial processors, and the LPM methodology has effectively guided the optimizing process even when hardware is not reconfigurable. On a real computing system that is not reconfigurable, we cannot change the hardware, thus we are only able to implement LPM algorithm via software approach. However, for computing systems that have reconfigurable supports, hardware approach also can be used.

8.2 Case Study II: LPM Optimization on Reconfigurable Architecture

Based on the C-AMAT definition and LPM definition, optimal hardware configurations can be found for mitigating the memory-wall impact. Architecture design space S includes all possible configurations of each parameter p_i, and can be expressed in the form of Cartesian product, $S = S_{p_1} \times S_{p_2} \times \cdots \times S_{p_n}$, where S_{pi} is the value range of architectural parameter p_i. The size of design space, $|S| = |S_{p_1} \times S_{p_2} \times \cdots \times S_{p_n}| = |S_{p_1}| \times |S_{p_2}| \times \cdots \times |S_{p_n}|$.

Dozens of parameters could be adjusted in a computer architecture, and each parameter has a set of different values. For the sake of brevity, only six architecture parameters are explored herein, including MSHR number, ROB size, and pipeline issue width. Provided that each architectural parameter can be set with 10 different values, the design space size of the six parameters is 10^6. Even with the limited consideration, there are one million possible configurations. As the design space is very large, exhausting search is not an option, and a LPM optimization algorithm becomes a must.

Under five configurations A to E, Table 5 shows the average LPMR values for "bwaves" benchmark from SPEC CPU2006. Let us cruise the general LPM algorithm in the architecture design space exploration. The goal of the optimization is to keep the memory stall time per instruction within

TABLE 5
LPMRs Under Configurations with Incremental Parallelism

Configuration	A	B	C	D	E
Pipeline issue width	4	4	6	8	8
ROB size	32	64	64	128	96
L1 MSHR numbers	2	8	16	16	16
L2 MSHR numbers	2	8	8	8	8
$LPMR_1$	8.1	6.2	2.1	1.2	1.4
T_1	2.8	2.8	2.8	2.8	2.8
$LPMR_2$	9.6	9.3	3.1	1.6	1.9
T_2	6.2	6.2	6.2	6.2	6.2
$LPMR_3$	6.4	8.1	5.8	2.3	2.6

1 percent $\times$ CPI_{exe}. First, in the current time interval, LPMRs are measured for each application. Table 5 shows the LPMR values in a memory hierarchy under configuration A. The threshold values of $LPMR_1$, $LPMR_2$ are 2.8 and 6.2 respectively based on the T_1 and T_2 given in Eqs. (43) and (47). Initially, both the LPMRs are higher than the threshold values so that the optimizations are carried in both layers at the same time.

We transform the architecture from configuration A to B in Table 5. With configuration B, the mismatches are still higher than their thresholds. Then we continue reduction and transform configuration B to C. The mismatch now in L2 layer is 3.1, which is already less than its threshold value, 6.2. Therefore, we no longer need continue reducing $LPMR_2$, and only need to focus on L1 layer mismatch.

We increase ROB, MSHR number and pipeline width. Configuration D is the first scheme which meets the requirement in the architecture exploration. The memory stall time is 0.97 percent of CPI_{exe}. The memory stall cost is already small. As an optional step, for the purpose of resource saving, we continue to check if hardware is over provided.

In our experiment, we set δ as 1 percent of T_1. According to the LPM algorithm, we do a fine tune to reduce possible hardware overprovision to achieve cost-efficiency, which leads to the final configuration E. Configuration E reaches our 1 percent goal with minimal hardware support. The saved resource can be used for other applications, making the computer more energy-efficient.

The $LPMR_1$ is reduced from 8.1 in the first configuration to 1.2 in the fourth configuration. As shown in this case study, the LPMR reduction algorithm provides an effective guide for the design space exploration. It presents a minimum but enough hardware support to achieve the layered performance matching in four steps. It avoids blind hardware overprovision while accommodating application diversity.

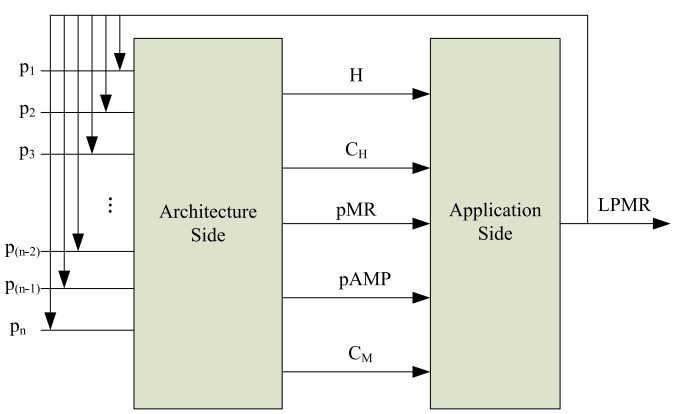

Fig. 9. Feedback-based LPM optimization on reconfigurable architecture ($p_i (i = 1, 2, \ldots, n)$ are architecture parameters).

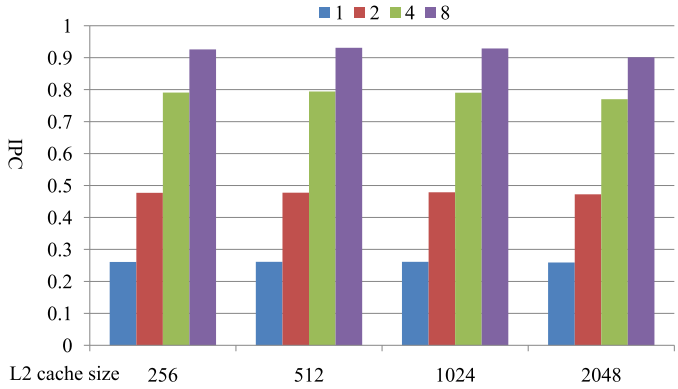

Fig. 10. The IPC values of wrf for different configurations of L2 cache size and L2 MSHR number.

The search for optimal architectural parameters can be guided by LPM. LPM quantifies the layered matching degree and thus decides which parameter should be optimized on demand. Fig. 9 shows how LPM algorithm works on reconfigurable architecture. With hardware configurability, architectural parameters are adjusted to reduce LPMR at each layer of a memory hierarchy [47]. Based on the feedback in terms of mismatching degree, an optimal architectural configuration is found quickly. Compared to traditional design space exploration, the LPM algorithm has two merits. First, its optimization is not based on global IPC, but based on local metric LPMR. Therefore, the measurement and feedback are completed much sooner. Second, its optimization is a focused one, finding an architectural configuration that eliminates the mismatching. Therefore, the optimization can be finished with much fewer iterations.

To fully understand the way how these parameters take effect, L2 cache size and L2 MSHR number, will be taken as two primary leverages for performance optimization. Fig. 10 shows the IPC values of the wrf benchmark for different configurations of L2 cache size and L2 MSHR number. Fig. 11 shows the values of the $bzip2$ benchmark. The horizontal axis shows the different values of L2 cache size from 256 KB to 2 MB. L2 MSHR number is set to be 1, 2, 4 or 8. L2 cache size impacts how many data items can be held by the LLC. In this sense, LLC cache is a traffic filter of the total number of off-chip memory accesses. LLC MSHR number impacts the upper bound of memory level parallelism, which refers to the maximum number of concurrent memory accesses issued simultaneously. In this sense, LLC MSHR number is a traffic limiter of the parallelism of off-chip memory accesses.

The wrf is a floating point benchmark, which does weather modeling from scales of meters to thousands of kilometers. The test case is from a 30 km area over 2 days. It is seen that wrf is sensitive to data concurrency rather than LLC cache size. For wrf, only 256 KB of LLC is enough, but 8 LLC MSHRs need to be allocated. In comparison, bzip2 is sensitive to LLC cache size rather data concurrency. The bzip2 needs only one LLC MSHR but 2 MB LLC.

In running the LPM algorithm, until the LPMR meets the requirement, the LLC capacity and LLC MSHR are allocated gradually for an arbitrarily given application. When the LLC

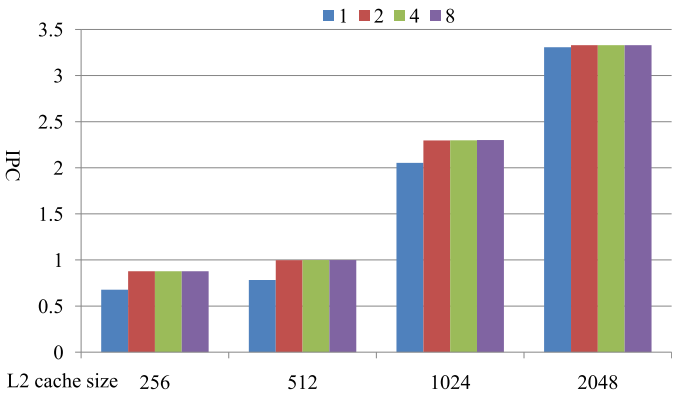

Fig. 11. IPC values of bzip2 for different configurations of L2 cache size and L2 MSHR number.

capacity is not large enough to hold the working set, the LLC misses would occur frequently, and the memory request rate would be high. As a result, LPMR would be large and mismatch exists. Taking bzip2 for example, when only 256 KB is available, 891K misses occur. However, when 2 MB is allocated, only 4K misses exist. Therefore, for applications such as bzip2, LLC miss rate has high impact on LPMR.

In another dimension, for applications such as wrf, data concurrency impacts LPMR. Taking wrf for instance, when only one LLC MSHR is allocated, the number of blocking cycles is 1.43×10^8. However, when 8 MSHRs are available, the blocking cycles is reduced to 3.2×10^4. Because the requests have already been issued from processors, the request rate is given. But due to the blocking cycles occurred on MSHR, the memory lacks the ability to provide replies timely, and therefore the reply rate becomes increasingly lower and the LPMR is increasingly larger. In this example, it is seen that the allocation of LLC MSHRs is vital for performance, because the total MSHRs are limited and different allocation schemes bring quite large performance difference.

8.3 Case Study III: LPM Optimization on Heterogeneous Main Memories

This section shows an example of LPM optimization on multicore architecture with heterogeneous main memories that are called Non-Uniform Memory Access architecture

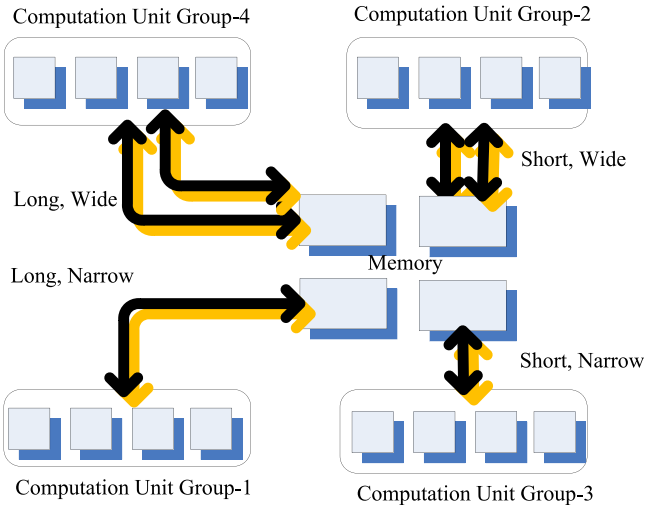

Fig. 12. Heterogeneous memory system of chip multiprocessors.

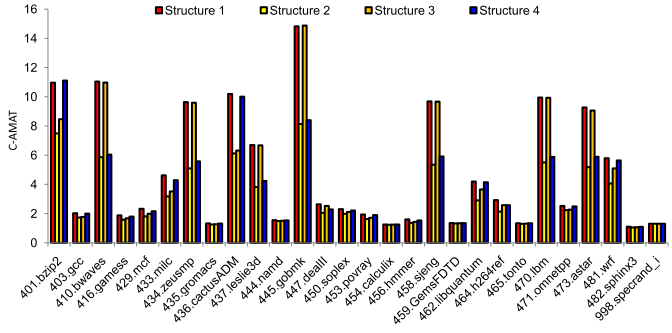

Fig. 13. C-AMAT values of heterogeneous structures (smaller is better).

(NUMA). As shown in Fig. 12, in case study II, processor group 1 and 4 are common computing units with long data access paths, which means that single memory access will cost hundreds of cycles. On the other hand, computing unit group 2 and 3 are processors under memory (PUM), which have less single access latencies by virtue of their close physical connection to a stacked set of memory dies. Computing unit group 4 and 2 can achieve higher memory bandwidth based on wider wire connections.

There exist four different groups of computing units for the diverse applications to match. We set δ as $T_1 \times 50\%$. We measure $LPMR_1$ and $LPMR_2$ on each of the different computing units. Fig. 13 shows the different C-AMAT values with the four different structures on four different group of computing units. The C-AMAT values of structure 2 (PUM) are the smallest, while those of structure 1 (common) are the largest.

The values in Fig. 13 can be obtained offline or online. The offline method is straightforward. Applications are run on different structures in turn, and the results are used for future runs. The advantage of offline is that the data is accurate, while the disadvantage is the need to run four times for each application. Alternatively, the online method can be used to estimate the performances of other structures at runtime. We adopt the latter approach in our simulations.

The way we assign applications to the groups affects the achieved system performance. For example, as shown in Fig. 13, 445.gobmk has similar C-AMAT values on structure 2 and 4, while 401.bzip2 has quite different C-AMAT values on structure 2 and 4. The optimal mapping for the two applications may be different. This observation is confirmed by our experiments: the optimal location for 401.bzip2 is group 2, while that for 445.gobmk is group 4.

With Eqs. (44) and (50), H_{sp} can be transformed into a function of LPMR. Intuitively, to prevent the memory stall, applications with high mismatch are assigned to PUM processor groups (#2 and #3), and assign the others to conventional processor groups (#1 and #4).

To automatically minimize the memory stall time, the LPM algorithm is implemented to provide a semi-optimal solution under a heterogeneous memory system environment and is referred to as the hybrid-aware scheduling algorithm (Hybrid-SA). We measure LPMRs for each of the units, and then by calculating the requirement of each application's LPMR, the applications are assigned to the cores according to their main memory access needs, rather than allocating randomly.

As shown in Eq. (52). Harmonic Weighted Speedup, H_{sp}, is a metric that strikes a balance between throughput and

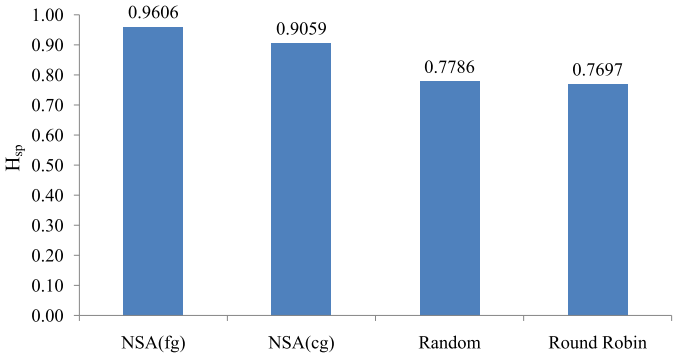

Fig. 14. H_{sp} values of different scheduling schemes on heterogeneous memory system (higher is better).

fairness [28]. When there is no contention, the value of H_{sp} is one. H_{sp} has been widely used to evaluate the schemes in a multiple program environment. In our study, to evaluate the scheduling effect, we also use this metric. As shown in Fig. 14, the throughput of multiple programs has been improved by 26 percent.

$$H_{sp} = \frac{N}{\sum_{i=1}^{N} \frac{IPC_{optimal}}{IPC_{real}}}. \quad (52)$$

9 Related Work

Improving memory system performance is a long-term goal of computer architects. In 1990, Sun and Ni proposed the memory-bounded parallel speedup model (Sun-Ni's law) [35] , which quantified how memory access influences system parallel speedup. However, while Sun-Ni's law shows the importance of memory system, it discusses mainly the application performance rather than memory system optimization. In 1994, the term memory wall was formally introduced based on the sequential latency model, Average Memory Access Time [48]. Note that in that time, memory level concurrency is rare, so the sequential latency makes sense. The AMAT model is recursive that can be used in each memory layer, so it can reflect the portion of latency consumed by each memory layer. Memory concurrency technologies have become common during the last two decades to improve memory system performance. The conventional sequential latency model is no longer adequate to characterize modern memory systems. Therefore, neither "low miss rate" nor "small AMAT" are the best performance criteria of modern memory systems.

LPM is a methodology to effectively utilize the available technologies and mitigate their negative interaction effects. While many optimizing technologies have been developed in past decades, the interaction among them is subtle and complex. Modern microprocessors utilize more than eighty percent of on-die transistors for caches rather than computing components, the on-chip hierarchy becomes increasingly dominant. The increasing resources are designed with lots of structures to make memory level concurrency ubiquitous. Meanwhile, to utilize caches, cache mapping, replacement, bypass and prefetch are long-standing hot topics of research. However, the considerable cache capacity and the concurrency provided by hardware are utilized without a matching perspective. Amdahl [23], Patterson [19], and Zhu [49] provided an alternative criterion of matching, respectively, in addition to ours [31]. We examine their criterions in the following.

Amdahl provides a rule of thumb that also known as Amdahl's balanced system law [23]. It is said that, a system needs a bit of IO per second for each instruction per second: about 8 MIPS per MBps. Here Million Instructions Per Second (MIPS) is the computing speed; Million Bytes Per Second (MBps) is the memory access speed. Although Amdahl's balanced system law considers the ratio of memory access speed and computing speed, it is a rule of thumb that is not accurate and cannot be adapted to different applications and architectures. As said in [23], Amdahl's balanced system law becomes more complex to interpret in multi-issue pipelined processors.

Williams, Waterman, and Patterson presented Roofline model for the attainable system performance [19]. If an application is bounded by the peak computing speed, it is compute-bound. If it is bounded by the product of peak memory bandwidth and operational intensity, it is memory-bound. The purpose of Roofline is not to be very accurate but just insightful. LPM moves further and is more accurate than Roofline in three aspects. LPM can be used not only for main memory layer but also for any other layers of a memory hierarchy. LPM quantifies the latency-hidden effect that is highly important for modern out-of-order execution processors. LPM reveals that the requirement of matching of each memory layer is determined by the expected efficiency and the achieved latency-hidden effect.

Zhu, Xiao, et al proposed a balanced design with the maximum criterion for a supercomputer [49]. The main idea of the balanced design is the maximum criterion that providing maximum bandwidth and enabling maximum number of computing nodes that can concurrently access I/O systems. The maximum criterion highlights the importance of memory bandwidth and memory level concurrency. It is applied in supercomputer system level, so the memory bandwidth and concurrency can be added flexibly by increasing memory nodes. However, in multi-core chip level, the management of memory bandwidth and concurrency should be conducted in memory access level. The maximum criterion is a rough result that is obtained by estimation, intuition and experience, thus may not be optimal due to the lack of an accurate model for matching. In comparison, LPM presents the definition and quantification of matching, which will significantly facilitate the performance optimization of memory systems while the work [49] is a motivation example of LPM design.

Many individual optimizations can benefit from the guideline of LPM, including cache allocation (e.g., Intel CAT [4]) and memory scheduling (e.g., [38]). LPM provides an accurate matching criterion for the resource management. While we acknowledge the criterion provided by [26], we found it only utilizes locality and it can be drastically enhanced with a guide of LPM.

10 Conclusions and Future Work

Matching the data request rate with the data supply rate in a memory system is a known idea for memory system design. However, how to identify the needed supply rate at each layer of a memory hierarchy for an arbitrarily given application, how the matching at each layer of a memory hierarchy

influence the overall system performance, and how to identify mismatching and improve matching in a timely fashion remain elusive and need to be addressed. In this study, we propose a novel performance optimization method, the Layered Performance Matching method for memory system design optimization. LPM emphasizes the performance matching between the layers of a hierarchical memory system and considers the integrated impact of data concurrency and locality. It is based on a data-centric view of computing and can be used to diagnose and identify performance bottlenecks in locality and/or concurrency of data accesses at each layer of a memory hierarchy and for the overall memory system. The LPM method and its associated LPM algorithm have been supported with rigorous mathematical analyses and illustrated with case studies on an Intel Xeon processors and on GEM5 simulators. Experimental testing has confirmed and demonstrated the feasibility and ingenuity of the LPM method in memory system design and optimization.

The proposed LPM facilitates the balanced exploration of locality and concurrency of memory accesses and provides a systematic and fast approach to match the data access demands of the running application to the underlying memory system. This matching can be achieved by adjusting the configuration of the underlying memory system or through scheduling applications appropriately in a heterogeneous environment. The data access considers both data locality and concurrency. To achieve better LPM, data access locality of the application should correspond to the locality of the underlying memory system. Equally important, the concurrent data accesses of the application should match the concurrency of the underlying memory system. Layered matching leads to improved measurement and analysis of data access delays and to a more effective and efficient method for obtaining a balanced memory system design and optimization.

We believe that the architectural support for layered performance matching methodology is a promising trend, this work makes a case for this direction. Meanwhile, many interesting open issues need to be addressed, including (1) how to add labels or identifiers in the server architecture (i.e., labeled architecture [5], [6]) to differentiate multiple applications or/and simultaneous multithreading (SMT) in applying LPM, (2) how to correlate the LPM with the quality of service of memory system for a mix of latency-sensitive (soft real-time) tasks and latency insensitive (throughput) tasks, (3) how to collect the needed parameters on-line with small overhead to apply LPM in existing and future parallel file systems, (4) how to add minimal support in reconfigurable chips to implement LPM? (5) how to make OS directly run on a labeled server to support individual-level or/and group-level LPM, and (6) how to apply LPM in heterogeneous platforms (e.g., host+accelerators, hybrid memory (DRAM+NVM)).

Acknowledgments

This work is supported by National Science Foundation of China (No. 61772497 and No. 61521092), State Key Laboratory of Computer Architecture Foundation (No. CARCH2601), National Key R&D Program of China (No. 2016YFB1000201), National Science Foundation of US (CCF-1536079, CNS-1162540 and CCF-0937877).

References

[1] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, and A. Basu, et al., "The gem5 simulator," *ACM SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, 2011.

[2] K. J. Min, D. H. Yoon, D. Sunwoo, and M. Sullivan, "Balancing DRAM locality and parallelism in shared memory CMP systems," in *Proc. IEEE Int. Symp. High-Perform. Comput. Archit.*, 2012, pp. 1–12.

[3] S. Phadke and S. Narayanasamy, "MLP aware heterogeneous memory system," in *Proc. Des. Autom. Test Eur. Conf. Exhibition*, 2011, pp. 1–6.

[4] K. T. Nguyen, "Intel CAT. Intel," 2018. https://software.intel.com/en-us/articles/introduction-tocache-allocation-technology.

[5] Y. G. Bao and S. Wang, "Labeled von neumann architecture for software-defined cloud," *J. Comput. Sci. Technol.*, vol. 32, no. 2, pp. 219–223, 2017.

[6] J. Ma, X. Sui, N. Sun, Y. Li, Z. Yu, B. Huang, T. Xu, Z. Yao, Y. Chen, and H. Wang, "Supporting differentiated services in computers via Programmable Architecture for Resourcing-on-Demand (PARD)," in *Proc. 20th Int. Conf. Archit. Support Program. Lang. Operating Syst.*, 2015, pp. 131–143.

[7] K. A. Yelick, "Ten ways to waste a parallel computer," in *Proc. 36th Annu. Int. Symp. Comput. Archit.*, 2009, pp. 1–1.

[8] 2019. [Online]. Available: http://www.spec.org/cpu2006/Docs/429.mcf.html.

[9] J. D. Mccalpin, "Memory bandwidth and machine balance in high performance computers," *IEEE Tech. Committee Comput. Archit. Newslett.*, vol. 2, no. 1925, pp. 1–7, 1995.

[10] S. Thakkar, P. Gifford, and G. Fielland, "The balance multiprocessor system," *IEEE Micro*, vol. 8, no. 1, pp. 57–69, Feb. 1988.

[11] J. Dongarra, K. London, S. Moore, P. Mucci, and T. Dan, "Using PAPI for hardware performance monitoring on linux systems", in *Proc. of the Con. on Linux Clusters*. The HPC Revolution: Urbana, Illinois, USA, 2009.

[12] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, and J. Mellor-Crummey, et al., "HPCTOOLKIT: Tools for performance analysis of optimized parallel programs," *Concurrency Comput. Practice Experience*, vol. 22, no. 6, pp. 685–701, 2009.

[13] M. Burtscher, B. D. Kim, J. Diamond, and J. Mccalpin, "PerfExpert: An easy-to-use performance diagnosis tool for HPC applications," in *Proc. ACM/IEEE Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2010, pp. 1–11.

[14] Intel, Memory Latency Checker, 2017. https://software.intel.com/en-us/articles/intelr-memory-latency-checker

[15] S. Borkar and A. A. Chien, "The future of microprocessors," *Commun. ACM*, vol. 54, no. 5, pp. 67–77, 2011.

[16] Y. Chou, B. Fahs, and S. Abraham, "Microarchitecture optimizations for exploiting memory-level parallelism," in *Proc. 31st Annu. Int. Symp. Comput. Archit.*, 2004, vol. 32, pp. 76–87.

[17] P. J. Denning, "The working set model for program behavior," *Commun. ACM*, vol. 11, no. 5, pp. 323–333, 1968.

[18] P. J. Denning, "The locality principle," *Commun. ACM*, vol. 48, no. 7, pp. 19–24, 2005.

[19] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for multicore architectures," *Commun. ACM*, vol. 52, no. 4, pp. 65–76, 2009.

[20] R. A. Hankins, T. Diep, M. Annavaram, et al., "Scaling and characterizing database workloads: Bridging the gap between research and practice," in *Proc. 36th Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2003, pp. 151–164.

[21] N. Hardavellas, I. Pandis, R. Johnson, et al., "Database servers on chip multiprocessors: Limitations and opportunities," in *Proc. Biennial Conf. Innovative Data Syst. Res.*, 2007, pp. 79–87.

[22] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. Amsterdam, Netherlands: Elsevier, 2018.

[23] J. Gray and P. Shenoy, "Rules of thumb in data engineering," in *Proc. Int. Conf. Data Eng.*, 2000, 2000, vol. 42, pp. 3–10.

[24] J. Hoogeveen, J. K. Lenstra, and B. Veltman, "Preemptive scheduling in a two-stage multiprocessor flow shop is NP-hard," *Eur. J. Oper. Res.*, vol. 89, no. 1, 172–175, 1996.

[25] I. Hur and C. Lin, "Memory prefetching using adaptive stream detection," in *Proc. 39th Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2006, pp. 397–408.

[26] G. Kurian, O. Khan, and S. Devadas, "The locality-aware adaptive cache coherence protocol," *ACM SIGARCH Comput. Archit. News*, vol. 41, no. 3, pp. 523–534, 2013.

[27] Y.-H. Liu and X.-H. Sun, "Reevaluating data stall time with the consideration of data access concurrency," *J. Comput. Sci. Technol.*, vol. 30, no. 2, pp. 227–245, 2015.
[28] K. Luo, J. Gummaraju, and M. Franklin, "Balancing thoughput and fairness in SMT processors," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw.*, 2001, pp. 164–171.
[29] V. Sarkar, W. Harrod, and A. E. Snavely, "Software challenges in extreme scale systems," *J. Phys.: Conf. Series*, vol. 180, no. 1, pp. 1–12, 2009.
[30] J. P. Shen and M. H. Lipasti, *Modern Processor Design: Fundamentals of Superscalar Processors*. Long Grove, IL, USA: Waveland Press, 2013.
[31] Y. H. Liu and X. H. Sun, "LPM: Concurrency-driven layered performance matching," in *Proc. Int. Conf. Parallel Process.*, 2015, pp. 879–888.
[32] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," *ACM SIGARCH Comput. Archit. News*, vol. 30, no. 5, pp. 45–57, 2002.
[33] S. Somogyi, T. F. Wenisch, A. Ailamaki, and B. Falsafi, "Spatio-temporal memory streaming," *ACM SIGARCH Comput. Archit. News*, vol. 37, pp. 69–80, 2009.
[34] C. D. Spradling, "SPEC CPU2006 benchmark tools," *ACM SIGARCH Comput. Archit. News*, vol. 35, no. 1, pp. 130–134, 2007.
[35] X.-H. Sun and L. M. Ni, "Another view on parallel speedup," in *Proc. ACM/IEEE Conf. Supercomput.*, 1990, pp. 324–333.
[36] X.-H. Sun and D. Wang, "APC: A performance metric of memory systems," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 40, no. 2, pp. 125–130, 2012.
[37] X.-H. Sun and D. Wang, "Concurrent average memory access time," *Comput.*, vol. 47, no. 5, pp. 74–80, 2014.
[38] K. Van Craeynest , A. Jaleel, L. Eeckhout, P. Narvaez, and J. Emer, "Scheduling heterogeneous multi-cores through Performance Impact Estimation (PIE)," *ACM SIGARCH Comput. Archit. News*, vol. 40, no. 3, pp. 213–224, 2012.
[39] J. L. Henning, "SPEC CPU2006 benchmark descriptions," *ACM SIGARCH Comput. Archit. News*, vol. 34, no. 4, pp. 1–17, 2006.
[40] R. R. Curtin, J. R. Cline, N. P. Slagle, W. B. March, P. Ram, N. A. Mehta, et al., "MLPack: A scalable C++ machine learning library," *J. Mach. Learn. Res.*, vol. 14, no. 1, pp. 801–805, 2012.
[41] J. Dongarra, M. A. Heroux, and P. Luszczek, "High-performance conjugate-gradient benchmark: A new metric for ranking high-performance computing systems," *Int. J. High Perform. Comput. Appl.*, vol. 30, no. 1, pp. 9505–9511, 2015.
[42] P. R. Luszczek, D. H. Bailey, J. J. Dongarra, J. Kepner, R. F. Lucas, R. Rabenseifner, et al., "The HPC Challenge (HPCC) benchmark suite," in *Proc. ACM/IEEE Conf. High Perform. Netw. Comput.*, 2006, Art. no. 213.
[43] C. Bienia, S. Kumar, J. P. Singh, K. Li, "The PARSEC benchmark suite: Characterization and architectural implications," in *Proc. Int. Conf. Parallel Archit. Compilation Techn.*, pp. 72–81, vol. 43, 2008.
[44] Graph500: 2019. [Online]. Available: http://www.graph500.org/
[45] L. Wang, S. Zhang, C. Zheng, G. Lu, K. Zhan, X. Li, et al., "BigDataBench: A big data benchmark suite from Internet services," in *Proc. 20th IEEE Int. Symp. High Perform. Comput. Archit.*, 2014, pp. 488–499.
[46] D. Wang and X. Sun, "APC: A novel memory metric and measurement methodology for modern memory system," *IEEE Trans. Comput.*, vol. 63, no. 7, pp. 1626–1639, Jul. 2014.
[47] Y. Wu, Y.-J. Chen, T.-S. Chen, Q. Guo, and L. Zhang, "An elastic architecture adaptable to various application scenarios," *J. Comput. Sci. Technol.*, vol. 29, no. 2, pp. 227–238, 2014.
[48] W. A. Wulf and S. A. Mckee, "Hitting the memory wall: Implications of the obvious," *ACM SIGARCH Comput. Archit. News*, vol. 23, no. 1, pp. 20–24, 1995.
[49] M. Zhu, L. Xiao, L. Ruan, and Q. Hao, "DeepComp: Towards a balanced system design for high performance computer systems," *Frontiers Comput. Sci.*, vol. 4, no. 4, pp. 475–479, 2010.

Yuhang Liu received the PhD degree in computer science from Beihang University, Beijing, China. He is an associate professor in Institute of Computing Technology (ICT), Chinese Academy of Sciences (CAS). He has been a postdoctoral researcher with the Computer Science Department of Illinois Institute of Technology (IIT), Chicago. He is a member of CCF, ACM and IEEE. His research interests include high performance computing, computer architecture, data intensive computing and memory performance optimization. He is currently working on multi-core memory scheduling, partitioning and prefetching area to improve multi-core memory access bandwidth utilization and minimize access latency.

Xian-He Sun is a distinguished professor of computer science at IIT. He is the director of the Scalable Computing Software laboratory at IIT, an IEEE fellow, the past chairman of the Computer Science Department of IIT, and is a guest faculty in the mathematics and computer science division at the Argonne National Laboratory. Before joining IIT, he worked at DoE Ames National Laboratory, at ICASE, NASA Langley Research Center, and at Louisiana State University, Baton Rouge. His research interests include parallel and distributed processing, memory and I/O systems, software systems, and performance evaluation and optimization.