2015 44th International Conference on Parallel Processing

LPM: Concurrency-driven Layered Performance Matching

Yu-Hang Liu, Xian-He Sun
Department of Computer Science
Illinois Institute of Technology
Chicago, IL 60616
{yuhang.liu, sun}@iit.edu

***Abstract*—Data access has become the preeminent performance bottleneck of computing. In this study, a Layered Performance Matching (LPM) model and its associated algorithm are proposed to match the request and reply speed for each layer of a memory hierarchy to improve memory performance. The rationale of LPM is that the performance of each layer of a memory hierarchy should and can be optimized to closely match the request of the layer directly above it. The LPM model simultaneously considers both data access concurrency and locality. It reveals the fact that increasing the effective overlapping between hits and misses of the higher layer will alleviate the performance impact of the lower layer. The terms pure miss and pure miss penalty are introduced to measure the effectiveness of such hit-miss overlapping. By distinguishing between (general) miss and pure miss, we have made LPM optimization practical and feasible. Our evaluation shows the data stall time can be reduced significantly with an optimized hardware configuration. We also have achieved noticeable performance improvement by simply adopting smart LPM scheduling without changing the underlying hardware configurations. Analysis and experimental results show LPM is feasible and effective. It provides a novel and efficient way to cope with the ever-widening memory wall problem, and to optimize the vital memory system design.**

***Keywords*-Memory wall; data stall time; layered performance matching (LPM); data access concurrency; concurrent average memory access time (C-AMAT)**

I. Introduction

Compared to two decades ago when the term memory wall was introduced by Wulf and McKee [1], the landscape of computing has changed significantly. Many concurrency driven technologies have been proposed such as multi-port, multi-banked, pipelined and non-blocking cache, and data intensive applications have become common in diverse fields [2] [3]. However, the memory-wall problem remains. Today, data stall time contributes 50% to 70% of the total application execution time and is the most prominent performance bottleneck of computing systems [4] [5]. Recall data stall time is the amount of time the processor is blocked from issuing instructions because it is waiting for data. That is, it is the time when memory system cannot meet the demand of computation.

Hierarchical memory is a standard design for modern computers to ease the memory wall problem. The idea behind memory hierarchy is data locality. That is, data previously accessed will be used again in the near future (temporal locality), and the data near the previously accessed data are likely to be used next (spatial locality). The locality of hierarchical memory systems has been well studied in the literature [6] [7] [8] [9]. However, a modern memory system is not only supported by a memory hierarchy but also by various concurrency-driven technologies. The overall performance of a memory system is a combined effort of the memory hierarchy and concurrency. Understanding the integrated impact of data locality and concurrency is a must to fully utilize the potential of a modern memory system. In this study, we use the newly proposed concurrent average memory access time (C-AMAT) [10] [11] [12] [13] model as the key analysis tool to optimize modern memory systems. C-AMAT is an extension of the conventional AMAT [1] [7] [14] model by including data access concurrency into average memory access time.

During the past decades, many specific techniques have been proposed for optimizing distinct components or features of a memory system. Such kind of specific techniques can be deemed a toolkit or a technique pool. However, these methods are difficult and need not to be deployed simultaneously under a given limitation of complexity, heat, power and space. These methods may compete and conflict with each other. How to reach a global optimization state is unclear and elusive. Memory hierarchy is designed to mask the performance gap between computing and memory systems. A fundamental question for a hierarchical memory system should be: how to quantify the matching of a memory hierarchy with the underlying computing? This question of matching was not important for a memory hierarchy system two decades ago when the memory wall impact was not very significant, and data access locality was the only concern. As data access concurrency becomes increasingly prevalent and applications become increasingly data intensive, the balance and optimization between locality and concurrency becomes a vital performance factor; therefore, the layered performance matching (LPM) becomes a key factor influencing performance.

By accurately matching the performance of each adjacent layer of a memory hierarchy, the performance of the memory system can precisely match the performance of its computation capability, which is the motivation of LPM concept proposed in this study. The motivation of LPM is based on the following four observations.

- Each time a word is requested by a load or store operation, the word is carried in by a cache block (line) [14].

0190-3918 2015
U.S. Government Work Not Protected by U.S. Copyright
DOI 10.1109/ICPP.2015.97

Therefore, if the cache block can be reused multiple times, one physical data movement can support multiple data requests.

- Memory concurrency can mask data access delay, including hit-hit (all the data accesses are hits), hit-miss (or miss-hit, that is, at least a hit exists with misses), and miss-miss (all the data accesses are misses, and is termed as pure miss) overlapping [10] [11]. The hit-hit scenario increases data access bandwidth. The hit-miss scenario is the most interesting one and requires more attention. If we have misses, we should overlap the misses with hits since the hits in a layer can hide the penalties of the simultaneously occurring misses in the same layer. The miss-miss scenario, which is referred to as pure misses, will cause data stall. We should reduce pure misses. But, if we cannot avoid them, we should let them occur at the same time. Thus increasing the concurrency of pure misses.
- In general, programs have periodic behaviors, and their data access patterns are predictable [15]. With a set of lightweight counters, we are able to deploy proper optimization techniques to timely adapt to the underlying data access patterns of an application.
- Configurable hardware is becoming prevalent [16], which presents more optimization options for adapting memory system to dynamic data access patterns. Moreover, more and more application-aware software scheduling and partitioning approaches have been proposed. These progresses enable us to achieve LPM.

LPM facilitates both hardware and algorithms to simultaneously consider data locality and concurrency. The layered performance matching provides a new method to utilize and optimize memory systems. By surmounting the memory-wall via concurrency-driven layered performance matching in a memory hierarchy, this study makes the following contributions:

- The LPM model is presented to quantify the performance match degree between the layers in a memory hierarchy. The methodology for the online measurement of request and supply is also proposed.
- For the first time, with minimal number of parameters, a formal mathematical model is proposed to quantify the impact of layered performance mismatch on data stall time. The model not only facilitates understanding performance but also presents opportunities for optimization. In this manner, our model presents guidance on when and how to use existing locality and concurrency driven techniques collectively.
- The LPM optimization algorithm is developed to reduce the mismatch and then mitigate the incurred data stall time. The algorithm is application-aware since all the parameter values needed by the models can be measured online. With LPM optimization, data stall time can be reduced significantly.
- Aided by the LPM algorithm, case studies of performance optimization were conducted. They confirm the potential of LPM. By simply matching layered performance in a heterogeneous multicore system, system throughput has been improved significantly even without hardware configurability. These case studies confirm the correctness and practical value of the layered performance matching approach proposed in this study.

The remainder of this paper is organized as follows. Section II introduces the state-of-the-art memory analytical model, C-AMAT. Section III and IV, describe the concurrency-driven LPM model and present the LPM algorithm, respectively. Section V presents LPM optimization case studies to reduce the data stall time. Section VI reviews related work. Finally, Section VII concludes this study and discusses potential future work.

II. The C-AMAT Model

In this paper, unless otherwise stated, a memory system is the whole memory hierarchy rather than only the main memory. The conventional AMAT formula for a memory hierarchy is shown in Eq. (1) [1] [7] [14], 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 does not consider the concurrency of memory accesses, in either the hit or the miss section of the formula. However, in an out-of-order processor when a miss occurs, other instructions can be executed while the memory system is serving the miss. This allows multiple outstanding reads and writes to co-exist at a given time in the memory system, depending on the underlying hardware support. Therefore, some of the memory access latencies can be hidden.

$$AMAT = H + MR \times AMP \tag{1}$$

To cover the concurrent read and write properties of modern memory systems, the C-AMAT model is proposed in Eq. (2) [10]. The first parameter H is the same as that in AMAT. The second parameter C_H represents hit concurrency; the third parameter C_M (capital M) represents the pure miss concurrency while the conventional miss concurrency is referred to as C_m (little m). C_H can be contributed by caches with multi-port, multi-bank or pipelined structures. C_M can be contributed by non-blocking cache structures. In addition, out-of-order execution, multi-issue pipeline, simultaneous multi-threading (SMT), 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 (MR). A pure miss here means that a miss contains at least one miss cycle that does not have any hit access activity. pAMP is the average number of pure miss cycles per pure miss access.

$$C\text{-}AMAT = \frac{H}{C_H} + pMR \times \frac{pAMP}{C_M} \tag{2}$$

Pure miss is an important concept introduced by C-AMAT. Based on the fact that not all the cache misses will cause processor stall, we introduced pure miss. Fig. 1 illustrates the pure miss concept. There are five different memory accesses in Fig. 1, and each access contains three cycles for cache hit operations. If it is a miss, additional miss penalty cycles will be required. The number of miss penalty cycles is uncertain, depending on where the missed data can be obtained and contention impact during the data access. 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.

According to Eq. (2), C-AMAT is cycles out of 5 accesses or 1.6 cycles per access; whereas by Eq. (1) AMAT is $3+0.4\times 2$ or 3.8 cycles per access. The difference between C-AMAT and AMAT is the contribution of concurrent data access. In this example, concurrency has doubled memory performance. In Fig. 1, there are 4 hit phases, namely Hit phase 1, 2, 3, 4, which contain 2, 4, 3, 1 concurrent hit cache assesses with lasting cycle 2, 1, 2, 1, respectively. Therefore, $C_H = 2 \times 2/6 + 4 \times 1/6 + 3 \times 2/6 + 1 \times 1/6 = 5/2$. And there is only one pure miss phase with 1 pure miss concurrency which lasts for 2 cycles. Therefore $C_M = 1 \times 2/2 = 1$; pAMP = 2/1 =2; pMR = 1/5. Thus formula (2) is equal to

$$C\text{-}AMAT = \frac{H}{C_H} + pMR \times \frac{pAMP}{C_M} = \frac{3}{5/2} + \frac{1}{5} \times \frac{2}{1} = 1.6$$

Please notice that the contribution of C-AMAT is not its measurement. The measurement can be obtained directly through APC (Access Per memory active Cycle) as shown in Eq. (3) [12] [13]. The measurement of C-AMAT does not depend on the measurement of its five parameters. The value of the parameters is in performance analysis and optimization. The invaluable contribution of C-AMAT is that it provides a unified formulation to capture the joint performance impact of locality and concurrency.

$$C\text{-}AMAT = \frac{1}{APC} \tag{3}$$

C-AMAT contains AMAT as a special case where memory concurrency does not exist. C-AMAT provides a means to evaluate and optimize the five performance parameters, individually or in combination. It provides a tool for design optimization.

The sixteen memory-system optimization mechanisms summarized by Hennessy and Patterson can be used to reduce one or more of the three parameters of AMAT [14]. Similar to AMAT, reducing C-AMAT can be achieved by optimizing its five parameters. In particular, it can be achieved by increasing C_H and C_M, and decreasing H, pMR, and pAMP. The five parameters in C-AMAT present five dimensions for memory system optimization. Except H, all the parameters are new directions that AMAT has not presented.

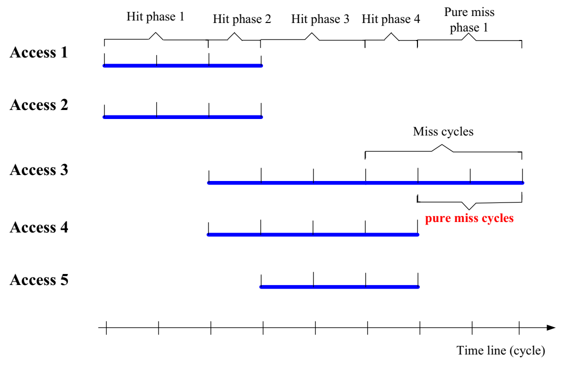

Fig. 1. A demo of C-AMAT and pure miss

Readers who are familiar with AMAT may recall that the average miss penalty of AMAT can be extended recursively to the next layer of a memory hierarchy. This recursiveness is true for C-AMAT as well. Eq. (4) shows the recurrence relation. Here, $C\text{-}AMAT_1$ is the L1 C-AMAT and $C\text{-}AMAT_2$ is the L2 C-AMAT. Unless indicated explicitly, the default C-AMAT is $C\text{-}AMAT_1$. Eq. (4) illustrates some interesting and valuable properties of concurrent data accesses: the impact of $C\text{-}AMAT_2$ can be mitigated by concurrency, in terms of pMR_1 and η_1. This potential for penalty mitigation is the theoretical foundation and motivation of the layered matching mechanism proposed in this research.

$$C\text{-}AMAT_1 = \frac{H_1}{C_{H_1}} + pMR_1 \times \eta_1 \times C\text{-}AMAT_2 \tag{4}$$

Where

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

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

$$\eta_1 = \frac{pAMP_1}{AMP_1} \times \frac{C_{m_1}}{C_{M_1}}$$

Pure-miss is more closely related to data stall time, compared to conventional miss. All the conventional misses of L1 will occur on L2. However, only a part of the misses are pure misses, and only pure misses affect the data stall time. Pure misses can be characterized by C_M and pAMP, while conventional misses can be featured by C_m and AMP. Therefore, the measurable parameter η_1 has a physical representation, which reflects the difference between pure miss and conventional miss [10]. A part of conventional miss penalty can be masked by hit activities. The parameter η_1 indicates these contributions. The impact of $C\text{-}AMAT_2$ toward the final $C\text{-}AMAT_1$ can be trimmed by pMR_1 and η_1. In a similar fashion as Eq. (4), C-AMAT can be further extended to the next layer of the memory hierarchy as well.

III. The LPM Model

We divide this section into two subsections. Firstly, we reveal the relationship between concurrent average memory access time (C-AMAT) and data stall time. Then, we derive the relationship between layered performance matching ratio (LPMR) and data stall time.

A. Impact of C-AMAT on Data Stall Time

The execution time of a computer processor consists of two parts [14]: processor busy time and data stall time. Here the processor busy time is the time when the processor is occupied executing the user program. This time includes the useful working time, as well as functional unit stalls due to data and control hazards. The data stall time is the time when the processor is stalled waiting for memory reference. This time consists of the access delay, contention delay, and, in multi-thread cases, the latency due to cache coherency and consistency. Eq. (5) is the classic formulation of the CPU-time in terms of these two components of time [14] [17].

$$\begin{aligned}&CPU\text{-}time\\ &= IC \times (CPI_{exe} + Data\text{-}stall\text{-}time) \times Cycle\text{-}time\end{aligned} \tag{5}$$

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

In an in-order processor, when an access miss occurs, the processor waits for the fetched data before continuing. This can result in a data stall lasting several cycles, depending on where in the memory hierarchy the data resides. Eq. (6) is the conventional data stall time formula based on AMAT [1] [14].

$$Data\text{-}stall\text{-}time = f_{mem} \times AMAT \tag{6}$$

With the same reason discussed in Section II for AMAT, Eq. (6) does not hold for out-of-order processors, and cannot reflect the concurrency in the modern complex memory systems.

Eq. (7) presents the relationship between the data stall time and the data access delay C-AMAT.

$$\begin{aligned}&Data\text{-}stall\text{-}time\\ &= f_{mem} \times C\text{-}AMAT \times (1 - overlapRatio_{\text{c-m}})\end{aligned} \tag{7}$$

Here, f_{mem} is the portion of the instructions that access memory; as shown in Eq. (8), $overlapRatio_{\text{c-m}}$ 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 $overlapRatio_{\text{c-m}}$ by enabling computation to continue while memory access is being conducted.

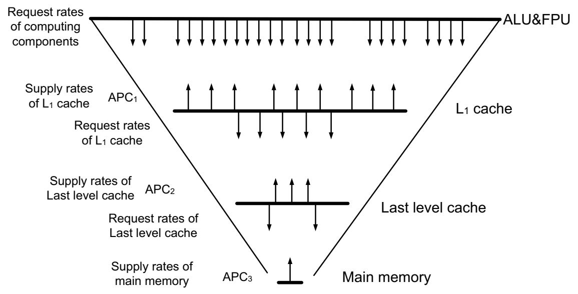
Request rates of computing components
ALU&FPU
Supply rates of L1 cache
APC1
Request rates of L1 cache
L1 cache
Supply rates of Last level cache
APC2
Request rates of Last level cache
Last level cache
Supply rates of main memory
APC3
Main memory

Fig. 2. The request-reply LPM model

$$overlapRatio_{\text{c-m}} = \frac{overlapCycles_{\text{c-m}}}{T_{memAcc}} \tag{8}$$

Eq. (7) holds regardless memory concurrency is involved or not. The formal proofs of Eq. (7) is given in [17].

B. Impact of LPM on Data Stall Time

We know the layered (memory) performance mismatch affects the memory system performance and, therefore, drags down the system CPU performance. In this subsection, we quantify these effects and, therefore, provide a theoretical foundation for possible performance optimization.

To optimize the performance of a memory system, we need to match the performance at each layer of the hierarchical memory system as closely as possible.

Fig. 2 illustrates the Layered Performance Matching (LPM) model of a memory hierarchy. For the sake of brevity, L2 is taken as the LLC in this study. The extension to additional cache levels is straightforward.

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

The ratios in the LPM model are the ratios of request rate and supply rate between any two layers. These ratios usually are greater than 1 due to the memory wall problem. Making these ratios as closely to 1 as possible is a way to ease memory wall effect.

As a result of matching performance at each layer of a memory hierarchy, the performance of the memory system can match the performance of the computing as closely as possible. The following equations are the definitions of the layered performance matching ratios (LPMR) that are abbreviated as $LPMR_1$, $LPMR_2$, and $LPMR_3$, respectively. We assume L2 is the last level cache in this study.

$$LPMR(ALU\&FPU, L1) = \frac{Request\ rate\ from\ ALU\&FPU}{Supply\ rate\ by\ L1\ cache}$$

$$LPMR(L1, LLC) = \frac{Request\ rate\ from\ L1\ cache}{Supply\ rate\ by\ LLC}$$

$$LPMR(LLC, MM) = \frac{Request\ rate\ from\ LLC}{Supply\ rate\ by\ main\ memory}$$

Note that supplies are activated by requests, the supply rate by a lower layer is impossible to be greater than the request rate from an upper layer. Therefore, LPMR ≥ 1. The optimal state occurs when LPMR = 1.

The supply rates of different layers in the memory hierarchy can be denoted by APC values of corresponding layers [12] [13]. Compute intensity multiplied by the memory access frequency equals the memory access intensity perceived by the L1 cache. Compute intensity can be denoted by IPC_{exe}, which is the reciprocal of CPI_{exe}. Therefore,

$$Request\ rates\ of\ ALU\ and\ FPU = IPC_{exe} \times f_{mem}$$

$$Request\ rates\ of\ L1\ cache = IPC_{exe} \times f_{mem} \times MR_1$$

$$Request\ rates\ of\ LLC = IPC_{exe} \times f_{mem} \times MR_1 \times MR_2$$

LPM is a concurrency-driven matching method in a memory hierarchy. It focuses on data access concurrency to optimize the matching between request and supply at each memory layer. In addition to the various data access needs of the applications and the concurrency-driven delay hidden technique, the cache line structure produces more data replies. One cache line may be taken as the common reply for numerous requests so that multiple accesses can be completed with the same cache line. According to Eq. (3), the supply traffic in each layer can be transformed from APC to C-AMAT for performance analysis and optimization. Therefore, in terms of C-AMAT, the three LPMRs can be expressed as in Eq. (9), (10), and (11), respectively.

$$LPMR_1 = \frac{C\text{-}AMAT_1 \times f_{mem}}{CPI_{exe}} \tag{9}$$

$$LPMR_2 = \frac{C\text{-}AMAT_2 \times f_{mem} \times MR_1}{CPI_{exe}} \tag{10}$$

$$LPMR_3 = \frac{C\text{-}AMAT_3 \times f_{mem} \times MR_1 \times MR_2}{CPI_{exe}} \tag{11}$$

With Eq. (7) and (9), we get the relationship between data stall time and LPM in layer one shown in Eq. (12).

$$\begin{aligned} &Data\text{-}stall\text{-}time \\ &= CPI_{exe} \times (1 - overlapRatio_{c\text{-}m}) \times LPMR_1 \end{aligned} \tag{12}$$

Combining Eq. (7) and (4), we derive

$$\begin{aligned} Data\text{-}stall\text{-}time = (&\frac{H_1}{C_{H_1}} + pMR_1 \times \eta_1 \times C\text{-}AMAT_2) \times f_{mem} \\ &\times (1 - overlapRatio_{c\text{-}m}) \end{aligned}$$

Then due to the expression of $LPMR_2$ in Eq. (10), we get

$$\begin{aligned} Data\text{-}stall\text{-}time = (&\frac{H_1}{C_{H_1}} + pMR_1 \times \eta_1 \times \frac{CPI_{exe} \times LPMR_2}{f_{mem} \times MR_1}) \\ &\times f_{mem} \times (1 - overlapRatio_{c\text{-}m}) \end{aligned}$$

Therefore,

$$\begin{aligned} Data\text{-}stall\text{-}time = (&\frac{H_1}{C_{H_1}} \times f_{mem} \\ &+ CPI_{exe} \times \eta_1 \times \frac{pMR_1}{MR_1} \times LPMR_2) \times (1 - overlapRatio_{c\text{-}m}) \end{aligned}$$

We define η as follows.

$$\eta = \frac{pAMP_1}{AMP_1} \times \frac{C_{m_1}}{C_{M_1}} \times \frac{pMR_1}{MR_1}$$

Therefore, we can get Eq. (13).

$$\begin{aligned} Data\text{-}stall\text{-}time = (&\frac{H_1}{C_{H_1}} \times f_{mem} \\ &+ CPI_{exe} \times \eta \times LPMR_2) \times (1 - overlapRatio_{c\text{-}m}) \end{aligned} \tag{13}$$

Eq. (13) shows the relationship between LPM in L2 layer and data stall time. The parameter η is a positive value and is less than one, which is the concurrency and locality effective factor. Note that η contains both the concurrent hit effect in the upper layer and the miss penalty incurred in accessing the lower layer. The effect of mismatch $LPMR_2$ can be eased by η. When η is close to zero, the effect of concurrency on miss penalties is significant. Therefore, once η is close to zero, the impact of layered performance mismatch will be small. The rationale of LPM optimization is to combine the effort of locality and concurrency to maximize the hidden effect and overlapping effect at each layer of a memory hierarchy, where the overlapping effect includes hit-hit overlapping, hit-miss overlapping, and miss-miss overlapping. Note that hit-miss overlapping also has hidden effect. In next section, we will give the algorithm to improve the degree of matching of the layered performance to optimize the memory subsystem performance.

IV. The LPM Algorithm

Eq. (12) and (13) provide the baseline for LPM optimization to achieve our final goal of minimizing data stall time.

Data stall time is 50% to 70% of the total application running time [4] [5], severely affecting application performance. That means data stall time is 1 to 2.3 times of pure computing time without any data stall. We consider any data stall time that is less than $\Delta\%$ of pure computing time as a "minimal data stall time". The goal of LPM is to achieve a minimal data stall time. For example, the reduction of data stall time from 230% to 1% of pure computing time implies that the memory performance is improved up to two hundred and thirty (230) times. As shown in our case studies, the "1%" condition is appropriate and can be met on reconfigurable architectures that have a huge design space. More stringent condition than "1%" may cause unaffordable optimization cost and have a minimal return.

While the "1%" requirement is achievable, it might be hard to achieve in some cases. For those cases, we can relax the requirement to "10%" or even higher. The optimizations for the two different requirements will be referred to as fine-grained (fg) and coarse-grained (cg) optimization, respectively.

In the following, we derive the conditions for $LPMR_1$ and $LPMR_2$ to achieve minimal data stall. From Eq. (12), we get Eq. (14).

$$LPMR_1 \le \frac{\Delta\%}{1 - overlapRatio_{c\text{-}m}} \tag{14}$$

From Eq. (13), to satisfy the $\Delta\%$ condition, we have Eq. (15).

$$LPMR_2 \le \frac{1}{\eta} \times \left(\frac{\Delta\%}{1 - overlapRatio_{c\text{-}m}} - \frac{H \times f_{mem}}{C_H \times CPI_{exe}}\right) \tag{15}$$

When Eq. (14) and (15) are met, optimization in the corresponding layer can be finished. The thresholds for $LPMR_1$ and $LPMR_2$, denoted as T_1 and T_2, are the right-side of Eq. (14) and (15).

Fig. 3 shows the algorithm for layered performance matching optimization. Initially, the mismatching information is measured and collected. Optimizations are necessary only when the $LPMR_1$ is large. If $LPMR_1$ is larger than T_1, we need to optimize the L1 layer and L2 layer simultaneously, or only optimize the L1 layer if $LPMR_2$ is not larger than T_2. In both cases, the optimization of the L1 layer is required.

Fig. 4 shows the detecting system of LPM. We call it C-AMAT analyzer. The Hit Concurrency Detector (HCD) counts the total hit cycles and records each hit phase in order to calculate the average hit concurrency (C_H). The HCD also contacts the Miss Concurrency Detector (MCD) whether a current cycle has a hit access or not. The MCD is a monitor unit which counts the total number of pure miss cycles and records each pure miss phase in order to calculate the average miss concurrency (C_M), pure miss rate, and pure miss penalty. With the information provided by the HCD, the MCD is able to determine whether a cycle is a pure miss cycle, and whether a miss is a pure miss. With the miss information, the pure miss rate (pMR) and average pure miss penalty (pAMP) can be calculated. Since these parameters can be measured at each layer of a memory hierarchy, C-AMAT also can be measured at each layer.

LPMR Reduction Algorithm

```
// Initially measure the metrics
  For each application or thread, measure the LPMRs in a memory hierarchy
  Get the threshold T_1 and T_2 according to Eq. (15) and (16)
Begin Do
  // LPM optimization loop
    // Case I when both L1 and L2 layer need an optimization
    While (LPMR_1 > T_1 and LPMR_2 > T_2) Do
        Optimizing at L_1 layer and L_2 layer,
        Update all the metrics (LPMR_1 , LPMR_2, T_1 and T_2)
    Until (LPMR_1 ≤ T_1 or LPMR_2 ≤ T_2)
    // Case II when only L1 layer needs an optimization
    While (LPMR_1 > T_1 and LPMR_2 ≤ T_2 ) Do
        Optimizing at L1 layer
        Update all the metrics (LPMR_1 , LPMR_2, T_1 and T_2)
    Until (LPMR_1 ≤ T_1)
    // Case III when no layer needs to optimize and overprovision may need to reduce
    // δ is a positive value
    While (LPMR_1 + δ < T_1) Do
        Reduce hardware overprovision
        Update all the metrics (LPMR_1, LPMR_2, T_1 and T_2)
    Until (LPMR_1 ≥ T_1 - δ )
    // Case IV when no layer needs to optimize and no overprovision needs to reduce
    If (T_1 ≥ LPMR_1 ≥ T_1 - δ)
        End the algorithm
    Endif
Until (End)
```

Fig. 3. Pseudo-code of the LPM algorithm

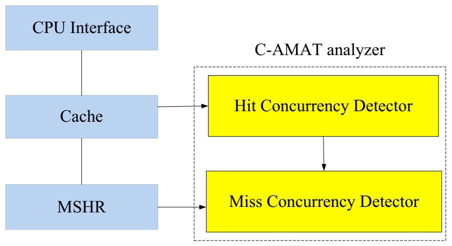

Fig. 4. The C-AMAT Detecting System

Only when $LPMR_1$ is more than T_1 and $LPMR_2$ is more 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 should be stressed in the diagram of Fig. 3 when $LPMR_1 < T_1$. Note that δ is a positive value. The value of δ can be set according to the contention status among the applications for the shared underlying hardware. If $LPMR_1$ is smaller than its threshold value T_1 by more than delta, we know that hardware has been over provided. This step is optional, but we include it in the algorithm for completeness. Therefore, we present the method for matching the application data access requirement with the underlying memory system with an optimized effort in a cost-efficient manner.

Note that all the steps are conducted on-line to adapt to the dynamic behavior of the applications. The LPMR reduction 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 implementation of a reconfiguration operation or a scheduling operation.

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, 96% of the burst data access patterns can be perceived and processed timely. When the interval size is set to 20 cycles, 89% 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% of the burst data access patterns can be perceived and processed timely.

Generally, a matching of A and B, can be achieved by reorganizing A to match B, or reorganizing B to match A, or simultaneously reorganizing A and B to agree to each other. In practice, for layered performance matching, if the architecture configuration is A, and the application data access pattern is B. The method optimizing A is the hardware approach, the method optimizing B is the software approach, and the method optimizing both A and B is the mixed approach.

For hardware approach, the architecture configuration can be adapted on-line to the data access patterns of the applications. On the other hand, for software approach, the architecture configuration is fixed but with heterogeneity, and through scheduling can also achieve a better match of the underlying memory systems for better performance. The scheduling can be implemented in two manners. We can schedule across heterogeneous processors to allocate application data access pattern to its suitable underlying hardware. Alternatively, we can reorganize data accesses at the memory controllers to reshape application data access pattern to adapt to its underlying hardware. Both of them belong to the optimizing B method.

The LPM algorithm in Fig. 3 is general and can be implemented in both hardware and software or a mixed approach. If the hardware can be improved, the optimization can be achieved 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 met via pure software optimization.

As will be shown in the next section, the LPM algorithms can facilitate architecture design space exploration to avoid exhausting search and over providing hardware parallelism, and also can facilitate software scheduling to achieve application awareness and heterogeneity awareness.

The optimization potential of hardware approach and software approach is determined by their available design space size. In most cases, the design space of the hardware approach is much larger than that of the software approach, and thus the former has a much larger room for performance improvement and the condition of T_1 and T_2 may not be met via pure software optimization.

The optimization in the hardware approach is through directly increasing hardware parallelism while the optimization in the software approach is through scheduling to find a best match.

In next section, two representative case studies for LPM optimization are conducted. Case Study I uses hardware approach on reconfigurable architectures. Case Study II deploys software approaches in heterogeneous environments.

V. Case Studies

The modern cycle-accurate simulator GEM5 [18] is used to provide an appropriate full system performance simulation. A detailed out-of-order CPU model and DRAMSim2 modual in the GEM5 simulator were adopted to achieve the most accurate simulation results. The C-AMAT analyzer is an add-in component of the simulation, where detectors for both hits and misses are implemented. SPEC CPU2006 benchmark suite [19] is used in our simulations. The benchmarks are compiled using GCC 4.3.0 and the -O3 optimization level, and are executed using reference input sets. For each benchmark, 10 billion instructions were sampled [15].

A. Case Study I: LPM Optimization on Reconfigurable Architecture

Based on the C-AMAT definition and LPM definition, optimal hardware configurations can be found for each program to mitigate the memory-wall impact.

There are dozens of parameters in computer architecture, and each parameter can be set to different values. For brevity of discussion, only six architecture parameters are explored, that is, MSHR numbers, IW size, ROB size, L1 cache port number, L1 cache interleaving, and pipeline issue width. Provided that each parameter can be set with 10 different values, the design space size is 10^6. There are one million different configurations, thus making the design space quite large. Exhausting search is not an option, and an LPM optimization algorithm becomes a must.

TABLE I
LPMRs under configurations with incremental parallelism

Configuration	A	B	C	D	E
Pipeline issue width	4	4	6	8	8
IW size	32	64	64	128	96
ROB size	32	64	64	128	96
L1 cache port number	1	1	2	4	4
MSHR numbers	4	8	16	16	16
L2 cache interleaving	4	8	8	8	8
$LPMR_1$	8.1	6.2	2.1	1.2	1.4
$LPMR_2$	9.6	9.3	3.1	1.6	1.9
$LPMR_3$	6.4	8.1	5.8	2.3	2.6

Under five configurations A to E, Table I shows the average LPMRs for 410.bwaves benchmark from SPEC CPU 2006. Let us cruise the general LPM algorithm Fig. 3 in the architecture design space exploration. The goal of the optimization is to keep the data stall time per instruction within 1% $\times$ CPI_{exe}.

Firstly, in current time interval, LPMRs are measured for each application. Table I states the LPMRs in a memory hierarchy under configuration A. 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 I. However, the mismatches are still higher than their thresholds. Then we continue reduction and transform to configuration C. The mismatch now in L2 layer is already less than its threshold value. Therefore, we no longer need continue reducing $LPMR_2$, and only need to focus on L1 layer mismatch.

We increase IW, ROB, L1 cache port number and pipeline width. Configuration C is the first scheme meets the 10% requirement we found in the architecture exploration. The data stall time is 9.6% of CPI_{exe}. Until this moment, the coarse-grained optimization is completed.

If more hardware parallelism is available, we can continue the optimization; then configuration D is found that meets the "1%" requirement. The data stall cost is already small. As an optional step, we continue to check if hardware is over provided. 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 which meets the "1%" requirement with minimal hardware cost.

The $LPMR_1$ is reduced from 8.1 in the first configuration to 1.2 in the fourth configuration. With LPMR reduction algorithm, the design space exploration has a clear goal. That is, to present a minimum but enough hardware parallelism to achieve the layered performance matching; this avoids blind hardware overprovision while accommodating application diversity.

As future architectures support hardware configurability, future architecture parameters can be adjusted to reduce LPMR at each layer of a memory hierarchy [16]. The LPM algorithm is practically feasible. It is also effective, since it is uniquely based on a quantitative measurement of the combined impact of data locality and concurrency. The search for optimal parameters can be guided by the LPM model. The LPM model is innovative because it has quantified the layered matching degree and thus we can decide which parameter should be optimized on demand.

In next subsection, a case study, from application-aware and hierarchy-aware perspective, is presented to illustrate the method to ease the layered performance mismatch via the software approach.

B. Case Study II: LPM Optimization on Heterogeneous Level-1 Caches

We show an example of LPM optimization on heterogeneous L1 caches. Multiple cores with different L1 cache size are called NUCA (Non-Uniform Cache Access architecture). We assume a multi-core architecture with heterogeneous L1 data caches.

As shown in Fig. 5, in our case study II, there are four different computing units with each unit has four cores. The

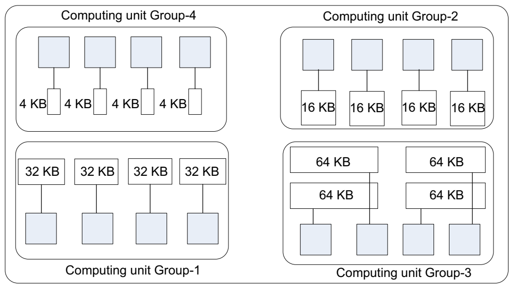

Fig. 5. CMP (16-core) with different private data cache size

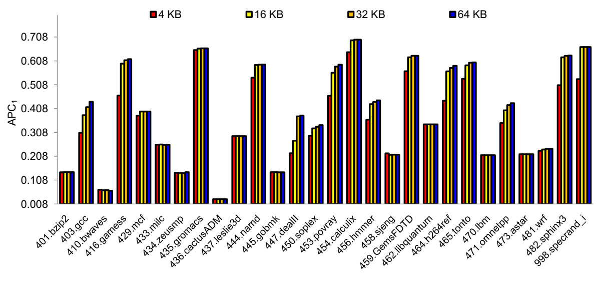

Fig. 6. APC_1 of the applications running on cores with different L1 data cache size

L1 data cache size of each different unit is 4 KB, 16 KB, 32 KB, and 64 KB, respectively. We set delta as $T_1 \times 50\%$. We measure $LPMR_1$ and $LPMR_2$ on each of the four different computing units. Fig. 6 shows the APC values of the L1 layer, and Fig. 7 shows the APC values of the L2 layer, on different computing units.

From Fig. 6 and Fig. 7, we have the following two observations. First, the optimal private data cache sizes are not all the same for different applications. For example, 4 KB is large enough for 401.bzip2, but 64 KB is needed for 403.gcc to achieve the optimal memory performance (in terms of APC_1). Second, the L2 cache performances (in terms of APC_2) of some applications are sensitive to their private data cache sizes, while those of others are not. For example, the APC_2 values of 401.bzip2 are stable while that of 403.gcc decreases in each step, and that of 429.mcf drops to its final APC value at the first cache size increase.

For 416.gamess, increasing L1 data cache size can improve its performance and reduce its L2 cache bandwidth requirement (in terms of APC_2) noticeably. However, for 433.milc, increasing L1 data cache size will get little performance improvement and has little influence on L2 cache bandwidth requirement (in terms of APC_2).

Based on the above observations, the scheduling policy of NUCA may significantly affect the memory performance of an application and its interferences with others. Following the LPM algorithm, we have a two-fold scheduling process: first for L1, in order to get the optimal memory performances for all the applications, the applications are assigned to the cores to get $LPMR_1$ as small as possible according to their cache size needs, rather than allocating randomly; second for L2,

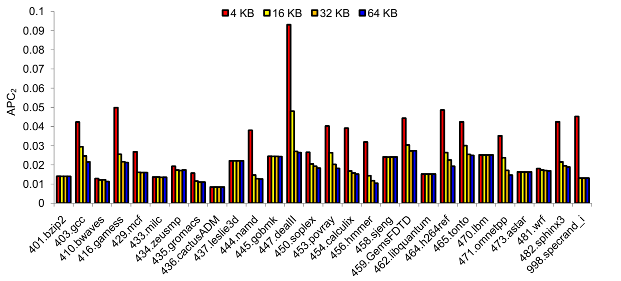

Fig. 7. APC_2 of the applications running on cores with different L1 cache size

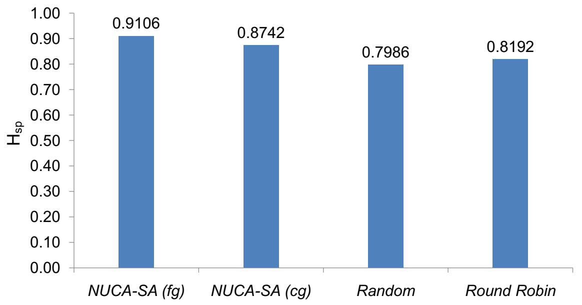

Fig. 8. H_{sp} of different scheduling schemes on an NUCA multiprocessor

to reduce the contention in the L2 cache, the applications are assigned to cores to get APC_2 requirement as small as possible (corresponding to $LPMR_2$ as large as possible).

For a given application, there may have a tradeoff between obtaining a better performance and reducing its interfering with others (assuming L2 is shared in our case studies). In the LPM algorithm Fig. 3, we have considered this tradeoff carefully, and we match $LPMR_1$ before match $LPMR_2$, and try to match both.

The motivation behind the above two-fold scheduling is that each program tries to obtain the optimal memory performance with minimum amount of resource, and, in the meantime, the assignment should have as little interference to other applications as possible. This comes at a tradeoff: sometimes, a program can achieve its optimal performance with a small amount of private resource; however, in this case, it will request more shared resource and thus bring significant interference impact on other applications. Therefore, there exists a semi-optimal scheduling scheme to mitigate the overall memory-wall impact on the overall performance of a computing system.

We use Harmonic Weighted Speedup H_{sp} [20] to evaluate our design. H_{sp} strikes a balance between throughput and fairness and thus has been widely used to evaluate the schemes in a multiple program environment.

To maximize H_{sp}, LPM algorithm is implemented to provide a semi-optimal solution under an NUCA environment and is referred to as the NUCA-aware scheduling algorithm (NUCA-SA). Sixteen benchmarks of SPEC CPU2006 are selected to run on the 16 cores of the architecture as shown in Fig. 5.

It is a known fact that multiprocessor scheduling is an NP-hard optimization problem. In practice, Random scheduling and Round Robin scheduling are the wildly used scheduling policies in both data-center and HPC environments.

As shown in Fig. 8, with fine-grain NUSA-SA, the throughput of multiple programs is improved by 12.29 % compared to Random and by 11.16 % compared to Round Robin. Fig. 8 also shows the result for coarse-grained NUSA-SA.

In the case study given above, the application-to-architecture mapping space size is extremely large(equals to 63,063,000). Using exhaustive search to find an optimal scheduling scheme is not realistic. But, with the help of the LPM algorithm, heterogeneity-aware scheduling has been achieved with polynomial time complexity.

Aided by the LPM model, application awareness and heterogeneity awareness can be achieved. The multicore memory-stall impacts are mitigated due to smarter utilizing of on-chip resources and less contention for the shared resources. The layered performance matching is shown to be feasible and effective in reducing data stall time and surmounting memory wall effect through of the case studies.

Based on the LPM algorithm, more and more specific algorithms, not limited to the case studies in this section, can be designed for diverse environments. As heterogeneous multicore is becoming a mainstream for chip multiprocessors [3], the LPM algorithm is a useful and timely tool for both processor design and software development.

VI. Related Work

We were the first group of researchers to report the importance of memory system performance. In 1990, we proposed the memory-bounded parallel speedup model [21], also known as Sun-Ni's law. The memory-bounded parallel speedup model revealed that data access is the key factor influencing performance, and scalable computing is bounded by memory capacity. In 1994, the term memory wall was formally introduced based on the Average Memory Access Time (AMAT) model [1]. After twenty years of responding to the memory-bound and memory-wall problem, intensive research has been conducted to improve memory system performance. Today, modern microprocessors utilize 80% or more of their transistors for the on-chip cache rather than computing components. Many advanced memory technologies have been developed, including numerous concurrency driven optimization technologies.

A formula is proposed to relate MLP to overall performance [22]. The goal of [22] is similar to LPM, but there are fundamental differences. The MLP metric is a special case of Access Per Cycle (APC) [12] [13], which focuses on measurement rather than analysis. LPM employs C-AMAT for performance analysis and optimization. With C-AMAT, we can unify and optimize the performance in pure miss ratio, pure miss penalty, pure miss concurrency, and hit concurrency. In addition to performance analysis, LPM proposes the notion of Layered Performance Matching and provides a set of associated feasible measurements, optimization mechanisms, and algorithms.

Scheduling heterogeneous multi-cores through performance impact estimation (PIE) [23] requires a data stall time formula. However, the PIE uses an approximate formula that is inaccurate. LPM can provide an improved result of the scheduling and will make the PIE more accurate.

With locality information, cache can be managed well by [24]. While we acknowledge the contribution of [24], we found that it only utilizes locality information and can be drastically enhanced with a guide from the LPM model.

As data stall plays an important role in many hardware and software design issues, an explicit expression of data stall time which simultaneously considers both the impact of locality and concurrency is urgently needed. The related works summarized above are examples that will directly benefit from this study.

In summary, the LPM model captures the minimal requirement for narrowing the disparity between computing and data access and presents a novel method to conduct effective optimizations. It is a promising tool to facilitate existing and future techniques to reduce data stall time and to mitigate the memory wall problem. The two case studies are general and represent a large class of applications. They demonstrate the potential of the LPM model in obtaining optimal and semi-optimal solutions for the NP-hard problem of system configuration design and scheduling.

VII. Conclusions

Concurrency technologies have been widely used in modern memory systems. However, utilizing these technologies in terms of overall performances and in terms of integrating with data locality remains elusive and remains to be a research issue. In this study, we propose a novel performance optimization model, the Layered Performance Matching (LPM) model for 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. The LPM model and its associated LPM algorithm have been illustrated with case studies. Experimental testing has confirmed and demonstrated the feasibility and ingenuity of the LPM model in memory system design and optimization.

LPM can be achieved by memory hardware reconfiguration, or through appropriate scheduling of applications in a heterogeneous environment. The LPM approach is practical, feasible, effective and is valuable in facilitating computing systems for data-intensive applications.

In the future, we plan to investigate more applications of LPM, including parallel file systems, reconfigurable chips, and heterogeneous platforms. We also plan to explore various methods to implement LPM, including memory parallelism partition, selective cache replacement, in addition to the methods discussed in our case studies.

Acknowledgment

This work is supported in part by the National Science Foundation, under grant CNS-1162540, grant CCF-0937877 and grant CNS-0751200.

References

[1] W. A. Wulf and S. A. McKee, "Hitting the Memory Wall: Implications of the Obvious," *ACM SIGARCH computer architecture news*, vol. 23, no. 1, pp. 20–24, 1995.

[2] S. A. McKee, "Reflections on the Memory Wall," in *Proceedings of the 1st conference on Computing frontiers*. ACM, 2004, p. 162.

[3] S. Borkar and A. A. Chien, "The Future of Microprocessors," *Communications of the ACM*, vol. 54, no. 5, pp. 67–77, 2011.

[4] N. Hardavellas, I. Pandis, R. Johnson, N. Mancheril, A. Ailamaki, and B. Falsafi, "Database Servers on Chip Multiprocessors: Limitations and Opportunities," in *Proceedings of the Biennial Conference on Innovative Data Systems Research*, no. 8, 2007.

[5] S. Somogyi, T. F. Wenisch, A. Ailamaki, and B. Falsafi, "Spatio-temporal Memory Streaming," in *ACM SIGARCH Computer Architecture News*, vol. 37, no. 3. ACM, 2009, pp. 69–80.

[6] P. J. Denning, "The Working Set Model for Program Behavior," *Communications of the ACM*, vol. 11, no. 5, pp. 323–333, 1968.

[7] ——, "The Locality Principle," *Communications of the ACM*, vol. 48, no. 7, pp. 19–24, 2005.

[8] S. Gupta, P. Xiang, Y. Yang, and H. Zhou, "Locality Principle Revisited: A Probability-based Quantitative Approach," *Journal of Parallel and Distributed Computing*, vol. 73, no. 7, pp. 1011–1027, 2013.

[9] X. Xiang, C. Ding, H. Luo, and B. Bao, "HOTL: A Higher Order Theory of Locality," in *ACM SIGARCH Computer Architecture News*, vol. 41, no. 1. ACM, 2013, pp. 343–356.

[10] X.-H. Sun and D. Wang, "Concurrent Average Memory Access Time," *Computer*, vol. 47, no. 5, pp. 74–80, 2014.

[11] X.-H. Sun, "Concurrent-AMAT: A Mathematical Model for Big Data access," *HPC Magazine*, 2014.

[12] X.-H. Sun and D. Wang, "APC: A Performance Metric of Memory Systems," *ACM SIGMETRICS Performance Evaluation Review*, vol. 40, no. 2, pp. 125–130, 2012.

[13] D. Wang and X. Sun, "Apc: A novel memory metric and measurement methodology for modern memory system," *IEEE Transactions on Computers*, vol. 63, no. 7, pp. 1626–1639, 2014.

[14] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. Elsevier, 2012.

[15] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically Characterizing Large Scale Program Behavior," *ACM SIGARCH Computer Architecture News*, vol. 30, no. 5, pp. 45–57, 2002.

[16] Y. Wu, Y.-J. Chen, T.-S. Chen, Q. Guo, and L. Zhang, "An Elastic Architecture Adaptable to Various Application Scenarios," *Journal of Computer Science and Technology*, vol. 29, no. 2, pp. 227–238, 2014.

[17] Y.-H. Liu and X.-H. Sun, "Reevaluating Data Stall Time with the Consideration of Data Access Concurrency," *Journal of Computer Science and Technology*, vol. 30, no. 2, pp. 227–245, 2015.

[18] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti *et al.*, "The Gem5 Simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.

[19] C. D. Spradling, "SPEC CPU2006 Benchmark Tools," *ACM SIGARCH Computer Architecture News*, vol. 35, no. 1, pp. 130–134, 2007.

[20] K. Luo, J. Gummaraju, and M. Franklin, "Balancing Thoughput and Fairness in SMT Processors." in *ISPASS*, vol. 1, 2001, pp. 164–171.

[21] X.-H. Sun and L. M. Ni, "Another View on Parallel Speedup," in *Supercomputing'90., Proceedings of.* IEEE, 1990, pp. 324–333.

[22] Y. Chou, B. Fahs, and S. Abraham, "Microarchitecture Optimizations for Exploiting Memory-level Parallelism," in *ACM SIGARCH Computer Architecture News*, vol. 32, no. 2. IEEE Computer Society, 2004, p. 76.

[23] K. Van Craeynest, A. Jaleel, L. Eeckhout, P. Narvaez, and J. Emer, "Scheduling Heterogeneous Multi-cores through Performance Impact Estimation (PIE)," *ACM SIGARCH Computer Architecture News*, vol. 40, no. 3, pp. 213–224, 2012.

[24] G. Kurian, O. Khan, and S. Devadas, "The Locality-aware Adaptive Cache Coherence Protocol," *ACM SIGARCH Computer Architecture News*, vol. 41, no. 3, pp. 523–534, 2013.