CaL: Extending Data Locality to Consider Concurrency for Performance Optimization

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

Abstract—Big data applications demand a better memory performance. Data Locality has been the focus of reducing data access delay. Data access concurrency, however, has become prevalent in modern memory systems in recent years. How to extend existing locality-based performance optimization to consider data concurrency becomes a timely issue facing the researchers and practitioners in the field of computing, especially in the field of big data computing. In this study, we introduce the concept and definition of Concurrency-aware data access Locality (CaL), which, as its name states, extends the concept of locality by considering concurrency. Compared to the conventional concept of locality, CaL accurately reflects the combined impact of data access locality and concurrency in modern memory systems and is very effective for data intensive applications. The value of CaL can be quantitatively measured directly by performance counters in mainstream commercial processors and is practically feasible. Two theoretical results are presented to reveal the relationships between CaL and existing memory system performance metrics of memory accesses per cycle (APC), average memory access time (AMAT), and memory bandwidth (B). In this way, we provide a methodology to use existing locality-based optimization methods directly or in combination with data concurrency optimizations, to improve the value of CaL and to improve the performance of a memory system. To demonstrate the practical value of CaL, we conduct four case studies to illustrate the power of concurrency-aware locality optimization. Compared with the conventional locality based optimization, the CaL-aware design has achieved significant performance improvement. It achieved a 3.12-fold speedup on K-means, which is a widely-used data analytic kernel from the big data benchmarks.

Index Terms—Memory wall, memory stall time, memory concurrency, memory hierarchy, memory access patterns

1 Introduction

Data is becoming an increasingly vital resource in many scientific and engineering domains. As data is the object and result of computation, any computer operation requires the accessing, managing, manipulating, and storing of data. However, data access has become the dominant performance bottleneck of computing systems. Processors commonly spend 50 to 70 percent of their total application execution time waiting for data to arrive [23], [44]. This large waiting for data ratio is stemmed by the large access time ratio among the memory layers (i.e., memory wall effect [52]) and the limitation of the pin bandwidth (i.e., bandwidth wall effect [28], [43]). It is likely to be even more problematic in the near future as modern processors have moved into the many-core era and big data applications are becoming a norm. On one hand, many-core structures provide increasingly high computing capacity, which needs to be matched with high data access speed. On the other hand, big data applications increase the complexity and intensiveness of data access patterns, making the memory wall problem even more severe. To mitigate the effects of memory wall for big data applications, memory performance optimizations are becoming increasingly critical on modern computer systems.

Memory performance optimizations generally fall into two categories, optimizations to improve data locality and optimizations to improve data concurrency. The end goal of such optimizations is to reduce the data stall time. Locality-based optimization is a well-studied topic and is the focus of data access optimization for many years. In the meantime, concurrency-based technologies have been deployed in modern memory systems to overlap or hide the data access latency [11]. For instance, a processor can conduct out-of-order execution to reduce access latency and a cache deployed with a miss handling architecture (MSHR) [30] can serve multiple outstanding misses simultaneously. Even when the cache miss rate is high, strong concurrency can significantly reduce the penalty of data accesses perceived by the processors. Concurrency has become an effective leverage for the performance optimization of memory systems.

While both optimization approaches are important and effective, data locality and data concurrency-based approaches, however, influence each other. It is hard to achieve the best data locality and best data concurrency at the same time. We use average memory access time (AMAT) to measure locality. As shown in Section 7, current applications rarely achieve both short AMAT and strong concurrency (C) simultaneously. Locality and concurrency have many different ways to influence each other. As an example, concurrency puts pressure on shared resources,

• 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 20 Dec. 2015; revised 21 Apr. 2017; accepted 4 Sept. 2017. Date of publication 17 Sept. 2017; date of current version 7 June 2018.
(Corresponding author: Yuhang Liu.)
Recommended for acceptance by J. Chen and H. Wang.
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/TBDATA.2017.2753825

such as the on-chip network bandwidth and the cache capacity. Therefore, increasing the degree of concurrency may intensify the bandwidth contention and the cache pollution, and then, as a result, increases AMAT.

We would like to optimize data locality and concurrency simultaneously. However, traditional locality models, such as reuse distance [37] and miss rate [20], are difficult to be used for concurrency-related performance analysis because they are based on a sequential data access trace and rarely consider data access concurrency. Although some literatures discussed the tradeoff between data access locality and thread level parallelism [8], [26], [29], the locality concepts in these works do not consider data access concurrency.

In this study, we revisit the definition of locality, and then introduce the concept and a formal definition of Concurrency-aware data access Locality (CaL). CaL is an extension of the conventional locality to consider concurrency. When there is no concurrency, it is the same as locality. When concurrency exists, it includes concurrency under its consideration, and therefore, has the power to utilize locality and concurrency simultaneously. Following the recent locality definition, CaL is given in an open mathematical form to accurately describe the relationship between concurrency and locality based optimizations. CaL can be applied for performance analysis at different levels, includes thread-level, socket-level, node-level and system-level.

This work makes the following major contributions:

(1) We propose a mathematical model CaL to quantify locality with the consideration of data access concurrency. Compared with miss rate and reuse distance, the new model is more practical in reflecting integrated impact of data locality and concurrency in modern memory systems. Moreover, methodology has been given to measure CaL by performance counters of mainstream commercial processors.

(2) We propose and prove two theorems to reveal the impact of concurrency-aware locality on memory system performance by analyzing its impact on two fundamental memory metrics, APC and AMAT. Understanding the impact allows us for better design of the memory hierarchy and for providing locality-centric designs which also are concurrency aware. The two theorems are general and foundamental to capture the key aspects of modern memory systems and show their connections, and thus are vital for rethinking and redesigning the memory systems.

(3) Using our model, we can decompose an application's entire execution into different phases based on its data access patterns. We can then optimize CaL on each of these phases individually. To verify and demonstrate the values of our theoretical results and optimization methodology, we have conducted four CaL-driven optimization case studies, one for the potential of CaL, and three for applying CaL in optimizing the hardware configurations of dynamic cache block size, dynamic MSHR, and selective cache array, respectively. As we expected, CaL can be optimized systematically or semi-systematically, and is extremely effective for big data applications. It achieves a performance gain of 312 percent on the K-means big data benchmark, which is a widely-used kernel in data analytics.

In this paper, unless otherwise stated, the term concurrency denotes the memory level parallelism rather than the thread level parallelism, and the term memory indicates the whole hierarchical memory system rather than only the main memory.

The remainder of the paper is organized as follows. Section 2 presents the background about data concurrency and locality. Section 3 defines and formulates the CaL model. Section 4 derives the impact of CaL on hierarchical memory system performance. Section 5 conducts software optimization on big data benchmark, K-means. Section 6 presents three hardware optimization case studies. Section 7 discusses more about the usage of the CaL model and its associated theorems. Section 8 reviews related work. Finally, Section 9 concludes this study.

2 Background: Data Concurrency and Locality

Locality of data accesses is the fundamental principle driving hierarchical memory system design. Many traditional optimizations on memory are focused on locality, to reduce miss rate (MR) or to reduce the number of misses. Given the significance of the locality principle, previous works have attempted to quantify locality to better understand reference patterns and to guide compiler and architecture design to exploit program locality. For temporal locality, the histogram of reuse distances [53] or LRU (least recently used) stack distances [8] is computed from a sequential address trace. For spatial locality, however, there is a lack of consensus for such a quantitative measure and several ad-hoc metrics are proposed based on intuitive notions [27], [34]. All the models assume an ordered list of data accesses, where concurrency is not considered. In this paper, we revisit the concept of locality with the consideration of concurrency.

There have been a host of optimizations related to improving data access concurrency in order to mitigate the penalty of long-latency data accesses. Larger instruction windows, reorder buffers and multithreading [49] collectively provide a high rate of memory requests that the underlying memory must handle. Some widely-used cache optimization methods, such as non-blocking cache [30], pipelined cache [2], multi-banked cache [42] and data prefetching [7], allow data accesses generated by processors to overlap with each other [11]. Mainstream processors now contain many cores running different applications with different access patterns but share the same memory system. A large amount of memory requests must be handled simultaneously to feed data for many-core computation to meet the high data access demand. Failing to do so will cause even longer data stall time on the order of 10x [50]. To achieve this, concurrency-aware optimization is needed.

The complexity of modern memory systems urgently calls for an explicit formula to link concurrency and locality of data access patterns of applications to the corresponding bandwidth and latency of hierarchical memory systems of computing systems. Please note that in 2004, Patterson presented his important observation, latency lags bandwidth [40]. Since latency and bandwidth are widely used

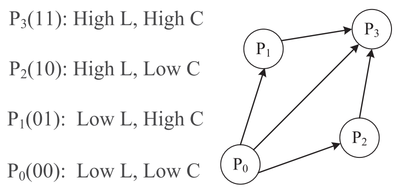

Fig. 1. Four classes of data access patterns with respect to locality (L) and concurrency (C).

for memory system performance optimization, it will be important and interesting to know the relationship between latency and bandwidth, and the impact of locality and concurrency of data access patterns on them. This question becomes increasingly more important for data-intensive computing, whereas data-intensive is the character of big data applications. The reason is that, to cope with the ever-widening memory wall and bandwidth wall, locality-oriented and concurrency-oriented technologies coexist and make memory systems increasingly more complicated.

To showcase how locality can influence miss concurrency and bandwidth utilization, we assume there exist 10 outstanding concurrent data accesses (please note that in real cases the concurrency degree can be even higher). On one hand, if all these accesses concurrently target a common data block which is not cached yet, the data access pattern would have a high miss concurrency. During this period, the cache miss rate is 100 percent, while data access locality is high. This is case 1 which has strong locality. On the other hand, if all the 10 accesses target different individual cache blocks none of which are cached, the data access pattern still has high miss concurrency. However, no locality exists and 10 different cache blocks need to be moved to the cache. This is case 2 which has no locality. Note in the previous case, only one cache block is enough to meet all the requests and thus a 10-fold disparity of bandwidth requirement is shown. The significance of the large disparity is important due to the ever widening memory wall and bandwidth wall. In addition, it is important to notice that the large disparity is caused by variance in data access patterns in terms of their locality and concurrency values.

With the CaL model, proper optimization techniques can be enabled to transform the patterns from the low-end to high-end as shown in Fig. 1, which illustrates how application access patterns can shift between different states. Assume the amount of data accesses needing to complete is fixed. There exist four different patterns, where “1” corresponds to high and “0” corresponds to low. P_3 is the optimal pattern, and P_0 is the worst pattern. If all the data access patterns are P_3, the data accesses can be completed in parallel with minimal bandwidth requirement, where the effect of memory wall and bandwidth wall would not be significant. On the other hand, if all the data access patterns are P_0, then the data accesses must be completed with a large amount of bandwidth consumption, where the impact of bandwidth wall would be significant. P_1 and P_2 can be analyzed in a similar manner. However, without a powerful model, it is difficult to quantify their differences.

Applications may move among the four data access patterns phase by phase during their runtime. In fact, there is no memory system configuration (e.g., the size of cache block size, the capacity of MSHR, the cache capacity and the number of cache channels as will be discussed in Case studies) that works best for all data access patterns. Each design has its own pros and cons, depending on the interaction between the data access patterns and underlying memory system. This motivates us to investigate pattern directed dynamic optimization. CaL can be used in both software runtime optimization and hardware ASIC (application-specific integrated circuit) design.

3 The Proposed Concurrency-Aware Locality (CaL)

Before deriving the formula of CaL, we need to define its concept. The conventional data locality includes spatial locality and temporal locality. Temporal locality indicates that after an address is referenced, the same datum is likely to be re-accessed in the near future. Spatial locality, on the other hand, indicates that some neighbors of the referenced address are likely to be accessed in the near future. Please note that temporal locality is a special case of spatial locality with the neighborhood size being zero bytes (i.e., multiple accesses target the same data) [21]. Therefore, we only need to focus on spatial locality.

Locality has been defined formally by Gupta et al. in terms of probability as: given the condition that an arbitrary address, A, is referenced, the likelihood of an address in its neighborhood to be accessed in the near future [21]. Such conditional probability can be expressed as Eq. (1), where X_0 is the address of current data access and X_n is the address of a data access in the near future. In our study, we regard Eq. (1) as the state-of-the-art definition of locality, and note it as L.

$$L = P(\exists X_n \ in \ A's \ neighborhood \ n \ < N \mid X_0 \ = \ A \ ) \quad (1)$$

L is measured based on the data access trace without accurate timing information, since the “near future” is in the unit of data access. L focuses on time-independent reference activity, where event ordering and interleaving are of prime importance, but the time duration and the overlapping of events are ignored. To extend locality to concurrency-aware locality, we first need to have accurate timing information among data accesses. With accurate timing information, we not only know whether data accesses are overlapping, but also the number of overlaps. In our model, the time window is measured in clock cycle for accuracy, so that the number of overlapping in each clock cycle can be measured and the data access concurrency is well measured.

We define concurrency-aware locality as the number of accesses occurred in a time window within the neighborhood of a previously fetched byte of data. Table 1 lists the notations that will be used in our work. Here, only when two accesses X and Y target the same cache block of size K, they are deemed in the neighborhood of each other. When two accesses are within the same neighborhood, the model partitions them into the same group. The access sequence is S for any given time window T. Assume there exist g cache blocks that are accessed concurrently during time window T. $\{S_1, S_2, , S_g\}$ is a partition of the data access set S with regard to the g different access groups. Then concurrency-aware locality (CaL) can be formulized as Eq. (2).

TABLE 1
Parameter Notations

Notation	*Description*
X, Y	Access
K	Cache block size
S	A set of accesses
G	A group of accesses targeted the same block
T	The total number of memory active cycles
g	The number of groups
N	The number of data accesses
S_i	The ith data access group ($1 \leq i \leq g$)
CaL	Concurrency-aware locality
B	Bandwidth
APC	Access per memory active cycle
C	The degree of data access concurrency
$AMAT$	Average memory access time

$$CaL = \frac{E(The\ group\ size\ of\ G \mid X \in G, G \in \{S_1, S_2, \ldots, S_g\})}{The\ size\ of\ feteched\ bytes\ for\ each\ group} \quad (2)$$

In Eq. (2), we take the size of access group as a random variable, and we use its conditional expectation to quantify how many times a data block can be reused. More detail of conditional expectation can be seen in [24]. Eq. (2) is based on the following observations: Given a fixed size of fetched data, if each data block can be reused more times, the locality is stronger. On the other hand, for a fixed reuse time for any data block, if the size of fetched data is smaller, the locality is also stronger. For example, the locality when reusing a cache block of size x for N accesses is larger than the locality when reusing a cache block of size $5x$ for N accesses.

As Eq. (2) is in closed-form that cannot be analyzed conveniently, we need an open-form metric to facilitate optimization. Due to concurrency in data access patterns, we cannot view data accesses as a sequential sequence which lacks the timing information. According to the definition of neighborhood, we can group the accesses if they fall in the same cache block. As shown in Fig. 2, each group includes at least one data access, and the number of groups is g. We propose a new formula shown in Eq. (3).

$$CaL = \frac{N}{g \times K} \quad (3)$$

Notice that the size of fetched bytes for each group, cache line size, is K. Moreover, we can derive that the expectation of group size is n/g. Therefore, Eq. (3) is equivalent to

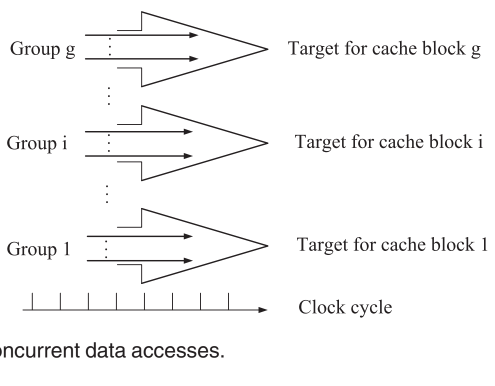

Fig. 2. Concurrent data accesses.

TABLE 2
CaL Values of Different Data Access Patterns

Case	*Pattern*	*CaL*	*L*	*B*
I	$<1>, <1>, <1>, <1>, <1>$	1	0	5
II	$<2>, <1>, <1>, <1>$	5/4	2/5	4
III	$<3>, <1>, <1>$	5/3	3/5	3
IV	$<2>, <2>, <1>$	5/3	4/5	3
V	$<4>, <1>$	5/2	4/5	2
VI	$<3>, <2>$	5/2	1	2
VII	$<5>$	5	1	1

Eq. (2) and thus can be taken as an easy-to-use analytical formula. Meanwhile, we can derive that the value range of CaL is $[1/K, n/K]$. Specifically, when g is one (i.e., all the data accesses target a common data block), CaL equals n/K. On the other hand, when every data access targets a different cache block, CaL equals $1/K$.

In mainstream processors, the cache line size (K) is constant with a typical value 64 B. In this case, we only need to see the Reuse-aware Locality (RaL) shown in Eq. (4), which is defined as how many data accesses can be met by one off-chip data movement. However, when the cache line size is dynamic and adjustable, we still need to use Eq. (3) to reflect the impact of parameter K.

$$RaL = \frac{N}{g} \quad (4)$$

To show the effectiveness of CaL and also to show the difference between CaL and L, let us consider a simple instance. When five data accesses come in at the same clock cycle, there are seven possible locality distributions. Case-I: there exist no two accesses targeting the same cache block. That is, we have five groups: $<1>, <1>, <1>, <1>, <1>$. Case-VII: all the five data accesses are in the same group, which is represented as $<5>$. In a similar manner, we can represent the other five cases as follows: Case-II: $<2>, <1>, <1>, <1>$; Case-III: $<3>, <1>, <1>$; Case-IV: $<2>, <2>, <1>$; Case-V: $<4>, <1>$; and Case-VI: $<3>, <2>$.

We present a quick test for the correctness of the formula in Eq. (3). First, by observation, we order the locality from high to low, and we can get the following results. Case VII has the highest CaL, while Case-I has the lowest CaL. The other five cases are falling between the two extremes. Case-V is close to Case-VI. They both have better CaL compared to Case II, III and IV. Both Case-III and Case-IV are better than Case-II. There is no difference between Case-III and Case-IV. Please note that for simplicity in Table 2 the T and K are taken as one since all the seven patterns have the same T and K. However, in the measurements, T and K will be taken as their real values that may be different in different cases.

In Table 2, all the accesses are misses, the MR is 100 percent for all the cases, so MR cannot reflect the pattern locality. As shown in the third column of Table 2, the CaL metric values calculated by Eq. (3) accurately fit the observation results. However, as shown in the fourth column of Table 2, the L values cannot well differentiate various patterns and thus departing from the observations. With regard to B, case III is the same as case IV, and case V is the same as case

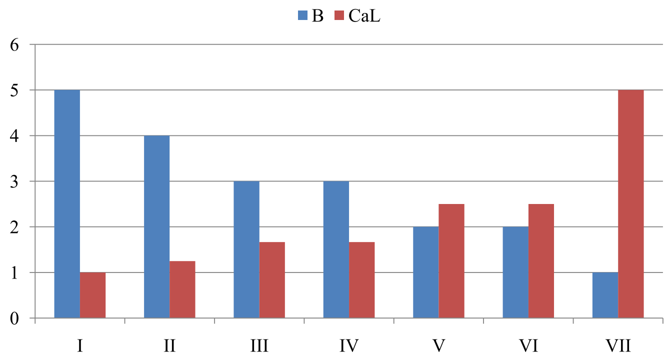

Fig. 3. B decreases with CaL when C and AMAT are fixed.

VI. Their CaL values can reflect the facts accordingly, but their L values cannot. CaL will faciliate performance optimization to use bandwidth efficiently. Given the same concurrency degree, taking the seven data access patterns in Table 2 for example, Fig. 3 shows the relationship between bandwidth B and locality CaL. We can classify locality optimizations into two categories. The designs based on CaL will be referred to as CaL-aware. In comparison, the designs based on L or MR will be referred to as CaL-oblivious.

Although good locality is traditionally thought can improve hit rate, CaL illustrates the importance of miss locality (the locality of cache misses) on bandwidth utilization and cache pollution. A many-core processor would require extremely many accesses to be completed in each clock cycle in order to keep the computation components busy. If there exists no miss locality, the chip needs to simultaneously move many blocks, which may cause significant on-chip network contention and cache pollution. However, when the miss locality is high, many misses can be satisfied by one cache block allowing one block movement to satisfy many misses. Therefore, CaL plays a key role in the many-core and data-intensive computing era. In the next section, we will formally quantify the impact of CaL on key metrics of memory systems.

4 Impact of CaL on Memory System Performance

Access per memory active cycle (APC) [50] and average memory access time [52] are two basic metrics that measure memory system performance and can be used in each layer of the hierarchy. CaL can considerably impact both APC and AMAT. To achieve high data access performance, under bandwidth (B) constraint, we want to maximize APC and minimize AMAT. Understanding the impact will allow better design of the memory hierarchy, which is locality-centric and concurrency-aware.

By definition [46], data access concurrency C is equal to Eq. (5).

$$C = \sum_{i=1}^{T} C_i \times \frac{1}{T} \tag{5}$$

In Eq. (5), T is the total number of memory active cycles. In each memory active cycle, the number of outstanding accesses is C_i ($C_i \geq 1$). The physical meaning of C is the average number of accesses in a memory system during a memory active cycle.

Here, an important issue is how to select the T value. To address this issue, we first need to decide how to count the cycles and how to identify the number of the counted cycles.

For counting cycles, please recall that "memory active cycles" [50] is used in C-AMAT and APC for memory measurement. This seems to be a natural choice, if you think from a memory-centric or data-centric viewpoint. But, traditionally we have been using CPU cycles for memory performance. Note that not every CPU cycle is a memory active cycle. Only when a cycle has at least one memory access, this cycle is referred to as memory active cycle. For example, during a 20 CPU cycle period, the CPU has issued two data access streams. The two access streams have the same number of data accesses. The first access stream, however, completes the data accesses in one cycle with a combined optimization of locality and concurrency. The second access stream, due to its inherited data access pattern, finishes the data accesses in ten cycles. The T value, then, of the first stream is one, while the T value of the second stream is ten. As a result, due to the difference of their inherited data access patterns, C and CaL of the first request are 10x larger than those of the second request.

The size of the time window T should be carefully chosen. Window size too small cannot reflect the stable state of the program. Window size too large cannot facilitate fine-grained optimization. In our practice, we set T as 1M cycles. That is, the performance parameters are updated and outputted every 1 million cycles. Programs' behaviors usually change every hundreds of millions of cycles. Therefore, the T size selected is small enough to adapt to the program's dynamic behavior.

The CaL metric and other metrics, AMAT, B, APC, can be applied in different levels, include thread-level, socket-level, node-level and system-level. Aided by PAPI and HPCtoolkit, the performance counter values can be collected for a given thread or a group of threads on one or more sockets or nodes. This property is valuable for programmers to conduct data access optimization for their applications.

HPCToolkit has an option "–force-metric", which can be enabled to force hpcprof to show all thread-level metrics. Programmers can use the "-t" option to set the id of the threads. In our case, we have used the option to calculate more complex metrics (i.e., CaL, APC, AMAT, and B).

Depending on the level of interest, the measured parameters can be used directly or need to be used with some simple summations. The N value in Eq. (3) means the number of L1 accesses. If CaL is for a single thread, N is the value of event "L1 accesses" of the given thread. If CaL is for multiple threads, N is the sum of event "L1 accesses" value of each given thread. Other parameters such as g in Eq. (3) are measured and calculated in a similar manner. Note that in mainstream commercial processors, there exist several performance counters per core (e.g., 11 counters per core for Intel Xeon E5 processors), which make the measurement feasible.

4.1 Impact of CaL on APC

Theorem 1 shows the relation of APC and CaL. It also reveals the role of (network/bus) bandwidth in memory system performance.

Theorem 1. *APC of a memory layer is the product of its lower layer's Bandwidth and CaL as given in Eq. (6).*

$$APC = B \times CaL \tag{6}$$

Proof. Assume there are N outstanding data accesses during any given consecutive T memory active cycles according to the definition of APC [50], we get

$$APC = N/T \tag{7}$$

Recalling the definition of bandwidth is "the rate of data transfer, measured in bits per second [41], we can express B as follows.

$$B = \frac{g \times K}{T} \tag{8}$$

Then, based on Eq. (3), we have Theorem 1. □

Corollary 1. *When Bandwidth is fixed, increase CaL will increase APC.*

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

Wang and Sun have proved that increasing APC will improve IPC [50]. For data intensive applications, the improvement of APC can increase IPC significantly. As shown in Eq. (9), Wang and Sun [50] define an application is data intensive if and only if its correlation coefficient of APC and IPC is equal to or larger than 0.9.

$$Data\ Intensive, Application \equiv coe(APC, IPC) \geq 0.9 \tag{9}$$

From memory system perspective, improving APC is equal to optimizing performance. The question remain to be answered is how to increase APC under different conditions. Theorem 1 shows that with more available bandwidth and higher concurrency-aware locality, more APC can be achieved.

From Eq. (6) and Eq. (9), we can derive the relationship between CaL and IPC into Eq. (10).

$$Data\ Intensive\ Application \equiv coe(B \times CaL, IPC) \geq 0.9 \tag{10}$$

CaL plays a vital role in efficiently utilizing the valuable bandwidth. The memory bandwidth should not be over-utilized nor under-utilized. If bandwidth is over-utilized, the contention among data accesses will be severe. If bandwidth is under-utilized, data access concurrency cannot be maximized for better performance.

In the following section, we will consider the impact of CaL on AMAT.

4.2 Impact of CaL on AMAT

Theorem 2 gives the relation between CaL and AMAT.

Theorem 2. *During any given consecutive T memory active cycles, AMAT equals to the ratio of Concurrency over the product of Bandwidth and CaL, as shown by Eq. (11).*

$$AMAT = \frac{C}{B \times CaL} \tag{11}$$

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

The summation of concurrency degree in each memory active cycle during the period of T, is the total time that would be taken when concurrency does not exist, which is referred to as serial time. If all the accesses are conducted serially, the total time required is $N \times AMAT$. Therefore, we have

$$\sum_{i=1}^{T} C_i = N \times AMAT \tag{12}$$

Recalling the definition of APC,

$$APC = N/T \tag{13}$$

we get

$$N = APC \times T \tag{14}$$

Combinning Eq. (12) and Eq. (13), we obtain that

$$\sum_{i=1}^{T} C_i = T \times APC \times AMAT \tag{15}$$

That is,

$$\frac{1}{T}\sum_{i=1}^{T} C_i = APC \times AMAT \tag{16}$$

Recalling the definition of C in Eq. (5), we have

$$C = APC \times AMAT \tag{17}$$

Combining Theorem 1 and Eq. (17), we have

$$C = B \times CaL \times AMAT \tag{18}$$

□

With Theorem 2, we have the following useful corollary 2 for the relation between Concurrency and CaL.

Corollary 2. *When Bandwidth is fixed, to avoid AMAT increase, CaL should increase proportionally with Concurrency, C.*

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

Theorem 2 connects latency and bandwidth (B), with concurrency (C) and concurrency-aware locality (CaL). The bandwidth of a memory system has its upper bound due to physical constraints, so B may be very low. Moreover, a memory system can simultaneously handle many active data accesses, so C may be very high. As a result, according to Theorem 2, AMAT would be very long due to the high concurrency degree and low available bandwidth. However, theorem 2 tells us a new direction to reduce AMAT by improving CaL. Based on Theorem 2, algorithm 2 is presented for shortening AMAT.

Theorem 1 and 2 can be applied in both single-thread and multi-thread environments, single-core and multi-core environments, and in single-node and multi-nodes. All the models discussed above (i.e., AMAT, C, B, and CaL) can be measured in each level of the memory hierarchy, and can be applied for optimizing individual performance or group performance. If we concern the performance of a given thread, we can attach a threadID to each of its data access and then obtain its individual values of AMAT, C, B, and CaL for that given thread. If we concern the average performance, then there is no need of threadID. We can measure the memory performance as all threads coming from the same program, and get the values as usual. In either case, Theorem 1 and 2 hold true.

Theorem 1 and 2 are applicable in both sing-node and multi-node environments. In a multi-node cluster

TABLE 3
The Experimental System Configuration

CPU chip	Intel Xeon E5-2630 v4
Chip socket number	2
Cores per chip socket	10
Threads per core	2
L1 dcache and L1 icache latency	4 cycles
L2 cache latency	12 cycles
L3 cache latency	40 cycles
DRAM memory latency	150 cycles
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

environment, the tasks in algorithm 1 and 2 can be distributed across multiple nodes. In a distributed memory environment, remote memory (memory located under other nodes) can be considered as another layer of the memory hierarchy. The interconnection network controller of a node can collect the model values for remote memory accesses. Therefore, all the theoretical/analytical results of single-node memory hierarchy can be applied to distributed memory environments.

The CaL model is designed for performance optimizations. Theorem 1 and 2 accurately show that a desired increase in CaL causes a desired increase in APC, and a decrease in AMAT. Once APC and/or AMAT can be improved, memory system performance and overall computing system performance in instructions per cycle (IPC) can be improved. Therefore, both theorem 1 and 2 suggest it is necessary to improve CaL for performance optimization. In the next two sections, we show increasing CaL can improve memory performance. In Section 5, we show the effectiveness of CaL for big data applications. In Section 6, we show how to improve CaL systematically under different conditions.

5 CaL-driven Software Optimization Case Study

In this section, we conduct a case study on big data benchmarks running on a sever cluster of Intel Xeon E5-2630 processors. Table 3 shows the configuration of the experimental system, which consists two Intel Xeon E5-2630 chip multiprocessor sockets. Experimental testing and case study can be conducted on a real machine or a simulator (we have conducted three case studies on simulator in the next section). In either way, the CaL testing and optimization is a challenging task. On a real machine, we cannot change the hardware, so we only can increase CaL via software approach, which means changing the algorithm or the implementation of the algorithm. A good optimization requires the in-depth understanding of the targeted applications (algorithms), in addition to the understanding of the underlying hardware support and the concept of CaL. Also, the optimization is case by case. It is for the targeted application only, and may or may not be extendable to other applications.

Conducting experimental testing on a simulator is more flexible, in the sense that we can change the hardware configurations. But, on the other hand, simulator is very slow [5], so

TABLE 4
Hotspot Code Segments of Bigdata Benchmarks

ID	Name	Footprint	ID	Name	Footprint
1	mpi-K-means	0.80 GB	10	omp-K-means	224.45 GB
2	mpi-K-means	53.73 GB	11	omp-K-means	0.256 GB
3	Bayes-predict	1.56 GB	12	Pagerank	2.72 GB
4	Bayes-predict	6.56 GB	13	Pagerank	0.704 GB
5	Bayes-predict	0.83 GB	14	Pagerank	0.16 GB
6	Bayes-train	1.76 GB	15	Wordcount	2.24 GB
7	Bayes-train	6.84 GB	16	Wordcount	4.96 GB
8	Bayes-train	0.74 GB	17	Wordcount	3.81 GB
9	omp-K-means	648.78 GB			

it is difficult to run large benchmarks. We measured that the speed of GEM5 is about 50~500 KIPS (Kilo Instructions Per Second), whereas the execution speed of an actual computing system is in the order of 10^6 to 10^7 KIPS. Even simulating a relatively small program that takes one minute to execute requires approximately one month to a year to simulate. Note that the simulation using GEM5 in Section 6 takes us 6 weeks to finish the evaluation of hardware designs on an Intel Xeon E5 cluster, which includes 320 cores (16 nodes, two CPU sockets per node, and 10-core per socket).

In this case study, we profile the banchmarks from BigDataBench. BigDataBench [18] is for large footprint workloads, modeling typical big data application domains: search engine, social networks, e-commerce, multimedia analytics, and bioinformatics.

Hardware counters, PAPI [15], HPCToolkits [16], Perfexpert [17], are used to identify the bottlenecks and their memory access patterns. A code segment that consumes a significant portion of total running time is referred to as hotspot. In our experiments, for each benchmark, hotspots that take at least 5 percent of the running time are indentified. Table 4 shows the indentified hotspots.

Fig. 4 shows the CaL values of hotspots of the benchmarks from BigDataBench. In the Intel Xeon E5 processor, the cache line size is a constant (64 B), so we use RaL to represent CaL. Fig. 5 shows the memory stall degrees of these hotspots. It shows that the CaL value of code segment

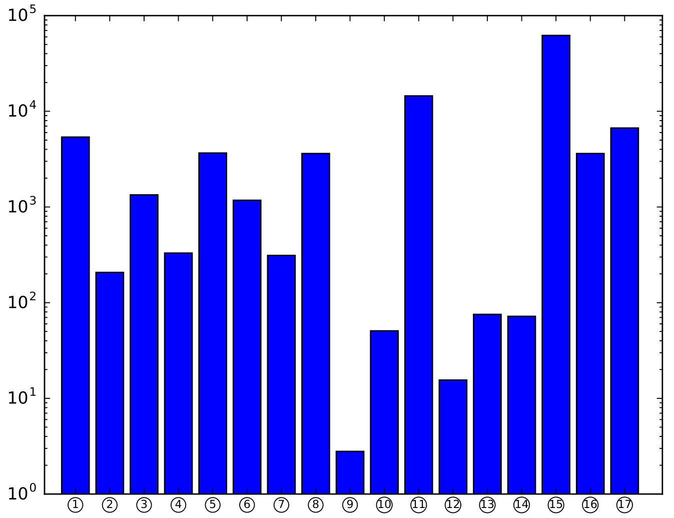

Fig. 4. CaL values of different benchmarks from BigDataBench. Note that in the experimental platform, the cache line size, K, is a constant (64B), so we use RaL value to represent CaL.

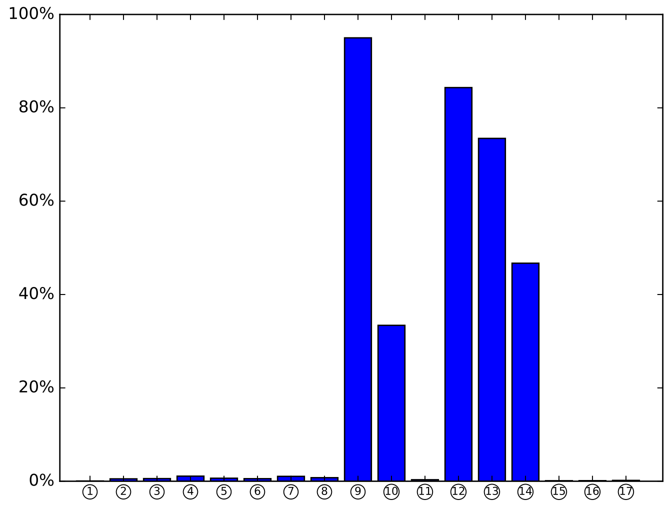

Fig. 5. Memory stall degrees of different benchmarks from BigDataBench.

⑨⑩ ⑫⑬⑭ is very low, where the memory stall degrees are also low. Note that the CaL value of ⑨ is only 2.78, which means one off-chip data movement only meets 2.78 accesses. In comparsion, ⑮ from $wordcount$ benchmark has a CaL value of 62,024.

The code segment ⑨ is the first hotspot of K-means algorithm implemented in the OpenMP programming model. This hotspot takes 75.84 percent of the total workload running time. The main step of K-means algorithm is searching the nearest cluster center for each object and calculating new cluster centers. Multiple threads process the data concurrently to explore data parallelism. Each thread processes part of the data elements, searching their nearest cluster center and calculating the sum of these elements (the average value will be calculated in main thread).The hotspot of K-means is shown below:

```
  #pragma omp parallel shared(...)
{
  int tid = omp_get_thread_num();
  #pragma omp for
  for (i = 0; i <numObjs; i++) {
    index = find_nearest_cluster(numClusters, numCoords,
    objects[i], clusters);
    ...
    membership[i] = index;
    ...
    local_newClusterSize[tid][index]++;
    for (j = 0; j <numCoords; j++)
    local_newClusters[tid][index][j] += objects[i][j];
  }
}
```

In this piece of code, the data accesses for two arrays named $objects$ and $membership$ have low temporal locality. During the loop, the data size of the two arrays are very large. As a result, the data ($local\_newCluster$ and $local\_newClusterSize$) with high temporal locality are pushed out of the cache frequently. To avoid that, the above code can be splited into two parts. The first part searches the nearest cluster centers, and stores it into the array named $membership$. The second part updates new cluster centers. The optimized code is shown below:

```
  #pragma omp parallel shared(...)
{
  int tid = omp_get_thread_num();
  #pragma omp for
  for (i=0; i <numObjs; i++) {
    index = find_nearest_cluster(numClusters, numCoords,
    objects[i], clusters);
    ...
    membership[i] = index;
    ...
    }
  #pragma omp for
    for (i = 0; i <numObjs; i++) {
    index = membership[i];
    local_newClusterSize[tid][index]++;
    for (j = 0; j <numCoords; j++)
    local_newClusters[tid][index][j] += objects[i][j];
  }
}
```

After optimization, the data $local\_newClusterSize$ and $local\_newClusters$ can mostly stay in on-chip caches without thrashing, so the efficiency of this code segment has been improved. Due to the optimization, the CaL value is changed from 2.78 to 126.4, and the total running time of the K-means benchmark has been reduced from 123.26 s to 39.43 s. That is, the CaL-driven software optimization has boosted performance of K-means by 3.12-fold.

As the K-means benchmark is a well-studied benchmark, the 3.12 speedup is amazing and has shown the effectiveness of CaL-driven software optimization. Besides the optimization method used on K-means, other software approaches can be developed for other benchmarks following the hints provided by the two theorems. In the next section, we use the GEM5 simulator for case studies, where hardware can be changed for performance optimization.

6 CaL-Driven Hardware Optimization Case Studies

The case studies given in this section focus on improving CaL, based on its formulation in Eq. (3). The optimizations will improve CaL without increasing the bandwidth requirement B. Note that, Eq. (3) includes three parameters. Each of the following three case studies will focus on improving one parameter while keeping the other two fixed.

- Given a fixed data block number g, and data access number N, case study I decreases cache block size K.
- Given a fixed data block number g, and cache block size K, case study II increases data access number N.
- Given a fixed cache block size K, and data access number N, case study III decreases data block number g.

As shown in Fig. 6, CaL can guide hardware optimization, workload characterization and code optimization. In this section, we will focus on hardware optimization where application workload characterization is also involved. The code optimization will be discussed in Section 6.

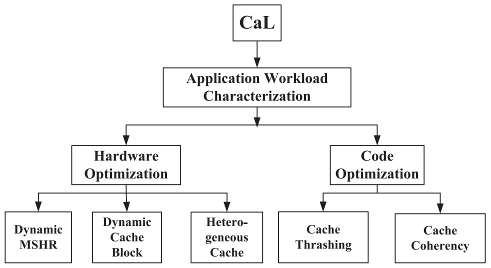

Fig. 6. CaL-driven optimization framework.

We use the modern cycle-accurate simulator GEM5 [5] for our study. In our simulator, as shown in Table 3, we model a many-core processor where each core is with 4-way issue, out-of-order superscalar, and a 128-entry reorder buffer. The memory hierarchy configuration is similar to that of an Intel Xeon processor [33]. The detailed out-of-order CPU model and the DRAMSim2 module in the GEM5 simulator are integrated. In our experiments, as shown in Table 4, we executed the *SPEC* CPU2006 benchmarks [45] with reference input sets. The benchmarks are compiled using GCC 4.3.0 and the -O3 optimization level.

SPEC CPU2006 benchmarks are not big data benchmarks. They are not as data intensive as big data benchmarks. So, the CaL performance gain on SPEC benchmarks is smaller than that on BigData Benchmarks. However, SPEC is easy to be runned in simulators and serves a good purpose to demonstrate CaL hardware optimization methods that cannot be conducted on real machines.

For each benchmark, we use 10 billion representative instructions aided by SimPoint [22]. SimPoint uses the phase clusterings generated by the off-line analysis to intelligently choose where to simulate. Therefore, Simpoint can help conduct efficient and accurate program analysis and architecture simulation, and several researchers in academia and at Intel are using SimPoint to accurately guide their architecture simulation research. In our study, we use Simpoint to find 64 phases that are the most representative to simulate. The data in the case studies are presented for phases randomly selected within the 64 phases.

6.1 Case Study I: CaL-Driven Dynamic Data Block Size

In this case study, we will consider using CaL to decide the optimal data block size. In hierarchical memory systems, the transfer units of data request and reply are different; a word is requested by a load or store operation, but the word is carried in by a cache block (line). If the cache block can be reused multiple times, one data movement can support multiple data accesses. A larger cache block generally consumes more bandwidth but can present more reuse opportunities for later accesses and thus the AMAT would be shorter if the locality is stronger. Therefore, depending on the locality, the effect of cache block size plays a key role in the tradeoff between bandwidth B and latency AMAT. Moreover, because different workloads can have distinct behaviors and even the same workload can have different phases, a fixed cache block size generally cannot achieve the fine matching between data access patterns and the underlying memory system.

Different implementation methods can be used to achieve the effect of dynamic block sizes [9], [19], [21]. For example, Gupta emulated large block size by augmenting the cache with next-n-line prefetching [21]. Dubnicki used split and merge operations executed by a protocol [19]. However, it is difficult to decide when to use large cache block and which block size is optimal.

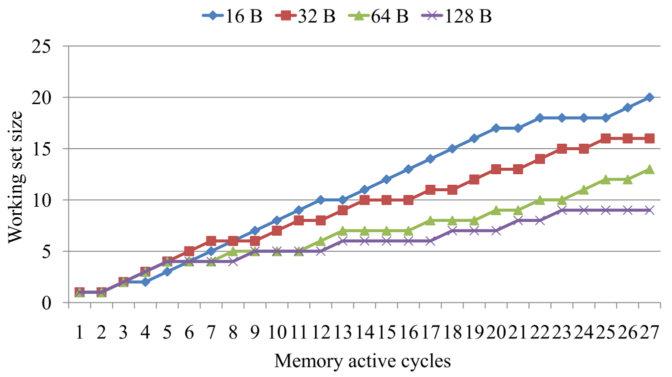

Fig. 7. Working set sizes for different cache block sizes during a randomly selected phase.

This issue is especially challenging to solve during application runtime, since data access patterns are dynamic. Gupta et al. used GPU to analyze off-line data access traces and plotted the locality values in a 3-D mesh [21]. Insights from the 3-D mesh can help determine optimal block sizes. In this work, a lightweight on-line optimization is conducted without any prior knowledge or off-line analysis. We use CaL to classify data access patterns and decide the optimal cache block size.

When the data access pattern has strong CaL so that enough bandwidth is available, a large cache block size is preferred. However, when the CaL of the data access pattern becomes poor leading to insufficient bandwidth, a short cache block size is preferred.

As shown in Fig. 7, the number of blocks needed (i.e., working set size) decreases as we increase the cache block size. However, the B requirement increases with the cache block size, as shown in Fig. 8. Therefore, we can find the optimal cache block size that maximizes the APC (i.e., B $\times$ CaL) during this period. Fig. 9 shows CaL values, in which the 16-byte cache line size achieves the best bandwidth efficiency for the selected data access pattern.

During application runtime, each phase has its preferred cache block size. As shown in Fig. 10, we randomly selected seven phases, which have different CaL values under four

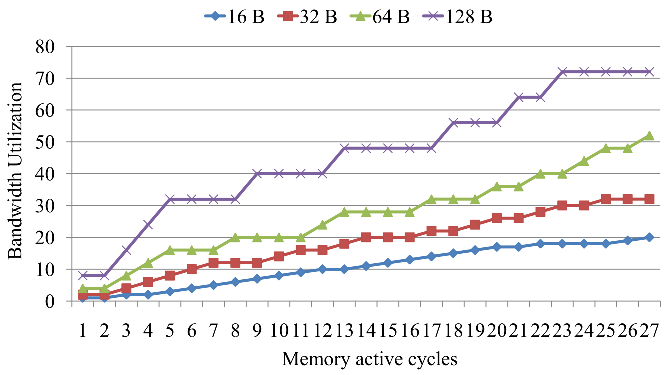

Fig. 8. Bandwidth utilization for different cache block sizes during a randomly selected phase.

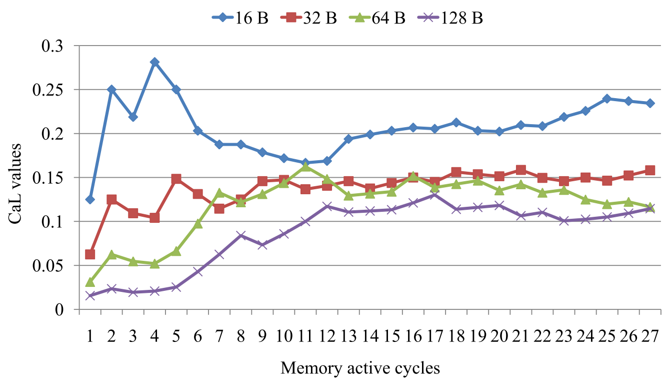

Fig. 9. CaL values for different cache block sizes during a randomly selected phase.

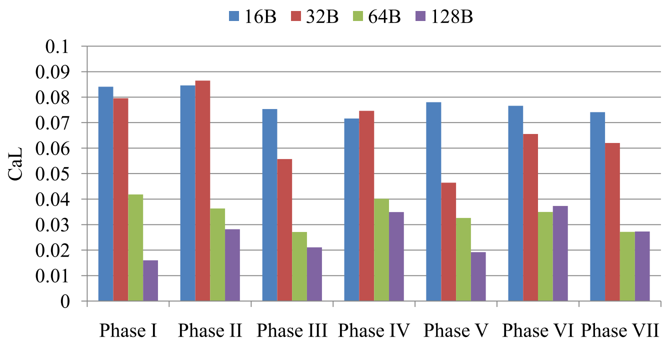

Fig. 10. CaL values with different cache block size during seven randomly selected phases.

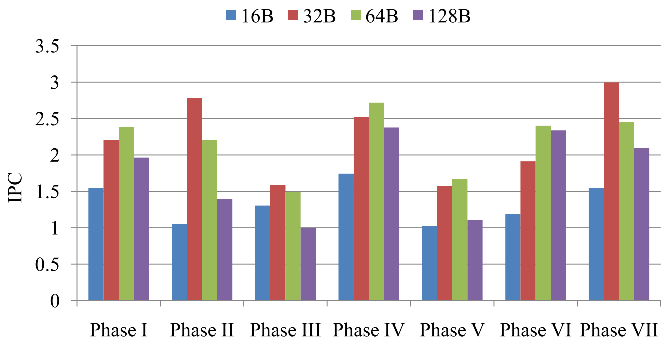

Fig. 11. IPC values with different cache block size during seven randomly selected phases.

cache block sizes. As shown in Fig. 11, different phases require different cache block sizes to achieve optimal performance. For phase I, IV, V, VI, the least amount of time is spent when cache block size equals 64 B while for others 32 B does the best. We see for phases, such as II and VII, 64 B is beaten by 32 B in terms of IPC due to high bandwidth consumption. Thus, we design a methodology for dynamic cache block sizes in order to have the optimal size for different data access pattern phases which are characterized by CaL.

As shown in Fig. 12, we use pattern directed prefetching to implement a dynamic cache block size. With the CaL and C runtime profiling information, we prefetch the next several cache blocks with the 16 B basic cache block size into a stream buffer. For example, when a 128-byte data block is needed, we prefetch the next 7 cache blocks. When a miss occurs in L1, say at address A, the stream buffer immediately starts to prefetch the next several cache blocks. We divided the CaL values into eight grades from low to high,

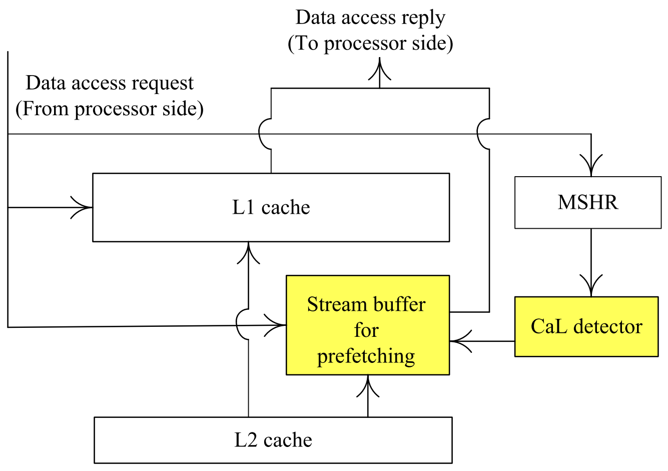

Fig. 12. CaL-driven prefetching.

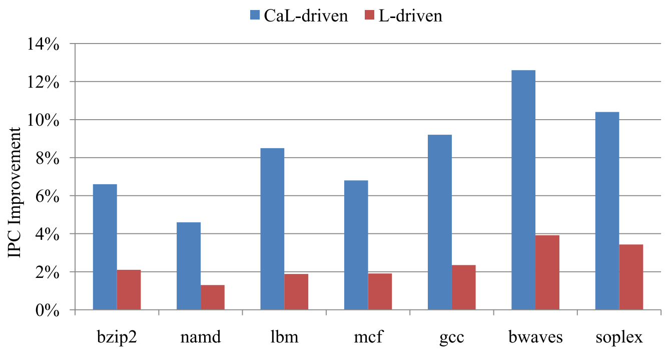

Fig. 13. IPC improvement of different applications.

0-7, which is the number of blocks needed to prefetch. Subsequent accesses check the head of the stream buffer before going to L2. Similar design of stream buffer can be seen in [21], but our experiment is completely driven by CaL and thus is easy to implement.

The stream buffer consists of 7 entries, and zero entry will be used when the minimal block size is preferred. Each entry consists of a tag, an available bit, and a data block. When a miss occurs, according to the CaL value, the stream buffer begins prefetching zero to seven successive blocks starting at the miss target. Note that the prefetched cache blocks are placed in the stream buffer not the cache. This avoids polluting the cache with data that may never be needed. Subsequent accesses to the cache also compare their address against the items stored in the buffer. If a reference misses in the cache but hits in the buffer, the cache can be reloaded in a single cycle from the stream buffer. In this manner, we are able to emulate different cache block sizes without changing the cache configuration. Even when only implemented for the L1 data cache, the performance can be improved by an average of 8.39 percent, and at best by 12.6 percent, as shown in Fig. 13.

We have replaced CaL detector with L detector to do the same optimization, and found that L-driven optimization can also achieve performance improvement. However, as shown in Fig. 13, there only exists 2.41 percent improvement on average, and at most 3.92 percent. Compared with L, CaL obtains a much better performance due to its unique ability of reflecting the combined effect of concurrency and locality on bandwidth.

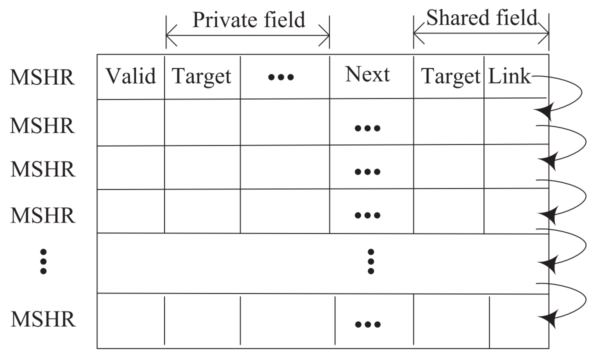

Fig. 14. Pattern directed dynamic MSHR (PDDM).

We also have replaced CaL detetor with MR detector to repeat the same experiment. However, MR detector cannot achieve performance improvement, making IPC less than before for all the applications. This verifies that MR cannot reflect the locality among the cache misses.

6.2 Case Study II: CaL-Driven Dynamic MSHR

Miss status handling registers (MSHRs) are structures added in non-blocking cache to facilitate memory level parallelism. MSHRs are needed to support concurrency $C > 1$. MSHRs keep track of outstanding misses. Each MSHR contains enough state to handle one or more accesses to a single memory line.

CaL is directly influenced by MSHR, which can be used as a floodgate for data access flow between cache layers. Each MSHR contains information about a particular miss to a given cache line. Each target within a MSHR contains metadata information such as the word requested in the cache line and the thread which requested the word. When a miss occurs for a cache block, the MSHR is scanned to see if this is the first miss to the cache block. If so, a new MSHR is allocated. If a previous miss has already occurred for the same cache block, the miss will be allocated a target in the cache block's MSHR. Thus, the number of allocated targets per MSHR is proportional to the miss locality, and the number of allocated targets of all the MSHRs is proportional to the miss concurrency.

There exist three cases for the utilization of MSHR: no locality, that is, each block loaded into the cache satisfies only one access; high locality, that is, each block loaded into the cache can simultaneously satisfy many accesses; and intermediate locality, which falls between the former two cases.

In state-of-the-art processors, due to high register cost, the number of targets per MSHR is fixed, which can be a performance bottleneck. For this issue, James et al. proposed a hierarchical MSHR structure [48]. Driven by CaL, we present a simpler design that dynamically modifies the number of targets per MSHR. In terms of the CaL model, the number of blocks, g, and block size, K, will remain fixed while the number of concurrent misses, N, increases. The number of MSHRs will be represented by g, since the number of MSHRs equals the number of requested blocks.

We propose a pattern directed dynamic MSHR(PDDM) which allows multiple MSHRs to share targets. We analyzed the variation in MSHR allocation and target allocation. When the number of allocated targets for an MSHR is high, most of the other allocated MSHRs have few allocated

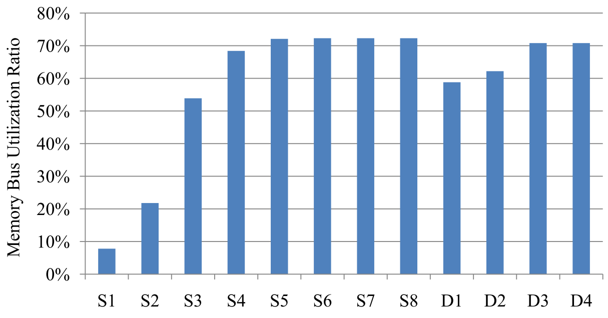

Fig. 15. Memory bus utilization ratio.

targets. A cache miss needs to wait to be serviced when either targets or MSHRs are not enough. PDDM provides higher quality of service by stealing the targets of other MSHRs, since the other MSHRs achieve low locality and they have unused targets. Since misses must wait to be served, a simple solution would be to create miss handling structure with more MSHRs and more targets per MSHRs. However, this is not a practical solution due to hardware constraints. Thus we need to provide a higher performing dynamic MSHR without increasing hardware costs.

We enhance the conventional MSHR structure to implement PDDM, which allows a larger amount of miss concurrency at the same hardware cost. As shown in Fig. 14, a dynamic MSHR includes a private target field and a shared target field. Each miss access corresponds with a MSHR in the PDDM structure, beginning with a “Valid” field showing whether the MSHR is used or not and a private target field, containing a number of private targets. The “Next” field falls between private field and shared field, and stores a link that points to another MSHR. The link is allocated when private targets have been used up. Note that a “Target/Link” pair fills the rest of the line. “Link” field in the Target/Link pair identifies the next “Target/Link” pair, which it links to, allowing the pairs to be organized as a singly linked list. All unused pairs make up a list as the free shared field. During application runtime, initially, the elastic target MSHR structure behaves as usual. When the private target field overflows, the free public target zone allocates an unused “Target/Link” pair to store the new target.

The two main improvements of PDDM are improved capability of tolerating a high amount of misses without increasing the register hardware cost. To better explain the role of theorem 1 in the design of the dynamic MSHR, Fig. 15 shows the memory bus utilization for the different cases shown in Table 5. As the number of MSHR entries (i.e., the parameter g) increases from configuration S1 to S8, the memory bus utilization rapidly increases, however the number of targets does not affect bus utilization. The PDDM uses this fact by allowing more targets to be allocated, holding more concurrent misses, and minimally increasing bandwidth.

Note that the MSHR lookup latency varies each time, depending on the data access patterns. This variation does little in influencing data access performance, since lookup operations are carried out very fast by hardware.

Theorem 2 is used to guide the optimization. We run four applications at the same time. These application interference each other on the shared bus, which can hurt their performance severely.

TABLE 5
Different MSHR Structures

	Structure	#MSHRs	#targets per MSHR
Static Structures	S1	1	64
	S2	2	32
	S3	4	16
	S4	8	8
	S5	16	4
	S6	32	2
	S7	64	1
	S8	128	128
Dynamic Structures	D1	4	Dynamic (1–8 targets)
	D2	4	Dynamic (1–16 targets)
	D3	16	Dynamic (1–16 targets)
	D4	16	Dynamic (1–32 targets)

Higher concurrency demands higher bandwidth when CaL is low, and thus the bandwidth contention may become heavier (AMAT will be longer). Therefore, higher concurrency does not always bring in higher performance, especially when many applications running together on shared resources.

Fig. 16 shows the overall performance in IPC as well as CaL and C values of various MSHR structures. Compared to infinite static configuration S8, the newly proposed PDDM (configuration D2) achieves 15.9 percent performance improvement in terms of IPC. Compared to S3, D2 achieves 7 percent improvement. D2 only uses 16 targets that are only 25 percent of that S3 used. Although S8 uses much more hardware than S3 and D2, its performance is not better. This tells us that more hardware does not always imply better performance if the hardware does not match the application's need. Performance tools, such as CaL, can help us to find a match.

The goal of the PDDM is to increase the concurrency and locality while minimizing the increase in bandwidth. The C/CaL of S8 is 1.56, while C/CaL of D2 is 1.2. This will decrease AMAT by 16 percent on average based on Theorem 2 and result in improved memory performance.

The C/CaL of S8 is high because configuration S8 can allow for a large amount of concurrent outstanding misses but since the locality of these misses is not large the bandwidth consumption is increased. Thus, misses take longer to fetch and thus IPC suffers. This suggests that simply issuing as many concurrent misses as possible is not optimal since bandwidth will be insufficient and requests will become queued. PDDM uses CaL to limit the increase in bandwidth which occurs when concurrency is increased, allowing the structure to handle high amount of concurrent misses efficiently.

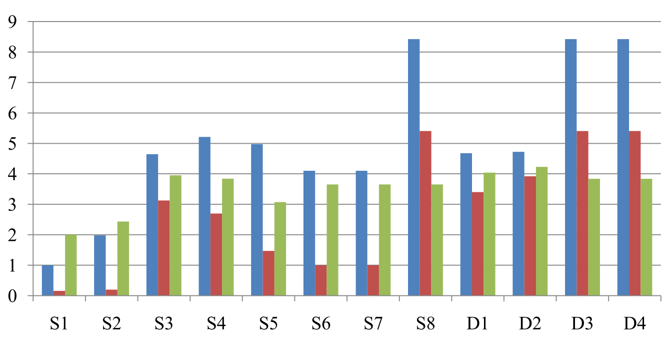

Fig. 16. IPC values of different MSHR structures.

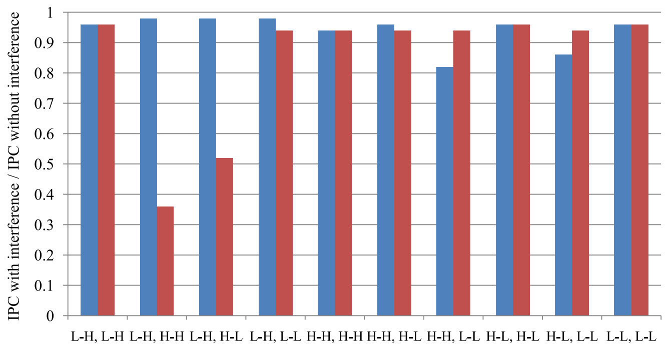

Fig. 17. Interference between data access patterns.

6.3 Case Study III: CaL-Driven Selective Cache Array

In this case study, we use CaL to classify data access patterns from many cores and then to reduce the interference among multiple applications by cache array selection. Running several independent applications on many-core processor are common in a wide range of computing platforms. A representative example is the simulation program for design space exploration in science and engineering where the same simulation program is run with different input data sets.

To reduce main memory interference in multicore systems, Muralidhara et al. proposed to partition memory channels among different applications [39]. In contrast to partition memory channels, we provide a cache partitioning structure for different data access patterns that are generated from many cores or threads.

According to the CaL and C values, we can classify data access patterns into different phases. All memory active cycles can be classified into four categories, H-H, H-L, L-H and L-L.

We randomly select four workloads with different data access patterns, and measure their performance when they run independently. Then we run any two of them simultaneously, and get the performances under interference. We can get the speed ratio between the performances with or without interference for each workload. As shown in Fig. 17, we can see that the same type of data access patterns do not significantly interfere with each other; however, the four patterns, once combined, can severely interfere with each other and cause the cache pollution. In particular, L-H patterns can significantly impact other data access patterns.

Based on these observations, as shown in Fig. 18, we partition the whole cache into four different cache arrays. The cache controller wisely selects its cache arrays according to the data access pattern features. When there exist four types of patterns, each cache array is responsible for processing only one type of data access pattern. When some types of the patterns do not exist, their corresponding cache arrays will be used for other types of patterns.

For the L-H data access patterns, its desired cache array has large capacity because the data block has little opportunity to be reused in a short future. If the capacity is small, a

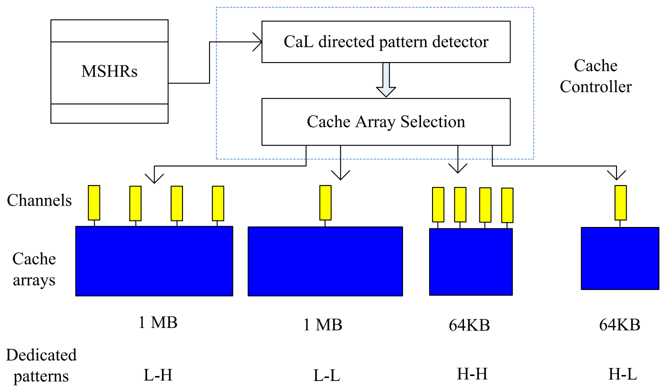

Fig. 18. Pattern directed selective cache array.

large number of conflict misses will occur. Also, more channels are needed to meet the high concurrency requirement. For the L-L patterns, large capacity and low concurrency are required. For the H-H patterns, smaller capacity and more channels are needed. For the H-L patterns, smaller capacity and fewer channels are provided. It is sensible to use on-demand capacities to accommodate different working sets of diverse patterns to reduce cache pollution, and it is sensible to let on-demand channels to move data in parallel without destroying their inherent parallelisms.

As shown in Fig. 19, the data block movements of the applications have been reduced on average by 21.1 percent, at best by 32.4 percent. That is, given a fixed block size K and access number N, selective cache array decreases the number of data movements g, and thus CaL has been improved. Fig. 20 shows that system performance IPC can be improved on average by 12.7 percent, and at best by 16.6 percent for the bwaves benchmark.

We have also used L and C to conduct the same classification and optimization. As shown in Fig. 19 and 20, the two types of classification have significant performance difference, which again verify that CaL can reflect concurrency aware locality more accurately than L.

7 Discussions

In this section we further discuss the usage of CaL, and present more future optimization directions.

7.1 Usage of the CaL Model

Memory wall is the killer of computing system performance. Memory wall refers to the performance gap between

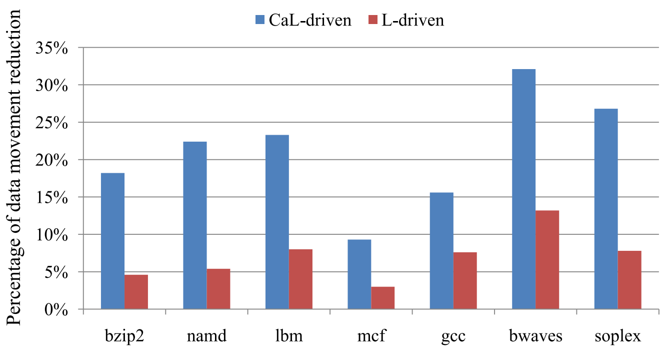

Fig. 19. Reduction of g (the number of data blocks need to move) due to selective cache array.

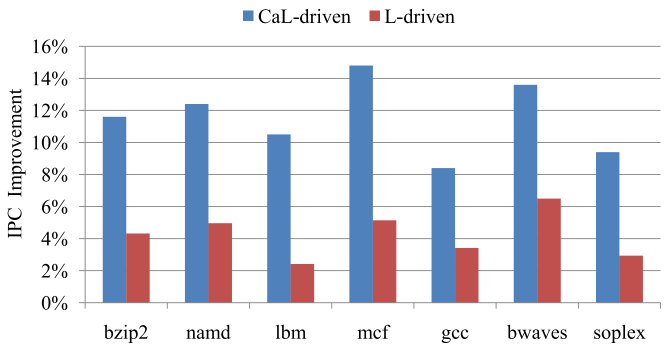

Fig. 20. Speedup ratio due to selective cache array.

CPU and memory. Memory wall is caused by latency wall and bandwidth wall. Latency wall is caused by the large performance gap between different layers of a memory hierarchy, while bandwidth wall refers to the limited number of off-chip pins can be manufactured for transferring data.

The CaL model and its associated two theorems can drive many optimizations for memory performance improvements, which are not limited to the case studies given above. Theorem 1 and 2 can be applied to each layer of a memory hierarchy with that layer's APC, CaL, AMAT, C and B. The default metrics for whole memory system are measured in L1. But all the five metrics can be applied and measured at each layer of a memory hierarchy. Therefore, layered optimization can be done by following the same theorems.

Given a fixed bandwidth B, if CaL is not improved, higher concurrency C will result in longer AMAT. This is due to the delay caused by contention on shared resources, including queuing delay, bus delay, etc. For each of the 28 benchmarks from SPEC CPU2006 [45], we sampled 10 billion instructions, and measured the AMAT and C values. The 28 benchmarks are ranked in terms of AMAT (from short to long) and C (from high to low), respectively. As shown in Fig. 21, most of the applications are close to the diagonal. We can see that without optimizations, current applications rarely achieve short AMAT and strong C simultaneously. For example, benchmark $458.sjeng$ has highest C but has long AMAT; benchmark $998.specrand$ has least AMAT but has low C. As we shown in Section 5, the situation is much worse in big data applications. CaL is timely important for big data applications.

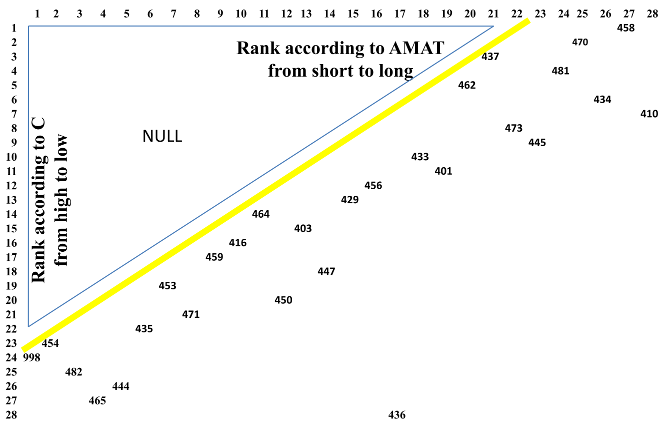

Fig. 21. SPEC CPU2006 benchmarks rarely achieve small AMAT and strong C simultaneously.

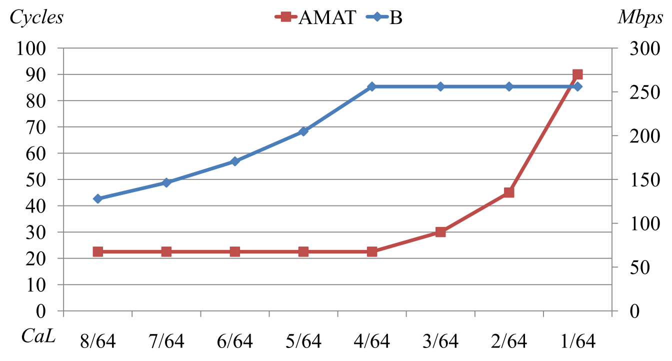

Fig. 22. Impact of CaL on AMAT.

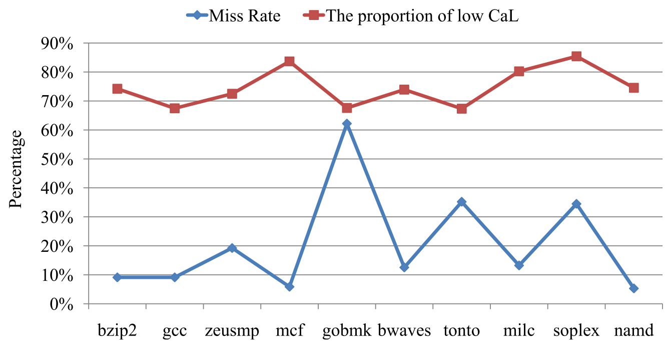

Fig. 23. Inconsistence between Miss rate and CaL locality.

Theorem 2 presents insight for reducing AMAT for large working set applications which are common in the big-data era. Applications with a large working set need to access low level memory to get their data. In this case, the cache miss rate is very large. To amortize single miss latency caused by the widening memory wall, modern processors allow multiple concurrent outstanding misses; the number of concurrent misses can be represented by C in CaL. However, bandwidth B provides an upper bound on the number of outstanding misses. As shown in Fig. 22, when locality is low and bandwidth approaches its upper bound, any further decrease in CaL will make AMAT increase significantly. Additionally, increasing concurrency C of misses at this time will further increase AMAT. The insights here are that: (1) Enabling higher memory concurrency does not always bring better performance. (2) When the bandwidth wall impact is significant, optimizations should focus on improving CaL rather than increasing C.

Improving CaL can reduce bandwidth utilization allowing higher miss concurrency. However, such optimizations cannot be done with miss rate since it does not directly relate to concurrency or bandwidth. For example, Fig. 23 shows miss rate and the percentage of low CaL. A low CaL means each cache miss will require a cache block to be fetched causing a direct effect on bandwidth. We can see that the two curves are inconsistent, and we can conclude that improving miss rate cannot provide the same benefits as improving CaL.

7.2 Future CaL-Driven Optimization Directions

The three case studies in Section 5 only exemplify CaL-driven hardware optimizations. Other optimization directions exist to be explored such as:

Evaluating and enabling current techniques: During the past decades, many 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 of CaL optimizations. One instance is disk defragmenter [25] which is designed to increase access speed by rearranging files stored on a disk to occupy contiguous storage locations. These existing techniques can be evaluated and selectively enabled with the newly proposed CaL and its associated theorems introduced in this study.

Labeling and reorganizing data: Based on CaL, data can be classified into high CaL data (termed as dense data) and low CaL data (termed as sparse data). Justin et al. used record access frequencies to identifying hot and cold data [32]. We plan to cluster the hot data to reduce the working set size to let this type of data reside in on-chip memory. We also plan to cluster the cold data to reduce the bandwidth requirement; however, it may not be necessary to let them reside in the on-chip memory. As such, for dense-hot, dense-cold, sparse-hot and sparse-cold data, algorithms can be designed to adaptively process them, respectively.

8 Related Work

The study of locality goes back about half a century ago when Denning established working set theory in 1968 [12], [13], and Mattson proposed LRU stack distance in 1970 [37]. Working set and reuse distance have served for a variety of locality analysis. In the CaL model, working set size can be denoted by parameter g, and the impact of reuse distance is also considered. The working set theory presented a misconception that good locality is deemed high hit rate which is no longer always true due to the prevalence of data concurrency in modern hierarchical memory systems. Fortunately, the CaL model has avoided the side-effects.

Following the working set theory and reuse distance concept, specific metrics were proposed. Weinberg presented weighted stride in HPC application optimization [51]. Berg et al. [4] and Gu et al. [20] proposed to quantify locality based on measuring the change of miss rates or reuse distances. Anghel et al. proposed to use a probability distribution of reuse distance to quantify locality [1]. Although these metrics are based on heuristics and lack of formal mathematical definition [21], we have considered their advantages when we developed the CaL model.

Ding and Xiang successfully discussed the relationships among the five locality metrics: footprint, inter-miss time, volume fill time, miss ratio and reuse distance [6], [14]. Although being based on a sequential data access trace, their consideration is insightful for us to develop a concurrency-aware model and link general memory metrics together.

Jiang et al. realized the shortcoming of the traditional reuse distance for sequential programs running on unicore processors and proposed "concurrent reuse distance for multithreaded programs [27]. The CaL model can also be used in a multi-threaded environment, but has a clearer formulation and is easier-to-use.

Gupta et al. quantified the data access locality as a conditional probability [21], making the locality concept no longer elusive. Gupta's work enlightened us to employ conditional expectation for modeling CaL. However, the research also assumes an ordered list of data access traces, where data access concurrency and the cost of data movement are not considered. Moreover, their model requires a significant amount of computation time, particularly for a large trace. However, the CaL model introduced in our work does not need trace-based computation, thus can be measured and used online to capture the concurrency feature.

Aggarwal et al. explored the significant impact of memory performance (in terms of bandwidth and latency) on the performance of three specific types of applications [3]. Their experimental results fit well to two theorems in our work, which clearly show the elusive relationships within complex memory systems and can motivate many new optimization methods.

Memory Level Parallelism (MLP) [11] has been proposed in recent years. MLP is a common memory metric and is the average number of long-latency main memory outstanding accesses when there is at least one such outstanding access. MLP does not consider locality. With MLP, we cannot derive the relationship between general memory metrics directly. In this paper, we have generalized MLP to data concurrency, C, that occurs in all the levels of the memory hierarchy, and CaL is proposed for quantifying C-aware locality.

C-AMAT considers locality and concurrency at the same time in optimizing data access time [35], [46]. Although the relationship between locality and concurrency is not explicitly considered in C-AMAT, it enlightens us to think about the trueness of the conventional rule of thumb that high locality is deemed low miss rate. Realizing that locality cannot completely be reflected by miss rate, we first distinguish between hit locality and miss locality and then redefined locality.

Compared to previous attempts on quantifying locality, the advantages of the newly proposed CaL model include: (1) CaL is a quantitative model with feasible measurements and thus can be incorporated into engineering practices; (2) the CaL model has a clear mathematical and physical meaning, as shown through the theorems, with which we can show the connection among bandwidth, concurrency, and locality; and (3) the CaL model considers the data concurrency and the cost of data movement, which is important for data-intensive computation in practice, not only in terms of performance, but also in terms of power consumption.

9 Conclusion

Big data applications have put unprecedented pressure on memory systems. Utilizing data locality and concurrency are two vital methods to boost memory system performance. However, while locality is a well studied subject, concurrency is a relatively new optimization method. They have been mostly optimized separately in current theoretic analysis and engineering practice. How does locality and concurrency influence each other and how to reach a balanced optimization of locality and concurrency is still a subject of study. The key contribution of this paper is an accurate definition and analytical formulation of locality with the consideration of data concurrency. Therefore, existing locality optimization methods can be applied to consider data concurrency and to reach a balanced optimization of locality and concurrency. We have derived the impact of the newly proposed concurrency-aware locality (CaL) model on memory system performance. We observed that high locality does not necessarily imply high cache hit rate (HR), since the data accesses that have high reuse opportunity may all be outstanding cache misses. The bandwidth and working set size are significantly impacted by CaL.

The newly proposed CaL model, with its combined power of locality and concurrency, can be used in various memory optimizations. We conducted four CaL-driven case studies in Section 5 and Section 6, with one case study to show the effect of CaL on big data applications, and three on different design approaches in dynamic block size, dynamic MSHR structure, and selective cache array, to demonstrate how to apply CaL in performance optimization. Experimental results show that CaL-driven optimizations have achieved significant improvement on final performance, a 3x speedup in the K-means big data benchmark. These three case studies given in Section 6 provide three different design approaches to improve memory performance and are important research results on their own right.

As future work, we plan to use CaL for the performance optimization of data-intensive applications on GPGPU and accelerators. In addition to the open problems and future works mentioned in Section 6, more CaL-driven optimizations can be conducted to significantly improve performance under highly concurrent environments.

Acknowledgments

This work is supported in part by the National Science Foundation, under grant CCF-1536079, CNS-1162540 and grant CCF-0937877, the National High Technology Research and Development Program(“863” Program) of China, under grant 2015AA015303, the National Natural Science Foundation of China, under grant 61521092, 61272132 and 61772497, and State Key Laboratory of Computer Architecture foundation under grant CARCH2601.

References

[1] A. Anghel, G. Dittmann, R. Jongerius, and R. Luijten, “Spatio-temporal locality characterization,” in *Proc. Workshop Near-Data Process*, 2013, pp. 1–5.

[2] A. Agarwal, K. Roy, and T. N. Vijaykumar, “Exploring high bandwidth pipelined cache architecture for scaled technology,” in *Proc. Conf. Des. Autom. Test Europe*, 2003, Art. no. 10778.

[3] A. H. A. Badawy, A. Aggarwal, D. Yeung, and C. W. Tseng, “Evaluating the impact of memory system performance on software prefetching and locality optimizations,” in *Proc. Int. Conf. Supercomputing*, 2001, pp. 486–500.

[4] E. Berg and E. Hagersten, “Fast data-locality profiling of native execution,” *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 33, no. 1, pp. 169–180, 2005.

[5] N. Binkert, et al., “The gem5 simulator,” *ACM SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, 2011.

[6] C. Ding and X. Xiang, “A higher order theory of locality,” in *Proc. ACM SIGPLAN Workshop Memory Syst. Perform. Correctness*, 2012, vol. 48, pp. 68–69).

[7] S. Byna, Y. Chen, and X. H. Sun, “Taxonomy of data prefetching for multicore processors,” *J. Comput. Sci. Technol.*, vol. 24, no. 3, pp. 405–417, 2009.

[8] M. J. Cade and A. Qasem, “Balancing locality and parallelism on shared-cache mulit-core systems,” in *Proc. IEEE Int. Conf. High Perform. Comput. Commun.*, 2009, pp. 188–195.

[9] J. F. Cantin, M. H. Lipasti, and J. E. Smith, “Improving multiprocessor performance with coarse-grain coherence tracking,” in *Proc. Int. Symp. Comput. Archit.*, 2005, vol. 33, pp. 246–257.

[10] D. Chandra, F. Guo, S. Kim, and Y. Solihin, “Predicting inter-thread cache contention on a chip multi-processor architecture,” in *Proc. Int. Symp. High-Perform. Comput. Archit.*, 2005, pp. 340–351.

[11] Y. Chou, B. Fahs, and S. Abraham, “Microarchitecture optimizations for exploiting memory-level parallelism,” in *Proc. Int. Symp. Comput. Archit.*, 2004, vol. 32, Art. no. 76.

[12] P. J. Denning, "The working set model for program behavior," *Commun. ACM*, vol. 26, no. 1, pp. 43–48, 1968.

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

[14] C. Ding and Y. Zhong, "Predicting whole-program locality through reuse distance analysis," *ACM Sigplan Notices*, vol. 38, no. 5, pp. 245–257, 2003.

[15] J. Dongarra, K. London, S. Moore, P. Mucci, and D. Terpstra, "Using PAPI for hardware performance monitoring on Linux systems," in *Proc. Conf. Linux Clusters: The HPC Revolution*, National Center for Supercomputing Applications (NCSA), University of Illinois, Urbana, EL, pp. 25–27, June 2001.

[16] L. Adhianto, et al., "HPCTOOLKIT: Tools for performance analysis of optimized parallel programs," *Concurrency Comput. Practice Experience*, vol. 22, no. 6, pp. 685–701, 2009.

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

[18] L. Wang, J. Zhan, C. Luo, and Y. Zhu, "BigDataBench: A big data benchmark suite from internet services," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, 2014, pp. 488–499.

[19] C. Dubnicki and T. J. Leblanc, "Adjustable block size coherent caches," in *Proc. Int. Symp. Comput. Archit.*, 1992, vol. 20, pp. 170–180.

[20] X. Gu, I. Christopher, T. Bai, C. Zhang, and C. Ding, "A component model of spatial locality," in *Proc. Int. Symp. Memory Manage.*, Jun. 2009, pp. 99–108.

[21] S. Gupta, P. Xiang, Y. Yang, and H. Zhou, "Locality principle revisited: A probability-based quantitative approach," *Inst. Electric. Electron. Eng.*, vol. 73, no. 7, pp. 1011–1027, 2012.

[22] G. Hamerly, E. Perelman, and B. Calder, "How to use simpoint to pick simulation points," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 31, no. 4, pp. 25–30, 2004.

[23] N. Hardavellas, I. Pandis, R. Johnson, N. Mancheril, A. Ailamaki, and B. Falsafi, "Database servers on chip multiprocessors: Limitations and opportunities," in *Proc. Biennial Conf. Innovative Data Syst. Res. OAI.*, 2007, pp. 79–87

[24] A. Inoue,. "On the worst conditional expectation," *J. Math. Anal. Appl.*, vol. 286, no. 1, pp. 237–247, 2003.

[25] C. Jensen, "Fragmentation, the Condition, the Cause, the Cure," Exec Sftwa, 2004.

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

[27] Y. Jiang, E. Z. Zhang, K. Tian, and X. Shen, "Is reuse distance applicable to data locality analysis on chip multiprocessors?" in *Proc. Compiler Construction Int. Conf.*, 2010, vol. 6011, pp. 264–282.

[28] A. Kagi, J. R. Goodman, and D. Burger, "Memory bandwidth limitations of future microprocessors," in *Proc. Int. Symp. Comput. Archit.*, 1996, vol. 24, pp. 78–89.

[29] K. Kennedy and K. S. Mckinley, "Optimizing for parallelism and data locality," in *Proc. ACM Int. Conf. Supercomputing*, 1997, pp. 323–334.

[30] D. Kroft, "Lockup-free instruction fetch/prefetch cache organization," in *Proc. 25 Years Int. Symp. Comput. Archit.*, 1998, pp. 195–201.

[31] G. Kurian, O. Khan, and S. Devadas, "The locality-aware adaptive cache coherence protocol," in *Proc. Int. Symp. Comput. Archit.*, 2013, vol. 41, pp. 523–534.

[32] J. J. Levandoski, P. A. Larson, and R. Stoica, "Identifying hot and cold data in main-memory databases," in *Proc. IEEE Int. Conf. Data Eng.*, 2013, pp. 26–37.

[33] D. Levinthal, "Performance analysis guide for Intel Xeon 5500 processors," *Intel Perform. Anal. Guide*, 2009.

[34] Y. Li, B. Lee, D. Brooks, and Z. Hu, "CMP design space exploration subject to physical constraints," in *Proc. 25th Int. Symp. High-Perform. Comput. Archit.*, 2006, pp. 17–28.

[35] Y. Liu and X. Sun, "Reevaluating data stall time with the consideration of data access concurrency," *J. Comput. Sci. Technol.* vol. 30, no. 2, 2015, Art. no. 227245.

[36] M. M. K. Martin, M. D. Hill, and D. J. Sorin, "Why on-chip cache coherence is here to stay," *Commun. ACM*, vol. 55, no. 7, pp. 78–89, 2012.

[37] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger, "Evaluation techniques for storage hierarchies," *IBM Syst. J.*, vol. 9, no. 2, pp. 78–117, 1970.

[38] J. Meng, J. W. Sheaffer, and K. Skadron, "Exploiting inter-thread temporal locality for chip multithreading," in *Proc. IEEE Int. Symp. Parallel Distrib. Process.* 2010, pp. 1–12.

[39] S. P. Muralidhara, L. Subramanian, O. Mutlu, M. Kandemir, and T. Moscibroda, "Reducing memory interference in multicore systems via application-aware memory channel partitioning," in *Proc. IEEE/ACM Int. Symp. Microarchitecture*, 2011, pp. 374–385.

[40] D. A. Patterson, "Latency lags bandwith," *Commun. ACM*, vol. 47, no. 10, pp. 71–75, 2004.

[41] R. Prasad, C. Dovrolis, M. Murray, and K. Claffy, "Bandwidth estimation: Metrics, measurement techniques, and tools," *IEEE Netw.*, vol. 17, no. 6, pp. 27–35, Nov./Dec. 2003.

[42] J. A. Rivers, G. S. Tyson, E. S. Davidson, and T. M. Austin, "On high-bandwidth data cache design for multi-issue processors," in *Proc. 30th IEEE/ACM Int. Symp. Microarchitecture*, 1997, pp. 46–56.

[43] B. M. Rogers, A. Krishna, G. B. Bell, K. Vu, X. Jiang, and Y. Solihin, "Scaling the bandwidth wall: Challenges in and avenues for CMP scaling," in *Proc. Int. Symp. Comput. Archit.* 2009, vol. 37, pp. 371–382.

[44] S. Somogyi, T. F. Wenisch, A. Ailamaki, and B. Falsafi, "Spatio-temporal memory streaming," *ACM SIGARCH Comput. Archit. News*, vol. 37, no. 3, pp. 69–80, Jun. 2009

[45] C. D. Spradling, "SPEC CPU2006 benchmark tools," *ACM Sigarch Comput. Archit. News*, vol. 35, no. 1, pp. 130–134, 2007.

[46] X. H. Sun and D. Wang, "Concurrent average memory access time," *IEEE Comput.*, vol. 47, no. 5, pp. 74–80, 2014.

[47] J. Torrellas, H. S. Lam, and J. L. Hennessy, "False sharing and spatial locality in multiprocessor caches," *IEEE Trans. Comput.*, vol. 43, no. 6, pp. 651–663, Jun. 1994.

[48] J. Tuck, L. Ceze, and J. Torrellas, "Scalable cache miss handling for high memory-level parallelism," in *Proc. IEEE/ACM Int. Symp. Microarchitecture*, 2006, pp. 409–422.

[49] H. M. Levy, J. L. Lo, J. S. Emer, R. L. Stamm, S. J. Eggers, and D. M. Tullsen, "Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor," *Proc. ACM Sigarch Comput. Archit. News*, vol. 24, no. 2, pp. 191–191, 1996.

[50] D. Wang and X. H. Sun, "APC: A novel memory metric and measurement methodology for modern memory systems," *IEEE Trans. Comput.*, vol. 63, no. 7, pp. 1626–1639, Jul. 2014.

[51] J. Weinberg, M. O. Mccracken, E. Strohmaier, and A. Snavely, "Quantifying locality in the memory access patterns of HPC applications," in *Proc. ACM/IEEE SC Conf. Supercomputing*, 2005, p. 50.

[52] 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.

[53] Y. Zhong, X. Shen, and C. Ding, "Program locality analysis using reuse distance," *ACM Trans. Programm. Languages Syst.*, vol. 31, no. 6, pp. 1–39, 2002.

Yuhang Liu received the PhD degree in computer science from the Beihang University, Beijing, China. He is an associate professor in the Institute of Computing Technology (ICT), Chinese Academy of Sciences. He has been a postdoctoral researcher in the Scalable Computing Software laboratory, Illinois Institute of Technology (IIT), Chicago. He is a member of the CCF, ACM and the IEEE. His research interests include high performance computing, computer architecture, 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 in the Computer Science, IIT. He is the director of the Scalable Computing Software laboratory, 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, the Argonne National Laboratory. Before joining IIT, he worked at DoE Ames National Laboratory, ICASE, NASA Langley Research Center, 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.