Thread-Aware Adaptive Prefetcher on Multicore Systems: Improving the Performance for Multithreaded Workloads

PENG LIU, Zhejiang University, State Key Laboratory of Mathematical Engineering and Advanced Computing

JIYANG YU, Huawei Technologies Co., Ltd.

MICHAEL C. HUANG, University of Rochester

Most processors employ hardware data prefetching techniques to hide memory access latencies. However, the prefetching requests from different threads on a multicore processor can cause severe interference with prefetching and/or demand requests of others. The data prefetching can lead to significant performance degradation due to shared resource contention on shared memory multicore systems. This article proposes a thread-aware data prefetching mechanism based on low-overhead runtime information to tune prefetching modes and aggressiveness, mitigating the resource contention in the memory system. Our solution has three new components: (1) a self-tuning prefetcher that uses runtime feedback to dynamically adjust data prefetching modes and arguments of each thread, (2) a filtering mechanism that informs the hardware about which prefetching request can cause shared data invalidation and should be discarded, and (3) a limiter thread acceleration mechanism to estimate and accelerate the critical thread which has the longest completion time in the parallel region of execution. On a set of multithreaded parallel benchmarks, our thread-aware data prefetching mechanism improves the overall performance of 64-core system by 13% over a multimode prefetch baseline system with two-level cache organization and conventional modified, exclusive, shared, and invalid-based directory coherence protocol. We compare our approach with the feedback directed prefetching technique and find that it provides 9% performance improvement on multicore systems, while saving the memory bandwidth consumption.

Categories and Subject Descriptors: C.1.0 [Processor Architectures]: General; C.1.2 [Multiple Data Stream Architectures(Multiprocessors)]; B.3.m [Miscellaneous]

General Terms: Design, Algorithms, Performance

Additional Key Words and Phrases: Data prefetcher, thread-aware, self-tuning, multithreaded, multicore

This work was supported in part by the Open Project Program of the State Key Laboratory of Mathematical Engineering and Advanced Computing under Grants No. 2014A08 and No. 2015A09; by Huawei Technologies Co., Ltd under Grant No. YB2014100047 and a gift; by the NSFC under Grants No. 60873112 and No. 61028004, by the National High Technology Research and Development Program of China under Grant 2009AA01Z109; and by the NSF under Grants No. 1217662, No. 1255729, No. 1314734, and No. 1351697. Extension of Conference Paper. The preliminary version of the article was published in the Proceedings of 2014 32nd IEEE International Conference on Computer Design (ICCD).

Author's addresses: P. Liu, College of Information Science and Electronic Engineering, Zhejiang University, Hangzhou, 310027, China, State Key Laboratory of Mathematical Engineering and Advanced Computing, Wuxi 214125, China; email: liupeng@zju.edu.cn; J. Yu, Department of Shannon Lab., Huawei Technologies Co., Ltd., Hangzhou, 310051, China; email: yujiyang@huawei.com; M. C. Huang, Department of Electrical and Computer Engineering, University of Rochester, Rochester, NY 14627-0231; email: michael.huang@rochester.edu.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.

© 2016 ACM 1544-3566/2016/03-ART13 \$15.00

DOI: <http://dx.doi.org/10.1145/2890505>

ACM Reference Format:

Peng Liu, Jiyang Yu, and Michael C. Huang. 2016. Thread-aware adaptive prefetcher on multicore systems: Improving the performance for multithreaded workloads. *ACM Trans. Archit. Code Optim.* 13, 1, Article 13 (March 2016), 25 pages.

DOI: <http://dx.doi.org/10.1145/2890505>

1. INTRODUCTION

Memory access latency has become one of the major bottlenecks to system performance. To hide memory access latency, researchers have proposed hardware data prefetching mechanisms. Prefetching refers to fetching data from memory into caches or a prefetching buffer before actually using them. In this way, the memory latency can be effectively hidden and the processors performance can be improved. Data prefetching techniques in conventional single-core processors have been proven useful. However, on a chip multiprocessor (CMP) system, all running threads usually share the last level cache (LLC) and off-chip memory. Memory requests from each core conflict with those from other cores, since prefetching requests need to traverse on-chip interconnect networks and the memory bus to arrive at memory banks as normal data requests. The presence of a prefetching engine would cause additional cache conflicts and memory bandwidth competition. The conflicts increase applications' execution time and limit the performance scaling of multicore processors.

Prefetching-caused contention of shared resources can be reduced if the shared resources are smartly managed. A lot of works have focused on the memory controller-based request arbitration [Kim et al. 2011; Mutlu and Moscibroda 2008; Nesbit et al. 2006], operating system-based thread scheduling [Jaleel et al. 2012; Kim et al. 2010; Zhuravlev et al. 2010a, 2010b], feedback-directed prefetching aggressiveness throttling [Ebrahimi et al. 2011a, 2009a, 2009b; Srinath et al. 2007], and other methods managing the shared LLC and main memory. However, most previous techniques are designed for multiprogrammed workloads, and prefetching on multithreaded parallel applications is confronted with new problems.

Generally multiple threads on multicore systems share data among themselves. In order to maintain consistency, if data in a shared cache is replaced by prefetching data, then all its sharers should invalidate their local copy of the cache block. By analyzing the impact of prefetching on the shared resource contention in multithreaded parallel applications, we found that prefetching requests cause interthread invalidations and lead to increased miss rates. Figure 1 shows the prefetching-caused interthread invalidations of multithreaded applications (Section 4 describes the experimental environment). The white-column on the left side indicates the proportion of shared cache blocks in all cache blocks; this value is sampled and averaged during the runtime of each application. The gray column on the right-hand side indicates the proportion of prefetching-caused invalidations in all L2 cache invalidations (caused by demand misses and prefetching requests). As shown in this figure, for some applications the percentage of shared cache blocks is high, and the large amount of prefetching-caused interthread invalidations will cause severe performance degradation.

In order to avoid such prefetching-caused interthread invalidations, we propose a filtering mechanism [Yu and Liu 2014] to filter such prefetching requests that cause interthread invalidations (attacking prefetches, as defined in Section 3). Meanwhile, in order to adapt to the runtime features of multithreaded parallel applications and reduce prefetching-caused contention on shared resources, we develop a thread-aware data prefetching mechanism based on low-overhead runtime information to tune prefetching aggressiveness, mitigating the resource contention in the shared memory multicore systems. Due to the properties of multithreaded applications, there is always a thread that is on the critical path of execution. We define the *critical thread* the one with the

Fig. 1. Prefetching-caused interthread invalidations.

Fig. 2. Basic prefetching structure.

longest completion time in the parallel region and propose a limiter thread acceleration mechanism to estimate and accelerate the critical thread.

Our contributions are summarized below.

- (1) We first propose a thread classifying directed (TCD) approach to adjust prefetching parameters dynamically based on the runtime information for each thread to reduce shared resource contention.
- (2) We employ an attacking prefetch filter (APF) which can be used to avoid prefetching-caused interthread invalidations.
- (3) We develop a limiter thread acceleration (LTA) mechanism to estimate and accelerate the critical threads of a multithreaded application that are on the critical paths of execution.
- (4) We extensively evaluate the proposed thread-aware adaptive data prefetcher on a 64-core system using multithreaded workloads from PARSEC [Bienia et al. 2008], SPLASH-2 [Woo et al. 1995], and some scientific applications.

The article is organized as follows. Section 2 describes the basic prefetching structure; Section 3 gives the proposed thread-aware prefetching mechanism in detail; Section 4 presents our evaluation methodology; Section 5 analyzes the experimental results; Section 6 outlines related work; and, finally, Section 7 concludes the article.

2. BASIC PREFETCHING STRUCTURE

The basic prefetching engine is a single-core, multimode prefetcher (MDP). It consists of a memory access pool (MAP) for recording history information and a pattern table (PTB) for the maintenance of predicted patterns. Both sequential and chained data address access patterns can be detected by the MDP. The structure of MAP is shown in Figure 2(a). The *Missaddr* field records the address of each cache miss. The *Stride* field records the value of the stored miss address subtracted by the newly missed address (for sequential streams) or the newly returned value (for chained streams).

Fig. 3. System overview.

The *Status* field is used for the training process of stream patterns. The *Type* field is used to distinguish between sequential patterns and chained patterns. The structure of PTB is shown in Figure 2(b). The *First* and *Last* fields in PTB record the first and last addresses of previous prefetching requests. The *Stride* field records the stride of the sequential stream for sequential patterns or the offset of next field in the linked list for chained patterns.

During program execution, the first address of a cache miss is recorded in a MAP entry indexed by the missed address. We use Address Prediction Heuristics, which is borrowed from the content-directed prefetching (CDP) approach [Cooksey et al. 2002], to determine if a return value is a potential address value. The Address Prediction Heuristics is based on an assumption of data structure that most data addresses tend to share common high-order bits. It compares the return value with the missed address; if the two addresses have the same high-order bits, then the return value is considered as a address like value. Then the *Type* field is set, implying a chained pattern training. Otherwise, the *Type* field is unset, implying a sequential pattern training. The most important distinction between our approach and CDP for chained stream prefetching is that prefetching requests in our mechanism are not immediately issued when an address like value is returned. Instead, the value will be used for the training of a chained stream, and only if a chained stream has been found will the corresponding prefetching requests be issued. This approach is used to avoid arbitrary issuing of every address like value which is of low accuracy. Whenever a new stream pattern is formed or a demand-request hits on an existing stream, a prefetching request is ready for issuing. The prefetching engine will check whether the prefetching address of the request data is already in the cache or if the Miss Status Holding Register (MSHR) has recorded the data request. If it is neither, then a prefetching request is finally issued.

We have evaluated the performance of MDP and a conventional stream prefetcher [Palacharla and Kessler 1994; Ganusov and Burtscher 2005]. The MDP engine performs 6% better than the stream prefetcher within a single-core system and performs 2% better within a multithreaded 64-core shared memory system. Thus we choose MDP as a reasonable baseline in our work.

3. THREAD AWARE PREFETCHING SYSTEM

Figure 3 shows the multicore system modeled in this article. Each node has private L1 instruction and data caches, and all nodes share a physically distributed LLC with integrated directory and communicate with each other through routers which are connected by the on-chip interconnection fabric. Caches are shared under a Modified, Exclusive, Shared, and Invalid- (MESI) based directory coherence protocol implementation. The prefetching engine on each node is based on the multimode prefetcher described in

Section 2. In order to exploit performance of the multithreaded environment, we made the following improvements based on the single-core MDP mechanism to build the thread-aware adaptive prefetcher (TAP): (1) a TCD prefetcher tuning unit, which can adjust prefetching engines dynamically according to the runtime information of multithreaded applications, to reduce shared resource contention; (2) an APF to avoid prefetching-caused interthread invalidations; and (3) a LTA module to estimate and accelerate the critical threads.

In the following, we discuss the rationale for thread categories in Section 3.1, the approach to prefetching adjustment in Section 3.2, the principle of an attacking prefetch filter in Section 3.3, and the mechanism for accelerating the limiter threads in Section 3.4.

3.1. Prefetching-Aware Thread Classification

The resources which are shared among threads include on-chip cache, off-chip main memory, memory bus, and so on. In a multithreaded application, multiple threads are used for processing each subset of data or performing different tasks, and thus they have different data requirements, making resource contention complicated. To reduce resource contention among the complex combination of threads, we classify threads by their memory behavior and characteristics of prefetching and adjust each thread's prefetcher algorithms according to the classifying results.

As mentioned above, contention for shared resources leads to a larger number of memory accesses and longer memory access time, and the presence of prefetching engine will even exacerbate this situation. In order to evaluate the shared resources contention among threads, we use the metric *tMH* (total time for miss handling), which is normalized to the result with no prefetching, to indicate prefetching-caused impact on system performance.

$$tMH = \sum_{i=0}^{total\ misses} handle\ time\ for\ ith\ miss. \quad (1)$$

Additionally, we use the following four metrics, which indicate different aspects of threads' memory access behavior, to evaluate the shared resource contention.

- (1) **MPKI** (misses per thousand instructions). This metric indicates the contention on limited cache capacity. It directly reflects thread's memory access intensity.
- (2) **PPKI** (prefetch requests per thousand instructions). This metric indicates the aggressiveness of prefetching for shared caches.
- (3) **PA** (prefetch accuracy). This metric reflects the effect of prefetch requests.
- (4) **PL** (prefetch lateness). This metric can be used to evaluate the timeliness of prefetches issuing.

To classify threads into several types, we need to analyze their runtime behavior. During the phase of program feature analysis, we have sampled all 24 applications (Section 4 for details) for the information of *tMH*, **MPKI**, **PPKI**, **PA**, and **PL**. A clustering algorithm (k-means [MacQueen 1967]) is used to analyze the data and decide the threshold value of the metrics among different thread types. Other basics for thread classification include the observation of data and our experimental experience.

According to the results of normalized *tMH*, we divide applications into three collections, as shown in Table I.

- (1) The applications in the first collection have a *tMH* increase of less than 10% (*tMH* < 1.1). This means that the influence of prefetching is slight, and these applications

Table I. Thread Classifying Policy

<i>tMH</i>	Metrics				Type of threads	Applications
	<i>MPKI</i>	<i>PPKI</i>	<i>PA</i>	<i>PL</i>		
<110%	<100	–	–	–	LM	blackscholes, freqmine, swaptions, jacobi, mp3d, shallow, tspo, fmm, radix
110%	>100	<200	–	–	MMLP	canneal, vips, em3d, radiosity, raytrace
~		>200	>0.45	<0.07	MPHA	barnes, cholesky, fft, lu, water-spatial
130%				>0.07	MPHAL	dedup
>130%	>100	>200	<0.45	–	MPLA	fluidanimate, streamcluster, ferret, ocean

have small memory requirements (also indicated by the metric *MPKI* lower than 100). We classify these applications as Light Memory requirement threads (*LM*).

- (2) In the second collection of normalized *tMH* ($1.1 < tMH < 1.3$), all applications have an *MPKI* higher than 100, which means these applications have larger memory requirements. If the applications have few data address patterns which can be predicted by the prefetching engine, then the number of prefetch requests would be small (*PPKI* < 200). We classify these applications as Mass Memory and Light Prefetching requirement threads (*MMLP*). If the number of prefetch requests is large (*PPKI* > 200) and the prefetch accuracy is high (*PA* > 0.45), then we classify the applications into one of two categories, that is, Mass Prefetching High Accuracy threads (*MPHA*) and Mass Prefetching and High Accuracy/Lateness threads (*MPHAL*).
- (3) In the third collection of normalized *tMH* ($tMH > 1.3$), the applications have large memory requirements (*MPKI* > 100) and proper data address patterns for prefetching engine (*PPKI* > 200), but the prefetch accuracy is low (*PA* < 0.45). A large number of inaccurate prefetch requests can cause serious shared resource wastes and increase the *tMH*. We classify these applications as Mass Prefetching Low Accuracy threads (*MPLA*).

We are inspired by the results that the impact of prefetching on shared resource contention is closely related to thread's memory behavior and prefetching performance. Therefore it is reasonable to classify threads and manage prefetchers according to these metrics. The threads classifying policy is shown in Table I. The meaning of classification for each application is that the probability of one application to perform as the corresponding type of threads is high. It should be noted that the thread classification is not static—it is changing during the runtime of threads. The real-time feedback information is collected by a hardware module in every statistical interval. We use the number of cache replacements (1024 by default) to define the statistical interval for adjustment. Since the purpose of prefetching adjustment is to reduce resource conflicts, and the frequency of cache replacements essentially reflects the cache conflict intensity, this approach can change the adjustment automatically depending on the runtime degree of resource contention.

3.2. Thread Classifying Directed Adjustment

Based on the thread classifying policy, we can adjust each thread's prefetcher algorithms according to the classifying results. As different types of threads have their own characteristics, we carry out the corresponding prefetching adjustment operations with the following considerations:

For *LM*: Since it is characterized by its sparse memory access requests, there is no need for such threads to worry about the problem of memory resource contention with other threads.

Table II. Prefetching Adjustment

	Degree	Distance
<i>LM</i>	—	—
<i>MMLP</i>	<i>turn off</i>	
<i>MPHA</i>	<i>increase</i>	—
<i>MPHAL</i>	—	<i>increase</i>
<i>MPLA</i>	<i>decrease</i>	<i>decrease</i>

Table III. Levels of Prefetcher Aggressiveness

Aggressiveness Level	Low → High				
Prefetch Degree	1	2	4	8	
Prefetch Distance	4	8	16	32	64

For **MMLP**: While it makes many memory accesses, the number of prefetch requests is small. That means the data address pattern of such threads does not fall into a pattern that can be predicted by the prefetcher. Thus we can temporarily shut down the threads' prefetching engine to reduce its contention for the shared resources. The prefetcher will be started again with default configuration (degree = 2, distance = 8) in the next statistical interval, and thus the metrics will be updated.

For **MPHA**: Prefetch requests are issued intensively with a high prefetch accuracy and low prefetch lateness, showing that the prefetcher is having the best impact on its performance. Thus, if there is no memory resource contention with other threads, then it should tune up the aggressiveness of its prefetcher as much as possible to make full use of prefetching engines to achieve performance improvement.

For **MPHAL**: Prefetch accuracy and lateness are both high. This means that although the prefetcher has been predicting accurate addresses, the issuing of prefetch requests is not timely enough. In this case, it is necessary to tune up the prefetch distance to achieve a better issuing timeliness.

For **MPLA**: It issues a large number of inaccurate prefetch requests which can cause serious shared resources wastes. When there are other threads competing for memory resources, the **MPLA** threads prefetching aggressiveness should be tuned down.

To recap, the TCD prefetching adjustment strategies are listed in Table II.

We adjust prefetching aggressiveness by changing the distance and degree of prefetchers separately. The prefetch distance is the interval of data blocks between the data currently being prefetched and the data that the processor actually uses along the stream. The prefetch degree indicates how many continuous data blocks are fetched every time when a prefetch request is issued. Table III shows the aggressiveness at all levels from low to high corresponding to the prefetch distance and degree.

3.3. Reducing Prefetching-Caused Interthread Invalidation

The sharing of data caches among threads results in additional cache misses. This is partially caused by the replacement of other threads' useful cache blocks when demanded data or prefetched data are returned from memory system. This situation is more complicated under a MESI-based cache coherence protocol. Since L1 cache misses and prefetching requests will access LLC, they can cause the replacement of a data block which is shared by other threads. The result is that the data copies of that block in its sharer's L1 caches will be invalidated, thus causing demand misses when the data block is later accessed.

If a demand miss is caused by a prefetching request, then it is called a prefetching-caused invalidation. Such prefetching requests are called attacking prefetches. An

Fig. 4. Working principle of an attacking prefetch filter.

Fig. 5. An attacking prefetch is filtered for sequential streams.

attacking prefetch can be identified in the following case: If an L1 prefetch misses in the LLC and the prefetched data tries to replace a shared cache block.

To avoid degradation of performance, one solution is to abandon all attacking prefetches. An APF is used to filter such prefetches. The filter works when a prefetching data request returns from main memory and tries to replace a shared cache line in L2. As shown in Figure 4, if the returned data is a prefetched data, and it is going to replace a shared L2 cache line, then the replacement of the shared cache line will be terminated to avoid interthread invalidations.

However, this approach will break a chained stream when prefetching a linked data structure. Since a new node's address cannot be calculated until its predecessor is returned. In this article, we propose solutions for sequential and chained streams separately. For sequential streams all attacking prefetches are abandoned and the address of an attacking prefetch is recorded in the pattern table for later use. For example, in Figure 5, one thread's L1 prefetcher detects a stream at address A and begins to issue prefetches. When an attacking prefetch is detected at address $A + 3N$, the prefetch is abandoned but still recorded in the pattern table. Later access to $A + 3N$ will hit in the pattern table to let further prefetches be issued.

For chained patterns, L1 prefetching misses are not abandoned immediately if they are identified as attacking prefetches. Instead, they are issued, but the return value is used to maintain the linked pattern streams. As shown in Figure 6, the prefetch for node C is identified as an attacking prefetch and is issued. When the value is returned, it is used to calculate the next node address in the linked stream and then discarded.

When the prefetched data are returned, it is inserted into the Least Recently Used (LRU) stack of the corresponding cache set. Figure 7 is an example of cache replacement with four-way associative structure. Normally, the prefetched data are treated as a

Fig. 6. An attacking prefetch is used to calculate the next node address for chained streams.

Fig. 7. An example of cache replacement.

Table IV. Thresholds Used for Cache Replacement

Prefetch accuracy	<0.35	0.35–0.45	0.45–0.55	>0.55
Insert position	Tail (LRU)	Third	Second	Head (MRU)

demand request and inserted into the most recently used (MRU) position in the LRU stack. This mechanism is efficient if the prefetch accuracy is high. However, if a prefetch request is miss-predicted, which means the prefetched data will not be used, then the prefetched data will occupy a part of the cache footprint until it is replaced. This leads to a waste of cache resources.

We utilize one kind of adaptive insertion policy [Jaleel et al. 2008] for our cache replacement mechanism to act in concert with the proposed attacking prefetch filter. According to the prefetching aware thread classifying, the prefetch accuracy can be estimated when the prefetched data are returned. For high prefetch accuracy, the prefetched data are inserted into the MRU position as usual. Prefetched data with lower prefetch accuracy is inserted into the second or third position of the LRU stack. If the prefetch accuracy is very low, the prefetched data are inserted into the LRU position to reduce the waste of cache footprint. Table IV shows the thresholds used for our cache replacement policy.

3.4. Limiter Thread Acceleration

When multithreaded workloads are concurrently executed, the execution time is determined by the critical path. Figure 8 is an example of barrier synchronization; four threads are executed during the stage between *Barrier0* and *Barrier1*. The execution time of this stage is exactly the time that *Thread1* spends. This means that, during this stage, prefetching for *Thread0*, *Thread2*, and *Thread3* will not bring any benefit for the system performance [Ebrahimi et al. 2011b], or even additional memory bandwidth competition, and slows down the execution of the limiter thread. Taking into account the interdependent nature of multithreaded workloads, we propose a LTA mechanism, which tunes up the prefetching aggressiveness of limiter threads (*Thread1* in this example) to achieve shorter execution time. For the threads that are not on the critical path and have the longest waiting time for barrier synchronization (*Thread3* in this example), LTA mechanism will turn down the prefetching aggressiveness to avoid unnecessary memory bandwidth consumptions. We find that performance can be improved by exposing information about limiter threads to the prefetching engine, which uses this information to accelerate the limiter threads and save memory bandwidth consumption of other threads.

Fig. 8. Barrier example of multithreaded workloads.

We utilize the method described in Ebrahimi et al. [2011b] to estimate the limiter threads. Algorithm 1 illustrates the method in detail. To implement this algorithm, the runtime system maintains one counter (*TotalWaitTime*) per lock to accumulate the total cycles that threads wait in that lock's queue, and keeps one counter (*ThreadWaitTime*) per thread to track the number of cycles that one thread waits for a lock. To keep the track of each lock's waiting time, when a lock j is successfully acquired by thread i , the runtime system adds *ThreadWaitTime_i* to *TotalWaitTime_j*. The thread which is holding the lock with longest *TotalWaitTime* is determined as limiter threads. A bit-vector *LimiterThreadBit* is added to identify which thread is the limiter thread. Every statistical interval, the runtime system compares the accumulated lock queue waiting time of all locks and finds the lock with the longest waiting time during the previous interval. The system saves this lock as *LockLongest*. Then the thread which is holding *LockLongest* is determined as *OwnerLongest*, and the corresponding bit in the *LimiterThreadBit* is set. Finally, when a thread acquires *LockLongest*, the bit corresponding to the previous *OwnerLongest* is reset, the new *OwnerLongest* is recorded, and the bit for new *OwnerLongest* in *LimiterThreadBit* is set.

Another bit-vector *FasterThreadBit* is added to identify the thread that has the longest waiting time for barrier synchronization. Since the faster threads are not on the critical path, tuning down its prefetching aggressiveness can avoid unnecessary memory bandwidth consumption and will not affect the execution time. The faster thread can be estimated by an algorithm which is similar to Algorithm 1. For *FasterThreadBit*, the runtime system searches for the lock with the shortest waiting time during the previous interval and saves this lock as *LockShortest*. The thread which is holding *LockShortest* is determined as *OwnerShortest*, and the corresponding bit in the *FasterThreadBit* is set.

ALGORITHM 1: Runtime Limiter Thread Estimation

```

1 foreach successfully Lockj acquired by Threadi do
2   TotalWaitTimej += ThreadWaitTimei;
3   if Lockj == LockLongest then
4     OwnerLongest = Threadi;
5     Set bit for Threadi in LimiterThreadBit;
6   end
7 end
8 foreach LimiterEstimationInterval do
9   Update LockLongest;           // the lock with the longest waiting time
10  Update OwnerLongest;         // the thread that holds LockLongest
11  Update LimiterThreadBit;     // the bit for OwnerLongest
12 end
```

The bit-vectors are updated and informed to the prefetching engine in order to prioritize the limiter threads. For the limiter threads, the LTA mechanism will tune up prefetching aggressiveness in order to achieve the potential improvement on execution

time. However, for the faster threads, the LTA mechanism will turn down prefetching aggressiveness to avoid unnecessary memory bandwidth consumption. Note that prefetching aggressiveness is adjusted by both TCD and LTA, and thus there may be a conflict between the two decisions. Since changing the prefetching aggressiveness of the faster threads will not affect the execution time (e.g., TCD turning up the faster threads' prefetching aggressiveness), when a conflict occurs, LTA will turn over TCD's decision and dominate the adjustment.

3.5. Implementation

To implement the proposed method, we need hardware support for computing the metrics used in our scheme, classifying running threads, and tuning the parameters. Note that we use *tmh* to evaluate the contention of shared resources. This statistic is collected by the software simulator (see Section 4) during the phase of program feature analysis and is used to help design the thread classification strategy. However, it will not be collected during the runtime of the proposed mechanism. The hardware prefetching engine collects only four metrics (**MPKI**, **PPKI**, **PA**, and **PL**) to direct the runtime thread classifying.

MPKI: To collect the **MPKI**, two hardware counters are used. The first counter, *inst-total*, tracks the number of instructions that have been committed. The second counter, *miss-total*, tracks the number of demand misses. The metric **MPKI** is computed by taking the ratio of *inst-total* to *miss-total*.

PPKI: A hardware counter (*pref-total*) is used to track the number of prefetching requests. The metric **PPKI** is computed by taking the ratio of *pref-total* to *inst-total*.

PA: A hardware counter (*pref-used*) is used to count the number of useful prefetches. When a prefetched block is inserted into the cache, an additional bit (*pref-bit*) associated with that block is set. When an L1 cache block with *pref-bit* set is accessed by a demand request, the *pref-bit* is reset and *pref-used* is incremented. The metric **PA** is computed by taking the ratio of *pref-used* to *pref-total*.

PL: A prefetching request is late if a demand request for the prefetched address is generated before the prefetched data has arrived. A hardware counter *pref-late* is used to track the number of late prefetches. To track the history of prefetching requests, we add a hardware structure, Prefetch History Table (*PHT*) to each L1 cache. The *PHT* has 16 entries, each entry has an *addr-tag* to record the prefetch address, and a *hit-bit* to indicate that a demand request for this address was generated. If a prefetching request is issued, then its address is inserted to the *addr-tag* of *PHT* and the associated *hit-bit* is unset. If a demand request is generated, then its address will be compared with all *addr-tags* in *PHT*; if matched, then the associated *hit-bit* is set. When the prefetched data is returned and the *hit-bit* has been set, the *pref-late* counter is incremented. The metric **PL** is computed by taking the ratio of *pref-late* to *pref-total*.

Attacking prefetch filter: To filter an attacking prefetch, there is an additional bit (*pref-request*) in each MSHR entry of the L2 cache. There are 32 MSHRs for each node. When a prefetch request is issued, an MSHR entry is acquired, and the bit *pref-request* is set to indicate that this memory request is generated by the prefetcher. If the prefetched data returns and tries to replace a shared cache line in L2, then this prefetch will be identified as an attacking prefetch. The replacement of the shared cache line will be terminated to avoid interthread invalidations.

4. METHODOLOGY

We implement the proposed mechanism on a cycle-level, execution-driven, in-house simulator. The processor microarchitecture, the cache coherence, and communication

Table V. System Parameters

Processor core (MIPS32 ISA compatible)	
Fetch / Decode / Commit	2 / 1 / 1
Reg(int, fp)	(32, 32)
Branch predictor	Bimodal, 512 entries branch history table
Branch misprediction penalty	at least 7 cycles
Memory hierarchy	
L1 Dcache (private)	16KB, 4-way, 32B line, 1 cycle
L1 Icache (private)	32KB, 4-way, 32B line, 1 cycle
L2 cache (shared)	256KB per slice, 8-way, 64B line, 15 cycles, 16MB total
Main memory	105 cycles first word + 5 cycles per word 8 memory controllers 2 memory banks per memory core-bus at 5:1 frequency ratio
NoC	
Network packets	Flit size: 72 bits Data packet: 9 flits, Meta packet: 1 flit
NoC interconnect	4 virtual channels, 2 cycles router Buffer: 5x12 flits, Wire delay: 1 cycle per hop
Prefetching	
Prefetching mode	Linear & Chained
MAP	16 entries
PTB	8 entries

substrate are faithfully modeled.¹ For the processor microarchitecture, we faithfully model the MIPS32 24KE [MIPS Technologies, Inc. 2008]. The simulator models a 64-node MESI-based directory protocol with a detailed model of both stable and transient states and queuing of requests. All memory transactions are modeled using an event-driven framework accounting for latency, bandwidth constraints, bank queuing, and other contentions. Miss status holding registers and nonblocking memory controllers are modeled. Memory is address interleaved, which means that every memory controller serves the addresses mapped to one-eighth of the CMPs and uses a separate router to connect to the cores. Synchronization instructions (load-link and store-conditional) and tree barriers are also supported. The multimode prefetching engine on each processor is configured to have a 16-entry MAP and an 8-entry PTB. The initial prefetch distance and degree are set to be 8 and 2, respectively. The details of the architectural parameters used for evaluation are shown in Table V.

PopNet [Li 2007] is used to model the packet-switched network. The network is a 10×8 mesh which connects the core tiles in the middle and the memory controller tiles at two sides. We use McPAT [Li et al. 2009] to evaluate the power consumption of the processor cores in the case of 65nm complementary metal-oxide-semiconductor (CMOS) process. The DSENT [Sun et al. 2012] tool is exploited to calculate the power consumption of the networks-on-chip (NoC).

Our evaluation is performed with a suite of parallel applications from SPLASH-2 [Woo et al. 1995], PARSEC [Bienia et al. 2008], and other multithreaded workloads. These applications are compiled with “-O3” optimization by a cross-compiler to generate MIPS32 binaries. The limitation of the cross-compiler prevents us from running certain applications. Table VI lists the applications used, along with the input parameters. All

¹Instruction interpretation and Linux system call emulation are partially borrowed from Burger and Austin [1997].

Table VI. Benchmarks

Benchmark	Application	Input parameters
PARSEC	dedup	184MB data
	vips	1 image, $2,662 \times 5,500$ pixels
	ferret	256 queries, 34,973 images
	freqmine	Kosarak_250k.dat 220
	streamcluster	16,384 points per block, 1 block
	canneal	400,000 elements
	blackscholes	65,536 options
	swaptions	64 swaptions, 20,000 simulations
	fluidanimate	990,000 transactions
SPLASH-2	barnes	16K particles
	cholesky	tk15.O
	fft	64K points
	fmm	16K particles
	lu	512×512 matrix, 16×16 blocks
	ocean	258×258 ocean
	radiosity	room, -ae 5000.0 -en 0.050 -bf 0.10
	radix	1M integers radix 1024
	raytrace	car
	water-spatial	512 molecules
Scientific	em3d	3,000,000 nodes, arity-128
	jacobi	512 nodes, 10 iters
	mp3d	40,000 molecules
	shallow	512×512 , 20 steps
	tspo	18×18

results are collected by running a portion of the benchmarks for 3 billion instructions. The first 1 billion instructions are used to warm up the system and the statistics are collected for the next 2 billion instructions. We use the number of cache replacements to define the statistical interval; it is a configurable parameter and is set as 1024 by default.

We compare our method with the feedback directed prefetching (FDP) [Srinath et al. 2007] technique. The FDP mechanism estimates prefetcher accuracy, prefetcher timeliness, and prefetcher-caused cache pollution to control the aggressiveness of the prefetcher dynamically and mitigates prefetcher-caused cache pollution by adjusting insertion policy of prefetched blocks. FDP tracks cache pollution caused by the prefetcher. If the degree of prefetcher-caused pollution is high, then FDP inserts all prefetches into the LRU position of the LRU stack. FDP inserts all prefetches into the LRU position when prefetcher-caused pollution is high, which means that some accurately prefetched blocks getting inserted into the LRU position. Inserting an accurate prefetch into the LRU position may lead to the cache block getting evicted from the cache before a demand request that needs it arrives, resulting in one additional miss for that block. The TAP mechanism uses machine learning-directed runtime adjustments to relieve the attack interference between cores, which improves the prefetching performance of multithreaded applications. As an application shows different characteristics in different phases of execution, we use the TCD mechanism to classify threads at runtime and tune the aggressiveness of prefetchers according to the classification. Furthermore, in our TAP mechanism, the estimated prefetching accuracy is used to determine the insert position of prefetched data.

We also implement and evaluate hierarchical prefetcher aggressiveness control (HPAC) mechanism [Ebrahimi et al. 2009b] to combine with our mechanism. HPAC

Fig. 9. Distribution of five types of threads.

mechanism consists of a hierarchy of prefetcher aggressiveness control structures that combines local and global interference feedback to maximize the benefits of prefetching on each core while optimizing overall system performance. In HPAC, Bandwidth Consumed by Core i (BWC_i) and Bandwidth Needed by Cores Other than Core i ($BWNO_i$) are used to keep track of prefetcher-caused intercore interference in the shared memory system, whereas in our mechanism, we use the attacking filter and thread classifying directed adjustment to reduce prefetching-caused interthread invalidations.

To evaluate the prefetching influence on system performance, we collect the performance results (execution time, energy consumption, memory bandwidth consumption, etc.) and normalize them to the results with no prefetching. The geometric mean of the results is calculated to find out the system performance speedup ratio on average.

5. EXPERIMENTAL RESULTS

5.1. Thread Classification

During the runtime of each application, the characteristics of memory access behavior change with time. We use thread classifying policy (as shown in Table I) to classify each thread dynamically at runtime. Adjusting prefetching engines according to the classifying result, the TCD approach changes the memory access behavior of each thread and has an impact on the distribution of thread classifying.

Figure 9 shows the distribution of five types of threads, using TCD optimization results (right side) compared to the normal results (left side) without TCD. Since the thread classification is changing during the runtime of threads, it is sampled every statistical interval, and the columns in this figure represent the proportion of each type of threads that appeared in the statistical intervals. For **MPLA** threads, TCD reduces the degree of the prefetcher in order to improve prefetch accuracy, and for **MPHAL** threads, the prefetching engine increases prefetch distance to improve the timeliness. As a result, we can see from the figure that TCD optimization increases the proportion of **MPHA** threads, especially in the *canneal*, *ferret*, and *ocean* applications. In addition, TCD stops the prefetching engine of **MMLP** threads, and the proportion of **MMLP** threads is decreased, such as *em3d* and *shallow*. TCD optimization improves the prefetch accuracy and timeliness and reduces cache pollution and competition for LLC. Thus the cache miss rate (indicated by the metric *MPKI*) is reduced, and the proportion of **LM** threads is increased. Furthermore, the details of memory requests will be discussed in Section 5.9.

Fig. 10. Comparison of normalized miss handling waiting time.

Fig. 11. Comparison of normalized synchronization waiting time.

5.2. Reducing Prefetching-Caused Invalidation

As discussed in Section 3, we choose *tMH* (total time for miss handling) as one of the metrics for thread classifying. *tMH* accumulates the total time of handling cache miss events and reflects the intensity of competition for shared resources.

The total miss handling time for each application is shown in Figure 10. All results are normalized to the result of an ideal situation which eliminates shared resource contention. As can be seen from the figure, by using the APF, prefetching-caused interthread invalidations are avoided, and the total time waiting for miss handling has been shortened in many applications, especially in *barnes*, *fft*, *lu*, and *raytrace*. On average, the proposed APF reduces the miss handling time by 7% compared to the MDP.

5.3. Limiter Thread Acceleration

When multiple threads concurrently execute in a CMP system, they are synchronized through locks or barriers. We propose the LTA mechanism to estimate the threads likely to be on the critical path of execution, and design the prefetching adjustment strategy to prioritize the critical threads in order to reduce the threads' synchronization waiting time and accelerate their execution.

Figure 11 shows the average synchronization waiting time of all threads for each application, all the results are normalized to the results with no prefetching. Since the LTA mechanism estimates and accelerates the critical threads, the average synchronization waiting time is reduced by 6% compared to the MDP.

5.4. System Performance

Figures 12 and 13 illustrate the influence of prefetching on multicore system performance by showing the execution time normalized to the result with no prefetching. We evaluate the system performance of the proposed techniques (described in Section 3, namely TCD, APF, and LTA) separately applied on the baseline MDP, as well as the

Fig. 12. Normalized execution time of MDP, TCD, APF, LTA, and TAP.

Fig. 13. Normalized execution time of MDP, TCD, TAP, and FDP.

performance of TAP, which simultaneously uses the three proposed techniques to get maximum improvement.

Figure 12 compares the normalized execution time among TCD, APF, and LTA. Taking threads' runtime behavior into consideration, TCD reduces the conflicts among threads and improves the prefetch accuracy, such as *canneal*, *dedup*, and *ferret*. As we can see, the TCD method increases the proportion of **MPHA** threads and decreases the percentage of **MPLA** threads for the *canneal*, *dedup*, and *ferret* applications. With high accuracy of prefetching, system performance increases. APF, which gains benefit from the attacking prefetching filter and cache replacement mechanism, reduces the negative impact of prefetching-caused invalidations. The LTA mechanism accelerates the limiter threads and reduces the threads' waiting time for synchronization. On average, TCD, APF, and LTA improve performance by 13%, 10%, and 9%, respectively.

Figure 13 compares the normalized execution time between MDP, TAP, and FDP. The results show that the use of the multimode prefetcher on the shared memory multicore system can improve system performance for some applications while it decreases system performance for others. The average speedup using the multimode prefetcher is 5%. TAP gathers the benefits from the proposed techniques (TCD, APF, and LTA) and improves performance by 18%, while FDP improves by 9% compared to the no-prefetching system.

5.5. Energy Consumption

The introduction of a prefetch engine increases the memory bandwidth consumption as well as prefetching-caused cache-related operations, thus inevitably increasing the energy consumption of the whole system. Figure 14 shows the system energy consumption; the results are normalized to that with no prefetching. On average, MDP, TAP, and FDP increase the energy consumption by 6%, 4%, and 5%, respectively. The energy consumption overhead is acceptable, since it brings notable improvement of system

Fig. 14. Comparison of normalized energy consumption.

Fig. 15. Comparison of normalized execution time with HPAC mechanism.

performance. When using the TAP approach, the number of useless prefetching requests is minimized, and this reduces unnecessary energy consumption.

5.6. Effect of TAP with HPAC Mechanism

As mentioned in Section 4, we have implemented and evaluated our mechanism with the HPAC mechanism. Figure 15 shows the performance of TAP and FDP when employed in a system with HPAC mechanism. FDP mechanism lacks the way to control prefetching-caused interference between cores, when combined with HPAC mechanism, the system performance is improved by 10% (“FDP” vs. “FDP with HPAC”), as the HPAC mechanism takes advantage of hierarchical prefetcher aggressiveness control structures to maximize the benefits of prefetching on each core. The TAP mechanism mainly focuses on interthread interference, while the HPAC mechanism takes memory bandwidth consumption into consideration; thus the HPAC mechanism increases the performance of a system that uses TAP by 4% (“TAP” vs. “TAP with HPAC”). Although there is no global control structure in TAP, the attacking prefetch filter is proposed to avoid prefetching-caused interthread invalidations. As a result, HPAC using TAP locally improves system performance by 3% over HPAC using FDP locally (“FDP with HPAC” vs. “TAP with HPAC”).

5.7. Impact on Memory Bandwidth Consumption

Prefetching can adversely affect the memory bandwidth consumption when prefetches are not used or when they cause cache pollution. Figure 16 shows the memory bandwidth impact of prefetching. The results are normalized to the system with no prefetching. Since the proposed TAP mechanism reduces the number of demand cache misses caused by interthread invalidations, it consumes 5% less memory bandwidth than FDP. Compared to a system only using TAP, a system using TAP with HPAC mechanism can decrease the memory bandwidth consumption by 1%, as HPAC has a

Fig. 16. Memory bandwidth consumption (normalized to no-prefetching).

Fig. 17. System performance of cache partitioning mechanism.

global control mechanism which can allow or override the decision of a local prefetcher to optimize overall system performance.

5.8. Effect of Cache Partitioning Mechanism

The cache partitioning mechanism [Chang and Sohi 2007; Qureshi and Patt 2006] is an effective method to improve shared cache performance. For multiprogrammed workloads, the cache conflicts are caused due to the limited cache capacity. Since different programs have different utilities of cache resources, the cache partitioning mechanism can intelligently decide the number of cache ways allocated to each core and adjust the partitioning options according to runtime information to improve system performance. However, for multithreaded workloads, multiple threads share data among themselves, and thus they cannot be independently treated.

We implement the cache partitioning mechanism [Qureshi and Patt 2006] in our multithreaded system and Figure 17 shows the system performance of the cache partitioning mechanism. As can be seen, there is no significant effect of this mechanism for multithreaded workloads. This is one of the reasons why we propose the TAP mechanism for shared-memory multicore systems running multithreaded workloads.

5.9. Distribution of Memory Requests

For a system with prefetch engines, its memory requests usually consist of two parts, demand requests and prefetch requests. A part of the demand requests is caused by the invalidation due to prefetches (*D-prefetch*), and the others are normally issued during the runtime (*D-normal*). Furthermore, it is also clear to divide the prefetch requests into three categories, accurate prefetch requests that are on time (*P-ontime*), accurate but late prefetch requests (*P-late*), and inaccurate prefetch requests (*P-miss*). In order to get further insight into the effectiveness of TAP mechanism, we analyze all kinds of memory requests and draw a breakdown figure, as shown in Figure 18. The FDP

Fig. 18. Distribution of memory requests.

Table VII. Sensitivity to L2 Cache Size and Memory Latency

L2 Cache Size (memory latency = 105 cycles)					
8MB		16 MB		32MB	
Δ IPC	Δ MBC	Δ IPC	Δ MBC	Δ IPC	Δ MBC
18.9%	-5.1%	12.1%	-4.0%	10.3%	-3.0%

Memory latency (L2 cache size = 16MB)					
70 cycles		105 cycles		140 cycles	
Δ IPC	Δ MBC	Δ IPC	Δ MBC	Δ IPC	Δ MBC
14.2%	-3.4%	12.1%	-4.0%	9.7%	-4.5%

results are on the left bars, while the TAP results are on the right bars; all of the results are normalized to that with no prefetching.

TAP utilizes the attacking prefetch filter mechanism to avoid prefetching-caused interthread invalidations, compared to FDP, the percentage of *D-prefetch* is reduced. On the other hand, taking the advantage of prefetching-aware thread classification mechanism, which uses a k-means clustering algorithm to analyze threads' characteristics, TAP can adaptively adjust the prefetch engines for multithreaded workloads at run-time, thus it enhances prefetching accuracy (*P-ontime*) and reduces useless prefetches.

5.10. Sensitivity Study

We evaluate the sensitivity of TAP to different L2 cache sizes and memory latencies. Table VII shows the change in average IPC (instructions per cycle) and MBC (memory bandwidth consumption) provided by TAP over FDP for each configuration. Considering the interthread confliction and interaction, the TAP mechanism cuts down the invalidations caused by prefetching requests and reduces the execution time by accelerating the limiter threads. Thus, TAP provides better performance and consumes less memory bandwidth than FDP for all evaluated cache sizes. On the other hand, as memory latency increases, the MBC improvement of TAP also increases. This is because the bandwidth-saving effect of TAP becomes more important when memory becomes a larger performance bottleneck.

Furthermore, we have a sensitivity study to the number of threads. Table VIII shows the change in average execution time when the number of threads changes. The results show that with the number of threads grows, the performance improvement of TAP over FDP increases. In a system with smaller number of threads, the contention for shared resources is mitigated, thus the proposed TAP mechanism lacks the potential for improving performance. While in a system with larger number of threads, there

Table VIII. Sensitivity to the Number of Threads

Cores/ Threads	Normalized execution time (Geomean)		Improvement (FDP / TAP -1)×100%
	TAP	FDP	
4	0.75	0.76	1.3%
8	0.77	0.80	3.9%
16	0.79	0.85	7.6%
32	0.80	0.87	8.7%
64	0.82	0.91	11.0%

Table IX. Storage Overhead of Thread-aware Adaptive Prefetching

		Closed form for N nodes	$N = 64$
<i>TCD</i>	Counters	16 bits / counter $\times$ 5 counters / node $\times N$	5,120 bits
	<i>pref-bit</i> in L1 cache	512 bits / node $\times N$	32,768 bits
	<i>PHT</i> in L1	28 bits / entry $\times$ 16 entries / node $\times N$	28,672 bits
	MAP	51 bits / entry $\times$ 16 entries / node $\times N$	52,224 bits
	PTB	81 bits / entry $\times$ 8 entries / node $\times N$	41,472 bits
<i>APF</i>	<i>pref-request</i> in MSHR of L2	1 bit / entry $\times$ 32 entries / node $\times N$	2,048 bits
<i>LTA</i>	<i>LimiterThreadBit</i>	1 bit / node $\times N$	64 bits
	<i>FasterThreadBit</i>	1 bit / node $\times N$	64 bits
	<i>TotalWaitTime</i>	32 bits / node $\times N$	2,048 bits
	<i>ThreadWaitTime</i>	16 bits / node $\times N$	1,024 bits
Total		2586 bits / node $\times N$	20.2 KB

are more interthread conflicts resulting in performance degradation, thus the TAP mechanism can provide more performance improvement over FDP mechanism.

5.11. Hardware Cost

Table IX shows the storage overhead of the proposed TAP mechanism in terms of the required state. The additional storage is 20.2KB for a 64-node system (one processor per node). This storage overhead of our mechanism is less than 0.13% of the data-store size of the baseline 16MB L2 cache. Meanwhile, combinational logic is required for the update of counters, the update of the flag bits in cache and MSHR, the update of the bit-vectors for limiter thread observation, calculation of classifying metrics at the end of each sampling interval, termination of the cache replacement of the attacking prefetch, insertion of prefetched blocks into appropriate locations in the LRU stack, and the tuning of prefetching aggressiveness. The complexity of the required combinational logic is not significant, and none of the required logic is on the critical path of the processor.

6. RELATED WORK

The key idea of hardware data prefetchers is to use a special hardware unit to detect data address sequences and find patterns. Based on these patterns, future data addresses can be predicted. Many hardware data prefetching techniques for sequential address pattern have been proposed, such as sequential prefetching [Jouppi 1990], nonunit stride address sequence prefetching [Palacharla and Kessler 1994], and multistride prefetching [Fu et al. 1992; Iacobovici et al. 2004]. Techniques for prefetching chained address patterns have also been proposed, such as dependence-based prefetching [Roth et al. 1998], the CDP method [Cooksey et al. 2002], and so on. Moreover, prior works [Joseph and Grunwald 1997; Wenisch et al. 2009] have studied the prefetchers for irregular patterns. Irregular Stream Buffer (ISB) [Jain and Lin 2013] uses the idea of introducing an extra level of indirection to create a new structural address space in which correlated physical addresses are assigned consecutive structural addresses and produces benefits in terms of coverage, accuracy, and memory traffic overhead. Chen et al. [2012] proposed an algorithm-level feedback-controlled adaptive data

prefetcher, which was based on a specifically designed hardware structure, to provide an algorithm-level adaptation to applications' access patterns. The above prefetching techniques can significantly improve performance for different types of applications on single-core processors. In multicore systems, however, the introduction of prefetching will increase conflicts and competition for shared resources, and prefetching in a multicore system without efficient management can lead to performance deterioration.

To relieve the impact of prefetching engines on shared resource contention, Dahlgren et al. [1993] proposed adaptive prefetching for shared-memory multiprocessors by dynamically configuring aggressiveness for each core according to prefetching accuracy. Jerger et al. [2006] observed prefetching-caused shared cache invalidations for multithreaded applications, and they presented a taxonomy that classifies the effects of multiprocessor prefetches as well as a characterization of the effects of different prefetching schemes. Srinath et al. [2007] considered the influence of a prefetching engine on memory bandwidth consumption and proposed an approach based on feedback information to direct the adjustment of prefetching dynamically, achieving the goal of reducing prefetching-caused bandwidth consumption. Huang et al. [2012] analyzed the effect of increasing prefetch distance on shared cache pollution and reduced the shared cache pollution by controlling prefetch distance. An adaptive prefetching scheme [Jiménez et al. 2012] in POWER7 is proposed to dynamically modify the prefetch settings according to the requirements of single-threaded and multiprogrammed workloads in order to improve performance and power consumption. In our article, we focus on how an uncached prefetch will invoke a costly interthread invalidation and propose the APF mechanism to address this problem.

Ebrahimi et al. [2009b] proposed an HPAC mechanism as a prefetcher throttling solution to improve prefetching performance in CMP systems. By gathering global feedback information about the effect of each core's prefetcher, the HPAC mechanism can dynamically adjust the aggressiveness of each prefetcher in two ways: local and global. The global mechanism can override the local decision by taking into account effects and interactions of different cores' prefetchers when adjusting each one's aggressiveness. HPAC aimed to reduce the cache pollution caused by evictions of prefetches. While in our TAP mechanism, we focus on the invalidation of shared cache lines and use an attacking prefetch filter to reduce interthread influences. Furthermore, HPAC tracked memory bandwidth consumption and memory access delay time of each core to make global decisions, whereas in TAP, a limiter thread acceleration mechanism is used to tune up limiter threads' prefetching aggressiveness to achieve potential improvement and turn down faster threads' prefetching aggressiveness to avoid unnecessary memory bandwidth consumption. This mechanism is more suitable for multithreaded applications to optimize overall system performance.

By exploiting the idea of "prefetch-aware," Ebrahimi et al. [2011a] and Lee et al. [2008] took the existence of prefetching and its accuracy into consideration when dealing with data requests and improved the effectiveness of memory scheduling. The "thread-aware" idea was also explored by Lee et al. [2010]: They proposed GPGPU-specific software and hardware prefetching mechanisms and developed a prefetch throttling mechanism that dynamically adjusts the level of prefetching to avoid performance degradation. Somogyi et al. [2006] proposed Spatial Memory Streaming (SMS) to predict code-correlated spatial access patterns and stream predicted blocks to the primary cache ahead of demand misses. In their subsequent studies, they observed that the order of spatial accesses repeated both within and across regions and proposed Spatio-Temporal Memory Streaming (STeMS) [Somogyi et al. 2009] to exploit the synergy between spatial and temporal streaming. The combination of temporal and spatial predictions into a single total predicted sequence prevented the predictors from interfering with each other and enhanced lookahead for spatial accesses.

PACMan [Wu et al. 2011] aimed to mitigate the degree of prefetch-induced LLC pollution. Based on dynamic prediction of the rereference behavior of prefetch requests, PACMan treated demand and prefetch requests differently and employed a prefetch-aware cache insertion policy. B-Fetch [Kadjo et al. 2014] was a data prefetcher driven by branch prediction and effective address value speculation. Based on a speculative control flow path from a branch predictor, B-Fetch computes the effective address of the load instructions along that path based on a history of past register transformations and uses a per-load filtering mechanism to reduce cache pollution. Sandbox Prefetching [Pugsley et al. 2014] combined the ideas of global pattern confirmation and immediate prefetching action to achieve high performance. It tracked candidate prefetch patterns at runtime by adding the prefetch addresses to a Bloom filter rather than actually fetching the data into the cache. A candidate prefetcher may be globally activated to immediately perform a prefetch action when the accuracy of evaluated prefetchers exceeds a threshold. Seshadri et al. observed that over 95% of useful prefetches are not reused after the first demand hit and proposed the Informed Caching policies for Prefetched blocks (ICP) mechanism [Seshadri et al. 2015] based on this observation. The ICP mechanism consisted of two components: ICP-D demotes a prefetched block to the lowest priority on a demand hit, and ICP-AP predicts the accuracy of prefetch requests and only predicted-accurate prefetches are inserted into the cache with a high priority. In our article, we assume that the runtime behavior of each individual thread will not change dramatically in a short time. Thus our mechanism *predicts* the prefetch accuracy as the value counted by the statistical model in the most recent interval.

We focus on how to reduce prefetching-caused resource contention in multithreaded applications. The work in this article differs from the above in that we discovered a new problem, that is, that prefetching can cause interthread invalidations in multithreaded applications when LLC is shared under MESI-based cache coherence protocols. We provide the thread-aware adaptive data prefetcher as the solution to this problem. In addition, instead of directing prefetchers based on the characteristics of each core, our approach directs each prefetcher based on the classification of running threads. The proposed method can effectively adjust prefetchers dynamically according to the real-time status of each thread, and estimate and accelerate the critical threads according to the properties of multithread applications.

7. CONCLUSION

This article presents a thread-aware adaptive prefetching engine for multicore systems. Compared to existing multicore prefetchers, our prefetching engine can reduce interthread shared resource contention on the shared LLC and off-chip memory. Our approach combines three ideas: (1) It takes the threads' runtime memory requests and prefetching efficiency into consideration and makes decisions to relieve the interthread contention on the shared LLC and memory system, (2) it filters attacking prefetching requests if they would invalidate other threads' shared data while avoiding the break of prefetching streams, and (3) it uses a limiter thread acceleration mechanism to estimate and accelerate the threads that are on the critical paths of execution. Our approach reduces shared resource contention, brings energy efficiency, and improves system performance noticeably.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous referees for their detailed comments and valuable suggestions, which have helped us to improve the quality of the article. The authors would also thank Dmitry V. Ponomarev and Mei Yang for their comments on early versions of this article.

REFERENCES

- Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC benchmark suite: Characterization and architectural implications. In *Proc. Int'l Conf. on Parallel Arch. and Compilation Techniques*. ACM, 72–81. DOI: <http://dx.doi.org/10.1145/1454115.1454128>
- D. Burger and T. M. Austin. 1997. The SimpleScalar tool set, version 2.0. *ACM SIGARCH Comput. Arch. News* 25, 3 (June 1997), 13–25. DOI: <http://dx.doi.org/10.1145/268806.268810>
- Jichuan Chang and Gurindar S. Sohi. 2007. Cooperative cache partitioning for chip multiprocessors. In *Proc. Int'l Conf. on Supercomputing*. ACM, 402–412. DOI: <http://dx.doi.org/10.1145/1274971.1275005>
- Yong Chen, Huaiyu Zhu, Hui Jin, and Xian-He Sun. 2012. Algorithm-level feedback-controlled adaptive data prefetcher: Accelerating data access for high-performance processors. *Parallel Comput.* 38, 10–11 (October/November 2012), 533–551. DOI: <http://dx.doi.org/10.1016/j.parco.2012.06.002>
- Robert Cooksey, Stephan Jourdan, and Dirk Grunwald. 2002. A stateless, content-directed data prefetching mechanism. In *Proc. Int'l Conf. on Arch. Support for Prog. Lang. and Operating Systems*. ACM, 279–290. DOI: <http://dx.doi.org/10.1145/605397.605427>
- Fredrik Dahlgren, Michel Dubois, and Per Stenstrom. 1993. Fixed and adaptive sequential prefetching in shared memory multiprocessors. In *Proc. Int'l Symp. on Parallel Processing*. IEEE, 56–63. DOI: <http://dx.doi.org/10.1109/ICPP.1993.92>
- Eiman Ebrahimi, Chang Joo Lee, Onur Mutlu, and Yale N. Patt. 2011a. Prefetch-aware shared resource management for multi-core systems. In *Proc. Int'l Symp. on Comp. Arch.* ACM, 141–152. DOI: <http://dx.doi.org/10.1145/2000064.2000081>
- Eiman Ebrahimi, Rustam Miftakhutdinov, Chris Fallin, Chang Joo Lee, Jos A. Joao, Onur Mutlu, and Yale N. Patt. 2011b. Parallel application memory scheduling. In *Proc. Int'l Symp. on Microarch.* ACM, 362–373. DOI: <http://dx.doi.org/10.1145/2155620.2155663>
- Eiman Ebrahimi, Onur Mutlu, Chang Joo Lee, and Yale N. Patt. 2009b. Coordinated control of multiple prefetchers in multi-core systems. In *Proc. Int'l Symp. on Microarch.* IEEE, 316–326. DOI: <http://dx.doi.org/10.1145/1669112.1669154>
- Eiman Ebrahimi, Onur Mutlu, and Yale N. Patt. 2009a. Techniques for bandwidth-efficient prefetching of linked data structures in hybrid prefetching systems. In *Proc. Int'l Symp. on High Performance Comp. Arch.* IEEE, 7–17. DOI: <http://dx.doi.org/10.1109/HPCA.2009.4798232>
- John W. C. Fu, Janak H. Patel, and Bob L. Janssens. 1992. Stride directed prefetching in scalar processors. In *Proc. Int'l Symp. on Microarch.* IEEE, 102–110. DOI: <http://dx.doi.org/10.1109/MICRO.1992.697004>
- Ilya Ganusov and Martin Burtcher. 2005. On the importance of optimizing the configuration of stream prefetchers. In *Proc. Workshop on Memory System Performance (MSP'05)*. ACM, New York, NY, 54–61. DOI: <http://dx.doi.org/10.1145/1111583.1111591>
- Yan Huang, Zhi-min Gu, Jie Tang, Min Cai, Jianxun Zhang, and Ninghan Zheng. 2012. Reducing cache pollution of threaded prefetching by controlling prefetch distance. In *Proc. Int'l Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW)*. IEEE, 1812–1819. DOI: <http://dx.doi.org/10.1109/IPDPSW.2012.224>
- Sorin Iacobovici, Lawrence Spracklen, Sudarshan Kadambi, Yuan Chou, and Santosh G. Abraham. 2004. Effective stream-based and execution-based data prefetching. In *Proc. Int'l Conf. on Supercomputing*. ACM, 1–11. DOI: <http://dx.doi.org/10.1145/1006209.1006211>
- Akanksha Jain and Calvin Lin. 2013. Linearizing irregular memory accesses for improved correlated prefetching. In *Proc. Int'l Symp. on Microarch.* ACM, 247–259. DOI: <http://dx.doi.org/10.1145/2540708.2540730>
- Aamer Jaleel, William Hasenplaugh, Moinuddin Qureshi, Julien Sebot, Simon Steely, Jr., and Joel Emer. 2008. Adaptive insertion policies for managing shared caches. In *Proc. Int'l Conf. on Parallel Arch. and Compilation Techniques*. ACM, 208–219. DOI: <http://dx.doi.org/10.1145/1454115.1454145>
- Aamer Jaleel, Hashem H. Najaf-abadi, Samantika Subramaniam, Simon C. Steely, and Joel Emer. 2012. CRUISE: Cache replacement and utility-aware scheduling. In *Proc. Int'l Conf. on Arch. Support for Prog. Lang. and Operating Systems*. ACM, 249–260. DOI: <http://dx.doi.org/10.1145/2150976.2151003>
- Natalie D. Enright Jerger, Eric L. Hill, and Mikko H. Lipasti. 2006. Friendly fire: Understanding the effects of multiprocessor prefetches. In *Proc. Int'l Symp. on Performance Analysis of Systems and Software*. IEEE, 177–188. DOI: <http://dx.doi.org/10.1109/ISPASS.2006.1620802>
- Victor Jiménez, Roberto Gioiosa, Francisco J. Cazorla, Alper Buyuktosunoglu, Pradip Bose, and Francis P. O'Connell. 2012. Making data prefetch smarter: Adaptive prefetching on POWER7. In *Proc. Int'l Conf. on Parallel Arch. and Compilation Techniques*. ACM, 137–146. DOI: <http://dx.doi.org/10.1145/2370816.2370837>

- Doug Joseph and Dirk Grunwald. 1997. Prefetching using markov predictors. In *Proc. Int'l Symp. on Comp. Arch.* ACM, 252–263. DOI: <http://dx.doi.org/10.1145/264107.264207>
- Norman P. Jouppi. 1990. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proc. Int'l Symp. on Comp. Arch.* IEEE, 364–373. DOI: <http://dx.doi.org/10.1109/ISCA.1990.134547>
- David Kadjo, Jinchun Kim, Prabal Sharma, Reena Panda, Paul Gratz, and Daniel Jimenez. 2014. B-Fetch: Branch prediction directed prefetching for chip-multiprocessors. In *Proc. Int'l Symp. on Microarch.* IEEE Computer Society, 623–634. DOI: <http://dx.doi.org/10.1109/MICRO.2014.29>
- Yoongu Kim, Dongsu Han, Onur Mutlu, and Mor Harchol-Balter. 2010. ATLAS: A scalable and high-performance scheduling algorithm for multiple memory controllers. In *Proc. Int'l Symp. on High Performance Comp. Arch.* IEEE, 1–12. DOI: <http://dx.doi.org/10.1109/HPCA.2010.5416658>
- Yoongu Kim, Michael Papamichael, Onur Mutlu, and Mor Harchol-Balter. 2011. Thread cluster memory scheduling. *IEEE Micro* 31, 1 (Jan./Feb. 2011), 78–89. DOI: <http://dx.doi.org/10.1109/MM.2011.15>
- Chang Joo Lee, Onur Mutlu, Veynu Narasiman, and Yale N. Patt. 2008. Prefetch-aware DRAM controllers. In *Proc. Int'l Symp. on Microarch.* IEEE, 200–209. DOI: <http://dx.doi.org/10.1109/MICRO.2008.4771791>
- Jaekyu Lee, Nagesh B. Lakshminarayana, Hyesoon Kim, and Richard Vuduc. 2010. Many-thread aware prefetching mechanisms for gpgpu applications. In *Proc. Int'l Symp. on Microarch.* IEEE, 213–224. DOI: <http://dx.doi.org/10.1109/MICRO.2010.44>
- Shang Li. 2007. PoPNet simulator. Retrieved from <http://www.princeton.edu/~peh/orion.html>.
- Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. 2009. McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proc. Int'l Symp. on Microarch.* ACM, 469–480. DOI: <http://dx.doi.org/10.1145/1669112.1669172>
- James MacQueen. 1967. Some methods for classification and analysis of multivariate observations. In *Proc. Fifth Berkeley Symp. on Mathematical Statistics and Probability*, Vol. 1. University of California Press, Berkeley, CA, 281–297.
- MIPS Technologies, Inc. 2008. MIPS32[®] 24KE[™] Processor Core Family Software User's Manual. (Dec. 2008). Document Number: MD00468.
- Onur Mutlu and Thomas Moscibroda. 2008. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared DRAM systems. In *Proc. Int'l Symp. on Comp. Arch.* IEEE, 63–74. DOI: <http://dx.doi.org/10.1109/ISCA.2008.7>
- Kyle J. Nesbit, Nidhi Aggarwal, James Laudon, and James E. Smith. 2006. Fair queuing memory systems. In *Proc. Int'l Symp. on Microarch.* IEEE, 208–222. DOI: <http://dx.doi.org/10.1109/MICRO.2006.24>
- Subbarao Palacharla and R. E. Kessler. 1994. Evaluating stream buffers as a secondary cache replacement. In *Proc. Int'l Symp. on Comp. Arch.* IEEE, 24–33. DOI: <http://dx.doi.org/10.1145/191995.192014>
- Seth H. Pugsley, Zeshan Chishti, Chris Wilkerson, Peng-fei Chuang, Robert L. Scott, Aamer Jaleel, Shih-Lien Lu, Kingsum Chow, and Rajeev Balasubramonian. 2014. Sandbox prefetching: Safe run-time evaluation of aggressive prefetchers. In *Proc. Int'l Symp. on High Performance Comp. Arch.* IEEE, 626–637. DOI: <http://dx.doi.org/10.1109/HPCA.2014.6835971>
- Moinuddin K. Qureshi and Yale N. Patt. 2006. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proc. Int'l Symp. on Microarch.* IEEE, 423–432. DOI: <http://dx.doi.org/10.1109/MICRO.2006.49>
- Amir Roth, Andreas Moshovos, and Gurindar S. Sohi. 1998. Dependence based prefetching for linked data structures. In *Proc. Int'l Conf. on Arch. Support for Prog. Lang. and Operating Systems.* ACM, 115–126. DOI: <http://dx.doi.org/10.1145/291069.291034>
- Vivek Seshadri, Samihan Yedkar, Hongyi Xin, Onur Mutlu, Phillip B. Gibbons, Michael A. Kozuch, and Todd C. Mowry. 2015. Mitigating prefetcher-caused pollution using informed caching policies for prefetched blocks. *ACM Trans. Archit. Code Optim.* 11, 4 (Jan. 2015), 51:1–51:22. DOI: <http://dx.doi.org/10.1145/2677956>
- Stephen Somogyi, Thomas F. Wenisch, Anastassia Ailamaki, and Babak Falsafi. 2009. Spatio-temporal memory streaming. In *Proc. Int'l Symp. on Comp. Arch.* ACM, 69–80. DOI: <http://dx.doi.org/10.1145/1555754.1555766>
- Stephen Somogyi, Thomas F. Wenisch, Anastassia Ailamaki, Babak Falsafi, and Andreas Moshovos. 2006. Spatial memory streaming. In *Proc. Int'l Symp. on Comp. Arch.* ACM, 252–263. DOI: <http://dx.doi.org/10.1109/ISCA.2006.38>
- Santhosh Srinath, Onur Mutlu, Hyesoon Kim, and Yale N. Patt. 2007. Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers. In *Proc. Int'l Symp. on High Performance Comp. Arch.* IEEE, 63–74. DOI: <http://dx.doi.org/10.1109/HPCA.2007.346185>

- Chen Sun, C.-H. O. Chen, George Kurian, Lan Wei, Jason Miller, Anant Agarwal, Li-Shiuan Peh, and Vladimir Stojanovic. 2012. DSENT: A tool connecting emerging photonics with electronics for opto-electronic networks-on-chip modeling. In *Proc. Int'l Symp. on Networks on Chip*. IEEE, Lyngby, Denmark, 201–210. DOI: <http://dx.doi.org/10.1109/NOCS.2012.31>
- Thomas F. Wenisch, Michael Ferdman, Anastasia Ailamaki, Babak Falsafi, and Andreas Moshovos. 2009. Practical off-chip meta-data for temporal memory streaming. In *Proc. Int'l Symp. on High Performance Comp. Arch.* IEEE, 79–90. DOI: <http://dx.doi.org/10.1109/HPCA.2009.4798239>
- Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. 1995. The SPLASH-2 programs: Characterization and methodological considerations. In *Proc. Int'l Symp. on Comp. Arch.* ACM, 24–36. DOI: <http://dx.doi.org/10.1145/223982.223990>
- Carole-Jean Wu, Aamer Jaleel, Margaret Martonosi, Simon C. Steely Jr., and Joel Emer. 2011. PACMan: Prefetch-aware cache management for high performance caching. In *Proc. Int'l Symp. on Microarch.* ACM, 442–453. DOI: <http://dx.doi.org/10.1145/2155620.2155672>
- Jiyang Yu and Peng Liu. 2014. A thread-aware adaptive data prefetcher. In *Proc. Int'l Conf. on Computer Design.* IEEE, 278–285. DOI: <http://dx.doi.org/10.1109/ICCD.2014.6974694>
- Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. 2010a. Addressing shared resource contention in multicore processors via scheduling. In *Proc. Int'l Conf. on Arch. Support for Prog. Lang. and Operating Systems.* ACM, 129–142. DOI: <http://dx.doi.org/10.1145/1736020.1736036>
- Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. 2010b. AKULA: A toolset for experimenting and developing thread placement algorithms on multicore systems. In *Proc. Int'l Conf. on Parallel Arch. and Compilation Techniques.* ACM, 249–260. DOI: <http://dx.doi.org/10.1109/MICRO.2010.51>

Received April 2015; revised January 2016; accepted January 2016