A Thread-Aware Adaptive Data Prefetcher

Jiyang Yu and Peng Liu
Department of Information Science and Electronic Engineering, Zhejiang University, Hangzhou, 310027, China
State Key Laboratory of Mathematical Engineering and Advanced Computing, Wuxi, 214125, China
Email: zju.isee.yjy@gmail.com, liupeng@zju.edu.cn

Abstract—Most processors employ hardware data prefetching to hide memory access latencies. However the prefetching requests from different threads on a multi-core 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 multi-core systems. This paper 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 two new components: 1) a filtering mechanism that informs the hardware about which prefetching requests can cause shared data invalidation and should be discarded, and 2) a self-tuning prefetcher that uses run-time feedback to adjust each thread’s data prefetching mode and arguments. On a set of parallel benchmarks, our thread-aware data prefetching mechanisms improve the overall performance of 64-core system by 11% and reduce the energy-delay product by 13% over a multi-mode prefetch baseline system with a two level cache organization and a conventional MESI-based directory coherence protocol. We compare our approach to the feedback directed prefetching (FDP) technique and find that it provides better performance on multi-core systems, while reducing the energy delay product.

I. 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 proven useful. However, on multi-core processors, 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 multi-core 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 [1]–[3], operating system based thread scheduling [4]–[7], feedback directed prefetching aggressiveness throttling [8]–[11], and other methods managing the shared LLC and main memory. However, most previous techniques are designed for multi-programmed workloads, prefetching on multi-threaded parallel applications is confronted with new problems. Generally multiple threads on multi-core systems share data among themselves. In order to maintain consistency, if data in a shared cache is replaced by prefetching data, all its sharers should invalidate their local copy of the cache block. Besides, in real applications, some threads do the same work with different data, while other threads process the same copy of data in different ways. The memory behaviors of each thread are complicated which make the sensible arrangements for each thread different in prefetching controllers.

By analyzing the impact of prefetching on the shared resource contention in multi-threaded parallel applications, we found that prefetching causes inter-thread invalidations which leads to increased misses of other threads in private caches. To filter such prefetches we propose a filtering mechanism. In order to reduce prefetching caused contention on the shared LLC and off-chip memory, we develop a thread-aware data prefetching mechanism based on low-overhead run-time information to tune prefetching modes and aggressiveness, mitigating the resource contention in the shared memory multi-core systems.

In this paper, we make the following new contributions:

• We develop an attacking prefetch filter which can be used to reduce prefetching-caused inter-thread invalidations.

• We propose a thread classifying directed approach to adjust prefetching parameters dynamically based on run-time information for each thread to reduce the shared resource contention.

• We extensively evaluate the proposed thread-aware adaptive data prefetcher on a 64-core system using multi-threaded workloads from PARSEC [12], SPLASH-2 [13], and some scientific applications.

The rest of the paper is organized as follows: Section II describes the basic prefetching structure; Section III gives our thread-aware prefetching mechanism in detail; Section IV presents our evaluation methodology; Section V analyzes the experimental results; Section VI discusses related work; and finally Section VII concludes.

This work is supported in part by the Open Project Program of the State Key Laboratory of Mathematical and Advanced Computing Program grant 2014A08, by the National Natural Science Foundation of China grants 60873112 and 61028004, and by the National High Technology Research and Development Program of China grant 2009AA01Z109.

*The corresponding author is Peng Liu.
II. BASIC PREFETCHING STRUCTURE

Figure 1 shows the multi-core system modeled in this paper. Each node has private L1 instruction and data caches, all nodes share a physically distributed LLC with integrated directory, and communicate with each other through routers which are connected by the on-chip interconnect fabric. Caches are shared under a MESI (Modified, Exclusive, Shared, and Invalid) protocol implementation. The prefetching engine on each node includes: 1) a filtered multi-mode prefetcher which can prefetch both sequential and chained stream patterns, and 2) a Thread Classifying Directed (TCD) prefetcher tuning unit. The self-tuning prefetcher consists of a physical statistical module for recording run-time information and a hardware engine implementing thread classifying and prefetch tuning.

Fig. 1. System overview.

The basic prefetching engine consists of a memory access pool (MAP) for recording history information and a pattern table (PTB) for the maintenance of predicted patterns. It is a multi-mode prefetching engine which can detect sequential and chained data address access patterns on each core of multi-core system. The structure of MAP is shown in Figure 2. The missaddr field records the address of each cache miss. The dist 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). 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 3. The first and last fields in PTB record the first and last address of previous prefetching requests. The dist field records the stride of the sequential stream for sequential patterns or the offset of next field in the linked list for chained patterns. The type field is used to distinguish between sequential patterns and chained patterns.

During program execution, the first address of a cache miss is recorded in MAP entry indexed by the missed address. If the value returned is an address like value, then the type field is set, implying chained pattern training. Otherwise, the type field is unset, implying sequential pattern training. The process of prefetching for sequential patterns follows the steps as shown in Algorithm 1. The process of prefetching for chained patterns follows the steps as shown in Algorithm 2.

The most important distinction between our approach and CDP [14] for chained stream prefetch 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 [15]. 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 the Miss Status Holding Register (MSHR) has recorded the data request. If neither is true then a prefetching request is finally issued.

III. THREAD AWARE PREFETCHING SYSTEM

A. Reducing Prefetching-caused Inter-thread 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 demand data or prefetched data is returned from memory. This situation is more complicated under a MESI-based cache coherence protocol. Since L1 cache misses and prefetcher requests will access LLC, they can cause the replacement of a data block which is shared by other threads. The result is that the copies of data 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 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 block.

To avoid degradation of performance, one solution is to abandon all attacking prefetches. 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 paper, we propose solutions for sequential and chained stream 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 4, 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.
Algorithm 1: Steps of finding a sequential stream.

```plaintext
Algorithm 1: Steps of finding a sequential stream.
SeqPrefetch(Addr) // Sequential stream pattern: cache miss at address A
if A hit on MAP and Status == 1 then
    NewDist = A - MissAddr;
    MissAddr = A;
    Update Status = 2 and Type = 0;
else
    if NewDist == Dist then
        ready for prefetching;
        Dist = NewDist;
        break;
    else
        Dist = 0;
        MissAddr = A;
        Update Status = 1;
end

Algorithm 2: Steps of finding a chained stream.
LinPrefetch(Addr, ReturnValue) // Chained stream pattern: cache miss at address A
if ReturnValue is an address like value then
    if A hit on MAP and Status == 1 then
        NewDist = ReturnValue - MissAddr;
        MissAddr = ReturnValue;
        Update Status = 2 and Type = 1;
        if NewDist == Dist then
            ready for prefetching;
            Dist = NewDist;
        end
    else
        Update Status = 1;
        Dist = NewDist;
    end
else
    Dist = 0;
    MissAddr = A;
    Update Status = 1;
end
```

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 5, 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.

![Fig. 4. An attacking prefetch is filtered for sequential streams.](image1)

![Fig. 5. An attacking prefetch is used to calculate the next node address for linked streams, but is not moved to cache.](image2)

B. Prefetching Aware Thread Classification

Contestation for shared resources leads to a larger number of memory accesses and longer memory access time. We use \( t_{MH} \) (total time for miss handling), which is normalized to the result with no prefetching, to evaluate the contention of shared resources.

\[
t_{MH} = \frac{\text{total } \text{misses}}{\text{handle time for } \text{ith miss}}
\]

Additionally, we use the following metrics, which indicate different aspects of memory access behavior, to evaluate the impact of 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 usefulness of prefetch requests.

4) PL (prefetch lateness). This metric can be used to evaluate the timeliness of prefetches issuing.

The real-time feedback information collected by a hardware statistical module is used to classify threads. According to the results of normalized \( t_{MH} \), we divide applications into three collections, as shown in Table I.

1) The applications in the first collection have a \( t_{MH} \) increase of less than 10% (\( t_{MH} < 1.1 \)). This means that the influence of prefetching is slight, and these applications 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 \( t_{MH} \) (1.1 \(< t_{MH} < 1.3 \)), all applications have an MPKI higher than 100, which means these applications have larger memory requirements. If the applications have no proper data address patterns which can be predicted by the prefetching engine, 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 \( > 0.45 \)), we classify the applications into one of two categories, Mass Prefetching High Accuracy threads (MPHA) and Mass Prefetching and High Accuracy/Lateness threads (MPHAL).

3) In the third collection of normalized \( t_{MH} \) (\( t_{MH} > 1.3 \)), 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 \( t_{MH} \). We classify these applications as Mass Prefetching Low Accuracy threads (MPLA).

We are inspired by the result that the impact of prefetching on shared resource contention is closely related to thread’s memory behavior and prefetching performance. So it is reasonable to classify threads and manage prefetchers according to these metrics. The threads classifying policy is shown in Table I.
### C. Thread Classifying Directed Adjustment

When thread classifying is done, we can adjust each thread’s prefetcher algorithms according to the classifying results. We use the number of cache replacements (1024 by default) to define the time 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 cycle automatically depending on the run-time degree of resource contention.

According to TCD results of the running threads, we carry out the corresponding prefetching adjustment operation 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.

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. So we can temporarily shut down the threads’ prefetching engine to reduce its contention for the shared resources.

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. So, if there is no memory resource contention with other threads, 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. So when there are other threads competing for memory resources, the **MPLA** threads prefetching aggressiveness should be tuned down.

To recap, prefetching adjustment strategies are as follows,

- Shut off the prefetcher for **MMLP** threads.
- If only executing with **LM** threads concurrently, increase the **MPHA** threads’ prefetching aggressiveness and **MPHAL** threads’ prefetch distance.
- If executing with any type of threads other than **LM**, reduce the **MPLA** threads’ prefetching aggressiveness.

### D. Cache Replacement Mechanism

When the prefetched data is returned, it needs to be copied into the caches. We use a Least Recently Used (LRU) mechanism to deal with cache replacement. Figure 6 is an example of cache replacement with 4-way associative structure. General implementations of LRU mechanism keep the “LRU-bits” for cache lines and encode the order in which the cache lines have been accessed. For ease of description, we use a “sorting queue” to represent the order encoded in LRU-bits. The tail position of the sorting queue represents the least recently used cache line, and the head position represents the most recently used cache line. Normally, the prefetched data is treated as the most recently used data and is placed in the head position of the sorting queue. This mechanism is efficient if the prefetch accuracy is high. On the other side, if a prefetch request is miss-predicted, which means the prefetched data will not be used, the prefetched data will occupy a part of the cache footprint until it is sorted to the tail position and then replaced. This leads to a waste of cache resources.

We utilize a kind of adaptive insertion policy for our cache replacement mechanism [16]. According to the prefetching aware thread classifying, the prefetch accuracy can be estimated when the prefetched data is returned. For high prefetch accuracy, the prefetched data is placed in the head of the sorting queue as usual. Prefetched data with lower prefetch accuracy is placed in the second or third position in the sorting queue. If the prefetch accuracy is very low, the prefetched data is placed at the tail of the sorting queue to reduce the waste of cache footprint. Table III shows the thresholds used for our cache replacement policy.

### TABLE I. THREADS CLASSIFYING POLICY.

<table>
<thead>
<tr>
<th>Classifying Policy</th>
<th>Metrics</th>
<th>Applications</th>
</tr>
</thead>
<tbody>
<tr>
<td>LM</td>
<td>&lt; 110% &lt; 100</td>
<td>blackscholes, freqmine, swaptions, jacobs, mp5d, shallow, tspo, fmm, radix</td>
</tr>
<tr>
<td></td>
<td>&lt; 110% 100 ~ 130%</td>
<td>cannal, vips, emdl, radiosity, restruce</td>
</tr>
<tr>
<td></td>
<td>&gt; 130% &gt; 100</td>
<td>barnes, cholesky, fft, lu, water-spatial</td>
</tr>
<tr>
<td></td>
<td></td>
<td>blackscholes, freqmine, swaptions, jacobs, mp5d, shallow, tspo, fmm, radix</td>
</tr>
<tr>
<td></td>
<td></td>
<td>cannal, vips, emdl, radiosity, restruce</td>
</tr>
<tr>
<td></td>
<td></td>
<td>barnes, cholesky, fft, lu, water-spatial</td>
</tr>
</tbody>
</table>

### TABLE II. LEVELS OF PREFETCHER AGGRESSIVENESS.

<table>
<thead>
<tr>
<th>Aggressiveness Level</th>
<th>Prefetch Distance</th>
<th>Prefetch Degree</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>16</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>32</td>
<td>8</td>
</tr>
</tbody>
</table>

### TABLE III. THRESHOLDS USED FOR CACHE REPLACEMENT.

<table>
<thead>
<tr>
<th>Insert Position</th>
<th>Prefetch Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tail</td>
<td>&lt; 0.35</td>
</tr>
<tr>
<td>Third</td>
<td>0.35 ~ 0.45</td>
</tr>
<tr>
<td>Second</td>
<td>0.45 ~ 0.55</td>
</tr>
<tr>
<td>Head</td>
<td>&gt; 0.55</td>
</tr>
</tbody>
</table>
E. Implementation

To implement the proposed method, we need hardware support for computing the metrics used in our scheme, classifying of running threads, and tuning the parameters. Note that we use IMH to evaluate the contention of shared resources. This statistic is collected by the software simulator (see Section IV) 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 run-time of TCD mechanism. The hardware prefetching engine collects only four metrics (MPKI, PPKI, PA, and PL) to direct the run-time 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 prefetch request 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, its address is inserted into the addr-tag of PHT and the associated hit-bit is unset. If a demand request is generated, its address will be compared with all addr-tags in PHT, if matched, 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.

### IV. METHODOLOGY

We implement our mechanism on a cycle-level, execution-driven, in house simulator. The processor microarchitecture, the cache coherence, and communication substrate are faithfully modeled. The simulator models a MESI-based directory protocol with a detailed model of both stable and transient states and queuing of requests. PopNet [18] is used to model the packet-switched network. The network is a $10 \times 8$ mesh which connects the core tiles in the middle and the memory controller tiles at two sides. The details of the architectural parameters used for evaluation are shown in Table IV.

For the processor microarchitecture, we faithfully model the MIPS32 24KE [19]. The hybrid prefetching engine on each processor is configured to have a 16-entry MAP and an 8-entry PTB. The initial prefetching distance and degree are set to be 8 and 2, respectively. We use McPAT [20] to evaluate the power consumption of the processor cores in the case of 65 nm process technology. The DSENT [21] tool is exploited to calculate the power consumption of the networks-on-chip (NoC).

We perform our evaluation with a suite of parallel applications from SPLASH-2 [13], PARSEC [12], and other multi-threaded workloads. These applications are compiled using a cross-compiler to generate MIPS32 binaries. The limitation of the cross-compiler prevents us from running certain applications. Table V lists the applications used. Abbreviations are used in the data figures and listed in the table. Inputs for each application are listed along with a brief description of the application. Each application is fast-forwarded past the initialization.

We chose the feedback directed prefetching (FDP) [8] technique to compare with our method. We implement the proposed TCD (Thread Classifying Directed) prefetcher, as well as the attacking prefetch filter and the optimized cache replacement mechanism. To evaluate the prefetching influence on system performance, we collect the execution time of each application and normalize them to the results without prefetching. Shorter execution time means better system performance. We also calculate the geometric mean of the results to find out the system performance speedup ratio on average. The Energy-Delay Product (EDP) is used to evaluate the magnitude of power consumption. Lower EDP represents that one application completes its work with less energy and lower latency.

<table>
<thead>
<tr>
<th>Processor core (MIPS32 ISA compatible)</th>
<th>Fetch / Decode / Commit</th>
<th>Reg (int, fp)</th>
<th>Branch predictor</th>
<th>Br. mispred. penalty</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 / 1 / 1</td>
<td>27 / 7 / 7</td>
<td>(32, 32)</td>
<td>Binodal, 512 entries BHT</td>
<td>at least 7 cycles</td>
</tr>
</tbody>
</table>

| Memory hierarchy                      | L1 D cache (private)    | 16KB, 4-way, 32B line, 1cycle |
|                                       | L1 I cache (private)    | 32KB, 4-way, 32B line, 1cycle |
|                                       | 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 VCs, 2 cycles router |
|                                       | buffer: 5x12 flits, wire delay: 1 cycle per hop |

| Prefetching                            | Prefetch mode           | Linear (tunable distance & degree) |
|                                       | Chained (tunable distance & degree) |
|                                       | MAP                      | 16 entries |
|                                       | PTB                      | 8 entries |

---

1Instruction interpretation and Linux system call emulation are partially borrowed from [17].
TABLE V. BENCHMARKS

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Application</th>
<th>Input parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>PARSEC</td>
<td>dedup (de)</td>
<td>184 MB data</td>
</tr>
<tr>
<td></td>
<td>vgs (v)</td>
<td>1 image, 2002 × 3000 pixels</td>
</tr>
<tr>
<td></td>
<td>ferret (f)</td>
<td>167 queries, 10,000 images</td>
</tr>
<tr>
<td></td>
<td>freqmine (f)</td>
<td>160 blocks, 2000 data</td>
</tr>
<tr>
<td></td>
<td>streamcluster (s)</td>
<td>40,334 points per block, 1 block</td>
</tr>
<tr>
<td></td>
<td>canneal (ca)</td>
<td>400,000 elements</td>
</tr>
<tr>
<td></td>
<td>mackees (b)</td>
<td>65,536 options</td>
</tr>
<tr>
<td></td>
<td>shotty (sw)</td>
<td>64 shottos, 20,000 simulations</td>
</tr>
<tr>
<td></td>
<td>fluidanimate (fl)</td>
<td>990,000 transactions</td>
</tr>
<tr>
<td>SPLASH-2</td>
<td>barnes (ba)</td>
<td>10K particles</td>
</tr>
<tr>
<td></td>
<td>cholesky (ch)</td>
<td>816,172</td>
</tr>
<tr>
<td></td>
<td>fft (ft)</td>
<td>64K points</td>
</tr>
<tr>
<td></td>
<td>fmm (fm)</td>
<td>512 × 512 matrix, 16 × 16 blocks</td>
</tr>
<tr>
<td></td>
<td>lu (lu)</td>
<td>512 × 512 matrix, 256 × 256 pages</td>
</tr>
<tr>
<td></td>
<td>ocean (oc)</td>
<td>256 × 256 ocean</td>
</tr>
<tr>
<td></td>
<td>radiosity (ra)</td>
<td>10M integers, 1024</td>
</tr>
<tr>
<td></td>
<td>radix (rd)</td>
<td>10M integers, 1024</td>
</tr>
<tr>
<td></td>
<td>raytrace (t)</td>
<td>car</td>
</tr>
<tr>
<td></td>
<td>water-spatial (w)</td>
<td>512 molecules</td>
</tr>
<tr>
<td>Others</td>
<td>em3d (em)</td>
<td>300,000 nodes, arity-128</td>
</tr>
<tr>
<td></td>
<td>jacobi (j)</td>
<td>512 nodes, 10 iterations</td>
</tr>
<tr>
<td></td>
<td>mp3d (mp)</td>
<td>40,000 molecules</td>
</tr>
<tr>
<td></td>
<td>shallow (sh)</td>
<td>512 × 512, 20 steps</td>
</tr>
<tr>
<td></td>
<td>tspo (ts)</td>
<td>18 × 18</td>
</tr>
</tbody>
</table>

V. EXPERIMENTAL ANALYSIS

A. Reducing Prefetching-caused Invalidation

As discussed in Section III, 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 7. 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 our approach, the total time waiting for miss handling has been shortened in many applications, especially in barnes, fft, lu, and raytrace. On average, the proposed TCD prefetcher reduces the miss handling time by 7% compared to the multi-mode prefetcher.

B. Thread Classification

During the run-time of each application, the characteristic of memory access behavior changes with time. We use thread classifying approach to classify each thread dynamically at run-time. TCD optimization, which adjusts prefetching engines according to the classifying result, changes the memory access behavior of each thread, and has an impact on the distribution of thread classifying.

Figure 8 shows the distribution of the five types of threads, using TCD optimization results (right-side) compared to the normal results (left-side) without TCD. 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 canneal, ferret, and ocean applications. In addition, TCD stops the prefetching engine of MMLP threads, so the proportion of MMLP threads is decreased, such as em3d and shallow. The proportion of LM threads has been increased in most applications, this is because TCD optimization improves the prefetch accuracy and timeliness of prefetching, reducing cache pollution and competition for LLC.

C. System Performance

Figure 9 illustrates the influence of prefetching on multi-core system performance by showing the execution time normalized to the result with no prefetching. TA (Thread-aware) refers to the prefetching engine optimized with Thread Classifying Directed method. TCD is based on TA, using the attacking prefetch filter and cache replacement mechanism described in Section III.

The result shows that the use of the multimode prefetcher on shared memory multi-core systems can improve system performance for some applications while decrease system performance for others. The average speedup using the multimode prefetcher is 5%. After taking threads’ run-time behavior into consideration, TA reduces the conflicts among threads and improves the prefetch accuracy, such as canneal, dedup, and ferret. As we can see, the TCD method can increase the proportion of MPHA threads and decrease the percentage of MPLA threads, for canneal, dedup, and ferret applications. With high accuracy of prefetching, system performance increases. On average, TA has a speed up 8% compared to the multimode prefetcher mechanism. TCD, which gains benefit from the attacking prefetching filter and cache replacement mechanism, reduces the negative impact of prefetching-caused invalidation. On average, TA and TCD improve performance by 13% and 16% respectively, while FDP improves by 9% compared to no-prefetching system.

D. Energy-Delay Product

Figure 10 shows the EDP of each application. All results are normalized to that with no-prefetching. When using the TCD approach, the number of useless prefetching requests is minimized, and this saves unnecessary energy consumption. This can be proven by the results where on average, TCD saves 13% of EDP compared with the multimode prefetcher. On average, TCD reduces energy-delay product 3% more than FDP does.

E. Effect of TCD with HPAC Mechanism

We also implement and evaluate our mechanism with the HPAC mechanism [10] (namely HPAC using TCD locally). The HPAC 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, $BW Ci$ (Bandwidth Consumed by Core $i$) and $BW NOi$ (Bandwidth Needed by Cores Other than Core $i$) are used to keep track of prefetcher-caused inter-core interference in the shared memory system. While our mechanism uses the attacking filter and thread classifying directed adjustment to reduce prefetching-caused inter-thread invalidation.

Figure 11 shows the performance of TCD and FDP when employed in a system with HPAC mechanism. HPAC increases the performance of a system that uses FDP by 10%, and increases the performance of a system that uses TCD by 5%. Although there is no global control structure in TCD, inter-core interference has already been considered in the prefetching adjustment strategy. As a result, HPAC using TCD locally improves system performance by 2% over HPAC using FDP locally.

### F. 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 12 shows the memory bandwidth impact of prefetching. The results are normalized to the system with no-prefetching. The proposed TCD mechanism consumes 4% less memory bandwidth than FDP. Compared to a system only using TCD, a system using TCD with HPAC mechanism can decrease the memory bandwidth consumption by 1%.

### G. Sensitivity to L2 Cache Size and Memory Latency

We evaluate the sensitivity of TCD to different L2 cache sizes and memory latencies. Table VI shows the change in average IPC (instructions per cycle) and MBC (memory bandwidth consumption) provided by TCD over FDP for each configuration. TCD provides better performance and consumes less memory bandwidth than FDP for all evaluated cache sizes.

### H. Hardware Cost

Table VII shows the hardware cost of the proposed mechanism in terms of the required state. The additional storage is 19.8KB for a 64-node system. None of the required logic is on the critical path of the processor. This storage overhead of our mechanism is less than 0.13% of data-store size of the baseline 16MB L2 cache.

### VI. RELATED WORK

The key idea of hardware data prefetchers is using a special hardware unit to detect data address sequences and find a pattern. Based on this pattern, future data addresses can be predicted. Many hardware data prefetching techniques for sequential address pattern have been proposed, such as sequential prefetching [22], non-unit stride address sequence prefetching [23], and multi-stride prefetching [24], [25]. Techniques for prefetching chained address pattern have also been proposed, such as dependence based prefetching [15] and content directed prefetching method (CDP) [14], etc. The above prefetching techniques can significantly improve performance for different types of applications on single-core processors. In multi-core systems, however, among different cores there are lots of conflicts and competition for shared resources, such as LLC and off-chip memory. Since the introduction of prefetching will increase such competition, prefetching on a chip multiprocessor (CMP) without efficient management can lead to performance deterioration.

To relieve the impact of prefetching engines on shared resource contention, Srinath et al. [8] considered the influence of 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. Ebrahimi et al [9], [10] analyzed the contention of memory ports, bus bandwidth and banks on CMP systems caused by prefetching engines. By expanding the FDP method on single core processors to multi-core processors, their approach made a hybrid prefetching engine more efficient for a CMP system. Ebrahimi et al. [11] found the existing multi-core shared resource management strategy did not distinguish between ordinary and prefetching requests which results in incorrect scheduling decisions. They considered the existence of prefetching and its accuracy when dealing with data requests and making scheduling decisions which can help to improve the effectiveness of memory scheduling.

We focus on how to reduce prefetching-caused resource contention in multi-threaded applications. The work in this paper differs from the above in that, we discovered a new problem, i.e. that prefetching can cause inter-thread invalidations in multi-threaded applications when LLC is shared under MESI-based cache coherence protocol. We provide the
thread-aware adaptive data prefetcher as the solution to this problem. Besides, instead of directing prefetcheers 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.

VII. CONCLUSION

This paper presents a thread-aware data prefetching system for multi-core systems. Compared to existing multi-core prefetchers, our prefetching system can reduce inter-thread shared resource contention on the shared LLC and off-chip memory. Our approach combines two ideas: 1) it filters attacking prefetching requests if they would invalidate other threads’ shared data while avoiding the break of prefetching streams, and 2) it takes threads’ run-time memory requests and prefetching efficiency into consideration, and makes decisions to relieve the inter-thread contention on the shared LLC and memory system. This reduction in shared resource contention also brings energy efficiency, while improving system performance noticeably.

ACKNOWLEDGMENTS

The authors would like to thank Dmitry V. Ponomarev, Michael C. Huang, and Mei Yang for their comments on early versions of this paper, and the anonymous referees for their careful reviews and suggestions.

REFERENCES