Reevaluating Helper Thread Prefetching on Partitioned Shared Cache

Min Cai[1], Zhimin Gu[1], Yinxia Fu[1], Xin Zhao[1], Xianhe Sun[2]

[1]School of Computer Science and Technology, Beijing Institute of Technology, Beijing, China
[2]Department of Computer Science, Illinois Institute of Technology, Chicago, Illinois, USA,
min.cai.china@gmail.com, zmgu@x263.net

Abstract—For big data-crunching workloads exhibiting irregular memory access patterns in the era of data intensive computing, helper threaded data prefetching on shared cache chip multiprocessors speculatively issue last-level cache requests to the predicted memory addresses ahead of main thread references. Effective helper thread on CMPs demands that the helper thread should issue correct and timely LLC requests just before the main thread references them. Unfortunately, this ideal case cannot be fulfilled in the practical implementations of helper thread on CMPs. For observing and analyzing the characterizations of cache interference, first, we adapt traditional approaches to characterizing hardware prefetches such as prefetch accuracy, coverage, lateness, useful vs. useless prefetches to characterize helper thread on CMPs. Based on the insights from these characterizations, we then propose a fine-grained breakdown of helper thread cache requests such as good, bad and ugly on partitioned shared cache, and give its implementing mechanisms and algorithms on our cycle-accurate simulation environment. Last, experimental results show that our performance evaluating model for partitioned shared cache with helper thread prefetching is very efficient and the saving-energy helper thread is also existent.

Keywords—Chip multiprocessors, helper thread, intra-application cache interference, breakdown of helper thread cache requests, saving-energy analysis of helper thread

I. INTRODUCTION

For big data-crunching workloads exhibiting irregular memory access patterns in the era of data intensive computing, helper threaded data prefetching on shared cache chip multiprocessors (CMPs) speculatively issue last-level cache (LLC) requests to the predicted memory addresses ahead of main thread references[1], [2], [3]. Effective helper thread on CMPs demands that the helper thread (HT) should issue correct and timely LLC requests just before the main thread (MT) references them. Unfortunately, this ideal case cannot be fulfilled in the practical implementations of helper thread on CMPs: inaccurate and/or untimely LLC requests coming from the helper thread could not contribute to the main thread performance, but instead stress and pollute LLC if no effective LLC replacement and pollution-aware feedback techniques are employed. Several metrics have been proposed in the past for evaluating the effectiveness of hardware based data prefetching, among which prefetch accuracy and coverage are the most intuitive ones [4]. We can easily adapt the definitions of accuracy and coverage for hardware based data prefetching to the helper thread scheme, all helper thread requests can be categorized into “useful” and “useless” helper thread requests. A “useful” helper thread request is one whose brought data is hit by a main thread request before it is replaced, while a “useless” helper thread request is one whose brought data is replaced before it is hit by a main thread request. However, the classification of “useful” and “useless” about helper thread requests don’t care about the LLC pollution and inter-thread interference caused by helper thread cache requests.

For saving above questions with multicore simulation tools[5], the main contributions of this paper can be summarized as follows: 1) We adapt traditional approaches to characterizing hardware prefetches such as prefetch accuracy, coverage, lateness, useful vs. useless prefetches to characterize helper thread on CMPs. 2) Based on the insights from these characterizations, we then propose a fine-grained breakdown of helper thread cache requests such as good, bad and ugly for partitioned shared cache interferences, and give its implementing mechanisms and algorithms on our cycle-accurate simulation environment. 3) Experimental results show that our performance evaluating model for partitioned shared cache with helper thread prefetching on CMPs is very efficient and the saving-energy helper thread is also existent.

The rest of this paper is structured as follows. Section 2 describes the performance evaluating and analysis model. Section 3 presents the relevant implementing mechanisms and algorithms. Section 4 describes the cycle-accurate simulation framework for the target CMP architecture and experimental setup. Section 5 discusses the results of performance analysis and saving-energy on the detailed experiments. Section 6 provides related work. Section 7 concludes the paper.

II. PERFORMANCE ANALYSIS MODEL

A. The Scheme of Helper Thread on Partitioned Shared-Cache

As illustrated in Fig.1, the work flow of helper thread on a partitioned shared cache we implement here can be described as follows:1) The helper thread is spawned in the entry point *main()* of the program; 2) The helper thread remains dormant until some caller of the target hotspot function has been invoked and code placed in the caller wakes up the helper thread to let it start the *prelude* where the code in the helper thread skips some iterations of pointer traversals (i.e., there is no prefetch issued) to compensate the long time used for data prefetching in the helper thread as compared to the short time used in computation work in the main thread; 3) The helper thread enters a *stable state* of issuing the partitioned shared cache prefetch requests in loop iterations of pointer traversals ahead of the main thread until the execution of the program has passed some point(s) in the target hotspot function; 4) The code in helper thread is synchronizing pointers with the main thread and begin the next turn of servicing hotspots; 5) After all the prefetching work is done, the helper thread is destroyed in main(). Two key parameters in the helper thread scheme control its aggressiveness: 1) the number of loop iterations of pointer traversals that the helper thread code skip

after synchronizing with the main thread in the prelude is called *lookahead*, which is necessary given that the helper thread needs to stay ahead of the main thread in the beginning in each turn to amortize the difference of amount of work to do in the main thread and the helper thread in CPU time; 2) the number of loop iterations of pointer traversals in which helper thread code issue the partitioned shared cache prefetch requests in the stable state is called *stride*, which implies how much data prefetching work can be done in the helper thread in each turn in the attempt to not lag behind the main thread and pollute the partitioned shared cache too much. See [1] about our helper thread method in detail.

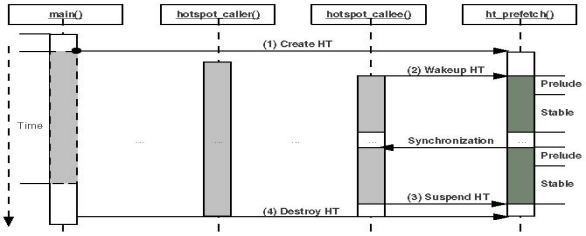


Figure 1. The Scheme of Helper Thread on CMPs

B. Adapting Traditional Hardware Prefetching Metrics to Helper Thread

Traditionally, prefetch accuracy, coverage and lateness are used to evaluate the effectiveness of hardware prefetchers [4]. In this section, we adapt the metrics to the helper thread scheme and describe what the metrics mean under the helper thread scheme.

a) Useful vs. Useless Helper Thread Requests: Helper thread requests can be categorized into useful and useless helper thread requests. A *useful helper thread request* is one whose brought data is hit by a main thread request before it is replaced, while a *useless helper thread request* is one whose brought data is replaced before it is hit by a main thread request.

b) Helper Thread Request Accuracy: Helper thread request accuracy is a measure of how accurately the helper thread scheme can predict and issue prefetch requests for the memory addresses that will be accessed by the main thread. It is defined as below

$$Accuracy = \frac{\#Useful\ Helper\ Thread\ Requests}{\#Helper\ Thread\ Requests} \quad (1)$$

where # Useful helper thread Requests is the number of cache lines brought by helper thread requests that are later on hit by helper thread requests. For the benchmarks with high helper thread request accuracy, performance increases as the aggressiveness of the helper thread scheme is increased.

c) Helper Thread Request Coverage: Helper thread request coverage is a measure of the fraction of main thread cache misses in the baseline version where helper thread is switched off that can be converted into hits by issuing helper thread cache misses ahead in the helper thread version. It is defined as below

$$Coverage = \frac{\#Useful\ Helper\ Thread\ Requests}{\#Main\ Thread\ Requests\ without\ Helper\ Thread} \quad (2)$$

d) Helper Thread Request Lateness: Helper thread request lateness is a measure of how timely the requests generated in the helper thread are with respect to the requests generated in the main thread that need the data brought by the helper thread. A helper thread request is said to be late if its requested data has not yet returned from the main memory by the time a main thread request references the data. Therefore, even though the helper thread request is accurate, it can only partially hide the latency incurred by a cache miss in the main thread. Helper thread request lateness can be defined as below

$$Lateness = \frac{\#Late\ Helper\ Thread\ Requeasts}{\#Useful\ Helper\ Thread\ Requests} \quad (3)$$

C. Cache-Pollution-Aware Breakdown for Helper Thread

To accommodate the case where one thread Tb accesses data element x that has been previously brought into the cache by another thread Ta, we introduce the notion of (Ta,Tb) inter-thread reuse distance. To show why the reuse distance is important in the helper thread scheme, we can consider the mst benchmark in the Olden suite. Its pointer traversing code structure inherently exhibits irregular memory access pattern in its original single threaded version, which renders the common LRU replacement policy inefficient to reduce cache misses. Here we use RD_{MT} to refer to the reuse distance in the main thread, and $ITRD_{HT\text{-}MT}$ to refer to the *inter-thread reuse distance* between helper thread and main thread where a data element is first brought to partitioned shared cache by helper thread, and later on used by the main thread. Therefore, the values of RD_{MT} in most main thread cache requests is high which reflects the irregular memory access pattern in mst. The values of $ITRD_{HT\text{-}MT}$ in helper thread cache requests should be very small when the helper thread scheme is efficient to reduce cache misses in main thread, an immediate follow-up main thread cache request will access the data and hit in the partitioned shared cache. Otherwise, large values of $ITRD_{HT\text{-}MT}$ indicate the inefficiency of the helper thread scheme where most data brought by helper thread is replaced before used by the main thread. Helper thread induced cache pollution with respect to main thread performance only happens when helper thread cache requests evict the data that are previously brought by main thread and immediately referenced again by main thread cache requests, but rarely happens when helper thread cache requests evict any data that was previously brought by main thread but will not be used by the main thread in the near future, which is typically the case in mst which exhibits a thrashing memory access pattern. Fortunately, as we will see, our inter-thread reuse distance prediction based cache replacement can be useful for mst with helper thread.

Based on $ITRD_{HT\text{-}MT}$ and RD_{MT}, we can construct a pollution-aware breakdown of helper thread cache requests where helper thread cache requests are classified into three types: good, bad and ugly. Let's assume when a helper thread cache request referencing data H evicts a cache line containing the victim data V which is brought by main thread, and afterwards the cache line containing H is evicted by a main thread cache request with data M. Therefore, there are two cache replacement involved: first H evicts V and then M evicts H. We use $RD_{MT}(V)$, $ITRD_{HT\text{-}MT}(H)$ and $RD_{MT}(M)$ to denote the reuse distance of V, H and M respectively. The greater the reuse distance of the data, the farther the data will be referenced again in time. A cache replacement is considered as optimal if the replacement makes the data with small reuse distance evicts the data with larger reuse distance, otherwise the replacement is considered as non-optimal. We have a *Cache-Pollution-Aware Breakdown for Helper Thread* as follows*:*

1) if $ITRD_{HT\text{-}MT}(H) < RD_{MT}(V) < RD_{MT}(M)$, then the helper thread cache request is considered as ***good*** because it evicts the data

that has larger reuse distance than its own. Its data is hit by the main thread before evicted. It has positive impact on the main thread performance since it reduces one main thread miss;

2) if $RD_{MT}(V) < ITRD_{HT\text{-}MT}(H) < RD_{MT}(M)$, then the helper thread cache request is considered as ***bad*** because it evicts the data that has smaller reuse distance than its own. It displaces a cache line that will later be needed by the main thread. It is harmful for main thread performance which should be prevented as much as possible;
3) if $RD_{MT}(M) < RD_{MT}(V)$ and $RD_{MT}(M) < ITRD_{HT\text{-}MT}(H)$, then the helper thread cache request is considered ***ugly*** because it has little performance impact on main thread performance because the requested data is not referenced by main thread before evicted and it does not evict any data that will be used by the main thread.

Good and bad helper thread cache requests are caused by the optimal and non-optimal cache replacement in the presence of helper thread on CMPs, respectively. We can conclude that Low $ITRD_{HT\text{-}MT}$ is an indicator of good performance of the helper thread scheme. As noted in the previous section, good helper thread cache requests can be further divided into timely requests and late requests. A helper thread request is called *late helper thread request* if its requested data has not yet returned from the main memory by the time a main thread cache request references the data. Late helper thread requests only partially hide the cache miss latency in main thread requests, but they are good indicators for potential performance improvement of the helper thread scheme because late helper thread requests may be converted to timely helper thread requests by adjusting the aggressiveness parameters of the helper thread scheme, i.e., lookahead and stride.

III. IMPLEMENTING MECHANISMS AND ALGORITHMS

A. Implementing Mechanisms

a) Cache request and replacement event tracking

To monitor the request and replacement activities in partitioned shared cache, we need to consider the event when the partition receives a request coming from the upper level cache, whether it is a hit or a miss. The event has a few important properties to be used in the experiment, e.g., the address of the requested cache line, the requester memory hierarchy access, line found in the partition, a boolean value indicating whether the request hits in the partition, and a boolean value indicating whether the request needs to evict some cache line.

b) Helper Thread cache request state tracking

In order to track the helper thread request states in the partition, we need to add one field named threadId to each cache line to indicate whether the line is brought by the main thread or the helper thread or invalid. Actually, another field named inflightThreadId is added to each cache line to indicate whether the line is being brought by the main thread or the helper thread to assist tracking late requests.

c) Helper Thread cache request victim state tracking

In order to track victims replaced by helper thread requests, we need to add an LRU cache named *Helper Thread Request Victim Cache (HTRVC)* to maintain the cache lines that are evicted by helper thread requests. The HTRVC has the same structure of the partition, but there is no direct mapping between cache lines and HTRVC lines. One field called HTRequestTag is added to HTRVC to enable reverse lookup in HTRVC by the helper thread request tag in the partition. HTRVC has the sole purpose of profiling, so it has no impact on performance.

d) Detecting Late Helper Thread cache Requests

Lastly, in order to measure late helper thread requests, we need to identify the event when a main thread request hits to a cache line which is being brought by an in-flight helper thread request coming from the upper level cache. This can be accomplished by monitoring the cache Miss Status Holding Register (MSHR). MSHR is a hardware structure that keeps track of all in-flight memory requests. A helper thread cache request is late if a main thread cache request for the same address is generated while the helper thread cache request is in the cache MSHR waiting for main memory.

B. Algorithms for Tracking Helper Thread Cache Requests and Victims

There are two invariants that should be maintained between the partition and HTRVC per set 1) Number of helper thread Lines in the Set = Number of Victim Entries in the HTRVC Set; 2) Number of Victim Entries in HTRVC Set + Number of Valid main thread Lines in Set ≤ partition Set Associativity. Helper thread lines refer to the cache lines that are brought by helper thread requests. From the above two invariants, we can easily conclude that: Number of helper thread Lines in the Set+ Number of Valid main thread cache lines in Set ≤ LLC Associativity. Actions should be taken in partition and HTRVC when filling a cache line and servicing an incoming cache request either from the main thread or the helper thread.

a) Actions taken on Inserting a cache Line

When filling a cache line, we should consider four cases 1) An helper thread request evicts an INVALID line. In this case, no eviction is needed; 2) An helper thread request evicts an cache line which is previously brought by a main thread request; 3) An helper thread request evicts an cache line which is previously brought by a helper thread request; 4) An main thread request evicts an cache line which is previously brought by a helper thread request. In the above last three cases, eviction is needed to make room for the incoming helper thread or main thread request. Specific actions taken on the above four cases are listed in Fig.2, where hitInLLC indicates whether the request hits in partition or not; requesterIsHelperThread indicates whether the request comes from helper thread or not; hasEviction indicates whether the request needs to evict some data; lineFoundIsHelperThread indicates whether the cache line found is brought by helper thread or not.

b) Actions taken on Servicing an Incoming cache Request

When servicing an incoming cache request, either hit or miss, we should consider four cases: 1) cache miss and victim hit, which indicates a bad helper thread request. This happens when

helper thread request evicts useful data; 2) Helper thread cache hit, which indicates a good helper thread request. This happens when helper thread requested data is hit by main thread request before evicted data; 3) Helper thread cache hit and victim hit. This happens when useful data is evicted and brought back in by helper thread request; 4) Main thread cache hit and victim hit. This happens when useful data is evicted and brought back in by helper thread request and hit to by main thread request;Specific actions taken on the above four cases are listed in Fig.3, where mainThreadHit indicates whether the request comes from main thread and hits in the partition; helperThreadHit indicates whether the request comes from helper thread and hits in the partition; and victimHit indicates whether the request comes from main thread and hits in the HTRVC.

```
//Case 1
if(requesterIsHelperThread && !hitInLLC && !hasEviction) {
  llc.setHelperThread(set, llcLine.way);
  htrvc.insertNullEntry(set);
}
//Case 2
else if(requesterIsHelperThread && !hitInLLC && hasEviction
      && !lineFoundIsHelperThread) {
  llc.setHelperThread(set, llcLine.way);
  htrvc.insertDataEntry(set, llcLine.tag);
}
//Case 3
else if(requesterIsHelperThread && !hitInLLC && hasEviction
      && lineFoundIsHelperThread) {
}
//Case 4
else if(!requesterIsHelperThread && !hitInLLC &&
     hasEviction && lineFoundIsHelperThread) {
  llc.setMainThread(set, llcLine.way);
  htrvc.invalidateVictimLine(set, wayOfVictimLine);
}
```

Figure 2. Actions Taken When Inserting a cache Line

```
//Case 1
if(!mainThreadHit && !helperThreadHit && victmHit) {
  badHelperThreadRequests++;
  htrvc.clearVictimLine(set, victmLine.way);
}
//Case 2
else if(!mainThreadHit && helperThreadHit && !victmHit) {
  llc.setMainThread(set, llcLine.way);
  (timely or late)HelperThreadRequests++;
  htrvc.invalidateVictimLine(set, wayOfVictimLine);
}
//Case 3
else if(!mainThreadHit && helperThreadHit && victmHit) {
  llc.setMainThread(set, llcLine.way);
  htrvc.invalidateVictimLine(set, wayOfVictimLine);
}
//Case 4
else if(mainThreadHit && !helperThreadHit && victmHit) {
  htrvc.clearVictimLine(set, victmLine.way);
}
```

Figure 3. Actions Taken When Servicing a cache Request

IV. Experimental Methodology

The simulated target CMP architecture has two cores where each core is a two-way SMT with its own private L1 caches (32KB 4-way data caches and 32KB 4-way instruction caches). Both cores share a 96KB 8-way L2 cache. MESI inclusive directory coherence is maintained between L1 caches. Both L1 and L2 caches use LRU replacement policy. An LRU cache called HTRVC is attached to the L2 cache to implement the breakdown of helper thread cache requests. Detailed micro-architecture parameters are listed in Tab.I. We use the open source Archimulator CMP architectural simulation environment (See http://github.com/mcai/archimulator/ for details) in our experiments mentioned in this work. Archimulator is an object-oriented execution-driven application-only architectural simulator written completely in Java and running on 32 and 64 bit Linux based operating systems. It provides three modes of functional simulation, cycle-accurate simulation and two-phase fast forward and measurement simulation of MIPS II executables on CMP architectures consisting of out-of-order super-scalar cores and configurable memory hierarchies with directory-based MESI coherence. It supports simulating Pthreads based multithreaded workloads.

We perform our evaluation using the original version and manually coded helper thread version of mst in the Olden benchmark suite. All applications are cross-compiled using gcc flag "-O3" and run until completion using cycle-accurate simulation. Input set for mst is "2048 1". Default Values of helper thread lookahead and stride parameters are 20 and 10, respectively, if not specified explicitly.

Table I BASELINE HARDWARE CONFIGURATIONS

Pipeline	4-wide superscalar out-of-order core				
	2 cores, 2 threads per core				
	Physical register file capacity	128			
	Decode buffer capacity	96			
	Reorder buffer capacity	96			
	Load store queue capacity	48			
	Branch predictor	Perfect			
FUs	FU Name	Count	Operation Lat	Issue Lat	
	Int. ALU	8	2	1	
	Int. Multiply	2	3	1	
	Int. Division		20	19	
	Fp. Add	8	4	1	
	Fp. Compare		4	1	
	Fp. Convert		4	1	
	Fp. Multiply	2	8	1	
	Fp. Division		40	20	
	Fp. Sqrt		80	40	
	Read Port	4	1	1	
	Write Port		1	1	
Cache	Name	Size	Assoc	Line Size	Hit Lat.
	I Cache	32KB	4	64B	1
	D Cache	32KB	4	64B	1
	L2 Cache	2MB	8	64B	10
Network	Switch based P2P topology, 32B link width				
Memory	4GB, 200-cycle fixed latency				

V. Results and Analysis

A. Performance Results

Here we assume the shared L2 cache is partitioned between different workloads on the CMP. Thus in the following discussion, the varying L2 size refers to the size of the effective partition of L2 cache that is assigned to the running of our workload, mst baseline version or mst helper thread version where the total L2 size is fixed at 2 MB. Tab.II shows the overall performance of mst benchmark baseline version, in which L2 size is varied from 96 KB to 2 MB and L2 associativity is varied from 2 to 32, respectively. In Tab.II (a), we can see that when the L2 size is greater than 128 KB, there is no difference in execution time and there is no eviction observed in the L2 cache, implying the working set of mst "2048 1" with helper thread switched off is near 128 KB. L2 sizes larger than 128 KB contribute little to the overall performance. Tab.II (b) shows that the execution time is insensitive to the L2 associativity when the latter is greater than 4.

Tab.III shows the overall performance of mst benchmark helper thread version, in which L2 size is varied from 96 KB to 2 MB and L2 associativity is varied from 2 to 32, respectively. In Tab.III (a), we can see that when the L2 size is greater than 512 KB, there is no difference in execution time and no eviction observed in the L2 cache, implying the working set of mst "2048 1" with helper thread switched on is near 512 KB. L2 sizes larger than 512 KB contribute little to the overall performance. Tab.III (b) shows that the execution time is reduced slightly as the L2 associativity is increased from 2 to 32. The increase of L2 associativity can also reduce the number of bad prefetches as well.

Overall, low IPCs are observed among the experiments shown in Tab.II and Tab.III, but IPCs in Tab.III is better. A maximum 22% reduction of execution is achieved when the lookahead is 20 and stride is 10 under L2 size greater than 512MB and L2 associativity greater than 8, between baseline and helper thread versions of mst in these basic experiments, which shows our implemented helper thread scheme is non-trivially effective for the mst benchmark. The helper thread parameters of lookahead and stride should and can be dynamically changed to balance the portions of timely and late prefetches using feedback directed mechanisms, thus leading to the maximal performance of helper thread on CMPs.

Table II MST BASELINE PERFORMANCE

L2 Size	L2 Assoc	Total Cycles	Speedup	IPC	Main Thread Hit	Main Thread Miss	L2 Hit Ratio	L2 Evictions	L2 Occupancy Ratio
96 KB	8	4001166884	1.0000	0.2100	14562	12695429	0.0010	943	0.4300
128 KB	8	4001145025	1.0000	0.2100	14560	12695141	0.0010	0	0.4390
256 KB	8	4001145025	1.0000	0.2100	14560	12695141	0.0010	0	0.2190
512 KB	8	4001145025	1.0000	0.2100	14560	12695141	0.0010	0	0.1100
1 MB	8	4001145025	1.0000	0.2100	14560	12695141	0.0010	0	0.0550
2 MB	8	4001145025	1.0000	0.2100	14560	12695141	0.0010	0	0.0270

(a) Impact of L2 Size

L2 Size	L2 Assoc	Total Cycles	Speedup	IPC	Main Thread Hit	Main Thread Miss	L2 Hit Ratio	L2 Evictions	L2 Occupancy Ratio
96 KB	2	4005895559	1.0000	0.2100	17299	12764949	0.0010	683068	0.4130
96 KB	4	4001435947	1.0011	0.2100	14581	12707953	0.0010	47911	0.4280
96 KB	8	4001166884	1.0011	0.2100	14562	12695429	0.0010	943	0.4300
96 KB	16	4001147526	1.0011	0.2100	14561	12695163	0.0010	135	0.5200
96 KB	32	4001145232	1.0011	0.2100	14560	12695141	0.0010	10	0.5790

(b) Impact of L2 Associativity

Table III MST HELPER THREAD PERFORMANCE

L2 Size	L2 Assoc	Look ahead	Stride	Total Cycles	Speedup	IPC	Main Thread Hit	Main Thread Miss	L2 Hit Ratio	L2 Evictions	L2 Occupancy Ratio	Helper Thread Hit	Helper Thread Miss	Redundant MSHR	Redundant Cache	Timely	Late	Bad	Ugly
96 KB	8	20	10	33237 36481	1.000 0	0.299 0	41050 24	89208 17	0.200 0	15733 04	0.607 0	19893 1	83298 35	11237	18769 4	38703 40	6640	12663	86238 1
128 KB	8	20	10	32766 85422	1.014 4	0.303 0	42138 78	87423 30	0.206 0	3667	0.486 0	19183 6	82206 27	340	19149 6	39831 29	4482	47	694
256 KB	8	20	10	32766 37422	1.014 4	0.303 0	42139 91	87403 50	0.206 0	1008	0.259 0	19186 7	82205 37	327	19154 0	39833 11	4436	40	536
512 KB	8	20	10	32766 22703	1.014 4	0.303 0	42141 52	87399 42	0.206 0	0	0.151 0	19191 9	82202 84	359	19156 0	39834 45	4413	0	620
1 MB	8	20	10	32766 22703	1.014 4	0.303 0	42141 52	87399 42	0.206 0	0	0.076 0	19191 9	82202 84	359	19156 0	39834 45	4413	0	1193
2 MB	8	20	10	32766 22703	1.014 4	0.303 0	42141 52	87399 42	0.206 0	0	0.038 0	19191 9	82202 84	359	19156 0	39834 45	4413	0	1591

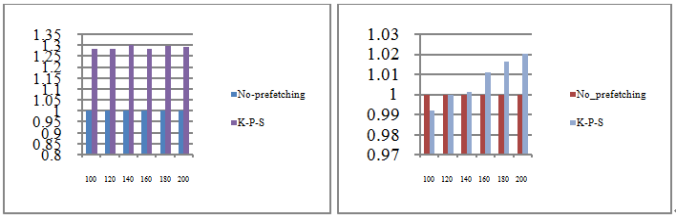
(a) Impact of L2 Size

L2 Size	L2 Assoc	Look ahead	Stride	Total Cycles	Speedup	IPC	Main Thread Hit	Main Thread Miss	L2 Hit Ratio	L2 Evictions	L2 Occupancy Ratio	Helper Thread Hit	Helper Thread Miss	Redundant MSHR	Redundant Cache	Timely	Late	Bad	Ugly
96 KB	2	20	10	33757 28053	1.000 0	0.296 0	38087 45	93391 29	0.183 0	44905 49	0.529 0	21471 4	85704 05	28156	18655 8	35703 56	6552	16219	19390 57
96 KB	4	20	10	33538 19652	1.006 5	0.298 0	39526 82	91118 09	0.191 0	28515 92	0.570 0	20618 1	84574 61	20359	18582 2	37178 97	4777	21284	13513 99
96 KB	8	20	10	33237 36481	1.015 6	0.299 0	41050 24	89208 17	0.200 0	15733 04	0.607 0	19893 1	83298 35	11237	18769 4	38703 40	6640	12663	86238 1
96 KB	16	20	10	32772 32422	1.030 1	0.303 0	42150 25	87498 87	0.206 0	51654	0.644 0	19235 2	82253 56	443	19190 9	39845 72	4566	30	3500
96 KB	32	20	10	32766 63422	1.030 2	0.303 0	42142 38	87400 76	0.206 0	695	0.646 0	19194 9	82203 86	350	19159 9	39834 59	4459	4	63

(b) Impact of L2 Associativity

B. Saving- Energy Analysis Results

We use SESC simulator to study the saving-energy case of mst benchmark under 8-way 32k L1 date cache,8-way 32k L1 instruction cache and 16-way 4M L2. In Fig. 4, K-P-S is our helper thread method where k is above lookahead, P is above stride, and S=K+P. We find that when K=100 and P=50 our helper thread method can be the best case with saving-energy and high performance.



(a) Speedup (b) Saving-Energy Cases

Figure 4. Saving-Energy Analysis with High Performance When k is Varied from 100 to 200

VI. RELATED WORK

Helper thread mechanisms have been studied for a long time, but cache interference and breakdown of helper thread cache requests have not been studied before. Here we briefly describe previous work in the hardware prefetching. Our breakdown of helper thread cache requests is mostly similar to the work in [6], which presented a hardware structure called the *Evict Table* to gauge the amount of shared cache pollution caused by hardware prefetching. The HTRVC in our proposal is similar to the evict table, however it is used for tracking helper thread request victims instead of hardware prefetch victims. Good, bad and ugly requests are identified based on cache replacement activities involved by helper thread. [7] took the bandwidth consumption into account, and developed a multiprocessor prefetch traffic and miss taxonomy that builds on an existing uniprocessor prefetch taxonomy. The problem of cache content management in the presence of data prefetching, are studied in the previous works in the forms of software [8], hardware [9], [10] and hybrid [4]. [11] proposed a prefetch buffer that can be attached to the L2 cache to filter the prefetched data from polluting the L2 cache, which is similar to but different from our proposed HTRVC cache structure whose sole function is for profiling helper thread cache requests' victims. [12] proposed Prefetch-Aware Cache Management (PACMan) to dynamically estimates and mitigates the degree of prefetch-induced cache interference by modifying the cache insertion and hit promotion policies. [13] proposed a low-cost feedback directed mechanism for hardware prefetching.

VII. CONCLUSION

For big data-crunching workloads exhibiting irregular memory access patterns in the era of data intensive computing, this paper first discusses the reuse distances about the helper thread on partitioned shared cache and their implications on performance. Based on this, we then propose the breakdown of helper thread cache requests and its implementation details on a cycle-accurate CMP architectural simulator. Experimental results of the memory intensive benchmark mst show that our performance evaluating model for partitioned shared cache with helper thread prefetching on CMPs is very efficient and the saving-energy helper thread is also existent. We will take the improving of bad state and ugly state in account in our future work.

ACKNOWLEDGMENTS this work was supported by the National Natural Science Foundation of China under the contract No. 61070029. (Corresponding author: Zhimin Gu,zmgu@x263.net)

REFERENCES

[1] Yan Huang, Jie Tang, Zhimin Gu, Min Cai, Jianxun Zhang, Ninghan Zheng. "The Performance Optimization of Threaded Prefetching for Linked Data Structures", *International Journal of Parallel Programming, Vol.40, no.2*, **2012, pp.**141-163

[2] S. Byna, Y. Chen, and X.-H. Sun, "A taxonomy of data prefetching mechanisms," in *Proceedings of the The International Symposium on Parallel Architectures, Algorithms, and Networks*, ISPAN '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 19–24.

[3] J. Lee, C. Jung, D. Lim, and Y. Solihin, "Prefetching with helper threads for loosely coupled multiprocessor systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 20, no. 9, pp. 1309–1324, sep 2009.

[4] V. Srinivasan, E. S. Davidson, and G. S. Tyson, "A prefetch taxonomy," *IEEE Trans. Computers*, vol. 53, no. 2, pp. 126–140, 2004.

[5] A. Rogers, M. C. Carlisle, J. H. Reppy, and L. J. Hendren, "Supporting dynamic data structures on distributed-memory machines," *ACM Transactions on Programming Languages and Systems*, vol. 17, no. 2, pp. 233–263, mar 1995.

[6] B. Mehta, D. Vantrease, and L. Yen, "Cache showdown: The good, bad and ugly," Tech. Rep., 2004.

[7] N. D. E. Jerger, E. L. Hill, and M. H. Lipasti, "Friendly fire: under-standing the effects of multiprocessor prefetches," in *ISPASS*. IEEE Computer Society, 2006, pp. 177–188.

[8] Z. Wang, K. S. McKinley, A. L. Rosenberg, and C. C. Weems, "Using the compiler to improve cache replacement decisions," in *Proc. 2002 International Conference on Parallel Architectures and Compilation Techniques (11th PACT'02)*. Charlottesville, Virginia, USA: IEEE Computer Society, sep 2002, p. 199.

[9] A.-C. Lai, C. Fide, and B. Falsafi, "Dead-block prediction and dead-block correlating prefetchers," in *Proceedings of the 28th Annual Inter-national Symposium on Computer Architecture*, Göteborg, Sweden, jun 30–jul 4, 2001, pp. 144–154.

[10] T.-F. Chen and J.-L. Baer, "Effective hardware-based data prefetching for high-performance processors," *IEEE Transactions on Computers*, vol. 44, no. 5, pp. 609–623, may 1995.

[11] W. Y. Chen, S. A. Mahlke, P. P. Chang, and W.-m. W. Hwu, "Data access microarchitectures for superscalar processors with compiler-assisted data prefetching," in *Proceedings of the 24th Annual International Symposium on Microarchitecture*, Albuquerque, New Mexico, nov 18– 20, 1991, pp. 69–73.

[12] C.-J. Wu, A. Jaleel, M. Martonosi and J. S. Emer, "Pacman: prefetch-aware cache management for high performance caching," in *MICRO*, 44rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2011, 3-7 December 2011, ACM, 2011, pp. 442–453.

[13] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt, "Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers," in *Proc. 13th International Conference on High-Performance Computer Architecture (13th HPCA'07)*. San Francisco, CA, USA: IEEE Computer Society, Feb 2007, pp. 63–74.