Characterizing Intra-Application LLC Interference Caused by Helper Threaded Data Prefetching on CMPs

Min Cai, Zhimin Gu[*†]

Abstract

For workloads exhibiting irregular memory access patterns, helper threaded data prefetching on shared cache chip multiprocessors (CMPs) speculatively issue last-level cache (LLC) requests to the predicted memory addresses ahead of computation core references. Effective helper threaded data prefetching 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 can not be fulfilled in the practical implementations of helper threaded data prefetching on CMPs: inaccurate and/or untimely LLC requests coming from 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.

This paper characterizes the degree by which intra-application LLC interference is caused by inter-core data prefetches in the above helper threaded data prefetching scheme. Based on cycle-accurate architectural simulation of the memory intensive benchmark mst from Olden, we first adapt traditional approaches to characterizing hardware prefetches such as prefetch accuracy, coverage and lateness, useful vs. useless prefetches to characterize helper threaded data prefetching on CMPs. Based on the insights from these characterizations, we then propose a fine-grained taxonomy of helper threaded inter-core data prefetches to study the intricate interactions between intra-application LLC interference and parameters of the helper threaded data prefetching scheme. Experimental results show that: (1) there is non-trivial intra-application LLC interference caused by inter-core data prefetches in helper-threaded data prefetching on CMPs; (2) selecting proper parameters, such as lookahead and stride, of the helper threaded data prefetching scheme play a key role in maximizing the performance of the scheme. Overall, our characterizations are important in interpreting the effectiveness of software-initiated helper threaded data prefetching on CMPs by cycle-accurate architectural simulation, and highlighting the opportunities and challenges of optimizing helper threaded data prefetching on CMPs in a dynamic and feedback-directed way.

Keywords: Chip multiprocessors, helper threaded data prefetching on CMPs, cache replacement, cache pollution, intra-application LLC interference

1 Introduction

For workloads exhibiting irregular memory access patterns, helper threaded data prefetching [1, 2] on shared cache chip multiprocessors (CMPs) speculatively issue last-level cache (LLC) requests to the predicted memory addresses ahead of computation core references. Effective helper threaded data prefetching 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 can not be fulfilled in the practical implementations of helper threaded data prefetching on CMPs: inaccurate and/or untimely LLC requests coming from helper thread could not contribute

[*]Min Cai, School of Computer Science and Technology, Beijing Institute of Technology, Beijing, China, e-mail: min.cai.china@gmail.com.

[†]Zhimin Gu, School of Computer Science and Technology, Beijing Institute of Technology, Beijing, China, e-mail: zmgu@x263.net.

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 [3]. We can easily adapt the definitions of accuracy and coverage for hardware based data prefetching to the helper threaded prefetching scheme. Helper thread request accuracy is defined as the ratio of the number of useful helper thread requests to the number of total helper thread requests. And helper thread request coverage is defined as the ratio of the number of useful helper thread requests to the number of main thread misses plus main thread hits to helper thread requested data. Here, a helper thread request is called useful when its requested data is referenced by main thread before evicted. Furthermore, similar to the approach to hardware prefetching, to measure coverage and accuracy, all helper thread requests can be categorized into “useful” and “useless” helper thread requests [3]. 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 above traditional accuracy and coverage metrics for helper thread requests and the classification of “useful” and “useless” helper thread requests don’t care about the LLC pollution and inter-thread interference caused by helper thread LLC requests, which are two kinds of deficiencies in the helper threaded prefetching scheme.

This paper characterizes the degree by which intra-application LLC interference is caused by inter-core data prefetches in the above helper threaded data prefetching scheme. Based on cycle-accurate architectural simulation of the memory intensive benchmark mst from the Olden pointer traversing benchmark suite [4], we first adapt traditional approaches to characterizing hardware prefetches such as prefetch accuracy coverage and lateness, useful vs. useless prefetches to characterize helper threaded data prefetching on CMPs. Based on the insights from these characterizations, we then propose a fine-grained taxonomy of helper threaded inter-core data prefetches to study the intricate interactions between intra-application LLC interference and parameters of the helper threaded data prefetching scheme. Experimental results show that: (1) there is non-trivial intra-application LLC interference caused by inter-core data prefetches in helper-threaded data prefetching on CMPs; (2) selecting proper parameters, such as lookahead and stride, of the helper threaded data prefetching scheme play a key role in maximizing the performance of the scheme. Overall, our characterizations are important in interpreting the effectiveness of software-initiated helper threaded data prefetching on CMPs by cycle-accurate architectural simulation, and highlighting the opportunities and challenges of optimizing helper threaded data prefetching on CMPs in a dynamic and feedback-directed way. We assume here a two level cache hierarchy where L1 caches are private and the L2 cache is shared among all processor cores on a single chip.

The main contributions of this paper can be summarized as answers for the following two questions:

1. What is the relationship between the shared cache behavior and the overall performance improvement in helper threaded data prefetching on CMPs;

2. How can the helper thread LLC requests be classified to reflect the inter-thread LLC interference caused by helper thread LLC requests in helper threaded data prefetching on CMPs.

The rest of this paper is structured as follows. Section 2 describes the cycle-accurate simulation framework for the target CMP architecture and experimental setup. Section 3 presents the characterization of LLC interference caused by helper threaded data prefetching on CMPs. Section 4 discusses the performance results on the detailed experiments. Section 5 provides related work. Section 6 concludes the paper.

2 Methodology

2.1 Target CMP Architecture

As shown in Fig.1, 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 taxonomy of helper thread LLC requests, as will be explained in the next section. Detailed microarchitecture parameters are listed in Tab.1.

2.2 Simulation Framework

We use the open source Archimulator[1] CMP architectural simulation environment 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.

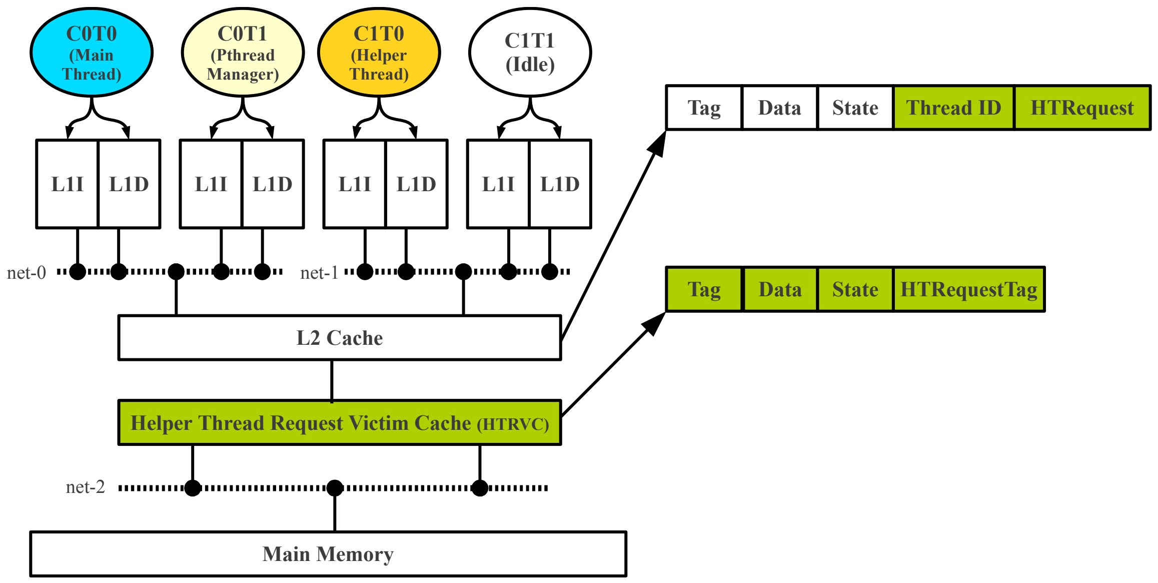


Fig. 1: Target CMP Architecture

Static Software Context to Hardware Thread Mappings In a typical Pthreads based helper thread program, there are three threads when running: main thread, helper thread and the Pthreads manager thread. The Pthreads manager thread takes the role of spawning, suspending and resuming helper thread by passing signals to helper thread. Consider a simulated target CMP machine which has two cores where each core supports two hardware threads. In our application-only simulation using Archimulator, without the OS intervention, one hardware thread can only run at least one software context (or simply called thread). Therefore, the typical software context to

[1] See http://github.com/mcai/archimulator/ for details.

hardware thread mappings can be: C0T0 → main thread, C0T1 → Pthreads manager thread, C1T0 → helper thread (C = core, T = thread), as shown in Fig.1. We use this context mapping in the following discussions.

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 Predictors	Perfect branch predictor				
Execution Units	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 Square Root		80	40	
	Read Port	4	1	1	
	Write Port		1	1	
Cache Geometries	Name	Size	Associativity	Line Size	Hit Lat.
	I Cache	32KB	4	64B	1
	D Cache	32KB	4	64B	1
	L2 Cache	96KB	8	64B	10
Interconnect	Switch based P2P topology, 32B link width				
Main Memory	4GB, 200-cycle fixed latency				

Tab. 1: Baseline Hardware Configurations

2.3 Benchmarks and Input Sets

We perform our evaluation using the original version and manually coded helper threaded version of mst in the Olden pointer traversal 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 helper threaded prefetching lookahead and stride are 20 and 10, respectively.

3 Characterizing LLC Interference Caused by Helper Threaded Prefetching

3.1 The Scheme of Helper Threaded Data Prefetching on Shared-Cache CMPs

As illustrated in Fig.2, the work flow of helper threaded data prefetching on shared-cache CMPs 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 LLC 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 parameters in the helper threaded prefetching scheme controls 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*; (2) the number of loop iterations of pointer traversals in which helper thread code issue LLC prefetch requests in the stable state is called *stride*.

In the traditional helper threaded prefetching configuration, the values of lookahead and stride are selected empirically and hard coded, which is unable to accommodate the unpredictable nature of L2 cache runtime behavior in the presence of helper thread requests. Based on our following characterization of helper thread L2 requests, a dynamic feedback directed mechanism can be built in which the processor changes the value of lookahead and stride using some temporary register to adjust the aggressiveness of the helper threaded prefetching scheme.

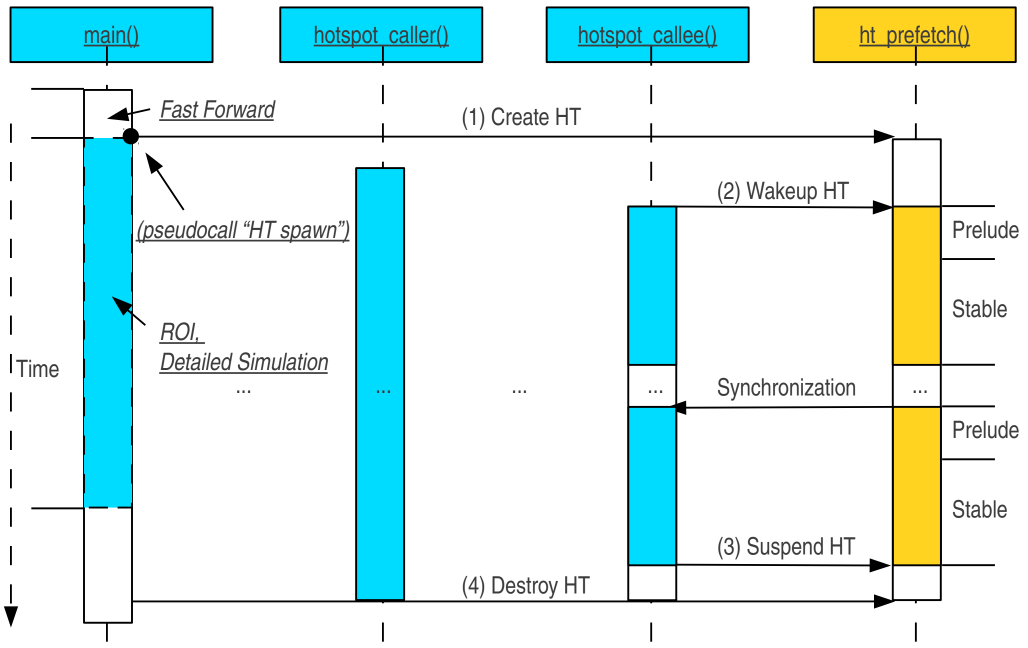


Fig. 2: The Scheme of Helper Threaded Data Prefetching on CMPs

3.2 Adapting Traditional Hardware Prefetching Metrics to Helper Threaded Prefetching

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

Useful vs. Useless Helper Thread LLC Requests Helper thread requests can be categorized into useful and useless helper thread requests [3]. 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.

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

$$Helper\,Threaded\,Prefetch\,Accuracy = \frac{Number\,of\,Useful\,Helper\,Thread\,LLC\,Requests}{Number\,of\,Helper\,Thread\,LLC\,Requests}$$

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

Helper Thread LLC Request Coverage *Helper thread LLC request coverage* is a measure of the fraction of all main thread LLC misses that can be eliminated by issuing helper thread LLC misses ahead. It is defined as below

$$Helper\,Threaded\,Prefetch\,Coverage = \frac{Number\,of\,Useful\,Helper\,Thread\,LLC\,Requests}{Number\,of\,Main\,Thread\,LLC\,Requests\,WIthout\,Helper\,Threaded\,Prefetch}$$

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

$$Helper\,Threaded\,Prefetch\,Lateness = \frac{Number\,of\,Late\,Helper\,Thread\,LLC\,Requests}{Number\,of\,Useful\,Helper\,Thread\,LLC\,Requests}$$

3.3 Characterizing Intra-application LLC interference

3.3.1 Intra-Application Reuse Distances in Helper Threaded Data Prefetching on CMPs

The notion of reuse distance can shed light on how a pollution-aware taxonomy of helper thread LLC requests can be constructed. The traditional intra-thread reuse distance of a reference to data element x is defined as the number of unique memory references between two consecutive accesses of x in the same thread (or ∞ if the element has not been referenced thereafter). In a k-way set-associative cache a cache miss with reuse distance rd can be classified by the value of reuse distance as below

1. rd $<$ k indicates it is a *conflict miss*;
2. k$<=$ rd $< \infty$ indicates it is a *capacity miss*;
3. rd $= \infty$ indicates it is a *cold miss.*

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, as compared to the traditional intra-thread reuse distance applied to either Ta or Tb.

To show why the notion of inter-thread reuse distance is important in the helper threaded prefetching scheme as compared to the traditional notion of intra-thread reuse distance, 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 LLC misses. Here we use RD_{MT} to refer to the *intra-thread reuse distance* in the main thread, and $ITRD_{HT-MT}$ to refer to the *inter-thread reuse distance* between helper thread and main thread where a data element is first brought to LLC by helper thread, and later on used by main thread. Therefore, the values of RD_{MT} in most main thread LLC requests is high which reflects the irregular memory access pattern in mst. The values of $ITRD_{HT-MT}$ in helper thread LLC requests should be very small when the helper threaded prefetching scheme is efficient to reduce LLC misses in main thread where for most helper thread LLC miss, an immediate follow-up main thread LLC request will access the data brought by the previous LLC request and hit in the LLC. Otherwise, large values of $ITRD_{HT-MT}$ indicate the inefficiency of the helper threaded prefetching scheme where most data brought by helper thread is replaced before used by main thread.

Helper thread induced cache pollution with respect to main thread performance only happens when helper thread LLC requests evict the data that are previously brought by main thread and immediately referenced again by main thread LLC requests, but rarely happens when helper thread LLC requests evict any data that was previously brought by main thread but will not be used by main thread in the near future, which is typically the case in mst which exhibits a thrashing memory access pattern and thus most of the data requested from its delinquent PCs have instant main thread intra-thread reuse distances but small helper thread-main thread inter-thread reuse distances, which renders traditional intra-thread reuse distance prediction based LLC replacement useless for mst with helper thread. Fortunately, as we will see, HT-MT inter-thread reuse distance prediction based LLC replacement can be useful for mst with helper thread.

3.3.2 A Simple Taxonomy of Helper Thread LLC Requests Based on Intra-Application Reuse Distances

Based on the notions of HT-MT inter-thread reuse distance ($ITRD_{HT-MT}$) and main thread intra-thread reuse distance (RD_{MT}), we can construct a pollution aware taxonomy of helper thread LLC requests where helper thread LLC requests are classified into three types: good, bad and ugly. Let's assume when a helper thread LLC request referencing data H evicts an LLC line containing the victim data V which is brought by main thread, and afterwards the LLC line containing H is evicted by a main thread LLC request with data M. Therefore, there are two LLC replacement involved: first H evicts V and then M evicts H. We use $RD_{MT}(v)$, $ITRD_{HT-MT}(h)$ and $RD_{MT}(m)$ to denote the number of distinct data elements that have been referenced between the time when H evicts V and the time V, H and M is accessed again, respectively. The greater the reuse distance of the data, the farther the data will be referenced again in time. An LLC 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

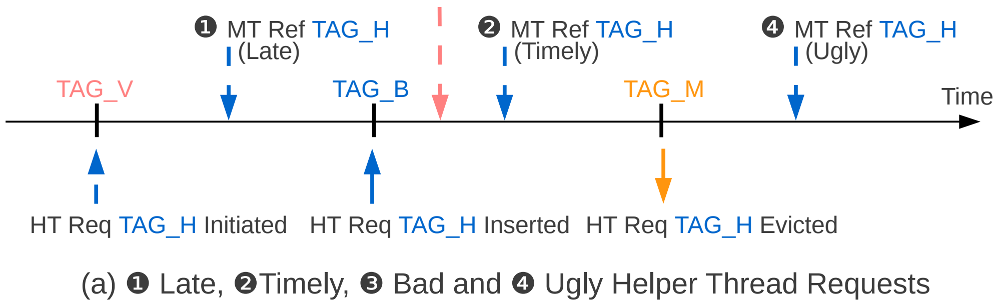
1. if $ITRD_{HT-MT}(H) < RD_{MT}(V) < RD_{MT}(M)$, then the helper thread LLC request is considered as *good* because it evicts the data that has larger reuse distance than its own. Its data is hit by the main thread before evicted. It has positive impact on the main thread performance since it reduce one main thread miss;

2. if $RD_{MT}(V) < ITRD_{HT-MT}(H) < RD_{MT}(M)$, then the helper thread LLC request is considered as *bad* because it evicts the data that has smaller reuse distance than its own. It displaces an LLC 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 $ITRD_{HT-MT}(H)$, then the helper thread LLC request is considered *ugly* because both its data and its victim data have larger reuse distances than the data H. 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 main thread.

As depicted in Fig.3, We can see that, good and bad helper thread LLC requests are caused by the optimal and non-optimal LLC replacement in the presence of helper threaded data prefetching on CMPs, respectively. We can conclude that

1. Low $ITRD_{HT-MT}$ is an indicator of good performance of the helper threaded prefetching scheme;
2. Medium $ITRD_{HT-MT}$ signals those potentially late helper thread prefetches that, depend on the LLC replacement policy, may be useful, partially useful or totally useless for main thread performance;
3. High $ITRD_{HT-MT}$ implies helper thread and main thread is not synchronized well or the amount of (computational and memory access) work is not distributed and balanced well between the main thread and helper thread.



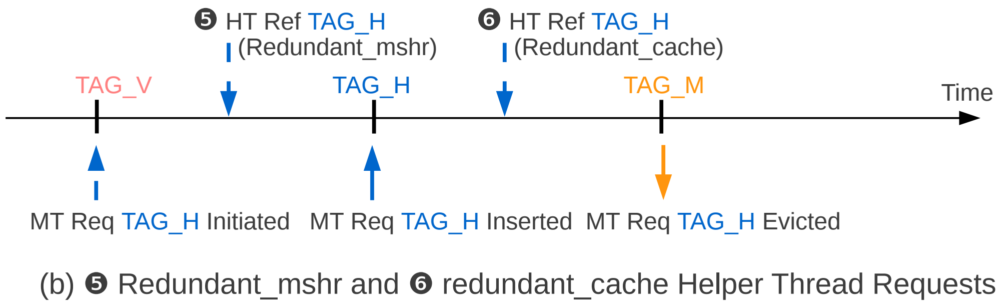


Fig. 3: Taxonomy of Helper Thread LLC Requests

3.3.3 Add Lateness to the Taxonomy of Helper Thread LLC Requests

As noted in the previous section, good helper thread LLC 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 LLC request references the data. Late helper thread requests only partially hide the LLC miss latency in main thread requests, but they are good indicators for potential performance improvement of the helper threaded prefetching scheme because late helper thread requests may be converted to timely helper thread requests by adjusting the aggressiveness parameters of the helper threaded prefetching scheme, i.e., lookahead and stride.

As we can observe that, if the helper thread request is too late, then when the request arrives at LLC, it find its referenced data is already present in either LLC or LLC MSHRs. Therefore, we can add *redundant_ mshr helper thread request* and *redundant_ cache helper thread request* to the taxonomy, as shown in Fig.3.

3.4 Mechanisms and Algorithms for Implementing the Taxonomy

3.4.1 LLC request and replacement event tracking

To monitor the request and replacement activities in LLC, we need to consider the event when the LLC 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 LLC line, the requester memory hierarchy access, line found in the LLC, a boolean value indicating whether the request hits in the LLC, and a boolean value indicating whether the request needs to evict some LLC line. This event is similar to one used in [5].

3.4.2 Helper Thread LLC request state tracking

In order to track the helper thread request states in the LLC, we need to add one field to each LLC line to indicate whether the line is brought by the main thread or the helper thread or otherwise invalid, as depicted in Fig.1.

3.4.3 Helper Thread LLC 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 LLC lines that are evicted by helper thread requests. As shown in Fig.1, the HTRVC has the same structure of the LLC, but there is no direct mapping between LLC 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 LLC. HTRVC has the sole purpose of profiling, so it has no impact on performance.

3.4.4 Detecting Late Helper Thread LLC Requests

Lastly, in order to measure late helper thread requests, we need to identify the event when a main thread request hits to an LLC 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 LLC Miss Status Holding Register (MSHR). MSHR is a hardware structure that keeps track of all in-flight memory requests. An helper thread LLC request is late if a main thread LLC request for the same address is generated while the helper thread LLC request is in the LLC MSHR waiting for main memory.

3.4.5 Tracking Helper Thread LLC Requests and Victims

There are two following invariants that should be maintained between LLC and HTRVC per set

1. Number of helper thread Lines in the LLC Set = Number of Victim Entries in the HTRVC Set;
2. Number of Victim Entries in HTRVC Set + Number of Valid main thread LLC Lines in Set $\leq$ LLC Set Associativity.

   Helper thread lines refer to the LLC 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 LLC Set+ Number of Valid main thread LLC Lines in Set $\leq$ LLC Associativity. Actions should be taken in LLC and HTRVC when filling an LLC line and servicing an incoming LLC request either from the main thread or the helper thread.

Actions taken on Inserting an LLC Line When filling an LLC 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 LLC line which is previously brought by a main thread request. In this case, eviction is needed to make room for the incoming helper thread request;

3. An helper thread request evicts an LLC line which is previously brought by a helper thread request. In this case, eviction is needed to make room for the incoming helper thread request;

4. An main thread request evicts an LLC line which is previously brought by a helper thread request. In this case, eviction is needed to make room for the incoming main thread request.

Specific actions taken on the above four cases are listed in Fig.4, where hitInLLC indicates whether the request hits in LLC 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 LLC line found is brought by helper thread or not.

```
//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.removeLRU(set);
}
```

Fig. 4: Actions Taken When Inserting an LLC Line

Actions taken on Servicing an Incoming LLC Request When servicing an incoming LLC request, either hit or miss, we should consider four cases:

1. LLC miss and victim hit, which indicates a bad helper thread request. This happens when helper thread request evicts useful data;

2. Helper thread LLC 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 LLC hit and victim hit. This happens when useful data is evicted and brought back in by helper thread request;

4. Main thread LLC 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.5, where mainThreadHit indicates whether the request comes from main thread and hits in the LLC; helperThreadHit indicates whether the request comes from helper thread and hits in the LLC; and victimHit indicates whether the request comes from main thread and hits in the HTRVC.

```
//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);
}
```

Fig. 5: Actions Taken When Servicing an LLC Request

4 Performance Results

4.1 Basic Results

Tab.2 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 16, respectively. In Tab.2 (a), we can see that when the L2 size is greater than 128 KB, there is no difference in execution time, implying the working set of mst “2048 1” with helper threaded prefetching switched off is near 128 KB. L2 sizes larger than 128 KB contribute little to the overall performance. Tab.2 (b) shows that the execution time is insensitive to the L2 associativity when the latter is greater than 4. Trivial speedups are observed in these basic experiments of mst benchmark baseline versions, which is however in contrast to the results observed in mst helper threaded prefetching versions as we will discuss shortly after.

L2 Size	L2 Assoc	Total Cycles	Speedup	Main Thread Hit	Main Thread Miss
96 KB	8	4001166684	1.0000	14562	12695429
128 KB	8	4001145025	1.0000	14560	12695141
256 KB	8	4001145025	1.0000	14560	12695141
512 KB	8	4001145025	1.0000	14560	12695141
1 MB	8	4001145025	1.0000	14560	12695141
2 MB	8	4001145025	1.0000	14560	12695141

(a) Impact of L2 Size

L2 Size	L2 Assoc	Total Cycles	Speedup	Main Thread Hit	Main Thread Miss
96 KB	2	4005695559	1.0000	17289	12764949
96 KB	4	4001435947	1.0011	14561	12707953
96 KB	8	4001166684	1.0011	14562	12695429
96 KB	16	4001147526	1.0011	14561	12695163
96 KB	32	4001145232	1.0011	14560	12695141

(b) Impact of L2 Associativity

Tab. 2: mst Baseline Performance

L2 Size	L2 Assoc	Lookahead	Stride	Total Cycles	Speedup	Main Thread Hit	Main Thread Miss	Helper Thread Hit	Helper Thread Miss	Redundant MSHR	Redundant Cache	Timely	Late	Bad	Ugly
20	10	96 KB	8	3323736491	1.0000	4105024	8920817	198931	8329835	11237	187694	3870340	6640	12663	862381
20	10	128 KB	8	3276685422	1.0144	4213878	8742330	191836	8220627	340	191496	3983129	4482	47	694
20	10	256 KB	8	3276637422	1.0144	4213991	8740350	191867	8220537	327	191540	3983311	4436	40	536
20	10	512 KB	8	3276622703	1.0144	4214152	8739942	191919	8220284	359	191560	3983445	4413	0	620
20	10	1 MB	8	3276622703	1.0144	4214152	8739942	191919	8220284	359	191560	3983445	4413	0	1193
20	10	2 MB	8	3276622703	1.0144	4214152	8739942	191919	8220284	359	191560	3983445	4413	0	1591

(a) Impact of L2 Size

L2 Size	L2 Assoc	Lookahead	Stride	Total Cycles	Speedup	Main Thread Hit	Main Thread Miss	Helper Thread Hit	Helper Thread Miss	Redundant MSHR	Redundant Cache	Timely	Late	Bad	Ugly
20	10	96 KB	2	3375728053	1.0000	3808745	9339129	214714	8570405	28156	186558	3570356	6552	16219	1939057
20	10	96 KB	4	3353819652	1.0065	3952682	9111809	206181	8457461	20359	185822	3717897	4777	21284	1351399
20	10	96 KB	8	3323736491	1.0156	4105024	8920817	198931	8329835	11237	187694	3870340	6640	12663	862381
20	10	96 KB	16	3277232422	1.0301	4215025	8749887	192352	8225356	443	191909	3984572	4566	30	3500
20	10	96 KB	32	3276663422	1.0302	4214238	8740076	191949	8220386	350	191599	3983459	4459	4	63

(b) Impact of L2 Associativity

Tab. 3: mst Helper Threaded Prefetching Performance

Tab.3 shows the overall performance of mst benchmark helper threaded prefetching version, in which L2 size is varied from 96 KB to 2 MB and L2 associativity is varied from 2 to 16, respectively. In Tab.3 (a), we can see that when the L2 size is greater than 512 KB, there is no difference in execution time, implying the working set of mst "2048 1" with helper threaded prefetching switched on is near 512 KB. L2 sizes larger than 512 KB contribute little to the overall performance. Tab.3 (b) shows that the execution time is reduced slightly as the L2 associativity is increased from 2 to 32. The presence of helper threaded prefetching increases the working set of mst "2048 1" from 128 KB in the baseline version to 512 KB, and also incur much more conflict misses. The number of bad prefetches is reduced from 3M to zero when L2 size is increased from 96 KB to 512 KB (the working set limit). The increase of L2 associativity can also reduce the number of bad prefetches as well. We can see that bad prefetches are in the forms of capacity and conflict misses in this scenario. About 1%-3% reductions of execution time are achieved in these basic experiments of mst benchmark helper threaded prefetching versions. A maximum 22% reduction of execution is achieved between baseline and helper threaded versions of mst in these basic experiments, which shows our implemented helper threaded prefetching scheme is non-trivially effective for the mst benchmark.

4.2 Impact of L2 Size

Fig.6 shows the normalized execution times in cycles and helper threaded L2 request breakdown for the mst helper threaded prefetching version when the L2 size is increased from 96 KB to 2 MB. For clarity, execution times are normalized to the first case in the figure where L2 size is 96 KB, where the 128 KB working set limit of mst "2048" helper threaded prefetching version is reiterated. A few ugly helper thread L2 requests are observed when the L2 size is as small as 96 KB, which results from where the helper thread first prefetch some potentially useful data into the L2 cache, but due to the limited capacity of the L2 cache, these data are evicted too early from the L2 cache before the main thread uses them.

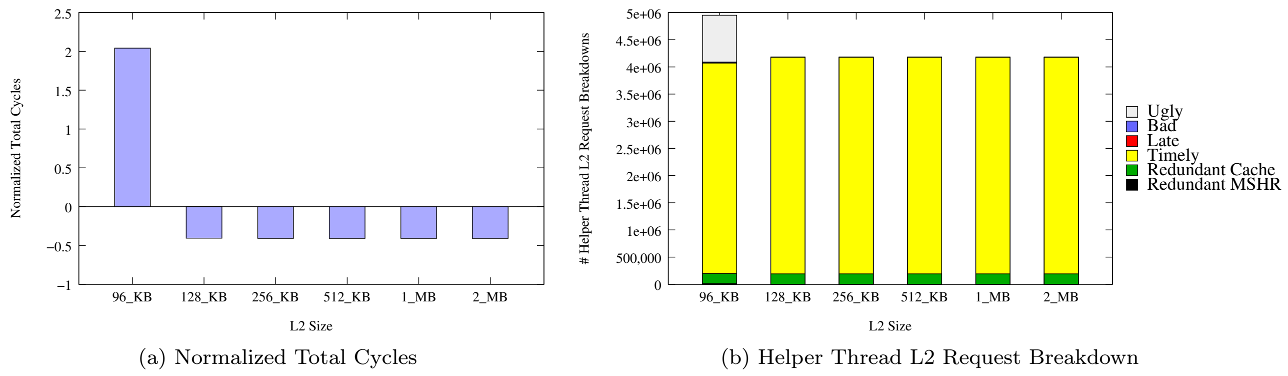


Fig. 6: Impact of L2 Size on mst Helper Threaded Prefetching Performance (lookahead=20, stride=10)

4.3 Impact of L2 Associativity

Fig.7 shows the normalized execution times in cycles and helper threaded L2 request breakdown for the mst helper threaded prefetching version when the L2 associativity is varied from 2 to 16. For clarity, execution times are normalized to the first case in the figure where L2 associativity is 2. A few ugly helper thread L2 requests are observed when the L2 associativity is smaller than 16, which results from where the helper thread first prefetch some potentially useful data into the L2 cache, but due to the limited associativity of the L2 cache, these data are evicted too early from the L2 cache before the main thread uses them.

One important point to notice is that, a few redundant hit-to-cache helper thread L2 requests are constantly observed among the experiments shown in Fig.6 and Fig. 7, which implies the helper thread is sometimes initiated too late to prefetch the data, as the data is already put in the L2 cache by the main thread, thus leading to the waste of cache resource and on-chip interconnect bandwidth. This deficiency can only be improved by fine-tuning the helper threaded prefetching program code, which is out of the scope of this work.

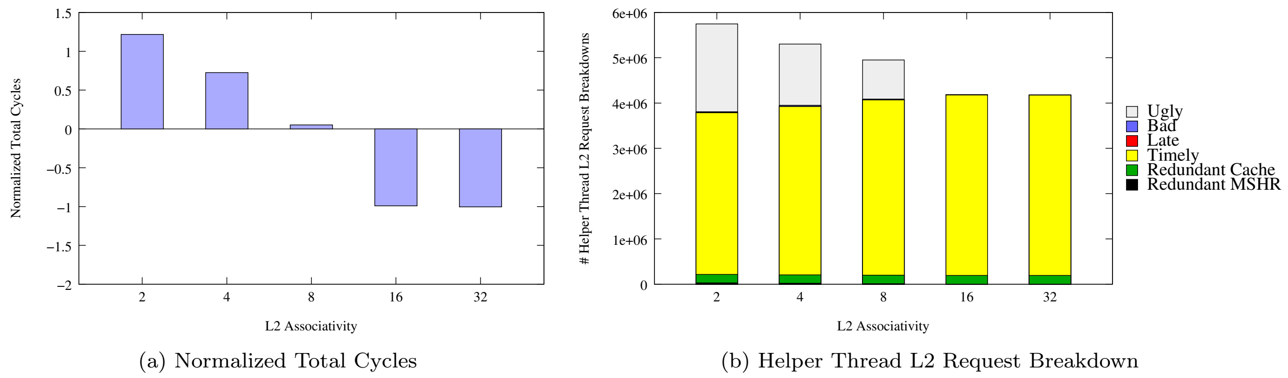


Fig. 7: Impact of L2 Associativity on mst Helper Threaded Prefetching Performance (lookahead=20, stride=10)

4.4 Impact of Prefetching Lookahead

Below we see how the two parameters lookahead and stride of the helper threaded prefetching scheme affects the helper thread L2 request breakdown and thus the overall performance. Fig.8 shows the normalized execution times in cycles and helper threaded L2 request breakdown for the mst helper threaded prefetching version when the lookahead is varied from 10 to 480. For clarity, execution times are normalized to the first case in the figure where the lookahead is 10. We can see that when the stride is less than 40, the execution time increases monotonically as the lookahead is increased from 10 to 640. But this is not the case when the stride is greater than 40, where the maximum reduction in execution time is achieved in the middle value of the lookahead.

We must note that when the lookahead is small and the stride is large, there is a large portion of late helper thread L2 requests which can be explained intuitively as the helper thread skips too little in the prelude and prefetches too much in the stable state to keep running ahead of the main thread to fetch the data. In the other hand, when the lookahead is increased from 10 to 480 and the stride is large, the number of timely helper thread L2 requests diminishes, which implies the helper thread does too little prefetching work to provide tangible overall performance improvement.

4.5 Impact of Prefetching Stride

Fig.9 shows the normalized execution times in cycles and helper threaded L2 request breakdown for the mst helper threaded prefetching version when the stride is varied from 10 to 480. For clarity, execution times are normalized to the first case in the figure where the stride is 10.

We can see the relationship between the values of stride and the overall performance varies under different values of lookahead:

1. When the lookahead is less than 40, the execution time increases monotonically as the stride is increased from 10 to 640. It can be obviously interpreted as the harmful impacts of the increasing prohibitively large portion of late prefetches shown in red in the figure;

2. When the lookahead are 40 and 80, the greatest speedups are achieved when the stride is in the middle values of 80 and 320, respectively. It reflects the ongoing battle between the positive and negative performance gains from timely and late helper thread prefetches, respectively;

3. When the lookahead is greater than 160, the number of late helper thread prefetches diminishes to zero. This can be explained as the large value of lookahead in the prelude prevents the helper thread from lagging behind the main thread in the stable state. However, in this case, the execution time increases monotonically again as the value of lookahead becomes greater.

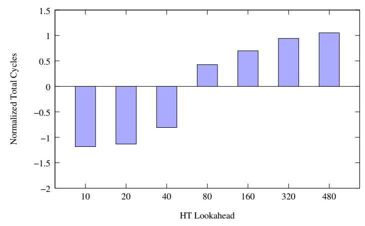
4.6 Summary

From the above discussion on experimental results, we can conclude that, (1) too much late prefetches do great harm to the performance improvement of helper threaded prefetching; (2) the helper threaded prefetching 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 threaded prefetching on CMPs; (3) there is little bad prefetches observed in our experiments on mst “2048 1”, but it is expected to be prevalent in larger input set of the mst benchmark and other memory-intensive workloads.

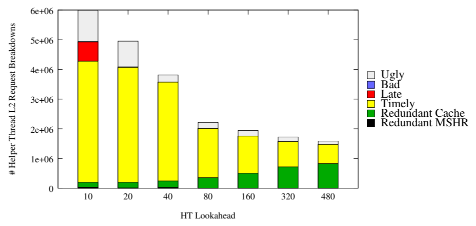
5 Related Work

5.1 Prefetch Taxonomies

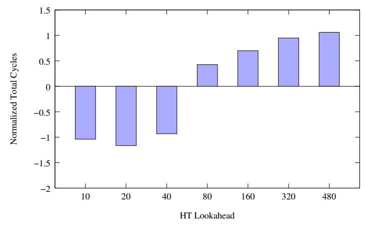
Even though helper threaded data prefetching mechanisms have been studied for a long time, LLC interference and taxonomy of helper thread LLC requests have not been studied before. Here we briefly describe previous work in



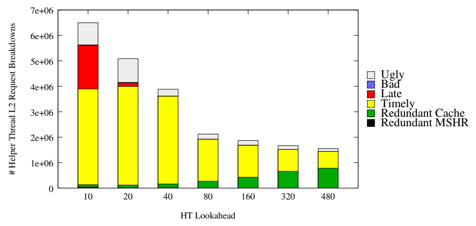
(1a) Normalized Total Cycles (stride=10)



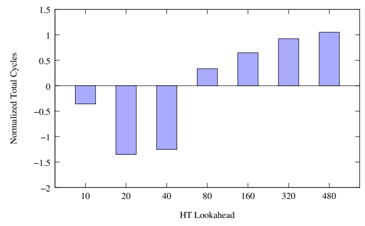
(1b) Helper Thread L2 Request Breakdown (stride=10)



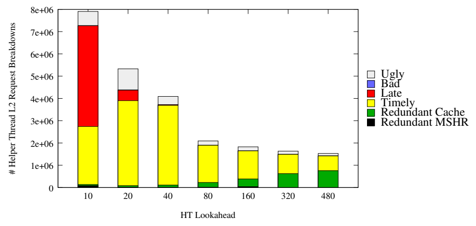
(2a) Normalized Total Cycles (stride=20)



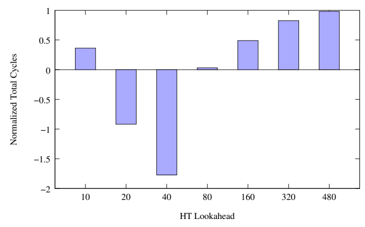
(2b) Helper Thread L2 Request Breakdown (stride=20)



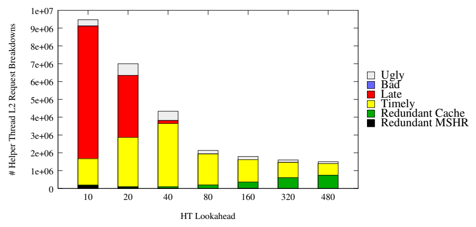
(3a) Normalized Total Cycles (stride=40)



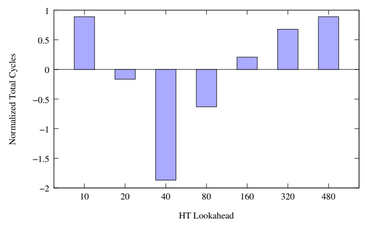
(3b) Helper Thread L2 Request Breakdown (stride=40)



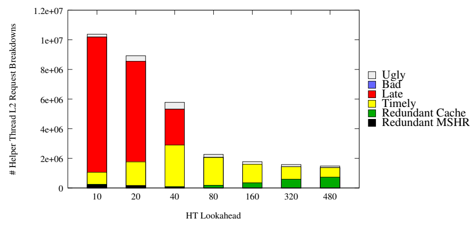
(4a) Normalized Total Cycles (stride=80)



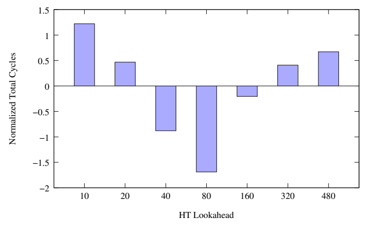
(4b) Helper Thread L2 Request Breakdown (stride=80)



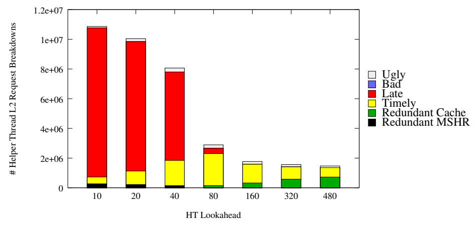
(5a) Normalized Total Cycles (stride=160)



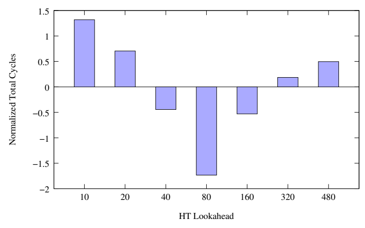
(5b) Helper Thread L2 Request Breakdown (stride=160)



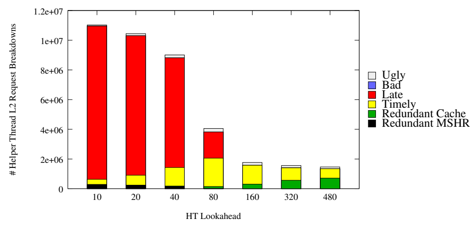
(6a) Normalized Total Cycles (stride=320)



(6b) Helper Thread L2 Request Breakdown (stride=320)

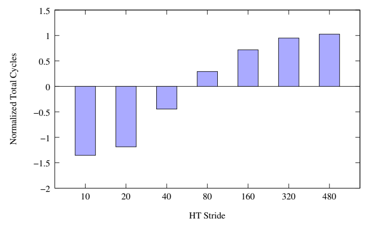


(7a) Normalized Total Cycles (stride=480)

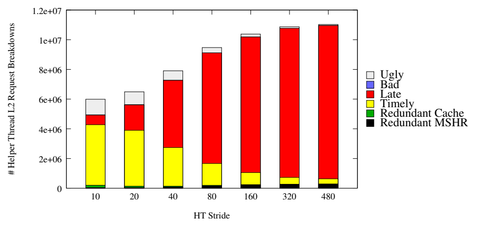


(7b) Helper Thread L2 Request Breakdown (stride=480)

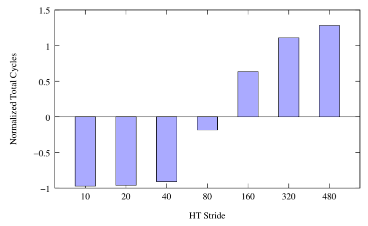
Fig. 8: Impact of Prefetching Lookahead on mst Helper Threaded Prefetching Performance



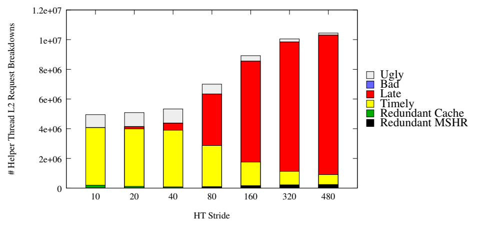
(1a) Normalized Total Cycles (lookahead=10)



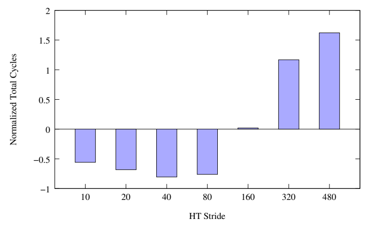
(1b) Helper Thread L2 Request Breakdown (lookahead=10)



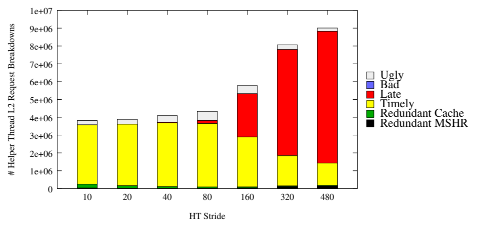
(2a) Normalized Total Cycles (lookahead=20)



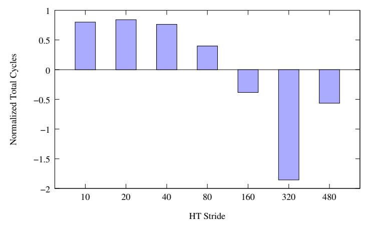
(2b) Helper Thread L2 Request Breakdown (lookahead=20)



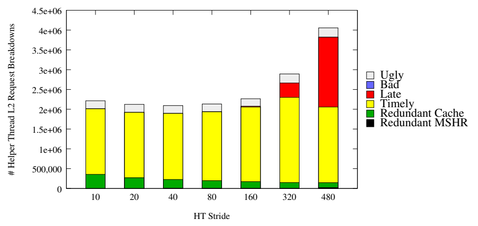
(3a) Normalized Total Cycles (lookahead=40)



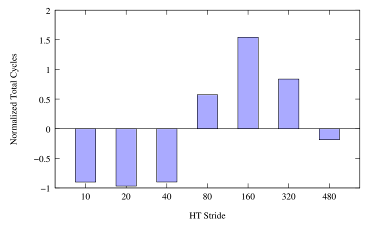
(3b) Helper Thread L2 Request Breakdown (lookahead=40)



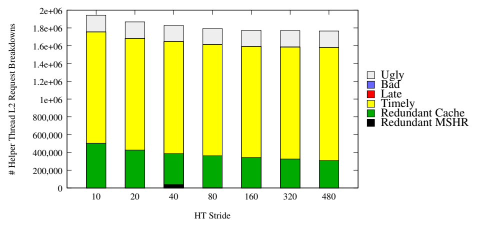
(4a) Normalized Total Cycles (lookahead=80)



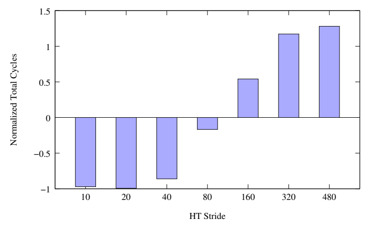
(4b) Helper Thread L2 Request Breakdown (lookahead=80)



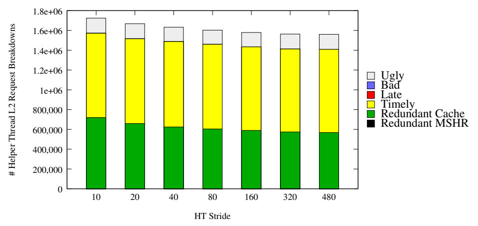
(5a) Normalized Total Cycles (lookahead=160)



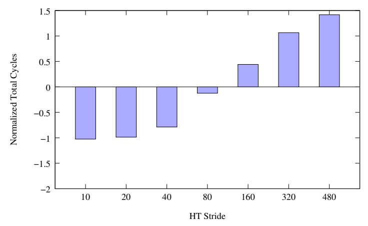
(5b) Helper Thread L2 Request Breakdown (lookahead=160)



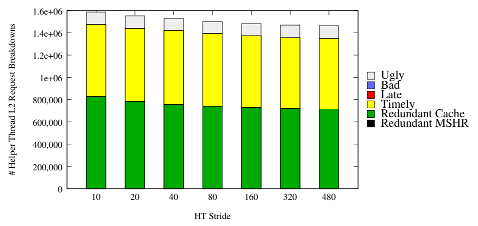
(6a) Normalized Total Cycles (lookahead=320)



(6b) Helper Thread L2 Request Breakdown (lookahead=320)



(7a) Normalized Total Cycles (lookahead=480)



(7b) Helper Thread L2 Request Breakdown (lookahead=480)

Fig. 9: Impact of Prefetching Stride on mst Helper Threaded Prefetching Performance

the context of hardware prefetching. Our taxonomy of helper thread LLC requests is mostly similar to the work in [5], which presented a taxonomy of hardware prefetches based on the idea of shared cache pollution in the hardware based data prefetching for shared L2 CMP. A hardware structure called the *Evict Table* (ET) is attached to the LLC 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 hardware prefetches. [6] took the bandwidth consumption into account, and developed a multiprocessor prefetch traffic and miss taxonomy that builds on an existing uniprocessor prefetch taxonomy.

5.2 Prefetch Aware Cache Content Management

Some previous works have studied the problem of cache content management in the presence of data prefetching, which are in the forms of software, hardware and hybrid. [7] 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 but different from our proposed HTRVC cache structure whose sole function is for profiling helper thread L2 requests' victims. [8] characterized the performance of state-of-the-art LLC management policies in the presence and absence of hardware prefetching. Prefetch-Aware Cache Management (PACMan) is proposed to dynamically estimates and mitigates the degree of prefetch-induced cache interference by modifying the cache insertion and hit promotion policies to treat demand and prefetch requests differently. [9] proposed a low-cost feedback directed mechanism for hardware prefetching. The mechanism can be applied to any hardware prefetchers such as sequential prefetchers, stream-based prefetchers, GHB based prefetchers and PC-based stride prefetchers.

6 Conclusion

This paper first discusses the intra-application reuse distances in the helper threaded prefetching on shared cache CMPs and its implications on helper thread performance. Based on this, we then proposes the intra-application interference aware taxonomy of helper thread LLC requests and its implementation details on a cycle-accurate CMP architectural simulator. Detailed experimental results of the memory intensive benchmark mst show that: (1) too much late prefetches do great harm to the performance improvement of helper threaded prefetching; and (2) the helper threading parameters of lookahead and stride should and can be dynamically changed to balance the portions of timely and late prefetches, thus leading to the maximal exploitation of helper threaded prefetching in improving the performance of irregular memory access workloads on shared cache CMPs. It can be part of our future work.

Acknowledgments

This work was supported by the National Natural Science Foundation of China under the contract No. 61070029.

References

[1] D. Kim, S.-W. Liao, P. H. Wang, J. d. Cuvillo, X. Tian, X. Zou, H. W. 0003, D. Yeung, M. Girkar, and J. P. Shen, "Physical experimentation with prefetching helper threads on intel's hyper-threaded processors," in *CGO*. IEEE Computer Society, 2004, pp. 27–38.

[2] 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.

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

[4] 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.

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

[6] N. D. E. Jerger, E. L. Hill, and M. H. Lipasti, “Friendly fire: understanding the effects of multiprocessor prefetches,” in *ISPASS*. IEEE Computer Society, 2006, pp. 177–188.

[7] 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.

[8] C.-J. Wu, A. Jaleel, M. Martonosi, S. C. S. J. , and J. S. Emer, “Pacman: prefetch-aware cache management for high performance caching,” in *MICRO*, ser. 44rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2011, 3-7 December 2011, Porto Alegre, Brazil, C. Galuzzi, L. Carro, A. Moshovos, and M. Prvulovic, Eds. ACM, 2011, pp. 442–453.

[9] 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.