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 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.

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.

(to be expanded)

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. 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.

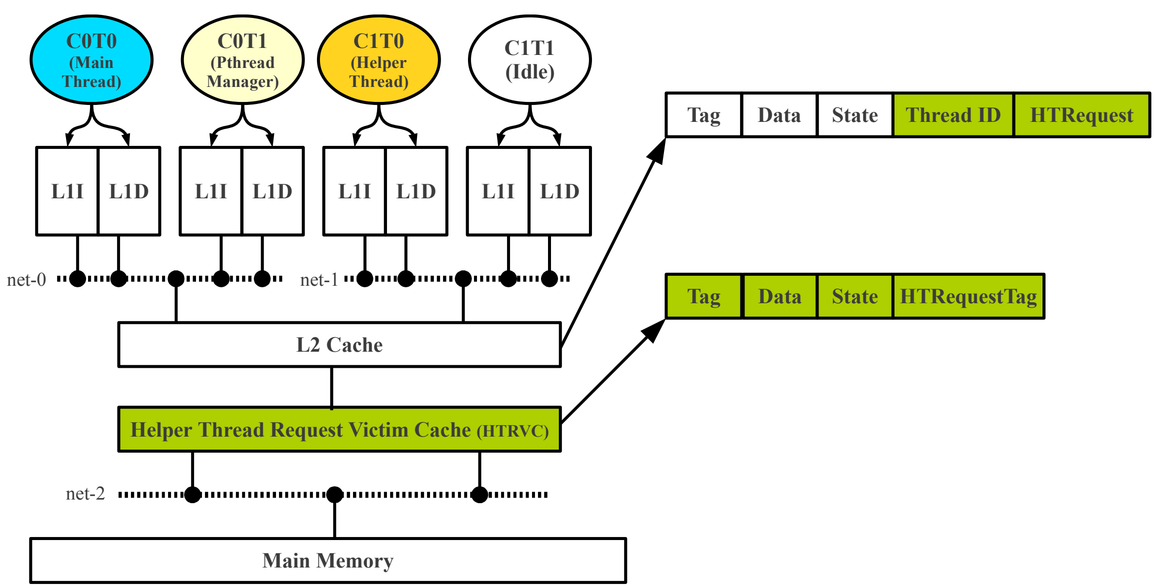


Fig. 1: Target CMP Architecture

<table>
<tr><td rowspan="5">Pipeline</td><td colspan="5">4-wide superscalar out-of-order core; 2 cores, 2 threads per core</td></tr>
<tr><td colspan="5">Physical register file capacity: 128</td></tr>
<tr><td colspan="5">Decode buffer capacity: 96</td></tr>
<tr><td colspan="5">Reorder buffer capacity: 96</td></tr>
<tr><td colspan="5">Load store queue capacity: 48</td></tr>
<tr><td>Branch Predictors</td><td colspan="5">Perfect branch predictor</td></tr>
<tr><td rowspan="12">Execution Units</td><td>Name</td><td>Count</td><td>Operation Lat.</td><td colspan="2">Issue Lat.</td></tr>
<tr><td>Int ALU</td><td>8</td><td>2</td><td colspan="2">1</td></tr>
<tr><td>Int Multiply</td><td rowspan="2">2</td><td>3</td><td colspan="2">1</td></tr>
<tr><td>Int Division</td><td>20</td><td colspan="2">19</td></tr>
<tr><td>Fp Add</td><td rowspan="3">8</td><td>4</td><td colspan="2">1</td></tr>
<tr><td>Fp Compare</td><td>4</td><td colspan="2">1</td></tr>
<tr><td>Fp Convert</td><td>4</td><td colspan="2">1</td></tr>
<tr><td>Fp Multiply</td><td rowspan="3">2</td><td>8</td><td colspan="2">1</td></tr>
<tr><td>Fp Division</td><td>40</td><td colspan="2">20</td></tr>
<tr><td>Fp Square Root</td><td>80</td><td colspan="2">40</td></tr>
<tr><td>Read Port</td><td rowspan="2">4</td><td>1</td><td colspan="2">1</td></tr>
<tr><td>Write Port</td><td>1</td><td colspan="2">1</td></tr>
<tr><td rowspan="4">Cache Geometries</td><td>Name</td><td>Size</td><td>Associativity</td><td>Line Size</td><td>Hit Lat.</td></tr>
<tr><td>ICache</td><td>32KB</td><td>4</td><td>64B</td><td>1</td></tr>
<tr><td>DCache</td><td>32KB</td><td>4</td><td>64B</td><td>1</td></tr>
<tr><td>L2 Cache</td><td>96KB</td><td>8</td><td>64B</td><td>10</td></tr>
<tr><td>Interconnect</td><td colspan="5">Switch based P2P topology, 32B link width</td></tr>
<tr><td>Main Memory</td><td colspan="5">4GB, 200-cycle fixed latency</td></tr>
</table>

Tab. 1: Baseline Hardware Configurations

2.2 Simulation Framework

We use an in-house CMP architectural simulator named Archimulator in our experiments mentioned in this work. Archimulator is a flexible execution-driven architectural simulator written in Java and running on Linux. It provides fast forward functional simulation and cycle-accurate application-only simulation of MIPS II executables on multi-core architectures consisting of out-of-order super-scalar cores and configurable memory hierarchy of directory-based MESI coherence. It has basic support of simulating Pthreads based multithreaded workloads.

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

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-traveral 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 workflow 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 *steady 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 steady state is called stride.

In a traditional helper threaded prefetching scheme configuration, the values of *lookahead* and *stride* are hard coded. In our following proposed feedback directed mechanism, the processor changes the value of lookahead and stride using some temporary register to adjust the aggressiveness of the helper threaded prefetching scheme.

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 [1]. 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.

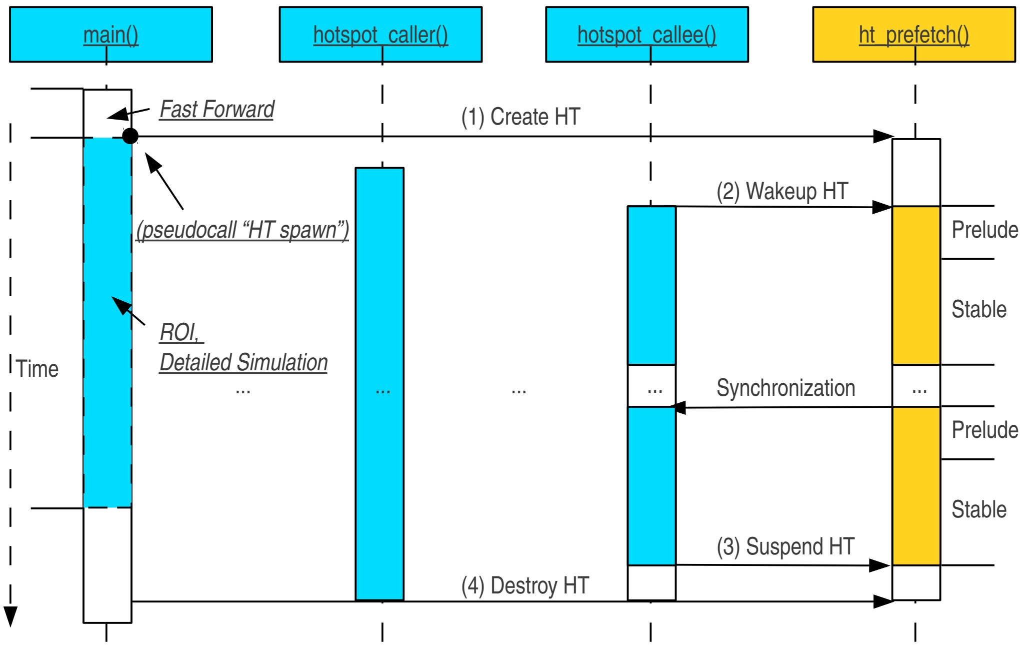


Fig. 2: The Scheme of Helper Threaded Data Prefetching on CMPs

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

$$Helper\ Threaded\ Prefetch\ Coverage = \frac{Number\ of\ Useful\ Helper\ Thread\ LLC\ Requests}{Number\ of\ Main\ Thread\ LLC\ Requests\ WIthout\ No\ Prefetch}$$

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 an 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 accomodate 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 followup 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 an 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 an 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 posive 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.

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.

3.3.3 The Refined 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 an 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. The final refined taxonomy of helper thread LLC requests is summarized in Fig.3. *(to be explained in detail)*

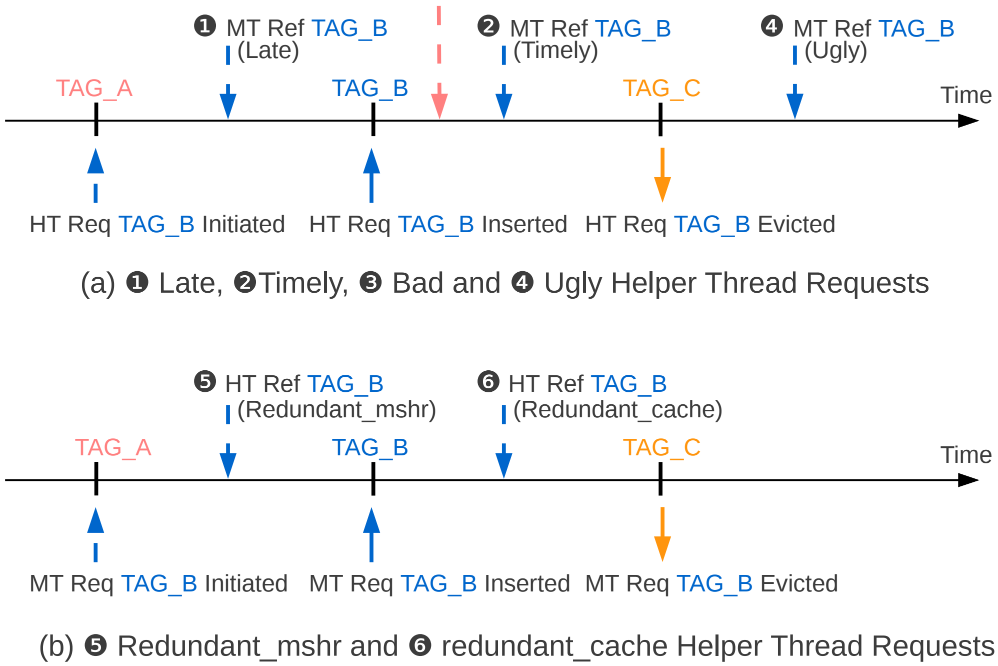


Fig. 3: Taxonomy of Helper Thread LLC Requests

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 [2].

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 an main thread request hits to an LLC line which is being brought by an inflight 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 an 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 invariants that should be maintained

1. # of helper thread Lines in the LLC Set = # of Victim Entries in the HTRVC Set;

2. # of Victim Entries in HTRVC Set + # of Valid main thread LLC Lines in Set $\leq$ LLC 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: # of helper thread Lines in the LLC Set+ # of Valid main thread LLC Lines in Set $\leq$ LLC Set Associativity. Actions should be taken in LLC and HTRVC when filling an LLC line or servicing an incoming LLC request.

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 an 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 an 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 an helper thread request. In this case, eviction is needed to make room for the incoming main thread request.

Specific actions taken on the above five cases are listed in Fig.4, where hitInLLC: whether the request hits in LLC or not, requesterIsHT: whether the request comes from helper thread or not, hasEviction: whether the request needs to evict some data, lineFoundIsHT: whether the LLC line found is brought by helper thread or not, and htRequestFound(): whether there is at least one line in the LLC set that is brought by helper thread.

```
//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 five cases are listed in Fig.5, where mtHit: whether the request comes from main thread and hits in the LLC, htHit: whether the request comes from helper thread and hits in the LLC, and vtHit: 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

3.5 Results Overview *(to be expanded)*

3.6 Impact of L2 Size *(to be expanded)*

3.7 Impact of L2 Associativity *(to be expanded)*

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

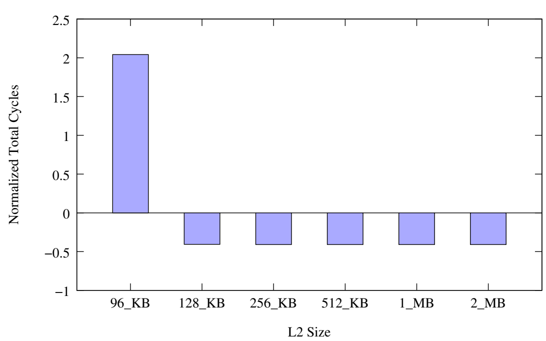
L2 Size	L2 Assoc	Lookahead	Stride	Total Cycles	Speedup	Main Thread Hit	Main Thread Miss	Helper Thread Hit	Helper Thread Miss	Redundant MSHR	Redundant Cache	Timely	Late	Bad	Ugly
20	10	96 KB	8	3323736491	1.0000	4105024	8920817	198931	8329835	11237	187694	3870340	6640	12663	862381
20	10	128 KB	8	3276685422	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

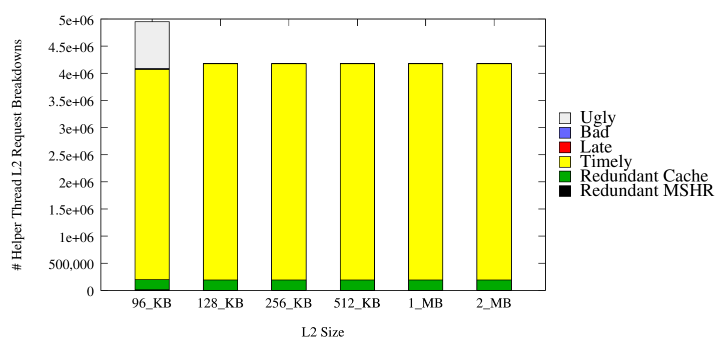
L2 Size	L2 Assoc	Lookahead	Stride	Total Cycles	Speedup	Main Thread Hit	Main Thread Miss	Helper Thread Hit	Helper Thread Miss	Redundant MSHR	Redundant Cache	Timely	Late	Bad	Ugly
20	10	96 KB	2	3375728053	1.0000	3808745	9339129	214714	8570405	28156	186558	3570356	6552	16219	1939057
20	10	96 KB	4	3353819652	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

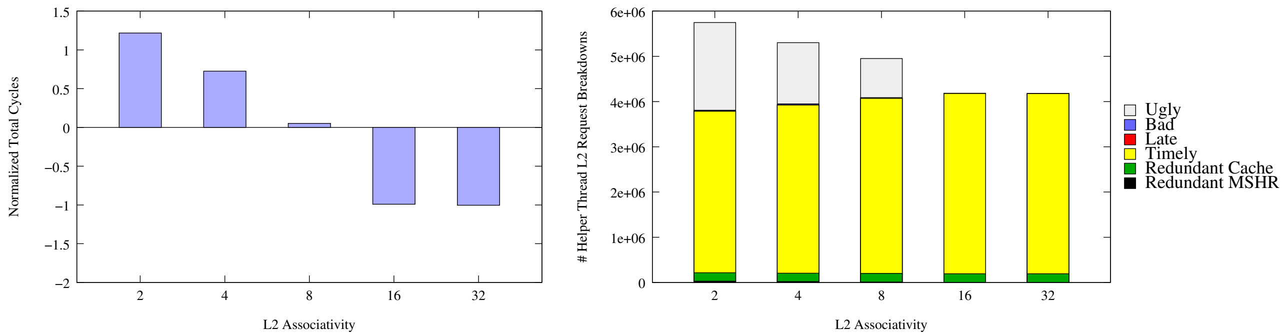


(a) Normalized Total Cycles



(b) Helper Thread L2 Request Breakdown

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



(a) Normalized Total Cycles (b) Helper Thread L2 Request Breakdown

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

3.8 Impact of Prefetching Lookahead *(to be expanded)*

3.9 Impact of Prefetching Stride *(to be expanded)*

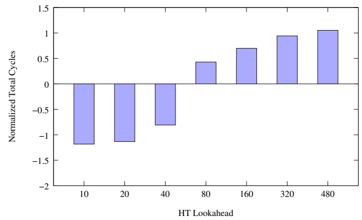
4 Related Work *(to be expanded)*

4.1 Prefetch Taxonomies

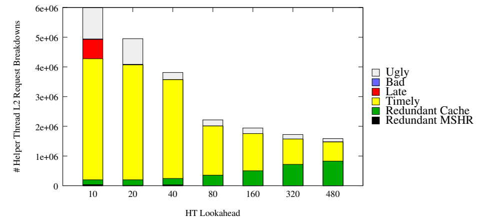
Even though helper threaded data prefetching mechanisms have been studied for a long time, LLC interferece and taxonomy of helper thread LLC requests have not been studied before. Here we briefly describe previous work in the context of hardware prefething. Our taxonomy of helper thread LLC requests is mostly similar to the work in [2], which presents 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. [3] developed a multiprocessor prefetch traffic and miss taxonomy that builds on existing uniprocessor taxonomy.

4.2 Prefetch Aware Cache Content Management

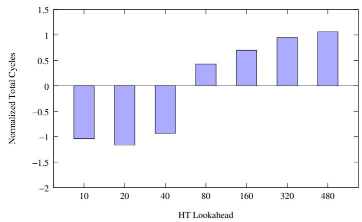
[4] characterizes the performance of state-of-the-art LLC management policies in the presence and absence of hardware prefetching. Prefetch-Aware Cache Management (PACMan) is proposed to dynamically estimates and mitigates the degree of prefetch-induced cache interference by modifying the cache insertion and hit promotion policies to treat domand and prefetch requests differently. [5] proposes 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.



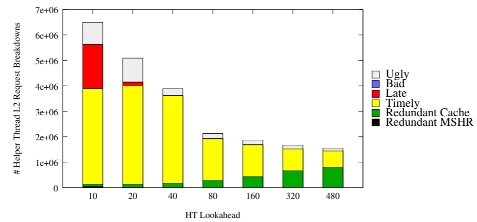
(1a) Normalized Total Cycles (stride=10)



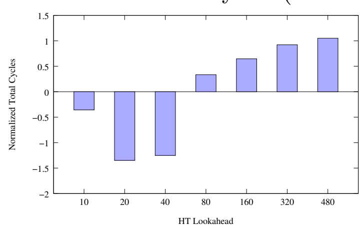
(1b) Helper Thread L2 Request Breakdown (stride=10)



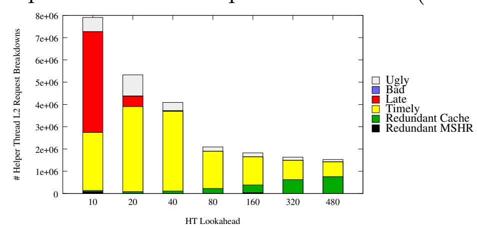
(2a) Normalized Total Cycles (stride=20)



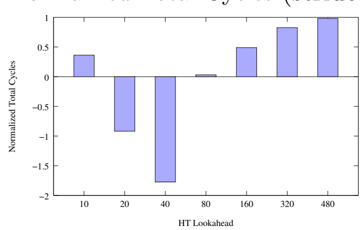
(2b) Helper Thread L2 Request Breakdown (stride=20)



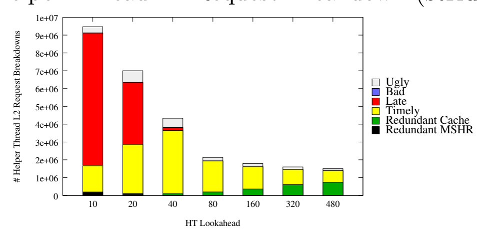
(3a) Normalized Total Cycles (stride=40)



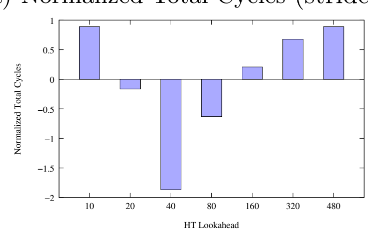
(3b) Helper Thread L2 Request Breakdown (stride=40)



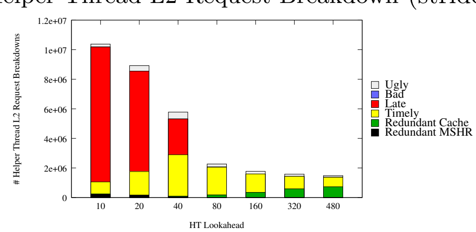
(4a) Normalized Total Cycles (stride=80)



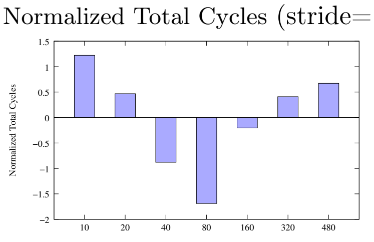
(4b) Helper Thread L2 Request Breakdown (stride=80)



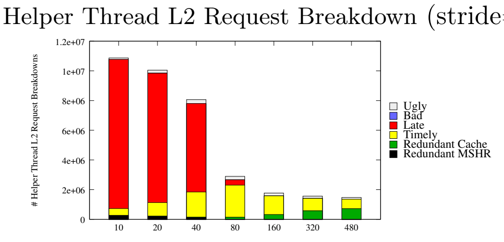
(5a) Normalized Total Cycles (stride=160)



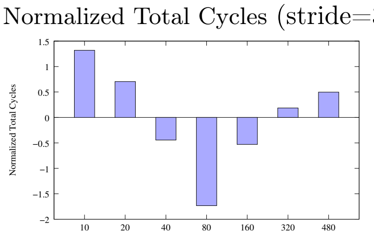
(5b) Helper Thread L2 Request Breakdown (stride=160)



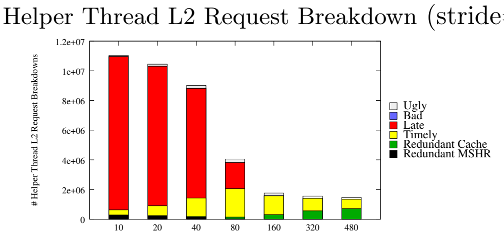
(6a) Normalized Total Cycles (stride=320)



(4b) Helper Thread L2 Request Breakdown (stride=320)

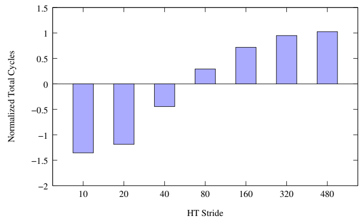


(7a) Normalized Total Cycles (stride=480)

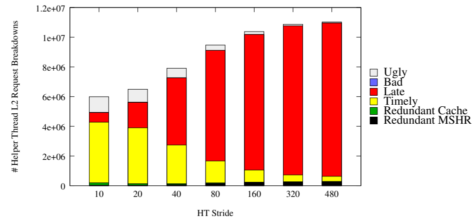


(5b) Helper Thread L2 Request Breakdown (stride=480)

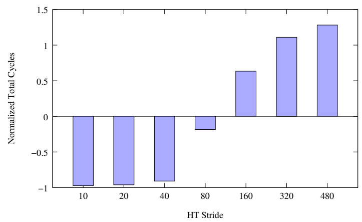
Fig. 8: Impact of Prefetching Lookahead on mst Helper Threaded Prefetching Performance



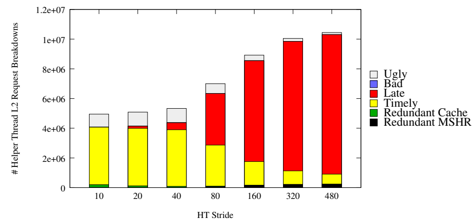
(1a) Normalized Total Cycles (lookahead=10)



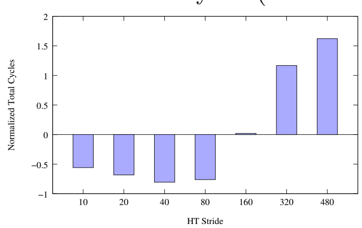
(1b) Helper Thread L2 Request Breakdown (lookahead=10)



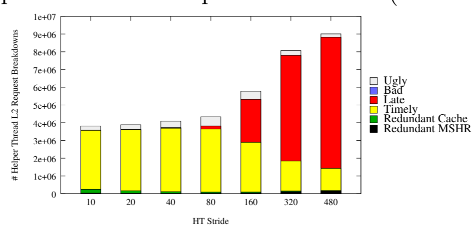
(2a) Normalized Total Cycles (lookahead=20)



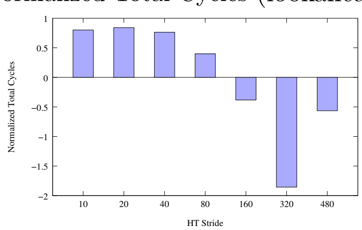
(2b) Helper Thread L2 Request Breakdown (lookahead=20)



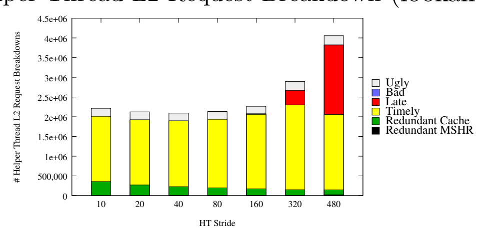
(3a) Normalized Total Cycles (lookahead=40)



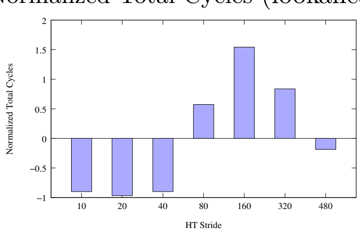
(3b) Helper Thread L2 Request Breakdown (lookahead=40)



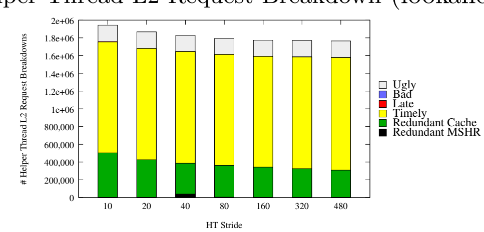
(4a) Normalized Total Cycles (lookahead=80)



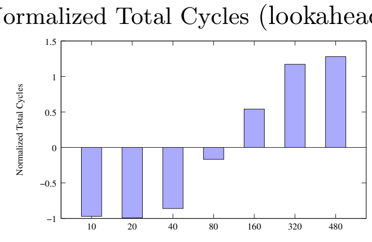
(4b) Helper Thread L2 Request Breakdown (lookahead=80)



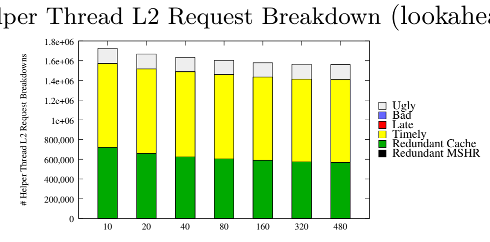
(5a) Normalized Total Cycles (lookahead=160)



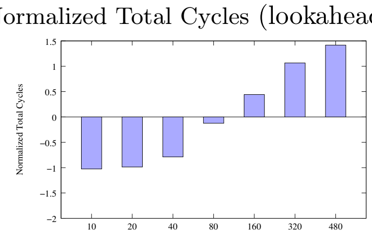
(5b) Helper Thread L2 Request Breakdown (lookahead=160)



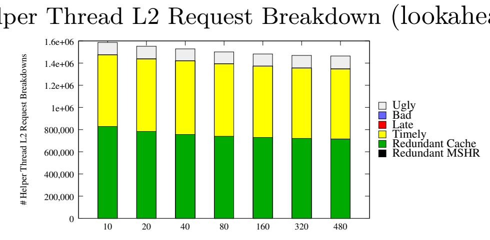
(6a) Normalized Total Cycles (lookahead=320)



(4b) Helper Thread L2 Request Breakdown (lookahead=320)



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



(5b) Helper Thread L2 Request Breakdown (lookahead=480)

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

5 Conclusion *(to be filled)*

Acknowlegments

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

References

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

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

[3] 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.

[4] 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. [Online]. Available: http://doi.acm.org/10.1145/2155620.2155672

[5] 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.