2014 International Conference on Embedded and Ubiquitous Computing

Understanding the Dynamic Caches on Intel Processors: Methods and Applications

Yi Zhang, Nan Guan
Department of Computer Science and Technology
Northeastern University, China
{*zhangyi,guannan*}*@ise.neu.edu.cn*

Wang Yi
Department of Information Technology
Uppsala University, Sweden
yi@it.uu.se

***Abstract*—The design and implementation of caches on a given platform has significant impacts to many areas in computer system design. On chip-multiprocessors (CMP), new cache architectures are proposed to meet the rapidly increasing performance requirements. However, the cache architectures are usually not well-documented for commercial processors. This raises difficulties for people to precisely understand the working principle of many components of the processors, not only the cache itself, but also the related components like the whole memory subsystem.**

This paper aims at disclosing the working principle of the last level cache of Intel Ivy Bridge processors. First, we identify the address translation logic on this cache. Second, we disclose the replacement policy of the cache. This is a dynamic insertion replacement policy, which is very different from the widely used LRU policy and its variants. Although this replacement policy has been proposed in academic literatures, our work is the first one showing it is actually used in commercial processors. To show the significance of our discovery, we design a methodology to generate controllable cache miss sequences under this new cache, and apply it to the design of a benchmark to model the memory concurrency. Evaluations on physical machines are conducted to show the effectiveness of the proposed method.

***Keywords*-Dynamic Cache; Cache Replacement Policy; Profiling;**

I. Introduction

Due to the large speed gap between processors and the memory system, memory accessing has become the performance and throughput bottleneck. On chip-multiprocessors (CMP), this problem is not alleviated but becomes even more serious and complicated. On CMP, the increased number of cores demand higher memory bandwidth. However the memory bandwidth doesn't scale up with the number of cores, due to the hardware limitations such as pin numbers and power. Another problem for CMP memory management is raised by memory sharing, which introduces new issues on how to manage memory resource between co-running applications.

To achieve optimal performance and/or obtain predictable performance, it is necessary to understand the detailed information of memory hierarchy and the source of memory contentions. Unfortunately, in many cases, such information cannot be easily obtained. For example, it is usually unlikely to get enough information about particular architectural feature from the processor manuals (processor manufacturers commonly treat this information as commercial secretes). Moreover, it is not straightforward to predict how much bandwidth a program will consume. To get such information, an effective method is to construct measurement-based benchmark which can perform the memory characterizations [1], [2], [3], [4].

To construct measurement-based benchmarks, a basic prerequisite is generating memory requests in a controllable manner. Because of the complexity of modern processors, there are many factors that could make the final access sequences very different from the original memory requests. One important factor is the cache behavior. In order to control the memory accesses that actually reach the memory, one must know the cache behavior of these requests. In this work we fucus on the influences that come from the new cache characteristics on Intel Ivy Bridge processor and explore how to build benchmarks for modeling memory concurrency with the new cache.

Over the last few decades, cache organization and cache replacement policies didn't have big changes on the real implementations. Therefore, many benchmarks are built with the common assumptions of the underlying cache parameters. However, on the newly proposed Intel processors, the last level cache (LLC) appears to have new features. One potential change is the cache address translation. According to the document [5], the addresses of LLC on Sandy Bridge and Ivy Bridge architecture are mapped by a hash function, and it doesn't work in the same way as the normal N-way cache. While in our test for identifying the cache address translation, we further discover that the cache replacement policy of Ivy Bridge LLC adopts dynamic insertion policy mechanism, which is very different from the widely used least-recently used (LRU) replacement policy and its approximations. As a result, the existing benchmarks that are built based on LRU replacement policy become unsuitable to the processors with these new cache architectures.

In this work, we study the new features of the LLC on Intel Ivy Bridge processors and propose a method to design benchmarks with the new caches. The contributions of this work include:

- We study the new cache address translation on Intel Ivy Bridge processors and discover new cache replacement policy applied on this processor.
- We propose a method for identifying the new cache replacement policy and a method to generate controllable

This work is partially supported by NSF of China under grant No. 61100023, 61300022 and 61370076.

978-0-7695-5249-1/14 $31.00 © 2014 IEEE
DOI 10.1109/EUC.2014.18

cache misses with the new cache.
- We implement memory characterization benchmarks with our cache generating method and give the evaluation on an Intel Ivy Bridge machine.

The remainder of this paper is organized as follows: Section II presents the test of cache address translation; Section III introduces the approach to identify the new replacement policy and the method to generate controllable cache miss sequences; Section IV introduces the implementation of the benchmarks for modeling memory concurrency under new cache policy and the experimental evaluation. Section V reviews related works, and finally conclusions are drawn in Section VI.

II. Problem statement

We observed the new cache characteristics when testing the address translation of the LLC on our Intel processor. In this section, we present our method and results of the address translation testing on the Intel Ivy Bridge processor. By this test, we also discover some abnormal phenomenons that cannot be explained assuming the common LRU cache is implemented. We will analyze the phenomenons in the next section.

A. Experimental platform

The experimental machine used in this work is with a 4-core Intel Core i7-3770 (Ivy Bridge) processor. Each core has a 32K 8-way set-associative L1 instruction cache, 32K 8-way set-associative L1 data cache, and a private 256K 8-way unified non-inclusive L2 cache. The L3 cache, i.e., the last-level cache (LLC) consists of 4 identical 2M 16-way set-associative inclusive cache slices, and all 4 slices are shared by the on-chip cores. The cache line size is 64B. The memory controller has two memory channels. The DIMM we use is the single-rank 2GB 1.5V 11-11-11 DDR3-1600 DIMM with 8 banks. In this section, experiments are conducted with one 2GB DIMM. The processor frequency is set to be 1.6GHz and the OS is Linux 2.6.39.

B. Cache organization

The cache used in modern computer is typically set-associative. A cache with associativity A is also called an A-way set-associative cache. The organization of a cache is specified by the following parameters in this paper:

- A: The associativity of cache (also the number of ways).
- B: The block (also called cache line) size in bytes.
- N: The number of cache sets
- W = B $*$ N : The cache capacity in bytes in one way.
- S = A $*$ W : The cache capacity in bytes.

For an L3 cache slice of our platform, the above parameters are: A=16, B=64B, N=2K, W=128KB, S=2MB.

C. Motivational experiment

Address translation decides the data location on cache. Commonly, address translation works according to the following rule:

Rule 1. *The addresses that differ by $n * W$ (n is nonzero integer) are located at the same set, different ways.*

However, according to the document [5], the addresses of LLC on Sandy Bridge and Ivy Bridge architectures are mapped by a hash function, and it works differently from the normal N-way caches. To get the address translation of LLC , we design a test with the knowledge from [6], according to which we follow the rule in below to design the test:

Rule 2. *For LRU and its approximations, if the number of items located in a cache set is greater than associativity and those items are sequentially accessed, then each access will cause a cache miss.*

```
Base = memAlloc(2*S);
assoc = 1;
while(assoc < maxAssoc){
      set = 0;
      while (set < N){
           init(Base + set, assoc);
           repeatAccess(Base + set);
           printtime();
           set = set + 1;
      }
      assoc = assoc + 1;
}
```

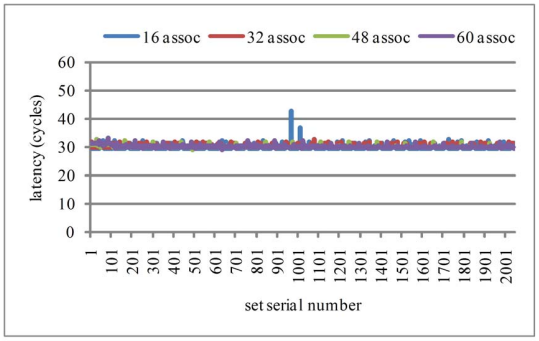
Listing 1. Test for address translation.

1) Description of Test: Our test is illustrated in Listing 1. The main idea in this test is: at each $assoc$, the program traverses all the cache sets, and on each set it repeatedly and sequentially accesses a list with $assoc$ items.

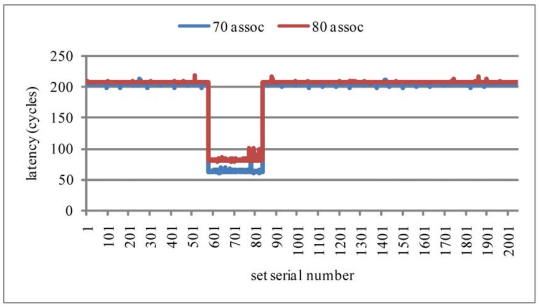
In the innermost loop, the function $init$ initializes the list with $assoc$ items. The addresses for those items have the offset starting at the $Base + B * set$ and differ by $n * W$. The $repeatAccess$ executes the item access and record the access time. To eliminate the interferences come from system, this function repeatedly accesses the list for 10000 times. And we directly read the time stamp counter to get the exact execution time.

To get an accurate latency for each access, we adopt following techniques to implement the test.

- To eliminate multiple issues of the access, we implement the item list in a form of "pointer chasing", where each item contains the address of the item which should be accessed immediately after it, thus the next access won't be issued until the previous one has returned. The body of the pointer chasing is implemented in the following method:
  $p = *(void**)p$
- To eliminate the latency caused by TLB misses, we allocate memory by huge pages.



(a) Number of items smaller than 64



(b) Number of items greater than 64

Figure 1. Results for address translation test.

- To guarantee the continuous physical address mapping on cache, we configure the size of continuous memory allocation in OS twice as much as the size of LLC(16MB in our test).
- To eliminate prefetching, we initialize the item list in a random order.
- Because write can be stalled by read, the access is implemented with read.

If the tested cache adopts the LRU policy or its approximations, and meanwhile the address translation adopts Rule 1, when $assoc$ is larger than the associativity of cache, the access of each item will generate a cache miss and the latency for each item access will be the memory access time; otherwise, when $assoc$ is smaller than the associativity of cache, the latency for each item access will be the cache access time.

Figure 1 shows parts of the test results. In our test, S is 8MB, N is 2048, W is 128KB, and $maxAssoc$ is 80. The results can be classified into two parts:

- When $assoc$ is from 8 to 64, the average latency for each access time is roughly 30 cycles, as shown in Figure 1-(a).
- When $assoc$ is greater than 64, latencies for accesses on different sets are shown in Figure 1-(b).

We don't show the results when $assoc$ is not greater than 8, which is the associativity of the L2 cache. During this range, most accesses are serviced on the L2 cache but not on the LLC, and the average latency for each access is roughly 10 cycles.

2) Observations: From Figure 1-(a) we can see that each access roughly takes 30 cycles (the latency for access LLC cache [5]) which implies all those accesses arrive at LLC cache. While in Figure 1-(b), the average access time on all sets is longer than 30 cycles, which implies there are cache misses generated under this condition.

From Figure 1-(b) we can also observe that the latencies on different sets are not uniform. On some sets, they are about 200 cycles (the order of memory accesses), which implies every access incurs a miss and coheres with Rule 2. While on some sets, the latencies are between 30 and 200 cycles, which implies not all of the accesses incur cache misses. We infer that on those sets the address translation still follow Rule 1, otherwise it won't incur cache misses on those sets in our test.

Thus through this test, we could get two informations about this LLC: (1) the associativity on the LLC is 64; (2) its address translation (at least for 16MB continuous address) follows Rule 1 and evenly distribute the addresses (continuous 128KB physical address covers 2K sets).

III. Discovery of the New Replacement Policy

We infer that abnormal phenomenon is due to a new cache replacement policy used in the Intel Ivy Bridge processors. In [7], [8], it has been pointed out that on the LLC of multi-core processors, the LRU policy uses cache space inefficiently. These works present two main ideas to design new cache replacement policy. The first one is *Set Dueling based Dynamic Insertion Policy* (SDDIP) [7], which could dynamically estimate the number of misses incurred by the two competing insertion policies and select the policy that incurs the fewest misses [7]. The other idea is the *Hit Promotion Policies* which do not place the incoming line at the MRU position as the LRU policy does. Instead, they would place the incoming line at LRU position or some middle position and promote that line to the MRU position when it is reused. The hit promotion mechanism gives an utility-based replacement policy and could reduce the thrashing problem when a working set is greater than the cache size. We could observe that, in Figure 1-(b), the cache miss line appears to have two patterns and in one pattern the sets still have cache hits when the number of the items is greater than the cache associativity.

Our goal in this part is to identify the cache replacement policy that matches the above phenomenon, and develop methods to issue predictable cache misses under the new policy. The main challenge is to solve the unpredictable cache behavior caused by SDDIP.

A. SDDIP identification

We first introduce the mechanism of SDDIP. Under the SDDIP mechanism, there are two policies which dynamically compete to be the dominant policy on the cache. The Set Dueling mechanism dedicates a small number of sets to each of the two competing policies. The policy that incurs fewer misses on the dedicated sets is used for the remaining

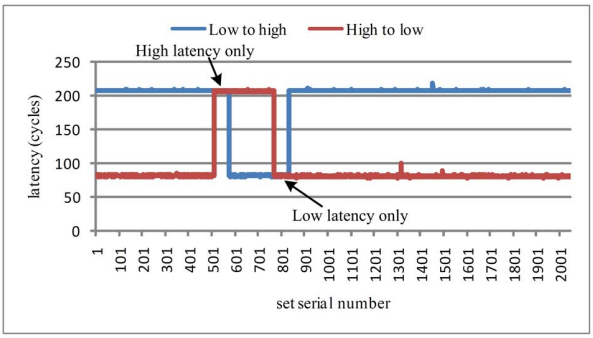


Figure 2. Results for SDDIP policy identification.

follower sets. Thus under SDDIP, most sets become adaptive to the access patterns. This mechanism is designed with the insight in [9] that the cache behavior can be approximated by sampling a small number of sets on the cache.

We identify the SDDIP by running *Double Scans* over the cache. The scanning program is implemented in the same way as presented in the address translation test. In this program, $assoc$ is greater than cache associativity which could ensure the generation of cache misses and distinguish the behaviors between LRU and hit promotion policy. The first scan starts from the set with the lowest address to the highest end, and the second scan is the other way around.

The goal of this program is to tell the location of the dedicated sets on Set Dueling based cache. By running this program, we shall observe three behaviors if SDDIP is implemented:

- When the scan is going on the follower sets: since there is no accesses on the dedicated sets, the cache replacement policy should stay the same, and hence cache miss pattern and scanning line pattern stay the same.
- When the scan comes to the dedicated sets: since the scan incurs cache misses on the current one, that means the other dedicated sets at this moment have fewer cache misses. When scan with this dedicated sets is finished, the cache miss pattern should change to the other one and the scanning line jumps into another pattern.
- Since we scan the cache from different directions, then the orders that two scans meet the dedicated sets are the opposite and the behaviors on follower sets are the opposite. But in both runs, the scanning line behavior on the dedicated sets keeps the same.

We run the double scan program with $assoc$ equals to 80. The results are shown in Figure 2, where the blue line represents the scan from low to high, and the red line is from high to low. From the Figure 2 we can observe that there are two small areas (each area covers 64 continuous sets) that only give the high access latency and low access latency respectively and all the other sets can have both latency patterns. According to the results, we infer that Set Dueling policy is implemented on this cache and get the locations of dedicated sets.

B. Predictable cache miss generation on SDDIP

To build a measurement-based benchmarks, it is critical to make the behavior of benchmark predictable. However, SDDIP would dynamically switch the running replacement policy and change the pattern of memory accesses. To deal with this problem, we propose to explicitly fix the available replacement policy in benchmarks. To achieve this goal, two steps need to be added into the benchmark building process.

- Identifying dedicated sets: this work can be achieved by running the double scan approach.
- Policy-fix: In the benchmark building process, we do not use the memory area maps to the dedicated sets with the needed policy. Instead we include the memory area maps to the other dedicated sets in use. Thus, the needed policy will be activated when the benchmark is running. Also, to minimize the unnecessary cache miss generations, the used dedicated sets should keep a low percentage in the overall used sets (this is achievable since the dedicated sets take a small percentage among overall sets).

By adding the above steps, the aimed policy can be applied when benchmark is running.

IV. Modeling Memory Concurrency

To evaluate the effectiveness of the policy-fix method, in this section we apply it into building the benchmarks for modeling the memory concurrency bottlenecks and perform examinations with our machine.

A. Background

1) Memory Hierarchy: Most memory system of modern computer is built on DRAM which is a three dimensional hierarchy organized by bank, row and column. A DRAM module is composed of several independent banks. Among banks, requests can be operated in parallel and within each bank, data is organized based on row and column. Memory controller (MC) is the mediator between cache and DRAMs, manages requests into and out of DRAMs while ensuring protocol compliance. DRAM modules are connected with MC via channel. When there are multiple channels, requests can be independently operated between channels. Transfer size for a memory request is typically the same as cache line (64 bytes on our machine), which means a bunch of words will be transferred on a single memory request.

To read or write a word of data on a bank, the row with requested data must be first loaded into row buffer which is single for each bank. The latency of a memory request therefore depends on whether or not the requested row is in the buffer. Caused by the operations on row buffer, a memory request can have three different access events: *row-hit*, requested row is already in the buffer and data can be accessed directly; *row-closed*, row buffer is empty and requested row needs to be loaded from the bank before

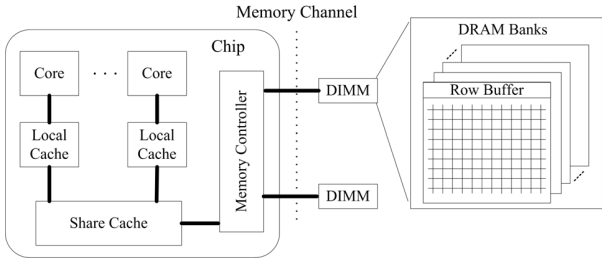


Figure 3. An illustration for memory hierarchy considered in this paper.

access is performed; *row-conflict*, row buffer is holding the contents of another row. In this case, the contents need to be first written back to the bank, and then load requested row into buffer and conduct access. In these three events, row conflict is with the longest access latency and row hit has the shortest. If there is a multi-socket multiprocessor system, memory access latency also depends on in which socket the contents are resided and how the MC is shared among processors.

The memory hierarchy explored in this paper is shown in Figure 3. It has multiple layer caches with the last level cache shared among cores and a on-chip MC manages the memory accesses for all the cores. When a memory request is issued, it will go from the top level cache down to the memory until it gets the requested data.

2) Medeling Memory Hierarchy: In the view of performance, there are three metrics for measuring memory hierarchy: *bandwidth, latency* and *concurrency*. By using Little's law [10] to give model, the three metrics can draw the following equation:

$$bandwidth = \frac{cacheline\_size * conc\_num}{latency} \quad (1)$$

where $conc\_num$ is the in-flight accesses in the memory system and $latency$ is the average service time for each cache miss.

The equation 1 tells the fact that the memory bandwidth a hardware platform can provide or an application will consume is decided by the memory access concurrency and latency. However, those two parameters varies at different levels of memory hierarchies. For example, at the bank level, access concurrency is limited by the number of banks and the access time spent on bank operations is the longest among all memory hierarchy levels. While at the cache level, more parallel memory accesses can be serviced with several or tens of cpu cycles.

By explicitly controlling the number of requests issued to memory system, Mandal et al. [3] studied the memory performance bottleneck affected by memory concurrency on multi-socket multi-core systems. In addition to measuring the performance impact of memory contention on real hardware, Eklov et al. [4] further utilized this method to analyze memory bandwidth and latency sensitivity of applications. But all those works take the assumption that the cache replacement policy is LRU or its approximations.

B. Building benchmark

To examine the memory concurrency, it is required that the benchmark could specify the number of issued memory accesses. To achieve this goal, this benchmark is implemented by traversing a number of linked lists. Each list is implemented with pointer-chasing method which guarantees each access on list generates one memory access at a time. By adjusting the number of linked lists in each run, we could control the number of parallel accesses issued into the system. In building this benchmark, we also use the techniques introduced in address translation test at Section II-C (such as huge pages and random order initialization) to eliminate the potential interferences from system.

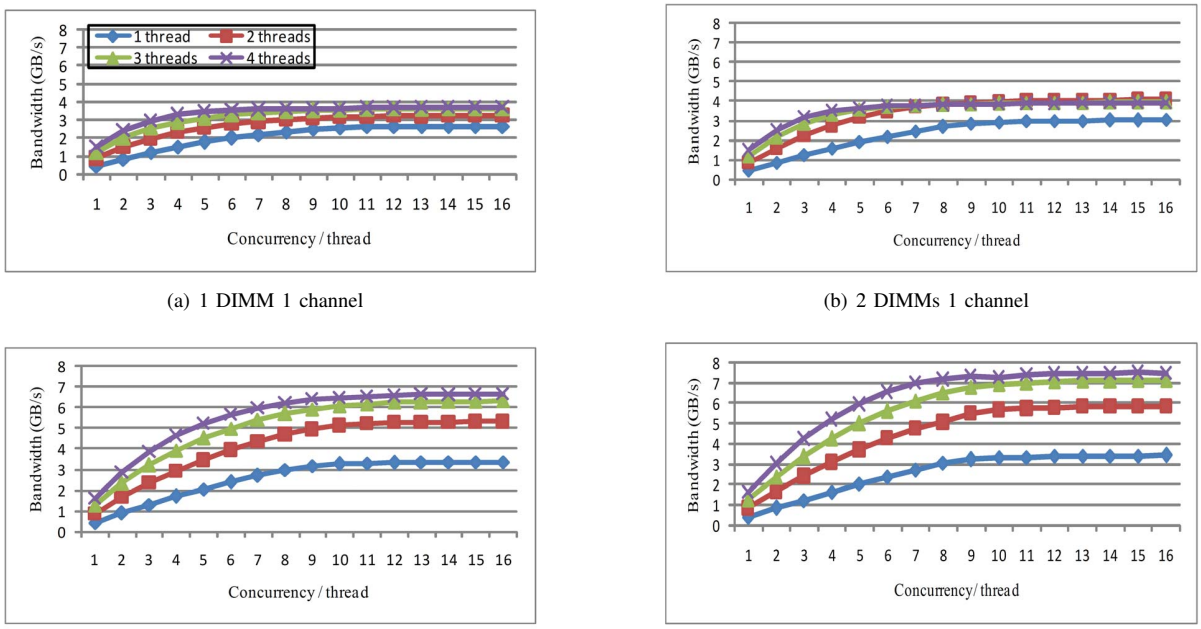
C. Experimental evaluation

We run the experiment on the same machine used in previous tests. The experiment was conducted under four different memory configurations which differed in the number of available DIMMs and channels. In each memory configuration, we varied the number of simultaneously running threads from 1 to 4 (each tied to its own core) and increased the number of each thread's parallel memory accesses from 1 to 16. There were 64 combinations of number of threads and concurrency in each memory configuration.

Each thread was running under LRU policy by using policy-fix method. For each thread, it monopolized 500 follower sets, 5 dedicated sets and had 96 (1.5 associativity) items located in each set. Using the above parameters, we could assume that every access on a linked list would cause a cache miss. The items in linked lists were initialized with the random order and the size of overall items took around 3MB memory. Therefore for most of the cases, the consecutive cache misses located in different rows and the average latency for each memory access was the row conflict latency (about 220 cpu cycles in our experiment).

To reduce the interferences, under each memory configuration we first enable the LRU policy by running a function to generate cache misses on the dedicated sets. After that, the threads start to repeatedly read the linked lists for 100000 times. The results are shown in Figure 4, illustrated with bandwidth and concurrency. According to Eg. 1, if the latency is fixed, the bandwidth will increase with respect to the concurrency. Hence, we can detect the memory concurrency bottlenecks by reading when the curves level off. From those results, we could get three different memory concurrency bottlenecks on our machine:

- Maximal concurrency at each core: in each figure, every 1 thread curve levels off at 10 concurrency. It means every single core can at most issue 10 memory accesses in parallel.
- Maximal concurrency in a channel: from Figure 4(a), 4(b), the curve for 3 threads levels off at 8 (3*8=24) and curve for 4 threads levels off at 6 (4*6=24), indicating the maximal concurrency of a channel is about 24.
- Maximal global concurrency: in Figure 4(d), curve for 3 threads levels off at 11 (3*11=33) and curve for 4



(a) 1 DIMM 1 channel

(b) 2 DIMMs 1 channel

(c) 2 DIMMs 2 channels

(d) 4 DIMMs 2 channels

Figure 4. Modeling memory concurrency with different number of DIMMs and channels.

threads levels off at 8 (4*8=32), indicating the maximal global concurrency is around 32.

V. Related Work

Researchers have studied CMP memory sharing and contention problem from different aspects. Zhuravlev et al. [11] presented the contention-aware scheduling policies for mitigating contentions on shared cache&memory and analyzed the various thread classification schemes for scheduler design. Mutlu et al. [12] showed that a MC with memory parallelism awareness can both improve system throughput and fairness. Tang et al. [13] and Dey et al. [14] studied the impact of shared-resource contention on multithreaded applications and investigated approaches on how to design a good thread-to-core mappings based on the resource sensitivity of applications.

Many measurement-based approaches have been proposed to obtain the details of memory hierarchy. Yotov et al. [1] presented micro-benchmarks for determining parameters of different memory hierarchy, including registers, all data caches and translation look-aside buffer. Molka et al. [15] revealed detailed performance characteristics of cache and memory subsystem of Intel Nehalem microarchitecture. Abel et al. [6] proposed an algorithm to automatically infer the cache replacement policy and applied this algorithm to various popular microarchitectures. Mandal et al. [3] modeled memory concurrency for multi-socket multi-core system and use the results to create a more accurate model of memory subsystem. Based on this model, Eklov et al. [4] presented a bandwidth stealing method to analyze the resource sensitivity of applications

In the architecture community, much work has been proposed on designing new efficient CMP cache replacement policies. The advantage for the works in [7], [8] is they use existing cache structure and require small changes into existing cache replacement policy. The dynamic insertion policy cache brings in big changes to conventional replacement policies, and existing approaches needs to be redesigned on this new architecture [16].

VI. Conclusion

In this work, we examine two characteristics of the LLC on Intel Ivy Bridge processors: the address translation logic and cache replacement policy. We discover that, the cache replacement policy on this processor adopts a new dynamic insertion mechanism, which is very different from the widely used conventional LRU policy and makes many existing techniques infeasible. We present a method to identify and profile this new policy. With the understanding of this new policy, we design a method to generate controllable cache misses on this cache. We then integrate this method into building the benchmark for modeling memory concurrency and evaluate the benchmark by examining the memory concurrency bottlenecks of a physical machine.

References

[1] K. Yotov, K. Pingali, and P. Stodghill, "Automatic measurement of memory hierarchy parameters," *ACM SIGMETRICS Performance Evaluation Review*, vol. 33, no. 1, pp. 181–192, 2005.

[2] D. M. Pase and M. A. Eckl, "Performance of the AMD opteron LS21 for IBM Bladecenter," Technical Report, IBM Corporation, Research Triangle Park, North Carolina, Tech. Rep., 2006.

[3] A. Mandal, R. Fowler, and A. Porterfield, "Modeling memory concurrency for multi-socket multi-core systems," in *IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*. IEEE, 2010, pp. 66–75.

[4] D. Eklov, N. Nikoleris, D. Black-Schaffer, and E. Hagersten, "Bandwidth bandit: Understanding memory contention," in *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2012, pp. 116–117.

[5] "Intel 64 and IA-32 architectures optimization reference manual," April 2012.

[6] A. Abel and J. Reineke, "Measurement-based modeling of the cache replacement policy," in *Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2013, pp. 65–74.

[7] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer, "Adaptive insertion policies for high performance caching," in *ACM SIGARCH Computer Architecture News*, vol. 35, no. 2. ACM, 2007, pp. 381–391.

[8] A. Jaleel, K. B. Theobald, S. C. Steely Jr, and J. Emer, "High performance cache replacement using re-reference interval prediction (RRIP)," in *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3. ACM, 2010, pp. 60–71.

[9] M. K. Qureshi, D. N. Lynch, O. Mutlu, and Y. N. Patt, "A case for mlp-aware cache replacement," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 2, pp. 167–178, 2006.

[10] J. D. Little, "A proof for the queuing formula: L= λ w," *Operations research*, vol. 9, no. 3, pp. 383–387, 1961.

[11] S. Zhuravlev, S. Blagodurov, and A. Fedorova, "Addressing shared resource contention in multicore processors via scheduling," *ACM SIGPLAN notices*, vol. 45, no. 3, pp. 129–141, 2010.

[12] O. Mutlu and T. Moscibroda, "Parallelism-aware batch scheduling: Enabling high-performance and fair shared memory controllers," *Micro*, vol. 29, no. 1, pp. 22–32, 2009.

[13] L. Tang, J. Mars, N. Vachharajani, R. Hundt, and M. L. Soffa, "The impact of memory subsystem resource sharing on datacenter applications," in *Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2011, pp. 283–294.

[14] T. Dey, W. Wang, J. W. Davidson, and M. L. Soffa, "Characterizing multi-threaded applications based on shared-resource contention," in *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2011, pp. 76–86.

[15] D. Molka, D. Hackenberg, R. Schone, and M. S. Muller, "Memory performance and cache coherency effects on an intel nehalem multiprocessor system," in *Parallel Architectures and Compilation Techniques, 2009. PACT'09. 18th International Conference on*. IEEE, 2009, pp. 261–270.

[16] A. Jaleel, H. H. Najaf-Abadi, S. Subramaniam, S. C. Steely, and J. Emer, "Cruise: cache replacement and utility-aware scheduling," in *ACM SIGARCH Computer Architecture News*, vol. 40, no. 1. ACM, 2012, pp. 249–260.