High Performing Cache Hierarchies for Server Workloads

Relaxing Inclusion to Capture the Latency Benefits of Exclusive Caches

Aamer Jaleel[†], Joseph Nuzman[‡], Adrian Moga[‡], Simon C. Steely Jr.[†], Joel Emer[§]

[†‡]Intel Corporation, [†]VSSAD
Hudson, MA

[§]Massachusetts Institute of Technology (MIT)
Cambridge, MA

[§]NVIDIA Research
Westford, MA

{aamer.jaleel, joseph.nuzman, adrian.moga, simon.c.steely.jr}@intel.com

jemer@nvidia.com

Abstract—***Increasing transistor density enables adding more on-die cache real-estate. However, devoting more space to the shared last-level-cache (LLC) causes the memory latency bottleneck to move from memory access latency to shared cache access latency. As such, applications whose working set is larger than the smaller caches spend a large fraction of their execution time on shared cache access latency. To address this problem, this paper investigates increasing the size of smaller private caches in the hierarchy as opposed to increasing the shared LLC. Doing so improves average cache access latency for workloads whose working set fits into the larger private cache while retaining the benefits of a shared LLC. The consequence of increasing the size of private caches is to relax inclusion and build exclusive hierarchies. Thus, for the same total caching capacity, an exclusive cache hierarchy provides better cache access latency.***

We observe that server workloads benefit tremendously from an exclusive hierarchy with large private caches. This is primarily because large private caches accommodate the large code working-sets of server workloads. For a 16-core CMP, an exclusive cache hierarchy improves server workload performance by 5-12% as compared to an equal capacity inclusive cache hierarchy. The paper also presents directions for further research to maximize performance of exclusive cache hierarchies.

Keywords—commercial workloads, server cache hierarchy, cache replacement, inclusive, exclusive

I. INTRODUCTION

As the gap between processor and memory speeds continues to grow, processor architects face several important decisions when designing the on-chip cache hierarchy. These design choices are heavily influenced by the memory access characteristics of commonly executing applications. Server workloads, such as databases, transaction processing, and web servers, are an important class of applications commonly executing on multi-core servers. However, cache hierarchies of multi-core servers are not necessarily targeted for server applications [8]. This paper focuses on designing a high performing cache hierarchy that is applicable towards a wide variety of workloads.

Modern day multi-core processors, such as the Intel Core i7 [2], consist of a three-level cache hierarchy with small L1 and L2 caches and a large shared last-level cache (LLC) with as many banks as cores in the system (see Figure 1) [1, 2]. The small L1 and L2 caches are designed for fast cache access latency. The shared LLC on the other hand has slower cache access latency because of its large size (multi-megabytes) and also because of the on-chip network (e.g. ring) that interconnects cores and LLC banks. The design choice for a large shared LLC is to accommodate varying cache capacity demands of workloads concurrently executing on a CMP.

In a three-level hierarchy, small private L2 caches are a good design choice if the application working set fits into the available L2 cache. Unfortunately, small L2 caches degrade performance of server workloads that have an *intermediate* working set that is a few multiples (e.g. 2-4x) larger than the L2 cache size. In such situations, server workloads spend a large fraction of their execution time waiting on shared cache access latency, most of which is on-chip interconnect latency.

The interconnect latency can be tackled by removing the interconnection network entirely and designing private LLCs or a hybrid of private and shared LLC [9, 10, 30, 12, 28]. While hybrid LLCs provide the capacity benefits of a shared cache with the latency benefits of a private cache, they still suffer from the added L2 miss latency when the application working set is larger than the available L2 cache.

Alternatively, prefetching can be used to hide L2 miss latency. However, the access patterns of server workloads are hard to predict [34, 35]. Existing prefetching techniques targeted for server workloads [13, 14, 34, 35] either do not perform well across a broad range of workloads [5] or the prefetching techniques are too complex to be adopted by industry. Since server workloads represent an important class of workloads across various market segments, it is imperative to design a general purpose cache hierarchy that performs well across a wide variety of workload categories.

A straightforward mechanism to reduce the overhead of shared LLC access latency in a three-level hierarchy would be to build large L2 caches. For example, AMD processors use exclusive cache hierarchies with large L2 caches ranging from 512KB to 1MB [18, 11] as compared to Intel processors that use inclusive cache hierarchies with small 256KB L2s. To-date there exists no comprehensive published study on the benefits of one cache hierarchy over the other. We conduct a detailed simulation-based server workload study on a 16-core

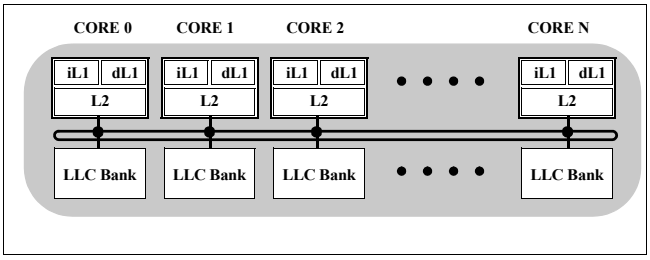


Figure 1: CMP with Three-Level Cache Hierarchy.

CMP using a three-level inclusive cache hierarchy (similar to Intel Core i7 [2]). Corroborating prior work [17], we find that server workloads spend a large fraction of their execution time waiting on LLC access latency. To minimize the impact of shared LLC latency, we make the following contributions:

- We find *front-end* instruction fetch unit misses in the L2 cache to constitute a non-negligible portion of total execution time. We propose simple techniques like *Code Line Preservation (CLIP)* to dynamically preserve latency critical *code* lines in the L2 cache over *data* lines. For server workloads, CLIP performs nearly the same as doubling the available L2 cache size. We advocate further research for practical code prefetching and cache management techniques to improve front-end performance of server workloads.
- Since server workloads benefit from large L2 cache sizes, we show that changing the baseline inclusive hierarchy to an exclusive cache hierarchy improves performance. This change also retains design constraints on total on-chip die space devoted to cache. We show that exclusive cache hierarchies provide benefit by improving average cache access latency.
- We show that exclusive cache hierarchies functionally break recent high performing replacement policies proposed for inclusive and non-inclusive caches [20, 21, 29, 36, 37]. We re-visit the Re-Reference Interval Prediction (RRIP) cache replacement policy used in commercial LLCs today [3, 4]. For an exclusive cache hierarchy, we show that adding support in the L2 cache (a single bit per L2 cache line) to remember re-reference information in the LLC restores RRIP functionality to provide high cache performance.

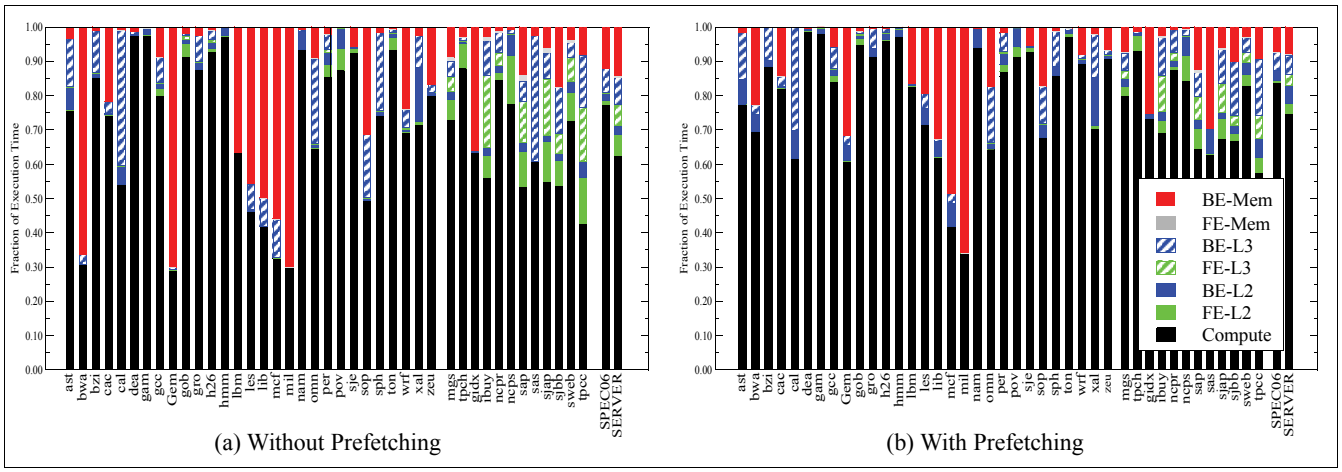
II. MOTIVATION

Figure 2 illustrates the normalized CPI stack of SPEC CPU2006 (a commonly used representative set of scientific and engineering workloads) and server workloads simulated on a processor and cache hierarchy configuration similar to the Intel Core i7. The study is evaluated by enabling only a single core[1] in a 16-core CMP with a 32MB shared LLC. The normalized CPI stack denotes the fraction of total execution time spent waiting on different cache hit/miss events in the processor pipeline. We separate the CPI stack into cycles spent doing *compute*, cycles where the front-end (FE) (i.e. instruction fetch unit) is stalled waiting for an L1 miss and L2 hit response (FE-L2), cycles where the back-end (BE) (i.e. load-store unit) is stalled waiting for an L1 miss and L2 hit response (BE-L2), cycles where the front-end is stalled waiting for an L1/L2 miss and LLC hit response (FE-L3), cycles where the back-end is stalled waiting for an L1/L2 miss and LLC hit response (BE-L3), cycles where the front-end is stalled waiting for an LLC miss response (FE-Mem), and finally cycles where the back-end is stalled waiting for an LLC miss response (BE-Mem). Note that FE-L3 and BE-L3 components of the CPI stack correspond to stalls attributed to shared LLC hit latency (which includes the interconnection network latency). Workload behavior is presented with and without aggressive hardware prefetching. Per benchmark behavior of SPEC CPU2006 and server workloads and average behavior of the two workload categories is shown.

The figure shows that several workloads spend a significant fraction of total execution time stalled on shared LLC access latency. These stalls correspond to both code and data misses in the private L1 and L2 caches. For example, the figure shows that server workloads can spend 10-30% of total execution time waiting on shared LLC access latency (e.g. server workloads *ibuy, sap, sjap, sjbb, sweb* and *tpcc*). This implies that the workload working-set size is larger than the available L2 cache. Specifically, the large code working set size corresponds to high front-end stalls due to shared LLC access latency. Similarly, the large data working set sizes of both server and SPEC workloads contribute to the back-end stalls related to shared LLC access latency. Furthermore, note that hardware prefetching does not completely hide the shared LLC access latency.

Figure 2 illustrates that even though shared LLCs provide capacity benefits, they move the memory latency bottleneck from main memory access latency to shared cache access

1. Details on experimental methodology is in Section V. We illustrate a single core study here for motivation purposes.



(a) Without Prefetching (b) With Prefetching

Figure 2: Performance Profile of SPEC CPU2006 and Server Workloads.

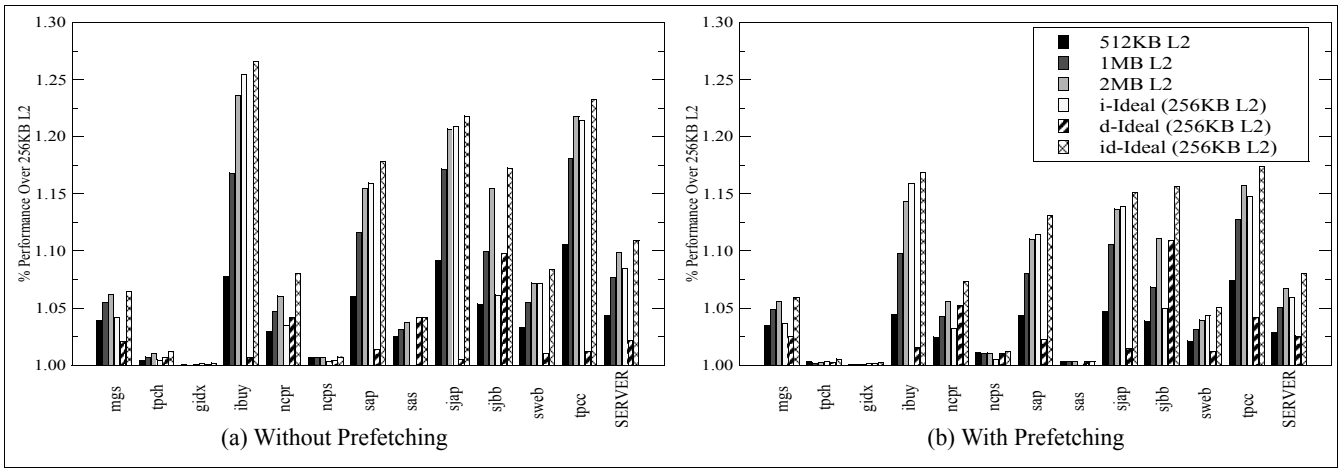


Figure 3: Performance Sensitivity of Server Workloads.

latency. This occurs when the working set size is larger than the available L2 cache but smaller than the available shared LLC size. Furthermore, the figure also shows that code misses in server workloads contribute to a significant fraction of overall execution time. Since L2 caches are typically designed to reduce average cache access latency and avoid shared cache accesses, Figure 2 motivates revisiting L2 cache design.

III. SERVER CACHE HIERARCHY ANALYSIS

To minimize stalls due to shared cache access latency, a natural step is to simply increase the L2 cache size to accommodate more of the application working set. Doing so enables more of the workload working set to be serviced at L2 cache hit latency as opposed to shared cache access latency. This section provides detailed cache hierarchy analysis for server workloads. Specifically, we analyze the tradeoffs of different sized L2 caches and manufacturing constrains of increasing on-chip cache sizes. While our conclusions confirm with existing trends observed in modern microprocessors from AMD and Intel, to the best of our knowledge, this is the first documented study of its kind.

A. Performance Benefits of Increasing L2 Cache Size

Simply increasing the L2 cache has several ramifications on the organization of the processor cache hierarchy. For example increasing L2 cache size can affect the L2 cache access latency and design choices on whether or not to enforce inclusion. Without going into these ramifications, we first investigate the performance potential of simply increasing the L2 cache size. To study these effects, we investigate the performance of increasing the L2 cache size assuming no impact on L2 cache access latency. Furthermore, to avoid inclusion side effects, we assume a large 32MB LLC.

Figure 3 illustrates the performance of increasing the L2 cache size for the server workloads. The x-axis presents the workloads while the y-axis presents the performance compared to a 256KB L2 cache. The first three bars represent three different L2 cache sizes: 512KB, 1MB, and 2MB. We omit showing performance behavior of the entire SPEC suite because across all 55 SPEC workloads we observed 2.5% average performance improvement when increasing the L2 cache to 2MB[2]. From the figure, we observe that seven of the 12 server workloads observe more than 5% performance improvements with a larger L2 cache both in the presence and absence of prefetching. Furthermore, the results suggest that a 1MB L2 cache provides bulk of performance improvement.

Increasing the L2 cache size naturally yields performance improvement because more requests are serviced at L2 access latency due to the higher L2 cache hit rate. Since the unified L2 cache holds both code and data cache lines, we raise the following question: *Which of the two requests (code or data) serviced at L2 cache latency provides the majority of performance improvements of increasing the L2 cache size?*

To answer this question we designed a sensitivity study in the baseline 256KB MLC. Specifically, for requests that miss in the L2 cache but hit in the LLC, we evaluate the following: (a) code requests always serviced at MLC hit latency (i.e. assume zero LLC hit latency) labeled as *i-Ideal* (b) data requests always serviced at L2 hit latency (labeled as *d-Ideal*) and (c) both code and data requests serviced at L2 hit latency (labeled as *id-Ideal*). In all configurations, both code and data requests are inserted into *all* levels of the hierarchy. Also, note that this sensitivity study is not a perfect L2 cache study. The sensitivity study accounts for latency due to misses to memory and only measures latency sensitivity for those requests that miss in the L2 cache but hit in the LLC.

Figure 3 shows that servicing code requests at L2 hit latency has much better performance than always servicing data requests at L2 hit latency (*ncpr* and *sjbb* are exceptions because of their small code footprints). In fact, always servicing code requests at L2 hit latency has performance similar to a 1MB L2 cache. This suggests that increasing the L2 cache size to 1MB effectively captures the large instruction working set of these applications. These results confirm with instruction working set sizes ranging between 512KB and 1MB (see Figure 8).

2. Two SPEC CPU2006 workloads, *calculix* and *bzip2* (*liberty* input), observe 30% and 7% performance improvement from a large 2MB L2 cache. This is because the larger L2 cache fits the entire working set of each application. The remaining SPEC CPU2006 workloads experience less than 3% performance improvement with larger L2 caches.

Thus, Figure 3 suggests significant opportunity to improve server workload performance by increasing the L2 cache size. Furthermore, we observe that the majority of performance improvement from a large L2 cache size is due to servicing code requests at L2 cache hit latency. This also suggests opportunities to improve L2 cache management by preserving latency critical code lines over data lines.

B. Consequences of Increasing L2 Cache Size

Though increasing the L2 cache improves performance of server workloads, a number of tradeoffs must be considered.

B.1 Longer L2 Cache Latency

Increasing L2 cache size can potentially increase the cache access latency. While increasing the L2 cache size reduces average cache access latency for workloads whose working set fits into the larger L2 cache, the increased L2 cache access latency can potentially hurt the performance of workloads whose working set already fit into the existing smaller L2 cache. We analyzed the impact on cache access latency using industry tools (and CACTI [6]) for the cache sizes studied in the previous section: 256KB, 512KB, 1MB, and 2MB. Our analysis with both methods showed that increasing the L2 cache size causes a cache latency increase of one and two clock cycles for a 1MB and 2MB L2 cache respectively, while a 512KB L2 cache has no access latency impact. Our studies showed that performance impact due to increasing the L2 cache latency is minimal (less than 1% on average).

B.2 On-Die Area Limitations for Cache Space

While caches have the potential to significantly improve performance, manufacturing and design constraints usually impose a pre-defined on-chip area budget for cache. Thus, any changes in cache size must conform to these area constraints. Furthermore, any changes in cache sizes of a hierarchy must also conform to the design properties of the cache hierarchy. For example, increasing the L2 cache of the baseline inclusive cache hierarchy requires increasing the shared LLC size to maintain the inclusion property [22]. Thus, increasing the L2 cache size by a factor of four would require a similar increase in the shared LLC size to avoid inclusion overheads [22, 15].

Consider a processor with a baseline inclusive three-level cache hierarchy that has a 256KB L2 cache and a 2MB LLC. Increasing the L2 cache size to 512KB while retaining a 2MB inclusive LLC causes the L2:LLC ratio to change from 1:8 to 1:4. The new L2:LLC cache ratio not only wastes cache capacity due to duplication but also creates negative effects of inclusion [22, 15]. To avoid the negative effects of inclusion, the LLC must be increased to 4MB to maintain the original L2:LLC ratio. Similarly, if the L2 were to be increased to 1MB, then the LLC would need to be increased to 8MB to maintain the original L2:LLC ratio. Clearly, increasing both the L2 cache size and the LLC is not a scalable technique.

Alternatively, the L2 cache size can be increased by physically stealing cache space from other on-chip caches. The most logical place to physically *steal* cache space is the LLC. Consequently, increasing the L2 cache size effectively reduces the shared LLC size and requires re-visiting design decisions on whether or not to enforce inclusion.

B.3 Relaxing Inclusion Requirements

An alternative approach to increase the L2 cache size, while meeting manufacturing constraints, is to relax inclusion requirements and design an exclusive or non-inclusive cache hierarchy instead. Unlike an inclusive cache hierarchy that duplicates lines in the core caches and the LLC, an exclusive cache hierarchy maximizes the caching capacity of the hierarchy by disallowing duplication altogether [22]. An exclusive hierarchy is designed by first inserting lines into the smaller levels of the hierarchy. Lines are inserted into larger caches only when they are evicted from smaller caches.

A non-inclusive hierarchy, on the other hand, provides no guarantees on data duplication [22]. A non-inclusive hierarchy is designed by inserting lines into all (or some) levels of the hierarchy and lines evicted from the LLC need not be invalidated from the core caches (if present).

The capacity of an inclusive hierarchy is the size of the LLC while the capacity of an exclusive hierarchy is the sum of all the levels in the cache hierarchy. The capacity of a non-inclusive hierarchy depends on the amount of duplication and can range between the size of an inclusive and exclusive hierarchy. While exclusive hierarchies can fully utilize total on-chip caching capacity, they can observe reduced effective caching capacity due to duplication of data in the larger private caches. Nonetheless, we focus on transitioning from an inclusive hierarchy to an exclusive cache hierarchy.

Relaxing inclusion and designing an exclusive cache hierarchy enables increasing the L2 size while maintaining the constraints on total on-chip real-estate devoted to cache space. Note that our reasoning to transition from an inclusive cache hierarchy to an exclusive cache hierarchy differs significantly from prior work. Specifically, prior work promotes exclusive hierarchies to increase the effective caching capacity of the hierarchy [7, 11, 27, 24, 38, 40]. Instead, we focus on reorganizing the cache hierarchy and show that transitioning from our baseline inclusive hierarchy to an exclusive hierarchy maintains the *effective caching capacity of the hierarchy while improving overall average cache access latency (by growing the size of smaller caches)*.

Let us revisit our example cache hierarchy with a 256KB L2 cache and inclusive 2MB LLC. As illustrated in Figure 4, this hierarchy can be reorganized as an exclusive hierarchy with 512KB L2 cache and a 1.5 MB LLC. An alternative design point is a 1MB L2 and a 1MB exclusive LLC. Note that both exclusive hierarchies maintain a total 2MB caching capacity like the baseline inclusive hierarchy. However, depending on the application working set sizes, these cache hierarchies can provide different average cache access latency.

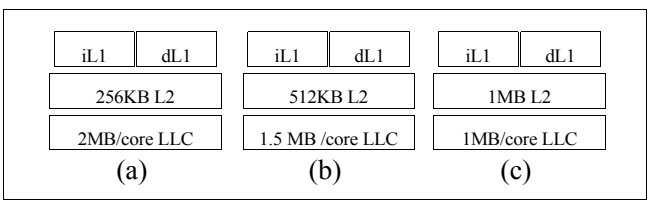


Figure 4: Cache Hierarchy Choices. (a) Baseline Inclusive Hierarchy with 256KB L2 (b) Exclusive Hierarchy with 512KB L2 (c) Exclusive Hierarchy with 1MB L2.

B.4 Tradeoffs of Relaxing Inclusion

We first discuss the performance tradeoffs and design considerations of relaxing inclusion:

- LLC Capacity: Increasing the size of the private L2 cache reduces the observed shared LLC capacity. The smaller LLC capacity can degrade performance of workloads whose working set exceeds the size of the smaller shared LLC but would have fit into the baseline shared LLC.
- Shared Data: Frequent accesses to read-only or read-write shared data require special consideration. Disallowing duplication of shared data in an exclusive cache hierarchy can incur long access latency to service data from a remote private L2 cache. This problem is usually addressed by allowing shared data to be duplicated in the LLC (like in inclusive caches) [11]. Doing so enables shared data to be serviced at LLC access latency. The amount of shared data replication in an exclusive hierarchy can directly impact the effective caching capacity of the hierarchy. We refer to a hierarchy that is exclusive for private data and inclusive for shared data as a *weak-exclusive* hierarchy. For brevity, here on, the terms weak-exclusive and exclusive are used synonymously.

We now discuss the storage and implementation complexities of relaxing inclusion:

- Unlike an inclusive hierarchy, clean cache lines that are evicted from smaller cache levels must be installed in the larger cache levels of an exclusive hierarchy. This strategy may require additional bandwidth support on the on-chip interconnection network.
- An exclusive hierarchy breaks snoop filtering benefits that naturally exists in an inclusive hierarchy [22]. Processor architects address this problem by devoting extra storage overhead for snoop filters [11, 38].

Despite these tradeoffs, the results from Figure 3 show that there is significant performance potential for commercial workloads with an exclusive cache hierarchy. We now focus on high performance policies for exclusive hierarchies.

IV. POLICIES FOR EXCLUSIVE HIERARCHIES

Having established that larger L2 caches and an exclusive LLC improve server workload performance, we now discuss mechanisms for a high performing exclusive cache hierarchy.

A. Exclusive Cache Management

Unlike inclusive caches where new lines are generally inserted in all levels of the cache hierarchy [22], an exclusive cache effectively acts like a *victim* cache [25, 24]. New lines are first inserted into the smaller levels of the cache hierarchy and are only inserted into the exclusive LLC upon *eviction* from the smaller cache levels. If a private cache line receives a hit in the exclusive LLC, the line is invalidated from the LLC to avoid duplication and make room for newly evicted cache lines from the smaller levels of the cache hierarchy. However, if a shared cache line receives a hit in the exclusive LLC, the line is not invalidated and the replacement state is updated instead (as in conventional inclusive LLCs). Shared cache lines are not invalidated in an exclusive LLC to enable fast access to subsequent requests by other cores sharing the same line.

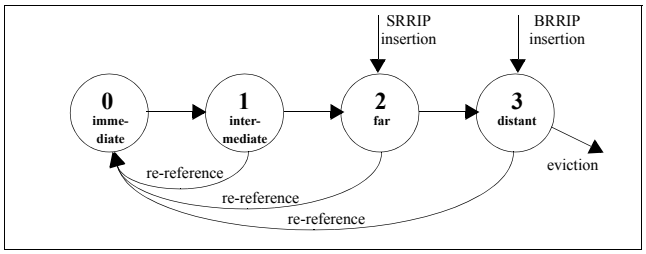


Figure 5: Dynamic Re-Reference Interval Prediction (DRRIP) Replacement for Inclusive LLCs.

The act of *invalidating* lines on cache hits in an exclusive hierarchy poses an interesting challenge for cache replacement policies. In general, the goal of cache replacement policies is to *preserve* lines that receive cache hits. Since cache hits discard cache lines from an exclusive LLC, simply applying recent state-of-the-art cache replacement policies [29, 21, 20, 36, 37] proposed for inclusive LLCs provide no performance benefits for exclusive LLCs. In an inclusive LLC, the LLC replacement state keeps track of all re-reference information. However, in an exclusive LLC, the re-reference information is *lost* on cache hits (due to invalidation). Thus, an exclusive LLC is unable to *preserve* cache lines that have been re-referenced. To improve exclusive cache performance, our key insight is that the LLC re-reference information must be *preserved* somewhere and the re-reference information be used upon *re-insertion* into the exclusive LLC (upon eviction from the smaller caches).

A natural place to store the LLC re-reference information in an exclusive hierarchy is the L2 cache. We propose a single bit per L2 cache line called the *Serviced From LLC (SFL)* bit. The SFL-bit tracks whether a cache line was serviced by main memory or by the LLC. If the line was serviced by the LLC, the line is inserted into the L2 cache with the SFL-bit set to one, otherwise the SFL-bit is set to zero. Upon eviction from the L2 cache, the SFL-bit can be used to restore functionality to recent state-of-the-art cache replacement policies.

To illustrate this, we use the simple and high performing *Re-Reference Interval Prediction (RRIP)* [20] replacement policy. Like LRU which holds the LRU position with each cache line, RRIP replaces the notion of the "LRU" position with a prediction of the likely re-reference interval of a cache line (see Figure 5). For example, with 2-bit RRIP, there are four possible re-reference intervals. If a line has re-reference interval of '0', it implies the line will most likely be re-referenced in the *immediate* future. If a line has re-reference interval of '3', it implies the line will be re-referenced in the *distant* future. In between *distant* and *immediate* re-reference intervals there are *intermediate* and *far* re-reference intervals. When selecting a victim, RRIP always selects a line with a *distant* re-reference interval for eviction. If no line is found, the re-reference interval of all lines in the set is increased until a line with *distant* re-reference interval is found. When inserting new lines in the cache, RRIP dynamically tries to learn the re-reference interval of a line by initially inserting ALL lines with *far* re-reference interval. This is done to

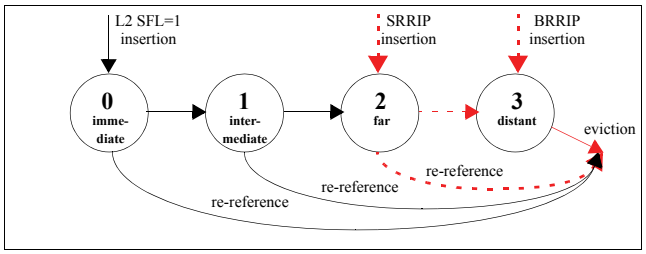


Figure 6: Dynamic Re-Reference Interval Prediction (DRRIP) Replacement for Exclusive LLCs.

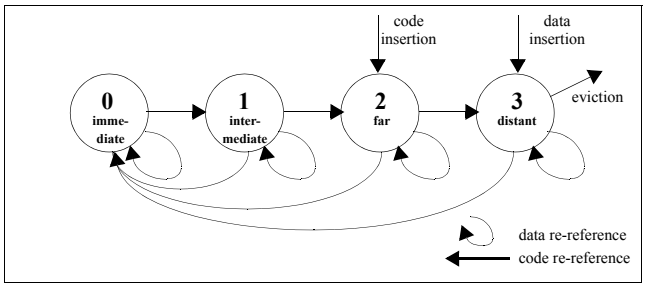


Figure 7: Code Line Preservation (CLIP).

dynamically learn the blocks re-reference interval. If the line has no locality, it will be quickly discarded. However, if the line has locality, the next re-reference to the line causes the line to have *immediate* state, hence preserving it in the cache.

Directly applying RRIP to an exclusive cache LLC removes RRIP functionality entirely by limiting RRIP to only 50% of the state-machine (as illustrated by the red dashed lines in Figure 6). To restore full RRIP functionality, we take advantage of the *SFL-bit* stored in the L2 cache and apply re-reference prediction on cache fills instead of cache hits. Specifically, if the line was originally serviced from memory (*SFL-bit* of L2 evicted line is zero), we predict the conventional *far* re-reference interval on cache insertion. However, if the line evicted from the L2 cache was originally serviced from the LLC (*SFL-bit* is one) we predict *immediate* re-reference interval on cache insertion. Doing so is equivalent to updating the re-reference interval to *immediate* on re-references in an inclusive LLC.

To illustrate this, assume a two-level hierarchy with a 1-entry L2 and 4-entry LLC, and the following access pattern:

... a, b, c, a, b, c, w, x, y, a, b, c,...

In an inclusive LLC, RRIP successfully preserves cache lines a, b, and c in the cache and incurs a 50% cache hit-rate at the end of the sequence above. However, in an exclusive LLC, simply applying RRIP is unable to preserve lines a, b, and c in the LLC and incurs a reduced 25% hit-rate. However, using the SFL-bit in the L2 cache restores RRIP functionality to exclusive caches and improves cache hit-rate to 50%. Note that the hit-rate is identical to RRIP on inclusive LLCs.

B. *L2 Cache Management for Server Workloads*

A unified L2 cache allocates both processor front-end *code* and back-end *data* requests. Conventional state-of-the-art cache management policies do not distinguish between code and data requests. However, front-end code request misses can tend to be more expensive than back-end data misses because out-of-order execution can overlap the latency effects of back-end data misses with independent work. Though decoupled front-end architectures hide the latency effects of code misses, the unpredictable nature of instruction streams in server workloads causes frequent front-end pipeline hiccups and limits instruction supply to the processor back-end. Since server workloads spend a significant fraction of their total execution time waiting for code misses to be serviced by the shared LLC, we investigate a novel opportunity to preserve latency critical *code* lines in the L2 cache over data lines.

A natural way of preserving (or prioritizing) lines in a cache is by taking advantage of the underlying cache replacement policy. We propose *Code Line Preservation (CLIP)* for the L2 cache. The goal of CLIP is to service the majority of L1 instruction cache misses by the L2 cache. This is accomplished by modifying the re-reference predictions of code and data requests on cache insertion and cache re-reference (see Figure 7). Specifically, CLIP always inserts code requests with *far* re-reference interval while data requests with *distant re-reference interval*. On L2 re-references (i.e. cache hit), *CLIP* updates code requests to *immediate* while it does not update any re-reference predictions on data requests.

While CLIP enables preservation of code lines in the L2 cache, blindly following CLIP can degrade performance when the instruction working set does not contend for the L2 cache (e.g. SPEC workloads). To address this, CLIP dynamically decides the re-reference prediction of data requests using *Set Sampling* [29]. CLIP samples a few sets of the cache (32 in our study) to always follow the baseline RRIP replacement policy. The sampled sets track the number of code and data misses in the L2 cache. Furthermore, CLIP also tracks the number of code and data accesses to the L2 cache. If the ratio of code accesses and misses to total accesses and misses exceeds a given threshold, θ, CLIP dynamically modifies the re-reference interval for data requests. Specifically, if the ratio of L2 code accesses and misses exceeds θ (25% in our studies) the L2 cache follows CLIP for the non-sampled sets. If not, CLIP follows the baseline RRIP policy. In doing so, CLIP dynamically learns the code working set of workloads and allocates L2 capacity accordingly.

Note that CLIP sacrifices allocating data cache lines in the L2 cache and relies on out-of-order execution to overlap the increased cache latency with useful work. Applying CLIP at the LLC can be catastrophic since data requests would need to be serviced from main memory. We advocate CLIP for caches smaller than the LLC.

V. EXPERIMENTAL METHODOLOGY

A. *Performance Simulator*

We use CMP$im [19] a trace-based, detailed event-driven x86 simulator for our performance studies. The core parameters, cache hierarchy organization and latencies are loosely based on the Intel Core i7 processor [2]. We assume a 16-core system with a three-level cache hierarchy consisting of a shared LLC. Each core in our CMP is a 2.8 GHz 4-way out-of-order (OoO) processor with a 128-entry reorder buffer and

a decoupled front-end instruction fetch unit [32]. We assume single-threaded cores with the L1 and L2 caches private to each core. The baseline L1 instruction and data caches are 8-way 32KB each. The L1 cache sizes are kept constant in our study. We model two L1 read ports and one L1 write port on the data cache. We assume a banked LLC with as many banks as cores in the system. All caches in the hierarchy use a 64B line size. For replacement decisions, the L1 cache follows the Not Recently Used[3] (NRU) replacement policy while the L2 and LLC follow the RRIP replacement policy [20]. The cache lookup latencies for the L1, L2, and LLC bank are 3, 10, and 14 cycles respectively. Note that these are individual cache lookup latencies only and do not account for queuing delays and any network latency (e.g. load-to-use latency for the L2 is 13 cycles plus queuing delay while for the LLC is 27 cycles plus on-chip network latency and any queuing delay). On average, the LLC load-to-use latency is roughly 40 cycles.

We model multiple prefetchers in our system. First, a next line instruction prefetcher at the L1 instruction cache that prefetches the next sequential instruction cache line on both hits and misses. We also model a decoupled front-end that also enables branch-directed instruction prefetching. Next, we model a stream prefetcher which is configured to have 16 stream detectors. The stream prefetcher is located at the L2 cache and trains on L2 cache misses and prefetches lines directly into the L2 cache. We also model an adjacent line prefetcher (also located at the L2 cache) that trains on L2 cache misses and prefetches the corresponding adjacent line of a 128B cache block into the L2 cache. All three prefetchers are private to each core and are based on the Intel Core i7[2].

We assume a *ring*-based interconnect that connects the 16 cores and LLC banks on individual *ring stops*. We assume a single cycle between two successive *ring stops*. Bandwidth onto the interconnect is modeled using a fixed number of MSHRs. Contention for the MSHRs models the increase in latency due to additional traffic introduced into the system. We model 16 outstanding misses per core to main memory. We assume 16GB of memory distributed across four memory channels (12.8GB/s per channel). We model a virtual memory system with 4K pages and a random page mapping policy. We use a detailed DRAM model with DDR3-1600 11-11-11-28 timing parameters. Finally, we also assume a MESI cache coherence protocol and a snoop filter that is 4X the capacity of the smaller levels in the hierarchy.

3. NRU is a hardware approximation for LRU replacement that uses one bit per cache line and performs similar to LRU [20].

B. CMP Configurations Under Study

For all of our studies, we assume a 16-core CMP with a three-level cache hierarchy and a 16-bank shared LLC. The L1 and L2 caches are private to each core. We assume that manufacturing constraints only allow 2MB effective capacity per core. Under these assumptions we study the following cache hierarchy configurations:

- **256KB L2 (Baseline):** Like the Core i7 [2], we evaluate a 256KB L2 cache and a 32MB inclusive shared LLC. The effective on-chip caching capacity here is 32MB.
- **512KB L2:** Like the AMD Athlon [18, 11], we evaluate a 512KB L2 cache with an exclusive LLC. We reduce the LLC size to 24MB (1.5MB / core). Effectively, the on-chip capacity is still 32MB when the system is fully utilized. However, note that a single core can directly access and store data in the 24MB shared LLC but cannot store data in the 7.5MB of cache distributed into the private L2 caches of the other 15 cores. Based on CACTI measurements, we assume no latency impact of increasing the L2 cache.
- **1MB L2:** Like the AMD Opteron [1], we evaluate a 1MB L2 cache with an exclusive LLC. We reduce the LLC size to 16MB (1MB / core). Effectively, the on-chip capacity is still 32MB when the system is fully utilized. However, note that a single core can directly access and store data in the 16MB shared LLC but cannot store data in the other 15MB of cache distributed into the private L2 caches of the other 15 cores. Based on CACTI measurements, we assume a one cycle load-to-use latency increase for the larger 1MB L2 cache.

C. Workloads

For our study, we use all benchmarks from the SPEC CPU2006 suite and 12 server workloads. The SPEC benchmark was compiled using the *ICC* compiler with full optimization flags. Representative regions for the SPEC benchmarks were all collected using PinPoints [31]. A detailed cache sensitivity study of the SPEC benchmarks used in this study is available here [23].

The server workloads were collected using a hardware tracing platform on both Linux and Windows operating systems. The server workloads include both user level and system level activity. Table I lists the 12 server workloads and their misses per 1000 instructions (MPKI) in the L1, L2, and LLC when run in isolation. To illustrate application cache utility, the MPKI numbers are reported in the absence of a prefetcher. Furthermore, to provide insights on the working set of the server workloads, Figure 8 illustrates cache misses as a

TABLE I. MPKI of Server Workloads In the Absence of Prefetching (Baseline Inclusive Cache Hierarchy)

Code Name Actual Benchmark	mgs	tpch	gidx	ibuy	ncpr	ncps	sap	sas	sjap	sjbb	sweb	tpcc
IL1 MPKI (64KB)	3.37	2.53	0.05	12.16	1.32	5.22	12.60	0.01	15.46	6.51	6.08	27.35
DL1 MPKI (64KB)	3.02	0.91	2.21	5.71	7.32	6.00	7.23	47.96	5.18	11.27	4.43	17.34
UL2 iMPKI (256KB)	0.86	0.92	0.04	6.80	0.57	0.07	4.51	0.06	5.96	1.51	1.69	8.91
UL2 dMPKI (256KB)	1.44	0.31	2.07	3.47	5.26	0.30	4.19	18.26	3.27	8.58	1.63	9.11
UL3 iMPKI (32MB)	0.13	0.26	0.02	0.29	0.05	0.01	0.53	0.01	0.33	0.04	0.15	0.12
UL3 dMPKI (32MB)	0.62	0.17	2.05	0.46	0.54	0.08	1.66	16.70	0.66	2.51	0.56	1.28

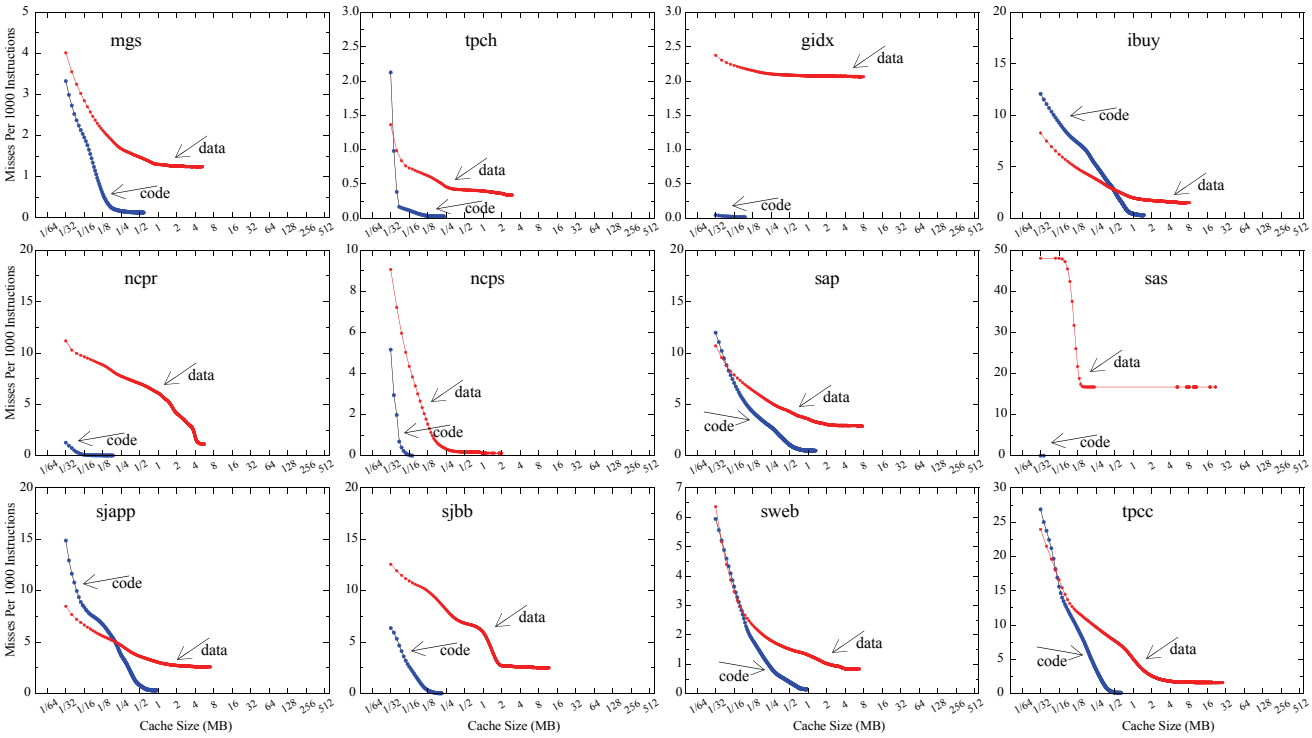


Figure 8: Cache Sensitivity Behavior of Server Workloads.

function of cache size for both *code* and *data* (both appropriately labeled). The figure shows server workloads have large code working set sizes.

To simulate a multi-core environment, we use multiple traces corresponding to different regions of the application. When the number of cores exceeds the number of traces, we replicate the traces across all cores in the system (offset by 1M instructions each). To simulate the sharing behavior of the instruction code footprint, we physically map code references into a shared memory region. The data footprint of each trace is physically mapped to a private memory region. While this approach ignores data sharing behavior, our methodology is limited by the absence of a full system infrastructure.

We simulate 500 million instructions for each benchmark. Our baseline 16-core study effectively evaluates the cache hierarchy over eight billion instructions. We verified that the simulated instruction count warms up all levels of the cache hierarchy. For the multi-core studies, simulations continue to execute until all benchmarks in the workload mix execute the required 500 million instructions. If a faster thread finishes its required instructions, it continues to execute to compete for cache resources. We only collect statistics for the first 500 million instructions committed by each application. This methodology is similar to existing work on shared cache management [20, 21].

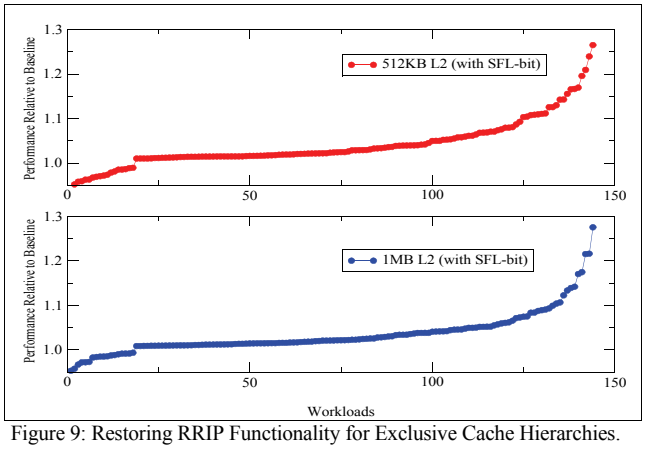


Figure 9: Restoring RRIP Functionality for Exclusive Cache Hierarchies.

VI. Results and Analysis

A. *Restoring RRIP Functionality*

We showed that transitioning from inclusive to exclusive caches breaks RRIP functionality. Figure 9 illustrates the performance improvements of restoring RRIP functionality by introducing the SFL-bit in the L2 caches. For a set of cache sensitive 1-core, 2-core, 4-core, 8-core, and 16-core workloads comprised of SPEC CPU2006 and server workloads, we show the performance improvements of using the SFL-bit. The figure shows two graphs, the top graph for a 512KB L2/core and 24MB shared LLC exclusive hierarchy while the bottom graph is for a 1MB L2/core and 16MB shared LLC exclusive hierarchy. The x-axis represents the workloads while the y-axis illustrates performance compared to the baseline exclusive hierarchy that does not use the SFL-bit. Using the SFL-bit restores RRIP performance by as much as 30% with as little as 5% performance degradation. The performance degradations are due RRIP side-effects of incorrectly predicting cache line re-reference interval [20].

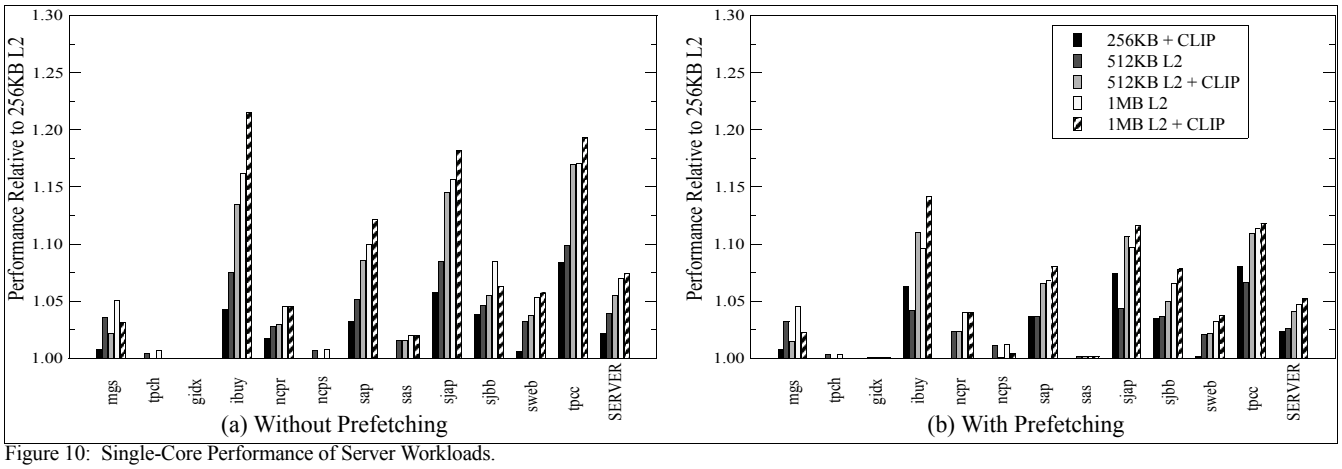


Figure 10: Single-Core Performance of Server Workloads.

B. *Single-Core Study*

We now compare our baseline inclusive cache hierarchy to CLIP and the two exclusive hierarchies with 512KB and 1MB L2 caches. In this subsection, we assume only one active core in our baseline 16-core processor system (the remaining are 15 assumed idle). Furthermore, we assume that the cache access latencies in all three cache hierarchies are identical.

B.1 System Performance

Figure 10 illustrates the performance of the three hierarchies both in the presence and absence of CLIP in the L2 cache. The x-axis shows the different workloads and the y-axis illustrates the performance compared to our baseline inclusive hierarchy with a 256KB L2 cache. The figure shows that simply applying CLIP to the baseline 256KB L2 cache improves performance of five of the 12 server workloads by 5-10% both in the presence and absence of prefetching. The benefits of CLIP reduce in the presence of prefetching because useful code prefetches hide latency. Where CLIP helps, we see that CLIP provides nearly the same performance as doubling the L2 cache size. This clearly indicates the criticality of code lines compared to data lines in the L2 caches. However, for many workloads (e.g., *mgs, ibuy, sap, sjap, sjbb, sweb, tpcc*), 256KB+CLIP still leaves significant performance potential compared to a 512KB or 1MB L2. This suggests opportunity for further research to identify critical code cache lines and preserve them in a small (e.g. 256KB) L2 cache.

B.2 L2 Cache Performance

To correlate the performance improvements of CLIP and increasing L2 cache size, Figure 11 presents the misses per 1000 instructions (MPKI) for the server workloads. For the remainder of the paper, we only illustrate system behavior in the presence of prefetching and observe similar behavior without prefetching. Figure 11a illustrates the front-end misses in the L2 cache while Figure 11b illustrates the total misses in the L2 cache (i.e., combined front-end and back-end misses). As expected, the figure shows that the server workloads with the largest front-end MPKI benefit the most from CLIP and increasing L2 cache sizes. From Figure 11a, CLIP performance can be correlated directly to the large reductions in L2 front-end MPKI. Note that while CLIP helps reduce front-end cache misses, it does so at the expense of increasing L2 cache misses (see Figure 11b). This is to be expected because CLIP prioritizes front-end code requests over data requests. Note that even though CLIP increases total L2 cache misses, overall system performance improves significantly. This clearly illustrates the criticality of code requests over data requests and the importance of avoiding hiccups in the processor front-end.

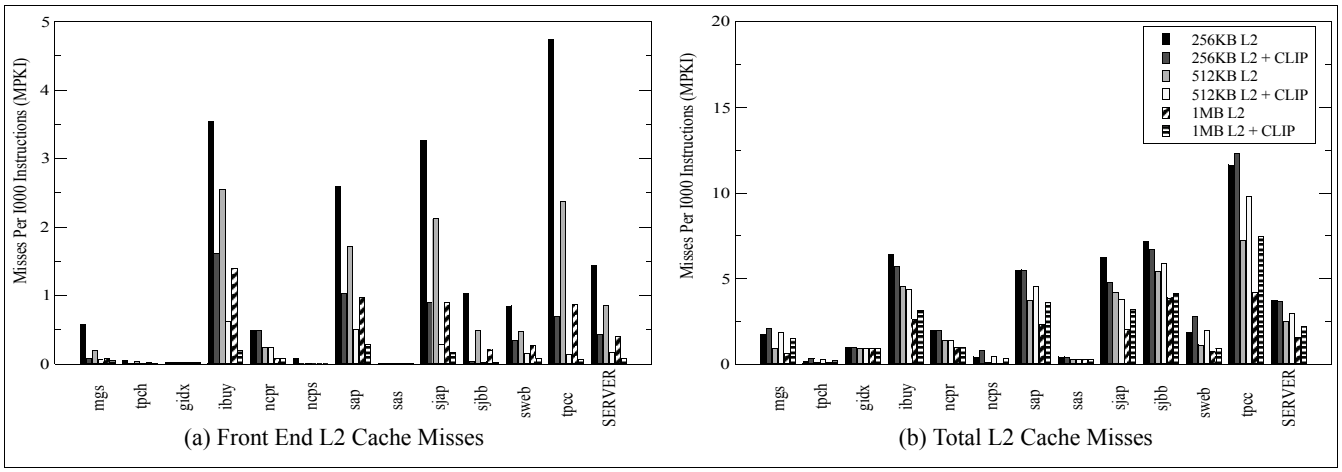


Figure 11: Comparison of L2 Cache Misses for Server Workloads (Prefetching Enabled).

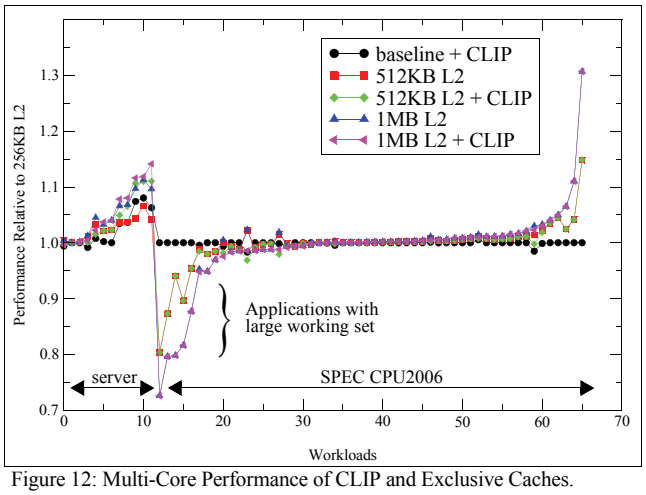


Figure 12: Multi-Core Performance of CLIP and Exclusive Caches.

C. *Multi-Core Study*

Increasing the private L2 cache size impacts system performance and cache hierarchy performance.

C.1 System Performance

Figure 12 characterizes the performance of CLIP and the two exclusive hierarchies for server and SPEC CPU2006 workloads. The x-axis shows the different workloads while the y-axis shows performance compared to the baseline inclusive hierarchy with 256KB L2 cache. The server workloads represent a 16-core configuration while the SPEC benchmarks are a combination of single-core, two-core, four-core, eight-core and 16-core workload mixes. The figure shows an "s-curve" with the workloads sorted based on 1MB L2+CLIP performance.

The figure shows that increasing the L2 cache and reducing the LLC has both positive and negative outliers for SPEC CPU2006 workloads. The positive outliers are for workloads that benefit from improved L2 cache access latency when the working set fits in the L2 cache. The negative outliers are for those workloads whose working set fits in the baseline large shared LLC but no longer fit in the exclusive hierarchy with smaller shared LLC. The outliers can be as significant as 30% performance degradation for a single-core run of *libquantum.* This is to be expected as *libquantum* has a 32MB working-set that fit nicely in the baseline inclusive hierarchy. On the other hand, *calculix* observes a 30% performance improvement because its working set was larger than the baseline 256KB L2 cache but fits nicely into a 1MB L2 cache. The bimodal performance behavior of SPEC CPU2006 workloads provides no clear indication on whether or not to increase the L2 cache size. However, server workloads clearly show the need for a larger L2 cache. Thus, this provides avenue for academic research work on a general solution that would be applicable to both workload categories.

C.2 Cache Hierarchy Performance

In a multi-core system, instruction working set duplication in private caches can reduce the effective caching capacity of an exclusive cache hierarchy (relative to the baseline inclusive cache hierarchy). Assuming a shared code working set of 1MB, an exclusive hierarchy with 1MB private L2 caches and 16MB LLC devotes only 15MB of the total 32MB cache hierarchy capacity for the data working set[4]. The baseline inclusive hierarchy on the other hand devotes 31MB of cache hierarchy capacity for the data working set and 1MB for the instruction working-set (preserved in the LLC)[5]. This reduction in effective caching capacity in the hierarchy for the data working can increase data traffic to memory. In fact, we see that server workloads such as *mgs, ibuy, sap, sjap, sjbb, sweb,* and *tpcc* observe a 10-20% increase in memory traffic (data not shown due to space constraints). This excess in memory traffic does not degrade memory performance since the processor front-end enables the back-end to exploit memory-level parallelism without too many front-end stalls. Nonetheless, we present yet another avenue for academic research work to improve exclusive cache hierarchy performance by reducing duplication in the private L2 caches.

VII. RELATED WORK

Several studies have focused on improving the memory system performance of commercial workloads. Barroso et al. presented a detailed characterization of commercial workloads and showed that they have different requirements than scientific and engineering workloads [8, 17]. Hardavellas et al. performed a similar study of database workloads and confirm our findings for a two-level cache hierarchy [17]. Other commercial workload studies have looked at improving prefetching. Ferdman et al. focused on improving instruction cache performance of commercial workloads by improving instruction prefetchers [13, 14]. Wenisch et al. focus on improving data cache performance by improving data prefetchers [34, 35]. Our studies do not include these high performing prefetchers because of their design complexity to be adopted into commercial processors.

A number of studies have attempted to address the overheads of shared cache latency by advocating private cache implementations [9, 10, 30, 12, 28]. A number of these studies recognize that private caches can significantly degrade performance compared to shared caches. As such there have been proposals to *spill* lines into neighboring private caches to achieve the capacity benefits of shared caches [9, 10, 30]. In a three-level hierarchy, these techniques still suffer from private cache access latency when the working set is larger than the L2 cache. Furthermore, the proposed *spill* techniques incur replication overhead for shared data in the private LLCs. Our proposals allow room for avoiding replication overhead because of the shared LLC.

Several studies advocate exclusive caches to increase the effective caching capacity of the hierarchy. Jouppi et al first proposed exclusive caches to reduce LLC conflict misses and to also increase the effective cache capacity of the hierarchy by not replicating lines in the LLC [24, 25]. Others show that

4. In an exclusive hierarchy with 1MB private L2s, the 1MB instruction working is duplicated in each of the 16 private L2s and an additional 1MB duplicated in the LLC (for quick access to shared data).
5. In the baseline inclusive hierarchy, the instruction working set is larger than the 256KB L2. This working set is always serviced by the LLC.

exclusive hierarchies are useful when the core caches are not significantly larger than the size of the LLC [22, 33, 40]. In addition to increasing caching capacity, we also note that exclusive caches can improve average cache access latency by using larger private L2 caches and smaller shared LLCs.

There has been extensive research on cache management [16, 20, 21, 29, 36, 37, 39] and improving the performance of cache hierarchies [26, 38, 15, 33, 22, 39]. The work most related to our exclusive cache management work is by Gaur et al [16]. The SFL-bit proposal simplifies the implementation complexity of Gaur et al while providing nearly identical performance. The work most relevant to our Code Line Preservation (CLIP) proposal is PACMan [37]. PACMan distinguishes between prefetch and demand requests while CLIP distinguishes between code and data requests.

VIII. SUMMARY AND FUTURE WORK

Process scaling and increasing transistor density have enabled a significant fraction of on-die real estate to be devoted to caches. The widening gap between processor and memory speeds has forced processor architects to devote more on-die space to the shared last-level cache (LLC). As a result, the memory latency bottleneck has shifted from memory access latency to shared cache access latency. As such, server workloads whose working set is larger than the smaller caches in the hierarchy spend a large fraction of their execution time on shared cache access latency.

To address the shared cache latency problem, this paper proposes to improve server workload performance by increasing the size of the smaller private caches in the hierarchy as opposed to increasing the shared LLC. Doing so improves average cache access latency for workloads whose working set fits into the larger private cache while retaining the benefits of a shared LLC. The trade-off requires relaxing the inclusion requirements and transition to an exclusive hierarchy. For the same caching capacity, the exclusive hierarchy provides better cache access latency than an inclusive hierarchy with a large shared LLC.

Exclusive hierarchies relax the cache design space and provide opportunity to explore cache hierarchies that can range from a private cache only hierarchy [28] to small shared LLC clusters. However, exclusive hierarchies must efficiently manage and utilize total on-die cache space. We observed that shared data duplication in private caches can significantly reduce the effective caching capacity of the proposed exclusive hierarchy relative to the baseline inclusive cache hierarchy. Continued research to improve exclusive cache hierarchy performance would greatly benefit industry.

ACKNOWLEDGEMENTS

The authors would like to thank Zeshan Chishti, Kevin Lepak, Sudhanva Gurumurthi, Moinuddin Qureshi, Carole-Jean Wu, Mohammed Zahran, and the anonymous reviewers for their feedback in improving the quality of this paper.

REFERENCES

[1] http://www.amd.com/us/products/server/processors/6000-series-platform/6300/Pages/6300-series-processors.aspx#2
[2] Intel. Intel Core i7 Processor. http://www.intel.com/products/processor/corei7/specifications.htm
[3] http://blog.stuffedcow.net/2013/01/ivb-cache-replacement/
[4] http://www.hotchips.org/wp-content/uploads/hc_archives/hc24/HC24-1-Microprocessor/HC24.28.117-HotChips_IvyBridge_Power_04.pdf
[5] http://www.jilp.org/dpc/ (Commercial workload targeted prefetcher was not in the top three performing prefetchers across several server, multimedia, games, scientific, and engineering workloads.)
[6] http://www.hpl.hp.com/research/cacti/
[7] J. L. Baer and W. Wang. "On the Inclusion Properties for Multi-level Cache Hierarchies." In ISCA, 1988.
[8] L. Barroso, K. Gharachorloo, E. Bugnion. "Memory System Characterization of Commercial Workloads" In ISCA, 1998.
[9] B. Beckmann, M. Marty, D. Wood. "ASR: Adaptive Selection Replication for CMP Caches." In MICRO, 2006.
[10] J. Chang and G. S. Sohi. "Cooperative Caching for Chip Multiprocessors". In ISCA'2006.
[11] P. Conway, N. Kalyanasundharam, K. Lepak, and B. Hughes. "Cache Hierarchy and Memory Subsystem of the AMD Opteron Processor". In IEEE Micro, April 2010.
[12] M. Ferdman et al. "Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware." In ASPLOS 2012.
[13] M. Ferdman, C. Kaynak, and B. Falsafi. "Proactive Instruction Fetch." In MICRO'2011.
[14] M. Ferdman, T. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos. "Temporal Instruction Fetch Streaming." In MICRO 2008.
[15] K. Fletcher, W. E. Speight, and J. K. Bennett. "Techniques for Reducing the Impact of Inclusion in Shared Network Cache Multiprocessors." Rice ELEC TR 9413, 1995.
[16] J. Gaur, M. Chaudhuri, and S. Subramoney. "Bypass and Insertion Algorithms for Exclusive Last-Level Caches". In ISCA 2011.
[17] N. Hardavellas et al. "Database servers on chip multiprocessors: limitations & opportunities". Innovative Data Systems Research, Jan'07.
[18] J. Huynh. "The AMD Athlon XP Processor with 512KB L2 Cache". White Paper, 2003.
[19] A. Jaleel, R. Cohn, C. K. Luk, B. Jacob. CMP$im: A Pin-Based On-The-Fly Multi-Core Cache Simulator. In MoBS, 2008.
[20] A. Jaleel, K. Theobald, S. Steely, and J. Emer. "High Performance Cache Replacement Using Re-Reference Interval Prediction (RRIP)". ISCA'10.
[21] A. Jaleel, W. Hasenplaugh, M. K. Qureshi, J. Sebot, S. Steely, and J. Emer. "Adaptive Insertion Policies for Managing Shared Caches". In PACT, 2008.
[22] A. Jaleel, E. Borch, M. Bhandaru, S. Steely, and J. Emer. "Achieving Non-Inclusive Cache Performance with Inclusive Caches -- Temporal Locality Aware (TLA) Cache Management Policies". In MICRO 2010.
[23] A. Jaleel. "Memory characterization of workloads using instrumentation-driven simulation–a pin-based memory characterization of the SPEC CPU2000 and SPEC CPU2006 benchmark suites". Intel Corporation, VSSAD, 2007 (www.jaleels.org/ajaleel/).
[24] N. Jouppi, and S. E. Wilton. "Tradeoffs in two-level on-chip caching." ISCA'94
[25] N. P. Jouppi. "Improving direct-mapped cache performance by the addition of a fully associative cache and prefetch buffers." ISCA, 1990.
[26] M. J. Mayfield, T. H. Nguyen, R. J. Reese, and M. T. Vaden. "Modified L1/L2 cache inclusion for aggressive prefetch." U. S. Patent 5740399.
[27] S. McFarling. "Cache Replacement with Dynamic Exclusion." ISCA'92.
[28] P. L-Kamran, et al. "Scale-Out Processors". In ISCA'12.
[29] M. K. Qureshi, A. Jaleel, Y. Patt, S. Steely, J. Emer. "Adaptive Insertion Policies for High Performance Caching." In ISCA 2006.
[30] M. K. Qureshi. "Adaptive Spill-Receive for Robust High-Performance Caching in CMPs". In HPCA'2009.
[31] H. Patil, R. Cohn, M. Charney, R. Kapoor, and A. Sun. "Pinpointing Representative Portions of Large Intel Itanium Programs with Dynamic Instrumentation." In MICRO 2004.
[32] G. Reinman, B. Calder, and T. Austin. "Optimizations Enabled by a Decoupled Front-End Architecture." In TOCS, 2001.
[33] J. Sim, et al. "FLEXclusion: Balancing Cache Capacity and On-Chip Bandwidth via Flexible Exclusion." In ISCA'12.
[34] T. Wenisch, M. Ferdman, A. Ailamaki, B. Falsafi, and A. Moshovos. "Temporal Streams in Commercial Server Applications." In IISWC'08.
[35] T. Wenisch, et al. "Practical Off-Chip Meta-Data for Temporal Memory Streaming." In HPCA'09.
[36] C. Wu, A. Jaleel, W. Hasenplaugh, M. Martonosi, S. Steely, and J. Emer. "SHiP: Signature-based Hit Predictor for High Performance Caching." In MICRO 2011.
[37] C. Wu, A. Jaleel, M. Martonosi, S. Steely, and J. Emer. "PACMan: Prefetch-Aware Cache Management for High Performance Caching." In MICRO 2011.
[38] M. Zahran. "Non-inclusion property in multi-level caches revisited.", in IJCA'07, 2007.
[39] M. Zahran. "Cache Replacement Policy Revisited.", WDDD, 2007.
[40] Y. Zheng, B. T. Davis, and M. Jordan. "Performance Evaluation of Exclusive Cache Hierarchies." In ISPASS, 2004.