2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture

Criticality Aware Tiered Cache Hierarchy: A Fundamental Relook at Multi-level Cache Hierarchies

Anant Vithal Nori*, Jayesh Gaur*, Siddharth Rai†, Sreenivas Subramoney* and Hong Wang*

*Microarchitecture Research Lab, Intel
{anant.v.nori, jayesh.gaur, sreenivas.subramoney, hong.wang}@intel.com
†Indian Institute of Technology Kanpur, India
sidrai@cse.iitk.ac.in

***Abstract*—On-die caches are a popular method to help hide the main memory latency. However, it is difficult to build large caches without substantially increasing their access latency, which in turn hurts performance. To overcome this difficulty, on-die caches are typically built as a multi-level cache hierarchy. One such popular hierarchy that has been adopted by modern microprocessors is the three level cache hierarchy. Building a three level cache hierarchy enables a low average hit latency since most requests are serviced from faster inner level caches. This has motivated recent microprocessors to deploy large level-2 (L2) caches that can help further reduce the average hit latency.**

In this paper, we do a fundamental analysis of the popular three level cache hierarchy and understand its performance delivery using program criticality. Through our detailed analysis we show that the current trend of increasing L2 cache sizes to reduce average hit latency is, in fact, an inefficient design choice. We instead propose Criticality Aware Tiered Cache Hierarchy (CATCH) that utilizes an accurate detection of program criticality in hardware and using a novel set of inter-cache prefetchers ensures that on-die data accesses that lie on the critical path of execution are served at the latency of the fastest level-1 (L1) cache. The last level cache (LLC) serves the purpose of reducing slow memory accesses, thereby making the large L2 cache redundant for most applications. The area saved by eliminating the L2 cache can then be used to create more efficient processor configurations. Our simulation results show that CATCH outperforms the three level cache hierarchy with a large 1 MB L2 and exclusive LLC by an average of 8.4%, and a baseline with 256 KB L2 and inclusive LLC by 10.3%. We also show that CATCH enables a powerful framework to explore broad chip-level area, performance and power trade-offs in cache hierarchy design. Supported by CATCH, we evaluate radical architecture directions such as eliminating the L2 altogether and show that such architectures can yield 4.5% performance gain over the baseline at nearly 30% lesser area or improve the performance by 7.3% at the same area while reducing energy consumption by 11%.

***Keywords*-Criticality, Caching, Prefetching**

I. Introduction

Large on-die caches can help hide the main memory latency and allow processors to scale the memory wall. However, it is difficult to build large caches that are simultaneously fast enough to match the fast processor cycle time and large enough to effectively hide the slow memory latency [23]. One popular solution is the three level cache hierarchy used by many modern microprocessors [11], [12], [30]. Smaller and faster caches, namely the level-1 (L1) and level-2 (L2) caches, are private and kept close to the CPU core, whereas a large last level cache(LLC), shared across cores, is used to hide the slow main memory accesses. Latency of a load hit in the large LLC is significantly higher than the latency of hits in the smaller L1 and L2 caches [10]. However, since most requests are serviced by the faster inner level caches (L1 and L2), the overall average latency for a cache hit in this cache hierarchy is still low. This has motivated the trend towards large L2 caches as is seen in recent microprocessor offerings [11], [30].

In this paper we do a fundamental analysis of this popular three level cache hierarchy and show that this trend of increasing L2 sizes is in fact an inefficient design choice. We leverage the well known notion that not all load accesses matter for core performance, and only those load accesses that lie on the critical path of execution can effect the core performance [1], [2], [4], [6]. Hence, current cache hierarchies that optimize for average load hit latency of all load accesses are clearly sub-optimal. To gain performance, the *critical loads* need to be served at the least possible latency whereas load accesses that are non-critical can be served at a slightly higher latency. We hence propose Criticality Aware Tiered Cache Hierarchy (CATCH) that utilizes an accurate detection of program criticality in hardware and using a novel set of inter-cache prefetchers, ensures that data accesses that lie on the critical path of execution are served from the L1 cache at the least possible latency. Specifically we make the following contributions:

1) We first do a detailed analysis of the three level cache hierarchy and develop an understanding of the performance delivered from this hierarchy using program criticality. We show through oracle studies how criticality can be used as a central consideration

† The author contributed to the work as an intern with Microarchitecture Research Lab (MRL), Intel India

2575-713X/18/$31.00 ©2018 IEEE
DOI 10.1109/ISCA.2018.00019

for achieving an efficient on-die cache hierarchy and how each level of such a cache hierarchy should be optimized.

2) We propose, and describe in detail, a novel and fast incremental method to learn the critical path using an optimized representation of the data dependency graph first proposed in the seminal work by Fields et al. [1] in hardware, that takes just 3 KB of area. We use this to enumerate a small set of critical load instructions.
3) We then propose the Timeliness Aware and Criticality Triggered (TACT) family of prefetchers for these identified critical loads. The TACT prefetchers prefetch data lines accessed by the critical load PCs from the L2 or the LLC to the L1 cache. TACT utilizes the association between the address or data of load instructions in the vicinity of the target critical load to issue prefetches which bring the data from the LLC or L2 into L1, just before the actual load access is issued by the core. TACT also proposes a code runahead prefetcher that eliminates code stalls because of code L1 miss. We should note that unlike traditional prefetchers [21], [39] that target LLC misses, TACT is a unique inter-cache prefetcher which tries to hide the latency of L2 and LLC for a select subset of critical load and code accesses that are otherwise served in the baseline from the slower outer level caches.
4) With TACT prefetchers incorporated into the design we demonstrate that most critical load accesses that would have hit in the outer level caches are served by the L1 cache. TACT-based critical path accelerations fundamentally enable the Criticality Aware Tiered Cache Hierarchy (CATCH). CATCH represents a powerful framework to explore broad chip-level area, performance and power trade-offs in cache hierarchy design. Supported by CATCH, we explore and analyze radical directions such as eliminating the L2 altogether to dramatically reduce area/cost and enable higher-performing CPUs at similar or lower power.

Our detailed results show that CATCH improves the baseline three level cache hierarchy of a state-of-the-art Skylake-like processor [30], that uses a large 1 MB L2 and an exclusive 1.375 MB LLC per core, by 8.4% for seventy single thread (ST) applications chosen from a wide category of workloads. CATCH also improves the performance for 60 4-way multi-programmed (MP) workloads by 8.9%. Interestingly, a CATCH-based two level cache hierarchy without the L2 delivers 4.5% higher performance for ST workloads at 30% lower die area. This die area savings can be used to increase the overall LLC capacity, yielding a 7.25% gain over the baseline and at 11% lower energy. CATCH improves performance of an inclusive LLC baseline with 256KB L2 by 10.3%. We also show that CATCH delivers large performance, power and area benefits across different implementations of the cache hierarchy, and marks a fundamental shift in our understanding of how on-die caches should be designed going forward.

II. Background

Three level cache hierarchy has been in popular use in modern processors [11], [12], [30]. For example, the Intel Skylake client processors use a 32KB L1 instruction cache, a 32 KB L1 data cache, a 256 KB L2 cache and an 8 MB LLC shared across four cores [12]. The L2 is not inclusive of L1, with allocations in L2 on L1 misses but no back-invalidates to L1 on L2 evictions. The LLC is strictly inclusive, with every cache line present in the L1 or L2 cache also present in the LLC. On an LLC eviction, back-invalidates will snoop out lines in the L1 and L2 to guarantee inclusion [16].

Because of its large capacity, the LLC has a significantly higher access latency than the L1 and L2 caches [10]. However, since most load accesses will be served by the L1 and L2 caches, the overall average latency will still be low. This has motivated the trend towards large L2 caches. Cache hierarchies used by recent AMD processors [11] and the Skylake Server processors from Intel [30] deploy a large L2 cache per core. To prevent the problem of duplication of cache lines in the L2 and the LLC, the LLC is made exclusive of the L2 and L1. A cache hit or miss in the LLC is filled in the L2 and an eviction from the L2 is filled in the LLC. A large L2 helps reduce the overall average latency of the on-die cache hits. Additionally, it helps applications with large code footprints and reduces stalls in the core front-end because of code lines that would need to be read from the slower LLC [19]. Moreover, the large L2 filters requests that would need to travel on the interconnect to the LLC, thereby saving power.

To understand the importance of this design point we simulate an experiment where the L2 is removed for ST workloads running on a single core[1]. The baseline for this experiment is a three level cache hierarchy, similar to a state-of-the-art Skylake-like Server processor [30], that has a 32 KB L1 cache for code, 32 KB L1 cache for data, 1 MB L2 cache and a 1.375 MB exclusive LLC per core, which is modeled as a 5.5 MB exclusive LLC shared across four cores. To keep the cache capacity the same for ST workloads, we increase the LLC by 1 MB when we remove the L2 cache ("noL2 + 6.5MB LLC configuration"). When the L2 is removed for each core, it frees up 4 MB of area for a four core system. Therefore, we also plot an iso-area configuration where we add 4 MB to the LLC size and increase it to 9.5 MB. Figure 1 shows the results for these experiments.

Figure 1 shows a significant 7.8% drop in performance on removing the L2. Interestingly, the iso-area configuration

[1]Simulation parameters and workloads are explained in Section V

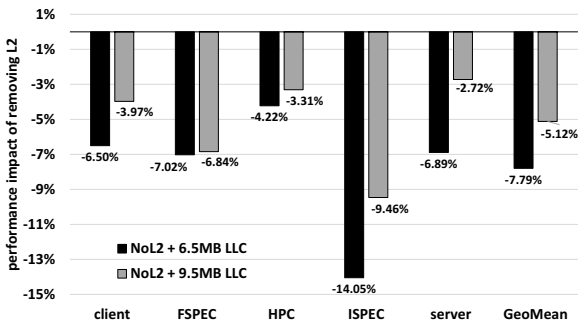

Figure 1. Performance impact of removing L2

that removes the L2 from all cores and adds it to the LLC, still loses 5.1% as compared to the baseline. This demonstrates the superior performance of the three level cache hierarchy and suggests that growing the L2 capacity, rather than increasing the LLC size, is more beneficial for performance. This explains the recent trend towards large L2 sizes in modern processors [11], [30] .

However, a close examination shows that this hierarchy is actually inefficient for area and power. For the inclusive cache hierarchies, cache area is wasted in creating a low latency L2 cache which is essentially replicating some of the data already present in the LLC. Making the LLC exclusive, can help prevent this wastage and hence allow a larger L2 cache. However, as the L2 cache is private to each core, the overall cache capacity seen by each core (L2 + shared LLC) is lower than what they would have seen with a large shared LLC. Having a large shared LLC is beneficial for single thread applications as well as when applications with disparate cache footprint requirements are run together [24].

Three level cache hierarchies also often replicate code in the L2 when symmetric processes run on many cores. Similar replication is also done for shared, read-only data [45]. Additionally, moving to an exclusive LLC also requires a separate snoop filter or coherence directory [25] that also adds area. All of these lead to large chip area redundancies making three level cache hierarchies an inefficient design choice for area and power. However, results in Figure 1 show that minimizing the average hit latency through a large mid-level L2 cache gives a significant performance boost. This significant performance edge - albeit at a large area and power overhead - is the primary reason for the widespread prevalence of this hierarchy in modern microprocessors.

A. *Program criticality*

The performance of the Out of Order (OOO) core is bound by the critical path of execution. Criticality can be described with the program's data dependency graph (DDG) first proposed in the seminal work by Fields et al. [1]. There exists one or more paths through the graph with a maximum weighted length and such a path is called a critical path. Any instruction that appears in this path (or paths) is critical.

Figure 2 shows an example of a data dependency graph (DDG) for a sample application. We use the notation used in [1] to create the DDG. Each instruction in the DDG has three nodes. The D node denotes allocation into the OOO, the E node denotes the dispatch of the instruction to the execution nodes, and the C node denotes the writeback of the instruction. An E-E edge denotes data dependency, C-C edges denote in-order commit and D-D nodes are for in-order allocation into the OOO [1].

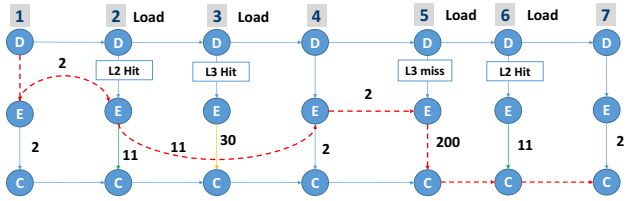

Figure 2. An example Data Dependency Graph (DDG) based on [1]

The critical path of execution for this sample application has been marked as a dashed line. Instructions 1, 2, 4 and 5 are on the critical path whereas the remaining instructions are not on the critical path. There are three loads that hit in the L2 or LLC in this example DDG, namely instructions 2, 3 and 6. Out of these three, only load instruction 2 is on the critical path. As is evident in the graph, if the latency of the non-critical L2 hits (11 cycles) is increased to LLC hit latency (30 cycles), the critical path of execution will remain the same. Likewise, if critical load instruction 2 is made a hit in the L1, the overall performance will improve as the critical path is shortened. This clearly shows that the performance benefit of the L2 cache is primarily coming from the critical loads that are hitting in the L2 cache. However, we should note that criticality is not just a function of the application being executed, but also of the underlying hardware that decides the execution latency of each instruction. For example, if load instruction 6, which hits in the L2 and is not critical, starts missing the LLC because of policy changes, the critical path of the machine may change. Hence it is imperative to determine the critical path dynamically while the application is executing. In Section IV-A we will describe a light-weight hardware mechanism to accurately determine critical instructions on the fly in hardware.

III. Motivation

In this section we will use program criticality to develop an understanding of how the three level cache hierarchy delivers performance. Through this analysis we will motivate the need for a Criticality Aware Tiered Cache Hierarchy.

A. *Latency sensitivity*

Figure 3 shows the sensitivity to performance when adding one, two and three cycles to the L1, L2 and LLC

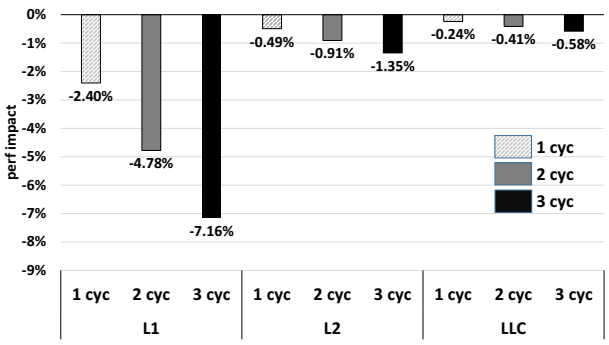

Figure 3. Impact of latency increase in L1, L2 and LLC

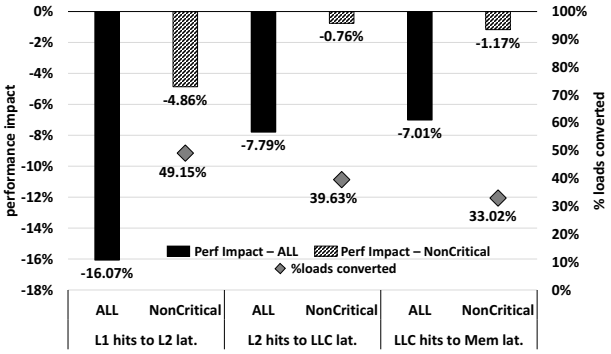

Figure 4. Impact of increasing non-critical load latency

of the baseline three level cache hierarchy, that has a 32 KB L1, 1 MB L2 and a shared 5.5 MB exclusive LLC, similar to the Skylake processors [30]. Adding three cycles to the L1 results in a 7.15% loss in performance, with lower performance sensitivity as we move to outer level caches.

To understand the high L1 sensitivity, we again refer to the DDG description of Section II-A. The critical path is the longest path from the D node of the first instruction to the C node of the last instruction retired. With an OOO core depth of 224 and a dispatch width of 4 instructions per cycle, the OOO takes a minimum of 56 cycles to dispatch all instructions. This path is represented by the D-D chain in the DDG. Since on-die caches typically have latencies less than 56 cycles, a single access to any cache cannot create critical paths. Most critical paths in the OOO are created by long latency LLC misses (memory accesses), a branch miss-speculation or by a long chain of dependent instructions whose latencies add up to a length of path greater than what the OOO depth can hide. Any instruction that a long latency LLC miss or a mis-speculating branch is data dependent on, will lie on the critical path of execution and hence will impact performance. Since L1 hits are *the most frequent*, they tend to also occur frequently on the chain of instructions that lead to the load miss or branch mis-prediction. This explains why the sensitivity to L1 latency is higher than the L2 or L3 latency.

B. Criticality and cache hierarchy

To understand which level in the cache hierarchy is more amenable to criticality optimizations, we devise a set of oracle studies. Since the critical path changes dynamically, we need a method to enumerate critical instructions on the fly while the application is executing. We hence use our hardware mechanism, that we will describe in Section IV-A, to determine dynamically the critical load instructions at a given level of the cache hierarchy.

Criticality at L1: Figure 4 shows the performance impact of converting L1 hits to L2 hits. Converting *all* L1 hits to L2 hits sees a huge 16% drop in performance. Using criticality information and converting only the *non-critical* L1 hits to L2 hits helps mitigate the performance loss from 16% to a still fairly big 4.8% on average. We calculated that an OOO core with depth of 224 and dispatch width of four can hide 56 cycles of latency. Let us assume we have a small chain of dependent instructions which have 8 load instructions, all of which hit the L1 at a latency of 5 cycles. The overall length of this dependent chain (40 cycles) is much smaller than 56 and hence these instructions are not critical. However, if we increase the latency of just four of these L1 requests to 13 cycles (the L2 hit latency), this path will have an overall weight of more than 72 cycles, and will hence create a new critical path. Such small dependence chains are very frequent in applications, but a large OOO depth can easily hide their latency. However, if the latency of these instructions is increased, new critical paths will be created dynamically in the OOO. Since L1 hits are the most frequent (we observed an average 85% L1 hit rate on our study list), small dependence chains that were non-critical will now create new critical paths. Hence, criticality based optimizations at the L1 are very challenging to implement.

Criticality at L2: Figure 4 also shows similar studies for the L2 cache. Converting all L2 hits to LLC hits results in a loss of 7.8%. However, converting all non critical L2 hits to LLC hits results in a much smaller 0.76% loss in performance. Since L2 hits are more infrequent than the L1 hits, increasing their latency has lesser likelihood of creating new critical paths. Hence we surmise that the L2 cache is a very good candidate for criticality optimization.

Criticality at the LLC: Finally we evaluate criticality at the LLC. Figure 4 shows that if we remove the LLC, the performance drops by a 7%. Our criticality detection mechanism flags 33% of all LLC hits as not critical and serving these at memory latency still loses 1.2% performance on average. This is roughly a linear scaling from the losses when all LLC hits (100%) are converted to LLC misses. This is simply because an LLC miss is served by the long latency memory (about 200 cycles of latency). This would mean that an LLC miss will have a very high likelihood of creating new critical paths and hence it is imperative that LLC policies be designed to prevent such long latency LLC

misses. A similar observation was put forward by Srinivasan et al. [5].

To summarize, we see that targeting the L1 or LLC will create new critical paths and can be detrimental to performance. However, the L2 is an ideal candidate for criticality optimization. This is especially important due to the recent trend towards large L2 caches [11], [30] which are clearly inefficient in terms of area as reasoned in Section II-A. The focus should be on serving critical loads that hit the L2 or LLC, to instead serve from the L1, optimize the L1 for latency and bandwidth and grow the LLC to further reduce memory misses. This forms the motivation for Criticality Aware Tiered Cache Hierarchy (CATCH).

C. CATCH - Performance potential

The goal of CATCH is to convert critical loads that hit in the on-die L2 or LLC into L1 hits. Speeding up these on-die loads will speed up the critical path translating into performance. This can be done through criticality-aware inter-cache prefetchers.

To understand the headroom for such prefetching, we design an oracle on-die prefetcher to convert L2/LLC hits to L1 hits. We use our hardware mechanism, that we describe in Section IV-A, to dynamically determine the critical load instructions that often hit in the L2 and LLC. On every instance of such a load instruction missing the L1, we query the L2 and the LLC. If the load would hit in the L2 or the LLC, we do a zero time prefetch for the 64B cacheline and fill in the L1. Evicted victims of these fill, if dirty, are written back to the L2. Additionally, all code lines are assumed to hit in the L1 instruction cache. For these studies, we turn off the hardware prefetchers, since training them in the presence of the oracle prefetcher is complicated.

Figure 5 shows the results of this oracle for a 1 MB L2 and 5.5 MB exclusive LLC, for ST workloads. We sweep the number of tracked critical load instructions in the oracle. Tracking 32 critical load instruction addresses (also called program counter or PC), nets a performance gain of 5.5% while converting just 14% of L1 misses in the baseline to L1 hits. The gains steadily increase with increasing the number of tracked load PCs. Importantly, we note that tracking a sufficient number of critical load PCs and accelerating just a fraction of L1 misses (17%) yields nearly the same performance as accelerating all load L1 misses from L2 or LLC irrespective of criticality (6.1% vs 6.6%). Under such an optimization, the L2 cache becomes irrelevant. We further validate this with one additional design point. From Section III-A, we saw that the no-L2 configuration loses 8.6% performance. However, with the oracle prefetcher in play, the last bar in Figure 5 shows the configurations with L2 or without L2 yield almost the same performance improvement.

The large L2 cache adds substantial area which can be eliminated by a two level criticality aware cache hierarchy. This area can then be used to reduce die cost, increase LLC or add more cores. This marks a radical shift from the current trend of increasing L2 sizes for performance. These oracle results clearly show that instead of growing the L2, the trend should be to optimize cache hierarchy for critical instructions and move towards a simplified two level cache hierarchy where the LLC saves on memory misses and prevents new critical paths, whereas the L1 serves the critical loads. Moreover, Figure 5 also shows that only a small number of tracked critical load PCs (32) can give most of the gains. Hence it is possible to devise special, directed prefetchers for this small subset of critical loads. This forms the motivation for a Criticality Aware Tiered Cache Hierarchy (CATCH).

We note that since L2 is private and only LLC is shared, removing L2 shouldn't effect coherency. Since L2 isn't typically inclusive of L1, snoops to core also snoop L1 for correctness, and removing L2 shouldn't increase L1 snoop traffic. Shared-data will also continue to be handled as in baseline, through the private L1 and shared LLC. With CATCH simplifying the cache-hierarchy (fewer levels), coherency/shared-data handling can be further optimized. We leave this area of discussion for future work.

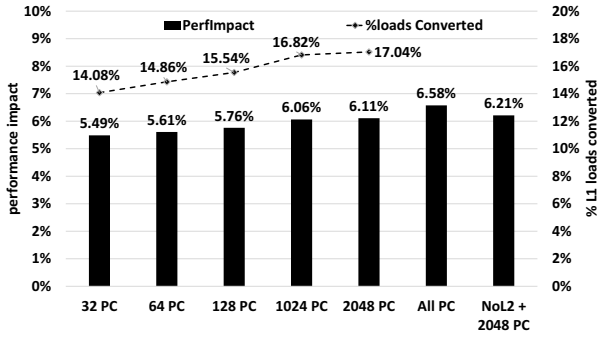

Figure 5. Performance impact of criticality aware oracle prefetch

IV. Criticality Aware Tiered Cache Hierarchy (CATCH)

The goal of CATCH is to ensure that critical loads that hit in the outer level caches (L2 and LLC) in the baseline are served at the L1 latency. To enable CATCH, we first propose an accurate detection of criticality in hardware. We then propose a family of prefetchers that prefetch cachelines corresponding to critical load accesses into the L1 cache and also prefetch code that miss the Code L1 cache and hence stall the machine's front end.

A. Criticality Marking

Identifying which loads are critical at runtime is the cornerstone of CATCH. There have been several proposals that propose heuristics to find out which load accesses will be critical [2], [6]. While using heuristics to identify

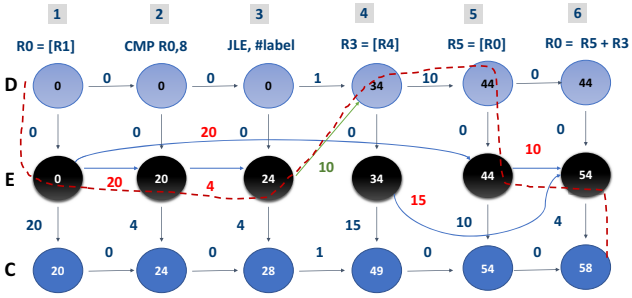

Figure 6. An example Data Dependency Graph (DDG) based on [1]

critical load PCs may be simple to implement, they often flag many more PCs than are truly critical. For instance, branch mis-predictions that lies in the shadow of a load miss to memory may still be flagged by heuristics-based mechanisms as critical. Instead of using heuristics, which would need to be finely tuned, we create a representation of the data dependency graph by Fields et al. [1] in hardware and use that to enumerate the critical path. We use the past history of the execution of the application to predict future instances of critical instructions. When instructions retire, they are added to the graph. When the graph has buffered instructions at least twice the reorder buffer (ROB) size, we walk through the critical path in this buffered portion of the graph and record the Program Counter (PC) of all the loads that were on the critical path and had hit in the L2 or LLC. However, there are two main problems with this approach. Firstly, the area of the graph can be prohibitive. Secondly enumerating the critical path needs a depth first search in the graph to find out the longest path. We now describe how we buffer the graph in hardware while addressing these two concerns.

Graph Buffering: We use the notation as used in [1] to create the data dependency graph. Each instruction has three nodes. The D node denotes the allocation in the OOO, E, the dispatch to execution units and C node the write-back time. The D-D edge accounts for in-order allocation, C-C for in-order retirement and D-E edge denotes the renaming latency. The E-D edge refers to bad speculation, C-D the depth of the machine, E-E actual data dependencies and the E-C edge represents the latency of execution. Each edge of the graph requires the edge weight and the node number to which it is connected to. However, many edges of the graph are implicit (D-E, E-C, D-D, C-C, C-D and D-E) and hence do not need to store the node number. The OOO core provides both data and memory dependencies between instructions (E-E), and information about bad speculation (E-D).

When an instruction retires, it is added to the end of the graph along with its node weights. The E-C edge (execution latency) is measured from the time the instruction was dispatched to the execution units till when it does the write-back. We quantize this latency (divide by 8) and store it as a 5 bit saturating counter. Once we have buffered X number of instructions, we can enumerate the critical path and identify the critical instructions. We found that a value of X that is equal to twice the reorder-buffer (ROB) size of the OOO is sufficient in most cases.

Enumerating the Critical Path: The critical path is the longest path from D node of the first instruction to the C node of the last instruction in the buffered graph. We propose an incremental method of finding the critical path. On allocation to the graph, each node of the retired instruction checks all its incoming edges to determine the one it needs to take to maximize its distance from the beginning of the graph. This distance is stored as the node-cost and the identified incoming edge as the *prev-node*. Since each node cumulatively stores its longest distance, incoming nodes only need to compare with their immediate edges. To understand this further, consider the example graph of Figure 6. When the first instruction is added to the graph, the node-cost of the D node will be 0. The E node has exactly one edge (with the D node) whose weight is 0. Therefore, its node-cost is 0 and its prev-node is 0. The C node's node-cost is 20 by virtue of the E-C edge and the prev-node is 1. Now when the next instruction 2 is added, the D node has only one edge (D-D) and so its node cost is 0 and prev-node is 0. The E node of instruction 2 has two incoming edges. The E node of instruction 1 with a cost of 20 (node-cost of instruction 1's E node + E-E edge weight) and the D node of instruction 2 with a cost of 0. Therefore, its node-cost will be 20 and prev-node will be 1. This process is repeated on every addition of a new instruction to the graph. Once the graph has buffered twice the ROB depth number of instructions, we simply need to walk through the prev-node pointers to enumerate all instructions on the critical path. We should note that we only care for the E nodes on the critical path, since these are the instructions whose execution latency impacts the critical path. Further, since we are only interested in load instructions we optimize the prev-node to simply store the previous E node of a load on the critical path.

Recording the Critical Instructions: During the critical path walk through in the graph, we record the PC of the load instructions that are on the critical path and hit in the L2 or LLC in a 32 entry critical load table which is 8-way set-associative and maintained using LRU. We also maintain a 2 bit saturating confidence counter for each table entry. The PC is marked critical only if it is in the table and its confidence counter has saturated. After every 100K instructions have retired, we reset the confidence counters of those PCs that have not yet reached saturation and ask them to re-learn. We should note that walking through the graph and recording in the critical table will take a finite number of clock cycles, depending on the length of the E-chain in the critical path. This should normally be just a few cycles, because the number of critical instructions on the critical path tend to be small [1]. However, there are

Edge Type	Description	Bits needed (b)
D-D, C-C, D-E, C-D	In-order Dispatch (D-D), In-order Commit (C-C), Dispatch to Execute (D-E), Depth limitation (C-D)	0 (implicit edges)
E-C	Execution latency	5b (quantized)
E-E	Data Dependency with a node	9b * 3 (sources) + 9b (memory dep.) = 32b
E-D	Bad speculation	1b (to signify)

Table I
Area calculations for buffering the DDG graph

cases when the path can be long and it may take several cycles to walk through the graph. In the meantime, the ROB continues to retire instructions. Therefore, we keep a larger buffered graph than actually needed, to find out the critical path. In our paper we keep it to be 2.5 times the ROB size, but we only walk through the buffered graph corresponding to twice the ROB size. Once we have walked through the critical path, we flush out the buffered instructions (this is done by just resetting the read pointer of the graph structure) and wait for the next set of instructions to be buffered. In case the graph overflows, we just discard and start afresh.

Area Calculations: Table IV-A summarizes the area requirements per instruction in the graph. For a processor with 224 ROB entries we need 2.3 KB of storage. Note that each node finally only needs to store the prev-node and node cost and can discard information on other incoming edges. Additionally each instruction needs to store the PC address. We use a 10 b hashed PC address instead of the full address. This needs about 1 KB additional storage. The total area of our critical path enumeration solution is about 3 KB.

B. Timeliness Aware and Criticality Triggered Prefetches

Once critical loads have been identified, we need to prefetch them in the L1 cache so that the critical path length can be shortened. We should note that this problem is very different from the memory prefetching problem, where the goal is to fetch requests from the DRAM memory and increase the LLC hit rate. Our goal is to do a timely prefetch of cachelines, which are present in the outer level caches, into the L1 cache. This means that our prefetches need to hide a much smaller latency than typical memory prefetchers. Moreover since the L1 bandwidth and capacity is small, it is important to direct these prefetches to only a select list of critical loads that matter for performance. Overfetching into the L1 can cause L1 thrashing, create new critical paths and hamper performance. To meet this goal, we propose a family of Timelness Aware and Criticality Triggered Prefetchers (TACT) that accurately prefetch data cachelines from outer level caches into the L1 just in time before the core would actually need them. TACT also prefetches L1 code misses to prevent front end stalls. Since the front end (FE) of modern OOO processors is still in-order, code misses can stall the entire FE pipeline and are hence crucial for performance.

1) TACT - Data Prefetching: In modern instruction set architectures (ISA) [26], [27], the most generic form of address computation for a load instruction (with Program Counter (PC) X) is of the form

$$LdAddress = Data(RegSrcBase) + Scale * Data(RegSrc) + Offset \quad (1)$$

Using (1) we can then express prefetching using a tuple *(Target-PC, Trigger-PC, Association)*. The Target-PC is the PC of the load that needs a prefetch. For TACT these loads are dynamic instances of the critical loads that were identified using the criticality detection in Section IV-A. The Trigger-PC is the load instruction that will trigger the prefetch for the Target. Attributes (address or data) of the Trigger-PC will have a relation to the address of the Target-PC. This relation needs to be learned by TACT to successfully issue *just in time* prefetches for the Target-PC.

On the dispatch or execution of an instance of a Trigger-PC from the OOO, the address of a subsequent instance of the Target-PC can be predicted using the relevant attributes of the Trigger-PC and its relation to the Target-PC. The specific instance of the Target-PC prefetched by a given instance of the Trigger PC is the measure of the *prefetch distance* and is related to the timeliness of the prefetching [32], [42]. For instance, prefetch distance of p is prefetching the p^{th} subsequent instance of the Target-PC on a trigger from the Trigger-PC. We should note that higher prefetching distance can pollute the small L1 caches, and hence TACT needs to arrive at an optimal, least possible, distance for each instance of the Target-PC.

Based on these expressions of prefetching, we propose three TACT prefetchers. We only do TACT learning and prefetching for the 32 critical loads learnt by the criticality marking scheme of Section IV-A. It should be noted that we evaluate our TACT prefetching on top of aggressive state-of-the-art prefetching mechanisms, namely the stride prefetcher in the L1 [41] and the aggressive multi-stream prefetcher [32], [35] in the L2.

TACT - Cross: Cross trigger address associations typically arise due to load instructions where the Trigger-PC and Target-PC have the same $RegSrcBase$ but different $Offsets$. They can also arise in other indirect program behavior when the source registers for the Trigger-PCs and the Target-PCs are loaded with data values with fixed deltas between them. To exploit the cross association between target and trigger in TACT, we first propose a simple mechanism in hardware to identify the cross Trigger-PCs.

We make the key observation that over 85% of cross address association delta values are well within 4 KB page. This means that both the Trigger-PC and the Target-PC are likely to access the same 4 KB page. To track possible

Trigger-PCs for any target we track the last 64 4 KB pages seen in a 64 entry 8 way set-associative *Trigger Cache*, that is indexed using the 4 KB aligned address. Each entry in the cache tracks the first four load PCs that touch this 4 KB page during its residency in the Trigger Cache. Critical Target-PCs intances, during training, lookup this Trigger Cache with their 4 KB aligned address and receive a set of four possible candidates for Trigger-PC. These load PCs may have a cross association with the target load.

Each Target-PC entry has a single current trigger candidate that is initially populated with the oldest of the four possible Trigger-PCs from the trigger cache and lasts till sixteen instances of the trigger. If a stable delta between the trigger and target isn't found by then, it switches to the next from the possible candidate Trigger-PCs. We allow wrapping around of the Trigger-PC candidates from the Trigger Cache a total of four times before we stop searching for a Cross Trigger PC. Once a stable Trigger-PC has been identified for the Target-PC, the TACT-Cross prefetcher will issue a prefetch whenever the OOO will dispatch the Trigger-PC. The prefetch address will be the address of the current instance of the Trigger-PC added with the offset that TACT has learnt during training.

TACT - Deep Self: The most common address association for loads is the one between addresses of successive instances or iterations of the same load PC. For example, loads inside a loop may see a stable offset between load addresses in succsessive iterations fo the loop. We call these associations as *self* trigger address associations. This is commonly used in stride prefetching [41], and is already employed in our baseline system. However, the baseline stride prefetcher uses a prefetch distance of one that may not be timely enough to save all of the L2 or LLC hit latency. Conversely, an increase in L1 latency due to excessive loading by too many prefetches would hurt performance. For these reasons, increasing the prefetch distance of all load PCs in the baseline stride prefetcher hurts performance. *TACT, therefore, adds increased, deep, prefetch distance prefetching for only a small subset of critical load PCs.* We cap the maximum distance of prefetches issued to sixteen to maintain a balance between timeliness and L1 cache pollution.

Deep distance prefetch addresses are predicted by multiplying the learnt stride/offset by the desired distance and adding to the triggering address. Even PCs that have a frequently occurring high confidence stride in their addresses don't necessarily have only a single stride in their access pattern. For example with loads in loops, a high frequency stride occurs for every iteration of the loop but loop exits followed by a re-enter at a later time will see a different stride. Therefore, indiscriminately doing prefetches at distance sixteen for all critical Target-PCs will cause L1 cache pollution.

TACT learns a *safe* length of stride seen by the critical Target-PC. We track the current length of the stride seen by the Target-PC (capped to 32 with a wraparound), and use it to update (increase or decrease) the safe length counter for the Target (again capped to 32). The confidence of this learnt "safe" length is tracked using a 2 bit safe length confidence counter. The safe length counter is initialized to four. TACT issues prefetches for both prefetch distance 1 and the maximum safe prefetch distance based on current length and safe length, if the deep distance confidence counter is saturated.

TACT - Feeder: When address associations don't exist for critical loads, TACT attempts data associations. The first step is to identify the Trigger-PC. Prior published work by Yu et al. [40] explored heuristics to determine the Trigger-PC. We propose an alternate simple hardware to track load to load dependencies and determine the Trigger-PC.

For TACT-Feeder, we are only interested in tracking the dependencies between load instructions. We do this by tracking the last load that *updates* an architectural register. For every architectural register, we store the PC of the last load that updated it. A load instruction directly updates the PC in its destination architectural register. For non-load instructions, the destination architectural register is updated with the youngest load PC across all of its source architecture registers. This mechanism propagates information on load PCs that directly or indirectly update architectural registers. The Trigger-PC for a Target-PC is the youngest load PC to have updated any of the load's source registers.

The TACT entry for a target increments a 2 bit confidence counter for the Trigger-PC. When the confidence saturates, the Trigger-PC is added to a Feeder-PC-Table and TACT begins to learn whether a linear relationship of the form $Address = Scale*Data+Base$, exists between the Trigger Data and Target PC address. To eliminate hardware required for complex division operations, we limit the possible scale values we use to powers of 2 namely 1,2,4 and 8. We therefore need at most three shift operations. We use a 2 bit confidence counter for both the Scale and Base learning. Once stable and high confidence values for the Scale and Base are learnt, data from a Trigger-PC can trigger a prefetch for the Target PC. For timeliness of Target-PC prefetch, we prefetch upto a prefetch distance of four for the Trigger-PC. The prefetch for the Trigger-PC, when data is available, triggers the prefetch for the Target-PC. If the Trigger-PC doesn't have a self trigger address association then we cannot do TACT-Feeder prefetching.

Figure 7 summarizes the three different forms TACT prefetchers for data through illustrations of a program.

2) TACT - Code Prefetching: An in-order Front End (FE) in microprocessors fetches instruction bytes of the program from a Code L1, decodes them, and feeds the decoded instructions to the OOO [31]. The address of the instruction

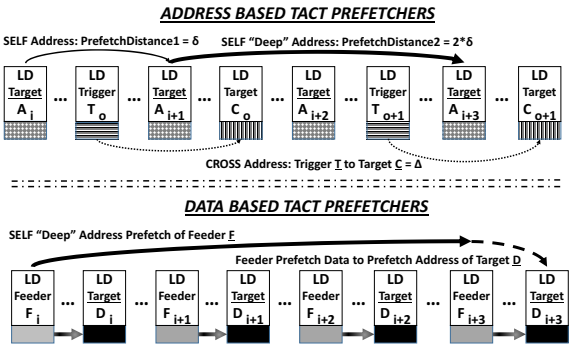

Figure 7. TACT Data Prefetchers - CROSS, Self Deep and Feeder.

bytes to be fetched, called the Next Instruction Pointer (NIP), uses the current NIP to predict the next NIP. This includes detecting branches in the bytes of the current NIP and predicting the target of the branch. The address pointed to by the NIP is then looked up in the code L1. An L1 code miss can stall the entire pipeline. During a stall, the NIP logic and the branch prediction units sit idle. To prevent code stalls, TACT proposes a code runahead to prefetch code lines while the FE is stalled serving the code miss.

As depicted in Figure 8 TACT adds a Code Next Prefetch IP (CNPIP) counter that is used to prefetch code lines. When the NIP logic is stalled by a code L1 miss, the current NIP is check-pointed and the NIP logic is queried with the CNPIP instead. This allows the CNPIP to *runahead* of the NIP and prefetch the bytes CNPIP points to into the Code L1. The branch predictor predicts the next instruction pointer, whenever a branch is encountered for a given CNPIP. The CNPIP is reset to the base NIP on a branch mis-prediction or when the base NIP moves ahead of it. The TACT CNPIP logic only operates when the base NIP is stalled in the FE. It adds no extra ports to the NIP query, Code L1 or to the instruction decode and dispatch logic/queues. The TACT-Code runahead is similar to decoupled branch predictor by Reinman et al. [49] and followups by Kaynak et al. [50] and Kumar et al. [51], as well as the data run-ahead proposed by Onur et al. [46].

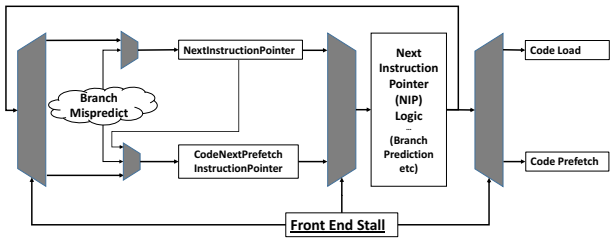

Figure 8. TACT Code Runahead Prefetcher

*3) **TACT Hardware Requirements**:* Figure 9 summarizes the structures needed for the TACT prefetchers and the area requirements. The total storage required by TACT for all the different structures is about 1.2 KB.

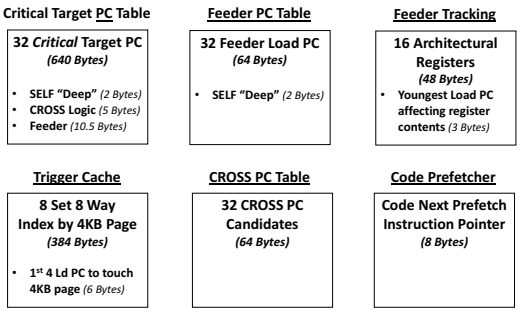

Figure 9. Structures introduced by TACT with area calculations

V. Evaluation Methodology

For our simulations, we model dynamically scheduled x86 cores using an in-house, cycle-accurate simulator. Each core is four-wide with 224 ROB entries and clocked at 3.2 GHz. The core micro-architecture parameters are taken from the Intel Skylake processor [12]. Each core has 32 KB, 8-way L1 instruction and data caches with latency of five cycles and a private 1 MB 16-way L2 cache with a round-trip latency of fifteen cycles. The cores share a 5.5 MB, 11 way exclusive LLC with data round-trip latency of forty cycles. These cache parameters are taken from the latest Intel Skylake server processors [30]. Each core is equipped with an aggressive state-of-the-art multi-stream prefetcher prefetching into the L2 and LLC. The L1 cache is equipped with a PC based stride prefetcher. The main memory DRAM model includes two DDR4-2400 channels, two ranks per channel, eight banks per rank, and a data bus width per channel of 64 bits. Each bank has a 2 KB row buffer with 15-15-15-39 (tCAS-tRCD-tRP-tRAS) timing parameters. Writes are scheduled in batches to reduce channel turn-arounds.

We selected 70 diverse applications from the SPEC CPU 2006, HPC, Server and Client category of workloads, that are summarized in Table II. **We use all 29 SPEC CPU 2006 benchmarks in this study** and call out the average category gains for integer SPEC (ISPEC) and floating point SPEC (FSPEC). Apart from these 70 ST workloads, we also created 60 four way multi-programmed traces, half of which are RATE-4 style (four copies of the application run on four cores) and the remaining are random mixes from our ST applications. We simulate 100 million dynamic instructions and use instructions per cycle (IPC) to measure performance in ST workloads and weighted speedup in MP workloads.

VI. Simulation Results

We first evaluate CATCH on single thread workloads in Section VI-A. The contributions of each TACT component is analyzed in Section VI-B followed by the results for workloads with all four cores active in Section VI-C. We show the sensitivity of CATCH to various system parameters in Section VI-D and do a detailed power analysis in

Benchmarks	Category
perlbench, bzip2, gcc, mcf, gobmk, hmmer, sjeng, libquantum, h264ref, omnetpp, astar, xalancbmk	SPEC INT 2006
bwaves, gamess, milc, zeusmp soplex, povray, calculix, gemsfdtd, tonto, lbm, wrf, sphinx3 gromacs, cactusADM, leslie3D, namd, deall,	SPEC FP 2006
blackscholes, bioinformatics, hplinpack, hpc applications	HPC
tpce, tpcc, oracle, specjbb, specjenterprise, hadoop, specpower	SERVER
Sysmark-excel, Face detection, h264 encoder	CLIENT

Table II
SUMMARIZED LIST OF APPLICATIONS USED IN THIS STUDY.

Section VI-E. In Section VI-F we will also evaluate CATCH on an inclusive baseline.

A. Large L2, Exclusive LLC, ST workloads

Figure 10 summarizes the performance impact of CATCH on the baseline with 1 MB L2 and a 5.5 MB of shared, exclusive LLC, averaged over 70 ST workloads. Removing the L2 while keeping the overall cache per core constant (NoL2 + 6.5MB LLC) results in a large 7.8% drop in performance. If we move to a two level hierarchy and remove the 1 MB L2 from each core, the LLC can be grown to 9.5 MB at the same area (NoL2 + 9.5 MB LLC). For ST workloads this increases the effective cache capacity but the configuration still suffers a 5.1% average loss in performance.

CATCH applied on the NoL2 configuration recovers the 7.8% loss incurred by removing the L2 and yields an impressive 4.55% gain. This configuration, based on estimates from the die plots of recent microprocessor offerings [30], is about 30% lower area than the baseline. The area savings can be used to grow the LLC and CATCH on the two level hierarchy with increased LLC (NoL2 + 9.5 MB LLC) yields 7.23% performance gain. Figure 10 also shows that CATCH optimization applied on the baseline gives an 8.4% average performance gain. These results clearly demonstrate the benefit of CATCH. Firstly criticality awareness in the baseline cache hierarchy itself improves performance gain by 8.4%. Secondly CATCH reduces the sensitivity to large L2 sizes since it ensures that critical loads, that were a hit in L2 or LLC, are served at L1 latency. This is seen in the results where CATCH on a two level hierarchy (NoL2 + 9.5 MB LLC) is close in performance to CATCH on a three level hierarchy, with both configurations having the same area. This result can be then be used to study interesting trade-offs for power which we will discuss in Section VI-E.

Figure 11 shows TACT prefetchers primarily target timely inter-cache prefetching, with 88% of critical TACT prefetches served by the LLC and over 85% of these prefetches saving more than 80% of LLC latency for subsequent critical loads. Prefetch fills into L1 increase by only 9% on average, tying back to the premise in Section III-C that only a small percentage of critical loads need to be accelerated.

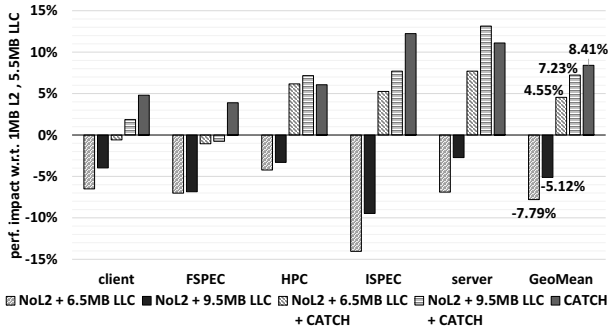

Figure 10. Performance gain on large L2, exclusive LLC baseline

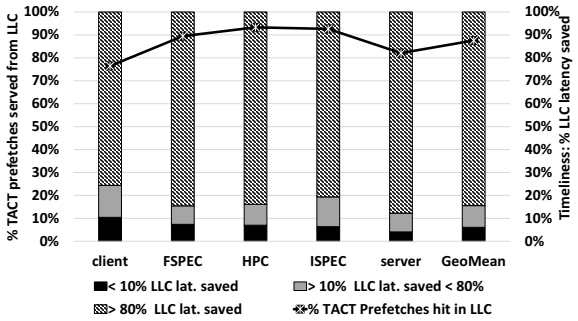

Figure 11. Timeliness of inter-cache TACT prefetching

Figure 12 shows the performance for each of our 70 workloads. CATCH recovers the losses on several workloads that lose significantly without an L2. For example *hmmer* loses nearly 40% performance, but with the CATCH hierarchy the loss is less than 5%. TACT Feeder prefetches lift *mcf* from a 30% loss to a 55% gain in performance. We note that while the two level CATCH hierarchy outperforms the baseline on average and for majority of the workloads, there are some workloads where we don't fully recover the loss in performance from removing the large L2. Workloads like *namd* and *gromacs* have some load PCs that are not amenable to prefetching and hence CATCH gains are limited for them. *Povray* suffers from a large number of critical load PCs and as a result 32 critical load PC table is insufficient to track all such loads. Targeted prefetching and better critical load table management can help these workloads significantly and we leave such investigations for future work.

B. Analysis of CATCH Components

Figure 13 summarizes the contribution of the various TACT prefetchers in a two level cache hierarchy with L2 removed (NoL2 + 6.5 MB LLC). TACT-Code prefetching boosts average performance by 0.75%. Server category of workloads tend to have large code footprints and they benefit

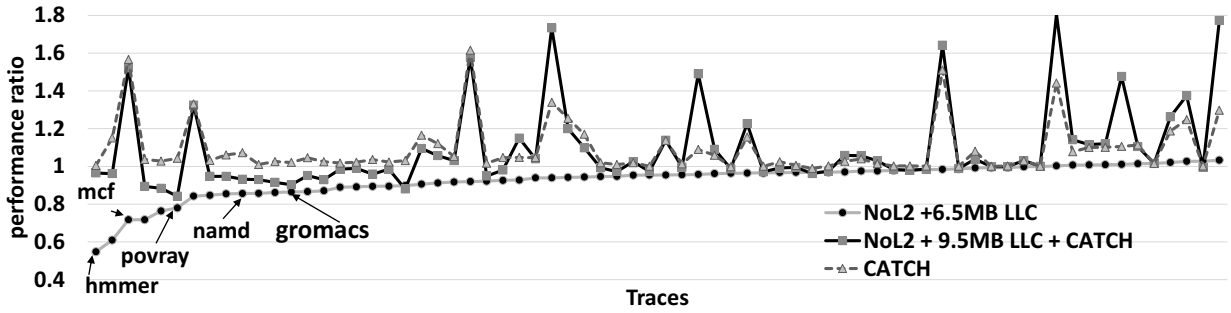

Figure 12. Per workload performance impact

the most with this prefetcher. The TACT-Cross prefetcher further boosts performance by 3.6% with SPEC 2006 and HPC workloads benefiting the most. TACT Deep SELF prefetchers provide 5.9% additional performance. Finally, the TACT Feeder prefetcher adds a 2.7% average increase in performance with the ISPEC category being the biggest beneficiary. Overall, the different TACT prefetchers together result in a performance gain of 13% on top of a NoL2 baseline.

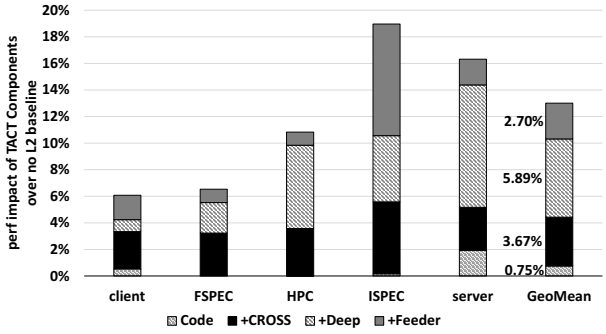

Figure 13. Performance gain from each component of TACT

C. Performance on MP Workloads

Figure 14 summarizes the performance on the four way MP workloads. A three level CATCH-optimized hierarchy outperforms the baseline by 8.95%. The two level CATCH-based hierarchy matches the three level hierarchy performance, outperforming the baseline by 8.45%. These gains are similar to what we see on ST workloads.

D. Sensitivity Studies

In this section we evaluate the robustness of our proposal to different system parameters.

1) Effect of LLC Latency : Higher LLC latency may be warranted in server processors with longer interconnects. Figure 15 shows the performance of CATCH with increasing LLC latency. Each six cycle addition to LLC latency reduces the performance of the NoL2 and the corresponding CATCH configuration by about 2%. This is expected since the TACT prefetchers may not be able to fully hide the higher LLC latency.

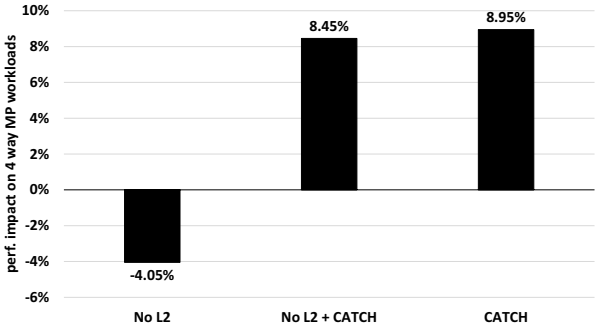

Figure 14. Performance impact on multi-programmed workloads

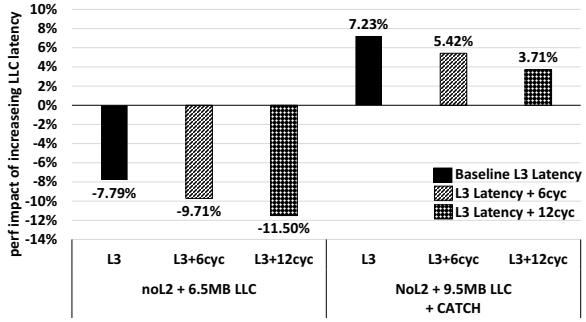

Figure 15. Sensitivity to LLC hit latency

2) Effect of Critical Load Table Size : For our evaluations, we use a 32 entry critical load table. While tracking more critical load PCs helps a few benchmarks, the overall performance gain was found to be minimal. This was because some of the loads tracked by larger tables had only a few instances when they were critical. As a result prefetching these loads caused more L1 thrashing. A smaller load table in fact helped weed away loads that were not critical frequently enough.

E. Power Analysis

Our results in Section VI-A showed that a two level cache hierarchy with CATCH optimization (NoL2 + 9.5 MB

LLC) outperformed the baseline, at the same area. However, we also showed that CATCH can also be applied on the baseline three level hierarchy and that also yields similar performance. Since performance delivered is similar, in this section we analyze the two level CATCH and the three level CATCH for power.

The baseline three level hierarchy with an exclusive LLC incurs both cache and interconnect overhead of moving victim data (clean as well as dirty victims) from the L2 to the LLC, on a fill into the L2. Single use blocks (that see few hits in the L2) make several trips between the L2 and the LLC. Each trip of such blocks costs L2 and LLC read and write power, since exclusive LLC needs to de-allocate the line on a hit. Additionally L2 stream prefetchers are often allocated early in to the L2, because of which they get victimized and move to the LLC without seeing a hit in the L2. This results in extra cache and interconnect traffic. However, L2 cache also filters away significant amount of LLC and interconnect traffic, especially for L1 write-backs that are merged in the L2, thereby saving costly writebacks to the LLC. Removing the L2 in a two level CATCH optimized hierarchy would lose this L2 filtering and add increased interconnect power to the system,. However, with the removal of the L2, we can use the L2 area to increase the LLC capacity visible to a core. This in turn reduces traffic to the off-die memory. Moreover not having L2 also helps reduces overall cache traffic.

To understand the above we take a simple example of 100 L1 misses that lookup the L2. Let's assume 80 of these hit in the L2. The 20 L2 misses will travel the interconnect and read the LLC (assuming LLC hit rate is 100%). The LLC reads will fill the L2 and the L2 victims will fill into the LLC. Overall the L2 will see a cache traffic (read + write) of 120 and the LLC will see 40 accesses. Therefore, the total cache traffic is 160 and the interconnect traffic is 40. Without the L2 all 100 accesses will be sent to the LLC. Therefore, the two level hierarchy will have 40% lower cache traffic $(100/160)$, but will have 2.5X more interconnect traffic. In summary, a two level CATCH will have lower cache and off-die memory traffic (because of increased LLC capacity), but will have higher on-die interconnect traffic.

To study the power impact of CATCH, we model the cache power using CACTI [29], estimate the interconnect power using Orion [43], [44] and model memory power using Micron DRAM power calculator [28]. For a 4-core system (that we evaluate in our studies) and a ring interconnect, we see on average 11% energy savings over the three level cache hierarchy baseline as shown in Figure 16. The two level CATCH has 37% lower cache traffic (L2 + LLC), 22% lower memory traffic but nearly 500% more interconnect traffic. For small interconnects, used by low core count processors, the savings in off-die DRAM memory power and on-die cache activity outweigh the substantial increase in the on-die interconnect power. However, this would not be true for large core count processors that would use a complex MESH as an interconnect. For such hierarchies optimized by CATCH, an L2 may still be needed for primarily reducing the interconnect traffic and not necessarily for increasing performance. However, instead of a large L2, which is used today primarily for performance, a smaller L2 maybe able to reduce the interconnect power significantly.

To summarize, CATCH represents a powerful framework to explore broad chip-level area, performance and power trade-offs in cache hierarchy design. Supported by CATCH, radical architecture directions such as eliminating the L2 altogether to dramatically reduce power or cost can be explored to create high performance CPUs.

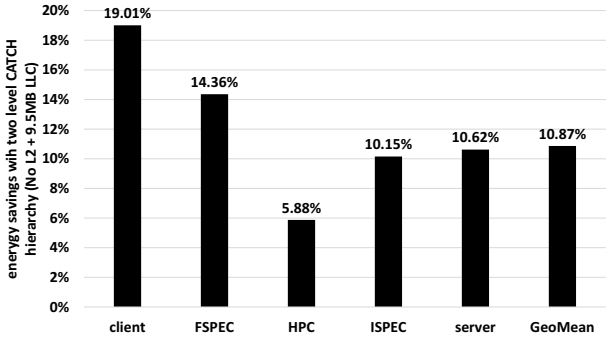

Figure 16. Energy Savings from CATCH on two level hierarchy

F. Small L2, Inclusive LLC

We also evaluate CATCH on a traditional inclusive LLC hierarchy with a 256KB L2 and a 8MB LLC, similar to [12]. Figure 17 summarizes the performance impact of CATCH on this configuration. Removing the L2 results in a 5.7% drop in performance. The two level CATCH hierarchy swings this to a 6.4% gain in performance over the three level baseline. Since we are removing the L2, this 6.4% gain is achieved with a significant saving in core area. Adding the L2 area (1 MB across four cores) boosts gains of the two level CATCH hierarchy to 7.2%. Finally, CATCH on the three level cache hierarchy baseline yields a 10.3% average gain in performance.

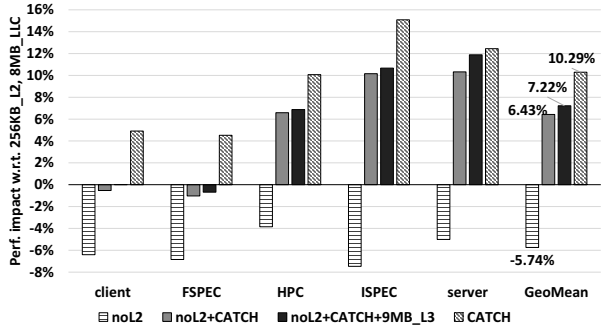

Figure 17. Performance gain on inclusive LLC baseline

VII. Related Work

Using program criticality to improve the performance of processors has been explored extensively in the past. The data dependence graph used in this work was described by Fields et al. [1] which also proposed a token passing based heuristic to determine the critical instructions on the fly. Several other works have described heuristics that can be used to determine critical instructions [2], [3], [6], [13]. CATCH uses an accurate and novel light weight detection of criticality via the data dependency graph but doesn't preclude the use of other finely tuned heuristics to determine critical instructions.

Criticality and caching were explored by Srinivasan et al. [5] where the conclusion was that LLC replacement policies should be based on locality and not criticality. This matches our findings and analysis in Section III-B with respect to the LLC. Misses in the LLC create new critical paths because of high memory latency, and hence using criticality to maintain the LLC is counter productive. However, managing on-die cache latency for critical instructions using our proposal, while maintaining same overall memory accesses, will only serve to shorten the critical path without creating new critical paths.

Criticality has also been explored in other areas of the processor. Ghose et al. [7] proposed techinques to prioritize critical loads in the memory controller. Subramniam et al. [6] proposed techniques using criticality to reduce the L1 load ports. Criticality has also been used to design energy efficient cores and caches by Balasubramonian et al. [20] and Seng et al. [3].

Recent research studies in the area of on-die caching have primarily focused on improving cache replacement in the LLC [8], [9], [14], [16]–[18], [22]. These techniques are applicable to the proposed criticality aware hierarchy as well. A comparison of exclusive to inclusive LLC by Jaleel et al. [19] and Gaur et al. [16] showed that large L2 caches and an exclusive LLC can give substantial performance. A replacement policy in the LLC using hints from lower level cache hierarchies was explored by Chaudhari et al. [8].

There have been several proposals for prefetching from memory into the LLC that exploit non-uniform but repeating spatial access patterns in a 4KB page [21], [33], [36]–[39]. We observe that the most of these non-uniform but repeating spatial access patterns of cachelines are due to stable address deltas between a series of IPs. It is this series of IPs that repeat during the course of the program. The cited prefetchers aim to reduce memory stalls by fetching a large number of cachelines in a page into the large LLC. On the other hand the TACT prefetchers learn complex associations specifically for doing accurate, just in time prefetching of critical loads from the L2 and LLC into the L1. Code prefetching techniques have been proposed in [47], [48].

To summarize, most of the prior work has focused on either improving the on-die cache hit rates through prefetching and better cache management or has used criticality as a local metric to improve latency in a certain part of the processor. To the best of our knowledge, this is the first work that uses program criticality to fundamentally understand how and why multi-level caches give performance gain. Through a holistic analysis of the trade-offs in a multi-level cache hierarchy, this work motivates the move towards simplified program criticality aware cache hierarchies and enable radical high-performance area-optimized designs.

VIII. Summary

In this paper we have presented CATCH, a fundamentally new way of designing multi-level cache hierarchies. Through sophisticated prefetch algorithms that are guided by an accurate criticality detection mechanism, a CATCH based on-die cache hierarchy outperforms the traditional three level cache hierarchy for single thread workloads by 10.3% for a 256KB L2, inclusive LLC baseline and by 8.4% for a baseline with an exclusive LLC and a large 1MB L2 cache. We also showed that CATCH enables a framework for exploring interesting trade-offs between area, power and performance and marks a fundamental shift in our understanding of how on-die caches should be designed going forward.

IX. Acknowledgment

The authors thank their colleagues at MRL, Pooja Roy, Shankar Balachandran and Harpreet Singh for their help with the work. We also thank the anonymous reviewers for their insightful comments and suggestions.

References

[1] B. Fields, S. Rubin, and R. Bodk. Focusing processor policies via critical-path prediction, in ISCA '01
[2] E. Tune, D. Tullsen, and B. Calder. Quantifying Instruction Criticality, in PACT '02
[3] J. Seng, E. Tune, and Dean M. Tullsen. Reducing power with dynamic critical path information, in MICRO '01
[4] S. Srinivasan and A. Lebeck. Load latency tolerance in dynamically scheduled processors, in MICRO '98
[5] R. Ju, A. Lebeck, and C. Wilkerson. Locality vs. criticality, in ISCA '01.
[6] S. Subramaniam, A. Bracy, H. Wang, and G. Loh. Criticality-based optimizations for efficient load processing, in HPCA '09
[7] S. Ghose, H. Lee, and J. Martnez. Improving memory scheduling via processor-side load criticality information, in ISCA '13
[8] M. Chaudhuri, J. Gaur, N. Bashyam, S. Subramoney, and J. Nuzman. Introducing Hierarchy-awareness in Replacement and Bypass Algorithms for Last-level Caches, in PACT '12
[9] A. Jain and C. Lin. Back to the future: Leveraging Belady's algorithm for improved cache replacement, in ISCA '16
[10] I. Cutress. The Intel 6th Gen Skylake Review: Core i7-6700K and i5-6600K Tested. August 2015. Available at http://www.anandtech.com/show/9483/intel-skylake-review-6700k-6600k-ddr4-ddr3-ipc-6th-generation/9.
[11] The AMD Zen and Ryzen Review. Available on https://www.anandtech.com/show/11170/the-amd-zen-and-ryzen-7-review-a-deep-dive-on-1800x-1700x-and-1700/9

[12] I. Cutress. The Intel Skylake Mobile and Desktop Launch, with Architecture Analysis. September 2015. Available at http://www.anandtech.com/show/9582/intel-skylake-mobile-desktop-launch-architecture-analysis/5.

[13] B. Fields, R. Bodk, M. Hill, and C. Newburn. Using Interaction Costs for Microarchitectural Bottleneck Analysis, in MICRO '03

[14] H. Gao and C. Wilkerson. A Dueling Segmented LRU Replacement Algorithm with Adaptive Bypassing, in the *1st JILP Workshop on Computer Architecture Competitions: Cache Replacement Championship*, June 2010.

[15] J. Collins, H. Wang, D. Tullsen, C. Hughes, Y. Lee, D. Lavery, and J. Shen. Speculative precomputation: long-range prefetching of delinquent load, in ISCA '01

[16] J. Gaur, M. Chaudhuri and S. Subramoney. Bypass and insertion algorithms for exclusive last-level caches, in ISCA '11

[17] J. Gaur, R. Srinivasan, S. Subramoney, and M. Chaudhuri. Efficient management of last-level caches in graphics processors for 3D scene rendering workloads, in MICRO '13

[18] A. Jaleel, K. Theobald, S. Steely, Jr., and J. Emer. High performance cache replacement using re-reference interval prediction (RRIP), in ISCA '10

[19] A. Jaleel, J. Nuzman, A. Moga, S. Steely, and J. Emer. High Performing Cache Hierarchies for Server Workloads – Relaxing Inclusion to Capture the Latency Benefits of Exclusive Caches, in HPCA '15

[20] R. Balasubramonian, V. Srinivasan, S. Dwarkadas, A. Buyuktosunoglu. Hot-and-Cold: Using Criticality in the Design of Energy-Efficient Caches, in PACS '03

[21] L. Peled, S. Mannor, U. Weiser, and Y. Etsion. Semantic locality and context-based prefetching using reinforcement learning, in ISCA '15

[22] S. Khan, Y. Tian, and D. Jimenez. Sampling Dead Block Prediction for Last-Level Caches, in MICRO '10

[23] S. Przybylski, M. Horowitz, and J. Hennessy. Characteristics of performance-optimal multi-level cache hierarchies, in ISCA '89

[24] M. Qureshi and Y. Patt. Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches, in MICRO '06

[25] S. Shukla and M. Chaudhuri. Tiny Directory: Efficient Shared Memory in Many-core Systems with Ultra-low-overhead Coherence Tracking, in HPCA '17

[26] x86 Architecture Overview. http://cs.lmu.edu/ ray/notes/x86overview/

[27] Arm Devloper Manual at https://developer.arm.com/products/architecture

[28] Micron Technology Inc. Calculating Memory System Power for DDR3. Micron Technical Note TN-41-01. http://www.micron.com/products/support/power-calc.

[29] N. Muralimanohar, R. Balasubramonian, and N. Jouppi. CACTI 6.0: A Tool to Model Large Caches. HP Labs Technical Report HPL-2009-85, HP Laboratories, 2009

[30] Drilling Down Into The Xeon Skylake Architecture.https://www.nextplatform.com/2017/08/04/drilling-xeonskylakearchitecture/

[31] Intel Haswell CPU micro-architecture. https://www.realworldtech.com/haswell-cpu/

[32] S. Srinath, O. Mutlu, H. Kim, and Y. Patt. 2007. Feedback Directed Prefetching: Improving the Performance and Bandwidth-Efficiency of Hardware Prefetchers, in HPCA '07

[33] M. Shevgoor, S. Koladiya, R. Balasubramonian, C. Wilkerson, S. Pugsley, and Z. Chishti. Efficiently prefetching complex address patterns, in MICRO '15

[34] S. Pugsley, Z. Chishti, C. Wilkerson, T. Chuang, R. Scott, A. Jaleel, S. Lu, K. Chow, and R. Balasubramonian, Sandbox Prefetching: Safe, Run-Time Evaluation of Aggressive Prefetchers, in HPCA '14

[35] F. Dahlgren , P. Stenstrm. Evaluation of Hardware-Based Stride and Sequential Prefetching in Shared-Memory Multiprocessors, in IEEE Transactions on Parallel and Distributed Systems, v.7 n.4, p.385-398, April 1996

[36] A. Jain, C. Lin. Linearizing irregular memory accesses for improved correlated prefetching, in MICRO '13

[37] S. Somogyi, T. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos. Spatial Memory Streaming, in ISCA '06

[38] Y. Ishii, M. Inaba, and K. Hiraki. Access map pattern matching for data cache prefetch, in ICS '09

[39] J. Kim, S. H. Pugsley, P. V. Gratz, A. N. Reddy, C. Wilkerson, and Z. Chishti. Path confidence based lookahead prefetching, in MICRO '16

[40] X. Yu, C. Hughes, N. Satish, and S. Devadas. IMP: indirect memory prefetcher, in MICRO '15

[41] J. Fu, J. Patel, and B. Janssens. Stride directed prefetching in scalar processors, in MICRO '92

[42] P. Emma, A. Hartstein, T. Puzak, and V. Srinivasan, Exploring the limits of prefetching, IBM J. R&D, vol. 49, no. 1, pp. 127-144, January 2005

[43] A. Kahng, B. Li, L-S. Peh, and K. Samadi. ORION 2.0: a fast and accurate NoC power and area model for early-stage design space exploration, in DATE '09

[44] Y. Hoskote, S. Vangal, A. Singh, N. Borkar, and S. Borkar. A 5-GHz Mesh Interconnect for a Teraflops Processor, IEEE Micro, v.27 n.5, p.51-61, September 2007

[45] S. Shukla and M. Chaudhuri.Sharing-aware Efficient Private Caching in Many-core Server Processors, in ICCD '17

[46] O. Mutlu, J. Stark, C. Wilkerson, and Y. Patt. Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-Order Processors, in HPCA '03

[47] V. Srinivasan, E. Davidson, G. Tyson, M. Charney, and T. Puzak. Branch History Guided Instruction Prefetching, in HPCA '01

[48] Y. Zhang, S. Haga, and R. Barua. Execution history guided instruction prefetching, in ICS '02

[49] G. Reinman, B. Calder, T. Austin. Fetch directed instruction prefetching, in MICRO '99

[50] C. Kaynak, B. Grot, and B. Falsafi. Confluence: Unified Instruction Supply for Scale-Out Servers, in MICRO '15

[51] R. Kumar, C-C. Huang, B. Grot, V. Nagarajan. Boomerang: A Metadata-Free Architecture for Control Flow Delivery, in HPCA '17.