CacheDoge: A watchdog for your cache

Artur Balanuta*
Carnegie Mellon University
Instituto Superior Técnico, Lisbon
artur@cmu.edu

Nidhi Bhatia
Carnegie Mellon University
nidhib@andrew.cmu.edu

ABSTRACT

Shared Memory is the dominant low-level communication paradigm in today's multi-core processors. In a shared-memory system, the cores communicate via loads and stores to a shared address space. The cores use caches to reduce the average memory latency and memory traffic. Caches are thus beneficial, but private caches lead to the possibility of cache incoherence. Therefore, in multi-core processor with multiple private caches, the cache coherence must be preserved. Cache coherence overhead becomes a major performance bottleneck when sharing patterns of parallel applications result in frequent invalidations followed by subsequent coherence misses. In this paper, we present a novel low cost micro-architecture component, CacheDoge, to determine and fix cache coherence performance bottlenecks in shared memory applications. CacheDoge uses existing processor features to reduce execution overhead by a significant degree and is transparent to the programmers.

KEYWORDS

Computer Architecture, Cache Systems, Optimization, Cache coherence, Parallel applications, Performance bottleneck, Latency, False Sharing, Write-Invalidate Protocol

1 INTRODUCTION

SRAM-based cache memory is an essential architectural component for CPU's (central processing unit) performance because it accelerates accesses to data stored in the slow and lower levels of memory hierarchy i.e. DRAM-based main memory. In general, a cache is a small, fast storage device that acts as a staging area for the data objects stored in a larger, slower device. Memory hierarchies based on caching works because programs tend to exhibit locality, i.e. *temporal locality* and *spatial locality*.

Because of *temporal locality*, the same data objects are likely to be reused multiple times by a program. Once a data object has been copied into the cache on the first miss, we can expect a number of subsequent hits on that object. Since the cache is faster than the storage at the next lower level, these subsequent hits can be served much faster than the original miss.

Cache Blocks usually contain multiple data objects. Because of *spatial locality*, we can expect that the cost of copying a block after a miss will be amortized by subsequent references to other objects within that block.

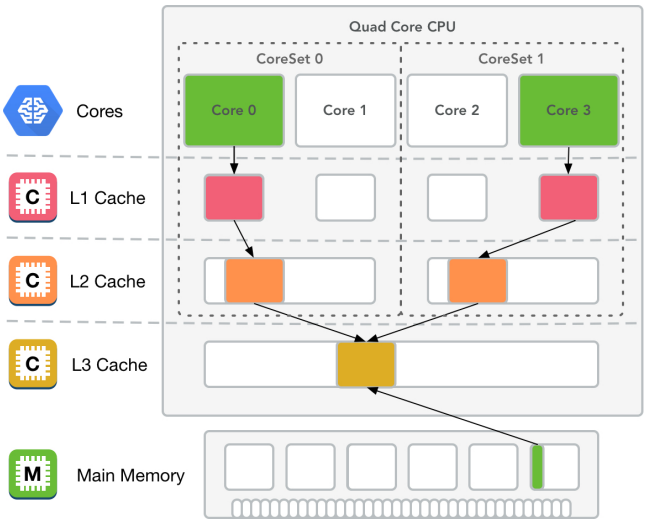

Figure 1: Example of an inefficient allocation of threads to cores (arrows represent memory dependencies). Application is using two thread, running on Core0 and Core3.

Modern multi-core processors implement shared memory inter-core communication methods. In computer software, shared memory is an efficient way of passing data between programs. For example, in numerical linear algebra, you are typically performing work on huge arrays like matrices and vectors. The most common construct here are long loops over the arrays. In a shared-memory approach, the whole array is accessible by all threads and the loop can be split among the threads in many fashions (if there are no loop-carried dependencies).

In computer hardware, shared memory refers to a (typically large) block of DRAM-based main memory that can be accessed by several different cores in a multiprocessor computer system. A shared memory system is easy to program since all cores share a single view of data and the communication between processors can be as fast as memory accesses to a same location. The issue with shared memory systems is that cores need fast access to memory and will use a small amount of very fast non-shared cache memory to exploit locality of reference in memory accesses.

With shared memory multi-core processors, maintaining cache coherence across shared memory has a significant overhead. Typically, shared memory multi-core processor uses inter-core communication between cache controllers to keep a consistent memory image when more than one cache stores the same memory location. For this reason, shared memory multi-core processor may perform poorly when multiple cores attempt to access the same memory area in rapid succession.

*Mr. Balanuta insisted his name be first.

18-742: Computer Architecture and Systems, December 2018, Pittsburgh, PA USA
2018. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

For example, shown in Figure 1 is a shared memory quad core processor. Each core i.e. Core 0, Core 1, Core 2 and Core 3, has a private L1 cache for faster memor y access and maximization spatial locality of reference memory access. Core 0-Core 1 and Core 2-Core 3 share a common L2 cache respectively. We say Core0 and Core1 respectively belong to the same core cluster, **CoreSet 0**. Similarly Core 2 and Core 3 belong to same core cluster, **CoreSet 1**. **PLEASE NOTE** We will be using **CoreSet** to refer to core clusters in the rest of the paper. All the four cores share L3 and DRAM-based main memory.

Parallel applications running on different cores might access data from similar block of memory in the DRAM-based main memory. This might not be a problem if the parallel applications load/read data. But it will lead to significant performance loss (in terms of additional delays) if parallel applications perform a save/write operations.

2 MOTIVATION

There are two main cache coherence protocols, namely, *write update protocol* and *write invalidate protocol*. In *write update protocol*, whenever a copy of shared data is changed, all the other copies must be "updated" to reflect the change, then it is a write-update protocol. Whereas in *write invalidate protocol*, a write to a cached copy by any processor requires other processors to discard or invalidate their cached copies. Since bus bandwidth is a precious commodity in shared memory multi-core processors, *write invalidate protocol* is the most commonly used.

Using write invalidate protocol, we analyze Figure 1 for **false sharing** scenario in which multiple cores (Core 0 and Core 3) belonging to different CoreSets attempt to access independent variables in the same cacheline. Initially, Core 0 and Core 3, read data from same block of memory in DRAM-based main memory (block highlighted in green in Figure 1). As a result, memory block is present in private L1 caches of Core 0 and Core 3 respectively. Assuming L2 and L3 caches are inclusive, cacheline is also present in L2 caches private to CoreSet 0 and Coreset 1 respectively. Similarly, cacheline is also present in shared L3 cache accessible to cores present in all CoreSets. Now when Core 0 wants to write to this cacheline. Since, this cacheline is also shared by Core 3, Core 0 sends an invalidate request to Core 3 to evict this cacheline. New data is updated by Core 0 in its private L1 cache. Since, Core 3 also needs to access another variable from the same memory block, Core 0 has to write back the updated cacheline to L3 cache (can be DRAM-based slow main memory for many shared memory multi-core processors). This write back operation adds extra delay to processes running on Core 3, leading to degraded parallel performance. In this case, due to *false sharing* there is a loss of parallel performance.

If the processes were running on Core 1 instead of Core 3, the extra delay will not be so significant. In this case cacheline just needs to be updated in the shared L2 cache (private to the CoreSet 0).

Amdahl's law states that in parallel workloads, if P is the proportion of a program that can be made parallel, and 1-P is the proportion that remains serial, then the maximum speedup that can be achieved using N number of processors is

$$1/((1-P) + (P/N)$$

If N tends to infinity then the maximum speedup tends to

$$1/(1-P)$$

Speedup is limited by the total time needed for the sequential (serial) part of the program.

In the above scenario, maintaining cache coherency introduces serialization into the parallel application. Cache coherency is inevitable because we can not compromise memory consistency (accuracy) for performance. The best we can do from is to minimize the performance overhead due to cache coherency. **In this paper, we track false sharing scenarios between different CoreSets and amortize its impact on the overall parallel performance with zero interference from the software programmers.**

3 DESIGN SPACE

In order to thoroughly evaluate our design space we studied both, software and hardware perspectives of the problem.

Software Perspective

Modern operating systems (OS) are intelligent and encourage scheduling threads based on their data locality in the main memory [1] [6]. Parallel applications which share a largely overlapping working set are scheduled by OS on the same CoreSet, which makes more effective use of shared caches. But this strategy is not foolproof. There are plenty of scenarios under which this strategy may not work. We will talk about two of such scenarios.

First scenario, consider a case where threads of other random applications (with not so much data overlap) are already assigned to random cores in the shared memory multi-core system based on as and when the request was made. This results into "fragmenting" shared cache memory, which could have been used efficiently by two parallel applications with similar data locality in the main memory. Here, OS will schedule the parallel applications with similar data locality to run on the next available cores. This may lead to ineffective utilization of shared cache leading to degraded performance of parallel multi-threaded applications. This is also a default setting used by the Linux scheduler.

Second scenario, this happens mainly for applications with random pointer arithmetic and random shared memory accesses. In this case, OS cannot predict the memory access patterns and data locality for the application. As a result, it leads to random scheduling of applications which may have good data overlap on cores belonging to different CoreSets.

Furthermore, to facilitate application development OS is supposed to provide an abstraction layer to programmers from the underlying hardware architecture. Above mentioned scenarios also highlight a situation where providing abstraction comes at the cost of performance (application does not take advantage of underlying hardware architecture resources).

In order to optimize application performance on a given architecture, programmers may have to do lot of extra work. For example, manage the data placement i.e. scratchpad registers/memory, avoiding false sharing scenarios, and handling the thread scheduling in such a way that application is efficiently optimized for performance (pinning threads to cores manually).

Hardware Perspective

As mentioned before, a cacheline may be falsely shared by several cores in a shared memory multi-core processor. Any core in a given CoreSet willing to write to the shared cacheline should inform all the other cores sharing the cacheline (some of which may belong to different CoreSets). If the other cores sharing the cacheline belong to the same CoreSet, they can resume running their threads after the cacheline in updated in the private shared memory of their CoreSet (L2 cache in case of Figure 1). But if the other cores sharing the cacheline do not belong to the same CoreSet, the cacheline has to be updated in the memory which is shared among all CoreSets i.e. all cores (L3 cache in case of Figure 1). In this case, the time taken to write back the cacheline in the shared memory plus the time taken to load the cacheline back into respective private L1 caches will degrade the performance of the parallel application. This typically happens when OS schedules the applications with significant working set overlap on random cores which may not belong to same CoreSets.

The performance impact of random scheduling by OS will be severe in case of Intel's Knights Landing architecture, where the LLC (Last Level Cache), L3 cache, is not present. Here the performance impact of false cacheline sharing will be much more significant due to the access latency of DRAM-based main memory, which is orders of magnitude higher than access latency observed in caches.

After looking at both hardware and software perspectives, the key questions which arise are:

- Is it possible to detect when different threads with significant working set overlap are not allocated to the same CoreSet?
- And if yes, is it possible to build a micro-architecture component that keeps track of cache misses due to coherence invalidation requests and proactively tries to minimize them?

Prior work on cache coherence focused on simulation of coherence protocols. Architectural simulators support a multitude of coherence models in their implementation. These simulators and systems operate at different levels of abstraction ranging from cycle-accuracy over instruction-level [5] [8] [4] to the operating system interface [10]. Past work on the performance tuning concentrates on program analysis to derive optimized code. More recent work on identifying coherence bottlenecks is based on tracing memory accesses via dynamic binary rewriting. These past approaches are slow because of the reliance on purely software based techniques to obtain data traces, either by means of slow hardware simulations or via software instrumentation with significant overhead per access point [7].

We propose a novel micro-architecture component, CacheDoge - A low level watch dog for the memory system, which, also shown in Figure 2

- *Keeps track of cache conflicts due to shared memory*
- *Identifies cores (belonging to different CoreSets) which register maximum number of cache conflicts*
- *Speculatively migrates identified cores to the same CoreSet and improves performance of the parallel application*

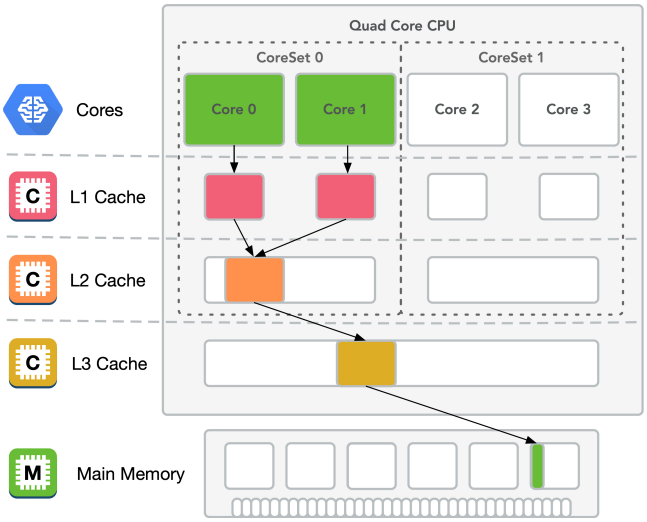

Figure 2: An example where two threads of an application is allocated to run on a CoreSet allowing for more efficient use of the caching architecture

Our CacheDoge implementation will motivate the future micro-architectures to migrate threads (with significant working set overlap) to the same CoreSet and optimize cache locality without any intervention of software.

In the following sections we will talk about our design choices, implementation of CacheDoge, evaluation methodology, results we got by implementing CacheDoge, future work for further improvement and finally the conclusion of this research.

4 DESIGN CHOICES

Figure 1 exemplifies an inefficient thread allocation which does not reap benefits of shared cache system. Here the threads interested to read and write to similar memory locations are executed on cores i.e. Core0 and Core3 belonging to different CoreSets. We can clearly see that in case of write operation done by any thread, the updated cacheline will require to propagate through all the cache levels, causing high latency in the operation of another thread.

Our CacheDoge will be able to detect-and-fix these unwanted latencies. For instance, it will migrate the thread running on Core 3 (CoreSet 1) to Core 1 (CoreSet 0). This thread migration will help improve parallel performance by exploiting private cache locality.

Detect

In order to detect these unwanted latencies, we send additional information along with each cache invalidation request. This additional information is the Core ID of the core that requested the cache invalidation. Private Cache Controllers that receive these invalidation requests log them into a one simple counter structure local to each core. These counters are used by CacheDoge to generate a matrix that shows how many times a particular core's cache was invalidated and by which core (shown in Figure 3)

Fix

CacheDoge takes this table as input and speculatively selects core pairs to be swapped. To make this selection, it finds the core pair with maximum number of total invalidation requests. Core pairs with highest number of invalidation requests showcase the cores have false sharing issues.

Since this is a novel approach, it needs to be further tuned by testing it on different parallel application test cases. Based on which, we can fine tune the threshold value for received cache invalidations from any given core, beyond which our algorithm should swap the cores/threads to maximize performance.

CacheDoge is fully implemented in micro-architecture without any instruction set support. This makes it transparent to programmers and requires only trivial additional hardware structures.

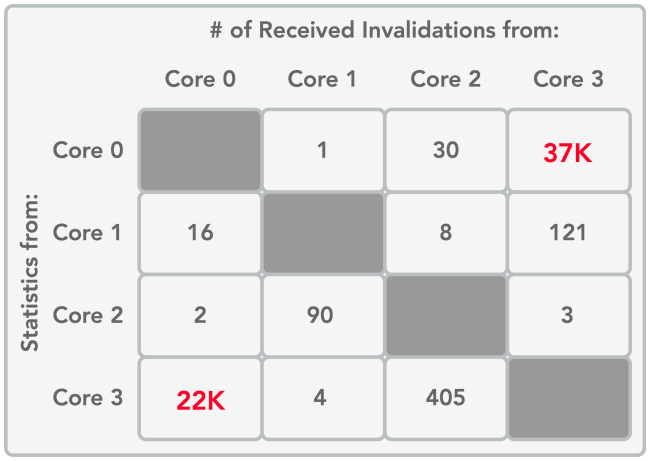

Figure 3: Representation of the structure used to store each Core's invalidation counters

5 IMPLEMENTATION

5.1 Cache Architecture

In order to test our solution we have developed a multi-cache instrumentation Tool using the Intel PINTool and their libraries [2]. We based our instrumentation tool on the provided example of a single core Cache written in the C++ programming language. We extended the basic functionally of the single core cache to a multi-core cache, based on architectures specified in the design section. The implemented Cache system provides a simplistic directory based coherence model that makes use of tags to identify the presence of data in the caches. For each level of the cache level we can specifies the Cache size, cacheline size, cache associativity (number of sets), write allocation policy and type of cache eviction mechanism (Round Robin, Directly mapped, LRU (Least Recently Used)).

Most modern CPUs are typically known to use LRU or Random Cache eviction mechanisms. The Round Robin mechanism generally performs similar to LRU systems and since this is not the main contribution of this work we have used Round Robin mechanism in our simulations. The implementation of more accurate CPU eviction mechanisms are scheduled for future work and are expected to have minimal impact on the results.

In our implementation we are targeting multi-core CPUs with cache architectures composed of individual private L1 Data and Instruction Caches per core. A unified L2 Cache is shared between sets of two cores and all the the cores share a common Unified Last Level Cache (most commonly L3 or in the some cases DRAM-based main memory).

In order to simplify the simulations we have chosen Quad Core CPUs with a cache hierarchy similar to the one shown in Figure 1. Several existing multi-core processors use or have used this type of cache configuration. The following list shows commercial micro-architectures from Intel and AMD that we have profiled and used for our experiments. The profile includes typical cache access latencies (namely L1 access latency, L2 access latency, L3 access latency and DRAM-based main memory access latency) and the associativity of all different cache levels for multi-core processors. To simplify result interpretation for the reader, we decided to put results for only Intel's Knights Landing Cache profile in this paper and limited its total number of cores to four. This is a modern CPU architecture used for super-computing applications and the speedup of parallel workloads can create a significant performance improvement in the current parallel applications.

- Intel(R) Core(TM)2 Quad CPU Q6600 @ 2.40GHz (Kentsfield)
- AMD FX(tm)-4350 Quad-Core @ 4.20GHz (Piledriver)
- Intel Xeon Phi 7290 72-Core @ 1.50GHz (Knights Landing)

5.2 Performance Counters

In order to keep track of cache conflicts and their consequences we make use of different performance structures. For each core we count the total number of instruction and data accesses. Each cores also has a counter representing the total delay caused to retrieve the instructions and data. Each Core has a structure that represents invalidation requests to other cores. Every time a thread running on a particular core executes a Write operation on a given cacheline and this cacheline is also accessed by some other core, cache manager invalidates the cacheline in the other core by removing the tag from its cacheline and updates this structure accordingly. This structure can be represented as a matrix shown in Figure 3. Every line corresponds to what each core's structure holds (For example, Line1 shows the number of times Core 0 has send invalidate requests to Core 1, Core 2 and Core 3 respectively). Because each core which updates a cacheline can only invalidate same cacheline present in caches of different cores and not its own cache, we see an empty diagonal in the matrix.

5.2.1 Virtual mapping. Our simulator uses a table of Virtual IDs to allocate threads to physical cores. During the execution of the simulator the mapping is randomly generated in order to avoid aligning biases. It is an one-to-one mapping between the thread id and their physical core. When a migration is performed the corresponding Virtual ID mappings are swapped. This mean that all the threads that where running on a particular physical core are migrated. We could have performed a mapping of a particular thread

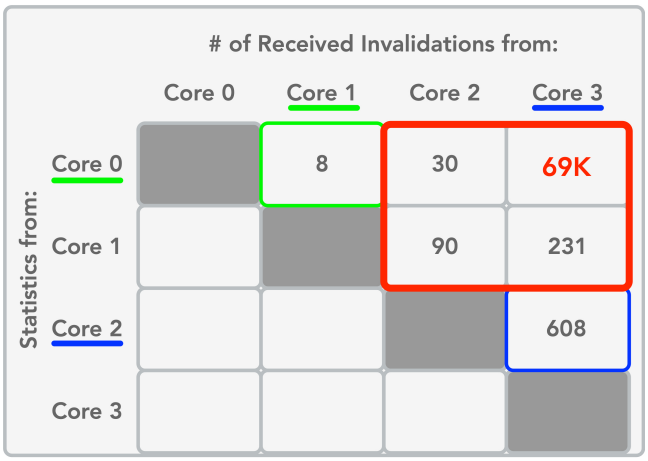

Figure 4: The Core pairs represented in the red box can be migrated, the ones in blue and green alre are taking benefit of the highest level of cache locality

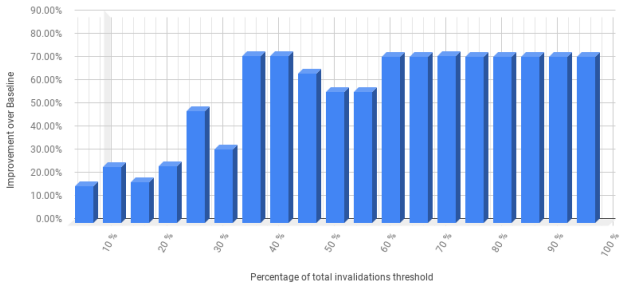

Figure 5: Decrease of average delay compared to baseline by using different values for the invalidation threshold for the Synthetic Workload

to a particular core, but since this algorithm will run in hardware this kind of information would require additional communication between the OS and micro-architecture. Furthermore, this keeps the algorithm as transparent as possible to the OS and the parallel applications.

5.3 Decision Algorithm

The decision algorithm performs core migrations based on the performance counters as shown in Figure 3. Since we are only interested in the core pair dependencies we can perform a diagonal fold of this matrix resulting in a top triangle matrix as shown in Figure 4. This in effect, sums the dependencies between the core pairs. For example, the figure shows in green the dependencies between the physical Core 0 and Core 1 and in blue the dependencies between Core 2 and Core 3. The core pairs in green and blue can be migrated but there is no better location for any of these cores, because they already share the best allowable cache Level (L2). The cores circled in red are the opposite, they represent cores pairs that only share the common Last Level Cache i.e. L3 or DRAM-based main memory. These form good candidates for migration to the other locations i.e. cores or CoreSets. Our algorithm picks the core pair that is responsible for the maximum number of invalidations. This core pair is then compared relative to the total number of cache invalidations since the last (previous) migration, this includes sum of all invalidations including the the core pairs in green and blue. If the value is above the predefined threshold, we proceed with the migration by resetting the counters and swapping the virtual ID of the corespondent Physical cores. This cycle is again repeated at every predefined check interval by the simulator.

6 EVALUATION METHODOLOGY

6.1 Simulation Environment

Our simulations are compiled and executed inside a portable and replicable Docker container that is based on latest version of Ubuntu Linux OS. It also contains the latest version of the Intel PinTool [2] and a pre-compiled version of the PARSEC benchmarking suit [3]. The container can be built by using the Dockerfile available in the Github repository [9].

6.2 Benchmarking

During the development stage, we implemented simple synthetic test applications to confirm the proper execution of the simulator. One of the most challenging part was to pin specific execution threads to use a particular core or CoreSet. In case of PinTool, we were only able to extract the thread ID of the execution thread and assign it to run on a specific Core. This also means that the OS is not involved in the load balancing process which leaves us to choose the most appropriate Core to execute our Threads. We have chosen to use Round-Robin allocation of the threads to the available cores. The benchmarks applications were also set to use only four Cores. Some benchmarks only ensure that this setting only sets the minimum number of threads when passed to the PARSEC benchmark manager.

6.3 Algorithm Tuning

The 'CHECK_INTERVAL' variable is a tunable parameter used in our simulator. It can be adjusted by specifying the "-ci" parameter in the PINTool CacheDoge [9] utility. The variable is used to decide the frequency at which the the CacheDoge Algorithm runs. This parameter also can be seen as a smoothing attribute of the algorithm. In case of high contention to the same cacheline address, a larger check interval allows other threads to increase their invalidations counters and avoid thread swaps that do not truly represent the global dependency map. On the other hand very large numbers might not catch all the available opportunities to perform a thread swap, leading to poor performance.

The 'INVALIDATION_RATIO_THRESHOLD' variable ("-irt") is another tunable parameter used to specifying the minimum threshold condition to execute the thread migration as explained before.

7 RESULTS

In this section we present some of the relevant results we observed during benchmarking our solution. Each test was averaged for at least 10 separate runs of the simulator. For the PARSEC benchmarks we have made use of the simdev dataset since others data-sets

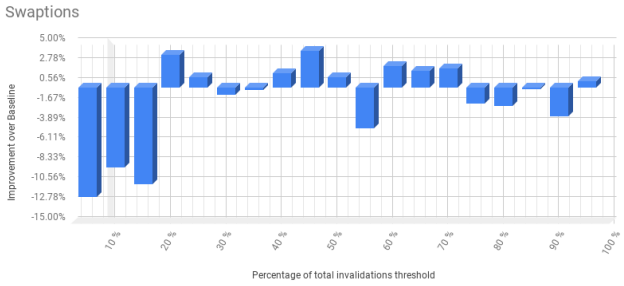

Figure 6: Decrease of average delay compared to baseline by using different invalidation thresholds for the PARSEC swaptions Workload

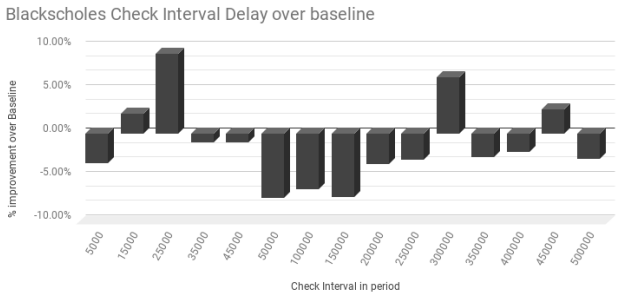

Figure 7: Blackscholes Check Interval impact on the average total delay compared to the baseline implementation

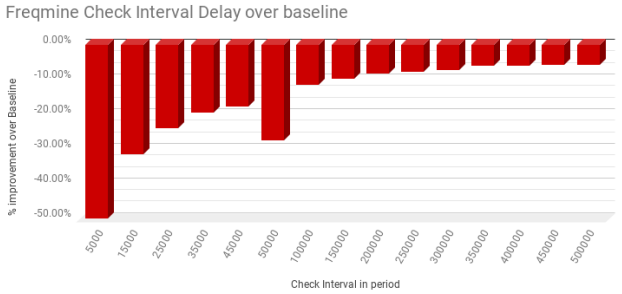

Figure 8: Freqmine Check Interval impact on the average total delay compared to the baseline implementation

required multiple hours per run and were not feasible given the available time.

7.1 Detecting Optimal Parameters

We have tested our solution with synthetic benchmarks that simulate two threads trying to get access the same variable, as represented in Figure 1. The same type of workload can be observed when multiple threads try to access synchronization primitives such as Locks or Atomic operations.

In order to fine tune our algorithm, we used our synthetic benchmarks to find the optimal tuning parameters.

Figure 5 shows how the invalidation threshold parameter affects the average delay observed compared to the baseline. As we can see, there is a significant advantage in using CacheDoge and a threshold above 35% shows the maximum gain (with a check interval of 25k Instructions).

Figure 6 on the other hand, shows how the invalidation threshold affects the average observed delay of a PARSEC workload (swaptions). For this workload a small invalidation threshold value will have a considerable negative effect on the performance. This is expected since a small value might cause a very high amount of migrations, as shown in Figure 5. We also expected to see a positive increase in performance after the 30% mark but this is not the case. The swaptions workload demonstrates in average a negative performance profile with some cases where the performance is positive and above the average. This is also similar to others workloads not shown here.

Next we tried to tune for the optimal check interval parameter. The check interval did not affect our synthetic results. Thus we have turned to the PARSEC benchmarking suite.

Figure 7 and Figure 8 are two different PARSEC workloads that have received the same input parameters (including invalidation threshold of 35% predetermined in the previous experiments). We can observe that the performance impact of CacheDoge on both of these workloads is mostly negative. Additionally, the frequemine workload only observes negative performance for this invalidation threshold. This takes us to believe that each workloads would have their optimal parameters that would have to be found using extensive search.

Instead of using brutal force and iterate over all ranges of the two mentioned tunable variables over all PARSEC Workloads we tried to find a small subset that would perform reasonable well for all workloads. Unfortunately, we were not able to find a particular value for both tunable variables that would showcase a positive performance impact for all workloads. We have tried the a manual gradient descent approach to find the best settings but where still getting below 5% overall performance improvement. One idea would be to classify the workloads into different categories based on their optimal parameters. Identify the categories during run time and apply these parameters.

When specific Workloads where targeted where able to get a maximum performance boost of 19.5% for the blacksholes workload for the MPKI (Misses per Kilo Instructions) metric (shown in Figure 10). We then used the same configuration for the rest of the workloads in the PARSEC benchmark suit as shown in Figure 9 and 10 we can see how poorly a static values performs across different workloads. The average improvement in delay across all workloads was 2.83% compared to isolated workloads of up to 20% in some cases.

8 FUTURE WORK

For future work, this project should build on the premise that workloads are dynamic, thus:

- Fine tune our CacheDoge for dynamic workloads to achieve better performance results across all workloads.

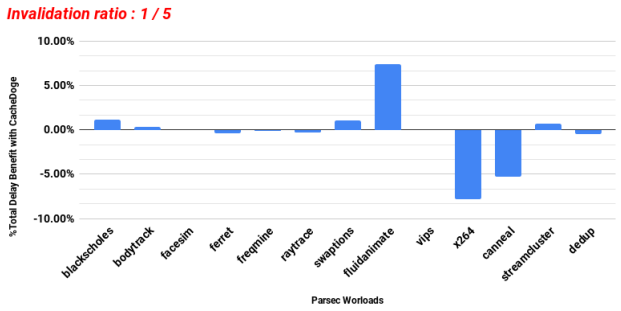

Figure 9: PARSEC Workloads average delay improvement over baseline (Invalidation Threshold: 20% and Check Interval 30k Instructions)

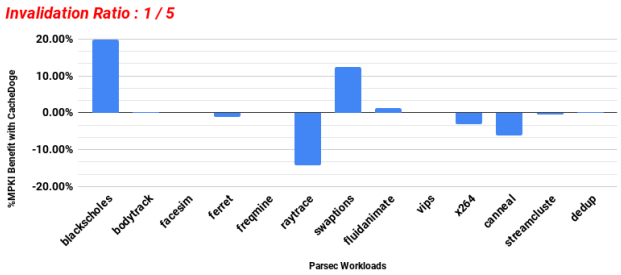

Figure 10: PARSEC Workloads Misses per Kilo Instructions (MPKI) improvement over baseline (Invalidation Threshold: 20% and Check Interval 30k Instructions)

- Test our algorithm for higher number of cores (more than sixteen cores) of Intel's Knights Landing architecture. We believe that with more number of cores i.e. 72 cores, sharing same cachelines, invalidation requests may increase tremendously. Due to which they may start using significant bus bandwidth, leading to steep drop in parallel performance. In such scenarios, CacheDoge may prove out be an effective solution.
- Test how the implemented memory smoothing algorithm may improves performance by maintaining some history about the migration patterns
- Classify workloads into categories and use the best possible profile (invalidation threshold, checking interval) to optimize parallel performance.
- One important point which we all may agree upon is that it is impossible to achieve a generic set of tunable parameters for all workloads. Reason being each workload has different functional resource needs, which may not be completely satisfied by the underlying micro-architecture. There is also noise in results introduced by these workloads because they may not be optimized for the underlying micro-architecture. Sometimes, they are just bound to give bad results. We need to quantify their noise impact on our results.
- Implement more accurate CPU eviction mechanisms and showcase what impact they have on the results.

9 CONCLUSION

In this paper, we present a novel micro-architecture component, CacheDoge, which detects-and-fixes cache coherence bottlenecks posed due to false sharing between parallel applications running on shared memory multi-core processors. The unique selling point of CacheDoge is that its fully implemented in micro-architecture and does not require any software support, due to which it remains transparent to programmers. Our implementation of CacheDoge is the only solution in the area of mitigating cache coherency bottlenecks which does not require any external support from the software i.e. OS and applications. It is a low cost hardware solution which can be easily adopted in the future micro-architectures to improve parallel application performance. The parallel applications and OS can continue the way they are right now. But unfortunately, we were not able to showcase the benefits of CacheDoge under diverse workloads. As also shown by the results obtained from the PARSEC benchmark suite, where CacheDoge fails to impress. The main weakness of CacheDoge is its pure, simplistic and static dependency on the calibration parameters that might be good for one particular workload but may not suit to accommodate all general purpose workloads. But if desired one can tune this weakness into strength by implementing it in non-general purpose multi-core micro-architectures. CacheDoge's calibration parameters can then be fine tuned for specific parallel workloads used in runtime.

REFERENCES

[1] James H. Anderson and John M. Calandrino. 2006. Parallel Task Scheduling on Multicore Platforms. *SIGBED Rev.* 3, 1 (Jan. 2006), 1–6. https://doi.org/10.1145/1279711.1279713

[2] Moshe Bach, Mark Charney, Robert Cohn, Tevi Devor, Elena Demikovsky, Kim Hazelwood, Aamer Jaleel, Chi-Keung Luk, Gail Lyons, Harish Patil, and Ady Tal. 2010. Analyzing Parallel Programs with Pin. 43, 3 (March 2010), 34–41.

[3] Christian Bienia. 2011. *Benchmarking Modern Multiprocessors.* Ph.D. Dissertation. Princeton University.

[4] Eric A. Brewer, Chrysanthos N. Dellarocas, Adrian Colbrook, and William E. Weihl. 1991. *PROTEUS: A High-Performance Parallel-Architecture Simulator.* Technical Report.

[5] Doug Burger, Todd M. Austin, and Steve Bennett. 1996. *Evaluating Future Microprocessors: the SimpleScalar Tool Set.* Technical Report.

[6] Shimin Chen, Phillip B. Gibbons, Michael Kozuch, Vasileios Liaskovitis, Anastassia Ailamaki, Guy E. Blelloch, Babak Falsafi, Limor Fix, Nikos Hardavellas, Todd C. Mowry, and Chris Wilkerson. 2007. Scheduling Threads for Constructive Cache Sharing on CMPs. In *Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '07).* ACM, New York, NY, USA, 105–115. https://doi.org/10.1145/1248377.1248396

[7] Jaydeep Marathe, Frank Mueller, and Bronis R. de Supinski. 2006. Analysis of Cache-coherence Bottlenecks with Hybrid Hardware/Software Techniques. *ACM Trans. Archit. Code Optim.* 3, 4 (Dec. 2006), 390–423. https://doi.org/10.1145/1187976.1187978

[8] Anthony-Trung Nguyen, Maged M. Michael, Arun Sharma, and Josep Torrellas. 1996. The Augmint multiprocessor simulation toolkit for Intel x86 architectures. In *1996 International Conference on Computer Design (ICCD '96), VLSI in Computers and Processors, October 7-9, 1996, Austin, TX, USA, Proceedings.* 486–490. https://doi.org/10.1109/ICCD.1996.563597

[9] Artur Balanuta Nidhi Bhatia. 2018. https://github.com/ABalanuta/CacheDogeSim. https://github.com/ABalanuta/CacheDogeSim

[10] Mendel Rosenblum, Stephen A. Herrod, Emmett Witchel, and Anoop Gupta. 1995. Complete Computer System Simulation: The SimOS Approach. *IEEE Parallel and Distributed Technology* 3 (1995), 34–43.