DOPPELGANGER CACHE

A ZSim Implementation

Purna Doddapanen, Tanmay Verma

Computer Architecture | Texas A&M University

ABSTRACT

The Internet of Things boom over the past decade has resulted in a significant increase in the amount of data collected. The analysis of this large-scale data cannot be finished within reasonable time frames and power efficient is not possible with traditional computing methods due to the sheer immensity of the data. Approximate computing shows promise for tackling many of the challenges presented by Big Data by introducing a small margin of error in a program's output. The Doppelgänger cache presents a novel cache specifically for approximate computing. Multiple tags pointing to a single approximate data reduce the area occupied by the Last-Level-Cache. Subsequently, this results in lower power consumption while having a reasonably small performance hit. The Doppelgänger cache shows a 1.51x reduction in area while suffering only a 4.5% hit in performance.

Total Length: 21 pages without appendix and abstract

TABLE OF CONTENTS

INTRODUCTION

The evolution of the microprocessor from the modest Intel 4004 to support a printing calculator to the behemoth Intel Xeon's which power the modern digital era was made possible through a series of fundamental breakthroughs [1]. Of these breakthroughs, the first and most vital was the transistor and THE successive shrinking of its size to fit more functional units on a given area. As successive generations of chips were produced, performance walls are erected as a result of component incompatibility. Particularly, the microprocessor had significantly higher performance than memory resulting in waiting periods for data to be fetched for computing. The second important breakthrough, caching, occurred to mitigate this cycle cost of fetching required data from memory. By storing data that is going to be required for upcoming instructions directly on the faster performing chip rather than memory, fewer cycles are wasted to access necessary data. Since its inception, the cache has received significant research attention and industry attention as an improvement in cache performance meant an overall increase in microprocessor's performance. One branch of improvements focused on adding more levels to the cache. This resulted in the modern memory hierarchy of having a Level 1(L1), Level 2(L2) and a Level 3(L3) cache present on the microprocessors die. This further improved the throughput of a microprocessor but came at the cost of having a large block of chip area reserved solely for the cache. Cache innovation still continues today as computing trends and needs dictate new requirements to efficiently handle modern workloads; particularly Big Data workloads.

CHALLENGES IN BIG DATA AND APPROXIMATE COMPUTING AS A SOLUTION

With the Internet of Things(IoT) boom over the past decade, an abundance of data is pouring into data centers across the globe. Processing this data in the 'Cloud' provides valuable insights and improvements in areas like medical analytics, smart cities or crime prevention. However, traditional computing techniques are unable to cope with Big Data in an energy-efficient manner. Data centers are powered by power-hungry server microprocessors such as the Intel Xeon. Computer researchers across the globe are addressing this problem with a myriad of techniques at all computer system levels. Approximate Computing has been proposed as a promising alternative to tackle the challenges presented by Big Data[2].

Approximate Computing is the concept that some programs have a degree of error which can be tolerated without drastically altering the programs functionality. For example, pixel color is defined by Red, Green and Blue(RGB) values ranging from 0-256. Computing on a value that differs an approximate value differing by a little (i.e. $\pm$ 10) from the exact value would not have a large impact on an image processing algorithm. By leveraging this approximate similarity that is present in many Big Data workloads, modifications can be made to existing cache models to create more energy-efficient and area-efficient caches. One such instance that demonstrates this principal is San Miguel et al proposed Doppelgänger Cache as the Last Level Cache(LLC) specifically for Approximate Computing.

PREVIOUS WORK

The first major issue in approximate computing was finding which family of data are actually conducive to approximate computing. After identification, the approximate computation needed to be compatible with modern computing schema. This posed a significant challenge because it was difficult for compilers to automatically label a piece of data approximate or precise. Programmer annotations to qualify a block of data as approximate allowed for differentiation between the types [4]. This requires extra work from the high-level programmer's end but allows for greater throughput and efficiency for their program. With the ability to distinguish approximate data from precise data, San Miguel et al. proposed *load value approximations* to allow approximations of data to serve as substitutes for actual memory loads, resulting in no latency from fetching data from memory [5]. Fetched data is then used to retrain its approximation for better accuracy on its next execution. Other work on de-duplication showed that the same data block in cache is often replicated multiple times over. One instance could not be associated with the other due to the nature of tag matching for a precise associative cache. The merits of these particular previous works are exploited in the Doppelgänger cache[6]. It also shifts the focus from inter-block approximation to intra-block approximation by using a hashing function across the individual data elements of a cache block.

PROPOSED SOLUTION: DOPPELGÄNGER CACHE

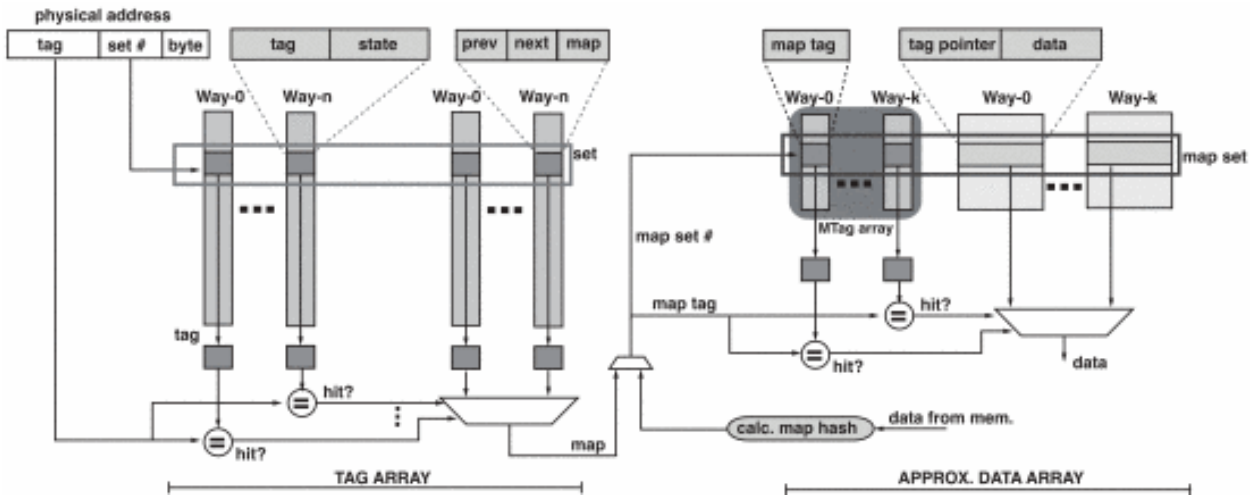

Figure 1: Proposed Doppelgänger Idea. Source: Doppelgänger: A Cache for Approximate Computing, MICRO 2015.

The basis of the Doppelgänger Cache is the assumption that content duplication, where a missed block is already in the cache but with a different tag, exists across any given cache. Since the data already exists in the cache, it is more efficient to simply access the other block instead of wasting cycles fetching the exact same value from memory [3]. The Doppelgänger Cache incorporates this nuance with approximate computing and postulates that "multiple approximately similar values exist across cache blocks and can be stored as a single data block in the Cache." The slight variance in the different numbers results in some error but allows microprocessor architects to save on the size of the LLC. The shrinkage of the LLC directly results in vital energy savings which are a strong industry evaluation standard for a chip's performance.

The basic building block of the Doppelgänger cache is the use of multiple approximate tags pointing to the same approximate data block. Figure 2 summarizes this basic block.

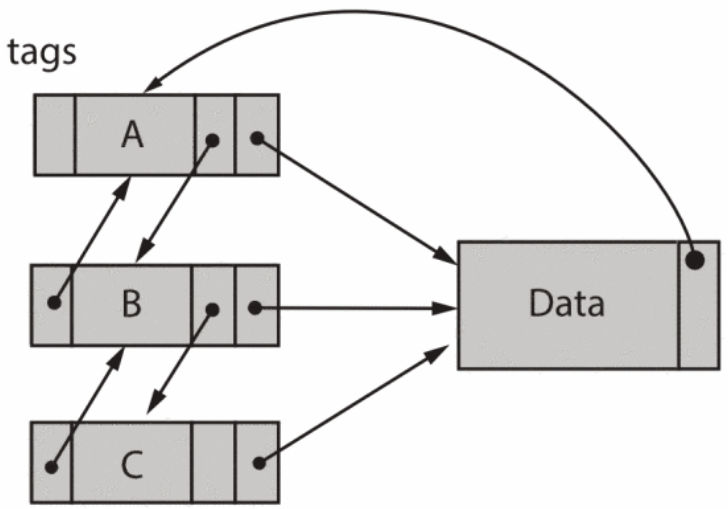

Figure 2: Proposed Doppelgänger Idea. Source: Doppelgänger: A Cache for Approximate Computing, MICRO 2015.

San Miguel et al. proposed in their paper. They utilize a double linked list of approximate tags to maintain the multi-tag format. A traditional(precise) cache has a single tag associated with a single data block. The data is generally bought in blocks due to temporal and spatial locality and each block requires significantly larger than a tag. By eliminating the 1-tag to 1-data-block requirement, fewer data blocks are required. The cost of each tag in the linked list is 77-bits and the cost of the map tag array(MTag) is another 38 bits per 64-byte data approximate data block.

Each tag in the linked list contains a previous, next, map value, address value and state bits. The next and previous values are needed to maintain the doubly-linked list. The address values are necessary for identification and the state bits(2-bits) are required for coherence protocols. Finally, the map value(14-bits) of the tag is created from a hash function and serves to index into the data array. The MTag contains index value in the lower 14 bits and the upper 24 bits are used to tag match like in a traditional cache. The data blocks are 64 bytes wide. The cache also requires a traditional precise cache working in conjunction with it in order to facilitate non-approximate data.

This Doppelganger implementation requires computability to the process of lookups, insertions, writes, replacements, and coherence. It also requires a special hash functionality to identify and map approximately similar data. The entire Doppelgänger cache and necessary supporting functionality was implemented by modifying the Z-

Sim simulator. The rest of this section will describe the above-mentioned cache functionality required for correct Doppelgänger cache behavior and corresponding Z-sim modifications.

LOOKUPS

Lookup is the procedure to see if a specific data is present in the cache. When a request is received from its lower level cache, traditional tag matching is performed to see if the necessary data block current resides in the cash. If a tag is matched, it is considered a cache hit. Then, the tag's map value is used to index its corresponding block in the MTag array and subsequently, the required data block is supplied. If a tag is not matched, this is considered a cache miss and we retrieve data from memory.

INSERTIONS

Insertion is the procedure to bring and place a data block into the cache when a cache miss happens. A traditional cache would simply replace a block in the cache with a new block. The Doppelgänger cache has the extra step of checking for a similarly approximate block already present. The data is first sent off to the L2 cache to minimize latency and the approximate check is performed of the critical path. The new values index and map are calculated and if similar data already exists in the Doppelgänger cache, the new tag is added to the head its doubly-linked list. If no similar data is present, utilize the microprocessor's replacement policy to evict a block and its associated tags. Then add the new data block with its calculated map, and set its tag's pointers to NULL as it is a single element.

WRITES

A write occurs when the value has been updated by the microprocessor path. The map of the value being written is calculated. If new map is the same as old map, dirty bit of the tag is set and no more action is required. If new map is different than old map, further map matching is performed to see if the new map is already in cache. If the new map is in cache, we move tags to the new map value. If the new map isn't in the cache, we insert it into an empty block.

REPLACEMENTS

Replacements occur when we are evicting data or tags in the Doppelgänger cache. When a tag needs to be evicted, the data block does not get evicted alongside like in a traditional cache. This is the case when there is at least one more tag still pointing to the data block. A data block gets evicted only when no tags are associated with it. When a tag gets evicted, we reattach the tags m_next and m_prev pointers as necessary. For data evictions, we also consider writing back to memory and updating coherence bits to invalidate as necessary.

COHERENCE

Coherence is necessary in all multi-core microprocessors to maintain data integrity. Unlike a traditional cache where coherence bits are placed on the data blocks, the Doppelgänger places coherence bits on a tag-by-tag basis instead. This is necessary because while one of the tags may be invalid, the other tags could still consider the value in the data block to be up to date. Naturally, placing the coherence on the data block would result in incorrect invalidations of data.

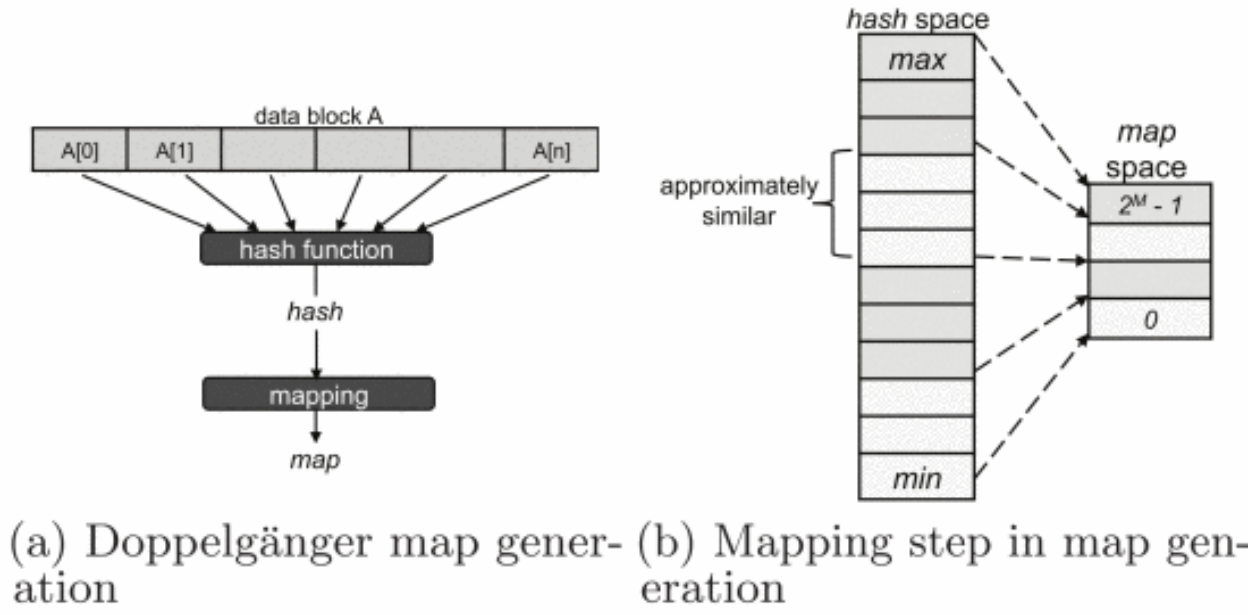

Figure 3: Hash and Map Generation on Cache Replacements and Write-backs

Source: Doppelgänger: A Cache for Approximate Computing, MICRO 2015.

Approximately Similar Mapping

In order to keep only one instance of one approximately similar data, hashing functions are necessary to quickly calculate a blocks map. This has to be done one all replacements and write-backs. Figure 3 illustrates the hashing function and its successive conversion to map used in the Doppelgänger cache. The generation of the map using the hash function avoids blindly searching through all tags and data blocks to find a map. The hashing function utilized for the Doppelgänger cache takes the entire data block as the input and outputs the average and the range of the block. These hash values effectively get splits into M equally spaced values where M is obtained through programmer annotations. All values in particular range of M share the same map space and thus obtain the same map value. The average is mapped to the lower bits of the map whereas the range is mapped to the higher bits of the map. The map value ranges from 12-14 bits. The smaller 12-bit map results in more blocks being deemed similar and mapped to the same data while the 14-bit map gives fewer maps to the same data at the cost of hardware.

UniDoppelgänger Cache

The Doppelgänger Cache described above has a significant demerit as far as performance is concerned. It allocates a significant portion off cache to Approximate data exclusively. This way reducing the actual cache available for workloads having smaller approximate region accesses. The solution to this problem is provided by a unified precise and doppelganger cache portions. There is an additional bit in tag portion which determines whether the data is approximate or not. If approximate the tag will have its own data line otherwise it will share the line with other blocks having identical maps. This way the entire data block can be utilized even for the workloads with small approximate regions.

DOPPELGÄNGER IMPLEMENTATION IN ZSIM

As the part of this project effort, we implemented Doppelgänger Cache described above in ZSim simulator to analyze its performance impact. We wanted to analyze how much performance degradation can be expected. Simulation results (L3 misses and the Cycles consumed) will give us an idea of which benchmarks are affected. So our implementation within ZSim framework is described below. The operations described before for Doppelgänger cache are emulated as below:

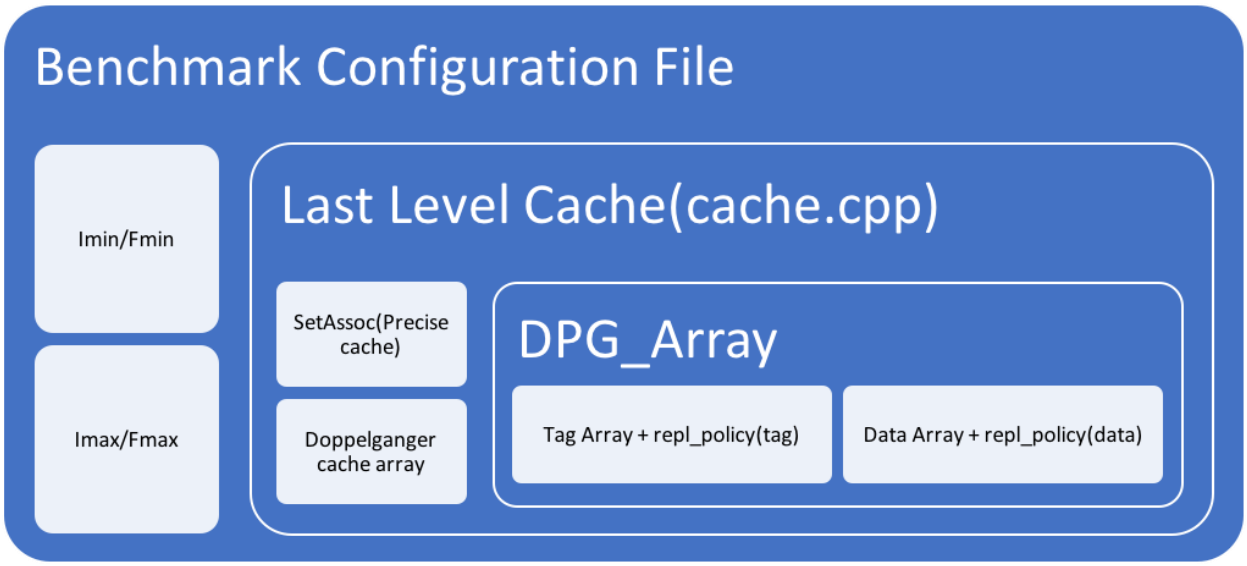

Figure 4: ZSim Implementation of Doppelganger Cache Oveview

Benchmark Configuration File(*.cfg): We provide the maximum and minimum values that we expect in the approximal region of our benchmark here. This is done to generate optimal hash functions for the data stored in our cache line.

Last Level Cache(Cache): The last level cache comprise of two cache array types, namely SetAssoc(for precise portion of the cache) and DPG_Array. In the Last Level Cache, if the access is to approximal region then the request is routed to DP_Array otherwise the it goes to SetAssoc array and processed as usual.

DoppelGanger Cache(DPG_Array): This cache array class is entirely described in file doppelganger_cache.cpp file. It is built upon SetAssoc cache array but differ in its structure to emulate the behavior of doppelganger cache. It maintains two separate arrays for storing tags and data elements. The class needs two separate instances for replacement policy as well to correctly select right candidate for eviction. Each Tag Array element consists of following fields:

- next: A pointer to next tag_id
- prev: A pointer to prev tag_id

- addr: Field to hold the actual lineAddr
- data_id: Pointer to an element in data array to which the tag is matched

The Data Array element on the other hand has the following fields:

- map_value: The map value of data block represented by the element
- tag_head_ptr: A pointer to the head of the Tag Linked List which are matched with the data element.

The member functions of this class are:

- remove_tag(uint32_t tagid): This function removes the mentioned tag_id from its current Linked List and returns the previous member.
- add_tag(int32_t tag_id, int32_t data_id): Adds the tag_id to the list pointed by the tag_head_id of the data element making it the new head of the list/
- lookup_data(const int32_t map): Analogous to lookup to tag_array called from cache.cpp. This function treats map as the lineAddr while pointing to a data array element.
- get_free_data(const int32_t map, CC* cc_dpg, const MemReq* req, uint64_t respCycle): Search for a free line in data array with the corresponding map value. If none free, then calls evict data with the rankCand to generate a free line.
- get_free_tag(const Address lineAddr, CC* cc_dpg, const MemReq* req, uint64_t respCycle): Search for free tag. If none found then calls evict tag with the rankCand to generate a free line.
- evict_data(const int32_t data_id, CC* cc_dpg, const MemReq* req, uint64_t respCycle): This function calls upon evict_tag in loops for all elements of the linked list pointing to the data element.
- evict_tag(const int32_t tag_id, CC* cc_dpg, const MemReq* req, uint64_t respCycle) : Instead of evicting cache line from Cache Object, we are evicting lines from this function as multiple evictions might be needed for a single access in DoppelGanger Cache. For instance, if data array got full , evict_data will call a series of evictions.
- lookup(const Address lineAddr, const MemReq* req, bool updateReplacement, CC* cc_dpg, uint64_t respCycle): It is one of the necessary function needed in CacheArrays. Our implementation, matches the lineAddr with tag_array[tag_id].addr to determine whether it is a hit or miss. Upon a tag_hit, the map is recalculated to see whether the linked list structure needs to be updated or not. It also updates the replacement policy counters for both tag and data arrays.
- preinsert(const Address lineAddr, const MemReq* req, CC* cc_dpg, uint64_t respCycle): If the access was a miss, this functions search for an ideal tag element and data element in the cache to store the new line. It also sets up the Linked list and pointers.
- postinsert(const Address lineAddr, const MemReq* req, uint32_t candidate_t): This function carries out the final insertion of lineAddr into tag_id returned by preinsert.

- get_map(const Address lineAddr): This function calculates the map value of the accessed data block as described in the paper. The authors didn't clearly state the expected range they assumed for each benchmark. This is a major pointer due to which our observations might differ.

LOOKUPS

Upon a lookup, the lookup() function in doppelganger cache is called which compares the lineAddr with the tag_element.addr to determine a hit. Upon a successful hit in tag array we can be sure that the data is in the cache and we return the tag_id. In case of a miss '-1' is returned.

INSERTIONS

The insertions are handled by preinsert and postinsert routines of the Doppelgänger cache. When a miss is encountered, preinsert() searches for an available tag_element in the tag array and a corresponding data_element in the data array. The map value of the block to be inserted is calculated to obtain the set of viable data elements. If a data block with the identical map value is already present then the tag is inserted to the linked list structure pointing to this data block. If no data block with matching map is found then a a new data element is allocated to the tag.

WRITES

The write is handled in the similar way as reads in lookup() routine. The additional step involves matching the map with every access(write type). Whenever a change in map value is registered, the tag is removed from the linked list pointing to the stale data block to the new linked list(given data block with similar map already existed) or directly points to a newly associated data block.

REPLACEMENTS

The evictions are handled in our implementation whenever lookup() and preinsert() routines search for available tag or data elements to allocate. We have two LRU type replacement policies storing the counters for each tag and data block. Whenever a replacement is needed, the rankCand function returns the Least Recently Used line for eviction. While evicting the tag, it is removed from the linked list structure and the line is made available. If it was the only

tag pointing to the data block, the data block is also freed. On the other hand, when a data block needs eviction, the process_evict() routine is called on all tags pointing to the data block and finally freeing up the data element.

COHERENCE

The coherence state is associated with each tag. The accesses, evictions and invalidations are maintained in tag-by-tag basis. The data array block is hidden from the implementation of coherency control. It is important to note that the coherency_control object is shared by precise and doppelganger cache hence instantiated 2*numLines in number. The lower half is allocated to precise cache and is accessed directly with tag_id whereas the upper half is dedicated to doppelganger cache and is accessed as (numLines + tag_id).

SIMULATION SETUP

We will describe our System Configuration used for running the benchmarks:

Given below is our baseline system which we use to perform the comparative study:

Processor	8 OoO Westmere cores
Private L1 cache(Instruction)	32KB, 4-way, 3 cycle Latency
Private L1 cache(Data)	32KB, 8-way, 4 cycle Latency
Private L2 cache	256KB, 8-way, 7 cycles Latency
Shared LLC	8-banks, 16MB, 16way, 27 Latency
Cache Coherence	MESI

The shared LLC configuration with DoppelGanger Cache is described below:

Precise Cache	8MB, 16-way, LRU, 27 Latency
Doppelganger Tag Array	8MB tag-equivalent (128K tags), 16-way LRU
Doppelganger Data Array	64K data lines(½ the Tag Array size), 16-way LRU
Doppelganger Access Latency	27 cycles
Map Space	12-bit

Some of the configuration is taken directly from the paper, which gave best area savings without greatly hurting performance.

The range of data present in approximate region:

Benchmarks	**Max Value**	**Min Value**
Blackscholes	110.00	0.00
Canneal	400	0
Fluidanimate	0.10	-0.10
Streamcluster	1.00	0.00
Swaptions	5.00	0.00
x264	No Approximate Region	

RESULTS

With the Doppelgänger's cache implemented using the ZSim simulator, it was evaluated using PARSEC benchmarks. Each benchmark had three different workload sizes, small medium and large and were run for a fully precise cache and the Doppelgänger cache. The programmer annotations were bought in through modifications of the Pin tool and changes to ZSim's tree structure to detect a benchmark's data is approximate or precise. The functionality and code for detecting an approximate block, along with the annotated benchmarks were provided by the TA. For our results, we decided to focus on three main ideas to evaluate the merits of the Doppelgänger Cache, throughput, misses per kilo instruction, and area.

THROUGHPUT

Throughput measures the number of instructions completed by the microprocessor in a given period of time and is a measure of the systems performance. Throughput is chosen as a standard measure of performance as the ultimate goal of the microprocessor is to complete the instructions as quickly as possible. The number of content cycles taken to complete a benchmark were utilized to represent this metric. For each benchmark, a small, a medium, and a large size workload was run. This was to help provide insight into if and how the size of a workload would have an effect on the system's performance with a Doppelgänger cache.

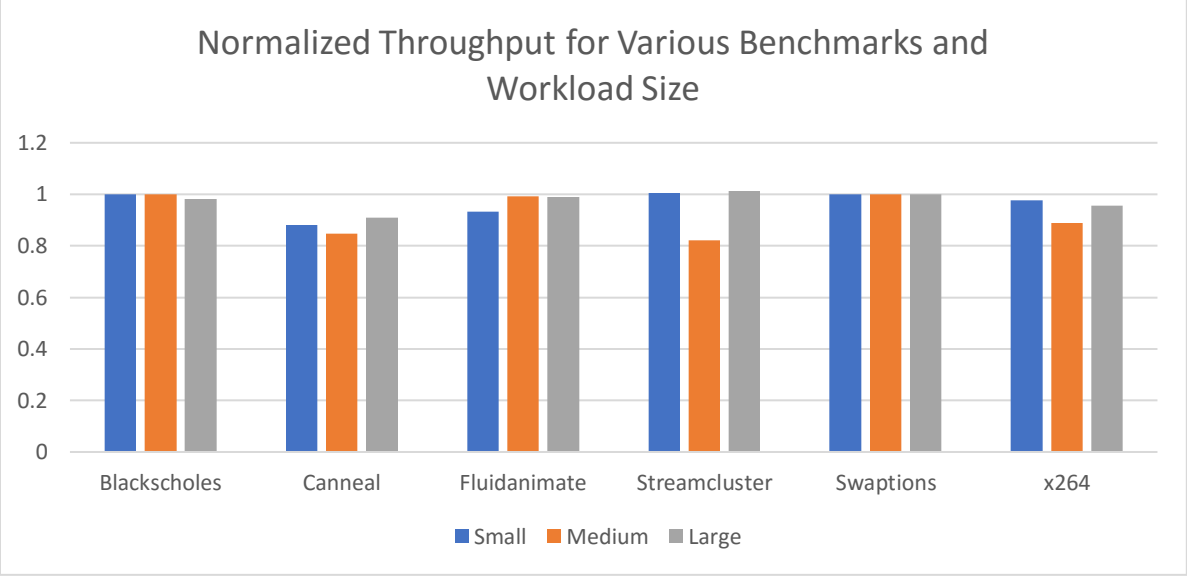

Figure 5: Normalized Throughput between Doppelgänger and Fully Precise Cache

Figure 5 displays the normalized throughput pf a fully precise cache to the Doppelgänger cache for the benchmarks we ran. The higher the bar, the better the performance of the cache with its parameters. The general trend shows that Doppelgänger cache performs as well or worse than the fully precise cache. Blackscholes and Swaptions show no throughput degradation across any of the benchmarks. Streamcluster exhibits inconsistent behavior with the medium sized workload shows significant degradation in performance but not for the small and large workloads. Reasons for this discrepancy will be discussed in the evaluation section of the report. Canneal shows consistent degradation across all sizes of workloads. A consistent pattern in benchmarks exhibiting performance loss is that the medium sized workloads show lower system performance than large and small workloads. As such, there seems to be no glaring relationship between the workload size and performance.

Misses per Kilo Instructions(MPKI)

MPKI was chosen as a measure of performance because it represents the number of misses in a benchmark. Cache misses have the largest effect on a systems performance as there is high latency when retrieving data from memory. This is particularly important in the analysis of a Doppelgänger cache's performance because we are stricter restrictions on the approximate storage blocks.

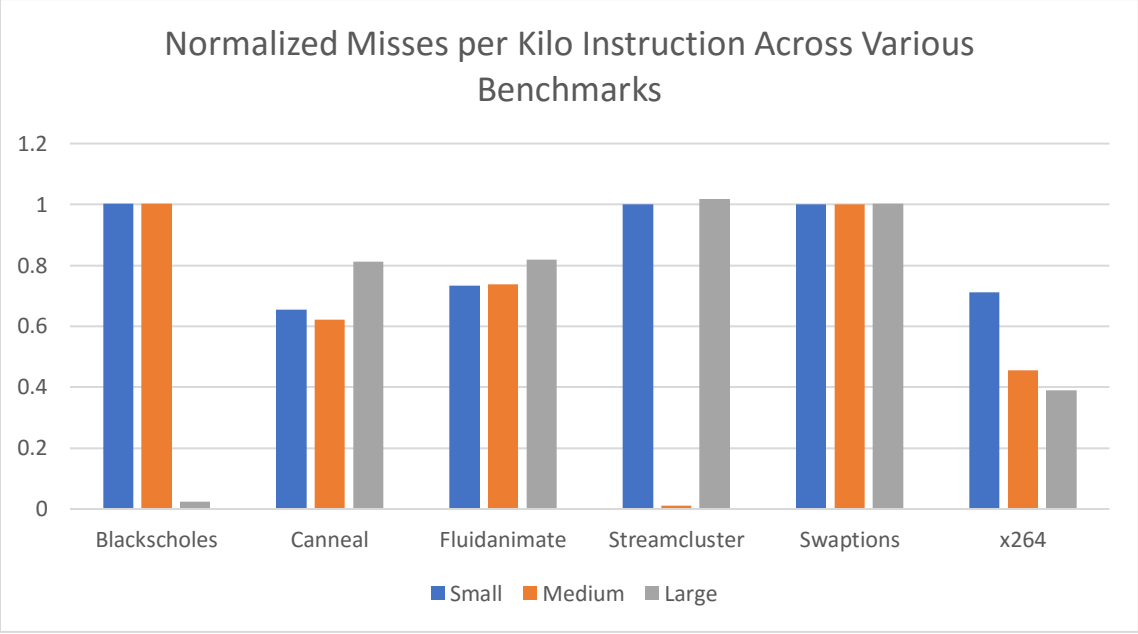

Figure 6: Normalized MPKI for Various Benchmarks and Workload Size.

Figure 6 summarizes the MPKI for the various benchmarks over varying workload sizes. The larger the magnitude of a bar, the better the performance of the Doppelgänger cache. The Doppelgänger cache consistently incurs greater miss rates across the various benchmarks. Blackscholes performs well for both small and medium workloads and indicates a significantly larger miss rate for the large workload. X264 performance degrades as the workload becomes larger. Canneal shows improvement between the medium and the large workloads. Similar to the throughput, Streamcluster's medium workload indicates a large miss rate.

AREA

The area of the system was calculated based on our own system configuration. It was performed with the naïve assumptions that we can calculate the number of bits needed to implement the Doppelgänger cache system in ZSim. A MATLAB script was written to obtain these values quickly in case the size of our cache was changed. As such we do not take into account support hardware like multiplexers and functional units for calculating Doppelgänger functionality. The following series equation was used to find the number of bits in our Doppelgänger cache:

$$Doppelganger\ Cache\ Size = \ Doppelganger\ Size + Precise\ Size$$

where

$$Doppelganger\ Size = Size\ per\ linked\ list\ tag * Number\ of\ tags$$

$$Precise\ Size = (Map\ Size + Block\ Size) * Number\ of\ lines$$

and

$$Size\ per\ linked\ list\ tag = tag + m_{next} + m_{previous} + map + coherence\ bits$$

We assume a 32-bit system and subtract 4-bits for 16-way set associativity and subtract another 4-bits for 16 byte-addressing to address value. The m_next and m_previous values are also derived by obtaining the $\log_2$size of 128K for 17-bits each. There are also 64K data lines which accessed by 16-bits of map. As Doppelgänger maintains coherency on a per-tag basis, we added 2 more bits for the MESI protocol. Each line of the precise cache requires

the 16-bit map value and 512 bits for data. This results in 86-bits for each linked list tag and 528-bits for each data line.

For a precise cache, the calculation for the number of bits required is more straightforward.

$$Bits\ per\ Cache\ Line = Tag\ Length + Block\ Size$$

We simply subtract byte-addressing bits and set associativity bits to obtain a 24-bit tag line and add 512 bits for a 64-byte addressable block. Next, we multiply by the number of block lines we can fit in our cache to obtain the total size.

$$Precise\ Only\ Cache\ Size = Number\ of\ Block\ Lines * Block\ Size + Tag\ Size)$$

The finds for the number of bits required for each component is summarized in Table 1 and Table 2 in the Appendix.

With obtaining number of bits required for the Doppelgänger and fully precise cache, we can utilize a naïve calculation to compute the amount of area improvement.

$$Area\ Savings = \frac{Area\ of\ Fully\ Precise\ Cache}{Area\ of\ Doppleganger\ Cache} = \frac{(Fully\ Precise\ Bits)^2}{(Dopplegeganger\ bits)^2} = 1.51$$

We have area savings of approximately 1.51x for our implementation of the Doppelgänger cache.

EVALUATION

The Doppelgänger cache occupies a smaller LLC area. This is made possible by having fewer data blocks than a fully traditional precise cache by having multiple tags pointing to the same block. There are two factors which determine the level of degradation while using Doppelganger cache. Firstly, the percentage of accesses going to Doppelganger cache as opposed to Precise. Secondly, the size of their working memory where the pattern repeats. If no accesses are made to doppelganger cache, then it is similar to running the benchmark effectively on a 8MB LLC cache.

THROUGHPUT AND MPKI

The throughput of the cache across all benchmarks and workload sizes decreased by 4.5% corresponding to a runtime increase of 5.1%. All the benchmarks when run on Doppelgänger cache registers a degradation in comparison to a fully precise cache. The MPKI of blackscholes(large) and streamcluster(medium) when run on Doppelgänger cache are much larger than the same run on baseline. This is because the working memory for these loads fits into the 16MB cache and incurs lesser misses. The streamcluster(large) returns back to similar performance as both the Doppelgänger Cache and Baseline cache performs equally bad for this working memory. This can be seen in the following graphs showing absolute values for the MPKI. X264's performance was significantly worse as workload increased because it did not have any approximate regions.

When the MPKI increase, throughput degrades as memory latency increases. However this was not the case for the large Blackscholes workload. Normalization, although a common practice, presented an misleading representation of the Blackscholes MPKI. The increased MPKI did not reflect the throughput of workload as it was still similar to the medium Blackscholes MPKI. Its effect was minimal because the MPKI magnitude is 0.25 misses per 1000 instruction. This is why we do not see a performance hit like in the case of streamcluster's medium workload.

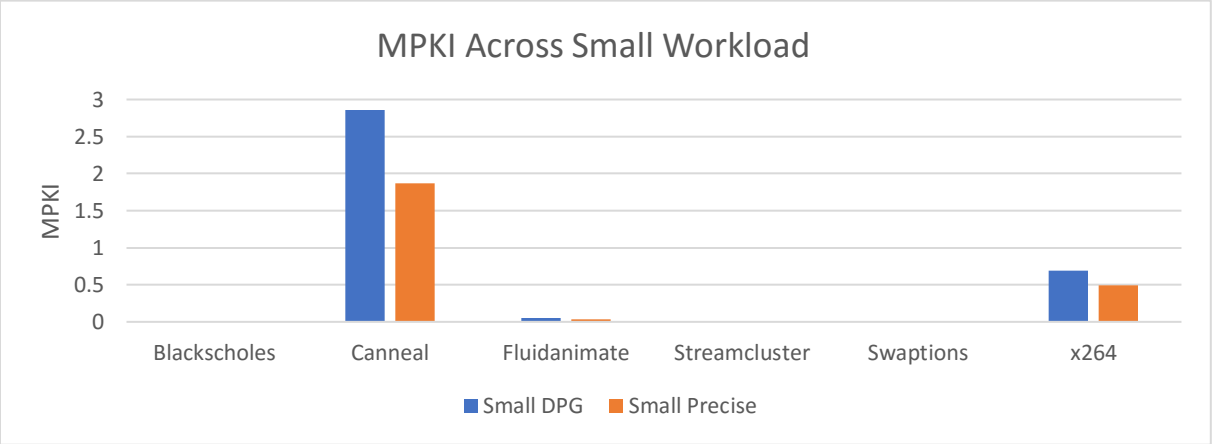

Figure 7: Raw MPKI Values for Small Workloads

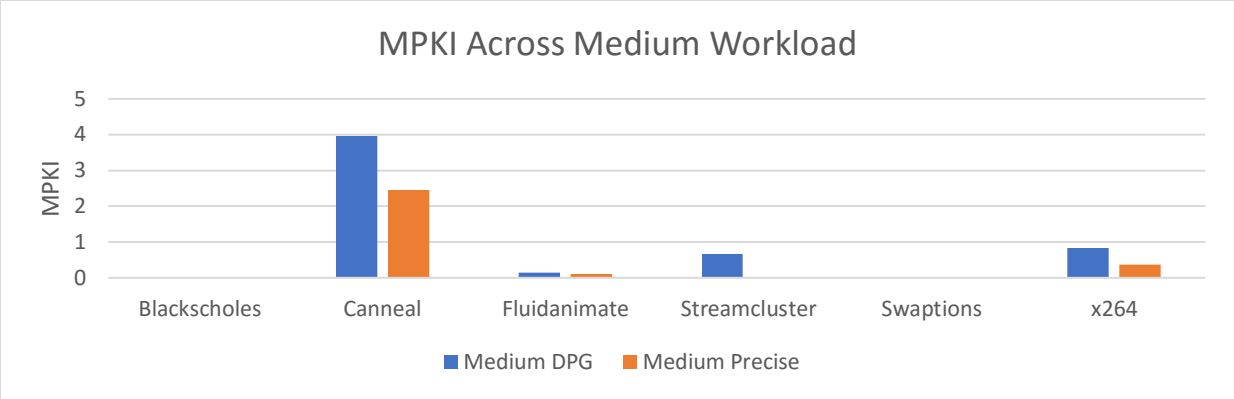

Figure 8: Raw MPKI Values for Medium Workloads

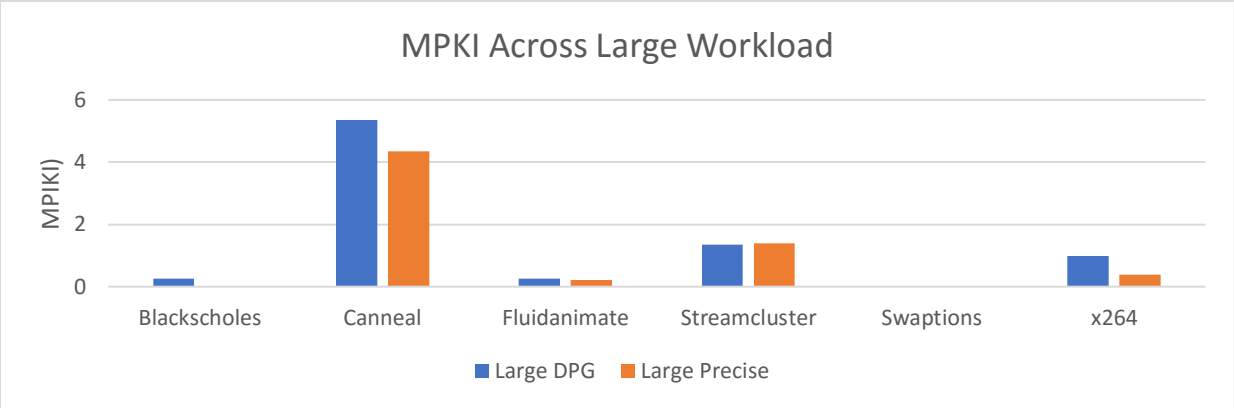

Figure 9: Raw MPKI Values for Large Workloads

AREA

The area calculation, while it provides us with an estimation of area improvement, makes 2 large ideal assumptions. The first assumption is that there no extra hardware other than the bits are needed for the implementation of the Doppelgänger cache. This is an incorrect assumption and would have drastic effect on the size of the proposed cache's implementation. However, since we also disregard any hardware overhead on the fully precise cache, we can assume that it we are still obtaining a fair comparison between the two caches. In fact, the fully precise cache will have significantly higher cost for cache overhead due to the routing for significantly larger data blocks. The second assumption is every bit is square shaped and is not true when dealing with VLSI. Several estimations exist for

VLSI space savings which our calculation does not account for. Once again, since this was applied to both the Doppelgänger and Precise cache, the calculated area improvement of 1.51 can be considered a fair comparison.

Comparison to Original Work

The original work reported improvements of 1.7x for LLC area savings with a performance hit of 2.3% increase in runtime (we obtained an increase in runtime of 5%). These differences can be attributed to the fundamental to configurations and area assumptions that we made. Due to the nature of our implementation, we utilized a significantly larger cache. Specifically, we were facing problems with the ZSim scheduler when we defined our system configuration based on the paper. However, utilizing Jiayi's configuration, we obtained proper running functionality. As more cache blocks are present within the cache, the time for tag matching also increase drastically. The smaller LLC improvement was also a direct result of this the larger cache. We needed more bits than were needed in the original Doppelgänger paper configuration to maintain correct functionality of our cache. Approximate percentage calculations were also causing heavy overhead and were removed due to computation resource scarcity. Another reason for the variations in performance is due to a lack of in-depth knowledge of the benchmarks. This resulted in often incorrect approximate range estimation and may have affected performance slightly.

Future Work

The paper proposes uni-Doppelganger cache for better performance across all benchmarks. This cache architecture removes the restrictions on the usage of cache lines. Hence, if a benchmark like x264 doesn't have any approximate regions, its performance will not suffer. Hence, the 4.5% hit can be further reduced using Doppelgänger cache. Furthermore, we wanted to analyze the power consumption and access times using Cacti tool but we couldn't accomplish this within the timeframe of the project.

Conclusion

The project was successful in implementing the Doppelgänger cache within the confines of ZSim. Our design showed a performance reduction of around 4.2% but provided a LLC area reduction of 1.51x. This area improvement

corresponds to the energy savings. With the advancements of several workloads which are conducive to approximate computing, the reduction in power consumption improves efficiency. Though it is a promising proposal paradigm with which to tackle Big Data challenge, its increase runtime may not be acceptable for consumers and industry,

SOURCES

1. Shirriff, Ken. "The surprising story of the first microprocessors." *IEEE Spectrum* 53, no. 9 (2016): 48-54. doi:10.1109/mspec.2016.7551353.
2. Nair, Ravi. "Big data needs approximate computing." *Communications of the ACM*58, no. 1 (2014): 104. doi:10.1145/2688072.
3. Kleanthous, Marios, and Yiannakis Sazeides. "CATCH: A Mechanism for Dynamically Detecting Cache-Content-Duplication and its Application to Instruction Caches." *2008 Design, Automation and Test in Europe*, 2008. doi:10.1109/date.2008.4484874.
4. Sampson, Adrian, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. "EnerJ." *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation - PLDI 11*, 2011. doi:10.1145/1993498.1993518.
5. Miguel, Joshua San, Mario Badr, and Natalie Enright Jerger. "Load Value Approximation." *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, 2014. doi:10.1109/micro.2014.22.
6. Miguel, Joshua San, Jorge Albericio, Andreas Moshovos, and Natalie Enright Jerger. "Doppelgänger." *Proceedings of the 48th International Symposium on Microarchitecture - MICRO-48*, 2015. doi:10.1145/2830772.2830790.
7. Bienia, Christian, “Benchmarking Modern Multiprocessors.” *Princeton University*,2011.
8. Sanchez and Kozyrakis "ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-Core Systems", ISCA-40, June 2013

APPENDIX

Component	Doppelgänger	Precise
Number of Tags	128K	128K
Linked List Tag Size	86 bits	
Block Size	528 bits	512 Bits
Tag Size		24 bits
Map	16 bits	
Total	4.42×10^7 bits	7.26×10^7 bits

TABLE 1: BITS FOR EACH DOPPELGANGER CACHE COMPONENT

Component	Fully Precise
Number of Tags	256 K
Tag	24 bits
Block Size	512 Bits
Total	1.41×10^8 bits

TABLE 2: BITS FOR FULLY PRECISE CACHE