A Comprehensive Review of the Challenges and Opportunities Confronting Cache Memory System Performance

Richard A. Kramer, Mathias Elmlinger, Abhishek Ramamurthy, Siva Pranav Kumar Timmireddy
kramerri@oregonstate.edu, elmlingm@oregonstate.edu, abhisher@oregonstate.edu ,timmires@oregonstate.edu
Oregon State University

***Abstract*—In computer systems, the cache memory architecture has a significant impact on both, system performance and system cost. Further, the gap between processor performance and cache memory performance is widening at the disadvantage of the overall system performance. In this paper, we explore the important aspects that impact cache memory architecture performance and cost, including: (1) An overview of present state-of-the-art cache memory architectures. (2) We examine the latest advances in cache controllers and energy management. (3) We explore important aspects of cache memory organization, including cache mapping, spatial cache and temporal cache techniques. (4) We provide an analysis of performance of state-of-the-art cache memory architecture implementations including new promising memory technologies. (5) We end by considering future research areas that may prove promising in narrowing the performance gap between cache memory performance and processor performance. Overall, improvements in cache memory architectures stand to make a significant impact in unlocking major improvements in high performance computer architectures.**

Keywords—cache memory architecture; cache data mapping; prefetching; low-power cache; cache coherency

I. INTRODUCTION

Modern high-performance computer architectures, such as the one shown in Figure 1, would not exist without cache memories. Nonetheless, since the first implementations of cache memories, the imbalance between the processor system performance and the cache memory system performance has had a detrimental impact on the overall system performance [2]. Amazingly, the gap between processor system performance and cache system performance was recognized as early as the 1970s [3].

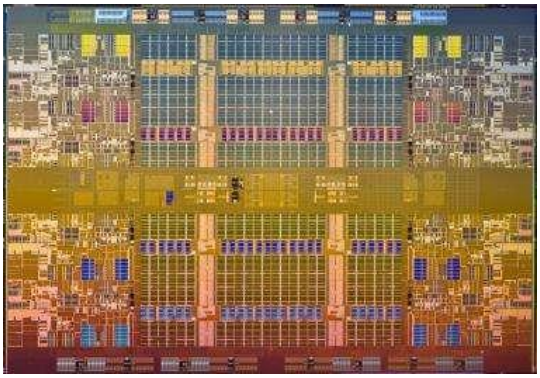

Figure 1: Photograph of Intel Xeon processor 7500 series die showing cache memories (center) [1]

Unfortunately, sub-optimal cache system performance still remains as one of the largest limiting factors to optimal system performance right up to present times. To put this into prospective, some facts that have been recognized for over 30 years include [3]:

1) It has been estimated that as processor gate counts continue to inevitably increase. To be precise: for every 10-fold increase in transistor gate count, the required memory bandwidth demand increases by 30-fold.
2) The small cache memories within a processor make up a larger cost impact, by percentage, than the larger external memories.
3) From the onset of cache memories in the 1970s, it has been estimated that the required bandwidth to supply the core processors with instructions and data exceeds the ability of the cache memory to supply the needed bandwidth by a factor of 300%.

Further, it is estimated that 50% of power consumption in advanced computer architectures is a direct result of how efficient (or inefficient) the cache memory system performs [4][5][6]. Thus, since the introduction of cache memory architectures, researchers have and continue to struggle with the very same topics of cache coherency [7][8][9], write-through versus write-back [10] and optimal cache size [3][11][12]. To address the abovementioned limitations, consistent topics that researchers have heavily researched, and continue to research, include the following areas [13]:

1) Cache memory access prediction improvements related to spatial memory access (e.g., locality of data accesses by address) and temporal memory access (e.g., locality of data accesses in time).
2) Optimization of cache memory associativity to main memory. In other words, finding the optimal methods to map cache memory to main memory.
3) The development of intelligent software compilers to attempt to improve cache accesses based on prediction (e.g., determining via software compilers, how likely certain memory addresses will be accessed).
4) Improvements in the mapping of L1 cache memory contents to that of L2 cache memory contents.
5) Advancement in the performance of mapping cache memory to main memory via the TLB (Translation Lookaside Buffer).
6) Hardware prefetching enhancements to better supply optimal memory prefetcher performance.

II. OBJECTIVES AND CONTRIBUTIONS

The objective of this paper is to take the reader to the forefront of the battle to improve the imbalance between processor system performance and cache system

performance. Specifically, we focus in a number of core areas that are further discussed below.

In Section 3 – "Advances in Cache Data Management: Prefetching, Bandwidth Management, Scheduling, and Data Placement", we point to the most recent research related to improving how cache memory is used. We include a review of novel advancements in cache prefetching, improvements in cache memory bandwidth utilization, and optimizations of data placement within the cache memory system [14][15][16][17]. Given the fact that cache memory to processor system bandwidth is a major bottleneck, we point to new research to utilize valuable bandwidth resources in the absolute most efficient manner.

1) As an example, we review promising techniques to efficiently[1] learn and intelligently associate an array of different types of prefetchers to the software that is being executed (e.g., selecting the best prefetcher based on the application(s) being run). Based on this technique, the solution offers a worst case 1.4% to 18.7% improvement over the best present day techniques, while at the same time, using less memory and logic overhead [14].
2) As yet another example, based on intelligent thread and data placement schemes, we point to research that provides a 46% increase in cache memory system performance as compared to present day NUCA (Non-Uniform Cache Architectures) [15].

In Section 4 – "Leading-Edge Hardware Implementations and Opportunities", we point to modern day challenges and potential breakthroughs related to the considerable impact that cache memories have on system power requirements, access speed, fault tolerance and reliability [4][5][6][18].

1) We are intrigued and examine advances that have allowed low power battery operated devices to employ cache based systems. Such advances offer significantly low power consumption, yet provide superior cache performance [4][5].
2) We further evaluate and provide insight into new opportunities to speed up cache memory accesses by as much as 11.3% (and an encouraging 8.6% speed up on average) when combined with present day NUAT (Non-Uniform Access Time) memory [6].

In Section 5 – "Special Topics in Cache Memory Architectures", we discuss advances in the overall processor and cache memory core architecture [19][20][21][22][23].

1) We examine the concept of "cloning" - a technique to simulate actual workloads of proprietary programs to find optimal cache memory architectures that can then be applied to actual real-world applications. By doing so, the processor / cache memory core architecture can more easily be evaluated and then optimized [19].
2) We point to promising new architectures. For example, we look to new breakthroughs in processor system stall avoidance, providing a 6% improvement on a 4-core processor system [23].
3) We look to advances that proactively and predicatively identify cache contents that will not be used in the future (e.g., dead blocks) so that the unused cache content can be replaced by relevant content, thus reducing wasted cache energy by 20% [21].

In Section 6 – "Conclusion", we summarize our findings and provide a case study of taking cache memory architecture research from "concept to reality" via the Intel Xeon Haswell processor [24]. We also consider new frontiers for future work including optical cache memory architectures [25].

III. Advances in Cache Data Management: Prefetching, Bandwidth Management, Scheduling and Data Placement

Advances in cache data management techniques offer a wide range of exciting opportunities to improve overall cache memory system performance. In this section, we discuss advances related to cache data management including:

- Advanced prefetching that employs a unique way to monitor and then select the optimal prefetcher.
- Bandwidth management techniques based on the prediction of bandwidth requirements for multiple threads of software running on multiple processor cores.
- Cache data scheduling, that creates virtual cache memories that transcend across multiple threaded applications and even multiple processors.
- Unique cache data placement management techniques entailing algorithms and architectures used to determine where to store data in relation to SRAM and STT-RAM (Spin-Transfer Torque RAM).

A. *Advanced Prefetching*

Cache Prefetching is a Technique used in modern day computer processors to improve the execution speed by prefetching instructions/data from main memory and supplying the instructions to cache memory. Modern day computer processors use high speed cache memory, whereas fetching of instructions and data for processing is much faster from cache memory as compared to accessing the same from main memory. There are multiple techniques to implement cache prefetching, and the techniques are broadly classified under: (1) hardware based and (2) software based implementations. In hardware based prefetching, there is dedicated hardware that monitors the stream of instructions/data being requested by the program under execution. The hardware prefetches the next set of data/instructions that the program being executed might request. Figure 2 is an example of a hardware based prefetching technique (Stream Buffer) as proposed by Norman Jouppi [26].

In contrast to hardware prefetching, for software based prefetching, the prefetching mechanism is applied during the compilation time of the program. Compiler based prefetching techniques are more widely adopted in the case of loops that contain a large number of iterations. At compilation time, the compiler predicts the future cache misses and inserts a prefetch instruction based on the miss penalty and execution time of the instruction. Through compiler based prefetching techniques, run time true data

[1] Efficiency in both memory space and hardware/logic/computational complexity implementation.

dependency issues cannot be resolved during compilation time. In this section, we will discuss the recent trends in cache prefetching techniques which involves hardware, software and a combination of both mechanisms involved.

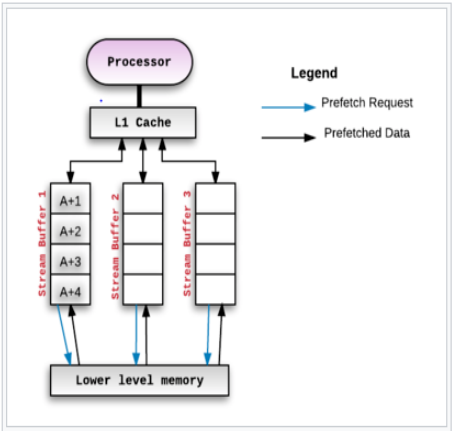

Figure 2: Stream buffer proposed by Jouppi [26] [27]

B. *The Sandbox Prefetching Technique*

The sandbox prefetching technique is based on the use of a Bloom filter. The Bloom filter was proposed by Burton Howard Bloom in 1970. The Bloom filter is a probabilistic model to test whether a data element is a member of a set. A query to a Bloom filter returns "possibly in set" if the element is present or "definitely not in set" if the element is not present in the set.

In the paper "Sandbox Prefetching: Safe Run-Time Evaluation of Aggressive Prefetchers", Pugsley et al. [14] presents a hardware base technique which provides features of aggressive prefetching, yet avoids bandwidth and cache capacity wastage due to aggressive prefetching. The key feature of the sandbox prefetching technique is the reduced latency overhead in prefetching by using a Bloom Filter among other methods. The sandbox prefetching technique uses the concept of global pattern confirmation and immediate prefetch action, thereby enabling better execution performance [14].

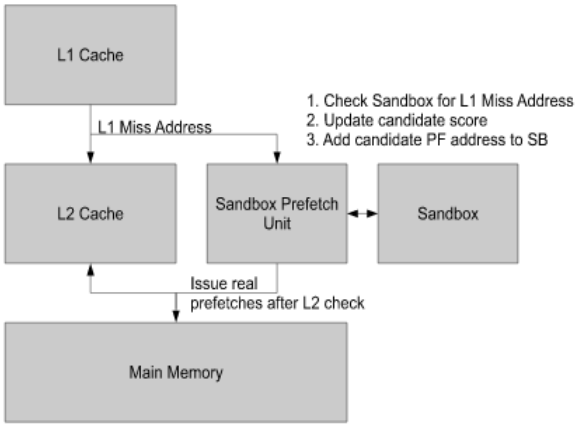

Figure 3: Figure showing sandbox prefetcher architecture [14]

Figure 3 shows the placement of the sandbox unit within the memory hierarchy. As shown in Figure 3, the sandbox unit doesn't impact normal cache actions. The sandbox prefetch mechanism proposed by Pugsley et al. [14] has a separate sandbox prefetch unit and a sandbox unit. The sandbox technique begins by monitoring multiple prefetcher algorithms, seeking to find the most effective prefetcher algorithm. The sandbox unit keeps the score (hits versus misses) of candidate prefetchers, based on the outcome of individual cache lines being a hit or a miss. Each time there is a cache access, the corresponding prefetcher candidate score is incremented based on a hit. Once the score of a candidate prefetcher crosses a threshold, the prefetch mechanism control is taken over by sandbox prefetch unit. Figure 4 shows the sandbox prefetching actions for each L2 access [14]. Sandbox prefetching maintains a set of 16 candidate prefetchers and each candidate is evaluated in a round-robin fashion [14].

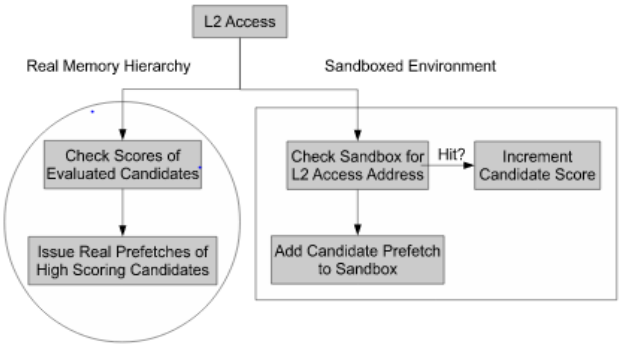

Figure 4: Sandbox prefetching action on each L2 access [14]

Figure 5 (see next page) shows the performance of SandBox Prefetching (SBP), normalized to a no-prefetch baseline. The sandbox technique is compared with No Prefetching (No PF), Feedback Directed Prefetching (FDP) and Address Map Pattern Matching (AMPM). Sandbox prefetching provides better performance when compared to the other prefetching mechanisms [14].

C. *Bandwidth Shifting*

Current modern day microprocessors have multiple cores and run multiple threads concurrently. Novel techniques have been proposed, with the idea of dynamically assigning needed bandwidth to applications based on the prefetch efficiency of each thread.

Increased in Multicore System Efficiency Through Intelligent Bandwidth Shifting

Jimnez et al. [16] introduces a technique that increases multicore system efficiency through intelligent bandwidth shifting. Data prefetching hides memory access latency, but not all of the prefetched data is accurately fetched, thus reducing the performance of the system. The technique employed by Jimnez et al. provides an efficient software mechanism for dynamically assigning memory bandwidth for each thread, based on the predicted prefetch efficiency.

The technique assures backward compatibility [16]. The technique further provides the following characteristics:

- Prefetch based bandwidth shifting to characterize performance.
- Metrics to estimate prefetch usefulness.
- Novel bandwidth shifting mechanisms to increase performance.
- Evaluation of bandwidth shifting.

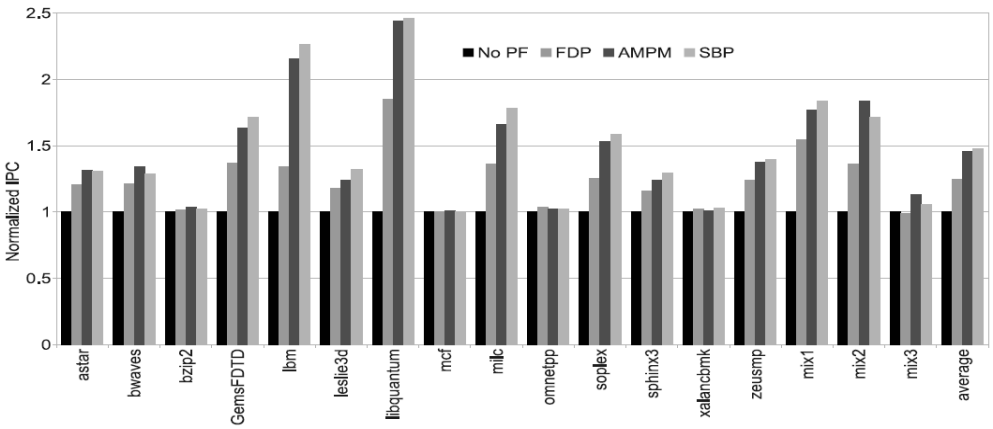

Figure 5: Performance normalized to no-prefetch baseline [14]

To expand, Figure 7 shows the throughput and bandwidth consumption of a subset of benchmarks defined in the SPEC CPU2006 benchmark specification. Figure 7, indicates Deep, Shallow and OFF regions. In the Deep region, the prefetcher uses the longest distance available for prefetching. The Shallow region uses the shortest distance for prefetching. Lastly, the OFF region refers to the prefetching action being turned off. Figure 7 clearly indicates that when more than 16 threads are being used, the bandwidth usage and performance saturates. All of the performance benchmarks are evaluated on an IBM POWER7 machine. Jimnez et al. [16] states that the benchmark results are not exclusive to the IBM POWER7 machine used by Jimnez et al. The efficiency of prefetching applications varies, depending on the memory access pattern and the availability of bandwidth. Jimnez et al. [16] also states that there were no severe impacts observed when changing to aggressive prefetch actions. The proposed technique of bandwidth shifting uses only DEEP and OFF settings for the prefetching mechanism [16].

The bandwidth shifting algorithm proposed by Jimnez et al. [16] uses an iterative approach. Initially the configuration is set to the most aggressive prefetch setting. Next, the algorithm computes the usefulness of prefetching an instruction for each thread and tabulates the result. The evaluation of prefetch usefulness is done by frequently turning on and off the prefetching for each thread and then measuring the Instruction Per Cycle (IPC) and bandwidth usage under both the on and off configurations. Figure 6 shows the base implementation of the algorithm [16].

The base algorithm shown in Figure 6 introduces a problem: there is a lack of hardware resources while high Prefetch Usefulness (PU) threads are running on the system due to the limited number of prefetch streams that can be allocated. To overcome this problem, as shown in Figure 8, Jimenez et al. [16] introduced a modified base algorithm which increases performance by 33% when compared to the performance of the algorithm shown in Figure 6 [16].

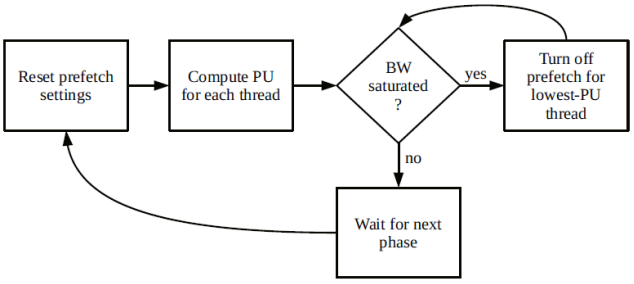

Figure 6: Base bandwidth Shifting algorithm [16]

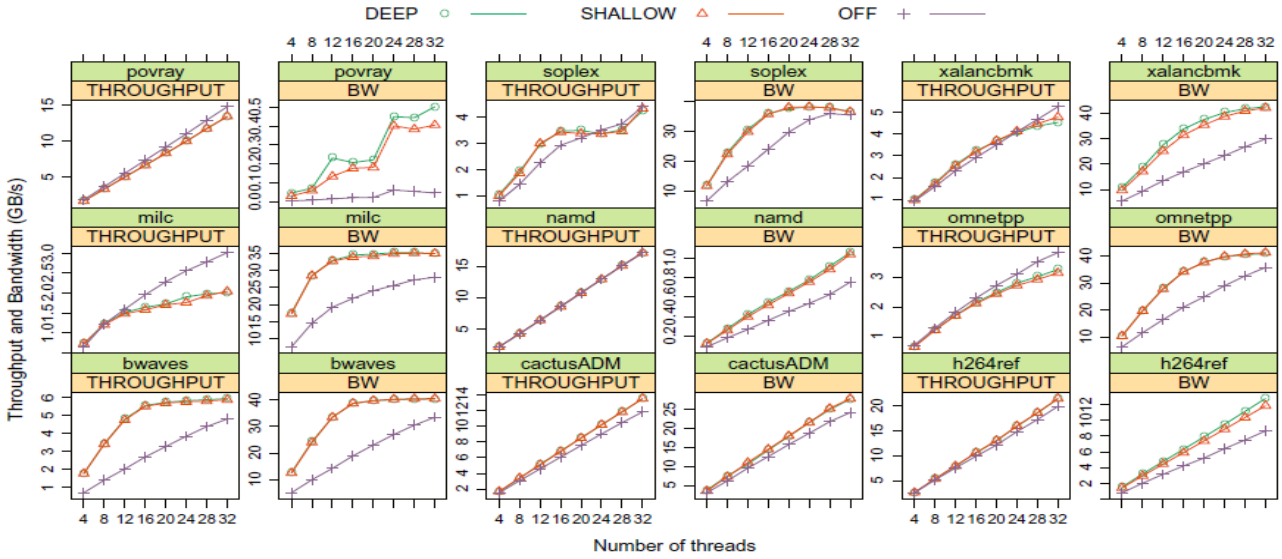

Figure 7: Throughout and memory bandwidth consumption characteristics for a subset of benchmarks [16]

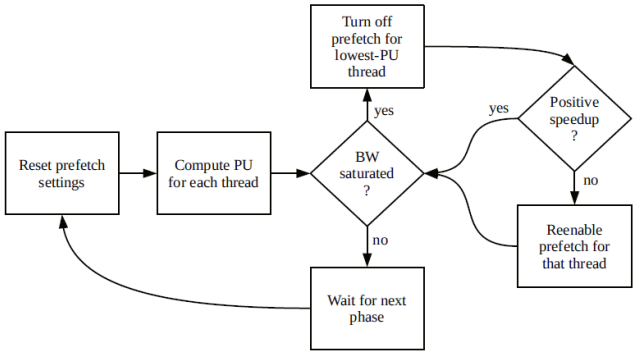

Figure 8: Modified base algorithm [16]

In the modified algorithm shown in Figure 8, the initial mechanism is the same as that of the base algorithm as shown in Figure 6. A number of additional steps are also added as follows:

- Step 1: Measuring system performance by turning "off" the prefetching for a thread.
- Step 2: Testing if there was a positive impact on the system when the prefetch mechanism is turned "off" for a given thread.
- Step 3: If there was improvement by turning "off" prefetching for a given thread, a decision to turn "on" or "off" the prefetch action for a given thread will be considered again in the next iteration.

Figure 9 illustrates the positive effect of the bandwidth shifting algorithm on system performance. Figure 9 plots a function of the prefetch friendly algorithm "bwaves" (which we assign the value "z" to the number of simultaneous thread instances running) and the prefetch unfriendly algorithm "omnetpp" (which we assign the value "x" to the number of simultaneous thread instances running) as benchmarks. Specifically, Figure 9 shows the amount of speedup for 32 processes running simultaneously, with the x-axis representing the number of unfriendly algorithm "omnetpp" simultaneous thread instances running ("x") as a function of the number of friendly algorithm "bwaves" simultaneous thread instances running ("z"). Thus, x + y = 32 [16].

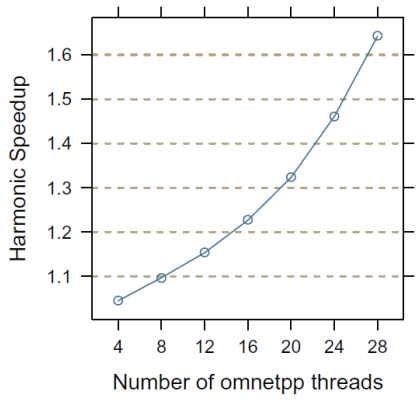

Figure 9: Effect on bandwidth shifting on system performance with prefetch efficient (bwaves) and inefficient (omnetpp) threads [16]

D. *Scaling Cache Hierarchies Through Computation and Data Co-Scheduling*

Today, Non-Uniform Cache Architecture (NUCA) is the most widely used method to extract improved performance from cache memory systems. Advanced techniques of NUCA include: (1) Reactive Non-Uniform Cache Architecture (R-NUCA) and (2) Static Non-Uniform Cache Architecture (S-NUCA). Recently, better techniques that further improve R-NUCA and S-NUCA have been proposed. Such improvements provide better cache memory management and improved thread scheduling to derive better system performance. One such technique is referred to Computation and Data Co-Scheduling (CDCS) [15].

Computation and Data Co-Scheduling technique (CDCS)

One example of cache memory scheduling is disclosed in the paper "Scaling Distributed Cache Hierarchies through Computation and Data Co-Scheduling" by Beckman et al. [15]. Beckman et al. proposed a technique called Computation and Data Co-Scheduling (CDCS), a technique that relates to the placement of threads and data using distributed shared caches in a multiprocessor environment. The main contributions the Beckman et al. [15] paper are as follows:

- A novel thread and data placement scheme that considers both data and access intensity by threads across multiprocessor tiles.
- An enhanced design of a geometric sampling curve monitors that scales within a very large NUCA.
- Hardware that enables incremental reconfiguration of NUCA caches.

The CDCS technique then tags data to the virtual cache using virtual cache "ids" (IDs). For every L2 level cache miss using the VC (Virtual Cache) "id", CDCS determines where the cache line resides in the memory subsystem. A Virtual Translation Buffer, referred to as a "VTB", as shown in Figure 10, stores the configuration for all virtual cache memory groups that a given executing thread can access. Virtual cache configurations are periodically changed by CDCS software (every 25ms); changing both the bank and partition sizes on the fly during runtime, based on how data is accessed by the executing threads. A block diagram of how the virtual cache is reconfigured is shown in Figure 10.

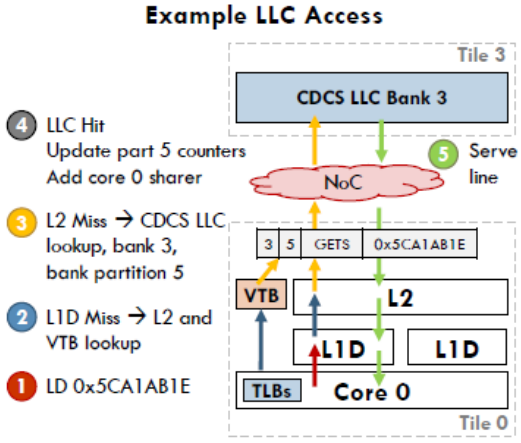

Figure 10: An example of LLC access using CDCS [15]

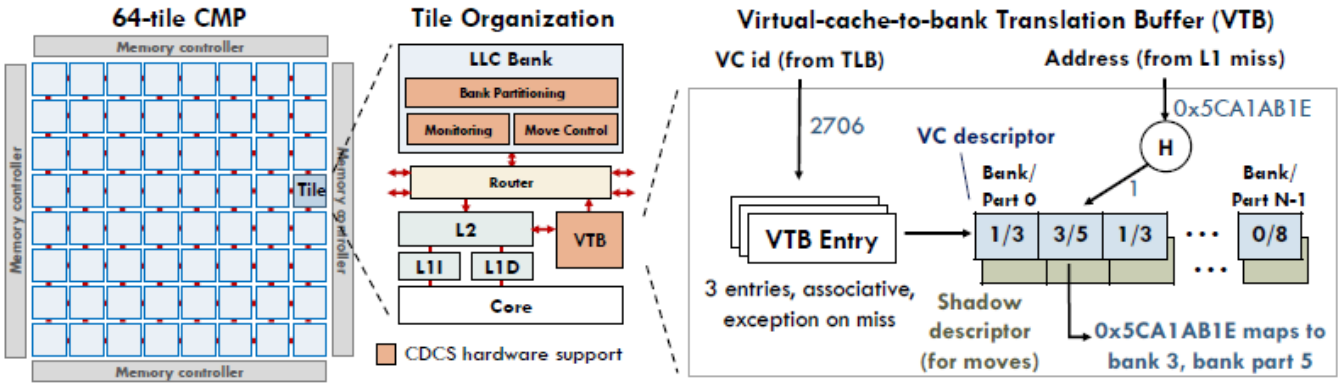

Figure 11: CDCS implementation with 64 tile CMP [15]

Figure 11 shows the hardware black box hardware implementation of CDCS. Each tile has a core and a slice of Last Level Cache (LLC). An on-chip network topology establishes connection between a tile and the memory controllers that reside at the edges.

CDCS is based on NUCA methodology and allows software to divide each cache bank into multiple partitions. Collections of portioned caches are grouped and are made visible to software threads as a single cache. The grouping of the caches provides the software with flexibility to define multiple virtual caches and to configure them into different sizes of virtual cache memory [15].

Figure 12 shows the thread and data placement under R-NUCA techniques, where thread private data is stored for threads in the processor's local memory bank. Figure 13 shows how the thread and data is placed using the CDCS technique provides a 400% higher speed-up over the R-NUCA technique [15].

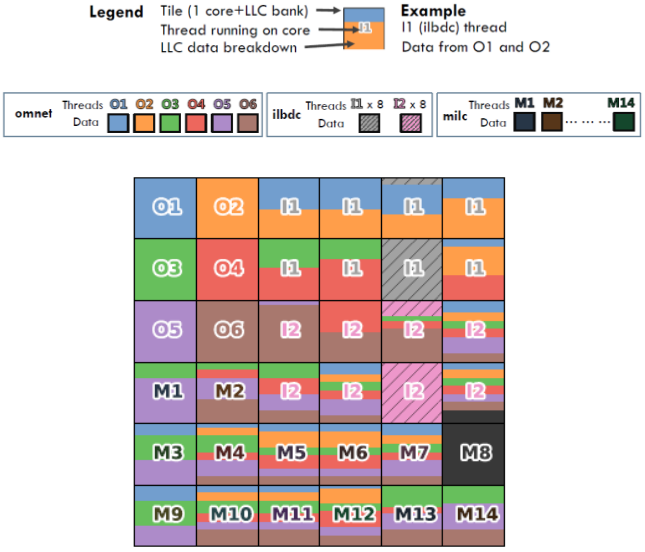

Figure 12: R-NUCA workload *organization schemes on 36 tile CMP [15]*

CDCS software provides different levels of virtual caches. During execution, each thread is provided with a thread private cache at the OS-level. Common data between the threads of the same process are placed in a process private cache, and common data between the processes are placed in a global virtual cache. Based on these techniques, faster access to data is provided and cache pollution is reduced. The CDCS technique provides a 46% increase in performance when compared other NUCA techniques, and provides 36% better energy efficiency when compared to S-NUCA [15].

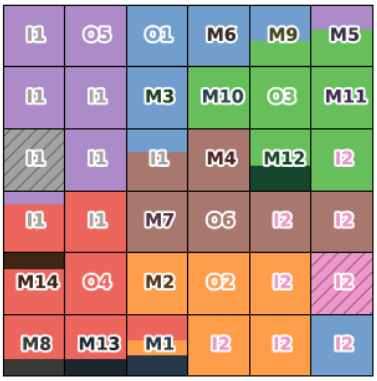

Figure 13: CDCS workload organization schemes on 36 tile CMP [15]

E. *Adaptive Placement Policies for Data in Cache Memory Systems*

Another leading area of research is the intelligent placement of cache memory contents in differing types of memory within cache memory systems and main memory. For example, a hybrid of cache memory system consisting of DRAM, SRAM and even STT-RAM.

An Adaptive Placement and Migration Policy for an STT-RAM Based Hybrid Cache System

One such paper that considers new data placement polices for data blocks in cache memory systems is the paper "Adaptive Placement and Migration Policy for an STT-RAM-Based Hybrid Cache" by Wang et al. [17]. \Wang et al. [17] proposes an Adaptive block Placement and Migration policy (APM) for hybrid caches. The technique proposed by Wang et al. places the block in either STT-RAM (Spin-Transfer Torque – RAM) or SRAM, based on an adaptive placement and migration policy algorithm. The technique proposed by Wang et al. combines the advantages of low leakage power and high packing density offered by STT-RAM with the low write overhead of SRAM [17].

To expand, Wang et al. categorizes LLC cache accesses into three distinct classes: (1) core-write, (2) prefetch-write and (3) demand-write. Turning to (1) - core-write, a core-write is a write from the core to the LLC. For a write through core cache, a core-write entails directly writing from the core

through to the LLC. For a write-back core cache, a core-write entails evicting dirty data from the core cache and a write back to the LLC. For (2) - prefetch-write, a prefetch-write is a write replacement of the block from LLC caused by a prefetch miss. For (3) - demand-write, a demand-write is a write block replacement from LLC caused by a demand miss. The technique proposed by Wang et al. [17] is based on block replacement if the request is initiated by a write access. Wang et al. [17] introduces an intelligent block placement policy as follows:

- SRAM should be used for the majority of the write actions, thus avoiding write overhead involved in STT-RAM.
- Frequently used blocks should be placed in LLC to achieve reduced memory access latency, reduced overhead, and less complexity within the overall design.
- Block placement is often initiated by a write access to the LLC which Wang et al. further subcategorizes to be either a (1) prefetch-write, (2) core-write or (3) demand write as discussed above [17].

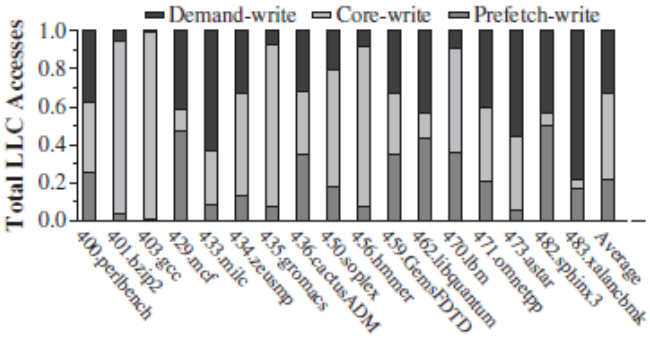

Figure 14: Distribution of LLC write accesses. Each type of write access accounts for a significant fraction of total write accesses [17]

Figure 14 shows the breakdown of block placement for (1) core-write, (2) prefetch-write and (3) demand-write to the LLC. Wang et al. [17] further teaches two types of ranges: (1) read-range and (2) depth-range, which is further described as follows:

- Read-Range: The read-range is a property of a cache block that fills the LLC by a demand-write or prefetch-write request. It is the largest interval between consecutive reads of the block from the time it is placed into the LLC until the time it is evicted [17].
- Depth-range: The depth-range is a property of a core-write access. It is the largest interval between accesses to the block from the current core-write access until the next core-write access to the same block. The "depth" refers to how deep the block descends into the LRU stack before it is accessed again [17].

In Figure 15, "Ra" represents the Read block "a" and "Wa" represents the Write block "a". The distance between successive block reads is referred to as "read-range" as discussed above. The distance between a write access to that of reading the same data is referred to as "depth-range" as discussed above. "Wa" equals 0 and represents an evicted block from cache, e.g., the least used data from cache is kicked out from the cache memory. Read-range/depth-range is further classified as follows [17]:

- Zero-read/depth-range: Data is filled into the LLC by a prefetch or demand request/core-write request, and it is never read/written to again before it is evicted.
- Immediate-read/depth-range: The read/depth-range "I" (which is further set to be smaller than a parameter "m", where m = 2 is the number of SRAM ways in the STT-RAM/SRAM hybrid cache configuration).
- Distant-read/depth-range: The read/depth-range is larger than m = 2 and at most, the associatively of the cache set which is 16 in STT-RAM/SRAM configuration.

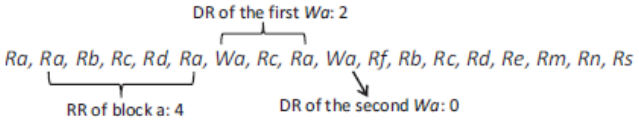

Figure 15: Example illustrating read-range and depth-range [17]

The technique proposed by Wang et al. [17] uses the read-range to analyze the access patterns of LLC. Figure 16 shows each access pattern and each category is further classified based on read-range/depth-range. A summary of the results are as follows:

- Zero-read/depth-range corresponds to 26% of all prefetches on average. For prefetch-writes, because the category is never used until a miss occurs and then a block is evicted from cache, the prefetched block should be placed in SRAM as to avoid the write overhead of STT-RAM.
- Immediate-read-range corresponds to 56.9% on average. The data associated with this category should likewise be placed in SRAM to provide fast access for immediate use. Using SRAM for this category mitigates STT-RAM involvement in eviction once the cache block is dead.
- Distant-read corresponds to 17.5% on average. For this category, data should be placed in STT-RAM to make use of large capacity to avoid cache misses.

In the proposed design by Wang et al. for core-write access misses, the data is directly written back to the main memory. Zero-read-range blocks should be bypassed from cache because the data will not be used except for eviction from cache of a dead block. Thus, bypassing zero-read-range blocks will reduce the write operations to LLC.

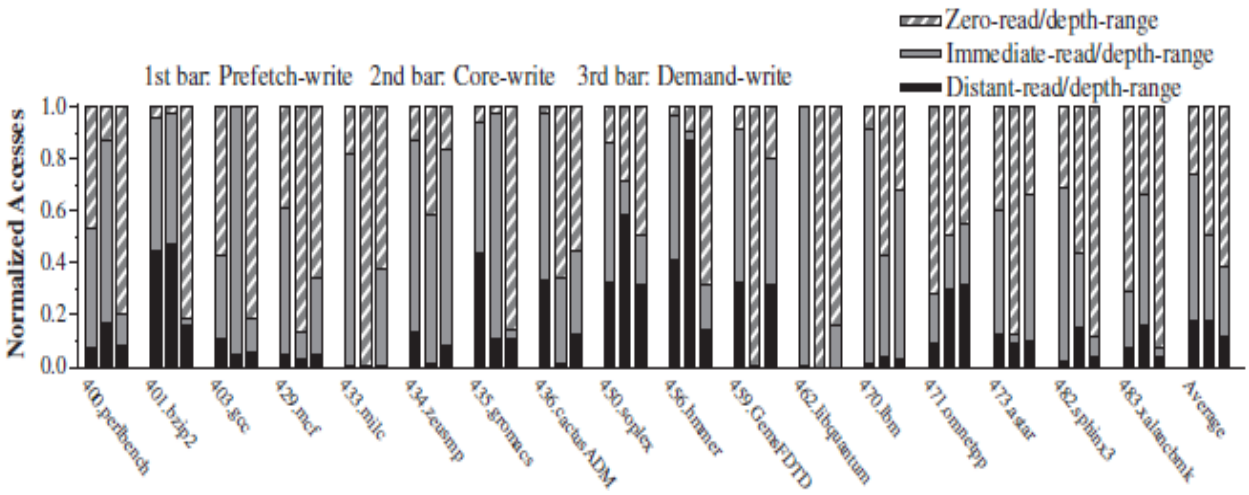

Figure 16: The distribution of access pattern of each type of LLC write access [17]

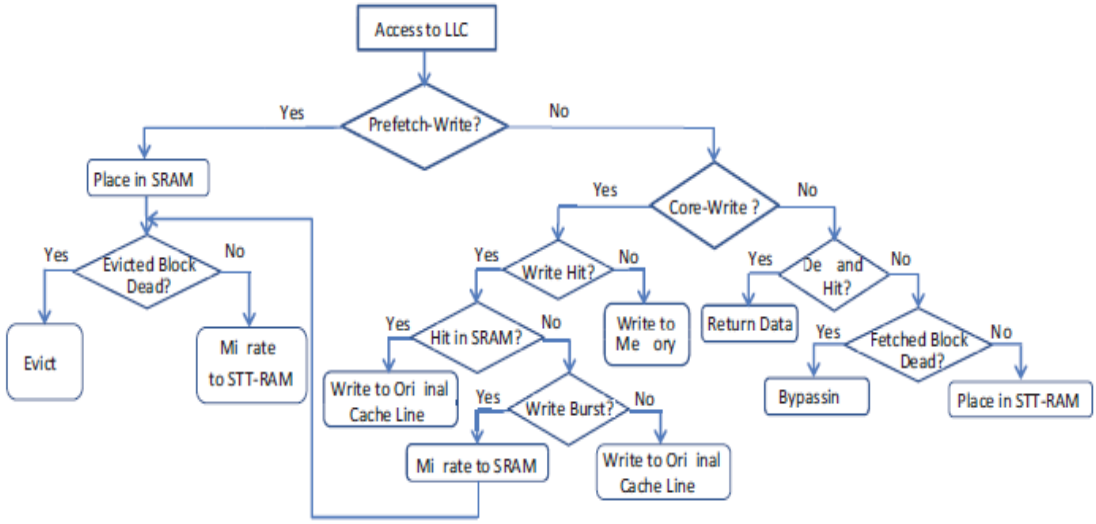

Figure 17: Flow chart of the adaptive block placement and migration mechanism (errors as shown in the original) [17]

Figure 17 shows the flow chart of the proposed design. Each block is associated with a prediction bit indicating whether the block is dead. On a cache miss the prefetched data is placed into the SRAM; and the prediction bit that predicts if the block is dead is set to 1 (e.g., it is assumed dead on arrival). An access bit pattern predictor, predicts whether the block in SRAM is dead. The proposed scheme reduces the overhead of STT-RAM by using the following schemes [17]:

- By bypassing dead on arrival blocks.
- By introducing an SRAM line filter to filter write operations caused by inaccurate and immediate-read-range prefetch requests.
- By placing frequently used core-write blocks in SRAM.

The access pattern predictor makes a prediction in the following three conditions: (1) when a core-write request is a hit within the STT-RAM lines, the write burst prediction table will be accessed to predict whether it is a write burst request; (2) for each read hit request within the SRAM lines, the dead block prediction table will be accessed to predict whether it is a dead block; (3) on a demand-write request, the dead block prediction table will be accessed to predict whether the request is a dead-on-arrival block request [17].

Overall, the block placement technique proposed achieves higher performance by placing distant-read-range blocks in STT-RAM and by bypassing the zero-read-range cache lines in order to avoid write overhead; SRAM provides better efficiency in evicting inaccurately fetched data blocks.

IV. Leading-Edge Hardware Implementations and Opportunities

Given the steadily growing market for battery-powered devices (e.g., mobile phones or wireless embedded sensor networked devices), energy efficiency has become a crucial factor in the development process. Advances in technology have and will further lead to even smaller device sizes, driven by voltages as small as possible. Given these advances, the system's overall energy dissipation will be influenced by up to 50% by the cache. New techniques have been proposed that optimize already existing architectures to minimize the overall power consumption in order to provide longer battery life, mitigate the design limiting effects of temperature, and provide better performance [4][5][18].

On-chip cache memories make up a large fraction of the overall chips size and therefore play a significant role in the overall power consumption of the system. Recent research has shown that the following factors influence the energy consumption by a significant amount [4][5][18]: (1) static leakage current, especially in multi-port architectures, (2) the use of Error Detection Codes (EDC) and (3) the use of Error Correction Codes (ECC). Additionally, a new concept that utilizes the charge leakage of a cell to improve access latency and ultimately also improves the energy efficiency is introduced. The following sub-sections provide a brief introduction into each of these areas.

A. *Leakage Current*

Two types of leakage currents mainly contribute to the overall cache leakage current: (1) cell leakage current and (2) bit line leakage current. Further, there are a number of factors that increase leakage current, including the use of multi-port caches and the fact that leakage current scales proportionally with the area of the circuit [4]. In the following, we explore two different promising approaches to reduce the cache memory power dissipation, namely, Dynamic Memory Configuration and Software Self-Invalidation and Data Compression.

1) Dynamic Memory Configuration

Figure 18 and Figure 19 show a six transistor single-port and dual-port SRAM cell, respectively. The additional word lines needed to access transistors T7 and T8 almost double the silicon area of the single-port configuration. Keeping the bit lines high, as well as pre-charging, contributes significantly to the overall power dissipation [4].

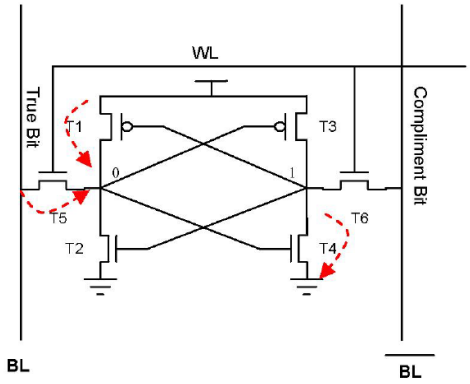

Figure 18: Single-port SRAM cell [4]

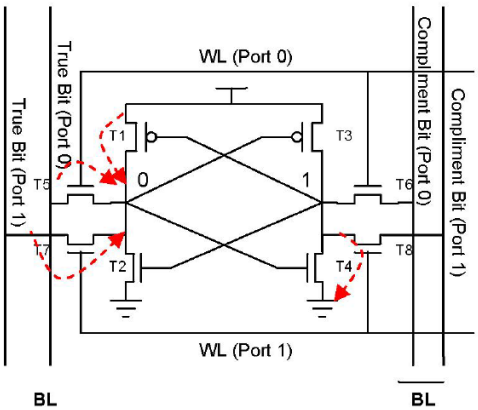

Figure 19: dual-port SRAM cell [4]

The following equations describe the leakage currents per cell displayed in Figure 18 and Figure 19 [4]:

$$I_{single\ port} = I_{T1} + I_{T5} + I_{T4}$$
$$I_{dual\ port} = I_{T1} + I_{T5} + I_{T4} + T_7$$

Previously used techniques to reduce leakage current were based on a fixed bank size and employed duplicated word and bit lines at the expense of either moderate performance degradation or a large area overhead. Bajwa et al. [4] proposes a new cache architecture using isolation nodes to partition a cache memory block into two virtually independent sections that also employ real-time access of addresses via multiple ports.

Figure 20 shows the proposed placement of the Isolation Control Line (ICL) and isolation node on the corresponding bit lines to divide the block into an upper port and lower port, respectively. This approach enables dual-port access without the need of a second pair of bit lines and thus reduces the leakage current and the silicon area needed. Even though additional ICLs are placed every n word lines, Bajwa et al. [4] states that the performance degradation for a value of n = 8 poses no negative effects. The statistical pattern of accesses of addresses of targeted applications determines the overall placement of the nodes.

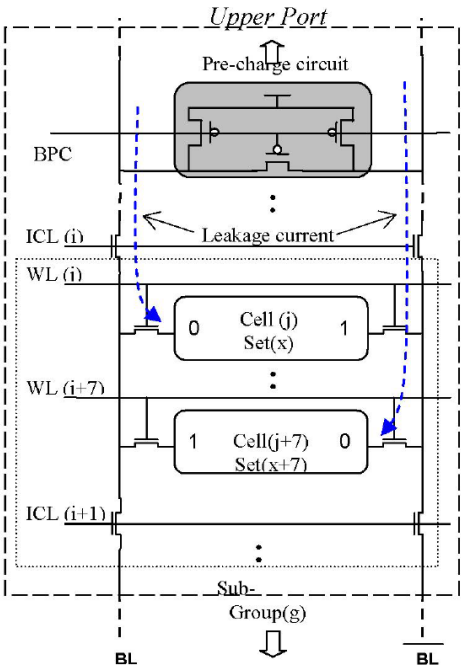

Figure 20: ICL and Isolation node placement [4]

The efficiency of this dynamically partitioning is based on an applied algorithm to determine the ICL and isolation node placement. Considerations that go into determining the optimal algorithm include: delay, power dissipation and the overall complexity of the proposed algorithm. Bajwa et al. [4] evaluates two algorithms: (1) an algorithm for optimal partitioning that minimizes bit line latency and power dissipation and (2) an algorithm that does not require a new partition for every memory access. The pseudo code for algorithm (1) is as follows:

addr(A) <1:n>; addr(B) <1:n>;
where adr(A) = i > addr(b) = j;
if i = j + 1 return ICL(j)
else return ICL(j) and ICL(i-1)

The pseudo code for algorithm (2) is as follows:

addr(A) <1:n>; addr(B) <1:n>;
where adr(A) = i > addr(b) = j;
k = current ICL;
if (j ≤ k < i) return NUL (no new DMP);
else return (j + (i-j)/2);

Applying the above described dynamic memory configuration reduces the silicon area that is needed because no additional bit lines and pass transistors are needed. This results in a reduced leakage current and reduced bit line pre-charge current by a factor of 50% of the value of a typical hardwired multi-port memory. Lastly, the dynamic configuration also introduces less latency due to shorter active bit lines. The leakage current of a memory core with N rows and M columns can now be calculated using the following formula:

$$I_{new} = \frac{N}{2} M(I_{T1} + I_{T5} + I_{T4} + T_7)$$

A paper entitled "Cache Memory Architecture for Leakage Energy Reduction" by Tanaka et al. [5] states that future high performance processors need even larger amounts of cache to bridge the speed gap between the processor and the external memory. Given the increase in cache size, it is said that energy dissipation in cache memory makes up 50% of the total energy dissipation of the processor system. Higher transistor counts and increased clock frequency result in decreased battery lifetime and higher temperature. To ensure performance improvement of future microprocessors, it is necessary to improve the energy efficiency of cache memory systems.

2) Software Self-Invalidation and Data Compression

Tanaka et al. [5] introduces a low-energy cache memory hierarchy for on-chip multiprocessors, which exploits gated-Vdd transistors and explicit gated-Vdd control. Two mechanisms are introduced: (1) leakage energy reduction by software self-invalidation and (2) leakage energy reduction by data compression. The memory hierarchy is displayed in Figure 21. It consists of L1 instruction and data caches, a write buffer, a L2 unified write-back cache on chip, and an external main memory. The compressor and decompressor blocks are used to exploit energy leakage reduction as explained later.

Cache blocks can become invalid if they receive an invalidation request. Turning off these invalid blocks using a gated-Vdd results in significant energy savings. In addition to this method, a self-invalidation mechanism to further increase the number of blocks that can be turned off is applied. This mechanism makes use of a modified load/store instruction called "last-touch load/store". In addition to the conventional load/store function, the new instruction can validate cache blocks after accessing them.

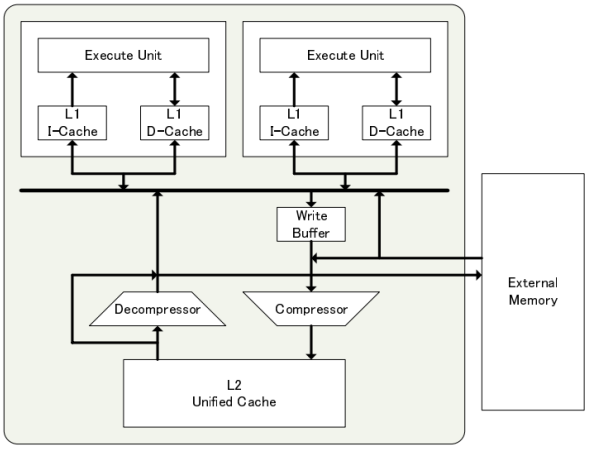

Figure 21: Cache memory hierarchy [5]

The invalidation is based on two conditions: (1) a cache block is invalidated at the same time as it is accessed, and (2) a word is marked when it is accessed. The cache block is invalidated when all words in the block get marked [5].

To enable the abovementioned improvements, slight modifications to the conventional L1 cache memory structure are necessary. Last-touch flag bits are added as a part of the L1 cache tag information. Each flag corresponds to a word in the block. Figure 22 illustrates the memory structure for a 16-byte block made up of four words each. Tanaka et al. states that "When a last-touch-word load/store instruction is executed the corresponding flag bit is cleared. On the other hand, when a last-touch-block load or store instruction is executed, all flag bits are cleared (as depicted in the second row in the figure). Then, a block is invalidated when all the flag bits are cleared." [5].

The gated-Vdd design is implemented as shown in Figure 23. It is worth mentioning that this figure is conceptual since the address tag and data parts of the block relate to one or more gated-Vdd transistors.

Valid	Last-touch flag bits				Address tag, etc.	Data			
						Word0	Word1	Word2	Word3
1	0	1	0	1	. . .	ltw ld		ltw ld	
0	0	0	0	0	. . .	ltb ld			
1	1	1	1	1	. . .				
0	0	0	0	0	. . .	ltw ld	ltw ld	ltw ld	ltw ld
1	1	1	1	1	. . .				
. . .	. . .				. . .	. . .	. . .	. . .	. . .

Figure 22: L1 cache memory structure [5]

The data compression technique employs data compression thresholds of ¾, ½ and ¼. Compressed blocks are stored in the L2 cache and the remaining space is turned off using gated-Vdd transistors. The overhead of compression and decompression is negligible because the L2 cache access frequency is not high. In general, a compression ratio as small as possible is desirable, because data

compression as a whole results in higher processing cost, larger chip area, and longer latency. These factors are important when considering the tradeoffs between cost, performance, and the amount of energy conserved [5].

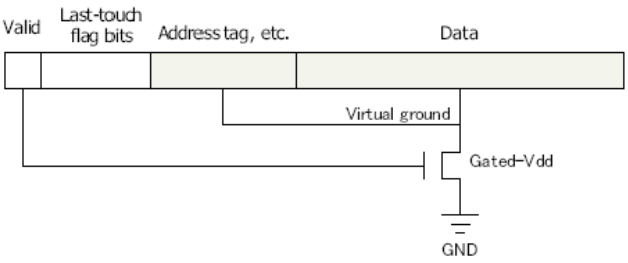

Figure 23: (conceptual) L1 gated-Vdd control [5]

The tag information for the L2 cache is shown in Figure 24. "c1" and "c0" correspond to the compression thresholds used above. A combination of "00" equals no compression, "01" equals ¾ compression, "10" equals ½ and "11" equals ¼ compression (of the original size of the data). Three transistors are needed to support this feature, as shown [5].

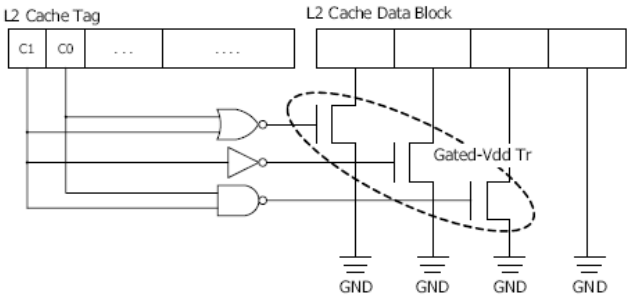

Figure 24: L2 gated-Vdd control [5]

Five kernel programs in the SPLASH-2 suite were used for the evaluation of the software invalidation technique.

Table 1 contains the input data size and input file. Table 3 displays the simulation results, normalized to "base", which refers to an execution without gated-Vdd control. "inv.off" is with gated-Vdd control of invalid blocks and "last-touch" is the execution with invalid blocks supported by the modified last touch instructions.

Table 2 lists the number of self-invalidations performed by last-touch word or block instructions. The results in Table 2 and Table 3 show that leakage energy was significantly reduced for last touch instructions for "LU-noncontig[uous]" and "RADIX" [5].

Table 1: Input data sizes / input file for SPLASH-2 programs [5]

Program	Input data size / input file
FFT	65,536 complex
LU contig	256x256 matrix
LU non-contig	256x256 matrix
RADIX	262,144 keys
CHOLESKY	wr10.O

Table 2: The number of self-invalidations [5]

Program	# of self-invalidation		# of ltw
	ltb	ltw	instructions
FFT	36	66,044	264,190
LU contig	18	82,150	396,571
LU non-contig	18	157,156	908,555
RADIX	25	272,420	1,179,675
CHOLESKY	9	14,839	71,630

Table 3: Results of last-touch load/store scheme in L1 cache [5]

Program	Exec. scheme	Exec. time	Leakage energy
FFT	base	1.0000	1.0000
	inv.off	1.0000	0.9938
	last-touch	0.9999	0.9752
LU-contig	base	1.0000	1.0000
	inv.off	1.0000	0.8447
	last-touch	0.9999	0.7943
LU-noncontig	base	1.0000	1.0000
	inv.off	1.0000	0.6697
	last-touch	0.9968	0.5369
RADIX	base	1.0000	1.0000
	inv.off	1.0000	0.9741
	last-touch	0.9989	0.5353
CHOLESKY	base	1.0000	1.0000
	inv.off	1.0000	0.9992
	last-touch	0.9997	0.9895

B. *Error Detection Codes (EDCs) and Error Correction Codes (ECCs)*

Energy particles can cause soft errors in cache memories. Modern processors employ EDCs and ECCs to counteract these errors. Employing these techniques result in a significant overhead in terms of area and energy. Farbeh [18] proposes a new cache architecture to reduce energy consumption and reduce the area overhead that result from using EDCs and ECCs in L1 caches.

Soft errors are a major reason for system failures. They can appear in the shape of Single Event Upsets (SEUs) or Single Event Multiple Bits Upsets (SEMUs). The technological advances mentioned in previous sections (improvement of feature size and supply voltage) result in a new challenge of handling the increased amount of SEUs and SEMUs. In a 40nm feature size, the probability of an SEMU caused by a particle strike is about 40%; this percentage increases if low power techniques are applied [18].

The newly proposed architecture called "Per-Set Protected Cache (PSP-Cache)" makes use of the fact that in a set associative L1 cache, data words in all cache ways are accessed in parallel. This enables minimization of the number of redundant bits without reducing the protection capability of EDCs and ECCs [18].

Figure 25 displays a conventional cache architecture (left side) and the proposed architecture (right side). In a conventional cache architecture, data is applied to "Way Selection Logic". Further, the output of "Tag Comparison Logic" selects data based inputs from the cache.

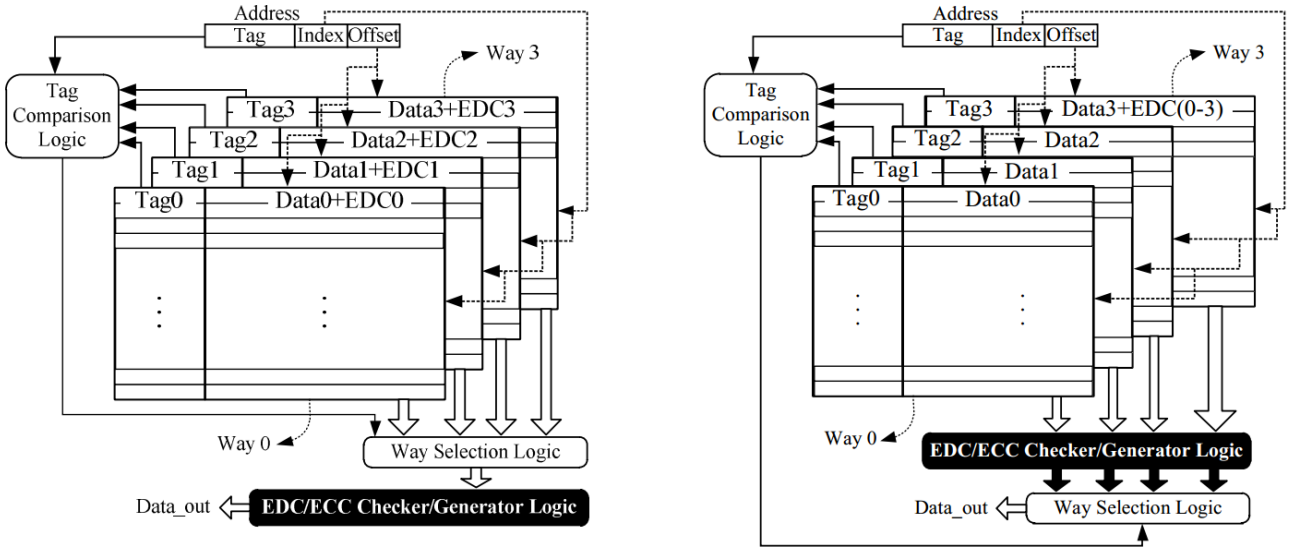

Figure 25: Abstract view of (left) conventional cache architecture and (right) proposed PSP cache architecture [18]

This data then proceeds to the EDC/ECC Checker/Generator Logic and is delivered to the data bus. As one can see in the right-hand side of Figure 25, a single code gets assigned to the data of all cache ways and the EDC/ECC Checker/Generator Logic operates on all accessed data [18].

Further, a parity code is applied, which is mostly used to protect instruction and data cache data integrity. For the parity code, the number of redundant bits needed to detect a specific number of bit errors is independent of the data length. Therefore, the number of bits required to protect a single cache way is equal to the number of bits required to protect all N cache ways [18].

"The main features of the proposed architecture are as follows:

- A negligible modification of cache architecture is required to implement PSP-Cache;
- It is applied to the tag array of cache memories in addition to data array. Moreover, both D-cache and I-cache can take advantages of this architecture;
- It is independent of cache protection granularity. Hence, all set-associative caches with per-X-bit EDC/ECC protection, when X is between a single byte to the cache line length, can be transformed to PSP-Cache architecture;
- The efficiency of the proposed architecture improves by increasing the cache associatively." [18]

This architecture was evaluated in terms of energy consumption, area, and reliability.

1) *Energy Consumption and Area Overheads*

Redundant bits are the major source of area and energy overheads. The Checker/Generator unit's contribution to both area and energy overhead is smaller than 1% and is therefore negligible. The results displayed in Table 4 show, that the reduction in the number of redundant bits in PSP-Cache is proportional to the cache associatively. Required redundant bits are reduced by 50%, 75%, 87.5% in 2-way, 4-way and 8-way associative caches, respectively [18].

Figure 26 displays the improvement in energy overhead of PSP-Cache for different cache architectures, normalized to the baseline cache. An overhead reduction by 49%, 73% and 85% for 2-way, 4-way and 8-way set-associative caches can be achieved, respectively.

2) *Reliability Analysis*

For this analysis, the effect of the newly introduced architecture on SEUs and SEMUs is taken into account. It was concluded that the architecture does not hurt the capability to detect and correct SEUs or SEMUs, regardless of the data length. Thus, it does degrade the protection capability of EDC/ECC codes [18].

Table 4: Number of redundant bits required to protect cache for different cache associatively and protection codes [18]

EDC/ECC type	Number of redundant bits			
	Baseline cache	**PSP-Cache**		
		2-way cache	4-way cache	8-way cache
1-bit parity per word	N	$\frac{N}{2}$	$\frac{N}{4}$	$\frac{N}{8}$
2-bit interleaved parity per word	$2N$	N	$\frac{N}{2}$	$\frac{N}{4}$
4-bit interleaved parity per word	$4N$	$2N$	N	$\frac{N}{2}$
8-bit interleaved parity per word	$8N$	$4N$	$2N$	N
SEC-DED (72,8)	$8N$	$\frac{9N}{2}$	$\frac{10N}{4}$	$\frac{11N}{8}$
N : Number of words in the cache				

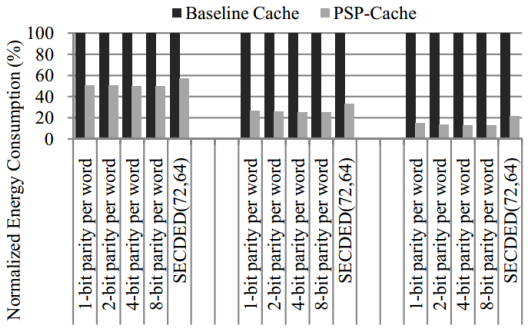

Figure 26: Normalized dynamic energy for baseline cache and PSP-Cache [18]

C. *Exploiting Row Access Locality*

DRAM is commonly used as the main memory of computer systems but its access latency continues to be a critical bottleneck for system performance. In the paper "ChargeCache: Reducing DRAM Latency by Exploiting Row Access Locality", Hassan et al. [6] introduces a new concept called ChargeCache. The goal of ChargeCache is to reduce the average DRAM access latency time without modifying existing chips, thus improving overall main memory and cache memory performance. The general principle is to keep track of the amount of charge of a recently accessed row and to use this information to determine the timing in which a row can be accessed [6].

The architecture mainly dictates the latency of DRAM, specifically the length of the bit line. Each pair of transistors is connected to sense amplifiers through a bit line as illustrated in Figure 27.

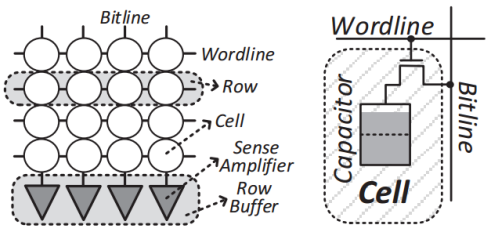

Figure 27: DRAM Sub-Array (left) and DRAM cell (right) [5]

Sense amplifiers are heavy in terms of cost; thus, many DRAM cells are typically connected to the same bit line. The additional length results in an increase in resistance and parasitic capacitance on the path between the cell and the sense amplifier, thus higher latency. To achieve an improvement in performance and energy efficiency, two major observations are exploited [6]:

- Due to bank conflicts, many applications tend to access rows that were recently closed. This form of temporal locality is referred to as Row Level Temporal Locality (RLTL). The important outcome of this observation is that a DRAM row remains in a highly-charged state when accessed for the second time within a short interval.
- DRAM cells leak charge over time. The charge is either replenished by an access to the row or a refresh operation. The current amount of charge determines the operation time of the sense amplifier and therefore dictates the access latency. Hence, recently replenished cells can be accessed using a significantly lower access latency than the case when the cell has less charge.

ChargeCache is a new mechanism that exploits these observations. The main idea is to keep track of the addresses of recently accessed DRAM rows and provide accesses to these with a latency that depends on their level of charge. The memory controller maintains a small table that contains the addresses of recently accessed rows. The memory controller then checks this table before accessing a new row to check if the address of the row is present. A hit in this sense means that the row can be accessed with lower latency; otherwise regular latency timing occurs. This process requires a mechanism to periodically invalidate entries from the table so that the table only contains addresses of cells with a high amount of charge [6].

Figure 29 illustrates the steps needed to transfer data from a DRAM cell to the sense amplifier and their mapping to DRAM commands. A detailed explanation can be found in [6]. States 4 and 5 refer to the fully recharged state of the cell after an access. State 6 represents the leakage of charge. If the cell has not been refreshed for a certain amount of time, thus has lost too much charge, its state may be flipped in the table. To avoid these cases, the controller refreshes DRAM cells within a certain interval, the so-called refresh interval. As stated earlier, a low amount of charge corresponds to a longer access latency. This means, that the sense amplifier takes longer to reach states 3 and 4. If the charge is high, the perturbation caused by the cell on the bit line voltage is high; the cell can be accessed earlier because it takes a shorter time to reach states 3 and 4. This enables a reduction of the time intervals tRCD and tRAS, as shown in Figure 29.

ChargeCache is implemented by adding (two main components to the memory controller): a tag-only cache that stores the addresses of the highly-charged rows and a set of two counters to invalidate entries from the table. An overview of these components is shown in Figure 28.

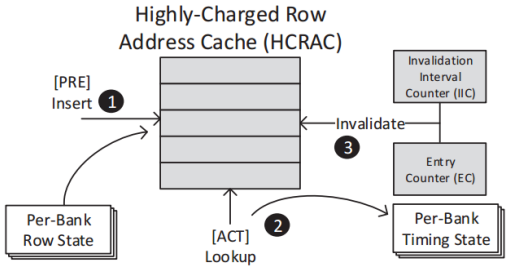

Figure 28: Components of the ChargeCache Mechanism [6]

To expand, if a PRE command is issued to a bank, the memory controller stores the address of the row that was activated in the table. Some interfaces allow the memory controller to pre-charge all banks with a single command. In this case, all addresses are inserted into the table. The table itself is finite, therefore the oldest entries may be evicted if no more space is available and new row addresses are entered [6].

If an ACT command is issued, ChargeCache searches for the corresponding row in the table. On a hit, lower tRCD and RAS are applied for subsequent READ/WRITE and PRE operations, respectively [6]. A miss results in using the default timing [6].

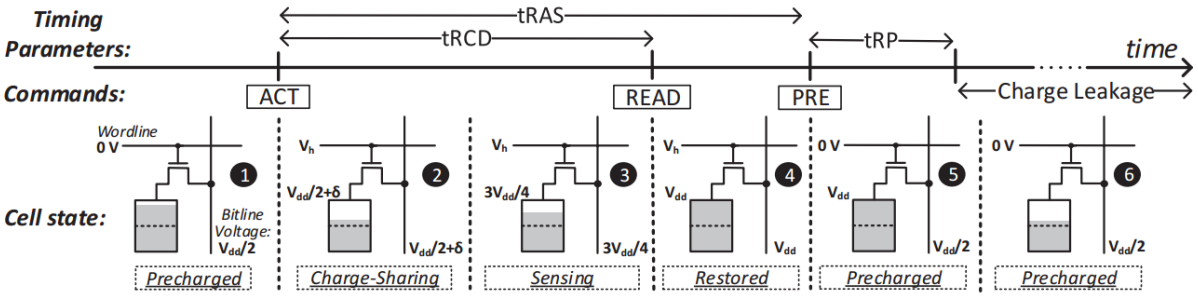

Figure 29: Timing parameters and commands used to read data from DRAM [6]

Given the continuous leakage of cells, entries must be invalidated after a certain amount of time. Using a clock to track the expiration time for each entry would result in increased storage cost and complexity of the implementation. Therefore, two counters that count clock cycles are used [6].

ChargeCache [6] evaluates the reduction in DRAM timing parameters, namely the time periods tRCD and tRAS using SPICE simulations. Different charge amounts will result in different bit line voltage levels during cell activation, as displayed in Figure 30.

As shown in Figure 32 below, ChargeCache achieves up to 8.1% (11.3%) speed-up for a single core (eight-core) processor and on average obtains 2.1% (8.6%) speed-up for a single core (eight-core) processor. Combining ChargeCache with NUAT leads to an improvement of 9.6% on average for an eight-core processor.

Because ChargeCache reduces the overall execution time, in addition to improving access time, ChargeCache leads to significant energy savings as well. Figure 31 illustrates that ChargeCache reduces energy consumption by up to 6.9% and an average of 1.8% for a single-core processor system, and a reduction of up to 14.1% and an average of 7.9% for an eight-core processor system.

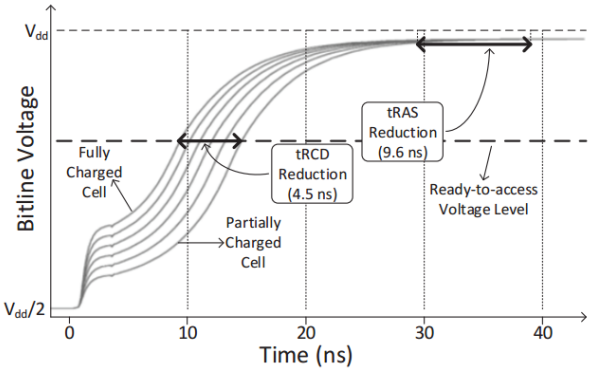

Figure 30: Effect of initial cell charge on bit line voltage [6]

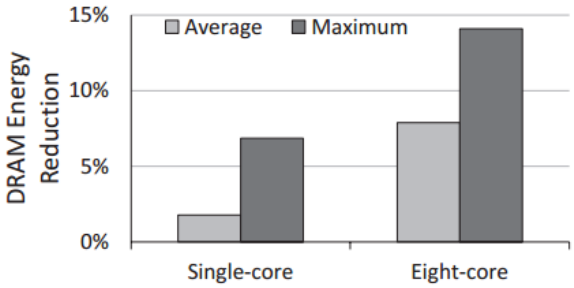

Figure 31: DRAM energy reduction of ChargeCache [6]

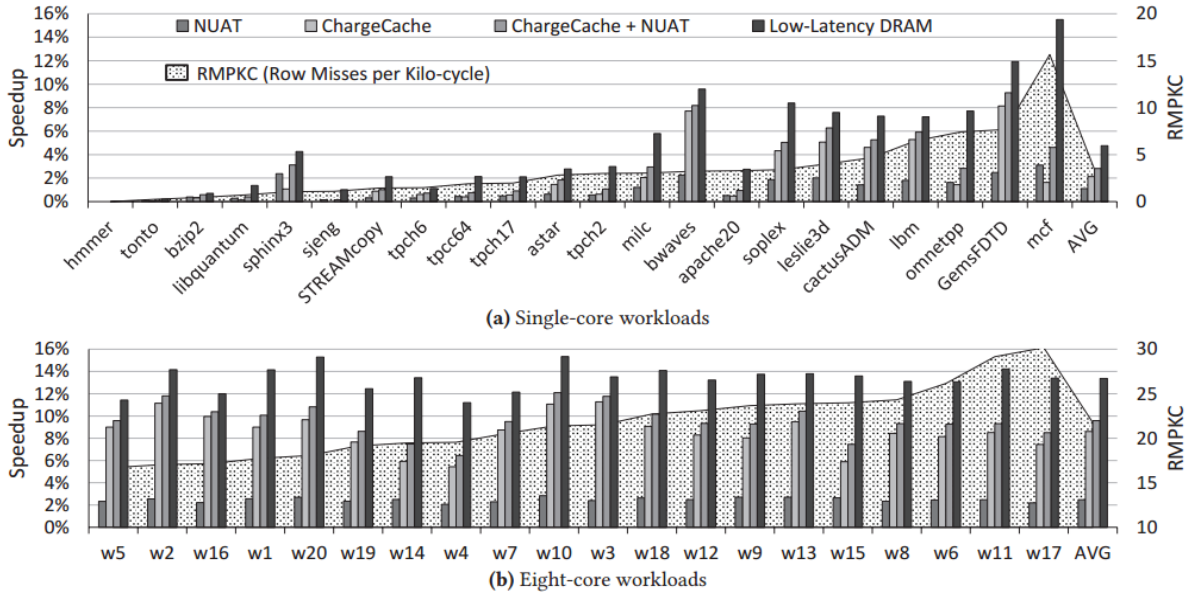

Figure 32: Speedup with ChargeCache, NUAT and Low-Latency DRAM for single-core and eight-core workloads [6]

V. Special Topics in Cache Memory Architecture

The cache memory architecture plays an important role in modern day computer systems. As discussed above in Section 1, the advancement of performance of cache memory systems is not as rapid as that of processor performance. In recent years, new techniques have been proposed to improve cache memory performance. This section provides insight into the analysis or core memory architecture performance related to real-world workload analysis versus simulated workload analysis. This section then focuses on the advances in core cache memory architectures, provides a study of the leading methods to improve core cache memory architectures, and illustrates their performance.

A. *Cloning of the Spatial and Temporal Memory Access Behavior*

We first start with a discussion and analysis related to verifying optimal core memory architectures on real-world workloads versus simulated workloads. One persistent problem faced by computer architects is that most clients or companies do not wish to share their proprietary workloads (e.g., their software). In other words, companies are not willing to provide system architects with their proprietary software that runs on high performance computer systems. Thus, proper analysis and optimization of cache memory is not possible. In order to address this problem, architects use open source versions of software or have to reconstruct the software based on the description provided to them. This practice is time consuming and still leaves architects guessing about the requirements of the clients. An alternative is to clone the software without reading or accessing the proprietary information in the software code.

1) *STM: Cloning of the Spatial and Temporal Memory Access Behavior*

Awad et al., proposes a solution which Awad et al. generally refers to as "STM" (Spatial and Temporal Memory) [19]. To clone the memory access behavior of a software program, the cloning methodology should read both spatial and temporal memory access behavior. STM gathers memory access trace behavior statistics and then generates clones that produce memory access models similar to the proprietary software program. The trace clone or synthetic clone generated is then passed through a simulator. The inter-working of the STM can then be understood.

Consider an example of the following memory access pattern at a cache block level: 0, 1, 2, 3, 1, 2, 3, 4, 2, 3, 4, 5, 3, 4, 5, 6, ...,: the access pattern shows a simple block address stream, containing both spatial and temporal locality behavior. Another assumption is that the cache is fully-associative with the size of four cache blocks and employs LRU (Least Recently used). In the example, the access pattern misses every four access, which results in a hit rate of 75%. Further, we consider optimal and sub-optimal stride values (where a stride value is the address increment value in memory between the start of successive memory elements, measured by the overall size of the memory element) [19]. The working of the STM can be understood by using six different approaches to generate a clone for the abovementioned example [19]:

- In the first approach as shown in Figure 33, a single dominant stride +1 is considered. In this case the stride value generates a hit rate of 100% which is not correct.
- In the second approach, as shown in Figure 33, an address transition graph is used, where each block address is a node and each edge connects the node

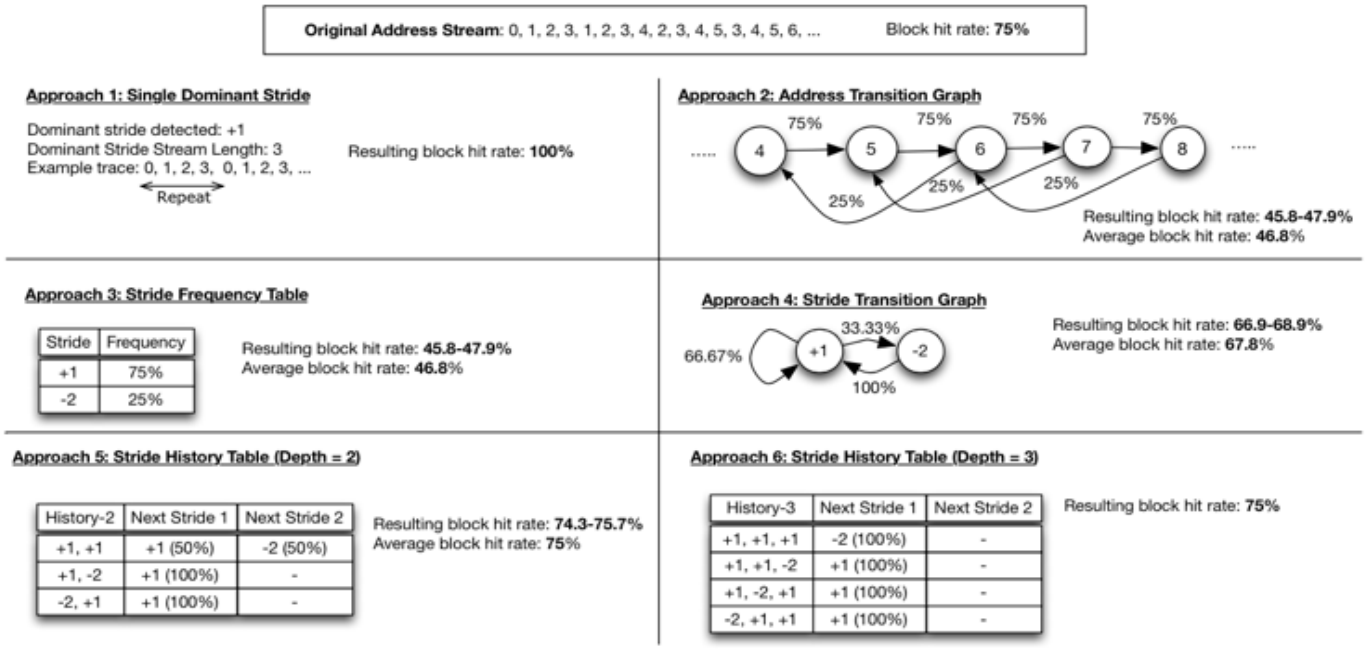

Figure 33: Methods for modeling memory access [19]

- to its successor and the transition probability. The measured hit rate, when 100,000 Monte Carlo simulations were run, was 46.8%. The measured hit rate is not even close the hit rate of the original stream access.
- In the third approach, strides are recorded instead of addresses. After construction, an address trace using the stride frequency table, the measured cache hit rate over a 100,000 Monte Carlo simulations was 46.8%, which is still not close to the desired hit rate.
- In the fourth approach, a stride transition graph is used, where each stride is a node and the probability of transition is the edge. Over a 1000,000 Monte Carlo simulations, the hit rate is found to be 67.8%, which is not the desired hit rate but much closer than the previous approaches.
- In the fifth approach, instead of just using the previous stride, the history of the stride transitions is maintained. With a history depth of two (2) the hit rate over a 100,00 Monte Carlo simulations is found to be between 74.3 to 75.7%.
- In the sixth approach, the depth of history of the stride, when increased to three (3) gives a hit rate of 75%. STM adopts the stride pattern approach as its foundation.

The profiling of memory access is done using the profiling structure as shown in Figure 34. The Stack Distance Probability (SDP) table shows the probability of accessing the most recently used entries at the head of the table. The Stride Pattern Probability shows the possible stride values that can be the successor for the past value. In Figure 34, M represents the depth of the stride history pattern, (Z0, F0) denotes the first stride value. The SDP tables are updated using the tags of the blocks which are most recently used. When a new address is updated the SDP table is searched for a matching tag. If the tags match, then the SDP table is updated, else the Stride Pattern Probability (SPP) table is updated. Figure 35 shows how the SPP table is updated. Apart from this, during profiling the fraction of updates to the SDP table (fSD) and the number of writes are also collected.

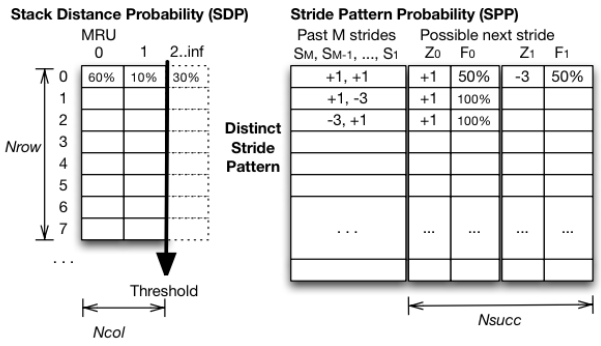

Figure 34: Profiling structures [19]

The next step is to generate a clone based on the profiling. Both the SDP and SPP tables are scaled by the scaling ratio, which is the ratio of the required number of memory references (N_{new}) to the number of memory accesses calculated during profiling ($N_{orginal}$). The clones are generated using four random numbers which help choose: (1) which table to use (SPP or SDP), (2) to generate a read or a write, (3) to select the row in the SDP table, and (4) to select the column in the SDP table. This process is repeated until N_{new} accesses have been generated or until relevant entries in the SP tables are exhausted [19].

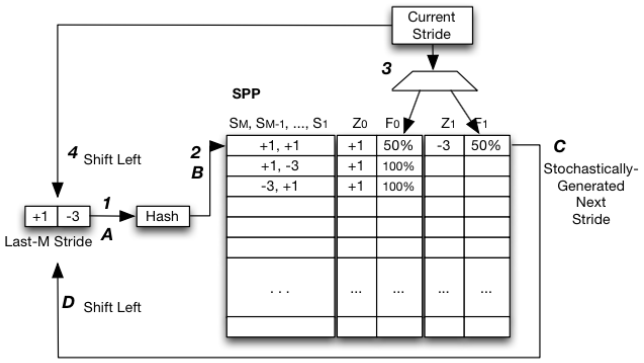

Figure 35: Updating SP tables [19]

2) Validation and Evaluation Analysis

To validate STM as taught by Awad et al., a simulator called the "gem5" was used to run 27 SPEC CPU2006 benchmarks. The system configurations used for collecting the profiles is shown in Table 5. For validating STM, 400 different configurations per the benchmarks were used for predicting accuracy of L1 cache miss rates, L2 miss rates, and TLB miss rates. The metrics used for validation rank the accuracy and correlation coefficient. The relative performance ranking is calculated by comparing every different cache/prefetcher/TLB configuration with an appropriate L1/L2/TLB miss rate.

Table 5: Configurations for the system used for profiling [19]

Component	Configuration
CPU	x86-64 processor, atomic mode, 2GHz
L1 Cache	64B blocks, 32 KB size, 2-way, LRU
L2 Cache	64B blocks, 512KB size, 8-way, LRU
Main memory	2 GB
OS	Linux, Gentoo release 1.12.11.1

Figure 36 (see next page) shows the rank accuracies for STM, Single Dominant Stride (SDS), and "West". SDS and West are proposed cloning mechanisms that have been evaluated against STM. The ranking accuracies are obtained by fixing the L1 cache configuration to 16KB size, 2-way associatively with a 64-block size. The L1 cache is augmented with a stream buffer prefetcher. From Figure 37 it can be seen that STM performs better than SDS and West. Although, SDS tries to capture the spatial locality behavior, it is unable to properly capture spatial behavior. For a further understanding, the miss rate map for 12 representative benchmarks for STM is plotted as shown in Figure 37. From Figure 37, it can be seen that STM performs well except against two benchmarks: "Zeus" and "h264". [19].

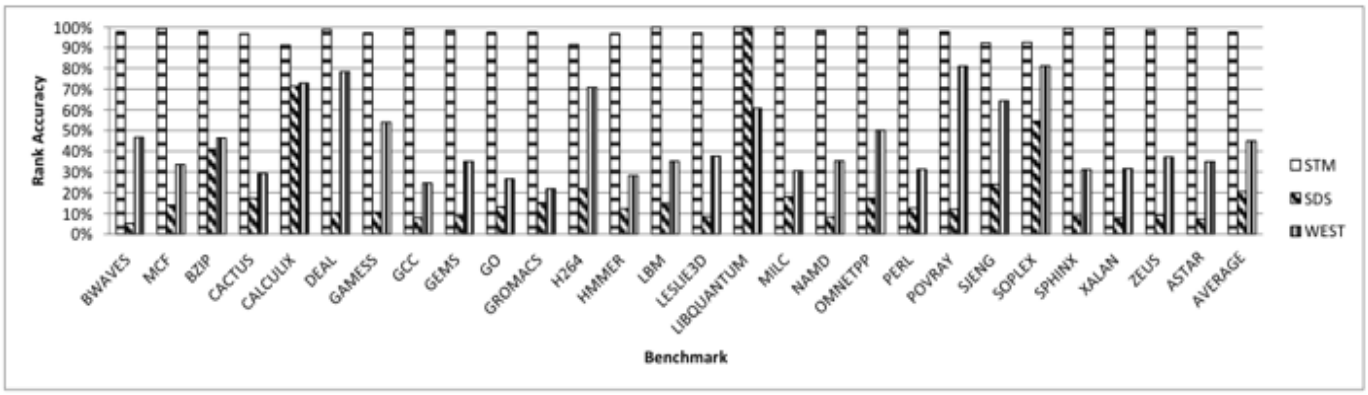

Figure 36: Rank accuracies [19]

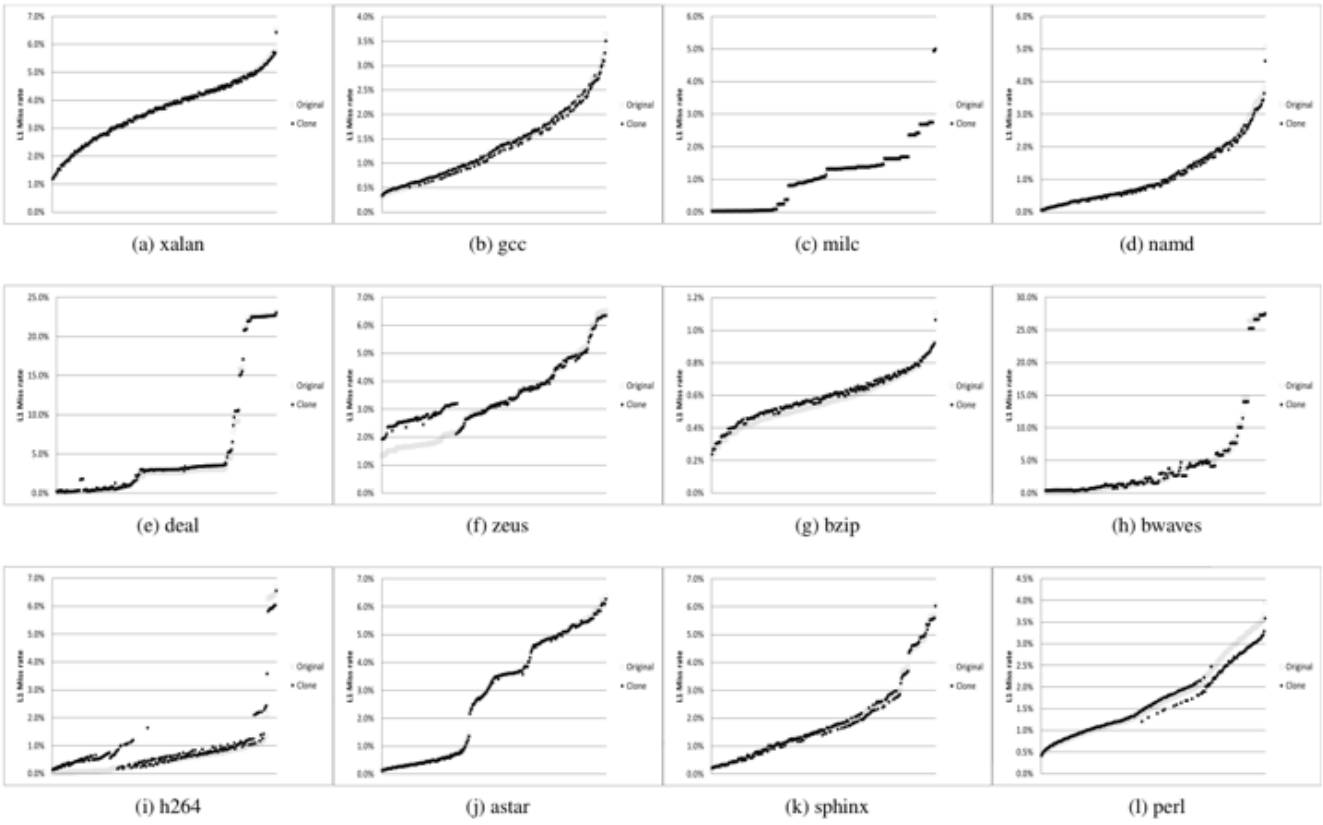

Figure 37 Comparisons between the original and the L1 miss rate [19]

B. *Read Write Partitioning (RWP)*

The processor is stalled when there is a cache read miss and the instructions following the miss are dependent on this data. In a typical code, write misses are usually not in the critical path of execution. Whereas read misses or load requests are in the critical path. Most cache management mechanisms do not take this into account. To exploit this disparity, the cache lines can be differentiated into clean and dirty lines.

1) Read Write Partitioning Improvements

The paper "Improving Cache Performance Using Read-Write Partitioning" by Kahn et. al. [23] provides improvements to Read Write Partitioning (RWP) that provides an average of 5% speed-up across the entire SPEC CPU2006 suite. In this section the motivation, framework and performance analysis is illustrated for RWP improvements.

The motivation of RWP is to exploit the disparity between read and write misses to improve the cache performance. There have been many attempts to differentiate between critical and non-critical lines. Load and store instructions are treated differently in the processor pipeline. The goal of RWP is to increase the probability of cache hits for critical read requests. For RWP, RWP divides the last-level cache into two logical partitions for clean and dirty lines. It also predicts the best partition sizes to increase the likelihood of future read hits.

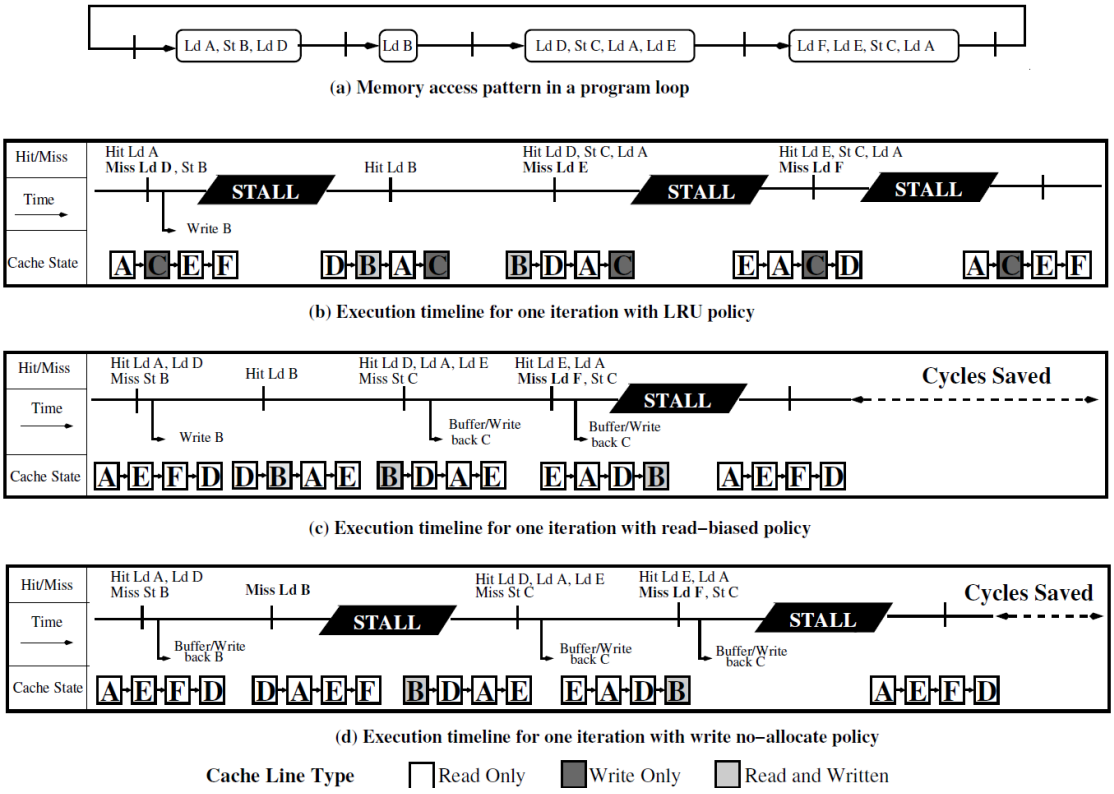

Figure 38: Drawbacks of not taking read-write differences into account [23]

The main motivation can be achieved by simply favoring cache read lines (e.g., read requests for cache lines) over write lines (e.g., write requests for cache lines). To achieve this, the cache can be sorted into two categories, read lines and write only lines, and then based on that, favor read lines when making replacement decisions. This classification is only possible by predicting if clean lines will be written to or if dirty lines are being read.

Workloads with write only lines can be categorized into two categories: (1) read-intensive and (2) write-intensive. The majority of the cache lines for read intensive workloads are clean lines. The write intensive workloads produce a large number of writes. Writes are often requested by subsequent reads while the cache lines are still residing in the L1 cache. After eviction from the L1 cache, however, dirty cache lines are rarely reused. Not all dirty lines are write-only lines, thus making it hard to solve the problem of write-only lines by not allocating write lines in cache. Different workloads exhibit different types of behavior based on different mixes of write-only, dirty-read, and clean lines. A much more sophisticated approach (which can identify write-only lines, or favor one type of access over another based on the likelihood of future reads) is required to improve performance across multiple workloads [23].

2) Example of the Need for RWP

To understand the benefits of read write partitioning, consider the example shown in Figure 38: Figure 38(a) shows a loop with a burst of memory references occurring at four different points in the execution of a loop. Out of the six (6) cache lines, "B" is read and written to and "C" is only written to. Using a LRU replacement algorithm for a fully associative cache with a total of 4 lines, at the end every iteration "D" gets evicted and the intermittent access to "B" and "D" causes three (3) stalls for each iteration. Figure 38(c) shows the timeline for one iteration with read-biased policy and Figure 38(d) shows the timeline for one iteration with write-biased policy. From this example it can be seen that differentiating between reads and writes in the cache improves execution time. While simple approaches as used in this example show improvement, the method could cause unwanted effects in performance in other areas [23].

3) Read Write Partitioning Framework

The objective is to maximize the number of read hits in the cache so that critical read requests can be executed without stalls. To achieve this, cache lines that are probable to be used as read lines are identified. Most applications have more read lines as compared to write lines. For example, benchmark 483.xalancbmk has more reads in clean lines than dirty lines. Each cache set can be logically divided into two partitions for clean and dirty lines. The per line dirty status bit is used to determine if the line is part of the read partition or write partition. When writing to a clean line, the dirty bit is set and it is logically considered to be part of the write partition.

With respect to partition size, the partition size is not monitored continuously. The partition sizes are adjusted when a new cache line is allocated. When a new cache line is allocated, the system has to decide which cache line to evict. The decision on which line to evict is decided based on the current number of dirty lines. There are three cases that determine which line gets evicted.

- The current number of dirty lines is greater than the predicted best dirty partition set. In this case the least recently used line in the dirty partition is picked by the RWP algorithm.

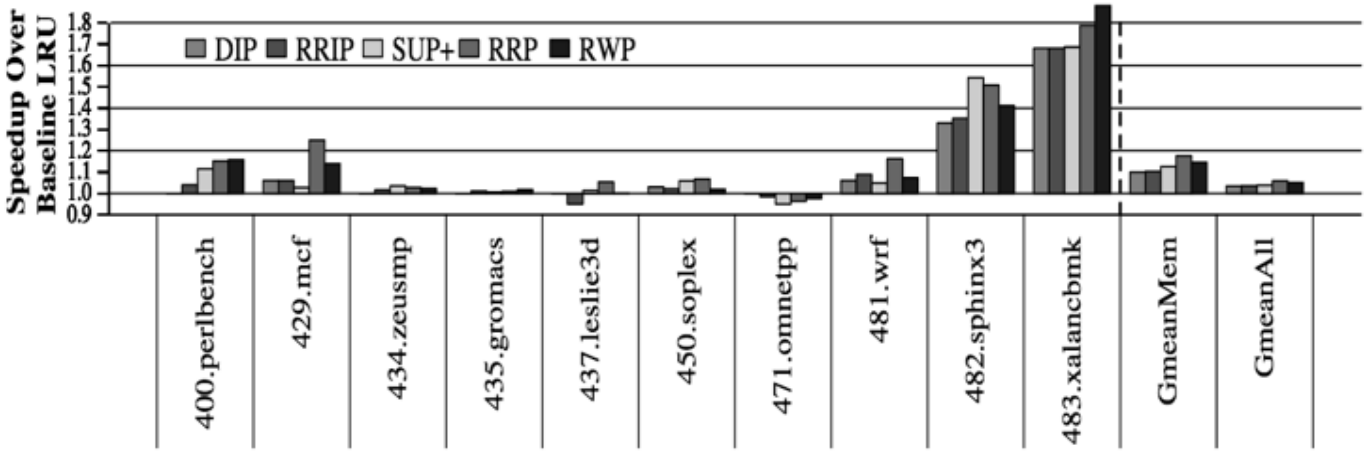

Figure 39: Speedup over Baseline LRU [23]

- The current number of dirty lines is smaller than the predicted best dirty partition set. In this case the least recently used line in the clean partition is picked by the RWP algorithm.
- The current number of dirty lines is equal to the predicted best dirty partition set. In this case the selection of the evicted line (a "victim" line) depends on the memory access type. If it is a read, then the RWP algorithm picks the victim line from the clean partition. If it is a write, then the line to evict is selected from the dirty partition.

RWP checks partition sizes only when clean cache lines are written to. To estimate partition sizes, RWP compares the read reuse exhibited by the clean and dirty lines as if each were given exclusive access to the entire cache. RWP uses the same mechanisms proposed by M. Qureshi et al. in "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches"[28]. To summarize: the RWP is confined only to the last-level cache. The RWP does not attempt to identify de-prioritized write-only cache lines. The write-only cache lines are evicted by changing the clean and dirty partition size [23].

4) Read Reference Predictor (RRP)

The abovementioned paper by Kahn, et. al. [23] further proposes an additional mechanism called Read Reference Predictor (RRP). RRP is used to differentiate between cache lines which are susceptible to further reuse versus cache lines that are not probable to be used again. This categorization is performed based on the likelihood of being reused. This is achieved by using the program counter (PC) of the memory instruction with an emphasis on identifying reuse by subsequent reads and not to predict general reuse by all memory instructions. The reason for focusing on only reads lines is because read misses are more critical as compared to write misses. This method bypasses any cache lines that are likely to only be written to [23].

The frame work of RRP is similar to that of RWP as discussed above. RRP also uses a shadow directory added to sample sets to measure the amount of read reuse. The shadow directory has a cache line tag which shows the amount of read reuse and a hashed PC value of the instruction which is allocated to the cache line. Figure 40 shows how the locality is tracked in an 8-way set associative cache.

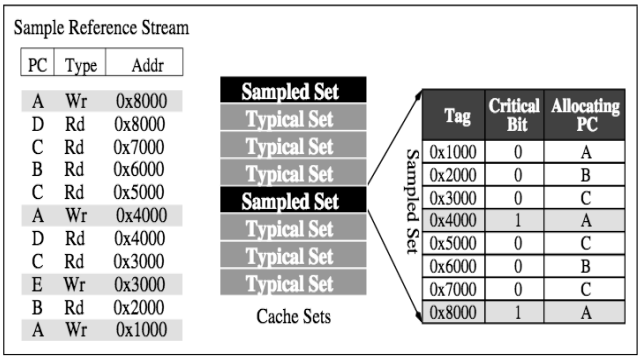

Figure 40: Sample sets in Read-Reference Predictor [23]

Turning to Figure 40 by way of example: the first instruction "A" writes to data address 0x8000. The initial write allocates an entry for cache line 0x8000 in the shadow directory with the critical bit initially set to 0. "D" also reads to the same cache line 0x8000. In this example, the critical bit for 0x8000 is set because it is reused. It can be seen that address 0x3000 is first read by instruction "C" and then subsequently written to by instruction "E". However, the critical bit is not set because RRP only focuses on read reuse. The most important advantage of RRP over RWP is that it explicitly classifies memory requests as exhibiting read and write reuse. However, implementation of RRP is much more complex: the most significant complication presented by RRP is that write back requests are not associated with any program counter (PC). The PC recorded for L1 has to be passed to L2 and then to LLC when the line is written back from L1 and L2. To do this, the RRP solution requires additional storage and logic complexity in the core cache to store and update the hashed-PC value with the cache line tags [23].

5) Evaluation Methodology and Performance Analysis

To evaluate the RRP and RWP methods, both are compared to prior solutions including: Dynamic Insertion Policy (DIP), Re-Reference Interval Predictor (RRIP), and Single-Use Predictor (SUP). All of these solutions do not

take the read write criticality into account. The results shown in Figure 39 are based on the evaluation methods, simulations, and assumptions as noted in Section 5 of the paper by Kahn et al. [23]. Figure 39 shows the speedup for the cache sensitive workloads for a variety of replacement algorithms as compared to a baseline cache with LRU. It can be seen that on average, RRP improves performance by 17.6% and RWP improves performance by 14.6%. The worst performance is for 571.omnetpp because of a sampling errors. For memory-intensive workloads, RRP achieves 7.9%, 7.4% and 5% performance gains over DIP, RRIP, and SUP+, respectively. For memory-intensive workloads, RWP delivers speedups of 6.8%, 6.3% and 3.9% over DIP, RRIP, and SUP+, respectively [23].

Figure 41 shows LLC memory load misses normalized to an LRU baseline for various benchmarks. As shown in Figure 41, the RRP and RWP techniques reduce load misses by 30% and 29% respectively when compared to the baseline. The reduction of load misses can also be noticed when compared to the other configurations which do not take the read write criticality into account [23].

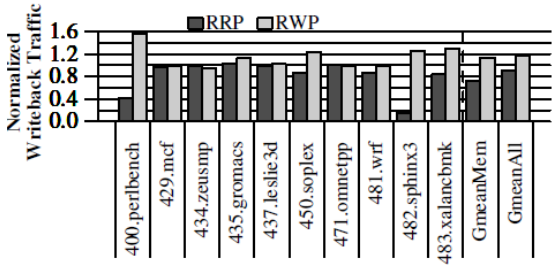

Figure 41: Load Traffic over Baseline LRU [23]

The key contribution of the Kahn et al. paper [23] is the manner in which read and write lines are processed (e.g., a process which favors read lines). This technique provides an improvement of speed-up and also reduces load misses. Overall, this method improves the performance of the cache memory. While Kahn et al, introduces marked improvements, there are a few drawbacks which were not addressed in the Kahn et al paper. For example, the method taught by Kahn et al. does not address multi-threaded workloads that share LLC lines [23].

C. *Removing Cliffs from Overall Cache Performance*

At times, LLC (Last Level Cache) suffers significant performance degradation (e.g., "cliffs") when there are only minor changes in program behavior or changes in the available cache space. Such minor changes results in large changes in the miss rate. To expand, performance cliffs are thresholds where performance suddenly changes as data tries to fit into the cache.

Consider the following example to better understand the cause and effect of performance cliffs in caches: if an application repeatedly scans a 32 MB array, yet the cache is less than 32 MB (say 31 MB by way of example), then the LRU policy will evict cache lines before the lines are hit. However, if the cache is increased from 31 MB to 32 MB, there will be a sudden increase in the hit rate. Figure 42 indicates this type of behavior based on the SPEC CPU 2006 benchmark workload "libquantum". Figure 42 plots Misses Per Kilo-Instructions (MPKI) against the cache size and plainly shows an LRU performance cliff. There is a sudden decrease of MPKI to near zero at a 32 MB cache size. Such performance cliffs give rise to the following three problems:

- Cliffs waste resources and degrade performance. The cache space used does not increase performance, but increases energy consumption and deprives other application of cache space.
- Cliffs make it difficult to assure Quality of Service QoS) by causing unstable and unpredictable performance; small fluctuations in effective cache capacity result in large swings in performance.
- Without the convex miss curves, optimal allocation becomes an NP- hard problem, therefore cliffs make cache management complicated.

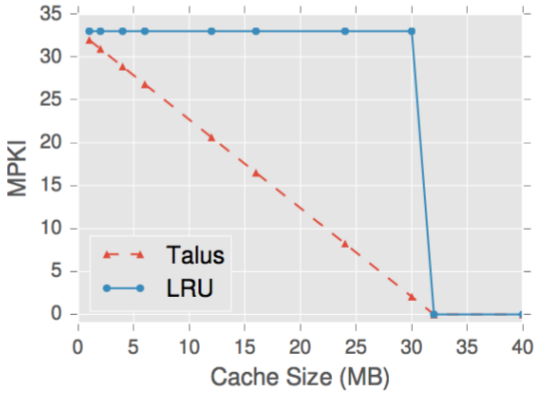

Figure 42: Performance of libquantum over cache sizes. LRU causes a performance cliff at 32MB. Talus eliminates this cliff [20]

1) *The Talus Example: A Simple Way to Remove Cliffs in Cache Performance*

A paper by Beckman et al, entitled "Talus: A Simple Way to Remove Cliffs in Cache Performance" [20] produced good improvement in the area of cache performance cliffs. Talus works by bifurcating the access pattern into two partitions. Talus decreases an application's miss rate in a convex fashion by controlling the sizes of the cache partitions. Talus works by partitioning an access stream. This is achieved by splitting the cache into two hidden shadow partitions. The sizes of these partitions are then controlled in order to monitor how accesses are distributed between the partitions in order to achieve the desired performance. This partition configuration is derived from the "miss curves" presented by Beckman et al [20]. For example: a miss curve is shown in Figure 43. Figure 43 shows a miss curve of LRU performance for an application that accesses 2 MB of data at random and an additional 3 MB sequentially.

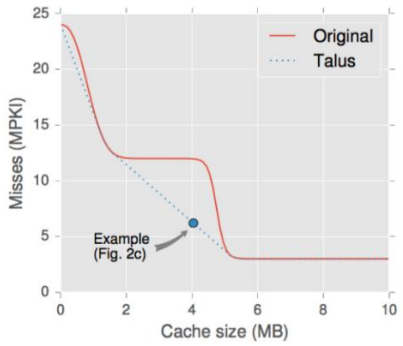

Figure 43: Example miss curve from the application with a cliff at 5MB. The dotted line shows how Talus smooths this cliff [20]

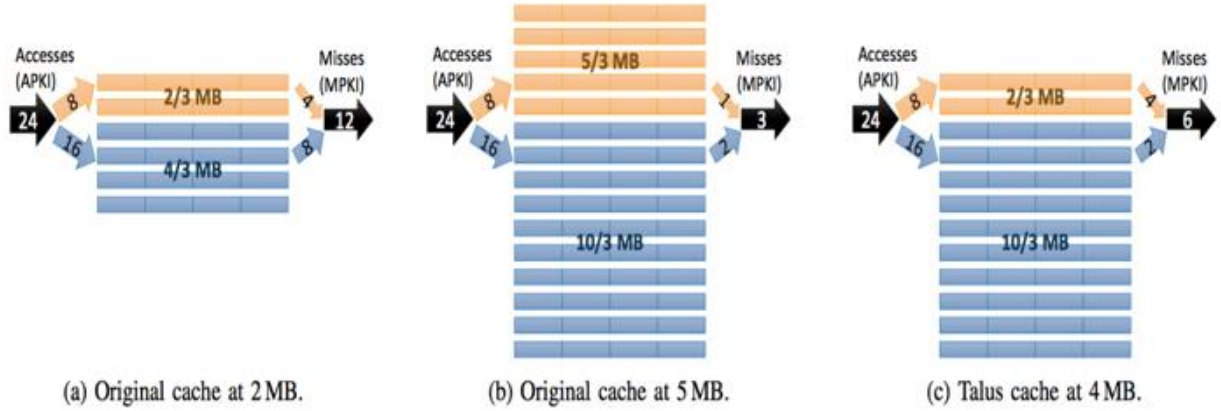

Figure 44: Performance of various caches for the miss curve in Figure 43 [20]

At 5 MB, a performance cliff occurs because MPKI drops from 12 to 3 MPKI. With a 2 MB cache, MPKI is 12 and stays at 12 until about 4 MB; thus there is no benefit from increasing the cache size from 2 MB to 4 MB. In this example, Talus achieves 6 MPKI at around 5 MB. The LRU policy is inefficient at 4 MB, but efficient at 2 MB and 5 MB. In contrast, Talus makes a part of the cache function as a 2 MB cache and the rest like a 5 MB cache. The 4 MB cache behaves like a combination of efficient caches and is therefore efficient overall.

How Talus works is further shown in Figure 44. Talus traces out the convex hull of the original miss curve. The convex hull is the smallest convex shape that contains the curve. Figure 44(a) shows the original 2 MB cache, split into parts in a 1:2 ratio. The application accesses the cache at the rate of 24 Accesses Per Kilo-Instruction (APKI). In a hashed cache, the accesses are evenly split between sets. The top third gets eight (8) APKI and the bottom third gets 16 APKI. The misses shown in Figure 43 at 2 MB will also be split by the same ratio. Figure 44(b) shows an original cache at 5 MB and Figure 44(c) shows how Talus manages a 4 MB cache using set portioning. The top part behaves like the top set of a 2 MB cache which yields a MPKI of 4, and the bottom sets behave like the bottom half of the 5 MB cache yielding a MPKI of 2. This results in a total MPKI of 6. It can be seen, that this values lies on the hull of the convex curve as shown previously in Figure 43 [20].

Figure 45 Talus divides cache space in two partitions of sizes s1 and s2, with miss rates m1(s1) and m2(s2), respectively. The first partition receives a fraction ρ of accesses [20]

2) Design and Implementation

Talus controls a range of parameters. For an application accessing a cache of size "*s*", with any replacement policy, Talus divides the cache into two shadow partitions of sizes "s_1" and "s_2". The first partition has a "ρ" fraction of the access stream, the other partition has a "*1-*ρ" fraction as further shown in Figure 45. Talus follows the following assumptions and rules:

- Miss curves are stable over time and change slowly relative to the reconfiguration interval.
- For a given access stream, a partition's miss rate is a function of its cache size alone; other factors (e.g., associatively) are of secondary importance.
- Given an application and replacement policy yielding miss curve m(s), pseudo-randomly sampling a fraction ρ of accesses yields miss curve m′(s′):

$$m'(s') = \rho m\left(\frac{s'}{\rho}\right)$$

- An application and replacement policy has a size "s" that is linearly interpolated between any two points on the curve, m(α) and m(β), where α ≤ s < β.

$$m_{shadow} = \frac{\beta - s}{\beta - \alpha}\, m(\alpha) + \frac{s - \alpha}{\beta - \alpha}\, m(\beta)$$

- Given a replacement policy and application yielding miss curve m(s), Talus produces a new replacement policy that traces the miss curve's convex hull [20].

Talus uses existing partitioning solutions with few extensions both in software and hardware. The implementation of Talus is further shown in Figure 46. Talus wraps around the systems partitioning algorithm. Talus allows the system portioning to safely assume convexity, then realizes convex performance, instead of proposing its own portioning. Talus generates appropriate shadow partition sizes from the partitioning algorithm. All this is done in a post-processing step.

In the hardware implementation, Talus works with existing partitioning schemes, either coarse grained or fine-grained, with the following extensions:

- Talus doubles the number of partitions in the hardware.
- Talus uses two shadow partitions per logical partition.
- Talus adds one configurable sampling function to distribute accesses between shadow partitions [20].

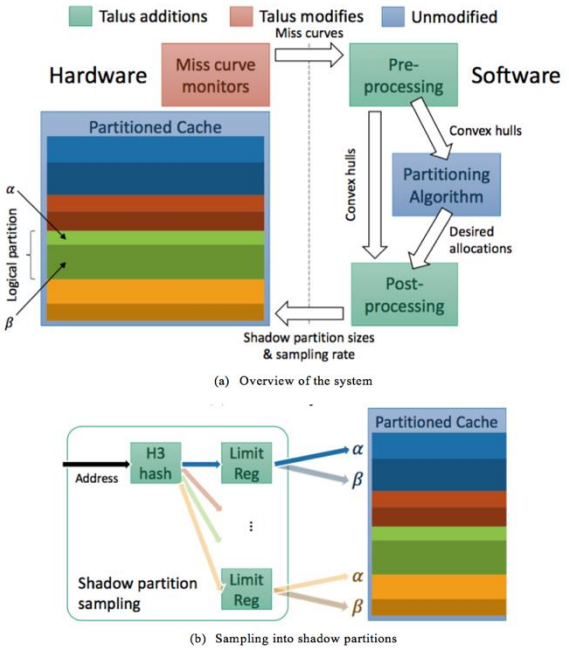

Figure 46 Talus implementation: pre- and post- processing steps in software plus simple additions and extensions to existing partition schemes in hardware [20]

3) *Evaluation and Performance analysis*

Talus [20] was previously evaluated in a variety of settings to demonstrate the following claims:

- Talus avoids performance cliffs, and does not rely on replacement policies and partitioning schemes.
- Talus achieves performance competitive with high performance policies and avoids pathologies.
- Talus is both predictable and convex, so simple convex optimization improves shared cache performance and fairness.

The results for Talus shown in this section are based on the methodologies as shown in the paper by Beckmann et al. [20]. To show that Talus follows the first claim stated above, Figure 47 shows miss curve performance with LRU for two SPEC CPU2006 applications: "libquantum" and "gobmk". Talus was evaluated for three different partition schemes: (1) Vantage (Talus+V/LRU), (2) way partitioning (Talus+W/LRU), and (3) idealized partitioning on a fully-associative cache (Talus+I/LRU). In all of the cases, Talus proved to be effective in removing performance cliffs.

To show that Talus with LRU performs well for a single program, miss curve performance from 0 MB to 16 MB for six (6) SPEC CP2006 benchmark applications are plotted. Talus+v/LRU is compared to a number of high performance policies: SRRIP, DRRIP and PDP. Figure 48 shows how Talus compares to these other policies. From these plots it can be seen that Talus avoids the inefficiencies of other LRU replacement policies, without sacrificing LRU predictability.

To prove that Talus simplifies cache management and improves performance of LLCs, Talus+v/LRU is evaluated on an 8-core CMP (Chip Micro-Processor) with a shared LLC. Figure 49 (see next page) shows the weighted and harmonic speed-ups as compared to un-partitioned LRU policies for 100 random mixes of the 18 most memory intensive SPEC CPU2006 applications. Talus+v/LRU, LRU and TA-DRRIP are compared along with two partitioning algorithms: (1) hill climbing and (2) look-ahead. Hill climbing allocates cache capacity in an increasing manner, based on which partition would benefit the most from the next increase in cache memory space.

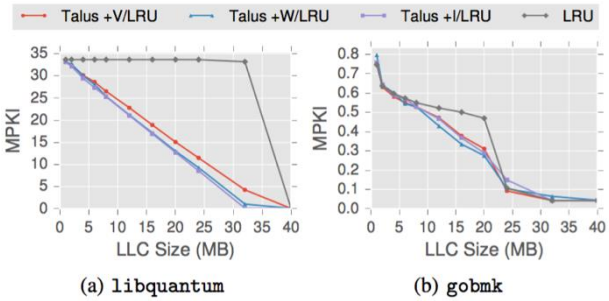

Figure 47: Talus on LRU replacement with various hardware policies: Vantage (V), Way partitioning (W), and Ideal (I) [20]

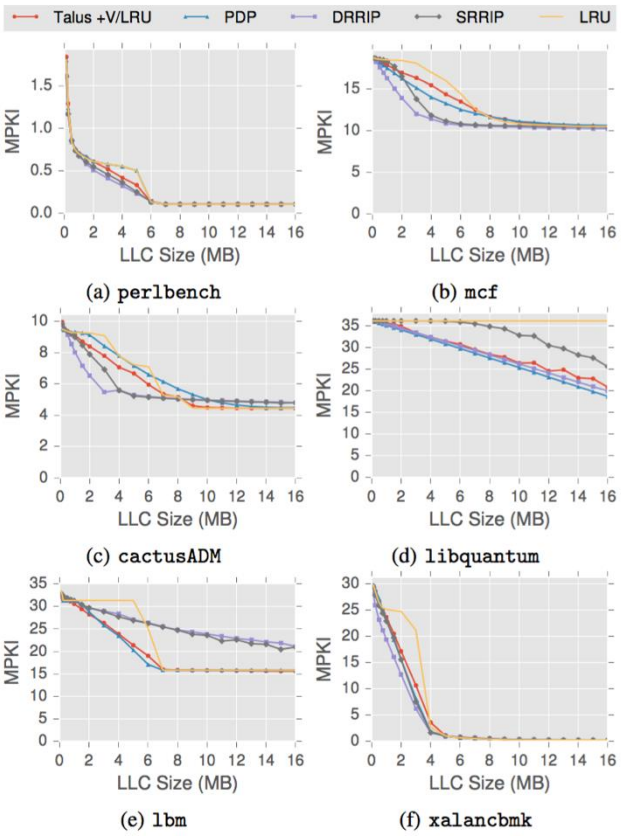

Figure 48: Misses per kilo-instruction (MPKI) of Talus (+V/LRU) and high-performance replacement policies on representative SPEC CPU2006 benchmarks from 128 KB to 16 MB. [20]

Look-ahead is a quadratic heuristic that approximates the NP-complete solution of the non-convex optimization problem. Weighted speedups over LRU are up to 41%/(of the geometric mean of) 12.5% for hill climbing on Talus+V/LRU, 34%/10.2% for Look-ahead on LRU, 39%/6.3% for TA-DRRIP, and 16%/3.8% for hill climbing on LRU. From these plots it can be seen that the only scheme that is competitive with Talus+V/LRU is Look-ahead, an expensive heuristic whose alternatives are complex. This shows that, by ensuring convexity, Talus makes partitioning simple and cheap [20].

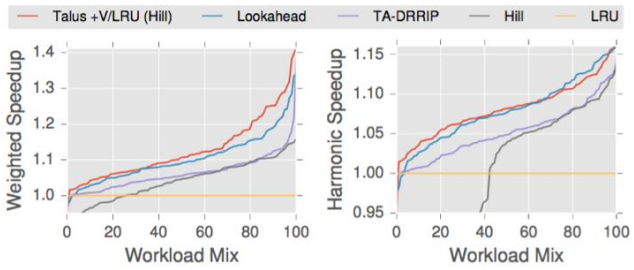

Figure 49: Weighted and harmonic speedup over LRU of Talus (+V/LRU), partitioned LRU (using both Look-ahead and hill climbing), and thread-aware DRRIP [20]

D. *Locality - Aware Data Replication in the Last - Level Cache*

In the future, multicore processors will process massive data with varying degrees of locality. Harnessing on-chip data locality to optimize the utilization of cache and network resources is of fundamental importance. To achieve this, in the paper, "Locality - Aware Data Replication in the Last - Level Cache" [22], Kurian et al. proposes a data replication protocol for the Last Level Cache (LLC). The goal of the protocol is to lower memory access latency and energy by replicating only high locality cache lines in the LLC slice of the requesting core, while keeping the off-chip miss rate low. The utility of data replication at the LLC can be best evaluated by measuring cache line reuse. Reuse at the LLC is defined as the number of accesses to a cache line by a core before the cache line is evicted or before a conflicting access by another core occurs.

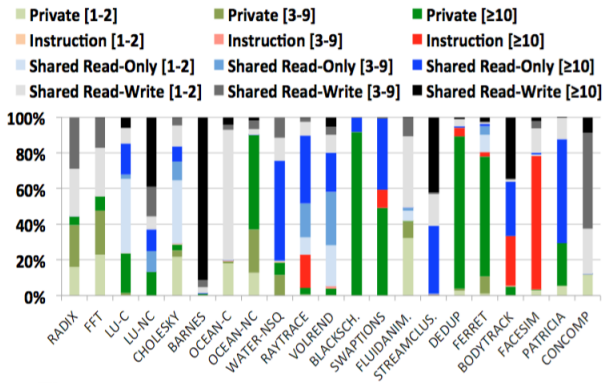

Figure 50: Distribution of instructions, private data, shared read-only data, and shared read-write data accesses to the LLC as a function of run-length [22]

Figure 50 shows the distribution of the number of accesses to a cache line as a function of run length. In Figure 50, it can be seen that 90% of the accesses for the application BARNES use shared data of a run length greater than or equal to 10. As the number of accesses to higher run length cache lines increase, it is beneficial to replicate the cache line in the requester's LLC slice. Nonetheless, if the replication is done when there are very few accesses to a higher run length line, such a policy would increase the LLC size without increasing performance. Therefore, the decision of replication should be based on the locality, instead of the type of data [22].

1) *Locality – Aware LLC Data Replication*

The most important components of data replication are:

- Choosing which cache lines to replicate.
- Determining where to place a replica.
- How to maintain coherence for replicas.

Figure 51 provides a better understanding on how the locality aware LLC technique works [22].

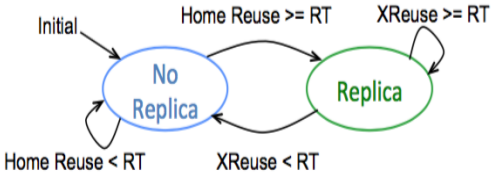

Figure 51: Mode transition at taught by Kurian et al. [22]

Figure 51 shows a transition graph. Initially, all cores with respect to all cache lines are initialized to the no-replica state, which means that no cache line replica is created at the LLC and all requests are serviced directly at the LLC home. The home reuse counter for each core tracks the number of accesses by that core to the corresponding cache line. If there are enough reuses, then a replica is created and that state goes to replica state. The number of reuses is determined by the replication threshold. If the home reuse counter reaches the replication threshold, the core is promoted to replica status and a replica is created in the LLC slice corresponding to the core. It is easy to see that if the replication threshold is high, it is harder to create a replica, and therefore, a lower number of replicas are created and vice versa. Once the replica is made the replica reuse counter keeps track of the number of accesses by the core to the replica location. If the replica reuse counter drops below the replication threshold, then the replica is evicted and it goes back to no replica state [22].

In Figure 52, the black data blocks are the data blocks with high reuse and local LLC replication is allowed.

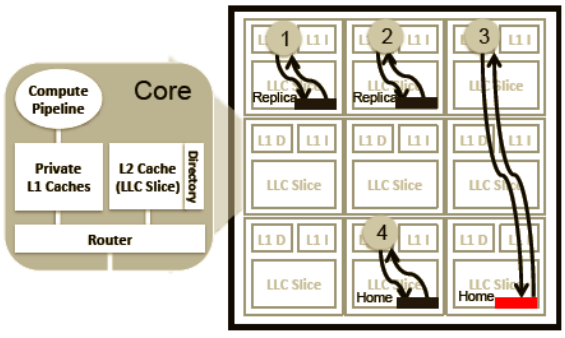

Figure 52: Mockup requests 1-4 showing the locality-aware LLC replication protocol [22]

These replica service requests are from instructions 1 and 2. The red data blocks indicate low reuse, and these blocks are not allowed to be replicated. The L1 cache miss requests from instruction 3 must access the LLC slice at the home core. The replication decision is based on the previous cache line reuse behavior.

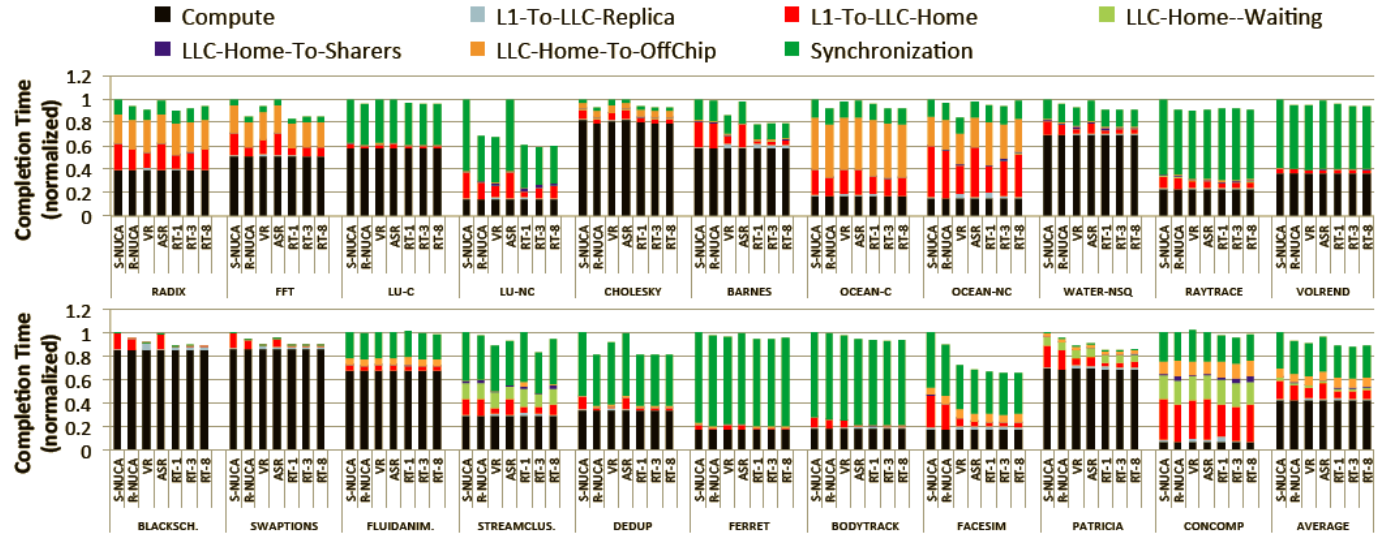

Figure 53: Completion Time breakdown for the LLC replication schemes evaluated. [22]

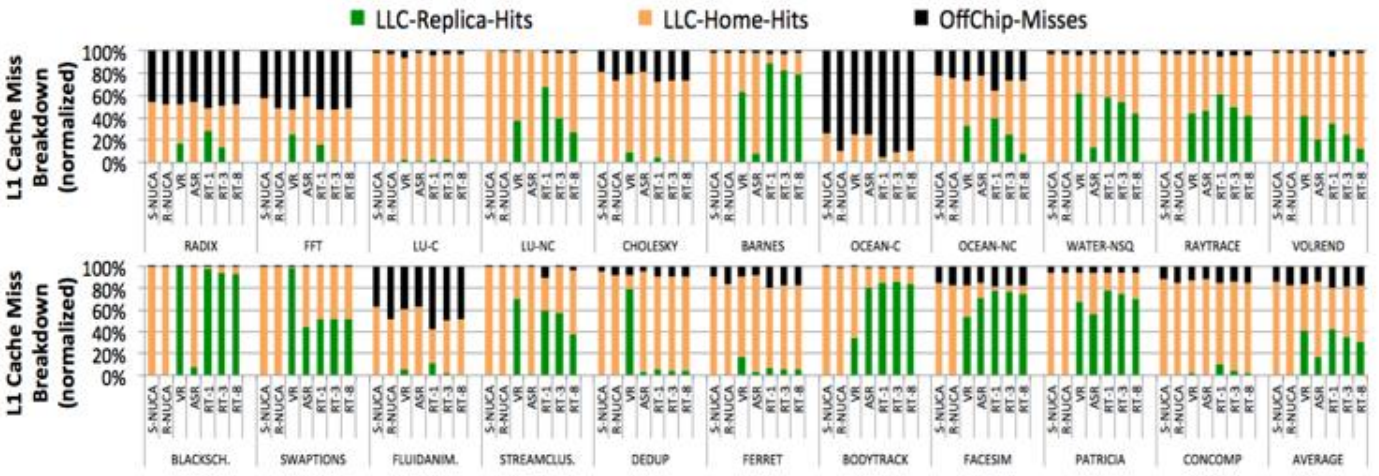

Figure 54: L1 Cache Miss Type breakdowns for the LLC [22]

2) Performance analysis

The results shown in this section are based on the performance models and metrics described in the paper "Locality – Aware Data Replication in the Last – Level Cache" [22]. Figure 53 shows the plot of completion times for the replication schemes evaluated. The RT-1, RT-3, RT-8 bars correspond to the locality aware scheme with replication thresholds of 1, 3 and 8 respectively. The completion time trends are based on the following factors:

- The type of data accessed at the LLC (instruction, private data, shared read- only data and shared read-write data).
- Reuse run-length at the LLC.
- Working set size of the benchmark.

Figure 54 shows how L1 cache misses are handled by the LLC. From Figure 54, it can be inferred that the locality aware protocol provides better performance than the other LLC data management schemes. It is trivial to balance the on-chip data locality and off-chip miss rate and overall, a replication threshold of 3 achieves the best trade-off. Overall, the locality-aware protocol has a 16%, 14%, 13% and 21% lower energy and a 4%, 9%, 6% and 13% lower completion time as compared to VR, ASR, R-NUCA and S-NUCA, respectively [22].

E. *Assisted Dead Region Management for Last Level Caches*

Last Level Caches (LLCs) bridge the performance miss-match between the processor and main memory. LLCs also reduce the amount of energy consumed per access. The most persistent problem caused in LLCs are dead blocks. Dead block are outdated blocks of data that stay in the cache in an unused state for a long time until they are evicted.

Already existing methods to predict dead blocks can be broadly classified into dynamic or static methods:

- The dynamic methods predict dead blocks based on the block access history.
- The static methods predict dead blocks by using control flow information to determine future access.

Because LLCs play an important role in reducing overall access time (as compared to access time to main memory), it is critical to manage LLCs effectively. The effective management of dead blocks within an LLC gives rise to significant efficiency improvements in the LLC.

1) RADAR (Runtime- Assisted DeAd Region) Management for Last Level Caches

RADAR, as proposed by Manivannan et al. [21] is an improved method to manage dead blocks within LLCs. RADAR is a hybrid static/dynamic dead block management technique that can accurately predict and evict dead blocks. RADAR is mainly based on a runtime system that collects static region-access information about the programming model and dynamic access information from the architecture. RADAR uses two orthogonal schemes: (1) look-ahead and (2) look-back schemes. Look–ahead schemes look into the near future to look for dead blocks. Look-back uses the per–region access history to predict how far into the future the next region access will occur [21].

2) RADAR Framework

After a task is completed, RADAR's runtime system predicts if the regions that have been accessed by a task are dead. If the region is predicted to be dead, the RADAR algorithm informs the LLC to demote these blocks to the LRU position. Figure 55 shows the overview of RADAR. RADAR is an interaction across three layers: (1) a programming model layer that conveys static data-dependency information by providing the regions that are used by tasks; (2) a runtime system layer responsible for detecting dead regions during execution and (3) an architecture layer responsible to provide dynamic feedback information about region access and demote the cache blocks that belong to dead regions.

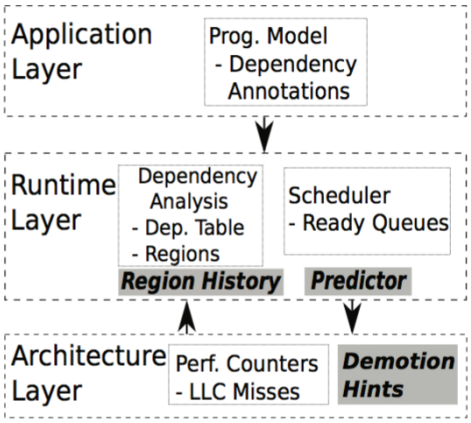

Figure 55: Overview of RADAR [21]

Three schemes can be implemented on top of the RADAR framework to accurately predict dead regions, namely: (1) a look-ahead scheme, (2) a look-back scheme and (3) combined schemes. In the look-ahead scheme, the runtime system dynamically constructs a data-flow graph of the tasks that are dispatched and are to be executed and those waiting for dependency resolution. This graph gives the system the ability to observe future tasks and their access region. If a region is not accessed by any of the tasks in the data-flow graph, then the region is deemed dead. Figure 56 shows a task dependency graph and Figure 57 shows how the runtime system tracks the state of regions using a dependency table and performs reference counting. In the example shown in Figure 56 and Figure 57, T0 is the first task and it does not have any dependencies. Subsequently, a new entry for A00 is allocated in the dependency table and the writer field is set to T0. When tasks T1, T2, T3, and T4 are generated, the runtime system detects a dependency with A00 and tasks T1, T2, T3 and T4 are added to the reader list. In addition, the new tasks are added to T0's successor list and the reference count of each new task is incremented by one as shown in Figure 57.

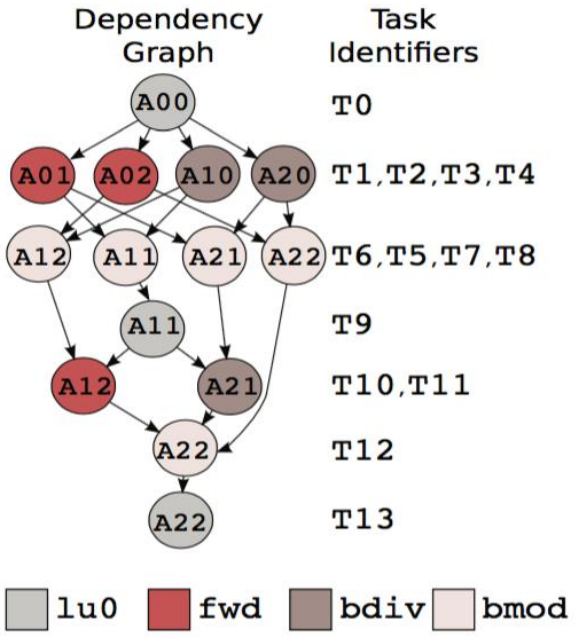

Figure 56: Sparse LU dependency graph [21]

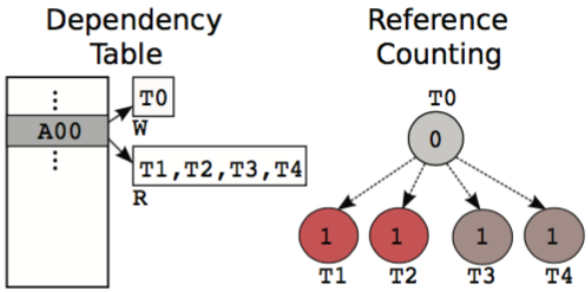

Figure 57: Dependency tracking mechanism [21]

When T0 completes execution, the writer field is cleared and the reference count of each of its successors is decremented by one. Once a successor's reference count becomes zero, it denotes that all its dependencies have been satisfied. The dependent tasks are queued in the ready queue for execution. Finally, when tasks T1, T2, T3, and T4 finish execution, they are removed from the readers list of A00.

It should be noted that the look-ahead scheme has two limitations. The first limitation is that the scheme does not provide temporal information. In other words, the scheme does not provide the information of when the next access to a region will occur. The second limitation is: if a reuse is not detected, it does not mean that the block is dead. Instead, this could happen if the master thread that generates tasks has not populated the look-ahead window fast enough.

The look-back scheme works on the observation that accesses tend to have a repetitive pattern. The scheme predicts future accesses using the same methodology as branch predictors: (1) classify the current region access as

hit/miss (analogous to taken/not taken) and (2) predict whether the next access to the region will be a hit/miss based on previous accesses to the region. Figure 58 shows the working of the look-back scheme on the same example used to show the working of the look-ahead scheme.

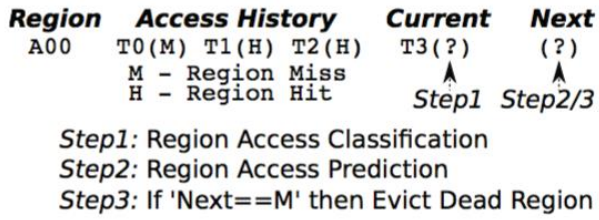

Figure 58: Overview of Look back Scheme [21]

The two orthogonal schemes could be combined to achieve better performance for the identification of unused blocks. There are two approaches for this combination: (1) the Aggressively combined look-ahead and look-back Scheme (AS) and the Conservatively combined look-ahead and look-back Scheme (CS). We define the set, LA (Look-Ahead) as the set of all the dead regions classified by the look-ahead scheme and the set LB (Look-Back) as the set of all dead regions classified by the look-back scheme. For CS, the set of regions of dead blocks is the combined set of regions that belongs to LA ∩ LB, i.e., the intersecting set of LA and LB. For AS, the dead region set is the combined set of dead block regions that belongs to LA ∪ LB, i.e., the union of the sets LA and LB [21].

3) Performance analysis

The different schemes used by RADAR to detect dead regions were evaluated by Manivannan et al. [21]. Figure 59 and Figure 60 show the LLC misses for different RADAR policies normalized to the LRU baseline scheme and the execution time for different RADAR policies normalized to the LRU baseline scheme respectively. The look-ahead scheme reduces LLC misses for all the applications. It can be seen that on average, LA reduces misses in the LLC by 23%. From the results, it can be seen that the look-back scheme outperforms the LRU scheme for all applications. Nonetheless, the look-back scheme does not perform as well as the look-ahead scheme. The aggressive combined scheme (AS) outperforms all other schemes. The average reduction in LLC misses for all applications is more than 26% when compared to the LRU baseline scheme. The conservative combined scheme (CS) outperforms the look-back scheme using the future view of look-ahead. Nonetheless, CS is outperformed by the AS [21].

RADAR is then compared to other state-of-the-art dead block predictors, like the Count-based Dead-Block Predictor (CD-BP), Sampling-based Dead Block Predictor (SDBP) and Signature-based Hit Predictor (SHiP). The focus of the analysis is to measure miss rates, which should decrease as dead blocks are replaced in order to improve LLC efficiency. The results shown are based on the evaluation methodology and metrics shown in the paper "RADAR: Runtime-Assisted Dead Region Management for Last-Level Caches" [21]. The four different schemes used by RADAR (e.g., RADAR, CD-BP, SDBP, SHiP) are compared with the baseline LRU scheme.

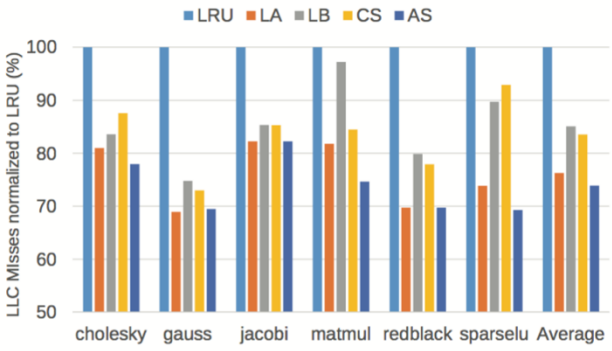

Figure 59: LLC misses for different RADAR policies normalized to the LRU baseline [21]

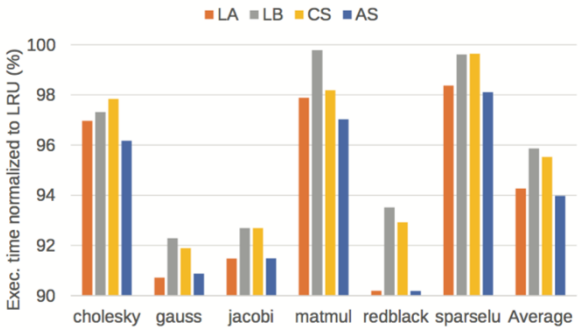

Figure 60 Execution time for different RADAR policies normalized to the LRU baseline [21]

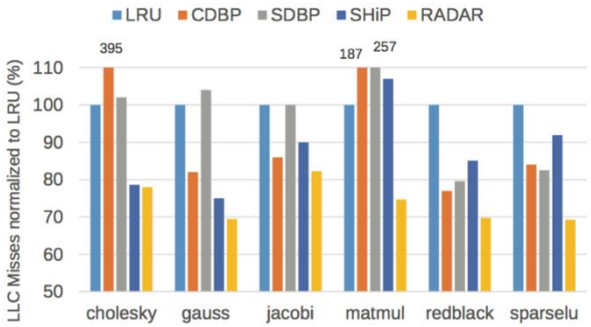

Figure 61: LLC misses for RADAR and state-of-the-art dynamic dead block prediction techniques normalized to the LRU baseline [21]

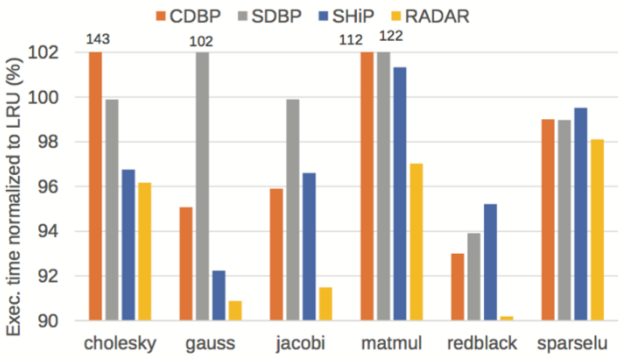

Figure 62: Execution time for RADAR and state-of-the-art dynamic dead block prediction techniques normalized to the LRU baseline [21]

Lastly, Figure 61 and Figure 62 show the comparison between RADAR and the other abovementioned dynamic dead block predictors (CDBP, SDBP and SHiP). Figure 61 shows the LLC misses for RADAR and state-of-the-art dynamic dead block prediction techniques normalized to the LRU baseline scheme. Figure 62 shows the execution time for RADAR versus the other state-of-the-art dynamic dead block prediction techniques normalized to the LRU baseline scheme. While the state-of-the-art techniques reduce the number of LLC misses, RADAR is on average at least 10% better. The data shows that RADAR is more effective than existing techniques at managing dead blocks. From the results it can be seen that RADAR performs better than other dead block LLC management techniques overall [21].

4 Conclusion and Future Work

As discussed above in this paper, many encouraging improvements in cache memory architecture performance have been identified and fully tested. For the most part, cache memory system performance testing has been done via simulations, without the benefit of seeing such improvements being truly implemented in real-world high-performance computer architectures. As discussed in the case study below, while it can take many years to get cache memory architecture research out of the lab and implemented into silicon, once implemented, the benefits of such research can and do make a dramatic impact.

Indeed, as shown for the Intel Xeon Haswell multi-core architecture (see "A Case Study – From Concept to Reality"), while the road from research to reality is long, the benefits are great. While strides in cache memory architectures continue, multi-core processor performance still outflanks cache memory performance, leaving seemingly limitless areas of research to be explored in the future. Compounding the gap between cache memory performance and multicore processor performance is the fact that individual cache memories associated with each processor core must at some point be mapped to a single shared memory, thus causing a bandwidth bottleneck. With zetta-flop (e.g., 10^{21}) performance computing systems projected by 2020, a continuum of new breakthroughs for cache memory designs are required [25].

Just one area ripe for additional research appears to be in the area of optical cache memories. Optical cache memories offer the ability for the memory speed to keep up with the ever increasing data bandwidth requirements presented by future multi-core processor systems. Figure 64 shows an envisioned multicore to optical cache memory architecture using an optical-to-digital interface. Encouragingly, simulation results have shown a 40% improvement in cache memory access speed at a clock rate of 16 GHz [25].

A Case Study - From Concept to Reality [24]

As far back as the mid 2000s, researchers envisioned techniques to significantly improved multi-core processor system architectures by employing breakthroughs in a research area called "Quality of Service (QoS)". QoS relates to reducing shared cache memory contention while co-running applications. Specifically, based on simulations, a few researchers identified two (2) new QoS prospective research areas to improve multi-core processor system architectures: (1) Cache Monitoring Technology (CMT), and (2) Cache Allocation Technology (CAT).

During the research phase, CMT technology promised improvements by intelligently monitoring how shared cache memory was actually used by a given workload. Separately, CAT technology offered enforcement of how much a given workload was allowed to use shared cache resources. Combined, CMT and CAT was envisioned to provide solid improvements.

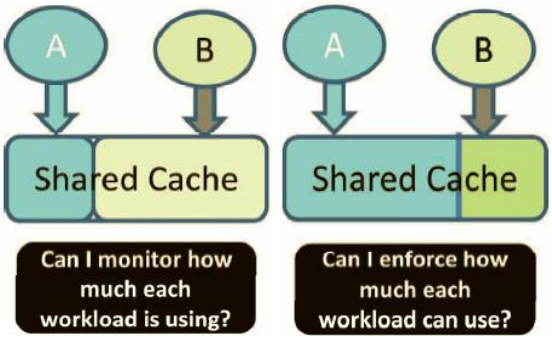

Figure 63 - Overview of CMT (Left) and CAT (Right) [24]

While CMT and CAT technology was seen by researchers to be promising, it took 10 years before such research was able to be applied in practice [24]. On June 4, 2013, Intel introduced the Xeon "Haswell" 4th generation processor employing both CMT and CAT technologies [24][29].

The evaluation of the CMT and CAT technologies incorporated in the Intel Xeon Haswell chip was proven to provide improvements as high as 450% without a single case of inferior performance over a wide spectrum of tested workloads [24].

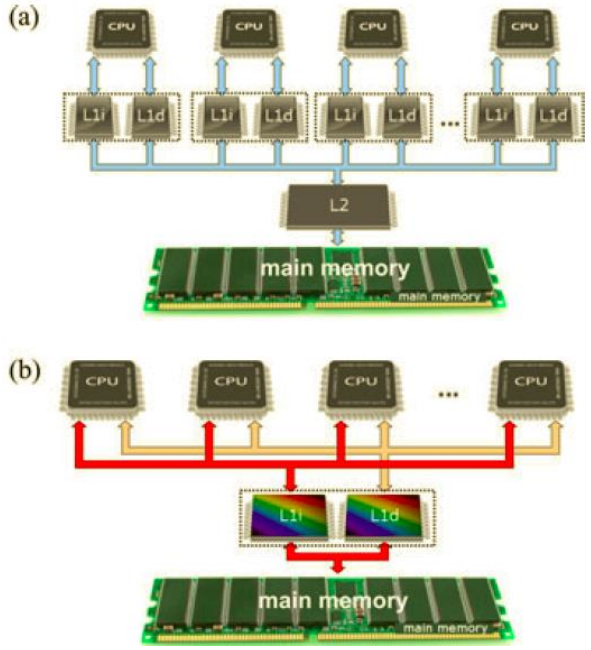

Figure 64 – Overview of (a) present day cache memory architecture versus (b) optical cache memory architecture [25]

Researchers will no doubt continue to rise to the challenge of bridging the gap between processor system performance and cache memory system performance as they have for nearly 40 years.

VI. BIBLIOGRAPHY

[1] Intel Corporation, "Photograph of Intel Xeon processor 7500 series die showing cache memories," 31 March 2010. [Online]. Available: https://phys.org/news/2010-03-intel-xeon-processor-series.html. [Accessed 20 February 2017].

[2] IBM Products Division, "System/370 model 168 theory of operation/diagrams manual (volume 1)," Poughkeepsie, NY, 1976.

[3] J. R. Goodman, "USING CACHE MEMORY TO REDUCE PROCESSOR-MEMORY TRAFFIC".

[4] H. Bajwa and X. Chen, "Low-Power High-Performance and Dynamically Configured Multi-Port Cache Memory Architecture".

[5] K. Tanaka, "Cache Memory Architecture for Leakage Energy Reduction," in *International Workshop on Innovative Architecture for Future generation Processors and Systems 2007*, 2007.

[6] H. Hassan, G. Pekhimenko, N. Vijaykumar, V. Seshadri, D. Lee, O. Ergin and O. Mutlu, "ChargeCache: Reducing DRAM Latency by Exploiting Row Access Locality," *IEEE*, 2016.

[7] C. K. Tang, "Cache system design in the tightly coupled multIproeessor system," *AFIPS Proceedings*, vol. 45, pp. 749-753, 1976.

[8] L. M. Censier and P. Feautrier, "A new solution to coherence problems in multicache systems," *IEEE Transactions on Computers*, Vols. C-27. No. 10, pp. 1112-1118, 1978.

[9] G. S. Rao, "Performance Analysis of Cache Memories," *Journal of the ACM*, vol. 25, pp. 378-395, 1978.

[10] R. L. Norton and J. L. Abraham, "Using write back cache to improve performance of multiuser multiprocessor," in *Int. Conf. on Par. Proc., IEEE cat. no. 82cH1794-7*, 1982.

[11] M. C. Easton and R. Fagin, "Cold-start versus warm-start miss ratios," *CACM*, vol. 21. No. 10, pp. 866-872, 1978.

[12] C. Bell, J. Judge and J. McNamara, "Computer engineering: a DEC view of hardware system design," *Digital Press*, 1978.

[13] E. Akanksha, H. Shanvas and V. Nallusamy, "Modern CPU's Memory Architecture - A Programmer's Outlook," *Modern Education and Computer Science Press*, no. Published Online April 2012 in MECS, 2012.

[14] S. H. Pugsley, Z. Chishti, C. Wilkerson, P.-f. Chuang, R. L. Scott, A. Jaleel, S.-L. Lu, K. Chow and R. Balasubramonian, "Sandbox Prefetching: Safe Run-Time Evaluation of Aggressive Prefetchers," *IEEE*, pp. 1-12, 2014.

[15] N. Beckmann, P.-A. Tsai and D. Sanchez, "Scaling Distributed Cache Hierarchies through Computation and Data Co-Scheduling," p. IEEE, 2015.

[16] V. Jimenez, F. P.O'Conell and F. Cazorla, "Increasing Multicore System Efficiency through Intelligent Bandwidth Shifting," *IEEE*, pp. 39-50, 2015.

[17] Z. Wang, D. A. Jimenez, C. Xu, G. Sun and Y. Xie, "Adaptive Placement and Migration Policy for an STT-RAM-Based Hybrid Cache," *IEEE*, 2014.

[18] H. Farbeh and S. G. Miremadi, "PSP-Cache: A Low-Cost Fault-Tolerant Cache Memory Architecture," *EDAA*, 2014.

[19] A. Awad and Y. Solihin, "STM : Cloning the Spatial and Temporal Memory Access Behavior," *IEEE*, 2014 .

[20] N. Beckmann and D. Sanchez, "Talus: A Simple Way to Remove Cliffs in Cache Performance," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, San Francisco, 2015.

[21] M. Manivannan, V. Papaefstathiou, M. Pericàs and P. Stenström, "RADAR: Runtime-Assisted Dead Region Management for Last-Level Caches," *IEEE*, 2016.

[22] G. Kurian, S. Devadas and O. Khan, "Locality-Aware Data Replication in the Last-Level Cache," *IEEE*, 2014.

[23] S. Khan, A. R. Alameldeen, C. Wilkerson, O. Mutlu and D. A. Jimenez, "Improving Cache Performance Using Read-Write Partitioning," *IEEE*, 2014.

[24] A. Herdrich, E. Verplanke, P. Autee, R. Illikkal, C. Gianos, R. Singhal and R. Iyer, "Cache QoS: From Concept to Reality in the Intel Xeon Processor E5-2600 v3 Product Family," *IEEE*, 2016.

[25] P. Maniotis, D. Fitsios, G. Kanellos and N. Pleros, "Optical Buffering for Chip Multiprocessors: A 16GHz Optical Cache Memory Architecture," *Journal of lightware technology*, 2013.

[26] N. P. Jouppi, "Improving Direct Mapped Cache Performance by the Addition of a Small Fully Associative Cache and Prefetch Buffers," *Digital Corporation - Western Research Laboratory*, Vols. WRL Technical Note TN-14, pp. 1-36, 1990.

[27] "Cache Prefetching citing N. Jouppi," [Online]. Available: https://en.wikipedia.org/wiki/Cache_prefetching. [Accessed 16 Feb. 2017].

[28] K. M. Qureshi and Y. N. Patt, "Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches," in *The 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, Orlando, 2006.

[29] P. Moorhead, "Intel's Newest Core Processors: All About Graphics And Low Power," *Forbes* , 4 June 2013.