CASH: C̲riticality-A̲ware S̲plit H̲ybrid L1 Data Cache

Shruthi Karunakar
shruthi.k.19@iitdh.ac.in
Indian Institute of Technology
Dharwad
India

Meenakshi Atkade
meenakshiatkade147@gmail.com
Indian Institute of Technology
Dharwad
India

Akash Poptani
200020005@iitdh.ac.in
Indian Institute of Technology
Dharwad
India

Rajshekar Kalayappan
rajshekar.k@iitdh.ac.in
Indian Institute of Technology
Dharwad
India

Sandeep Chandran
sandeepchandran@iitpkd.ac.in
Indian Institute of Technology
Palakkad
India

ABSTRACT

Computer architects continue to explore newer ways to provide the abstraction of a large but fast memory to the processor. This work proposes a memory system that achieves this abstraction using a hybrid cache – a combination of an SRAM array and a Spin-Transfer Torque Magnetic RAM (STTRAM) array, at the highest level (L1) of the memory hierarchy. We overcome the issue of longer access latency of STTRAM arrays by placing those cache lines which are likely to be accessed by a critical load instruction (or delay-sensitive loads) into the SRAM array. Our characterization of the CPU SPEC2017 benchmarks shows that most load instructions are tolerant to access latency of STTRAM array, which makes a small (but fast) SRAM array amenable. The higher densities and lower leakage power of the STTRAM array also make it amenable to provision for larger capacity without significant area overhead. Through extensive simulations of SPEC2017 benchmarks, we show that a combination of a small but fast SRAM array, and a large STTRAM array yields an average performance gain of up to 6.1% when compared to a baseline system that uses only an SRAM-array-based cache of similar area. This performance improvement comes at a cost of a 1.7% increase in the energy consumption of the private caches.

CCS CONCEPTS

• **Computer systems organization → Architectures**; • **Hardware → Emerging architectures**; **Spintronics and magnetic technologies**; *Very large scale integration design.*

KEYWORDS

Microarchitecture, Cache design, Hybrid caches, STTRAM, Instruction Criticality

ACM Reference Format:
Shruthi Karunakar, Meenakshi Atkade, Akash Poptani, Rajshekar Kalayappan, and Sandeep Chandran. 2024. CASH: C̲riticality-A̲ware S̲plit H̲ybrid L1 Data Cache. In *Great Lakes Symposium on VLSI 2024 (GLSVLSI '24), June 12–14, 2024, Clearwater, FL, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3649476.3658716

GLSVLSI '24, June 12–14, 2024, Clearwater, FL, USA
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0605-9/24/06
https://doi.org/10.1145/3649476.3658716

1 INTRODUCTION

The gap in processor computation speeds and memory access times, better known as the *Memory Wall*, continues to exist even after several decades of research in this direction. Several advances in processor micro-architecture, such as caching, prefetching, memory pipelining, and way/bandwidth partitioning, try to bridge this gap by providing an abstraction of a large yet fast memory store to the processor core.

The introduction of Non-Volatile Memories (NVMs), such as Spin Transfer Torque Random Access Memory (STTRAM), has opened new opportunities to architect a memory hierarchy that supports this abstraction because the higher density and lower leakage power of STTRAM help increase the capacity of caches. A straightforward replacement of SRAM arrays with STTRAM arrays is not feasible because STTRAM cells suffer from two problems (elaborated in Section 2): (i) reads and writes take longer as compared to traditional Static RAM (SRAM) cells (with writes taking longer than reads), and (ii) writes consume more dynamic energy as compared to SRAM cells. A prominent approach to overcome these limitations is to use STTRAMs at lower levels of the memory hierarchy where the accesses are fewer and access latencies are typically higher. This is further extended by combining the high-capacity STTRAM cache banks with SRAM cache banks to create hybrid caches. The SRAM bank helps reduce the dynamic energy consumed by holding write-intensive cache lines. A state-of-the-art technique under this approach also avoids writing dead (brought to the cache but never used) cache lines into the STTRAM bank [27]. An orthogonal approach is to employ a reduced retention STTRAM caches which require complex refresh mechanisms, but provide for lower access latency and energy [24].

In this work, we propose a novel hybrid cache, called Criticality Aware Split Hybrid cache (CASH), that makes it amenable for use at the L1 level. Our proposed hybrid cache consists of two partitions: (i) P_0 - partition implemented using SRAM array, and (ii) P_1 - partition implemented using STTRAM array. The P_0 partition will hold only

This work was partially supported by the Science and Engineering Research Board [CRG/2020/001001], Department of Science and Technology, India.

those cache lines that are likely to be accessed by critical `load` instructions – delaying these `load` instructions will increase the execution time of the application, and the P_1 partition will hold cache lines that are likely to be accessed by delay-tolerant `load` instructions.

Our characterization of the SPEC CPU2017 (elaborated in Section 3) shows that the number of `load` instructions that have a *global slack* lesser than the access latency of a STTRAM array is small. The global slack of an instruction is the number of cycles it can be delayed without extending the completion time of the application [9]. This observation makes it amenable to use a P_0 partition that is smaller than a traditional SRAM-based L1 cache, which reduces the access latency of the SRAM portion. This improves the application's performance by servicing critical `load` instructions faster. The higher density and lower leakage power of STTRAMs help implement a P_1 partition of a similar capacity as a traditional SRAM-only cache in a lesser area. Therefore, our hybrid L1 cache offers a higher cache capacity while occupying a similar area.

Through extensive experimentation on the CPU2017 benchmarks, we show that our proposed hybrid L1 cache offers an average performance improvement of 6.1% with a modest increase in energy consumption of 1.7% as compared to an SRAM-only cache occupying a similar area, thereby achieving the abstraction of a fast (for critical `loads`) yet large (for delay-tolerant `loads`) memory store.

2 BACKGROUND AND RELATED WORK

2.1 STTRAM based on-chip caches

STTRAM is a promising memory technology for building on-chip caches [1, 5] because it offers two primary advantages over conventional SRAM: (i) higher density ($> 3x$) and (ii) lower leakage power consumption ($< 0.05x$). These advantages enable the incorporation of high-capacity on-chip caches that reduce the number of misses, thereby increasing the overall performance. Several academic proposals [13, 25, 27], prototypes, and commercial offerings have since used STTRAM-based on-chip caches at all levels of the cache hierarchy from the L1 level [16, 17, 20] to the Last Level Cache (LLC) [15, 26, 27]. However, some challenges arise as the technology nodes scale to smaller sizes (less than $32nm$).

2.1.1 The issue with read operations. With scaling to sizes smaller than 32nm, the current required to read an STTRAM cell has stayed steady. The increasing effect of process variation on both the magnetic tunnel junctions (MTJs) and the CMOS transistors degrades the sensing margin, requiring a large current for a correct and quick reading of the cell contents. However, using a large read current may lead to accidental modification of the cell contents because with scaling, the MTJ critical switching current has reduced. This is termed a *Read Disturbance Error* (RDE) [7, 12, 14, 17, 18, 26, 28]. This has led researchers to believe that readability, rather than writability, forms the ultimate bottleneck in STTRAM-based systems [14, 28].

There are traditionally two techniques to handle this situation. The first is termed *Low Current Long Latency* (LCLL) and uses a low current over a long sensing period to perform the read operation. Since a low current is employed, RDEs do not occur. However, the read latency increases. The second traditional technique is termed *High Current Restore Required* (HCRR). HCRR uses a high current to perform the read operation that could cause an RDE. But, the cell's content is then immediately written back or *restored*, thereby undoing the effect of any RDE. Though the high current allows the read operation to complete faster than in the case with a low read current, the subsequent restore operation delays subsequent reads. In many applications, the overall performance with a high read current is actually lower than that with a low read current. Additionally, the read operation is quite expensive in terms of energy due to the restore operation [26]. In summary, regardless of the technique used, the cell read latency in an STTRAM cache is usually much higher than that in an SRAM cache [6].

Selective restoration [18, 26] may be employed to reduce the overhead of HCRR. The system may switch between HCRR during periods of low activity and LCLL during periods of high activity [12]. Differential sensing may be employed to reduce the read current, thereby reducing the chance of an RDE. Since this approach requires both the datum and its complement to be stored, it halves the capacity of the cache and doubles the power consumed [14, 17]. Our proposal is an architecture-level approach that is orthogonal to all of the prior work. We propose a hybrid SRAM-STTRAM cache, with the STTRAM partition operating in an LCLL fashion, and being used to service read requests that are delay-tolerant from an application's point of view.

Since the STTRAM cell occupies a lesser area than the SRAM cell, the size of the interconnects within the cache is smaller. Therefore, signals have to travel shorter distances. This offsets the high cell-read latency in STTRAM caches to some extent in large-capacity caches. However, the cell-read latency dominates in the case of small-capacity caches [7] such as those used at the L1.

2.1.2 The issue with write operations. STTRAM caches have large write latencies and write energies because a large amount of current has to be applied for a significant duration to perform a write operation (the magnetic direction of the free layer of the MTJ needs to be set/reset to a particular direction).

A circuit-level approach to reduce the write energy is to use MTJs with lower thermal stability to implement the cache. Although this enables cells to be written to using a lower current [24], these cells have lower retention times. Therefore, such caches require refresh mechanisms to maintain correctness (similar to DRAMs), which can be expensive [15].

There are two architecture-level approaches to reduce write energy. The first approach is to reduce the number of write operations. This is typically done by identifying the lines with low reuse and not placing them in the cache. Another technique is to use an auxiliary buffer that combines temporally local writes to the same line to a single write [2, 15]. The second approach is to employ hybrid caches [8, 27]. The state-of-the-art hybrid cache APM [27] preferentially places write-intensive lines in the SRAM partition and does not cache lines that have no reuse (dead lines). Our work adopts the spirit behind the heuristics proposed in APM but customizes the design of deadness and write-intensity predictors to make it suitable for use at L1 (instead of LLC as proposed). We evaluate our work against the original APM for reference.

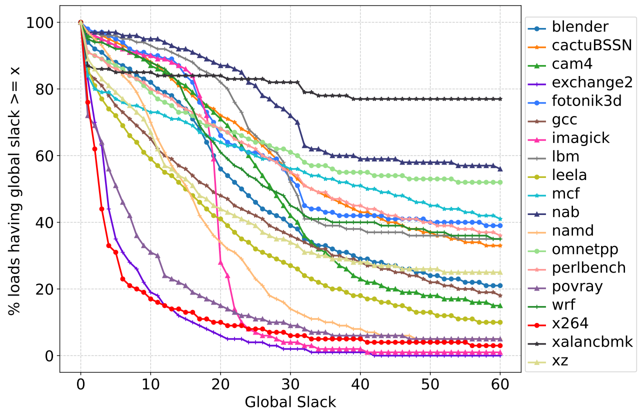

Figure 1: Criticality distribution of load instructions

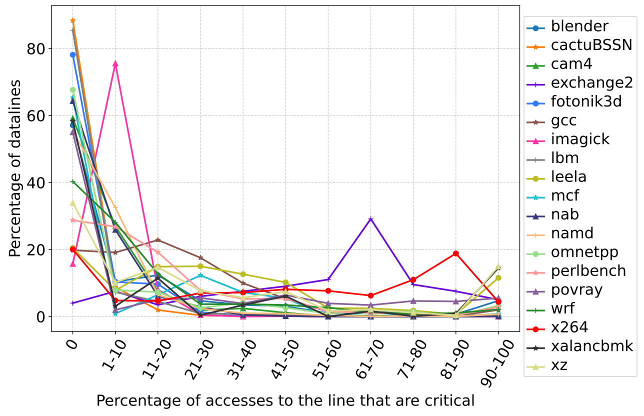

Figure 2: Criticality distribution of data lines

2.2 Instruction Criticality and the Fields Model

Delays in execution of some instructions on an out-of-order processor may affect the performance of an application more than other instructions. The seminal work on identifying such critical instructions models the execution of an application as a graph [10]. Here, each dynamic instruction is represented by three nodes that correspond to its Dispatch (D), Execute (E), and Commit (C). Edges between nodes represent constraints (both hardware and software) in the latter's schedules. An edge $i \xrightarrow{w} j$ indicates that node j can be scheduled only w cycles after i. The As-Soon-As-Possible (ASAP) schedule and the As-Late-As-Possible (ALAP) schedule of all the nodes are then calculated. The application's deadline, or the maximum duration of the ASAP schedule is used to compute the ALAP schedule. The difference of these schedules at each node indicates the global slack of that node [9]. This slack of a node indicates the duration for which it can be delayed without affecting overall execution time.

Several works have used instruction criticality to optimize various micro-architectural structures such as instruction issue logic and cache management [4, 19]. In this work, we employ an implementation of the aforementioned model to identify critical instructions [19]. All the `load` instructions tagged as non-critical by this implementation are serviced from the P_1 partition (LCLL STTRAM bank).

3 CRITICALITY CHARACTERIZATION

Figure 1 shows the percentage of `load` instructions in each benchmark that have a global slack greater than the slack shown along the x-axis. We notice that over 60% of load instructions in 16 out of 19 SPEC2017 benchmarks have a global slack of at least 8 cycles (which is the typical access latency of a $32KB$ STTRAM based cache). This large fraction of delay-tolerant loads presents us with an opportunity for employing an STTRAM-based L1 cache. These results are in line with the observations made in prior work [4]. Therefore, if the small fraction of critical load instructions are accelerated, and the latency of the remaining load instructions is kept within reasonable limits, then we stand to achieve large performance gains.

For the next study, we first determine the criticality (global slack < 8) of each load instruction that accessed a particular cache line. Then we categorize the cache lines into bins based on the fraction of critical accesses to them. We plot the histogram of the percentage of cache lines in each bin for all the 19 SPEC2017 benchmarks. Figure 2 shows this histogram. For example, the bin $1-10\%$ critical shows the number of L1D cache lines which were accessed by non-critical load instructions over 90% of the time.

We observe that across all benchmarks, only $\leq 20\%$ of L1D cache lines were accessed often (over 80%) by load instructions that were critical. This motivates the case for having a small but fast SRAM partition. We also observe that in 11 out of 19 benchmarks, over 50% of the cache lines accessed were solely accessed by non-critical loads (bin 0). This motivates the case for a relatively larger but a slower STTRAM partition.

4 SPLIT HYBRID CACHE ARCHITECTURE

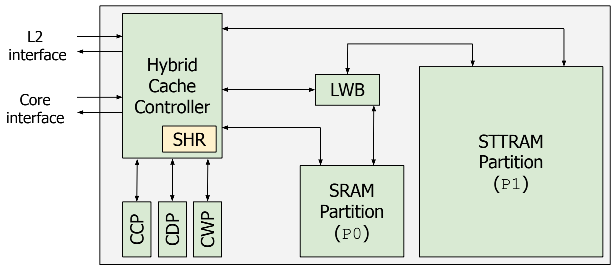

Figure 3: Design of the proposed Criticality-Aware Split Hybrid Cache (CASH) at the L1 level

Cache structure: Figure 3 shows the high-level architecture of our proposed Criticality-Aware Split Hybrid Cache (CASH). It has two partitions: (i) low latency, low capacity SRAM partition, called P_0, and (ii) longer latency, high capacity STTRAM partition, called P_1. The SRAM partition (P_0) is smaller ($16KB$) as compared to that of the traditional cache ($32KB$). This helps reduce the access latency of the SRAM partition to 3 cycles ($3c$) as compared to the base case of $4c$. The STTRAM partition (P_1) is of the same capacity as the traditional cache ($32KB$). Since the density of STTRAMs is over twice that of SRAM, the P_1 partition occupies less than half the area of the SRAM arrays of the traditional cache [13, 25, 27]. Thus, there is enough space to accommodate the two partitions and the additional logic and structures for managing the two partitions without incurring any area overhead as compared to a traditional SRAM-based L1 cache. The CASH controller has a Status Holding Register (SHR) that can hold a maximum of sixteen entries. Each SHR entry corresponds to one ongoing pipeline request. Each entry contains the following fields: (i) the type of request (read/write/prefetch), (ii) word address, (iii) word value (relevant only for stores), (iv) P_0 status (waiting for port/searching/miss), and (v) P_1 status. The controller also has a Line Write Buffer (LWB). The LWB contains twenty entries, one entry for each line waiting to be written to any one of the L1 partitions. The line could have originated from the L2, or may have originated from P_1 to be written to P_0 (termed a "line migration", which is discussed ahead). Each LWB entry contains the following fields: (i) address, (ii) line contents, (iii) destination partition (P_0/P_1), and (iv) status (waiting for port/writing).

The P_0 and P_1 partitions are maintained in a strictly exclusive manner. This removes the need for any coherence mechanism between the two partitions, thereby simplifying the control logic. Such exclusivity of cache lines also increases the effective capacity of the proposed cache as compared to the traditional cache ($48KB$ as compared to $32KB$). Further, we assume that the private L1 cache follows the Write-Through Write-No-Allocate policy, while the private L2 cache follows the Write-Back policy.

Cache access: The pipeline waits until there is space in the SHR and the LWB before it issues a request to the L1 cache. Upon receiving a request from the pipeline, the CASH controller initiates a lookup in both the partitions, as well as the LWB. If the requested data is found in P_0 or the LWB, the lookup in P_1 is aborted. If the requested data is not present in P_0, then a request is immediately sent to the L2 cache (even before the completion of the lookup in P_1). Additionally, if it is read request, then the L1 prefetcher is exercised. If the requested data is found in P_1, then a request to abort the search is sent to L2. Since modern cache operations are usually pipelined in a lock-up free manner, such an abort mechanism is easily realised [11].

Cacheline placement, eviction, and migration: As motivated earlier, the idea is to place lines that serve critical loads in the fast P_0 partition, and those that serve non-critical loads in the slower P_1 partition, thereby increasing overall performance. However, care must be taken to reduce the number of write operations in P_1 because of its high dynamic write energy. The chosen L1 policies must also not unduly increase the number of L2 accesses (including both demand and prefetch) as this will again increase the energy consumption since the L2 is a larger cache.

When a line $\mathcal{L}$ arrives from the L2 level, the three predictors – Cacheline Criticality Predictor (CCP), Cacheline Deadness Predictor (CDP), and Cacheline Write-Intensity Predictor (CWP) – are accessed to predict the criticality, deadness, and write-intensity of $\mathcal{L}$ respectively. These predictions are then used according to the procedure `PlaceLine()` in Algorithm 1 to place $\mathcal{L}$ in the L1 cache.

Whenever a line is evicted from either partition, it is discarded as the L1 level follows a Write-Through policy. We also adopt a line migration strategy as given by `MigrateOnHit_`P_1`_to_`P_0 in Algorithm 1. This is expected to help the cache contents adapt to changing program phases.

```
def PlaceLine($\mathcal{L}$):
    if $\mathcal{L}$ is dead then
        bypass the L1 cache;
    else
        if $\mathcal{L}$ is critical then
            place $\mathcal{L}$ in $P_0$;
        else
            if $\mathcal{L}$ is write-intensive then
                bypass the L1 cache;
            else
                place $\mathcal{L}$ in $P_1$;
            end
        end
    end
def MigrateOnHit_$P_1$_to_$P_0$():
    if access to line $\mathcal{L}_1^h$ hits in $P_1$
      and $\mathcal{L}_1^h$ is not dead and
      ((access is a write and $\mathcal{L}_1^h$ is write-intensive)
        or ($\mathcal{L}_1^h$ is critical)) then
        migrate $\mathcal{L}_1^h$ to $P_0$;
    end
```
Algorithm 1: Placement and migration policies in CASH

Whenever a line needs to be placed in any of the partitions, it is first staged in the LWB until a write port in the corresponding partition is available. If there is no place in the LWB, then the placement is not attempted.

Design of the predictors: The CCP's design is inspired by the criticality predictor employed by Nori et al. [19]. Whenever an instruction commits, an entry is made in a post-commit buffer. Each entry contains information to capture the three nodes – Dispatch, Execute, and Commit – as described in Section 2.2. When the buffer is half full, the contents are processed to determine the global slack of (the E nodes of) each instruction. For each load instruction, if the global slack is less than the threshold (latency of the P_1 partition), then the load is deemed critical, else, it is not. A 2048-entry predictor table, that is indexed by the line address of the line being accessed by the load instruction, is trained accordingly. We employ the counter model as prescribed by Fields et al. [10]. Once trained, this half of the post-commit buffer is discarded. A sampling-based approach is employed, and so if at commit time the buffer is found to be full, no entry is made. The committing of instructions never stalls. The area of the CCP is $5KB$. It is worth noting that the components of the CCP can be used to guide a variety of policies in addition to those in the hybrid cache controller such as instruction steering, scheduling, and prefetching [4, 19].

The CDP and the CWP are designed as done in APM [27], but with smaller dimensions. A pattern simulator unit (32-set, 6-way read, 2-way write) is employed to dynamically study the access patterns that the different lines in the cache are going through. Accordingly, dedicated predictor tables (1024 entries; 2-bit saturating counters) for deadness and write-intensity are trained. The area of the CDP and the CWP together is $1.4KB$.

Table 1: Simulation Parameters

Base processor			
microarchitecture	Intel Kaby Lake		
L1 i-cache	SRAM, 32KB, 8-way, 4c lat		
Base L1 d-cache	SRAM, 32KB, 8-way, 4c lat, 0.5c/access B/W		
L2 cache (unified)	SRAM, 256KB, 4-way, 12c lat, 0.5c/access B/W		
L3 cache (unified)	SRAM, 4MB, 16-way, 44c lat, 2c/access B/W		
Main memory	132c latency		
Hybrid L1-d cache			
P_0	SRAM, 16KB, 8-way, 3c lat, 0.5c/access B/W		
P_1	STTRAM, 32KB, 8-way, 8c read lat, 4c/read B/W; 105c write lat, 4c/write B/W		
Relative energy parameters (relative to 32KB SRAM cache)			
Cache	dyn energy /read	dyn energy /write	leakage energy /ns
16KB SRAM	0.758	0.758	0.678
32KB SRAM	1.000	1.000	1.152
256KB SRAM	5.688	5.688	10.715
32KB STTRAM	0.909	36.970	0.053

5 EVALUATION

5.1 Experimental setup

We used the Tejas architectural simulator [21] to evaluate our proposed architecture. The simulation configurations were tuned to resemble the Intel Kaby Lake architecture. We implemented stride and stream prefetchers in all caches (including the L1 i-cache). We integrated the publicly available TAGE-SC-L branch predictor implementation [22] into the simulator. Table 1 gives the details of the simulation parameters used for evaluating the proposal. The Cacti-STT [3] tool was used to derive the parameters of the different SRAM and STTRAM caches [7]. For the STTRAM cache parameters, we have considered the cache to be non-volatile in this evaluation. We do not explore reduced retention alternatives in this work although it could give better results provided an efficient refresh mechanism is incorporated.

We simulated 19 out of the 23 benchmarks of the SPEC CPU2017 benchmark suite. We used the ref inputs. We determined the representative portion of each benchmark using SimPoint [23] and simulated 100 million instructions from it.

5.2 Results

5.2.1 Performance improvement. We first present the analysis of the improvement in performance brought about by CASH. Figure 4 shows the observations. On average, CASH achieves an average performance improvement of 6.1% in the SPEC CPU2017 suite as compared to the baseline system, with a maximum performance improvement of 52.5% in the `lbm` benchmark.

We begin delving deeper by first comparing against a system that has a 16 KB SRAM L1 cache with a 3 cycle latency (equivalent to P_0 in our proposal). We find that CASH outperforms this cache in almost all of the benchmarks, and on average as well (see Figure 4). This indicates that L1 capacity is important for application performance. We next consider a hypothetical system that has a 48 KB L1 cache with a 1 cycle latency. This is a manifestation of our desired cache with a large capacity and a small latency. Benchmarks `exchange2`, `imagick`, and `wrf` show no change in performance when run on this hypothetical system, indicating that the performance of these benchmarks is predominantly determined by factors other than the L1 cache performance such as branch behavior or structural hazards. Without considering these three benchmarks, CASH provides an average performance improvement of 7.2%. As seen in Figure 4, CASH also outperforms the hypothetical $< 48KB, 1c >$ cache on average. This is because CASH is criticality-aware and promotes the availability of critical lines closer to the pipeline (instead of operating on the principle of locality alone).

We additionally compare against the seminal hybrid cache design APM [27] (see Figure 4), which was designed for the Last Level Cache. We implemented it at the L1 level, which is made up of a 16 KB, 3 cycle latency SRAM partition, and a 32 KB, 8-cycle read latency, 105-cycle write latency, STTRAM partition. We present the results of the experiment which employed a 32-entry pattern simulator with 6-way read associativity and 2-way write associativity as this showed the best performance with the lowest energy consumption. APM chooses to move write-intensive lines away from the STTRAM partition, and to not cache dead lines in either partition. However, APM is not cognizant of the STTRAM partition having a higher read latency than the SRAM partition. As a result, we find that when APM is directly implemented at the L1 level, there is an average performance loss of 1.5%. Thus, it is imperative to incorporate some measure like criticality in the policies of a hybrid SRAM-STTRAM cache to accommodate the higher read latencies of STTRAM-based caches.

To further understand the impact of having criticality-aware cache policies, we define a new metric *memory access penalty* (MAP):

$$\text{MAP} = \frac{\sum_{i \in loads} max(0, exec_i - slack_i)}{\text{total number of loads}}$$

where $exec_i$ and $slack_i$ are the execution latency and the global slack of a load instruction i respectively. Intuitively, if a lower value of MAP is observed, it indicates that critical data was often available closer to the pipeline, which in most cases is expected to translate to higher performance.

Figure 5 shows the MAP for the baseline system and the system with CASH at the L1 level. It can be seen that in benchmarks like `fotonik3d`, `lbm`, `omnetpp`, `x264`, and `xalancbmk`, CASH achieved a significantly lower memory access penalty as compared to the base case. This translated to significant performance gains (as seen in Figure 4).

5.2.2 Energy consumption. We next study the relative change in the energy consumed by the private portion of the data memory hierarchy (L1 d-cache, L2 cache, and all its associated circuitry). Figure 6 shows the observations. We observed that, on average (geomean), CASH consumes an excess of 1.7% energy as compared to the base system.

We analyze the energy consumption further. We see that CASH consumes lesser leakage energy (10.1% less) as compared to the baseline, which is due to two reasons: (1) the P_1 partition being STTRAM-based consumes low leakage energy, (2) the application completes execution in lesser time. CASH, however, consumes greater dynamic energy (29% more). On studying the different components of the dynamic energy consumed, we see that although CASH consumes lesser dynamic energy in the L1 SRAM (24.7% less; since P_0 is smaller than the base L1 cache), CASH consumes

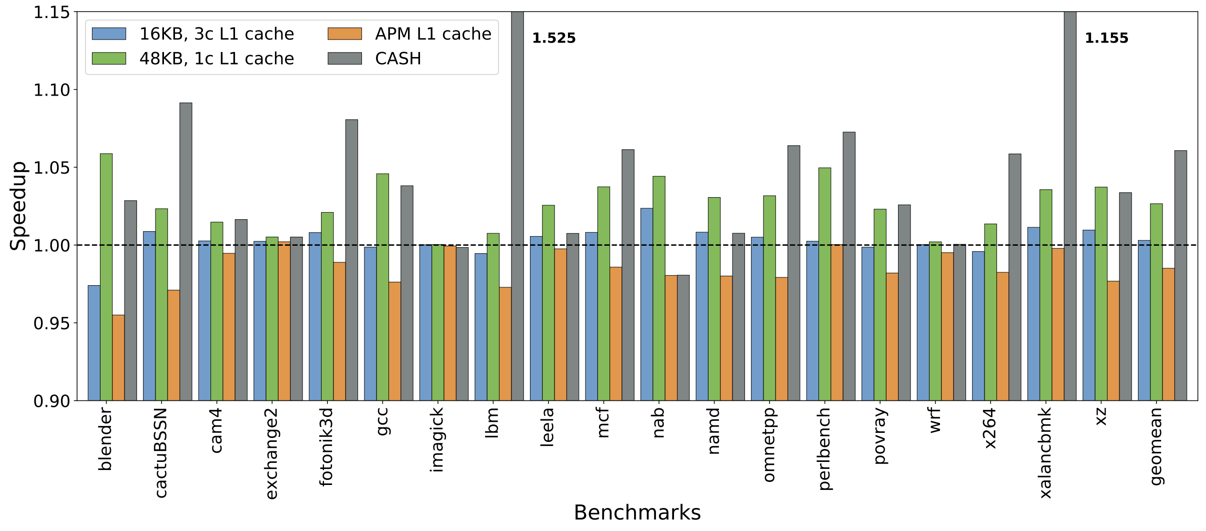

Figure 4: Performance evaluation of the proposed split hybrid L1 cache

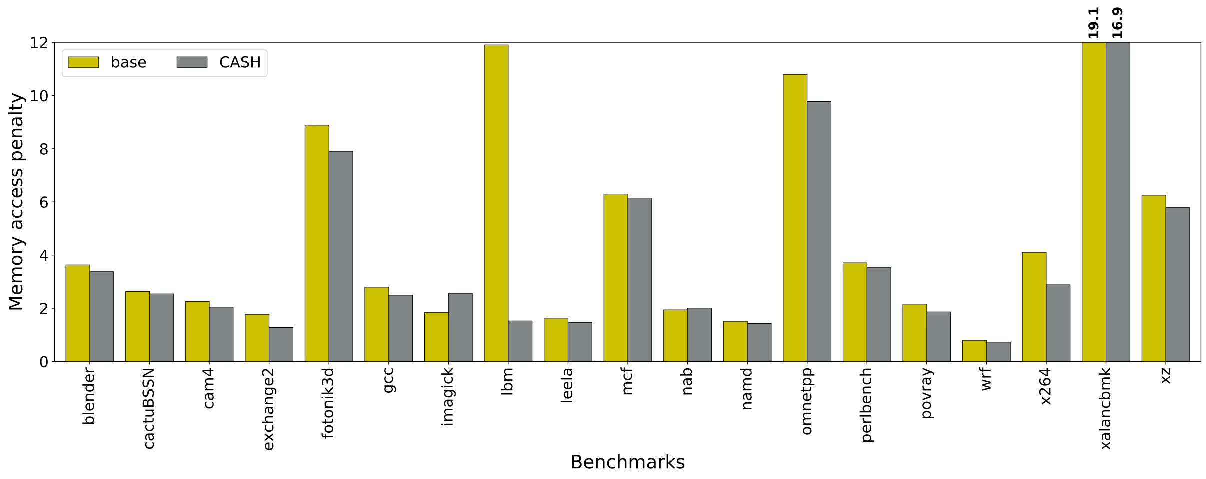

Figure 5: Comparison of the *Memory access penalties* (MAP) observed in different L1 cache systems

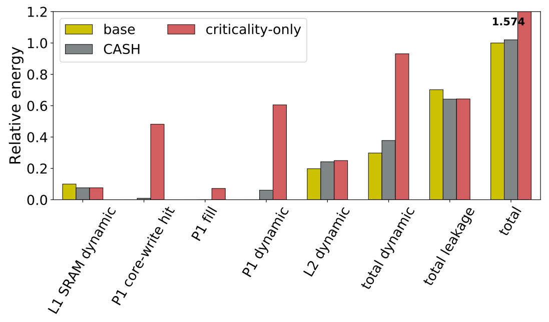

Figure 6: Breakdown of the average energy consumed (relative to the total energy consumed in the base case)

more dynamic energy at the L2 level (25.2% more). CASH also has the P_1 component (15.7% of the total dynamic energy) that is not present in the base system. Though P_1 is STTRAM-based and therefore write operations to it are energy-intensive, CASH employs write-intensity-aware and deadness-aware policies to reduce the number of writes to this partition (see the bars "P1 core-write hit" and "P1 fill" in Figure 6), thereby keeping the P_1 dynamic energy low. Performance is not sacrificed as CASH attempts to keep lines that have high reuse and that serve critical loads closer to the pipeline. To study the energy consumption further, we experiment with a version of CASH that is only criticality-aware (see the red bars in Figure 6), and does not consider the write-intensity and deadness of blocks. This version of CASH displays an average energy consumption overhead of 57.4%. It has a large number of write operations in the P_1 partition – both due to core-writes as well as due to line placements. This leads to a large P_1 dynamic energy consumption (65% of the total dynamic energy). Thus, it is imperative to take write-intensity and deadness into account while designing the policies of a hybrid SRAM-STTRAM cache.

6 CONCLUSION

We have proposed CASH – a design of a hybrid SRAM-STTRAM L1 cache that accommodates the higher read latencies of the STTRAM-based partition by incorporating criticality-aware policies. Such a criticality-oriented approach enables critical load instructions to find their data closer to the pipeline, thereby improving performance. CASH also accommodates the higher write energy of the STTRAM-based partition by suitably adapting the state-of-the-art heuristics of write intensity and deadness to ensure that the number of write operations to the STTRAM partition are few in number, and to lines that have high reuse. This helps achieve the desired abstraction of a low-latency, high-capacity L1D cache, which exhibits an average performance improvement of 6.1%, and an average energy overhead of 1.7%, as compared to a baseline SRAM-only cache of the same area.

REFERENCES

[1] Sukarn Agarwal, Shounak Chakraborty, and Magnus Själander. 2023. Architecting Selective Refresh based Multi-Retention Cache for Heterogeneous System (ARMOUR). In *Design Automation Conference (DAC)*.

[2] Junwhan Ahn, Sungjoo Yoo, and Kiyoung Choi. 2014. DASCA: Dead write prediction assisted STT-RAM cache architecture. In *International Symposium on High Performance Computer Architecture (HPCA)*.

[3] S Arcaro, Stefano Di Carlo, Marco Indaco, D Pala, Paolo Prinetto, and Elena I Vatajelu. 2014. Integration of STT-MRAM model into CACTI simulator. In *International Design and Test Symposium (IDT)*.

[4] Rajeev Balasubramonian, Viji Srinivasan, Sandhya Dwarkadas, and Alper Buyuktosunoglu. 2005. Hot-and-cold: Using criticality in the design of energy-efficient caches. In *Power-Aware Computer Systems (PACS)*.

[5] Mu-Tien Chang, Paul Rosenfeld, Shih-Lien Lu, and Bruce Jacob. 2013. Technology comparison for large last-level caches (L 3 Cs): Low-leakage SRAM, low write-energy STT-RAM, and refresh-optimized eDRAM. In *International Symposium on High Performance Computer Architecture (HPCA)*.

[6] Ping Chi, Shuangchen Li, Yuanqing Cheng, Yu Lu, Seung H Kang, and Yuan Xie. 2016. Architecture design with STT-RAM: Opportunities and challenges. In *Asia and South Pacific Design Automation Conference (ASP-DAC)*.

[7] Ki Chul Chun, Hui Zhao, Jonathan D Harms, Tae-Hyoung Kim, Jian-Ping Wang, and Chris H Kim. 2012. A scaling roadmap and performance evaluation of in-plane and perpendicular MTJ based STT-MRAMs for high-density cache memory. *Journal of Solid-State Circuits* (2012).

[8] Carlos Escuin, Asif Ali Khan, Pablo Ibáñez, Teresa Monreal, Jeronimo Castrillon, and Víctor Viñals. 2023. Compression-Aware and Performance-Efficient Insertion Policies for Long-Lasting Hybrid LLCs. In *International Symposium on High-Performance Computer Architecture (HPCA)*.

[9] Brian Fields, Rastislav Bodik, and Mark D Hill. 2002. Slack: Maximizing performance under technological constraints. *ACM SIGARCH Computer Architecture News* (2002).

[10] B. Fields, S. Rubin, and R. Bodik. 2001. Focusing processor policies via critical-path prediction. In *International Symposium on Computer Architecture (ISCA)*.

[11] Kanad Ghose and Milind B Kamble. 1999. Reducing power in superscalar processor caches using subbanking, multiple line buffers and bit-line segmentation. In *International Symposium on Low Power Electronics and Design (ISLPED)*.

[12] Lei Jiang, Wujie Wen, Danghui Wang, and Lide Duan. 2016. Improving read performance of stt-mram based main memories through smash read and flexible read. In *Asia and South Pacific Design Automation Conference (ASP-DAC)*.

[13] Adwait Jog, Asit K Mishra, Cong Xu, Yuan Xie, Vijaykrishnan Narayanan, Ravishankar Iyer, and Chita R Das. 2012. Cache revive: Architecting volatile STT-RAM caches for enhanced performance in CMPs. In *Design Automation Conference (DAC)*.

[14] Wang Kang, Yuanqing Cheng, Youguang Zhang, Dafine Ravelosona, and Weisheng Zhao. 2014. Readability challenges in deeply scaled STT-MRAM. In *Non-Volatile Memory Technology Symposium (NVMTS)*.

[15] Kunal Korgaonkar, Ishwar Bhati, Huichu Liu, Jayesh Gaur, Sasikanth Manipatruni, Sreenivas Subramoney, Tanay Karnik, Steven Swanson, Ian Young, and Hong Wang. 2018. Density tradeoffs of non-volatile memory as a replacement for SRAM based last level cache. In *International Symposium on Computer Architecture (ISCA)*.

[16] Kyle Kuan and Tosiron Adegbija. 2019. Mirrorcache: An energy-efficient relaxed retention l1 sttram cache. In *Great Lakes Symposium on VLSI (GLSVLSI)*.

[17] Yong Li, Yaojun Zhang, Hai Li, Yiran Chen, and Alex K Jones. 2013. C1C: A configurable, compiler-guided STT-RAM L1 cache. *Transactions on Architecture and Code Optimization (TACO)* (2013).

[18] Sheel Sindhu Manohar, Sparsh Mittal, and Hemangee K Kapoor. 2022. CORIDOR: Using COherence and TempoRal LocalIty to Mitigate Read Disurbance ErrOR in STT-RAM Caches. *Transactions on Embedded Computing Systems (TECS)* (2022).

[19] Anant Vithal Nori, Jayesh Gaur, Siddharth Rai, Sreenivas Subramoney, and Hong Wang. 2018. Criticality aware tiered cache hierarchy: A fundamental relook at multi-level cache hierarchies. In *International Symposium on Computer Architecture (ISCA)*.

[20] Farzane Rabiee, Mostafa Kajouyan, Newsha Estiri, Jordan Fluech, Mahdi Fazeli, and Ahmad Patooghy. 2020. Enduring non-volatile L1 cache using low-retention-time STTRAM cells. In *Annual Symposium on VLSI (ISVLSI)*.

[21] Smruti R Sarangi, Rajshekar Kalayappan, Prathmesh Kallurkar, Seep Goel, and Eldhose Peter. 2015. Tejas: A java based versatile micro-architectural simulator. In *International Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS)*.

[22] André Seznec. 2016. TAGE-SC-L branch predictors again. In *JILP Workshop on Computer Architecture Competitions: Championship Branch Prediction (CBP-5)*.

[23] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. 2002. Automatically characterizing large scale program behavior. *ACM SIGPLAN Notices* (2002).

[24] Clinton W Smullen, Vidyabhushan Mohan, Anurag Nigam, Sudhanva Gurumurthi, and Mircea R Stan. 2011. Relaxing non-volatility for fast and energy-efficient STT-RAM caches. In *International Symposium on High Performance Computer Architecture (HPCA)*.

[25] Zhenyu Sun, Xiuyuan Bi, Hai Li, Weng-Fai Wong, Zhong-Liang Ong, Xiaochun Zhu, and Wenqing Wu. 2011. Multi retention level STT-RAM cache designs with a dynamic refresh scheme. In *International Symposium on Microarchitecture (Micro)*.

[26] Rujia Wang, Lei Jiang, Youtao Zhang, Linzhang Wang, and Jun Yang. 2015. Selective restore: An energy efficient read disturbance mitigation scheme for future STT-MRAM. In *Design Automation Conference (DAC)*.

[27] Zhe Wang, Daniel A Jiménez, Cong Xu, Guangyu Sun, and Yuan Xie. 2014. Adaptive placement and migration policy for an STT-RAM-based hybrid cache. In *International Symposium on High Performance Computer Architecture (HPCA)*.

[28] Yaojun Zhang, Wujie Wen, and Yiran Chen. 2012. The prospect of STT-RAM scaling from readability perspective. *Transactions on Magnetics* (2012).