Compute Caches

Shaizeen Aga, Supreet Jeloka, Arun Subramaniyan, Satish Narayanasamy, David Blaauw, and Reetuparna Das
University of Michigan, Ann Arbor
{*shaizeen, sjeloka, arunsub, nsatish, blaauw, reetudas*}*@umich.edu*

***Abstract*—This paper presents the Compute Cache architecture that enables in-place computation in caches. Compute Caches uses emerging bit-line SRAM circuit technology to re-purpose existing cache elements and transforms them into active very large vector computational units. Also, it significantly reduces the overheads in moving data between different levels in the cache hierarchy.**

Solutions to satisfy new constraints imposed by Compute Caches such as operand locality are discussed. Also discussed are simple solutions to problems in integrating them into a conventional cache hierarchy while preserving properties such as coherence, consistency, and reliability.

Compute Caches increase performance by 1.9× and reduce energy by 2.4× for a suite of data-centric applications, including text and database query processing, cryptographic kernels, and in-memory checkpointing. Applications with larger fraction of Compute Cache operations could benefit even more, as our micro-benchmarks indicate (54× throughput, 9× dynamic energy savings).

I. Introduction

As computing today is dominated by data-centric applications, there is a strong impetus for specialization for this important domain. Conventional processors' narrow vector units fail to exploit the high degree of *data-parallelism* in these applications. Also, they expend disproportionately large fraction of time and energy in *moving data* over cache hierarchy, and in instruction processing, as compared to the actual computation [1].

We present the Compute Cache architecture for dramatically reducing these inefficiencies through in-place (in-situ) processing in caches. A modern processor devotes a large fraction (40-60%) of die area to caches which are used for storing and retrieving data. Our key idea is to re-purpose and transform the elements used in caches into active computational units. This enables computation in-place within a cache sub-array, without transferring data in or out of it. Such a transformation can unlock massive data-parallel compute capabilities, dramatically reduce energy spent in data movement over the cache hierarchy, and thereby directly address the needs of data-centric applications.

Our proposed architecture uses an emerging SRAM circuit technology, which we refer to as *bit-line computing* [2], [3]. By simultaneously activating multiple word-lines, and sensing the resulting voltage over the shared bit-lines, several important operations over the data stored in the activated bit-cells can be accomplished without data corruption. A recently fabricated chip [2] demonstrates feasibility of bit-line computing. They also show a stability of more than six sigma robustness for Monte Carlo simulations, which is considered industry standard for robustness against process variations.

Past processing-in-memory (PIM) solutions proposed to move processing logic *near* the cache [4], [5] or main memory [6], [7]. 3D stacking can make this possible [8]. Compute Caches significantly push the envelope by enabling *in-place* processing using existing cache elements. It is an effective optimization for data-centric applications, where at least one of the operands (e.g., dictionary in WordCount) used in computation has cache locality.

Efficiency of Compute Caches arises from two main sources: massive parallelism and reduced data movement. A cache is typically organized as a set of sub-arrays; as many as hundreds of sub-arrays, depending on the cache level. These sub-arrays can potentially compute concurrently on data stored in them (KBs of data) with little extensions to the existing cache structures (8% of cache area overhead). Thus, caches can effectively function as large vector computational units, whose operand sizes are orders of magnitude larger than conventional SIMD units (KBs vs bytes). To achieve similar capability, the logic close to memory in a conventional PIM solution would need to provision more than hundred additional vector functional units. The second benefit of Compute Caches is that they avoid the energy and performance cost incurred not only for transferring data between the cores and different levels of cache hierarchy (through network-on-chip), but even between a cache's sub-array to its controller (through in-cache interconnect).

This paper addresses several problems in realizing the Compute Cache architecture, discusses ISA and system software extensions, and re-designs several data-centric applications to take advantage of the new processing capability.

An important problem in using Compute Caches is satisfying the *operand locality* constraint. Bit-line computing requires that the data operands are stored in rows that share the same set of bit-lines. We architect a cache geometry, where ways in a set are judiciously mapped to a sub-array, so that software can easily satisfy operand locality. Our design allows a compiler to ensure operand locality simply by placing operands at addresses that are page aligned (same page offset). It avoids exposing the internals of a cache, such as its size or geometry, to software.

When in-place processing is not possible for an operation due to lack of operand locality, we propose to use *near-place* Compute Caches. In near-place design, the source operands are read out from the cache sub-arrays, the operation is performed in a logic unit placed close to the cache controller, and the result may be written back to the cache.

Besides operand locality, Compute Caches brings forth several interesting questions. How to orchestrate concurrent computation over operands spreading across multiple cache sub-arrays? How to ensure coherence between compute-enabled caches? How to ensure consistency model constraints when computation is spread between cores and caches? Soft errors are a significant concern in modern processors. Can ECC be used for Compute Caches? When not possible, what are the alternative solutions? We discuss relatively simple solutions to address these problems.

Compute Caches supports several in-place vector operations: copy, search, compare and logical operations (and, or, xor, and not) which can accelerate a wide variety of applications. We study two text processing applications (word count, string matching), database query processing with bitmap indexing, copy-on-write checkpointing in OS, and bit matrix multiplication (BMM); a critical primitive used in cryptography, bioinformatics, and image processing. We re-designed these applications to efficiently represent their computation in terms of Compute Cache supported vector operations. Section V identifies a number of additional domains that can benefit from Compute Caches: data analytics, search, network processing etc.

We evaluate the merits of Compute Caches for a multi-core processor modeled after Intel's SandyBridge [9] processor with eight cores, three levels of caches, and a ring interconnect. For the applications we study, on average, Compute Caches improve performance by 1.9× and reduce energy by 2.4× compared to a conventional processor with 32-byte wide vector units. Applications with a higher fraction of Compute Cache operations can benefit significantly more. Through micro-benchmarks that manipulate 4KB operands, we show that Compute Caches provide 9× dynamic energy savings over a baseline using 32-byte SIMD units while providing 54× better throughput on average.

In summary, this paper makes the following contributions:

- We make a case for caches that can compute. Using bit-line computing, our Compute Caches naturally support vector processing over large data operands (several KBs). This dramatically reduces overhead due to data movement between caches and cores. Furthermore, in-place computing even avoids data transfer between a cache's sub-array and its controller.

- We present the Compute Cache architecture that addresses various architectural problems: operand locality, managing parallelism across various cache levels and banks, coherency, consistency, and reliability.
- To support Compute Cache operations without operand locality, we study near-place processing in cache.
- We re-designed several important applications (text processing, databases, checkpointing) to utilize Compute Cache operations. We demonstrate significant speedup (1.9×) and energy savings (2.4×) compared to processors with conventional SIMD units. While our savings for applications are limited by the fraction of their computation that can be accelerated using Compute Caches (Amdahl's law), our micro-benchmarks demonstrate that applications with larger fraction of Compute Cache operations could benefit even more(54× throughput, 9× dynamic energy).

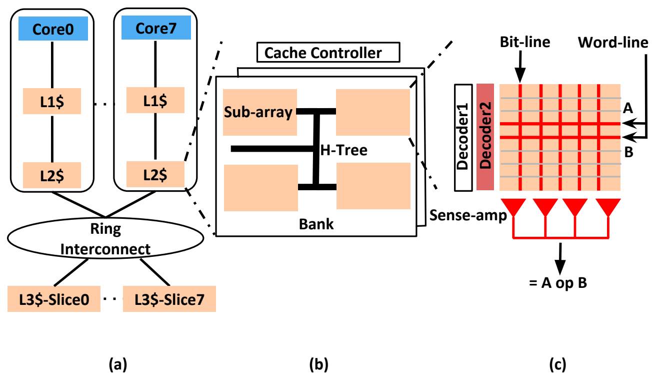


Figure 1: Compute Cache overview. (a) Cache hierarchy. (b) Cache Geometry (c) In-place compute in a sub-array.

II. Background

This section provides a brief background of cache hierarchy, cache geometry, and bit-line computing in SRAM.

A. *Cache Hierarchy and Geometry*

Figure 1 (a) illustrates a multi-core processor modeled loosely after Intel's Sandybridge [9]. It has a three-level cache hierarchy comprising of private L1 and L2, and a shared L3. The shared L3 cache is distributed into slices which are connected to the cores via a shared ring interconnect. A cache consists of a cache controller and several banks ((Figure 1 (b)). Each bank has several sub-arrays connected by a H-Tree interconnect. For example, a 2 MB L3 cache slice has a total of 64 sub-arrays distributed across 16 banks.

A sub-array in a cache bank is organized into multiple rows of data-storing bit-cells. The bit-cells in the same row are connected to a word-line. The bit-cells along a column share the same bit-line. Typically, in any cycle, one word-line is activated, from where a data block is either read from, or written to, through the column bit-lines.

B. *Bit-line Computing*

Compute Caches use emerging bit-line computing technology in SRAMs [2], [3] (Figure 2) which observes that,

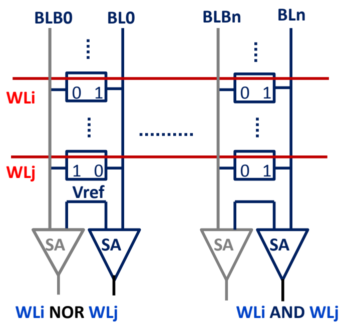


Figure 2: SRAM circuit for in-place operations. Two rows (WL_i and WL_j) are activated. An *AND* operation is performed by sensing bit-line (BL). All the bit-lines are initially pre-charged to '1'. If both the activated bits in a column have a '1' (column 'n'), then the BL stays high and it is sensed as a '1'. If any one of the bits were '0' it will lower the BL voltage below V_{ref} and will be sensed as a '0'. A *NOR* operation can be performed by sensing bit-line bar (BLB).

when multiple word-lines are activated simultaneously, the shared bit-lines can be sensed to produce the result of `and` and `nor` on the data stored in the two activated rows. Data corruption due to multi-row access is prevented by lowering word-line voltage to bias against write of the SRAM array. Jeloka *et al.* [2]'s measurements across 20 fabricated test chips demonstrate that data-corruption does not occur even when 64 word-lines are simultaneously activated during such an in-place computation. They show a stability of more than six sigma robustness for Monte Carlo simulations, which is considered industry standard for robustness against process variations. Also, note that, by lowering the word-line voltage further, robustness can be traded off for increase in delay. Even with it, Compute Caches will still deliver significant savings given its potential (Section VI, 54× throughput, 9× dynamic energy savings).

Section IV-B discusses our extensions to bit-line computing enabled SRAM to support additional operations: `copy`, `xor`, equality comparison, search, and carryless multiplication (`clmul`).

III. A Case for Compute Caches

In-place Compute Cache has the potential to provide massive data-parallelism, while also dramatically reducing the instruction processing and on-chip data movement overheads. Figure 3 pictorially depicts these benefits by comparing a scalar core, a SIMD core with vector processing support, and Compute Caches.

The bottom half in Figure 3 depicts the area proportioning and processing capability of the three architectures. Significant fraction of die area in a conventional processor is for caches. A Compute Cache re-purposes the elements used in this large area into compute units for a small area overhead (8% of cache area). A typical last-level cache consists of hundreds of sub-arrays distributed across different banks which can potentially compute concurrently on cache blocks stored in them. This enables us to exploit large scale data level parallelism (e.g. a 16MB L3 has 512 sub-arrays and can support 8 KB operands) dwarfing even a SIMD core.

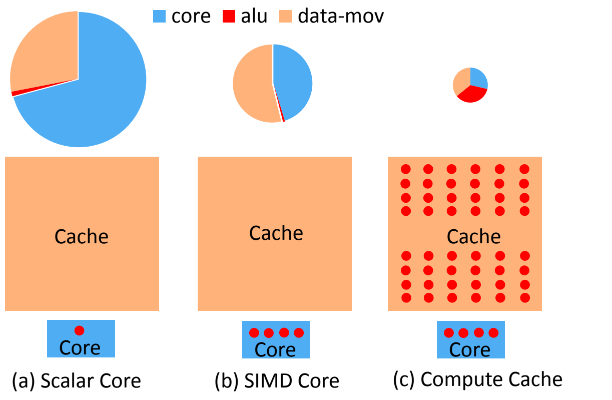


Figure 3: Proportion of energy (top) for bulk comparison operation and area (bottom). Red dot depicts logic capability.

Cache	cache-ic (h-tree)	cache-access
L1-D	179 pJ	116 pJ
L2	675 pJ	127 pJ
L3-slice	1985 pJ	467 pJ

Table I: Cache energy per read access

The top row of Figure 3 shows relative energy consumption for a comparison operation over several blocks of 4KB operands (Section VI-D). In a scalar core, less than 1% of the energy is expended on the ALU operation, while nearly three quarters of the energy is spent in processing instructions in the core, and one-fourth is spent on data movement. While vector processing (SIMD) support (Figure 3 (b)) in general-purpose and data-parallel accelerators reduce the instruction processing overhead to some degree, it does not help address the data movement overhead. Compute Cache architecture (Figure 3 (c)) can reduce the instruction processing overheads by an order of magnitude, by supporting SIMD operation on large operands (tens of KB). Also, it avoids the energy and performance cost due to data movement.

In-place Compute Cache reduces on-chip *data movement overhead*, which consists of two components. First, is the energy spent on data transfer. This includes not only the significant energy spent on the processor interconnect's wires and routers, but also the H-Tree interconnect used for data transfer within a cache. A near-place Compute Cache solution can solve the former but not the latter. As shown in Table I, H-Tree consumes nearly 80% of cache energy spent in reading from a 2MB L3 cache slice.

Second, is the energy spent when reading and writing in the higher-level caches. In a conventional processor, a data block trickles up the cache hierarchy all the way from L3 to L1 cache, and into a core's registers, before it can be operated upon. An L3 Compute Cache can eliminate all this

Opcode	Src1	Src2	Dest	Size	Description
cc_copy	a	-	b	n	$b[i] = a[i]$
cc_buz	a	-	-	n	$a[i] = 0$
cc_cmp	a	b	r	n	$r[i] = (a[i] == b[i])$
cc_search	a	k	r	n	$r[i] = (a[i] == k)$
cc_and	a	b	c	n	$c[i] = a[i]\ \&\ b[i]$
cc_or	a	b	c	n	$c[i] = (a[i]\ \vert\vert\ b[i])$
cc_xor	a	b	c	n	$c[i] = a[i] \oplus b[i]$
cc_clmulX	a	b	c	n	$c_i = \oplus(a[i]\ \&\ b[i])$
cc_not	a	-	b	n	$b[i] =!(a[i])$
a,b,c,k: addresses			r:register	$\forall i, i \in [1, n]$, X = [64/128/256]	

Table II: Compute Cache ISA.

overhead. A shared L3 Compute Cache can also reduce the cost of sharing data between two cores, as it would avoid write-back from a source core's L1 to shared L3, and then a transfer back to a destination core's L1.

IV. Compute Cache Architecture

Figure 1 illustrates the Compute Cache (CC) architecture. We enhance all the levels in the cache hierarchy with in-place compute capability. Computation is done at the highest level where the application exhibits significant locality. In-place compute is based on the bit-line computing technology we discussed in Section II. We enhance these basic in-place compute capabilities to support `xor` and several in-place operations (copy, search, comparison, and carryless multiplication).

In-place computing is possible only when operands are mapped to sub-arrays such that they share the same bit-lines. We refer to this requirement as operand locality. We discuss a cache geometry that allows a compiler to satisfy operand locality by ensuring that the operands are page-aligned.

Each cache controller is extended to manage the parallel execution of CC instructions across its several banks. It also decides the cache level to perform the computation and fetches the operands to that level. Given that a Compute Cache can modify data, we discuss its implication in ensuring coherence and consistency properties. Finally, we discuss design alternatives for supporting ECC in Compute Caches.

In the absence of operand locality, we propose to compute near-place in cache. For this, we add a Logic Unit in the cache controller. Although near-place cache computing requires additional functional units, and cannot save H-Tree interconnect energy inside caches, it successfully helps reduce the energy spent in transferring and storing data in the higher-level caches.

A. Instruction Set Architecture (ISA)

Compute Cache (CC) ISA extensions are listed in Table II. It supports several vector instructions, whose operands are specified using register indirect addressing. Operand sizes are specified through immediate values and can be as large as 16K. It supports vector copying, zeroing, and logical operations. It also supports vector carry less multiply instruction (*cc_clmul*) at single/double/quad word granularity.

It also supports equality comparison and search. We limit the operand size (n) of these instructions to 64 words (512 bytes), so that the result can be returned as a 64-bit value to a processor core's register. For the search instruction, the key size is set to 64 bytes. For smaller keys, the programmer can either duplicate the key multiple times starting from the key's address (if its size is a word multiple), or pad the key and source data operands to be 64 bytes.

B. Cache Sub-arrays with In-Place Compute

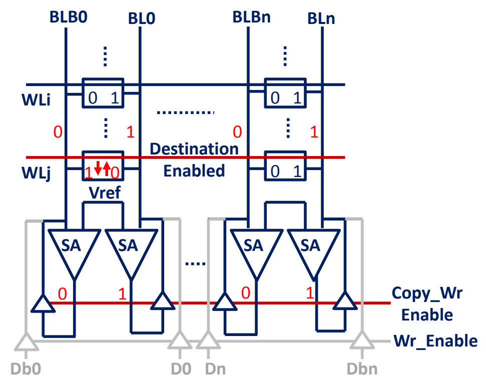


Figure 4: In-place copy operation (from row i to j).

Compute Caches is made possible by our SRAM sub-array design that facilitates in-place computation. We start with the basic circuit framework proposed by Jeloka *et al.* [2], which supports logical `and` and `nor` operations. To a conventional cache's sub-array, we add an additional decoder to allow activating two wordlines, one for each operand. The two single-ended sense amplifiers required for separately sensing both the bit-lines attached to a bit-cell are obtained by re-configuring the original differential sense amplifier.

In addition to `and` and `nor` operations, we extend the circuit to support `xor` operation by NOR-ing bit-line and bit-line complement. We realize compound operations such as *compare* and *search* by using the results of bitwise `xor`. To compare two words, the individual bit-wise `xor` results are combined using a wired-NOR. Comparison is utilized to do iterative *search* over cache blocks stored in sub-arrays.

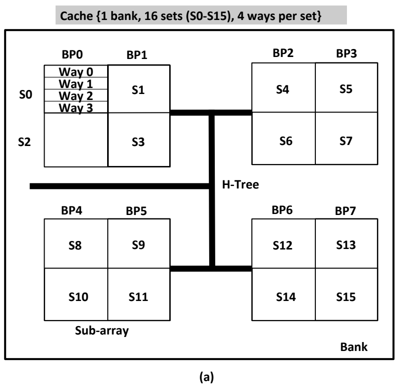
By feeding the result of the sense-amplifiers back to the bit-lines, one word-line can be copied to another without ever latching the source operand. We leverage the fact that the last read value is same as the data to be written in the next cycle, and coalesce the read-write operation to enable more energy-efficient *copy* operation as shown in Figure 4. By resetting input data latch before a write we can enable in-place zeroing of a cache block.

Finally, the carryless multiplication (`clmul`) operation is done using a logical `and` on two sub-array rows, followed by `xor` reduction of all the resultant bits. This is supported by adding a `xor` reduction tree to each sub-array.

Our extensions have negligible impact on the baseline read/write accesses as they use the same circuit as the baseline, including differential sensing. An in-place operation takes longer than a single read or write sub-array access, as it requires longer word-line pulse to activate and sense two rows to compensate for the lower word-line voltage. Sensing time also increases due to the use of single-ended sense amplifiers, as opposed to differential sensing. However, note that this is still less than the delay baseline would incur to accomplish an equivalent in-place operation, as it would require multiple read accesses and/or write access. Section VI-C provides the detailed delay, energy and area parameters for compute capable cache sub-arrays.

C. Operand Locality

For in-place operations, the operands need to be physically stored in a sub-array, such that they share the same set of bitlines. We term this requirement as *operand locality*. In this section, we discuss cache organization and software constraints that can together satisfy this property. *Fortunately, we find that software can ensure operand locality as long as operands are page-aligned, i.e., have the same page offset. Besides this, the programmer or the compiler does not need to know about any other specifics of the cache geometry.*



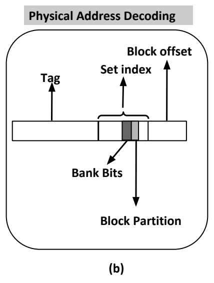


Figure 5: Cache organization example, address decoding ([i][j] = set i, way j), alternate address decoding for parallel tag-data access caches

Operand locality aware cache organization: Figure 5 illustrates a simple cache with one bank each with four sub-arrays. Rows in a sub-array share the same set of bitlines.

We define a new term, *Block Partition* (BP). Block partition for a sub-array is the group of cache blocks in that sub-array that share the same bitlines. In-place operation is possible between any two cache blocks stored within a block partition. In our example, since each row in a sub-array has two cache blocks, there are two block partitions per sub-array. In total, there are eight block partitions (BP0-BP7). In-place compute is possible between any blocks that map to the same block partition (e.g. blocks in sets `S0` and `S2`).

Cache	Banks	BP	Block size	Min. address bits match
L1-D	2	2	64	8
L2	8	2	64	10
L3-slice	16	4	64	12

Table III: Cache geometry and operand locality constraint.

We make two design choices for our cache organization to simplify operand locality constraint. First, all the ways in a set are mapped to the same block partition as shown in Figure 5(a). This ensures that operand locality would not be affected based on which way is chosen for a cache block.

Second, we use a portion of set-index bits to select the block's bank and block partition, as shown in Figure 5(b). As long as these are the same for two operands, they are guaranteed to be mapped to the same block partition.

Software requirement: The number of address bits that must match for operand locality varies based on the cache size. As shown in Table III, even the largest cache (L3) in our model requires that only least 12 bits are the same for two operands (we assume pages are mapped to a NUCA slice closest to the core actively accessing them). Given that our pages are 4KB in size, we observe that as long as the operands are page aligned, i.e., have the same page offset, then they will be placed in the address space such that the least significant bits (12 for 4 KB page) in their addresses (both virtual and physical) match. This would trivially satisfy the operand locality requirement for all the cache levels and sizes we study. Note that, we only require operands to be placed at the same offset of 4KB memory regions, and it is not necessary to place them in separate pages. For super-pages that are larger than 4KB, operands can be placed within a page while ensuring 12-bit address alignment.

We expect that for data-intensive regular applications that operate on large chunks of data, it is possible to satisfy this property. Many operating system operations that involve copying from one page to another are guaranteed to exhibit operand locality for our system. Compiler and dynamic memory allocators could be extended to optimize for this property in future.

Finally, a binary compiled with a given address bit alignment requirement (12 bits in our work) is portable across a wide range of cache architectures as long as the number of address bits to be aligned is equal to or less than what they were compiled for. If the cache geometry changes such that it requires greater alignment, then the programs would have to be recompiled to satisfy that stricter constraint.

Column Multiplexing: With column multiplexing, multiple adjacent bit-lines are multiplexed to a single bit data output, which is then observed using one sense-amplifier. This keeps area overhead of peripherals under check and improves resilience to particle strikes. Fortunately, in column multiplexed sub-arrays, adjacent bits in a cache block are interleaved across different sub-arrays such that their bitlines are not multiplexed. In this case, the logical block partition that we defined would be interleaved across the sub-arrays. Thus, an entire cache block can be accessed in parallel. Given this, in-place concurrent operation on all the bits in a cache block is possible even with column multiplexing.

Our design choice of placing ways of a set within a block partition does not affect the degree of column multiplexing as we interleave cache blocks of different sets instead.

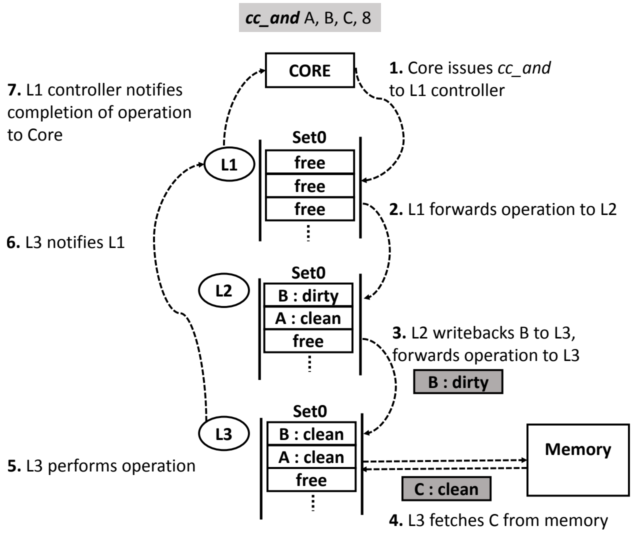
Way Mapping vs Parallel Tag-Data Access: We chose to place all the ways of a set within a block partition, so that operand locality is not dependent on which way is chosen for a block at runtime. However, this prevents us from supporting parallel tag-data access, where all the cache blocks in a set are pro-actively read in parallel with the tag match. This optimization is typically used for L1 as it can reduce the read latency by overlapping tag match with read. But it incurs a high read energy overhead ($4.7\times$ higher energy per access for L1 cache) for modest performance gain (2.5% for SPLASH-2[10]). Given the significant benefits of L1 Compute Cache, we think it is a worthy trade-off to forgo this optimization for L1.

D. *Managing Parallelism*

Cache controllers are extended to provision for CC controllers which orchestrate the execution of CC instructions. The CC controller breaks a CC instruction into multiple simple vector operations whose operands span at most a single cache block and issues them to the sub-arrays. Since a typical cache hierarchy can have hundreds of sub-arrays (16MB L3 cache has 512 sub-arrays), we can potentially issue hundreds of concurrent operations. This is only limited by two factors. First, the bandwidth of the shared interconnects used to transmit address/commands. Note that we do not replicate the address bus in our H-tree interconnects. Second, number of sub-arrays activated at same time can be limited to limit peak power drawn.

The controller at L1-cache uses an `instruction table` to keep track of the pending CC instructions. The simple vector operations are kept track of in the `operation table`. The instruction table tracks metadata associated at instruction level (i.e., result, count of simple vector operations completed, next simple vector operation to be generated). The operation table, on other hand, tracks status of each operand associated with the operation and issues request to fetch the operand if it is not present in the cache (Section IV-E). When all operands are in cache, we issue the operation to the cache sub-array. As operations complete they update the instruction table, and the L1-cache controller notifies the core when an instruction is complete.

To support search instruction, CC controller replicates key in all the block partitions where the source data resides. To avoid doing this again for the same instruction, we track such replications per instruction in a `key table`.



cc_and A, B, C, 8
CORE
1. Core issues cc_and to L1 controller
7. L1 controller notifies completion of operation to Core
L1
Set0
free
free
free
2. L1 forwards operation to L2
6. L3 notifies L1
L2
Set0
B : dirty
A : clean
free
3. L2 writebacks B to L3, forwards operation to L3
B : dirty
L3
Set0
B : clean
A : clean
free
Memory
5. L3 performs operation
C : clean
4. L3 fetches C from memory

Figure 6: Compute Caches in action

Finally, if the address range of any operand of a CC instruction spans multiple pages, it raises a pipeline exception. The exception handler splits the instruction into multiple CC operations such that each of its operands are within a page.

E. *Fetching In-Place Operands*

The Compute Cache (CC) controllers are responsible for deciding the level in the cache hierarchy where CC operations need to be performed, and issuing commands to the cache sub-array to execute them. To simplify our design, in our study, the CC controller always performs the operations at the highest-level cache where *all* the operands are present. If any of the operands are not cached, then the operation is performed at lowest-level cache (L3). Cache allocation policy can be improved in future by enhancing our CC controller with a cache block reuse predictor [11].

Once a cache level is picked, CC controller fetches any missing operands to that level. The controller also pins the cache-lines the operands are fetched in while the CC operation is under way. To avoid the eviction of operands while waiting for missing operands, we promote the cache blocks of that operand to the MRU position in the LRU chain. However, on receiving a forwarded coherence request, we release the lock to avoid deadlock and re-fetch the operand. Getting a forwarded request to a locked cache line will be rare for two reasons. First, in DRF [12] compliant programs, only one thread will be able to operate on a cache block while holding its software lock. Second, as operands of a single CC operation are cache block wide, false sharing will be low. Nevertheless, to avoid starvation in pathological scenarios, if CC operation fails to get permission after repeated attempts (set to two), processor core will translate and execute a CC operation as RISC operations.

Figure 6 shows a working example. Core issues operation *cc_and* with address operands A, B and C to L1 controller (❶). Each is of size 64 bytes (8 words) spanning an entire cache block. For clarity, only one cache set in each cache level is shown. None of the operands are present in L1 cache. Operand B is in L2 cache and is dirty. L3 cache has clean copy of A and a stale copy of B. C is not in any cache.

L3 cache is chosen for the CC operation, as it is the highest cache level where all operands are present. L1 and L2 controllers will forward this operation to L3 (❷, ❸). Before doing so, L2 cache will first write-back B to L3. Note that caches already write-back dirty data to next cache level on eviction and we use this existing mechanism.

On receiving the command, L3 fetches C from memory (❹). Note that, as an optimization, C need not be fetched from memory as it will be over-written entirely. Once all the operands are ready, L3 performs the CC operation (❺) and subsequently notifies the L1 controller (❻) of it's completion, which in turn notifies the core (❼).

F. Cache Coherence

Compute Cache optimization interacts with the cache coherence protocol minimally and as a result does not introduce any new race conditions. As discussed above, while the controller locks cache lines while performing CC operation, on receipt of a forwarded coherence request, the controller releases the lock and responds to the request. Thus, a forwarded coherence request is always responded to in cases where it would be responded to in the baseline design.

Typically, higher-level caches writeback dirty data to the next-level cache on evictions. Coherence protocols already support such writebacks. In the Compute Cache architecture, when a cache level is skipped to perform CC operations, any dirty operands in the skipped level need to be written back to next level of cache to ensure correctness. To do this, we use the existing writeback mechanism and thus require no change to the underlying coherence protocol.

G. Consistency Model Implications

Current language consistency models (C++ and Java) are variants of the DRF model [12], and therefore a processor only needs to adhere to the RMO memory model. While ISAs providing stronger guarantees (x86) exist, they can be exploited only by writing assembly programs. As a consequence, while we believe stronger memory model guarantees for Compute Caches is an interesting problem (to be explored in future work), we assume RMO model in our design. In RMO, no memory ordering is needed between data reads and writes, including all CC operations. Individual operations within a vector CC instruction can also be performed in parallel by the CC controller.

Programmers use `fence` instructions to order memory operations, which is sufficient in the presence of CC instructions. Processor stalls commit of a fence operation until preceding pending operations are completed, including CC operations. Similar to conventional vector instructions, it is not possible to specify a fence between scalar operations within a single vector CC instruction.

H. Memory Disambiguation and Store Coalescing

Similar to SIMD instructions, Compute Cache (CC) vector instructions require additional support in the processor core for memory ordering. We classify instructions in CC ISA into two types. CC-R type (`cc_cmp`, `cc_search`) only read from memory. The rest of the instructions are CC-RW type as they both read and write from memory. Under RMO memory model, CC-R can be executed out-of-order, whereas CC-RW behaves like a store. In the following discussion, we refer to CC-R as load, and CC-RW as store.

Conventional processor cores use a load-store queue (LSQ) to check for address conflicts between a load and the preceding uncommitted stores. As vector instructions can access more than a word, it is necessary to enhance the LSQ with the support for checking address ranges, instead of just one address. For this reason, we use a dedicated vector LSQ, where each entry has additional space to keep track of address ranges for the operands of a vector instruction.

Similar to LSQ, we also split the store buffer into two, one for scalar stores and another for vector stores. The vector store buffer supports address range checks (max 12 comparisons/entry). Our scalar store buffer permits coalescing. However, it is not possible to coalesce CC-RW instructions with any store, because their output is not known till they are performed in a cache. As the vector store buffer is non-coalescing, it is possible for the two store buffers to contain stores to the same location. If such a scenario is detected, the conflicting store is stalled until the preceding store is complete which ensures program order between stores to the same location. We augment the store buffer with a field which points to any successor store and a stall bit. The stall bit is reset when the predecessor store completes.

Data values are not forwarded from vector stores to any loads, or from any store to a vector load. Code segments where both vector and scalar operations access the same location within a short time span is likely to be rare. If such a code segment is frequently executed, the compiler can choose to not employ Compute Cache optimization.

I. Error Detection and Correction

Systems with strong reliability requirements employ Error Correction Codes (ECC) for caches. ECC protection for conventional and near-place operations are unaffected in our design. For *cc_copy* simply copying ECC from source to destination suffices. For *cc_buz*, ECC of zeroed blocks can be updated. For comparison and search, ECC check can be performed by comparing the ECCs of the source operands. An error is detected if data bits match, but the ECC bits don't, or vice versa.

For in-place *logical* operations (cc_and, cc_or, cc_xor, cc_clmul, and cc_not), it is challenging to perform the check and compute the ECC for the result. We propose two alternatives. One alternative is to read out the xor of the two operands and their ECCs, and check the integrity at the ECC logic unit ($ECC(A\ xor\ B) = ECC(A)\ xor\ ECC(B)$). This unit also computes the ECC of the result. Our sub-array design permits computing the xor operation alongside any logical operation. Although the logical operation is still done in-place, this method will incur extra data transfers to and from the ECC logic unit. Cache scrubbing during cache idle cycles [13] is a more attractive option. Since soft errors in caches are infrequent (0.7 to 7 errors/year [14]), periodic scrubbing can be effective while keeping performance and energy overheads low.

J. Near-Place Compute Caches

In the absence of operand locality, we propose to compute instructions "near" the cache. Our controller is provisioned with additional logic units (not arithmetic units) and registers to temporarily store the operands. The source operands are read from the cache sub-array into the registers at the controller, and then computed results are written back to the cache. In-place computation has two benefits over near-place computation. First, it provides massive compute capability for almost no additional area overhead. For example, a 16 MB L3 with 512 sub-arrays allows 8KB of data to be computed in parallel. To support equivalent computational capability, we would need 128 vector ALUs, each of width 64-bytes. This is not a trivial overhead. We assume one vector logic unit per cache controller in our near-cache design. Second, in-place compute avoids data transfer over H-Tree wires. This reduces in-place compute latency (14 cycles) compared to near-cache (22 cycles). Also, 60%-80% of total cache read energy is due to H-Tree wire transfer (See Table I), which is eliminated with in-cache computation. Nevertheless, near cache computing retains the other benefits of Compute Caches, by avoiding transferring data to the higher-level caches and the core.

V. Applications

Our Compute Cache design supports simple but common operations, which can be utilized to accelerate diverse set of data intensive applications.

Search and Compare Operations: Compare and search are common operations in many emerging applications, especially text processing. Intel recently added seven new instructions to the x86 SSE 4.2 vector support that efficiently perform character searches and comparison [15]. Compute Cache architecture can significantly improve the efficiency of these instructions. Similar to specialized CAM accelerators [16], our search functionality can be utilized to speed up applications such as, search engines, decision tree training and compression and encoding schemes.

Configuration	8 core CMP
Processor	2.66 GHz out-of-order core, 48 entry LQ, 32 entry SQ
L1-I Cache	32KB, 4-way, 5 cycle access
L1-D Cache	32KB, 8-way, 5 cycle access
L2 Cache	inclusive, private, 256KB, 8-way,11 cycle access
L3 Cache	inclusive, shared, 8 NUCA slices, 2MB each, 16-way, 11 cycle + queuing delay
Interconnect	ring, 3 cycle hop latency, 256-bit link width
Coherence	directory based, MESI
Memory	120 cycle latency

Table IV: Simulator Parameters

Logical Operations: Compute Cache logical operations can speedup processing of commonly used bit manipulation primitives such as *bitmaps*. Bitmaps are used in graph and database indexing/query processing. Query processing on databases with bitmap indexing requires logical operation on large bitmaps. Compute Caches can also accelerate binary *bit matrix multiplication (BMM)* which has uses in numerous applications such as error correcting codes, cryptography, bioinformatics, and Fast Fourier Transform (FFT). Given its importance, it was implemented as a dedicated instruction in Cray supercomputers [17] and Intel processors provision a x86 carryless multiply ($clmul$) instruction to speed it. Inherent cache locality in matrix multiplication makes BMM suitable for Compute Caches. Further, our large vector operations can allow BMM to scale to large matrices.

Copy Operation: Prior research [7] makes a strong case for optimizing copy performance which is a common operation in many applications in system software and warehouse scale computing [18]. The operating system spends a considerable chunk of its time (more than 50%) copying bulk data [19]. For instance copying is necessary for frequently used system calls like fork, inter-process communication, virtual machine cloning and deduplication, file system and network management. Our copy operation can accelerate checkpointing, which has a wide range of uses, including fault tolerance and time-travel debugging. Finally, our copy primitive can also be employed in bulk zeroing which is an important primitive required for memory safety [20].

VI. Evaluation

In this section we demonstrate the efficacy of Compute Caches (CC) using both micro-benchmark study and a suite of data-intensive applications.

A. Simulation Methodology

We model a multi-core processor using SniperSim [21], a Pin-based simulator per Table IV. We use McPAT [22] to model power consumption in both cores and caches.

B. Application Customization and Setup

In this section we describe how we redesigned applications in our study to utilize CC instructions.

WordCount: WordCount [23] reads a text file (10MB) and builds a dictionary of unique words and their frequency of appearance in the file. While the baseline does a binary search over the dictionary to check if a new word is found, we model the dictionary as alphabet indexed (first two letters of word) CAM (1KB each). As the dictionary is large (719KB) we perform search operations in L3 cache. CC search instruction returns a bit vector indicating match/mismatch for multiple words and hence we also model additional mask instructions which report match/mismatch per word.

StringMatch: StringMatch [23] reads words from a text file (50MB), encrypts them and compares them to a list of encrypted keys. Encryption cannot be offloaded to cache, hence, encrypted words are present in L1-cache and we perform CC search in it. By replicating an encrypted key across all sub-arrays in L1, a single search instruction can compare it against multiple encrypted words. Similar to WordCount we also model mask instructions.

DB-BitMap: We also model FastBit [24] a bitmap index library. The input database index is created using data sets obtained from a real physics experiment, STAR [25]. A sample query performs logical OR or AND of large bitmap bins (several 100 KBs each). We modify the query to use cc_or operations (each processes 2KB of data). We measure average query processing time for a sample query mix running over uncompressed bitmap indexes.

BMM: Our optimized baseline BMM implementation (Section V) uses blocking and x86 CLMUL instructions. Given the reuse of matrix we perform cc_clmul in L1-cache. We model 256×256 bit matrices.

Checkpointing: We model in-memory copy-on-write checkpointing support at page granularity for SPLASH-2 [10] benchmark suite (checkpointing interval of 100,000 application instructions).

C. Compute Sub-Array: Delay and Area Impact

Compute Caches have negligible impact on the baseline read/write accesses as we still support differential sensing. To get delay and energy estimates, we perform SPICE simulations on a 28nm SOI CMOS process based sub-array, using standard foundry 6T bit-cells. [1] A and/or/xor 64-byte in-place operation is $3\times$ longer as compared to single sub-array access while rest of CC operations are $2\times$ longer. In terms of energy, cmp/search/clmul are $1.5\times$, copy/buz/not are $2\times$, and the rest are $2.5\times$ baseline sub-array access. The area overhead is 8% for a sub-array of size 512×512 [2]. Note, our estimates account for technology variations and process, voltage and temperature changes. Further, these estimates are conservative when compared to measurements on silicon [2] in order to provision for robust margin against read disturbs and to account for circuit parameter variation across technology nodes.

cache	write	read	cmp	copy	search	not	logic
L3	2852	2452	840	1340	3692	1340	1672
L2	1154	802	242	608	1396	608	704
L1	375	295	186	324	561	324	387

Table V: Cache energy (pJ) per cache-block (64-byte)

We use the above parameters in conjunction with energy per cache access from McPAT to determine the energy of CC operations (Table V). CC operations cost higher in lower-level caches as they employ larger sub-arrays. However, they do deliver higher savings (compared to baseline read/write(s) needed) as they have larger in-cache interconnect components. For search, we assume a write operation for key; this cost will get amortized over large searches.

D. Microbenchmark Study

To demonstrate the efficacy of Compute Caches we model four microbenchmarks: copy, compare, search and logical-or. We compare Compute Caches to a baseline($Base\_32$) which supports 32-byte SIMD loads and stores.

Figure 7 (a) depicts the throughput attained for different operations for operand size of 4KB. For this experiment, all operands are in L3 cache and the Compute Cache operation is performed therein. Among the operations, for baseline, search achieves highest throughput as it incurs single cache miss for the key and subsequent cache misses are only for data. Compute Cache accelerates throughput for all operations: $54\times$ over $Base\_32$ averaged across the four kernels. Our throughput improvement has two primary sources: massive data parallelism exposed in presence of independent sub-arrays to compute in, and latency reduction due to avoiding data movement to the core. For instance, for copy operation, data parallelism exposes $32\times$ and latency reduction exposes $1.55\times$ throughput improvement.

Figure 7 (b) depicts the dynamic energy consumed for operand size of 4KB. Dynamic energy depicted is broken down into core, cache data access (cache-access), cache interconnect (cache-ic) and network-on-chip (noc) components. We term data movement energy to be everything except the core component. Overall, CC provides dynamic energy savings of 90%, 89%, 71% and 92% for copy, compare, search and logical (OR) kernels relative to $Base\_32$. Large vector CC instructions help bring down core component of energy. Further, CC successfully eliminates all the components of data movement. Writes incurred due to key replication limit efficacy of search CC operation in bringing down L3 cache energy components. As data size to be searched increases, key replication overheads will get amortized increasing effectiveness of CC.

Figure 7 (c) depicts total energy consumed broken down into static and dynamic components. Due to reduction in execution time, CC can significantly reduce static energy.

[1]SRAM arrays we model are 6T cell based. Lower-level caches (L2/L3) are optimized for density and employ 6T-based arrays. However, L1-cache can employ 8T cell based designs. To support in-place operations in such a design, a differential read-disturb resilient 8T design [26] can be used.

[2]The optimal sub-array dimension for L3 and L2 caches we model are 512×512 and 128×512 bits respectively.

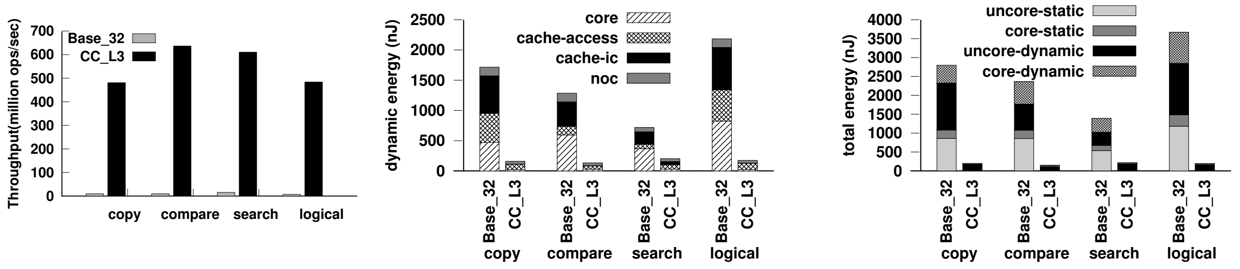


Figure 7: Benefit of CC for 4KB operand. a) Throughput b) Dynamic energy c) Total energy

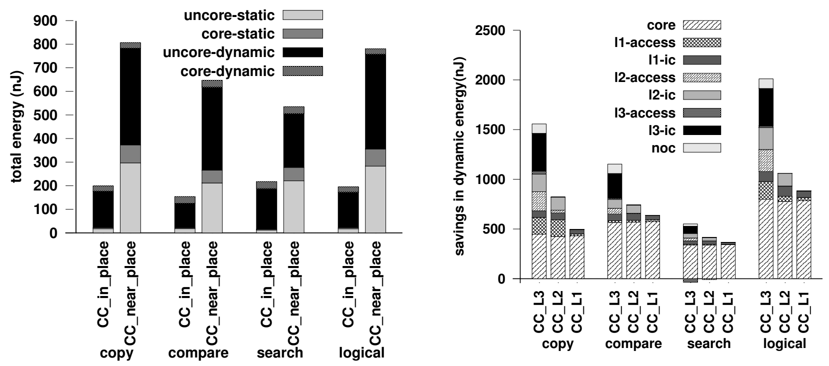


Figure 8: a) Total energy of in-place vs near place for 4KB operand b) Savings in dynamic energy for 4KB operand for different cache levels

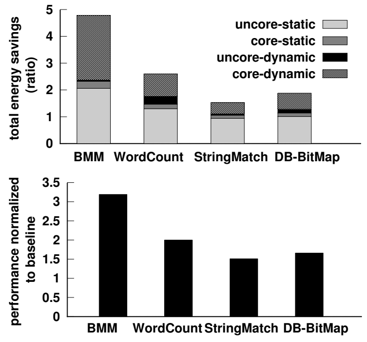


Figure 9: a) Total energy benefit b) Performance improvement of CC for applications

Overall, averaged across the four kernels studied, CC provides 91% in total energy savings relative to $Base\_32$.

Near-place design: In our analysis so far, we have assumed perfect operand locality i.e. all Compute Cache operations are performed in-place. Figure 8 (a) depicts the total energy for near-place and in-place CC configurations. Recall that in-place computation enables far more parallelism than near-place and offers larger savings in terms of performance and hence total energy. For example, our L3-cache allows 8KB data to be operated in parallel. Near-place design would need 128 64-byte wide logical units to provide equivalent data parallelism. This is not a trivial overhead. As such, for 4KB operands, in-cache provides 3.6× total energy savings and 16× throughput improvement on average over near-place. Note however that, near-place can still offer considerable benefits over the baseline architecture.

Computing at different cache levels: We next evaluate the efficacy of Compute Caches when operands are present in different cache levels. Figure 8 (b) depicts the difference in dynamic energy between CC configurations and their corresponding $Base\_32$ configurations. As expected, the absolute savings are higher, when operands are in lower-level caches. However, we find that doing Compute Cache operations in L1 or L2 cache can also provide significant savings. As the number of CC instructions stays same regardless of cache level, core energy savings is equal for all cache levels. Overall, CC provides savings of 95% and 34% for L1 and L2 caches respectively relative to $Base\_32$.

E. Application Benchmarks

In this section we study the benefits of Compute Caches for five applications. Figure 9 (b) shows the overall speedup of Compute Caches for four of these applications. We see a performance improvement of 2× for WordCount, 1.5× for StringMatch, 3.2× for BMM, and 1.6× for DB-BitMap. Figure 9 (a) shows ratio of total energy of CC to baseline processor with 32-byte SIMD units. We observe average energy savings of 2.7× across these applications. Majority of benefits come from three sources: data parallelism exposed by large vector operations, reduction in number of instructions and data movement.

For instance, recall that while baseline WordCount does a binary search over dictionary of unique words, Compute Cache does a CAM search using cc_search instructions. Superficially it may seem that binary search will outperform CAM search. However, we find that CC version has 87% fewer instructions by doing away with book keeping instructions of binary search. Further, our vector cc_search enables energy efficient CAM searches. These benefits are also evident in StringMatch, BMM and DB-BitMap (32%, 98% and 43% instruction reduction respectively). The massive data level parallelism we enable benefits data intensive range and join queries in DB-BitMap application. Recall that this benchmark performs many independent logical OR

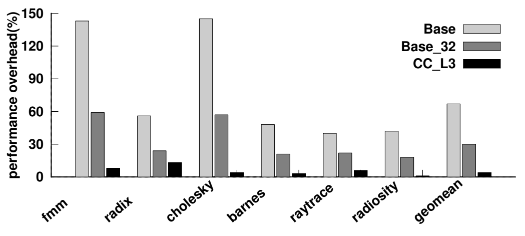


Figure 10: Performance overhead of CC for checkpointing

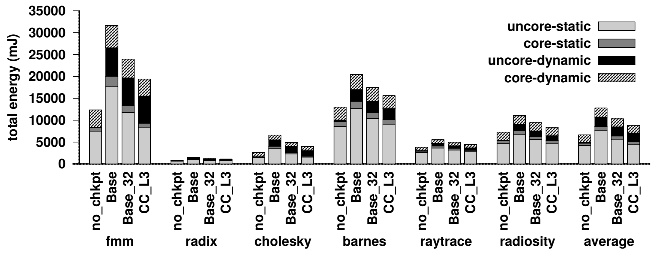


Figure 11: Total energy with and without checkpointing

operations over large bitmap bins. Since these operations are independent, many of them can be issued in parallel.

Significant cache locality exhibited by these applications makes them highly suitable for Compute Caches. As cache accesses are cheaper than memory accesses, computation in cache is more profitable for data with high locality or reuse. The dictionary in WordCount has high locality. BMM has inherent locality due to the nature of matrix multiplication. In DB-BitMap, there is significant reuse within a query due to aggregation of results into a single bitmap bin, and there is potential reuse of bitmaps across queries. In StringMatch, locality comes due to repeated use of encrypted keys.

Figure 10 depicts the overall checkpointing overhead for SPLASH-2 applications as compared to baseline with no checkpointing. In absence of SIMD support, this overhead can be as high as 68% while in presence of it the average overhead is 30%. By further reducing instruction count and avoiding data movement, CC brings down this overhead to a mere 6%. CC successfully relegates checkpointing to cache, avoids data pollution of higher level caches and relieves the processor of any checkpointing overhead. Figure 11 shows significant energy savings due to Compute Caches. Note that, for checkpointing, all operations are page-aligned and hence we achieve perfect operand locality.

VII. Related Work

Past processing-in-memory (PIM) solutions move compute *near* the memory [6]. This can be accomplished using recent advancements in 3D die-stacking [8]. There have also been few proposals that talk about adding hardware structures *near* the cache, which track information that helps improve efficiency of copy [5] and atomic operations [27]. Associative processor [28] uses CAMs (area and energy inefficient compared to SRAM caches) as caches and augments more logic around CAM to orchestrate computation on them. None of these solutions exploit the benefits of in-place bit-line computing cache noted in Section III. We get massive number of compute units by re-purposing cache elements that already exist. Also, in-place Compute Cache reduces data movement overhead between a cache's sub-arrays and its controller. On the flip side, in-place cache computing imposes restrictions on the type of operations that can be supported and placement of operands, which we address in the paper. When in-place operation is not possible, we used near-place Compute Cache for copy, logical, and search operations, which has also not been studied in the past.

Row-clone [7] enabled data copy from a source DRAM row to a row buffer and then to a destination row. Thereby, it avoided data movement over the memory channels. A subsequent CAL article [29] suggested that data could be copied to a temporary buffer in DRAM, from where logical operations could be performed. Row-clone's approach is also a form of near-place computing, which requires that all operands are copied to new DRAM rows before they can be operated upon. Bit-line in-place operations may not be feasible in DRAM, as DRAM reads are destructive (one of the reasons why DRAMs need refreshing).

Recent research enhanced non-volatile memory technology to support certain in-memory CAM [16] and bitwise logic operations [30]. Compute Cache architecture is more efficient when at least one of the operands has cache locality (e.g., dictionary in word count). Ultimately, the locality characteristics of an application should guide in which level of memory hierarchy the computation must be performed.

Bit-line computing in SRAMs has been used to implement custom accelerators: approximate dot products in analog domain for pattern recognition [31] and CAMs [32]. However, it has not been used to architect a compute cache in a conventional cache hierarchy, where we need general solutions to problems such as operand locality, coherence and consistency which are addressed in this paper. We also demonstrated the utility of our Compute Cache enabled operations to accelerate a fairly diverse range of applications (databases, cryptography, data analytics).

VIII. Conclusion

In this paper we propose Compute Cache (CC) architecture which unlocks hitherto untapped computational capability present in on-chip caches by exploiting emerging SRAM circuit technology. Using bit-line computing enabled caches, we can perform several simple operations in-place in cache over very-wide operands. This exposes massive data parallelism saving instruction processing, cache interconnect and intra-cache energy expenditure. We present solutions to several challenges exposed by such an architecture. We demonstrate the efficacy of our architecture using a suite of data intensive benchmarks and micro-benchmarks.

IX. Acknowledgments

We thank the anonymous reviewers for their comments which helped improve this paper. This work was supported in part by the NSF under the CAREER-1149773 and SHF-1527301 awards and by C-FAR, one of the six SRC STARnet Centers sponsored by MARCO and DARPA.

References

[1] B. Dally, "Power, programmability, and granularity: The challenges of exascale computing," in *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*, 2011.

[2] S. Jeloka, N. B. Akesh, D. Sylvester, and D. Blaauw, "A 28 nm configurable memory (tcam/bcam/sram) using push-rule 6t bit cell enabling logic-in-memory," *IEEE Journal of Solid-State Circuits*, 2016.

[3] M. Kang, E. P. Kim, M. s. Keel, and N. R. Shanbhag, "Energy-efficient and high throughput sparse distributed memory architecture," in *2015 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2015.

[4] P. A. La Fratta and P. M. Kogge, "Design enhancements for in-cache computations," in *Workshop on Chip Multiprocessor Memory Systems and Interconnects*, 2009.

[5] F. Duarte and S. Wong, "Cache-based memory copy hardware accelerator for multicore systems," *Computers, IEEE Transactions on*, vol. 59, no. 11, 2010.

[6] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick, "A case for intelligent ram," *Micro, IEEE*, 1997.

[7] V. Seshadri, Y. Kim, C. Fallin, D. Lee, R. Ausavarungnirun, G. Pekhimenko, Y. Luo, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Rowclone: Fast and energy-efficient in-dram bulk data copy and initialization," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-46.

[8] J. Ahn, S. Yoo, O. Mutlu, and K. Choi, "Pim-enabled instructions: A low-overhead, locality-aware processing-in-memory architecture," in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ser. ISCA '15, 2015.

[9] O. L. Lempel, "2nd generation intel core processor family:intel core i7, i5 and i3," ser. HotChips '11, 2011.

[10] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The splash-2 programs: Characterization and methodological considerations," in *Proceedings of the 22Nd Annual International Symposium on Computer Architecture*, 1995.

[11] J. Jalminger and P. Stenstrom, "A novel approach to cache block reuse predictions," in *Parallel Processing, 2003. Proceedings. 2003 International Conference on*, 2003.

[12] S. V. Adve and M. D. Hill, "Weak ordering&mdash;a new definition," in *Proceedings of the 17th Annual International Symposium on Computer Architecture*, ser. ISCA '90.

[13] J. B. Sartor, W. Heirman, S. M. Blackburn, L. Eeckhout, and K. S. McKinley, "Cooperative cache scrubbing," in *Proceedings of the 23rd international conference on Parallel architectures and compilation*, 2014.

[14] M. Wilkening, V. Sridharan, S. Li, F. Previlon, S. Gurumurthi, and D. R. Kaeli, "Calculating architectural vulnerability factors for spatial multi-bit transient faults," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, 2014.

[15] "Xml parsing accelerator with intel streaming simd extensions 4 (intel sse4)," Intel Developer Zone, 2015.

[16] Q. Guo, X. Guo, Y. Bai, and E. İpek, "A resistive tcam accelerator for data-intensive computing," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-44, 2011.

[17] "Cray Assembly Language (CAL) for Cray X1 Systems Reference Manual. version 1.2. Cray Inc., ," 2003.

[18] S. Kanev, J. P. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G.-Y. Wei, and D. Brooks, "Profiling a warehouse-scale computer," in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ser. ISCA '15.

[19] M. Calhoun, S. Rixner, and A. Cox, "Optimizing kernel block memory operations," in *Workshop on Memory Performance Issues*, 2006.

[20] X. Yang, S. M. Blackburn, D. Frampton, J. B. Sartor, and K. S. McKinley, "Why nothing matters: The impact of zeroing," ser. OOPSLA '11.

[21] T. Carlson, W. Heirman, and L. Eeckhout, "Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation," in *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*, 2011.

[22] S. Li, J. H. Ahn, R. Strong, J. Brockman, D. Tullsen, and N. Jouppi, "Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, 2009.

[23] R. M. Yoo, A. Romano, and C. Kozyrakis, "Phoenix rebirth: Scalable mapreduce on a large-scale shared-memory system," in *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, ser. IISWC '09, 2009.

[24] "Fastbit: An efficient compressed bitmap index technology," "https://sdm.lbl.gov/fastbit/,2015".

[25] "The STAR experiment." http://www.star.bnl.gov/.

[26] J.-J. Wu, Y.-H. Chen, M.-F. Chang, P.-W. Chou, C.-Y. Chen, H.-J. Liao, M.-B. Chen, Y.-H. Chu, W.-C. Wu, and H. Yamauchi, "A large sigma vth vdd tolerant zigzag 8t sram with area-efficient decoupled differential sensing and fast write-back scheme," *Solid-State Circuits, IEEE Journal of*, 2011.

[27] J. H. Lee, J. Sim, and H. Kim, "Bssync: Processing near memory for machine learning workloads with bounded staleness consistency models," in *Proceedings of the 2015 International Conference on Parallel Architecture and Compilation (PACT)*, ser. PACT '15.

[28] L. Yavits, A. Morad, and R. Ginosar, "Computer architecture with associative processor replacing last-level cache and simd accelerator," *IEEE Transactions on Computers*, 2015.

[29] V. Seshadri, K. Hsieh, A. Boroum, D. Lee, M. Kozuch, O. Mutlu, P. Gibbons, and T. Mowry, "Fast bulk bitwise and and or in dram," *Computer Architecture Letters*, 2015.

[30] S. Li, C. Xu, Q. Zou, J. Zhao, Y. Lu, and Y. Xie, "Pinatubo: A processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories," in *Proceedings of the 53rd Annual Design Automation Conference*, ser. DAC '16.

[31] M. Kang, M. S. Keel, N. R. Shanbhag, S. Eilert, and K. Curewitz, "An energy-efficient vlsi architecture for pattern recognition via deep embedding of computation in sram," in *2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2014.

[32] K. Pagiamtzis and A. Sheikholeslami, "Content-addressable memory (cam) circuits and architectures: a tutorial and survey," *Solid-State Circuits, IEEE Journal of*, 2006.