SQRL: Hardware Accelerator for Collecting Software Data Structures

Snehasish Kumar, Arrvindh Shriraman, Viji Srinivasan†, Dan Lin, Jordan Phillips
School of Computer Sciences, Simon Fraser University
† IBM Research

Abstract

Many recent research proposals have explored energy efficient accelerators that customize hardware for compute kernels while still relying on conventional load/store interfaces for data delivery. Emerging data-centric applications rely heavily on software data structures for data accesses which makes it imperative to improve the energy efficiency of data delivery. Unfortunately, the narrow load/store interfaces of general-purpose processors are not efficient for data structure traversals leading to wasteful instructions, low memory level parallelism, and energy hungry pipeline operations (e.g., memory disambiguation).

We propose SQRL[1], a hardware accelerator that integrates with the last-level-cache (LLC) and enables energy-efficient iterative computation on data structures. SQRL integrates a data structure-specific LLC refill engine (Collector) with a lightweight compute array (PE) that executes the compute kernel. The collector runs ahead of the PEs in a decoupled fashion and gathers data objects into the LLC. Since data structure traversals are structured, the collector does not require memory-disambiguation hardware and address generation logic like conventional processors; furthermore it can exploit the parallelism implicit in the data structures.

We demonstrate three types of collectors, a Vector collector, a Key-Value collector, and a BTree collector. On datacentric programs, SQRL eliminates the data structure access instructions and executes the compute kernel on lightweight PEs to achieve 13×–90× energy reduction relative to a a 4-wide out-of-order (OOO) processor. For the kernel portion of the benchmarks, SQRL achieves a performance improvement between 13×–121× relative to the 4-wide OOO processor, and upto 4× compared to even an 8 core system.

1 Introduction

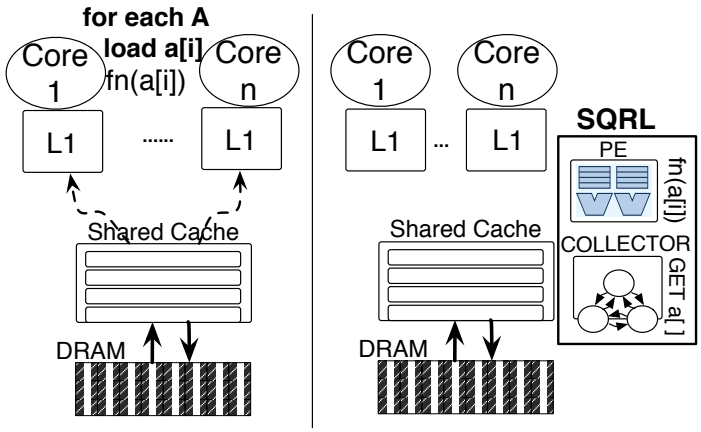
Emerging applications in a diverse set of fields depend on fetching and iterating over large data structures [16]. Interestingly, the main benefit of parallelizing such applications is to access multiple data elements[2] simultaneously to hide long memory latencies. It is unclear whether multicores are a power-efficient solution to achieve the required memory level parallelism (MLP). [3] In addition, due to scalability challenges in the general-purpose processor's memory interface, it may not be possible to achieve the required data rates. Finally, energy efficient access to data structures is also influenced by the specifics of the memory hierarchy and the computation-communication ratio thereby, requiring continuous adaptation.

[1]SQRL is pronounced as Squirrel

[2]The term element is used to refer to objects stored in a data structure.

[3]The term memory level parallelism is used to refer to the number of memory accesses that can be concurrently outstanding in the memory system.

General-purpose processors are inefficient when iterating over data structures and execute many instructions to obtain the required element in the data structure. These instructions waste energy in the instruction fetch stages and increase register pressure. In addition, associative searches of the load/store queue are required and streaming data structure traversals cause many cache misses, which in turn triggers associative searches in the MSHR (miss status registers). Profiling data structure usage in the benchmark applications revealed 9–66% instructions spent in data structure operations. This may lead to significant power consumption as fetching instructions can consume upto 20% of the core's dynamic power and supplying data from the L1 cache can account for 30% of the core's dynamic power [11, Penryn]; the load/store queues alone may consume upto 6% of the dynamic power.



Left: Multicore processor. Right: *SQRL* integrated with the LLC. In multicore, instruction regions show instructions eliminated by *SQRL*. Broken lines indicate data transfers eliminated by *SQRL*.

Figure 1: *SQRL* Overview. Example: Accelerating an iterative computation on an array.

Data access inefficiency poses a significant constraint to researchers focused on energy-efficient compute units. Recent proposals that advocate specialization (e.g., Conservation cores [22] and Dyser [3]), past work in kilo-instruction processors [19] have all had to contend with power hungry load/store units to support the required memory level parallelism [20]. Even heavyweight (e.g., 256 entry) load/store queues sustain only a few (10s of) cache refills [20, 4:] which falls well short of the requirements for iterative kernels that stream over software data structures. Current load/store queue interfaces are unable to make effective use of available memory bandwidth. It is therefore imperative that energy-efficient techniques to achieve the requisite memory level parallelism are explored. A promising avenue of research is

to raise the abstraction of hardware execution to data algorithms (e.g., queries) and enable more efficient execution [8, 24, 6].

We began this research with a similar spirit asking the question whether exposing the traversal of software data structures to hardware would help improve energy efficiency? We propose *SQRL*, an accelerator that integrates with the last-level-cache (LLC) and enables energy-efficient iterative computation on data structures (see Figure 1). Inspired by the decoupled access-execute paradigm [21], *SQRL* partitions iterative computations on data structures into two regions: the data collection and the compute kernel. Figure 1 illustrates a straightforward example of an iterative computation on an array. *SQRL* employs a data structure-specific controller (collector) and a set of processing elements (PEs) near the LLC. The collector traverses the data structure fetching the elements and staging the corresponding cache lines in the LLC until the PE consumes them. The PEs operate on the data objects implicitly supplied by the collector. The collector provides a more scalable memory interface since it is aware of the organization of the data structure and the traversal algorithm.

Overall, *SQRL* eliminates the instructions required for the data structure accesses (bold instructions in Figure 1), eliminates data transfers through the cache hierarchy (broken lines in Figure 1), and runs the compute kernel on the lightweight PE. Since the collector is aware of the data structure and the objects required in each iteration, it runs ahead (limited by only the LLC size) to hide the memory latency and make effective use of the available DRAM bandwidth. *SQRL* lets the application manage complex tasks such as allocation, synchronization and insertion while providing accelerator support for iterative code regions which tend to be energy-inefficient.

We evaluate *SQRL* using a variety of algorithms spanning text processing, machine learning and data analysis. In this paper, SQRL accelerates three types of data structures: dynamic array (Vector), Hash Table and a BTree. Overall, *SQRL* increases memory level parallelism thereby, on average, improving performance by 39×, reducing core energy consumption by 27× and reducing cache hierarchy energy consumption by 98×. In summary, *SQRL* provides architectural support to exploit memory-level parallelism by exposing the abstractions available at the programming language level (i.e., data structures) to hardware, and uses system support to execute hot general-purpose compute kernels on a compute accelerator.

Key Features of *SQRL*

- *SQRL* includes a data-structure specific controller,collector, which exploits parallelism in the data structures to fetch data for compute kernels. *SQRL* is related in spirit to hardware TLB walkers [4]. However, TLB-walkers only perform lookups.
- *SQRL* seeks to exploit semantic information about data structures and is related to prior work on prefetching [18]. However, unlike prefetchers, *SQRL* ensures data is resident in the LLC until the compute finishes processing the element.
- *SQRL* implicitly supplies the data to the computation and eliminates the data structure access instructions. Specialized co-processor approaches [3, 10, 22] and kilo- instruction processors [20] alike can use *SQRL* to increase the memory level parallelism in an energy-efficient manner.
- *SQRL* targets supporting specific software data structures but supports general-purpose compute kernels. Although related to recent work on hardware accelerators for entire datacentric algorithms [7, 23], *SQRL* does not include custom on-chip storage. Instead, *SQRL* uses the LLC for storing the data elements and uses an accelerator-aware line locking to support scratchpad-like fixed-latency accesses.

2 Background and Motivation

Params: Spot[], strike[], rate[], volatility[], time[], type[]
are vectors of length nOptions.
compute_d1(),compute_d2() are re-entrant. Loop is data parallel.

```
for i:= 0 to nOptions do
  d1 = compute_d1(volatility[i],time[i],spot[i],strike[i],rate[i]);
  d2 = compute_d2(volatility[i],time[i]);
  prices[i] = CNF(d1)*spot[i]-CNF(d2)*strike[i] *exp(-rate[i]*time[i])
end for
```

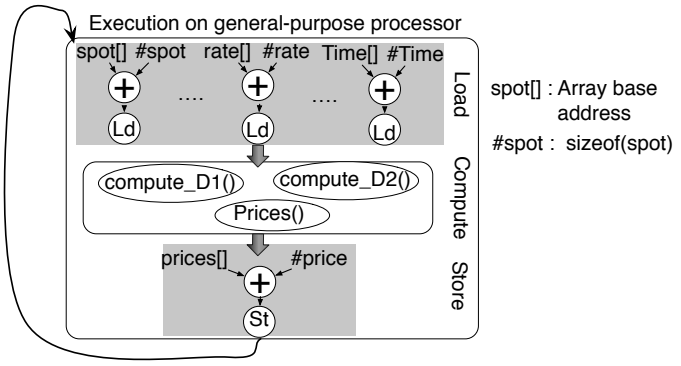


Figure 2: Blackscholes [5] Kernel.

Figure 2 illustrates the compute kernel from *Blackscholes*. We use this as an illustrative example to discuss the overheads associated with storing and retrieving data elements held in software data structures.The program kernel reads in a list of *"options"* data stored across 6 array of structs implemented using C++ STL vectors (`spot[]`,`strike[]`,`rate[]`,`volatility[]`,`time[]`, `and type[]`) and calculates the pricing (`prices[]`). Figure 2 shows the execution on a typical general-purpose processor, where the kernel is organized into load-compute-store slices. Every iteration requires 6 `add` and 6 `load` instructions for generating the load slice and 1 `add` and 1 `store` instruction. Note that even a simple stride 0 iteration over these vectors requires many CPU instructions that increase instruction window pressure. In addition, for architectures with limited registers (e.g., x86) many stack operations are introduced to preserve the address of the vectors (these overheads are not shown above). Resource limitations such as load/store queue entries and MSHRs further limit the parallelism that can be exploited. Increasing the hardware resources for the low-level memory interface in a power-efficient manner is a significant challenge. For example, the Penryn processor core expends 30% of core dynamic power in the data supply from the L1 cache. Scaling the MSHR and load/store queues by 2× would increase power consumption of the processor core by 15% while achieving only 2–4% improvement in performance. Mining the MLP from programs require significant increase in overall instruction window. Finally, the inefficiency of general-purpose processors

is more acute with other data structures such as key-value based hash tables, and pointer-based trees/graphs.

Table 1: Data structure usage in Datacentric Applications

Benchmark	% Time	% DS Ins.	% INT	% BR	% MEM
Blackscholes	99.7	9	13.3	13.5	26.6
Recomm.	44.6	65.9	26.9	6.4	51.9
Datacubing	34.1	14.3	32.8	12.3	54.7
Text Search	64.7	30.9	32.6	14.1	53.1
Hash Table	25.9	34.7	34.1	17.0	48.7
BTree	100	63.7	32.7	15.7	51.5

The potential for data structure acceleration in our benchmarks is quantified in Table 1. The % execution time and the breakdown of instructions in the benchmarks are analyzed. Overall in our application suite, between 25—100% of the time is spent in operations iterating over the data structure. The number of instructions spent on data structures can be significant as well; for instance in the Recommender benchmark, 66% of the dynamic instructions are related to data structure operations and contribute to the dynamic power consumption. In data structures such as HashTable and BTree, the traversal's control flow is determined by the value of the data element and involves many branches (15.7%-17% of total instructions). Although, the condition checks are simple arithmetic checks (<,=,>), conventional processors require dynamic instructions to evaluate the condition itself. Recent research results [15] have demonstrated that such checks can constitute upto 2× performance overhead. Interestingly, the computation kernels themselves possess loop-parallelism and can reap performance gains from exposing the memory level parallelism in the data structure. Finally, compute kernels in different applications may exhibit significant diversity in the instruction breakdown and latencies per iteration, even when accessing the same type of data structure (e.g, vector). This indicates a need for flexibility when fetching the data structure; for instance when computational kernel latencies are long, the data fetch can be throttled to save memory bandwidth.

Summary

- Programs spend significant time performing iterative computations on data structures. Overall efficiency can be improved by removal of data structure "access" instructions.
- While data structures possess implicit parallelism, general-purpose processors need a significant increase in power-hungry resources (e.g., instruction window, load/store queue) to mine this parallelism. Scalable approaches to extract memory level parallelism need to be explored.
- The hardware memory interface needs to provide abstractions to exploit information about data structures. Moreover, a flexible interface is required as the compute kernels within the applications vary.

3 *SQRL*: Collecting data structures

Efficient iterative computation on data structures inherently requires the compute kernels to be collocated with the data storage. *SQRL* is a step in that direction, and targets energy-efficient acceleration of iterative computation on data structures. A key design question is where in the memory hierarchy should we locate *SQRL*? The compute kernel software functions from various applications have varied data processing latency making it hard to manage scratchpad buffers. Therefore, we choose to integrate *SQRL* with the last-level-cache (LLC). *SQRL* leverages the last-level cache (LLC) to stage the data fetched by streaming ahead of the compute kernel. The large LLC enables *SQRL* to achieve high memory level parallelism. Moving the computation closer to the LLC eliminates power hungry data transfers through the on-chip caches. Finally, the LLC also enables us to efficiently communicate results between the accelerator and the CPU.

3.1 Overview

Figure 3 shows the overall architecture of *SQRL*. *SQRL* consists of three major components: i) the *collector* is a data-structure specific cache refill engine . It includes a few configuration registers to specify the data structures needed for each iteration in the computation. ii) the object cache (*OBJ-$*) holds the data elements needed by a tile of the compute iteration. and iii) the processing array (PE) which consists of a set of lightweight execution lanes running the unrolled compute kernel. The PEs have no address generation logic and operate on the data residing in the *OBJ-$*; the *OBJ-$* is auto-filled by the collector to drive the computation. Since the PE and the collector run asynchronously they each maintain a cursor. The PE's cursor indicates the loop iteration being executed while the collector's cursor indicates the index of the elements being fetched.

We illustrate *SQRL*'s operation based on the *Blackscholes* example from Figure 2. The software (programmer or compiler) marks the iterative computation in the application and indicates the load slice and store slice. The load slice (and store slice) information include information on the type of data structures (arrays in this case), base addresses of the arrays and the object indices needed within an iteration. In *Blackscholes*, the load slice consists of the following vectors, spot[],strike[],rate[],volatility[], time[] and type[] and the store slice consists of price[]. The compute kernel in *Blackscholes* employs a unit stride array, and the i^{th} iteration reads the i^{th} element from these arrays. The markers are used to initialize *SQRL*'s hardware components as shown in Figure 4.

The overall computation proceeds in wavefronts which is a by-product of both loop tiling (to ensure that data fits in the *OBJ-$*) and loop unrolling (to exploit all the available PEs (8 in *SQRL*)). Each PE executes a single iteration of the compute kernel; the loop is unrolled by the number of PEs. A wavefront is initiated by the collector which pushes data into the *OBJ-$* (❶) to commence execution (❷). Each iteration in *Blackscholes* (Figure 4) reads 5 vector elements of type *float* (spot[],strike[],rate[],volatility[], and time[]) and one vector of type *char* (type[]) and writes price[] which is of type *float*. Overall, each iteration requires space for 4×5 (floats) + 1 (char) + 4 (float) = 25 bytes. Given a 1 KB *OBJ-$* it can hold data for 40 iterations of the *Blackscholes* kernel (a loop tile) which proceeds in waves of 8 iterations (the unroll parameter). The collector runs asynchronously (❸), fetches the cache lines corresponding to the data structure objects from each of the vectors into the LLC. The collector maintains its own cursor that runs independent of the compute kernel. This enables the collector to continue fetching even after a PE stall due to possible

resource constraints. The primary resource constraints for collector are the LLC cache size and available memory bandwidth. In ❹ the collector locks down each of the lines as they are fetched into the LLC to prevent them from being evicted and guarantees the PE fixed-latency accesses.

In ❺ when the PE completes a loop iteration it checks the cursor and decrements the Ref.# field. When the Ref.# counter reaches 0, the cache line is unlocked and can be replaced. If the collector stalls due to the LLC filling up with locked lines, the PE releases lines as it completes the iterations which implicitly restarts the collector. Finally, to ensure deadlock-free progress for the CPU cores we ensure that at least one way per LLC set remains unlocked.

3.2 *SQRL* vs. Existing approaches

We qualitatively compare *SQRL* against OOO (Out-of- Order) cores, Multicores, decoupled access-execute architectures [21], and hardware prefetching. Decoupled architectures use compiler support to separate memory accesses from the rest of the execution into a separate thread strand to run ahead to fetch the data structure. The memory strand executes on a separate pipeline and communication with the compute strand includes the data values and branch decisions [21]. While software prefetching has the potential to improve memory level parallelism for certain classes of applications in which data access patterns can be statically determined, it still requires power-hungry instruction window and front-end resources similar to an out-of-order processor, and so we do not discuss software prefetching separately below.

We use five design features in our comparison. i) **Memory level parallelism**: *SQRL*'s memory level parallelism is limited by the parallelism available in the data structure (i.e., linked-list vs array), the number of MSHRs at the LLC, and the size of the LLC. Multicores and decoupled architectures are scalable, but require many pipelines and power-hungry register files to support the active threads. OOO cores need a significant increase in the overall instruction window size to exploit MLP and hardware prefetching may require large history tables. ii) **Data-structure instructions**: *SQRL* completely eliminates data structure instructions. All other approaches need to execute data structure instructions and expend energy in the processor front-end and load/store interface. iii) **Fixed-latency Loads:** *SQRL* does not invoke the compute kernel in an iteration until the requisite data is in the LLC and enables low-complexity pipelines that can assume fixed latency for all instructions. Decoupled architectures include similar efficiency by isolating non-deterministic latency cache-misses in the access pipeline. OOO Processor pipelines need to include significant complexity (e.g., pipeline interlocks) to deal with non-deterministic latencies that arise from cache misses. The major concern with prefetching is timeliness of data fetch which introduces non-determinism. iv) **Big data-structures:** *SQRL* uses the LLC to hold data fetched ahead of their usage. Conventional OOO and multicore are unable to are unable to discover the parallelism inherent in large data structures due to limitations in the instruction window and number of thread strands. v) **Value-dependent traversal**: There exists many data structures (e.g., binary search tree), where the traversal of the data structure is control dependent on the data element

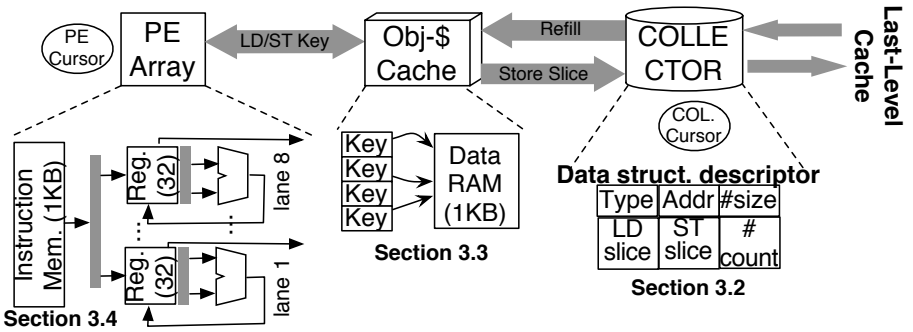
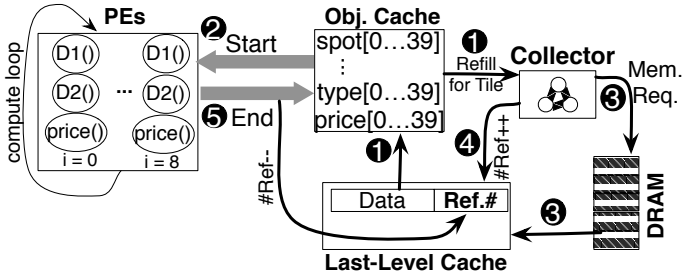


Figure 3: *SQRL* Architecture Overview

```
begin blackscholes
   // Initialize collectors
   c_spot = new coll(LD,&spot,FP,#length,0,VEC)
    ...
   c_time = new coll (LD,&time,FP,#length,0,VEC)
   // Run collectors in step.
   group.add (c_spot…c_time);

  // Unroll to # of PEs. Tile based on Obj.$ size.
  start(kernel(),#iterations)
end
```



Blackscholes Execution. Left: Blackscholes kernel modified to work with *SQRL*. ❶ Loop tile refill from LLC triggered every 40 iterations as *OBJ-$* can hold 40 iterations worth of data. ❷ Execution proceeds in tiles. In Blackscholes a tile of 40 iterations executed on the PE as 8 unrolled iterations at a time. ❸ Memory request issued by collector to fetch the cache blocks corresponding to the data structure. LLC refills from memory. ❹ collector locks line by setting Ref.# to number of elements needed in the iteration. ❺ When the compute iteration completes, the PE decrements the reference count to release the cache lines.

Figure 4: Blackscholes Execution on *SQRL*.

value. OOO, Multicore, and Decoupled Architectures require instructions to evaluate the value- dependent condition. Recent research [15] has shown that evaluating such branch conditions expend significant energy in many applications. *SQRL* exploits data-structure knowledge to evaluate the conditional checks in hardware and integrates it with the data accesses to fully exploit the parallelism inherent in the data structure.

4 Evaluation

Table 2: System parameters

Cores	up to 8 cores. 2.0 GHz, 4-way OOO, 96 entry ROB, 6 ALU, 2 FPU, INT RF (64 entries), FP RF (64 entries) 32 entry load queue, 32 entry store queue
L1	64K 4-way D-Cache, 3 cycles, 32K I-Cache[4]
LLC	4M shared 16 way, 4 Banks, 20 cycles Directory-based MESI
Main Memory	4ch,open-page, DDR2-400, 16GB 32 entry cmd queue,
Energy Params	L1 (100pJ Hit). LLC (230pJ hit) [12] L1-LLC link (6.8pJ/byte [2, 14]), LLC-DRAM (62.5 pJ per byte [9])
***SQRL* Components**	
	8 PEs, 2.0 Ghz, 4 stages in-order. 1 FPU and 1 ALU (per PE) Instruction buffer (1KB, 256 entries)
	Obj-$ (1KB fully-assoc. sector cache, 32 tags). 1cycle INT RF: 32 entries, FP RF: 32 entries

The out-of-order core is modeled using Macsim [1], the cache controllers and on-chip memory hierarchy using Ruby from GEMS [13] and main memory using DRAMsim2 [17]. The host system is based on the Intel Penryn (45nm) architecture. McPAT [11] is used to derive the energy consumption.

4.1 *SQRL* vs. OOO Core Performance

Result: *SQRL improves performance on average by 39× compared to a general purpose processor. Maximum speedup (*BTree*): 121×. Minimum speedup (*Recommender*): 13×. SQRL utilizes available memory bandwidth to prefetch and reduce average memory access latency which further improves performance over parallel execution on the PEs.*

The speedup obtained for different applications is dependent on the limiting hardware resource on the OOO processor. With an application such as *DataCubing*, the kernel code is small (only 16 static instructions) and the 59× performance improvement obtained is the result of removing the data structure instructions. In contrast, *Blackscholes* includes a complex kernel with many long latency floating point instructions and the OOO processor is hindered by these instructions filling up the ROB before the independent memory instructions can be discovered. We observe that even with a 256-entry instruction window the OOO processor sustains at most 7 concurrent cache misses. *SQRL* improves performance by decoupling the data structure accesses from the compute kernel enabling it to slip far ahead.

In *BTree* and *HashTable*, accessing the data structures constitute the core of the application. The PEs do not perform any work and these applications serve to highlight the overheads of the collector hardware and its ability to discover the implicit MLP in the data structure. With an application such as *BTree* although

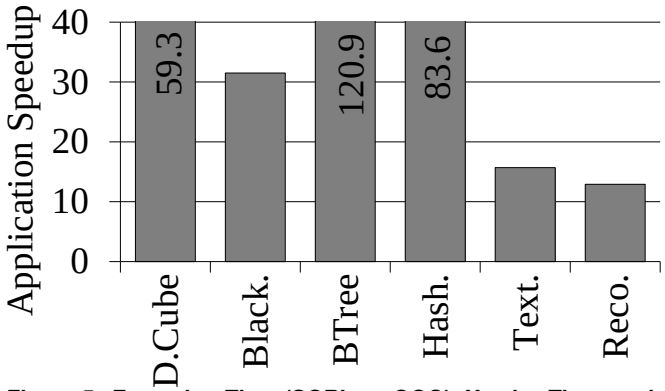


Figure 5: Execution Time (SQRL vs OOO). Y-axis: Times reduction in execution cycles with SQRL. Higher is better.

traversing between tree-levels is sequential there does exist implicit parallelism when searching for the range within the internal node. The OOO processor encounters performance overhead due to the serialized pointer dereferencing instructions and inability to mine the MLP across pointer dereferences. *SQRL* improves performance by fetching in parallel the data elements for the range search and eliminating the pointer dereference instructions. With *HashTable*, the performance improvements are primarily due to eliminating the data movement from the on-chip hierarchy and eliminating the search function from the OOO core.

In *TextSearch* and *Recommender* we observe only 16× and 13× reduction in cycles, respectively. In *TextSearch*, unlike the *HashTable*, the search kernel runs on the PE; both the `ExactMatch` and `HammingDist` functions require many instructions and the fraction of instructions wasted on data structures is small. Nevertheless, *SQRL* improves performance by discovering the MLP. In the case of *Recommender* the kernel processes a sparse matrix and the kernel execution is predicated on the value of each element in the matrix. The lack of support in *SQRL* for conditional execution of a kernel limits the performance improvement. On the OOO processor, the kernel is executed only by ≃2% of the loop iterations i.e., 98% of the loop iterations evaluate the condition and short-circuit the kernel. With *SQRL* an execution lane that does not execute the kernel has to simply idle and wait for other PEs executing the kernel. PE utilization can be improved by dynamically scheduling other iterations on the idle PEs, but this requires additional logic to manage the loop cursor.

For a system, with multiple OOO cores we can improve the performance of the benchmarks by partitioning the work and spawning individual threads on each core. However, though there is an increase in performance, the overall power consumption will increase due added complexity. We use parallel (8 thread) version of *Blackscholes* and *Recommender* to illustrate our case; each core in the multicore is the same as our baseline OOO. At 8 threads, *Blackscholes* demonstrated a 7.3× improvement in performance, however energy consumption increases by 42%. SQRL improves performance by 4.3× over even the 8 thread version by extracting more MLP, while reducing energy by 29×. With *Recommender* the parallel version demonstrates near linear speedup (8×) with a 24% increase in energy consumption. SQRL improves performance by 60% over the multicore while reducing

energy by 18×.

4.2 Energy Reduction: *SQRL* vs OOO

Result 1: *SQRL is able to reduce overall energy in the PEs relative to the OOO core by 27×.*
Result 2: *SQRL's shallow memory hierarchy helps reduce on-chip cache access energy by 98×.*

Core vs. PEs: Figure6 shows the reduction in dynamic energy when executing the kernels on the lightweight PE. Compared to the OOO, *SQRL* saves energy by eliminating data structure instructions. *SQRL*'s PE also uses a simple pipeline which requires minimal energy for instruction fetch (1KB instruction buffer vs I-cache on OOO) and does not include any of the complex load/store interfaces. *BTree* and *HashTable* are not included in Figure6 as they do not include any work in the compute kernel and simply traverse the data structure. On *DataCubing* we see a 109× reduction in energy as the data structure instructions (completely eliminated in *SQRL*) constitute a major part of all the instructions on the OOO version of the program. With *Blackscholes* and *TextSearch* which have compute intensive kernels, the primary energy reduction is due to the simpler PE pipeline compared to the OOO. The main benefit of *SQRL* is being able to extract the MLP to improve performance without requiring the PE to sacrifice its energy efficiency. A 13× reduction is observed for *Recommender* which is primarily due to un-availability of option such as power-gating when the execution lane in the PE idles, even if the kernel does not need to be executed.

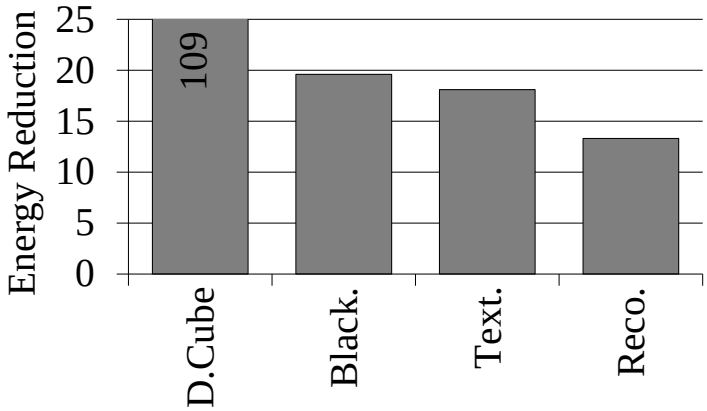


Figure 6: Dynamic energy reduction (Unit: × Times) in PEs vs. OOO core. *BTree* and *HashTable* only use the collector to traverse the data structure and do not use the PE. Higher is better.

5 Summary

While modern applications tend to have diverse compute kernels, they tend to use a set of common data structures. We have focused on exploiting data structure information to provide compute accelerators [22, 3, 10] and kilo-instruction processors [20] with energy-efficient interfaces to the memory hierarchy. We developed SQRL, a hardware accelerator that integrates with the LLC and supports energy efficient iterations on software data structures. SQRL includes cache refill engines that are customized for specific software data structures. This increases the memory level parallelism, effectively hides memory latency, eliminates energy hungry data structure instructions from the processor core, and energy-hungry transfers over the memory hierarchy

References

[1] Macsim : Simulator for heterogeneous architecture - https://code.google.com/p/macsim/.

[2] D. Albonesi, K. A, and S. V. *NSF workshop on emerging technologies for interconnects(WETI)*, 2012.

[3] J. Benson, R. Cofell, C. Frericks, C.-H. Ho, V. Govindaraju, T. Nowatzki, and K. Sankaralingam. Design, integration and implementation of the DySER hardware accelerator into OpenSPARC. *High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on*, pages 1–12, 2012.

[4] R. Bhargava, B. Serebrin, F. Spadini, and S. Manne. Accelerating two-dimensional page walks for virtualized systems. In *PROC of the 13th ASPLOS*, 2008.

[5] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: characterization and architectural implications. In *PROC of the 17th PACT*, 2008.

[6] L. Carrington, M. M. Tikir, C. Olschanowsky, M. Laurenzano, J. Peraza, A. Snavely, and S. Poole. An idiom-finding tool for increasing productivity of accelerators. In *ICS '11: Proceedings of the international conference on Supercomputing*. ACM Request Permissions, May 2011.

[7] S. R. Chalamalasetti, K. Lim, M. Wright, A. AuYoung, P. Ranganathan, and M. Margala. An FPGA memcached appliance, 2013.

[8] E. S. Chung, J. D. Davis, and J. Lee. LINQits: big data on little clients. In *PROC of the 40th ISCA*, 2013.

[9] B. Dally. Power, programmability, and granularity: The challenges of exascale computing. In *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 878–878, 2011.

[10] S. Gupta, S. Feng, A. Ansari, S. Mahlke, and D. August. Bundled execution of recurring traces for energy-efficient general purpose processing. In *MICRO-44 '11: Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM Request Permissions, Dec. 2011.

[11] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi. McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In *PROC of the 42nd MICRO*, 2009.

[12] S. Li, K. Chen, J. H. Ahn, J. B. Brockman, and N. P. Jouppi. CACTI-P: Architecture-level modeling for SRAM-based structures with advanced leakage reduction techniques, 2011.

[13] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacet's general execution-driven multiprocessor simulator (gems) toolset. *SIGARCH Comput. Archit. News*, 33(4):92–99, Nov. 2005.

[14] N. Muralimanohar, R. Balasubramonian, and N. Jouppi. Optimizing NUCA Organizations and Wiring Alternatives for Large Caches with CACTI 6.0. In *PROC of the 40th MICRO*, 2007.

[15] A. Parashar, M. Pellauer, M. Adler, B. Ahsan, N. Crago, D. Lustig, V. Pavlov, A. Zhai, M. Gambhir, A. Jaleel, R. Allmon, R. Rayess, S. Maresh, and J. Emer. Triggered instructions: a control paradigm for spatially-programmed architectures. In *PROC of the 40th ISCA*, pages 1–12, Apr. 2013.

[16] P. Ranganathan. From Micro-processors to Nanostores: Rethinking Data-Centric Systems. *Computer*, 44(January):39–48, 2011.

[17] P. Rosenfeld, E. Cooper-Balis, and B. Jacob. Dramsim2: A cycle accurate memory system simulator. *Computer Architecture Letters*, 10(1):16 –19, jan.-june 2011.

[18] A. Roth and G. S. Sohi. Effective jump-pointer prefetching for linked data structures. In *PROC of the 26th ISCA*, 1999.

[19] K. Sankaralingam, R. Nagarajan, R. McDonald, R. Desikan, S. Drolia, M. S. Govindan, P. Gratz, D. Gulati, H. Hanson, C. Kim, H. Liu, N. Ranganathan, S. Sethumadhavan, S. Sharif, P. Shivakumar, S. W. Keckler, and D. Burger. Distributed Microarchitectural Protocols in the TRIPS Prototype Processor. In *PROC of the 39th MICRO*, 2006.

[20] S. Sethumadhavan, R. McDonald, D. Burger, S. S. W. Keckler, and R. Desikan. Design and Implementation of the TRIPS Primary Memory System. In *Computer Design, 2006. ICCD 2006. International Conference on*, pages 470–476, 2006.

[21] J. E. Smith. Decoupled access/execute computer architectures. In *25 years of the international symposia on computer architecture (selected papers)*, 1998.

[22] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. B. Taylor. Conservation cores: reducing the energy of mature computations. In *PROC of the 15th ASPLOS*, 2010.

[23] L. Wu, R. J. Barker, M. A. Kim, and K. A. Ross. Navigating Big Data with High-Throughput, Energy-Efficient Data Partitioning .

[24] L. Wu, M. Kim, and S. Edwards. Cache Impacts of Datatype Acceleration. *IEEE Computer Architecture Letters*, 11(1):21–24, Apr. 2012.