DASX: Hardware Accelerator for Software Data Structures

Snehasish Kumar, Naveen Vedula, Arrvindh Shriraman and Vijayalakshmi Srinivasan†
School of Computing Science, Simon Fraser University and †IBM Research
{ska124, nvedula, ashriram}@sfu.ca and viji@us.ibm.com

ABSTRACT

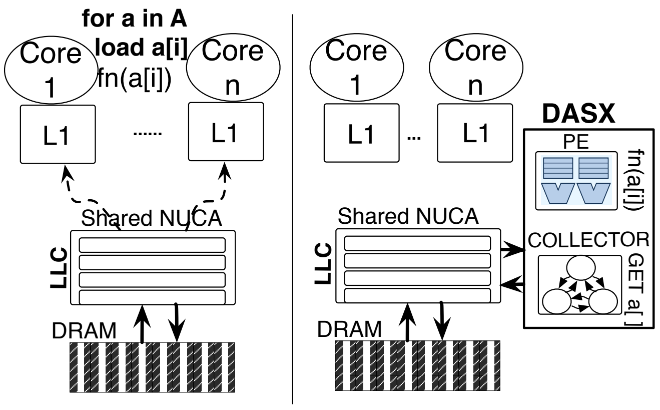
Recent research [3, 37, 38] has proposed compute accelerators to address the energy efficiency challenge. While these compute accelerators specialize and improve the compute efficiency, they have tended to rely on address-based load/store memory interfaces that closely resemble a traditional processor core. The address-based load-/store interface is particularly challenging in data-centric applications that tend to access different software data structures. While accelerators optimize the compute section, the address-based interface leads to wasteful instructions and low memory level parallelism (MLP). We study the benefits of raising the abstraction of the memory interface to data structures.

We propose DASX (Data Structure Accelerator), a specialized state machine for data fetch that enables compute accelerators to efficiently access data structure elements in iterative program regions. DASX enables the compute accelerators to employ data structure based memory operations and relieves the compute unit from having to generate addresses for each individual object. DASX exploits knowledge of the program's iteration to i) run ahead of the compute units and gather data objects for the compute unit (i.e., compute unit memory operations do not encounter cache misses) and ii) throttle the fetch rate, adaptively tile the dataset based on the locality characteristics and guarantee cache residency. We demonstrate accelerators for three types of data structures, Vector, Key-Value (Hash) maps, and BTrees. We demonstrate the benefits of DASX on data-centric applications which have varied compute kernels but access few regular data structures. DASX achieves higher energy efficiency by eliminating data structure instructions and enabling energy efficient compute accelerators to efficiently access the data elements. We demonstrate that DASX can achieve 4.4× the performance of a multicore system by discovering more parallelism from the data structure.

1. Introduction

Accelerators are gaining importance as energy-efficient computational units due to the breakdown of Dennard scaling. However, these accelerators continue to rely on traditional load/store interfaces to access data. For example, recent proposals that advocate specialization (e.g., Conservation cores [38] and Dyser [3]), as well as past work in kilo-instruction processors [32] have all had to contend with power hungry load/store units [33]. Even large (e.g., 256 entry) load/store queues can sustain only a few (10s of) cache refills [33] which falls short of the requirements for kernels that stream over large data structures. A promising avenue of research is to improve the efficiency of data accesses via hardware acceleration using techniques such as specialized ISA extensions [40], and customized hardware (e.g., SQL-like language constructs [12] and hash lookups [19]).

ICS'15, June 8–11, 2015, Newport Beach, CA, USA.
Copyright © 2015 ACM 978-1-4503-3559-1/15/06 ...$15.00.
DOI: http://dx.doi.org/10.1145/2751205.2751231.



Left: Multicore processor. Right: DASX integrated with the shared LLC (NUCA tiles not shown). In multicore, instruction regions show instructions eliminated by DASX. Broken lines indicate data transfers eliminated by DASX.
Figure 1: DASX Overview. Accelerating an iterative computation on an array.

A key application domain seeking to use compute accelerators are data centric applications that perform iterative computation over large data structures [30]. Unfortunately, the conventional address-based load/store interface does not scale to exploit the available memory bandwidth and expends significant energy on instructions required to fetch the objects from within the data structure. The main benefits of parallelizing datacentric applications come from fetching multiple data objects simultaneously and hiding long memory latencies. Parallelization with threads can increase the number of memory accesses issued and improve the MLP, however as our evaluation shows, increasing MLP (memory-level-parallelism) using the same thread as the compute kernel is energy inefficient.

Our work makes the key observation that by raising the abstraction of the memory interface from addresses to object indices (or keys), we can enable a high performance (i.e., more MLP) and energy efficient data-structure specific memory interface for compute accelerators. We propose DASX (Data Structure Accelerator), a hardware accelerator that integrates with the LLC and enables energy-efficient iterative computation on data structures (see Figure 1). Supporting iterative computation requires DASX to adapt to the locality requirements of the compute kernel and effectively supply data as the compute kernel streams over the entire dataset. Inspired by the decoupled access-execute paradigm [36], DASX partitions iterative computation on data structures into two regions: the data collection and the compute kernel. Figure 1 illustrates an example of an iterative computation on an array. Since DASX removes the data structure instructions from the compute kernel, it also eliminates complex address generation logic from the compute unit which instead interfaces with memory by issuing loads and stores to object keys (unique identifiers for each object). It uses specialized hardware to translate object keys to memory addresses, and issue the requests.

DASX relies on the observation that the MLP available in data structures is independent of ILP and TLP limitations. DASX employs a novel data structure-specific controller (Collector) to traverse the data structure, mine the MLP, and stage the corresponding cache lines in the LLC until the compute unit consumes the data. In this paper, our compute unit is a PE (processing element) array, a set of light-weight in-order cores that execute data parallel loop iterations. The PEs operate on the data objects in the Obj-Store supplied by the Collector and do *not* generate addresses. DASX is aware of the compute iterations and appropriately tiles the data to fit in the LLC to prevent thrashing. The Collector is aware of the organization of the data structure and runs ahead of the compute kernel so as to prefetch the objects to hide memory latency. Overall, DASX eliminates the instructions required for the data structure accesses (bold instructions in Figure 1), eliminates data structure transfers over the cache hierarchy (broken lines in Figure 1), and runs the compute kernel on the lightweight PE.

We evaluate DASX using a variety of algorithms spanning text processing, machine learning and data analytics, and accelerate three types of data structures: Vector, Hash Table, and a BTree. Relative to a multicore system (2 cores, 4 threads per core), DASX improves performance by 4.4×, reduces core energy consumption by 27× and reduces cache hierarchy energy by 7×. Overall, DASX has the potential to improve the efficiency of language-level data structures and idioms such as *list* and *map* by exploiting the parallelism implicit in the data structures that is often lost in the implementation due to the address-based memory interface of the processor core that requires many instructions to fetch a single object. Our contributions are:

- DASX exposes MLP available in data structures and uses specialized hardware, the Collector, to access data using object keys. Other specialized programmable co-processor [3, 18, 37, 38] approaches can also use the Collector to achieve high MLP and energy efficiency.
- DASX targets iterative computation and leverages data structure-awareness to run far ahead of the compute kernel and prefetches data to improve performance.
- DASX exploits information about compute iterations to guarantee that the data is cache-resident until the kernel finishes processing and efficiently drives the compute kernel based on the presence of data in the cache.

2. Background and Motivation

In this section we discuss the overheads associated with software data structures using *Blackscholes* [5] with unit stride vectors as an example. Figure 2 illustrates the compute kernel from *Blackscholes*.

```
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
```

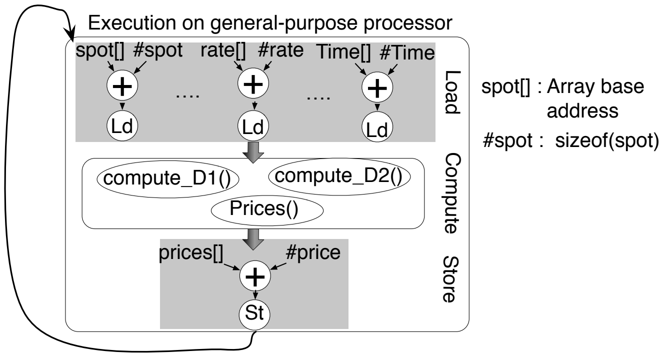


Figure 2: *Blackscholes* [5] Kernel.

The program kernel reads in a list of *"options"* data stored across six C++ STL vectors (`spot[], strike[], rate[], volatility[], time[], and type[]`) and calculates the pricing (`prices[]`). Note that the use of vector data structures gives rise to easily accessible MLP, but the compute kernel itself could include dependencies that limit parallelization and SIMDization. Figure 2 also shows the execution on a typical general-purpose processor. Every iteration requires 6 `add` and 6 `load` instructions (for generating the load slice), and 1 `add` and 1 `store` instruction. As the compute kernel streams over the 6 data structures it is also imperative to ensure that i) the elements are fetched into the cache ahead of the iteration that needs them and ii) the elements continue to reside in the cache until the corresponding iteration finishes processing them. The former requires the processor to aggressively fetch data into the caches while the latter requires the processor to fetch conservatively so as to not thrash the caches. Finally, the memory interface needs to be flexible to enable access to multiple independent data structures (6 read-only data structures and 1 read/write) and possibly multiple elements from the same data structure. The energy inefficiencies of general-purpose processor tend to be more acute with complex key-value based hash tables and pointer- based trees/graphs. Even a simple strided iteration incurs significant address generation overheads.

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
2D-Datacube	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

Table 1: Data structure usage in Data-centric Applications

Table 1 shows the % execution time, % data structure instructions and the breakdown of instructions[1] in the benchmarks. Overall in our application suite, between 25—100% of the time is spent in operations iterating over the data structures. The number of instruc-

[1]We used gcc -OO optimization to enable us to identify STL library calls in the binary and use pintool to profile. Our performance evaluation builds with -O2.

tions spent on data structures can be significant (up to 66% in *Recommender*) and contribute to dynamic power consumption. In data structures such as *HashTable* and *BTree*, the value of the data element determines the control flow and involves many branches (15.7%-17% of total instructions). Condition checks can constitute up to a 2× overhead in data traversals [27]. Additionally, compute kernels in different applications may exhibit significant diversity in temporal and spatial locality even when accessing the same type of data structure (e.g, vector) which requires the data structure accesses to be carefully managed to ensure optimal cache behaviour. The parallelism available from the data structure can differ from the parallelism available in compute kernel. Consider a reduction operation ($\Sigma_{i=1}^{N} A[i]$), even though the compute region's parallelism is limited, the data structure itself is embarrassingly parallel and exploiting the available MLP will improve overall speedup. To summarize:

- Programs may spend significant time performing iterative computations on data structures. Overall, removing the data structure instructions will improve energy efficiency.
- Even if compute parallelism is limited(e.g. reduction), we can exploit the MLP to reduce the average memory access latency and improve performance. We need to develop scalable energy efficient techniques to extract the available MLP.
- The implementation of the memory interface needs to be flexible and adapt to compute kernels with varied temporal and spatial locality.

3. DASX: Data Structure Accelerator

DASX consists of two components (Figure 3): i) An array of processing elements (PEs), each PE is an in-order core running the compute kernel (§ 3.1) and ii) a data-structure specific refill engine, the Collector (§ 3.2). The PEs only access data elements using data structure based memory operations. The PEs share a common object cache, Obj-Store. The Obj-Store (object-cache) holds the data elements needed to execute and is explicitly managed by the Collector which ensures the data structure memory operations do not miss in the Obj-Store. The Collector translates object keys to address locations and manages data fetches needed by the compute kernel. The PEs do not perform address generation and do not require miss-handling logic. We begin with a description of the data structure memory operations to help the reader understand the design requirements of the Collector.

3.1 Processing Elements

Table 2: Special instructions and registers

Special Registers		
%CUR	contains the compute cursor (the loop trip count)	
%BAR	synchronization barrier across PE array	
Special Instructions		
Type	Instruction	Description
Mem	LD [Key],%R	load key from Obj-Store.
	ST [Key],%R	store key from Obj-Store.
Loop	NEXT	Advance cursor and trigger a refill of the Obj-Store.

The PE array consists of a set of light-weight 32 bit in-order, 4-stage integer/floating-point pipelines. All the execution pipelines share both the front-end instruction buffer (256 entries) and the back-end Obj-Store. The memory operations only access the data in the Obj-Store which is loaded in bulk by the Collector. Each execution pipeline has 32 integer and 32 floating point registers. Within our framework each PE's pipeline runs an instance of the compute kernel; most importantly each PE has its own PC and is capable of taking a different branch path within the iteration. Table 2 shows the special instructions added to support DASX. The load and store operations access data structures using a unique identifier (the key) and the PE's cursor indicates the loop trip count of the compute kernel. All PEs sync up at a barrier after performing the set of iterations statically assigned to the PE array. The PEs do not communicate with each other except via loads/stores to the data structure. The number of execution pipelines depends on the bandwidth of the shared front end and Obj-Store; with the MLP extracted by the Collectors, we find that limited compute parallelism(4—8 PEs) is sufficient (see § 4.5). The key to DASX's performance improvement is the MLP exploitation using the Collector.

Key-based loads and stores (PE↔Memory interface)

DASX raises the abstraction of the memory interface from addresses and the compute PE accesses elements in the data structure through *keys*. The *key* is an index to identify a particular element in the data structure. A unique key value is associated with every element in the data structure. Some of the key values may be implicit (e.g., array index) while others may be explicit (e.g., hash table). The unique identification provides a name space with which the compute units can indicate the load/store slices to the Collector. With containers such as arrays/vectors, *key* refers to the index of the element. For a hash table, *key* refers to the search key and with structures like the BTree, the *key* is a unique integer. All the address generation logic is centralized in the Collector and is data-structure specific. Using keys to access data elements allows DASX to eliminate data structure instruction, specialize the address generation logic to reduce overhead and extract memory level parallelism implicit in the data structure. Further benefits include the ability for DASX to coalesce and amortize the cost of address generation and virtual-to-physical translation for multiple objects in the same data structure. The key-based memory operations access data from the Obj-Store.

Obj-Store

The Obj-Store shared between the PEs serves as a data store for elements from the data structure. The PE accesses the Obj-Store using key-based memory instructions. When starting a set of iterations the Collector performs a bulk refill of the Obj-Store. The Obj-Store does *not* encounter misses and is explicitly managed by the Collector. The main utility of the Obj-Store is to provide a flexible interface to access the data structure elements without requiring the PEs to generate addresses. In addition, the Obj-Store also captures the locality within an iteration and filters small width (4-byte) accesses from the LLC to save energy. Obj-Store is organized as a fully associative decoupled sector-cache [34] with a sector size of 4 bytes (8 sectors/line) [24] and 32 tag entries (Figure 4). The decoupled sector organization provides the following benefits : i) by varying the number of sectors allocated data objects of varying length are supported ii) the small sector sizes (4 bytes) support structures of primitive type (e.g., array of floats), iii) tag overhead is minimized. The tag stores the Collector id dedicated to the data structure and up to 8 adjacent keys. When the compute iteration completes execution, the Obj-Store flushes the dirty objects back to the LLC triggering any required coherence operations. Writebacks use the LLC backpointer in the Obj-Store entry (Figure 4); the backpointer is set when the Collector loads the Obj-Store.

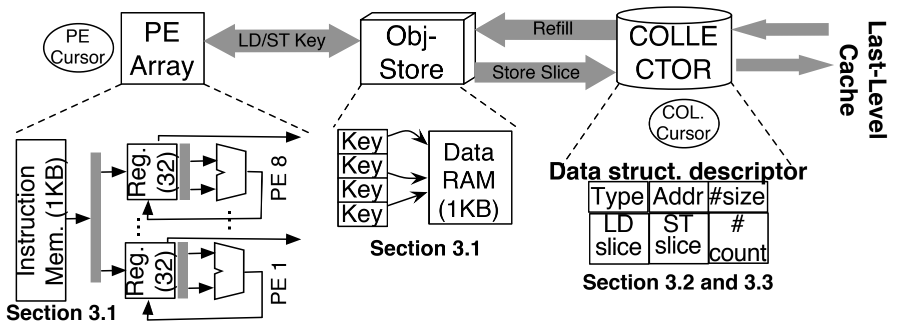


Figure 3: DASX Architecture Overview

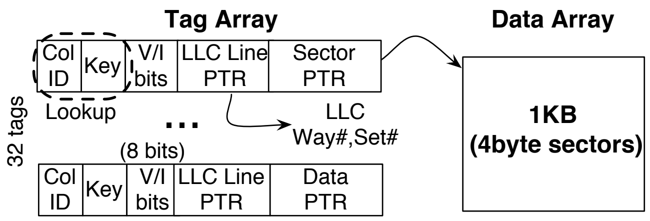


Figure 4: Obj-Store organized as a decoupled sector cache.

3.2 Collector

The Collector is a data-structure specific refill engine. A Collector has three tasks, i) it decodes the load and store slices , ii) translates the object keys to addresses and fetches them into the LLC and iii) it explicitly manages the Obj-Store. The Collector writes back data from the Obj-Store and refills it, setting up the appropriate tags and Obj-Store lines at the beginning of each set of iterations. To supply data objects to the PEs the Collector runs ahead and fetches the cache lines corresponding to the objects and locks the LLC lines until the compute kernel has processed the objects.[2] The accelerators do not need to perform address-generation or miss-handling since the Collector ensures the data is present in the Obj-Store for the duration of the computation. Before we discuss the details of individual Collectors, we illustrate the overall working of DASX with *Blackscholes*.

Illustration: Blackscholes Execution on DASX

In *Blackscholes*, the load slice consists of elements from the vectors spot[], strike[], rate[], volatility[], time[] and type[] while the store slice consists of elements from price[] (see Figure 2); all vectors employ a unit stride. Figure 5 (Left) illustrates the programmer's C++ API for DASX. `new coll(..)` initializes a new Collector and specifies the data structure parameters; specific details of each Collector are discussed in the next section. `group.add(..)` binds Collectors together to run in lock-step fetch elements from different data structures needed by a specific iteration. The need to run Collectors in a synchronized fashion is described in § 3.3. In this example, a Collector is initialized for each vector accessed in the compute kernel.

For this example, we assume 8 PEs and a 1KB Obj-Store. Each iteration requires space for 4×5 (floats) + 1 (char) + 4 (float) = 25 bytes. The 1 KB Obj-Store can hold data for 40 iterations of the *Blackscholes* kernel. The loop is unrolled with each PE executing a single iteration of the compute kernel; PEs sync up after each iteration. A group of iterations processing the data held in the Obj-Store is called a *tile*. The DASX software runtime loads the VEC descriptor for each of the vectors being accessed and invokes DASX by setting the memory mapped control register.

Figure 5 illustrates the steps involved in the execution of *Blackscholes* on DASX. The Collectors issue requests to DRAM (❸) for the first tile while the PEs are stalled. Subsequent DRAM requests overlap with execution of the PEs. The Collectors use some of the ways in each set of the last level cache to prefetch and "lock" (❹) in place, data which will be consumed by the PEs in the future. Locked cache lines may not be selected for replacement. The Collectors load the tile of data into the Obj-Store (❶) and the PEs commence execution of a group of concurrent iterations (❷). For *Blackscholes*, 40 iterations of the original loop consumes all the data in the Obj-Store. The Collectors then write back the dirty data, unlock the cache lines (❺) and initiate a refill (❸) of the Obj-Store. We implement LLC line locking using a logical reference counter (Ref.# in Figure 5). The Ref.# is set to the number of data objects in the LLC line needed by the computation. The Ref.# is zeroed out when the tile of iterations complete execution.

Overall, the key performance determinant is the ability of the Collector to generate addresses and run ahead to fetch the cache lines corresponding to the object keys. The logic required to generate addresses from the object keys is dependent on the type of Collector.

3.3 Types of Collectors

The DASX system evaluated includes three types of Collectors, VEC (vector/array), HASH (hash-table) and BTREE. Each Collector includes a data structure descriptor that contains information to enable the Collector to fetch and stage data independent of the compute kernel.

Vector Collector (VEC)

Figure 6 shows the descriptor that software needs to specify for vectors. The *Keys/Iter.* specifies the per-iteration load slice of the particular vector. For instance, in *Blackscholes* (see Figure 2), each of the vectors are accessed with a unit stride i.e., in the i^{th} iteration accesses the i^{th} element. The *Keys/Iter* is a single entry with offset 0. The descriptor limits the number of vectors accessed per-loop iteration (8 in this paper). The descriptor for spot[] in *Blackscholes* would be: <LD,&spot[0], FLOAT (4 bytes), nOptions (length of vector), [0] (unit stride access) >. The address generation to map data structure *keys* to cache block addresses is simple and shown in Figure 6.

[2]To ensure deadlock-free progress for the CPU cores we ensure that at least one way per set in the LLC remains unlocked.

```
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. Store size.
   start(kernel(),#iterations)
end
```

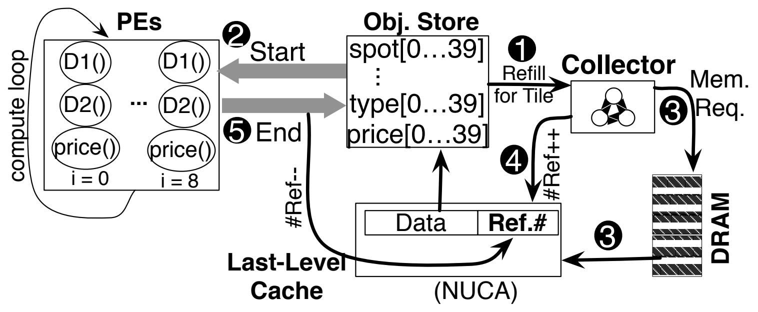
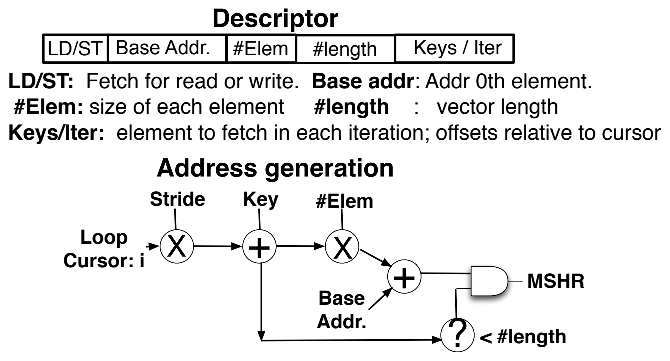


Figure 5: Blackscholes execution on DASX

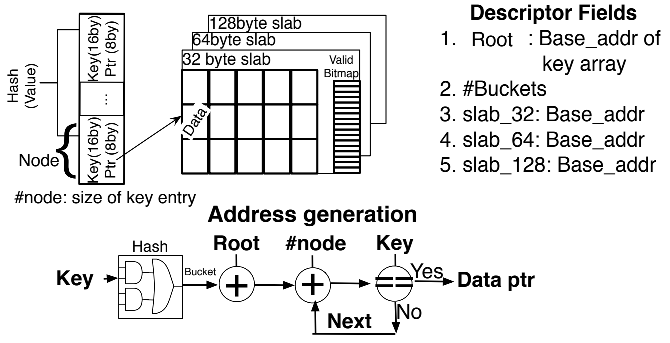


LD/ST: Fetch for read or write. **Base addr**: Addr 0th element.
#Elem: size of each element **#length** : vector length
Keys/Iter: element to fetch in each iteration; offsets relative to cursor

Figure 6: Descriptor for vector Collector

Hash-Table Collector (HASH)

The hash Collector we study in this work supports only lookups similar to [19]; we use the general purpose processor to perform insertions. We only support a fixed organization for the hash table with fixed-size keys (128bits) and three data slabs 32byte, 64byte, and 128byte.



Descriptor Fields
1. Root : Base_addr of key array
2. #Buckets
3. slab_32: Base_addr
4. slab_64: Base_addr
5. slab_128: Base_addr

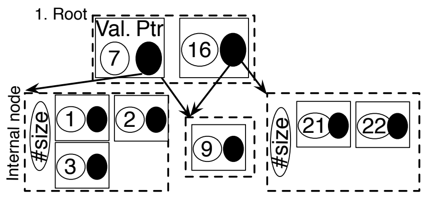
Figure 7: Descriptor for HASH Collector

As shown in Figure 7 the layout consists of a contiguous array for buckets (key-value pairs); the value could be either a 4 byte blob or a pointer to an entry in the data slabs. To search for a single key, the Collector iterates over the bucket array dereferencing each key. To enable better use of the accelerator we typically search for multiple keys simultaneously.

The hash-table also has to perform a comparison between the expected key and the key in the bucket. To amortize the cost of invoking the accelerator we search for multiple keys simultaneously.

BTree Collector (BTREE)

The BTree is a common data structure widely used in databases; it arranges the key entries in a sorted manner to allow fast searches. Similar to the hash table, we do not accelerate insertions.



Descriptor
1. Root pointer
2. Internal_node:
   #size: vector length
   vector <value> ;
3. Value
   char payload[]
   int Key
   Internal_node* subtree

Figure 8: Descriptor for BTree Collector

Our BTree implementation supports keys up to 64 bit length. The key parameters required by the Collector are the root pointer and the description of the internal node. As shown in Figure 8, each sub-tree in the BTree is essentially a vector of value nodes. Each value node contains the key value, the data payload, and a pointer to the sub-tree (if present). When dereferencing each level, the Collector performs a range check and obtains the required sub-tree pointer. There is moderate parallelism in the data structure; the range checks that searches for the key position in the internal node vector can be parallelized.

Collector overhead

To estimate the overhead of the state machine of the Collector we use Aladdin [35], a pre-RTL power-performance simulator. Aladdin uses an ISA independent dynamic trace to estimate the power,area, and energy for custom hardware. Table 3 shows the overhead for the address generation logic of the individual Collectors. We quantify the data transfer latency and account for the Collector energy in § 4 as part of the energy consumed in the cache hierarchy.

Table 3: Design parameters for Collector state machine

	Area	Latency (cycles)	Energy
VEC	$0.23mm^2$	3	138pJ
HASH	$0.23mm^2$	5	266pJ
BTREE (order 5)	$1mm^2$	20	1242pJ

Obtained using Aladdin [35]. OpenPDK Library 45nm

Handling Corner Cases: Collector Groups

Here, we describe the subtle corner case conditions that affect the overall working of DASX. DASX requires that the data needed by one iteration of the loop be LLC resident before the iteration commences. In our benchmarks, we find that *Hash Table* needs 1, *2D-Datacube*, *Text Search*, *BTree* need to have 2, *Recommender* needs 3 and *Blackscholes* needs 7 lines locked in the LLC for DASX to make progress.

In DASX, a single loop iteration's working set cannot exceed the LLC's set capacity (i.e., bytes/set) lest all the data needed by an iteration map to the same set. Using a victim cache can relax this upper bound. Compiler techniques such as loop fission can also reduce the data footprint of the loop by splitting it into multiple loops. This guarantees that PEs can access data from the Obj-Store using *keys* and eliminates the non-determinism due to memory instructions missing in the cache. However, Figure 9 illustrates the possible deadlock that can occur when an iteration traverses multiple data structures. In *Blackscholes*, each iteration needs to access the element from seven different vectors. When a separate Collector is set up for each of the vectors, they run independent of each other and can issue accesses to the LLC in any order. The example illustrates one such case where the LLC is filled by locked lines for spot[0]..spot[255] and does not have any remaining space for the other vector's elements (e.g., type[0]). However, to even commence iteration 0, DASX requires all the data needed by that iteration i.e., the load slice including spot[0],rate[0],volatility[0],type[0], and time[0] and the store slice including price[0] has to be LLC resident. The deadlock arises because we have locked space in the LLC for data elements needed in a future iteration without ensuring there is space available for the data elements in the current iteration.

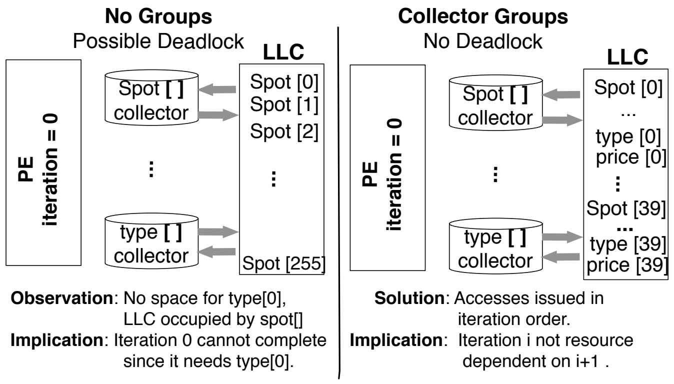


Figure 9: Collector Group. Left: Illustration of possible deadlock in Blackscholes without Collector groups. Right: Collector groups avoiding the deadlock issue.

To resolve the deadlock, we introduce the notion of *Collector group*. In *Blackscholes*, the six Collector (spot[]...type[]) form a Collector group (group() in Figure 2). DASX ensures that the individual Collectors in a group run in lock-step and issue accesses in *iteration order* to the LLC i.e., accesses are issued in the order spot[i]...type[i] before spot[i+1]...type[i+1]. We only require that the LLC allocate space for these elements in- order, the refill requests from DRAM itself can be in any order. Collector groups avoid deadlocks since they guarantee that data needed by every iteration i is allocated space in the LLC before iteration i+1.

4. Evaluation

We perform cycle accurate simulation of DASX with the x86-ISA based out-of-order core[3] modeled using Macsim [1], the cache hierarchy using GEMS [25] and main memory using DRAMsim2 [31]. We added support for DASX's cores into Macsim, and model the state machine of DASX's Collectors faithfully. We model energy using McPAT [22] and use the 45nm technology configuration; our caches are modeled using CACTI [26]. The energy consumption of DASX components are modeled with the templates for the register files, buffers, and ALUs from the Niagara 1 pipeline. Table 4 lists the parameters. The applications include multiple iterative regions and we profile the entire run apart from the initialization phase. We compare DASX against both an OOO (Out-of-Order) core and a multicore (with in-order processors). The baseline DASX system studies 8 PEs with 1KB Obj-Store (in Section 4.5 we vary the number of PEs). We compare the baseline DASX system against a 2C-4T (2core, 4threads/core) multicore with the same level of TLP as DASX (in Section 4.5 we compare DASX against 2C-8T and 4C-8T systems).

[3]We assume a 32K I-cache when modeling fetch energy; the performance simulation assumes all accesses hit to ensure fair comparison with DASX's PEs.

Table 4: System parameters

Cores	2 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
LLC	4M shared 16 way, 8 tile NUCA, ring, avg. 20 cycles. Directory MESI coherence
Main Memory	4ch,open-page, DDR2-400, 16GB 32 entry cmd queue,
Energy Params	L1 (100pJ Hit, HP transistors). LLC (230pJ hit, LP transistors) [23] L1-LLC link (6.8pJ/byte [2, 26]), LLC-DRAM (62.5 pJ per byte [15])
Multicore	2C-4T (2 cores, 4 Threads/core), 2C-8T, 4C-8T 2Ghz, In order, FPU, 32K L1 cache per core, 4MB LLC
DASX Components	
	2GHz. 8 PEs, 4 stages in-order at 2 Ghz, 1 FPU and 1 ALU (per PE) INT Reg./ PE : 32 , FP Reg. / PE: 32
	Shared Obj-Store (1KB fully-assoc. sector cache, 32 tags). Shared Instruction buffer (1KB, 256 entries)

4.1 DASX : Performance Evaluation

Result: *DASX improves performance of data-centric applications by 4.4× over 2C-4T. Maximum speedup (Hash Table): 26.8 ×. Minimum speedup (Recommender): 1.3×.*

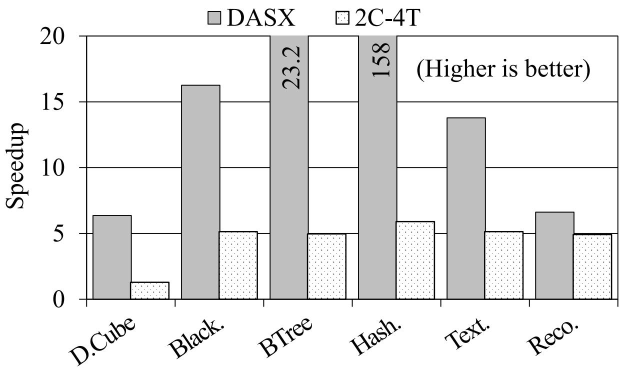


Figure 10: Execution Time (DASX vs 2C-4T). Y-axis: Times reduction in execution cycles over OOO.

We use the 2C-4T multicore to illustrate the benefits that DASX's MLP extraction can provide above the TLP. With *2D-Datacube*, the compute region in the loop body is small (only 21 static instructions); DASX removes the extraneous data structure instructions and mines the MLP implicit in the data structure to achieve a 6.4× performance improvement. 2C-4T suffers from synchronization overheads between the private L1 caches while DASX's simpler cache hierarchy minimizes synchronization overhead. *Blackscholes* includes many (up to 65% of executed instructions) 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. While 2C-4T extracts the available TLP (5× speedup) it exploits minimal MLP since the in-order pipeline resources are occupied by the compute instructions. DASX improves performance further (3.2× speedup over 2C-4T) by extracting the MLP without wasting execution resources. In *Text Search* the compute region requires many instructions to compare strings. DASX improves performance over the 2C-4T primarily by discovering MLP beyond the TLP. For *Blackscholes*, the OOO and the multicore also suffer from L1 thrashing due to repeated iterations over the dataset (3MB); DASX moves the compute closer to the LLC and is less susceptible to cache trashing. Similarly, with the *BTree* the working set fits in the large LLC but as lookups iterate over the tree they thrash the L1 leading to poor performance; DASX moves the compute closer to the LLC.

BTree and *Hash Table* highlight the ability of the Collector hardware to discover the MLP implicit in the data structure. With the *BTree* the traversal between tree-levels is sequential and requires pointer chasing; however DASX does parallelize the range search within a tree. Both the OOO and 2C-4T find it challenging to mine the MLP across pointer dereferences and data dependent branches. DASX improves performance by fetching in parallel the data elements for the range search. With the *Hash Table* the Collector eliminates the data structure instructions to check for the matching key; the removal of the LLC-L1 transfers also helps in improving performance.

In the case of *Recommender* the compute region processes a sparse matrix and the loop body execution is predicated on the value of each element in the matrix. With DASX the hardware barrier that we use to coordinate the PEs at the end of a group (8 in this case corresponding to 8PEs) of iterations introduces an inefficiency. We find that for 8% of the iterations, 1 or more PE executes the loop body while the other PEs are stalled at the barrier. An interesting extension to DASX would be to add support for predication checks into the Collector similar to [27], which advocates "triggered instructions" for spatially programmed architectures.

To summarize the benefits of DASX are due to i) the exploitation of data structure knowledge to run ahead and stage data in the LLC, and ii) the leaner cache hierarchy and small Obj-Store to reduce latency of local accesses within an iteration compared to the conventional cores which has to access the L1, and iii) elimination of the data structure access instructions from the iterative region.

4.2 DASX : Energy Evaluation

Result 1: *DASX is able to reduce energy in the PEs between 3.5—20× relative to 2C-4T.*
Result 2: *Data transfer energy is reduced by 4× relative to 2C-4T by eliminating L1-LLC transfers.*
Result 3: *DASX's shallow memory hierarchy helps reduce on-chip cache access energy by 7×.*

Core vs. PEs

Figure11 shows the reduction in dynamic energy. *BTree* and *Hash Table* are not included in Figure 11 as they do not include any work in the compute region and use only the Collector to traverse the data structure. DASX's PEs also use a simple pipeline which requires minimal energy for instruction fetch (1KB I-buffer vs I-cache on OOO/2C-4T) and data fetches from the Obj-Store

In *2D-Datacube* we see a 11× reduction in energy as the data structure instructions (eliminated in DASX) constitute a major part of all the instructions. With *Blackscholes* and *Text Search* which have compute intensive iteration, the primary energy reduction is due to the simpler PE pipeline and reduction in fetch overhead. The main benefit of DASX is being able to reduce execution time by extracting MLP without requiring the PE to sacrifice its energy efficiency.

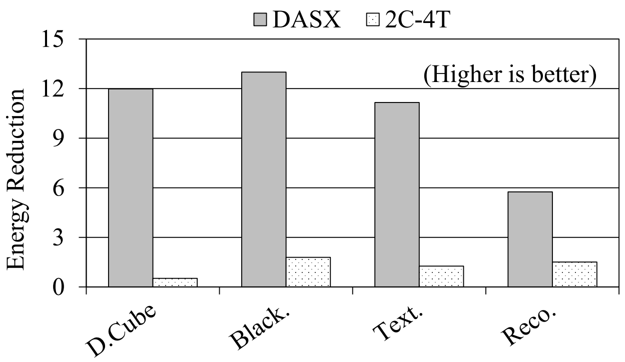


Figure 11: Dynamic energy reduction (Unit: × Times) in PEs vs. 2C-4T (Baseline: OOO core). *BTree* and *Hash Table* do not use the PE.

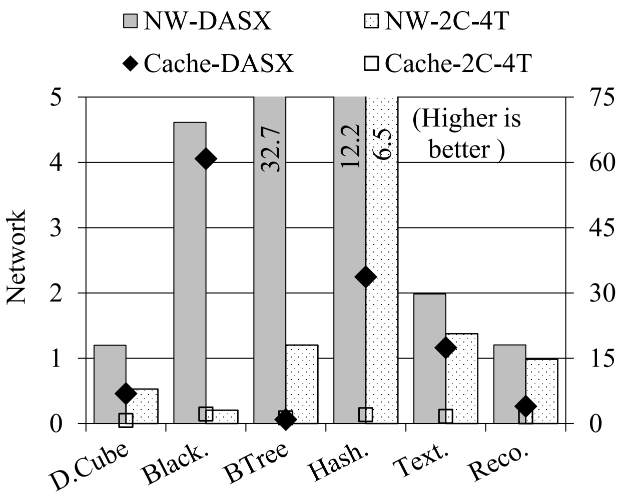


Figure 12: Energy reduction (Unit: ×times). DASX vs 2C-4T normalized to OOO. Left Y-axis: Network energy. Right Y-axis: Cache energy.

Network

Another key benefit of DASX is moving the compute closer to the LLC. Eliminating the L1-LLC transfers result on an average reduction of 3.7× of data transfer energy consumption. For example, *BTree* has low spatial locality which results in compulsory L1 cache misses, but the working set is contained within the LLC. Consequently, DASX achieves 17× reduction in network energy. With *2D-Datacube*, and *Recommender* the data transfer energy is dominated by the LLC-DRAM data transfer energy which is 11× that of LLC-L1 data transfer energy (see Table 4). With applications such as *Hash Table* which exhibit good spatial locality the multicore can exploit TLP to reduce execution time and thereby energy; DASX minimizes LLC-L1 transfers to further reduce energy by 2× compared to the multicore.

Cache

DASX employs a lean memory hierarchy (1KB Obj-Store + 4MB LLC). As the Collector runs ahead of the PEs in DASX and stages the data in the LLC, it requires two accesses to the LLC for each cache line : (1) when the data is fetched from DRAM into the LLC, and (2) when the Obj-Store is refilled. Comparatively, both the OOO and multicore access the LLC only on L1 misses.

Overall, DASX still reduces cache access energy, on average, by 13.6× (Figure12). This reduction comes primarily from two factors, namely, spending less energy in loading data from the Obj-Store relative to the L1 cache and remove stack references by the elimination of register spills due to a larger physical register file shared by the PEs.

We observe that *Blackscholes* and *Hash Table* benefits from the leaner memory hierarchy of DASX and from significant reduction in stack accesses. On the other hand, for *2D-Datacube* and *Recommender* both the OOO and 2C-4T benefit from the spatial locality in the L1 cache, and DASX incurs the additional penalty of accessing the LLC twice for each cache line, thereby limiting the energy savings in the cache. Finally, for *BTree* the higher probability of finding the non-leaf nodes in the L1 cache benefits both the OOO and 2C-4T and consequently limit the cache energy gains observed using DASX.

For all benchmarks apart from *Hash Table* and *BTree*, the energy for the Collector with respect to overall energy consumed was less than 0.25%. For the *Hash Table* and *BTree* benchmark, the Collector accounts for 6% and 4% respectively and when compared with the cache energy consumption for DASX, they represent 16% and 51% respectively. The OOO baseline consumes an order of magnitude higher energy in the cache hierarchy relative to DASX (Figure 12).

4.3 DASX MSHRs

To run ahead of the compute and prefetch far ahead, the Collectors need Miss Status Handling Registers (MSHRs) to keep track of outstanding cache misses. In this section we study the effect of MSHRs on DASX execution time (Figure 13). We found that for *Blackscholes*, which has a compute intensive iteration (many floating point operations), 8 MSHRs (1MSHR/PE) was able to hide the memory latency. With *BTree*, in which each level examines 2 or 3 nodes (we use a binary search within each level to find the next pointer), since we traverse by level and only prefetch within each level, again 8 MSHRs suffice. *Text Search* benchmark builds a heap structure and its size depends on the length of the strings to be compared, these strings can have locations which do not have spatial locality and hence, needed 32 MSHRs. *2d-Datacube* has a small compute region and iterates over a large data set (20Mb) and so requires a lot of MLP, thus 32 MSHRs. Both *Hash Table* and *Recommender* demonstrate high spatial locality and required at most 16 MSHRs.

What about a bigger OOO?

Overall, we find that more ILP $\neq$ more MLP for datacentric applications. We explored a more aggressive OOO to understand the limiting factor to finding more MLP. The instruction window was resized to 256 instructions with 2× the register file and load/store queue entries. Table 5 shows the maximum number of concurrent cache misses sustained and the challenges introduced by a particular data structure. Multicores with TLP help improve MLP but the gains are limited by the ILP of the individual cores.

Collector as Prefetcher

To evaluate the upper bounds of the Collector as a prefetcher, we modelled a perfect LLC for *Blackscholes*, *Recommender* and *2D-Datacube* and found that maximum performance improvement was 22% for OOO core due to the overheads of the cache hierarchy. Note that any prefetcher is best effort and does not guarantee a timely LLC hit; additionally the core will issue the load instruction anyway (energy overhead). The Collector eliminates these instructions and minimizes energy by using special purpose hardware.

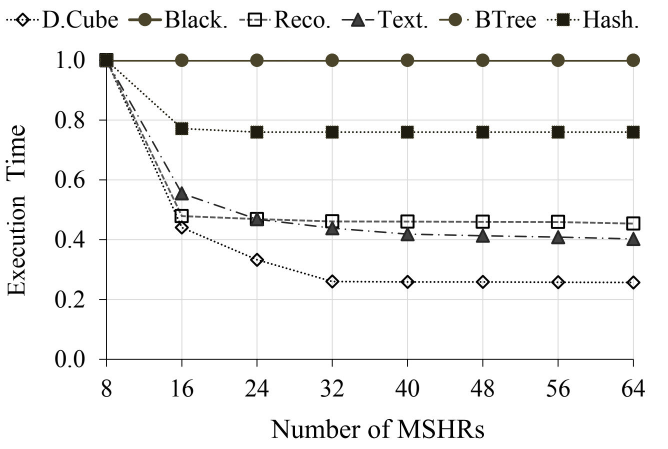


Figure 13: DASX Execution time for 8–64 MSHRs (normalized to 8 MSHRs).

Table 5: Max # of MSHR entries used for a 256 ROB core

Max. Misses	Application	Possible primary cause for limited misses
3	Text Search	Data-dependent branches
4	Hash Table	Data-dependent branches
5	BTree	Pointer chasing. MLP only within tree-level
5	Recommender	ROB filled by compute, waiting on data
6	2D-Datacube	Stalled accesses fill load/store queue
7	Blackscholes	ROB filled by compute, waiting on data

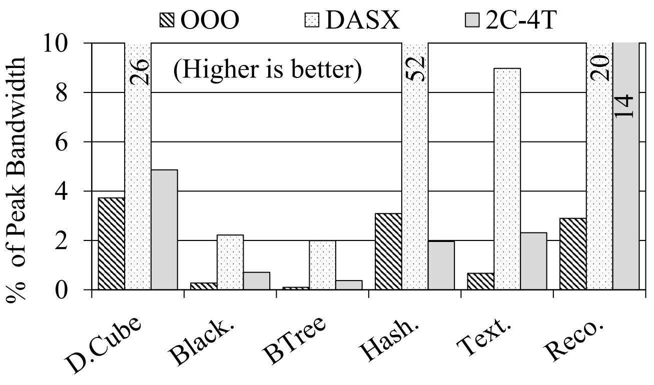


Figure 14: DRAM bandwidth utilization (OOO, 2C-4T and DASX) normalized to peak DRAM bandwidth (6.4GB/s)

4.4 Memory Bandwidth Utilization

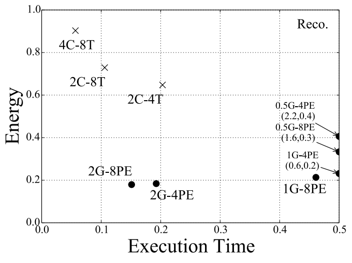
Result: *DASX attains 13.9× increase in off-chip DRAM bandwidth compared to the OOO. Max: 82.4×(*BTree*).*

With data structure specific information DASX is able extract the MLP and issue timely requests to stage the data in the LLC before the PEs consume it. As shown in Figure 14, large increases in DRAM bandwidth utilization are observed for *2D-Datacube*, *BTree* and *Hash Table*. For *Hash Table* DASX is able to approach within 52% of the theoretical peak bandwidth of 6.4GB/s. *Text Search* and *Recom-*

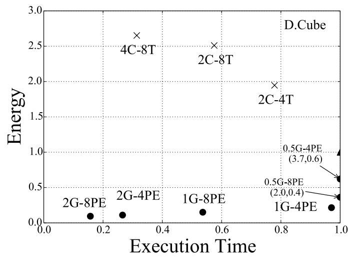
mender both include computationally intensive kernels which lag far behind the Collector. This causes the Collector to fill up the LLC and then wait for the PEs to catch up and free space. The memory fetch rates are essentially limited by the latency of the compute kernels. DRAM bandwidth utilization improves by 3.9× and 1.4× for *Text Search* and *Recommender* respectively, relative to 2C-4T. The increase in bandwidth is observed due to the reduced run time of the kernels. All the DASX accelerated benchmarks access the same amount of data as their baseline counterparts apart from *Text Search* which overflows the LLC and leads to a large number of replacements as it iterates over the dataset repeatedly.

4.5 TLP vs MLP

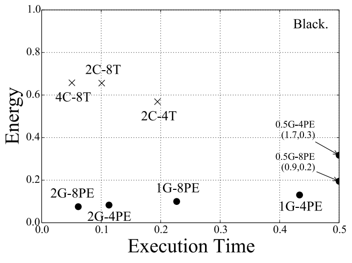
Result: *Even with 50% lower TLP, (DASX: 4PEs vs 2C-4T) DASX exploits MLP to improve performance by 2.3× and reduce energy by 12.3× (Figure 15).*



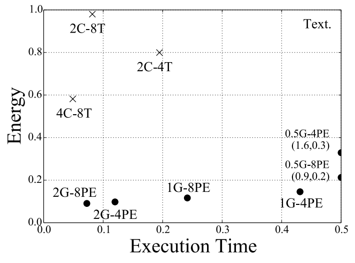
(a) Recommender



(b) 2D-Datacube



(c) Blackscholes



(d) Text Processing

Figure 15: Energy vs Execution Time plot. Y-Axis: Energy. Lower is better. X-axis: Execution time (scale different for each benchmark). towards origin is better. Baseline : OOO (1,1). × : Multicore systems 2C-4T,2C-8T,4C-8T 2Ghz frequency. • DASX systems: 0.5G,1G,2G - Clock frequency in Ghz.

We compare the energy and performance of a variety of multicore and DASX configurations to understand the benefits of extracting MLP versus TLP. We use parallel versions of each of the benchmarks where the work is partitioned across multiple threads. We varied the PE configuration of DASX between 2—10 and the frequency of the PE (0.5Ghz,1Ghz,2Ghz) to understand the dependence of our workloads on the execution pipeline performance, if the Collector can extract the MLP independent of the compute units. We study three different multicore configurations 2C-4T (2core, 4threads) which has the same level of parallelism as the 8-PE system, 2C-8T (16 threads) and 4C-8T (32 threads); all multicores run at 2Ghz. Figure 15 plots the dynamic energy vs execution time (we only show the 4 PE and 8 PE DASX for brevity).

Overall, we find that DASX extracts MLP in an energy efficient manner and provides both energy and performance benefits over multicores which only exploit TLP. However, even with DASX the performance of compute units is important to ensure that the compute kernel does not limit the performance gains of DASX. When the PE clock is reduced to 0.5Ghz or 1Ghz resulting in increase in execution time of the compute region, the performance loss varies 2–5× compared to the 2C-4T multicore; *2D-Datacube* is an exception since the compute kernel has less work. The multicore performance benefit and energy reduction are limited due to synchronization overheads (*2D-Datacube* performs a reduction). For embarrassingly parallel workloads (*Blackscholes*, *Recommender*, *Text Search* and *Hash Table*), multicores improve performance proportionate to the TLP but have minimal reduction in energy as execution resources are required to maintain the active threads. DASX is able to provide significant benefits with half the compute resources since the Collectors offload the MLP extraction from the compute cores. A 2G-4PE (2Ghz,4PE) DASX improves performance by 1.1–4.6× and reduce energy by 3.6–38× compared to a 2C-4T (8 threads).

Multicore with OOO cores

We used parallel (8 thread) versions of *Blackscholes* and *Recommender* to illustrate the pitfalls of using an OOO multicore to extract MLP. With 8 cores (single thread per OOO core), *Blackscholes* shows a performance improvement of 7.44× with a net increase in energy of 42% vs a single OOO core. With *Recommender* the parallel version demonstrates a 8× speedup with a 24% increase in energy consumption. While the OOO multicore has performance better than DASX (16% and 21% respectively), the energy consumption is far greater than DASX (17× and 7× reduction in energy consumption by DASX).

4.6 Breakdown of performance gains

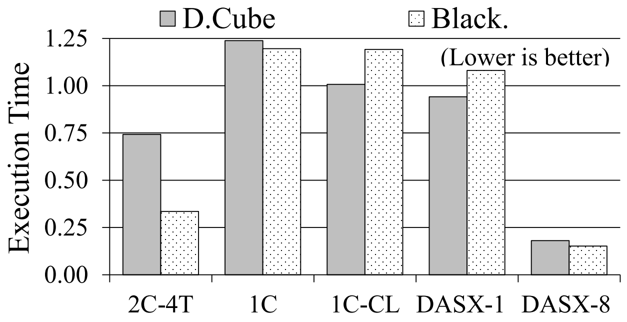


Figure 16: Motivation for Collector and Obj-Store to interface accelerators (PEs) to the memory system. (Lower is better)

To further illustrate the impact of each design decision of the DASX architecture, we conducted the following experiment. Using in-order cores as compute units, we measured the performance while a) varying the number of cores b) placing them near the LLC with only a 1KB L1 c) using a Collector to stage data in to the LLC d) adding an Obj-Store and removing address generation from the cores to form a complete DASX system and e) increasing the number of PEs to exploit loop level parallelism. Figure 16 shows the five configurations in the order they have been described herein. The performance of each configuration, for each benchmark, was normalized to the execution of the benchmark on an OOO core. We illustrate our findings using 2 benchmarks based on their behaviour on an OOO core, i.e memory bound (*2D-Datacube*) and compute bound (*Blackscholes*).

We find that the 2C-4T configuration only improves performance by 25% for the memory bound *2D-Datacube* benchmark. Placing a single in-order core near the LLC (1C configuration) is able to buy back some of the performance drop and is within 24% of the OOO baseline. This illustrates the benefit of computing near memory for data centric iterative workloads. Adding a Collector to ensure all accesses to the LLC are hits (1C-CL configuration) works well for the memory bound workload and a single in-order core is able to match the performance of the OOO baseline. Adding an Obj-Store and removing data structure access instructions from the compute kernel allows DASX-1 to further improve performance by 6% for *2D-Datacube* and 11% for *Blackscholes*. Finally, using 8 PEs in DASX-8 improves performance by 5× for *2D-Datacube* and 7× for *Blackscholes*.

4.7 Area Evaluation

We use McPAT to model the area overhead of the DASX's PEs. To model the DTLB, Obj-Store, and the reference count (Ref.# for locking cache lines), we use CACTI 6. The dominant area overhead in DASX is the FPU macro which is adopted from the Penryn processor in McPAT. Out of the six data-centric applications we investigated, five of them require minimal FPU support; however, we require an FPU per PE for Blackscholes (up to 65% of the compute kernel instructions are floating-point). It may be feasible to share the FPUs between the PEs like Sun-Niagara multicores to reduce the overhead. In comparison, the OOO core occupies 25.701 mm^2.

Table 6: Area Analysis (45nm ITRS HP)

Component	Area (mm^2)
ALUs (1 each)	INT (0.078), FP (2.329), MUL/DIV (0.235)
Reg File (32+32)	0.346
Total (PE)	2.988
Ins. Buffer (256 entries)	0.052
DTLB (64 entries)	0.143
Ref.#	6b/line. 47KB/4MB LLC
DASX w/ FPU (8*PEs)	24.099
w/o FPU	5.46

5. Related Work

Table 7 qualitatively compares DASX's design against existing architectures. We discuss individual designs below.

Specialized Core Extensions

Recently, researchers have proposed specialized compute engines that are closely coupled with the processor [3, 16, 29, 38]. In kernels which operate on large data structures [11] the load/store interface imposes high energy overhead for address generation [33]. The collectors in DASX exploit data structure knowledge to minimize the complexity of the address generation and improve MLP to supply data to compute accelerators. Loop accelerators [14] exploit repeated instruction streams to save fetch energy; DASX focuses on data structures. SIMD extensions reduce per instruction overheads, but are limited by the register width and difficult to apply to value dependent traversals. However, the PEs themselves may be enhanced with SIMD support similar to the GPGPU execution model.

Data structure Appliances

Recent research has focused exclusively on key-value stores [9, 17]. FPGA-Memcached [9] developed an FPGA-based hash-table and TCAM exploited power efficient content-addressable memory [17]. Unfortunately, these systems are challenging to use in general-purpose programs. Key-value stores do not efficiently represent associative containers; an integer array represented in TCAM [17] would require 2× the storage as a software data structure. More recently, HARP [39] developed support for database partitioning. Their memory system includes an application-specific on-chip streaming framework to interface with a custom on-chip computational logic. DASX seeks to accelerate iterative computation that access data structures and exhibit temporal and spatial locality. DASX also develops techniques for tiling the dataset at runtime to fit in the LLC and build a general-purpose memory interfaces to support software compute kernels.

Prior work have proposed using limited capacity hardware data structures [10, 13, 20, 40]. While this enables high performance, the hardware exports a limited interface to software which also needs to provide virtualization support [6]. DASX accelerates software data structures and does not require any specialized storage for the data structures. Since software manages the allocation DASX does not require complex virtualization support. Past work have also focused on CPU instructions for assisting software queues [21, 28].

Hardware walkers, both TLB [4] and more recently hash-table [19] focus on supporting efficient lookups in indexed data structures. While both DASX and hardware walkers exploit semantic information about data structures, DASX targets iterative compute kernels that stream over the entire data structure. DASX permits the compute kernels to be written in software. DASX adaptively tiles the dataset to handle cache contention between multiple data structures in an iteration and adapts to the compute kernel's temporal and spatial locality. While hardware walkers use a narrow queue-like interface to supply data, DASX supports a more flexible general purpose interface to the compute kernel. Programmable memory controllers [7, 8] provide an effective site to embed DASX's data-structure Collectors.

6. Summary

We have focused on exploiting data structure information to provide compute accelerators with energy-efficient interfaces to the memory hierarchy. We developed DASX, a hardware accelerator that integrates with the LLC and supports energy efficient iterative computation on software data structures. DASX exploits information about the computation kernel and actively manages the space in the LLC to ensure that no data fetch is wasted. DASX deploys cache refill engines that are customized for software data structures which increases the MLP, eliminates the penalty of transfers through the cache hierarchy, and removes data structure instructions.

7. 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. *IEEE 18th International Symposium on High Performance Computer Architecture (HPCA), 2012*, 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] G. Bloom. Operating System Support for Shared Hardware Data Structures. *GWU Thesis*, pages 1–136, Apr. 2013.

[7] M. N. Bojnordi and E. Ipek. PARDIS: a programmable memory controller for the DDRx interfacing standards. In *ISCA '12: Proceedings of the 39th Annual International Symposium on Computer*

Table 7: DASX vs Existing Approaches

	OOO+SIMD	Decoupled Architecture [36]	HW Prefetching	**DASX (this paper)**
Memory-Level Parallelism	**More MLP (limiting factor listed below)** $\longrightarrow$.			
	Register width	Strands and inter-pipeline queues	Predictor table	MSHRs
Data-Struct. Instructions	Core executes data structure instructions (increases dynamic energy).			No (saves dynamic energy)
Big data-structure support	Limited by L1 cache size and load/store queue interface.		Limited by predictor table	Limited by LLC size
Fixed-latency Loads	No; cache misses	Yes (hardware queue)	No; cache misses	Yes (LLC line locking)
Value-Dependent Traversal	Challenging	Limited by branch prediction.	X (No support)	Efficient. Data structure-specific.

Architecture. ACM, June 2012.

[8] J. Carter, W. Hsieh, L. Stoller, M. Swanson, L. Zhang, E. Brunvand, A. Davis, C.-C. Kuo, R. Kuramkote, M. Parker, L. Schaelicke, and T. Tateyama. Impulse: building a smarter memory controller. In *PROC of the 5th HPCA*, 1999.

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

[10] R. Chandra and O. Sinnen. Improving application performance with hardware data structures. *Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010*, pages 1–4, 2010.

[11] J. Chang, P. Ranganathan, T. Mudge, D. Roberts, M. A. Shah, and K. T. Lim. A limits study of benefits from nanostore-based future data-centric system architectures, 2012.

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

[13] E. S. Chung, J. C. Hoe, and K. Mai. CoRAM: an in-fabric memory architecture for FPGA-based computing. In *PROC of the 19th FPGA*. ACM Request Permissions, Feb. 2011.

[14] N. Clark, A. Hormati, and S. Mahlke. VEAL: Virtualized Execution Accelerator for Loops. In *PROC of the 35th ISCA*, 2008.

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

[16] H. Esmaeilzadeh, A. Sampson, and D. Burger. Neural Acceleration for General-Purpose Approximate Programs . *PROC of the 44th MICRO*, 2012.

[17] Q. Guo, X. Guo, Y. Bai, and E. Ipek. A resistive TCAM accelerator for data-intensive computing. In *MICRO-44 '11: Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM Request Permissions, Dec. 2011.

[18] S. Gupta, S. Feng, A. Ansari, S. Mahlke, and D. August. Bundled execution of recurring traces for energy-efficient general purpose processing. In *PROC of the 44th MICRO*, Dec. 2011.

[19] O. Kocberber, B. Grot, J. Picore, B. Falsafi, K. Lim, and P. Ranganathan. Meet the Walkers. *PROC of the 46th MICRO*, pages 1–12, Oct. 2013.

[20] S. Kumar, C. J. Hughes, and A. Nguyen. Carbon: architectural support for fine-grained parallelism on chip multiprocessors. In *PROC of the 34th ISCA*, 2007.

[21] S. Lee, D. Tiwari, Y. Solihin, and J. Tuck. HAQu: Hardware-accelerated queueing for fine-grained threading on a chip multiprocessor. In *HPCA '11: Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture*. IEEE Computer Society, Feb. 2011.

[22] 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.

[23] 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.

[24] K. T. Malladi, I. Shaeffer, L. Gopalakrishnan, D. Lo, B. C. Lee, and M. Horowitz. Rethinking dram power modes for energy proportionality. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 131–142. IEEE Computer Society, 2012.

[25] 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.

[26] 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.

[27] 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.

[28] Y. M. Patt. Microarchitecture choices (implementation of the VAX). In *PROC of the 22nd annual workshop on microprogramming and microarchitecture*, 1989.

[29] W. Qadeer, R. Hameed, O. Shacham, P. Venkatesan, C. Kozyrakis, and M. Horowitz. Convolution Engine: Balancing Efficiency & Flexibility in Specialized Computing. *PROC of the 40th ISCA*, pages 1–12, Apr. 2013.

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

[31] 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.

[32] 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.

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

[34] A. Seznec. Decoupled sectored caches: conciliating low tag implementation cost. In *PROC of the 21st ISCA*, 1994.

[35] Y. S. Shao, B. Reagen, G.-Y. Wei, and D. Brooks. Aladdin: A pre-rtl, power-performance accelerator simulator enabling large design space exploration of customized architectures. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, 2014.

[36] J. E. Smith. Decoupled access/execute computer architectures. In *25 years of International Symposia on Computer Architecture*, 1998.

[37] S. Srinath, B. Ilbeyi, M. Tan, G. Liu, Z. Zhang, and C. Batten. Architectural specialization for inter-iteration loop dependence patterns. In *Intl Symp. on Microarchitecture (MICRO)*, 2014.

[38] 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.

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

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