**X-Cache**: A Modular Architecture for Domain-Specific Caches

Ali Sedaghati, Milad Hakimi, Reza Hojabr, Arrvindh Shriraman
School of Computing Sciences, Simon Fraser University
Canada
{asedagha, milad_hakimi, shojabro, ashriram}@cs.sfu.ca

**Abstract**

With Dennard scaling ending, architects are turning to domain-specific accelerators (DSAs). State-of-the-art DSAs work with sparse data [37] and indirectly-indexed data structures [18, 30]. They introduce non-affine and dynamic memory accesses [7, 35], and require domain-specific caches. Unfortunately, cache controllers are notorious for being difficult to architect; domain-specialization compounds the problem. DSA caches need to support custom tags, data-structure walks, multiple refills, and preloading. Prior DSAs include ad-hoc cache structures, and do not implement the cache controller. We propose X-Cache, a reusable caching idiom for DSAs. We will be open-sourcing a toolchain for both generating the RTL and programming X-Cache. There are three key ideas: i) **DSA-specific Tags** (Meta-tag): The designer can use any combination of fields from the DSA-metadata as the tag. Meta-tags eliminate the overhead of walking and translating metadata to global addresses. This saves energy, and improves load-to-use latency. ii) **DSA-programmable walkers** (X-Actions): We find that a common set of microcode actions can be used to implement the DSA-specific walking, data block, and tag management. We develop a programmable microcode engine that can efficiently realize the data orchestration. iii) **DSA-portable controller** (X-Routines): We use a portable abstraction, coroutines, to let the designer express walking and orchestration. Coroutines capture the block-level parallelism, remain lightweight, and minimize controller occupancy. We create caches for four different DSA families: Sparse GEMM [35, 37], GraphPulse [30], DAX [22], and Widx [18]. X-Cache outperforms address-based caches by 1.7× and remains competitive with hardwired DSAs (even 50% improvement in one case). We demonstrate that meta-tags save 26–79% energy compared to address-tags. In X-Cache, meta-tags consume 1.5–6.5% of data RAM energy and the programmable microcode adds a further 7%.

**CCS Concepts**

• Computer systems organization → Architectures; • Hardware → Hardware-software codesign.

**Keywords**

Domain-specific Architectures, Caches, Coroutines, Dataflow architectures

---

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ISCA ’22, June 18–22, 2022, New York, NY, USA © 2022 Association for Computing Machinery. ACM ISBN 978-1-4503-8610-4/22/06 . $15.00 https://doi.org/10.1145/3470496.3527380

---

**1 Introduction**

Dally et al. [8] and Hennessy-Patterson [14] note that while parallelism and arithmetic intensity are essential, the key to energy efficiency is exploiting extreme locality, making fewer DRAM accesses, and maximizing bandwidth utilization. Thus, most of the resources in current DSAs are dedicated to walking data structures, packing them in SRAMs, and orchestrating data movement and computation. Existing DSAs [6, 20] predominantly work with dense data and organize them into scratchpads with explicit DMA.

Emerging DSAs target a broad set of applications characterized by: i) **Non-affine data structures**: DSAs typically organize data in DRAM with a minimal footprint. Hence, they employ sparse data structures [37], indirect-indexes [30], and hash tables [18]. Getting an element’s global address requires data structure traversal. ii) **Dynamic accesses**: Both the data structure and loop pattern cause emerging DSAs to have dynamic (i.e., indirectly addressed) and irregular non-linear accesses. A cache is necessary to capture the reuse. [35] iii) **Walkers**: Since data is stored in non-linear data structures, a walker is required to traverse [5] and preload the cache. A single miss could trigger multiple nested preloads [37]. iv) **Explicit orchestration**: Finally, DSAs need to explicitly orchestrate cache replacement and refill with the computational datapaths. In this paper, we introduce an architectural template for domain-specific caches and provide a toolflow to achieve DSA-specialization through

---

**Figure 1: Address-based Cache vs X-Cache**

---

**ACM Reference Format:**

the cache controllers. Some prior DSAs include cache-like structures [7, 35, 37] that are inextricably tied to the underlying DSA. They do not implement the cache controller.

Figure 1 illustrates the challenges of incorporating an address-based cache in a sparse matrix DSA: i) Metadata to address translation (Wasted energy): The sparse GEMM DSA works with row/col indices, while the cache is based on the addresses. Logic would be required to convert row/col indices; in sparse matrices, this would require processing CSR/CSC metadata. ii) Address tags (High load-to-use latency): Conventional caches are tagged by blocks’ address, i.e. even if the required row/col is cached, we cannot determine a hit until we find out the address corresponding to the row/col. This would lead to some extra access to the cache or even the DRAM. iii) Walker logic (High NRE): Finally, significant design effort is required for creating custom walkers for each DSA. Walkers would need to handle parallel refills, nested walks, and pipelining.

X-Cache removes the memory layout and address-generation concerns from the DSA. It dispenses with the flat address space, and DSAs are not required to manipulate raw addresses. X-Cache’s architecture includes multiple novel ideas: DSA-specific tags (Meta-tags): X-Cache permits any combination of metadata fields to serve as cache tags. For example, in sparse GEMM, the Row:Col in CSR/CSC representations will serve as the meta-tag. The computational datapath uses meta loads/stores, and we implicitly locate the data on-chip; X-Cache uses global addresses only on misses. While prior DSAs [30, 35, 37] include fields that mirror the functionality of meta-tags, we are the first to define and generalize the concept.

**DSA-programmable walking (X-Routine):** On a miss, X-Cache implicitly walks and finds the relevant data. In X-Cache, each DSA can specify a custom walker. Our insight is that the walker can be implemented using a common set of microcode actions for data and tag management.

**DSA-agnostic controller architecture:** X-Cache’s controller implements the walkers and data orchestration as coroutines. Coroutines cooperatively yield on long-latency events, in contrast to prior work that used blocking threads [5, 6]. Coroutines conveniently capture the underlying parallelism across meta-tags, minimize occupancy, and multiplex walkers for memory parallelism.

We evaluate X-Cache by creating domain-specific caches for five different DSAs, SpArch [37], GraphPulse [30], Widx [18], DASX [22], and Gamma [35]. Both SpArch and Gamma can use the same X-Cache microarchitecture i.e., we only had to reprogram the controller. We demonstrate that we can auto-generate the meta-tags across different DSA families, vertex ids (GraphPulse), database keys (Widx), data structure indices (DASX), Compressed-matrices (SpArch and Gamma). Overall, our performance was comparable to the hard-coded caches in existing DSAs. In the case of Widx, we even improved performance by 50% by reducing the load-to-use latency. Compared to the best-performing address-based cache for each DSA, we improved performance by 1.7×. Between 66—89% of X-Cache’s energy was spent on data; only 1.5—6.6% required for tags. The energy penalty of a programmable controller is less than 7%, i.e., a minimal penalty for being reusable compared to a hardwired design. Our contributions:

- We propose a reusable caching idiom that supports dynamic access patterns and irregular data structures in DSAs.
- We generalize the concept of domain-specific tags (meta-tags) and create a high-performance programmable cache controller.

**We demonstrate that walkers can be expressed as a common set of microcode actions.**

- We create domain-specific caches targeting five different DSAs from three domains sparse-matrix computation, databases, and graph processing. We will be open-sourcing a Chisel generator and microcode table compiler.
- Domain-specific caches improve performance 1.7× by short-circuiting data structure walks and reducing memory accesses by 2—8×. The controller requires 7% of energy, and tags only require 1.5—6.5% of data RAM energy.

### 2 Motivation and Scope

#### 2.1 Why not scratchpads

Figure 2 illustrates the walker of an inner-product sparse GEMM DSA. Matrix A is stored in CSR format and matrix B in CSC. As the DSA streams in matrix A, it fetches the non-zeros from the corresponding column in matrix B. There are multiple metadata (META) accesses, e.g., for $A[0,0]$ we consider non-zeros from B’s column 0, but for $A[0,1]$ we refill B’s column 1. Since B is sparse, DSA traverses the indices (IDX) to retrieve the coordinates of the non-zeros. We then check for matches (MATCH) and find non-zero elements for which A is also non-zero. The accesses to matrix B are: i) dynamic, i.e., the tile of matrix B refilled from DRAM depends on the coordinates of matrix A’s non-zero element. ii) conditional, i.e., elements in a column may be skipped over depending on A’s non-zero pattern, and iii) the reuse depends on the sparsity pattern.

Figure 3 illustrates the state-of-the-art scratchpads. We first consider Scratchpad+DMA. There has been a long history of work on decoupled engines for shifting dense tensors from DRAM onto on-chip SRAMs. This includes scatter/gather engines [10, 13, 17, 31], memory controllers [4], and tiled DMAs [1, 6, 28]. All designs introduce additional address spaces, either local [17, 28], carved from the physical [31] or virtual [4] addresses. Thus, DSAs that include unordered and dynamic accesses would need to implement address-translation from meta-tags to these local addresses [35, 37]. Further, they can only support access patterns driven by affine loops and
strides. We focus on algorithms with indirect indexes, sparse tensors, and conditional accesses. Scratchpad-AE (programmable access engine) There have also been works on augmenting scratchpads in GPUs [21] and FPGAs [5, 6]. Scratchpad-AE targets DSAs, which can split into coarse-grain access-execute regions. The access patterns have to be regular, i.e., the order of accesses has to be known statically upfront and cannot be conditional. They are also heavy-weight since the access engine is mapped to an FPGA kernel or GPU threads. They target tile-granularity reuse (e.g., dense GEMM) and cannot support fine-grain misses and refills. Since the access engine iterates over elements in a specific order, they cannot accommodate dynamic input-dependent reuse behaviour (e.g., indirect indexes).

2.2 Summary: X-Cache vs Other Idioms
Figure 3 and Table 1 qualitatively compare X-Cache against the other popular storage idioms, scratchpads with decoupled DMAs, scratchpads with user-managed access engines, and FIFOs. We use the following taxonomy: i) Implicit vs. explicit The walking and orchestration tasks are broken down into sub-tasks (e.g., miss refill, eviction). Each of these sub-tasks can be implicitly triggered by memory accesses or may be explicitly invoked by the datapath. ii) Coupled vs. decoupled: Coupled lacks domain-awareness and refills only a single data element. Decoupled models have domain-awareness and preload multiple data elements. Prior works support decoupled fetches for static access patterns. However, emerging DSAs require support for dynamic-decoupled accesses. iii) Dynamic vs. Static: The data access order is fixed in DSAs with static access patterns, and the addresses can be calculated using affine functions. Emerging DSAs exhibit dynamic data-dependent access pattern; the access order of global addresses are determined by the input pattern.

3 Deconstructing DSA Caches
Our thesis is that we should facilitate the adoption of domain-specific caches by deconstructing them into modular components.

3.1 Choice 1: Address Tags vs. Meta Tags
We propose that DSAs should issue meta loads/stores that directly reference the elements of the data structure. X-Cache is responsible for implicitly figuring out the addresses and moving the data between DRAM and on-chip. Once the data is brought on-chip, X-Cache tags the data with actual DSA-specific metadata fields so that subsequent accesses can directly reference the on-chip data. Figure 4 plots the load-to-use latency using a domain-specific meta-tag vs. address-based tag. Meta-tags notably improve the load-to-use latency. In some sense, domain-specific tags are similar to TLB tags; they short
circuit the metadata-to-address translation and eliminate the nested data-structure walks. In contrast, the address-tags require many more access for walking, even when the required element is cache resident. Multiple DSAs in the same family will employ the same type of meta-tags, e.g., sparse GEMM accelerators [29, 35, 37].

3.2 Choice 2: Fixed vs. Programmable Walker
A central design consideration is whether to make the walkers fixed-function. To understand this, we consider two DSAs from the same family, sparse GEMM: inner-product and Gustavson product [35] (Figure 5). Both DSAs work with the same data structures (CSR/CSC compressed matrices) and use the same meta-tag, but require different walker logic. In both DSAs, the multiplicand, matrix A, is streamed in. However, matrix B requires a walker, and this varies between the DSAs due to the loop order. In the Gustavson DSA, [35] we perform an outer product between elements of matrix A and the corresponding row of matrix B, e.g., here we multiply A[0,0]:m by B[0,0]:. The walker preloads all the non-zeros in B[0,0]:. There is only local reuse within a tile. Within a tile, accesses could be unordered and dynamic, but tile order is known upfront. In contrast with inner-product the elements in matrix B are dynamically determined, e.g., for element A[0,0] walker refills B[0,0]:a and B[2,0]:c. The reuse pattern is entirely dependant on the pattern on non-zeros in matrix A’s rows, e.g., B[0,0]: a brought in first for A[0,0], reused by A[1,0]. A programmable cache controller will enable design reuse and portability across these DSAs. We can retain the same physical structures while varying the logic based on the GEMM algorithm.

3.3 Choice 3 — Threads vs. Coroutines
Here we consider the question of how to decouple the walkers and run multiple in parallel. Past approaches have explored the use of threads for static access patterns [5, 6, 18]. Each walker is assigned to a hardware pipeline and fetches a tile into an on-chip buffer. The main problems are caused by the data-dependent branches (MATCH, IDX), indirect memory operations (META), and DRAM accesses (FILL). (see Figure 5 and Figure 6). These lock up the pipeline and reduce the number of active walkers, and minimize memory parallelism.

This paper expresses the walkers as a coroutine and multiplexes them on a pipeline. (see FSM shown in Figure 5). Coroutine breaks up a sequential program into regions and enables multitasking. Here, we break up the walker logic into stages/states and yield the thread at annotated long-latency operations, e.g., memory accesses (FILL), dependence chains (AGENT), and data-dependent branches (MATCH). When the long latency event completes, the walker is rescheduled from where it has been left off. Multiple benefits with coroutines: i) we can have multiple walkers and consequently refills from DRAM active, and ii) we can minimize pipeline stalls, and lockups require less hardware. Figure 7 compares the occupancy of controller when walker implemented with coroutine vs. threads. Occupancy is measured as \#active-reg \times \text{sizebytes} \times \text{lifetimecycles}. With threads, we experience 1000× more occupancy since resources are allocated/freed at a coarse granularity. As the fraction of data residing off-chip increases, long latency transactions increase, and thread occupancy increases.
4 X-Cache Architecture

4.1 X-Cache’s Microarchitecture Modules

Figure 8 illustrates the modules in X-Cache. 1 Meta-Tag It is created by our Chisel generator based on the parameters set by the DSA architect. The architect defines the metadata fields that the controller should write explicitly to the tag entry when a refill is completed. These fields will be used for tag matches and determining hits. We also maintain a bitmap to track the active meta-tags, i.e., metadata for which a walker is already active. This helps to continue the execution of a transaction from the stalled point (e.g., on receipt of the DRAM response).

2 Meta-tag State and Event Triggers The fetch or starting point of the pipeline is the message queues at the controller. The trigger table provides a mapping from the incoming set of messages to events in the protocol. Also, in a conventional cache controller, states are for maintaining coherence. However, in X-Cache the states represent the status of blocks in the walker. And, pairing it with the incoming event leads to sequencing through the walking logic. So, the current state of an entry is maintained along with the meta-tags. The default is the starting state for misses, i.e., no entry in the meta-tag array. The combination of the input event and the current state triggers a routine, determined by the Routine Table.

3 Routine Table Similar to cache coherence protocols [32], we have developed a table-based specification. The table encodes the walker logic’s states, events, and transitions. Each cell is a pointer to a routine in the microcode RAM.

There are multiple benefits: i) it supports highly parallel pipelines since X-Cache can process the routines of different meta-tags in parallel. ii) it naturally eliminates structural hazards since routines are not triggered until all the hazard conditions are eliminated. So, it simplifies the pipeline implementation.

4 Routines and μ-coded RAM X-Cache compiles the actual procedures implementing the walking and orchestration down to a microcode binary and stores it in the routine μ-code RAM. The RAM is partitioned into multiple routine handlers. Each routine would be programmed as a sequence of microcode actions finalized with an update to the state. Based on the sequence of the actions, routines can support multiple tasks, i.e., walking the DSA metadata in memory, arbitrating among accesses to the data, calculating addresses and updating the meta-tag arrays.

5 Actions The controller can only invoke a predefined set of memory and communication primitives. Actions are low-level microcode that specifies the control signals of each internal structure. An important consideration, what is the “proper” granularity for the actions. We adopt actions that can be implemented atomically in hardware with fixed latency in 1 cycle. There are five different categories of actions targeting each hardware module: address generation, message queue, Meta-tag, control flow, and data RAMs. The table in Figure 8 summarizes the list of actions. An action’s operands can be explicit (e.g. immediate operand for add or shift), implicit (e.g., queuing request to DRAM), or DSA-specific (e.g., data size).

6 Data RAMs The data RAM is physically banked based on the number of words supplied to the compute datapaths. Logically, the data RAM is organized as fixed-granularity sectors. Each data element can occupy multiple sectors depending on the size (e.g., number of non-zeros in a row). The meta-tag organization is completely decoupled from the data RAM. Meta-tag only serves to determine whether the data is in the cache or not, i.e., a miss map. Each meta-tag entry includes explicit pointers to start and end sectors within the data RAM (like decoupled sector-caches). On a hit, we use the pointers to retrieve the sectors and pipeline them through a crossbar switch to the datapath. Since the miss walkers are programmable, they copy the DRAM response sector-by-sector into the data RAM.

4.2 Execution Model: Coroutines + Decoupled

We illustrate using an example. X-Cache walks over a hash-table bucket maintained as a linked list. On a meta access, X-Cache first checks the meta-tags for a hit. On a hit, X-Cache short-circuits the search and returns the data from the cache. The meta-tag hit is handled by a dedicated read port in the data RAMs; it’s fully pipelined and supports a 3-cycle load-to-use latency. On a miss, X-Cache needs to trigger the walker. First, we consider the programmer’s view (Figure 9) of the walker.
Programmer’s view

In the walker’s C code, there is one potentially expensive memory operation, specifically, on the first access to current.key, the condition that checks to see if the key at the current node matches the meta key. Once the cache-block corresponding to this node is loaded, the subsequent access to current.next and current.payload are accessed with minimal latency. We break the C code into a series of coroutines at the long-latency events (see Figure 9: FSM). The main purpose of this step is to convert stall events into yields. Yields release the controller pipeline, enabling us to multiplex multiple walkers cooperatively. We provide a table-driven template to help the programmer develop walkers. Each line in the coroutine description specifies a transition. It includes the current phase/state of the walker, the event that triggers the transition, the set of actions that need to be executed, and the next phase/state of the walker. The action sequence is run in program order (left-to-right). We compile this specification to a routine table and microcode RAM. The routine table is a two-dimensional array (see Figure 9: Execution Model). The rows of the routine table correspond to the coroutine states; the columns correspond to the events that can occur. Each entry is a pointer to a routine in the microcode RAM.

Execution sequence

From the programmer’s view, each meta access is driven by a logically separate walker, and within each walker, the execution is event-driven. All walkers are physically multiplexed on a single hardware pipeline. The hardware pipeline (see Figure 8: Pipeline) consists of two parts. The front-end serves as the event loop. It monitors the message buffers (internal and external) for events and wakes up one active walker per cycle. The current state of the walker is held by a combination of structures: the meta-tag array and the X-registers (a set of temporaries allocated for the duration of the walker).

The back-end serves as the routine execution pipeline; it is a conventional in-order pipeline. On wakeup the [state, event] pair index into the routine table, and retrieve a pointer to microcode RAM. Each routine, once triggered, will run in a non-blocking fashion. It starts execution from the pointer provided by the routine table, the logical “PC”. It runs each action in order and terminates with an update to the next state in meta-tag or X-register.

We now describe the execution step-by-step. 1 As it is a miss, we kickstart from the default state. The MISS routine allocates an entry in the meta-tag and the data RAMs for the data element, enqueues a Ptr (calculate pointer) event, and transitions to the Agen state. In this state 2 the walker is ready to check if the current node contains the key. However, the required access to current->key will be expensive (might be a cache miss). The step, therefore, issues a DRAM fill on current and yields a hardware pipeline. To track intermediate state while dormant, routines allocate temporary X-register to store the access key and the address of the DRAM refill being waited on. Active walkers, on other meta elements, can proceed to use the hardware pipeline. In 3, when Current block arrives from the DRAM, the walker peeks and extracts the block’s key. In 4 the match checks the block’s key against the meta-tag in X-register. If the key matches, the walker copies the block into the data RAM, releases the X-register’s entry, and terminates. Otherwise, updates current and continues. Note that the match condition determines the next state.
5 DSA Walker Examples

This section provides an overview of integrating X-Cache in DSAs. Table 2 summarizes the features of X-Cache that are most relevant to each DSA.

This section presents the details for the walkers of two families: i) Widx [18]: a DSA for relation-database hash indices. And ii) SpArch [37]: a DSA for outer-product sparse GEMM. We walk through their data orchestration pipeline and discuss their access patterns. We then describe their walkers in X-Cache. We intend X-Cache to augment (not replace) scratchpads and streams in DSAs. X-Cache targets access patterns that currently do not have a scratchpad/stream solution. The DSAs described below partition the data and use streams for one of the data structures, while employing X-Cache for the other. This is the efficient and performance-approach, as the meta-tags are not required for dense affine accesses. Our primary focus is to demonstrate that X-Cache is reusable and portable across the dynamic access patterns included in multiple DSAs.

Table 2: X-Cache features benefiting DSAs

<table>
<thead>
<tr>
<th>DSA</th>
<th>Tag</th>
<th>Preload</th>
<th>Coupling</th>
<th>Data</th>
<th>DS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Widx</td>
<td>Key</td>
<td>No</td>
<td>Coupled</td>
<td>Rid</td>
<td>Hash Table</td>
</tr>
<tr>
<td>DASX</td>
<td>Key</td>
<td>Yes</td>
<td>Decoupl.</td>
<td>Rid</td>
<td>Hash Table</td>
</tr>
<tr>
<td>Graph</td>
<td>Node Idx</td>
<td>No</td>
<td>Decoupl.</td>
<td>Event</td>
<td>Graph</td>
</tr>
<tr>
<td>SpArch</td>
<td>Col Idx</td>
<td>Yes</td>
<td>Decoupl.</td>
<td>B.Row</td>
<td>CSR</td>
</tr>
<tr>
<td>Gamma</td>
<td>Col Idx</td>
<td>Yes</td>
<td>Decoupl.</td>
<td>B.Row</td>
<td>CSR</td>
</tr>
</tbody>
</table>

Widx: Meet the Walkers [18]

The data structure that X-Cache has to walk is a hash-index in the database. In hash-indexes, each bucket is a chained list. The original Widx paper did not employ a domain-specific cache; it relied on an address-based cache and, hence, always walked. X-Cache makes a key improvement over Widx (see X-Cache in Figure 10a) — it caches the actual nodes in the hash table and tags them with the hash keys. The DSA datapath interfaces with X-Cache by issuing meta-loads and stores the keys in the hash table. On a hit, X-Cache eliminates the hashing (which could be up to 60 cycles in some TPC-H queries) and short circuits the walking. On a miss, the walker uses the address of the hash table (table) and a key to search for the Rid (row id). Note that X-Cache itself is caching the actual hash-index entries (not the row in the database). We break down the walker into multiple coroutines: i) In the first state walker hashes the key to obtain the index of the bucket (IDX). With the index and table of bucket root nodes, it uses a simple function to access the bucket root (META). Then it iterates over its nodes and loads them (AREF, state: DATA) to find the matching record (MATCH). Representing the walker as a state machine has two benefits i) each state explicitly indicates the type of hardware module required (e.g., IDX and AGEN require ALUs, while META and AREF RAM ports). This helps schedule away port conflicts within the controller, ii) we can improve memory-level parallelism with multiple keys actively walked independently.

SpArch: Outer-product sparse GEMM [37]

[37] implements an outer-product-based matrix-matrix multiplication between two sparse matrices (A×B). In each multiplication round, SpArch streams in a subset of columns of the multiplier matrix, A, stored in the CSC format. The multiply phase consists of a set of cross products on the pairs of columns (from A) and rows (from B). In SpArch, the data path uses the row index of an element in the multiplier matrix to index and fetch the corresponding row of the Matrix B, stored in the CSR format, from memory. As the
non-zero pattern in matrix A varies, the rows of B required to be kept on-chip also varies. Thus, SpArch needs a preload walker that runs ahead in decoupled fashion and caches the required rows. X-Cache caches a variable number of rows based on the number of non-zeros in the matrix A's column. The walker logic is quite similar to tiled DMAs, except the tile size varies based on the number of non-zeros in the required row. We initially read the row_ptr metadata of corresponding row (META) and set up a tiled refill. The walker generates addresses (AG) and fetches the consecutive elements from DRAM (DATA), and stores them in cache blocks. The meta-tag, in this case, is row ids of matrix B. There are certain key differences with Widx’s walker i) The data fill (DATA) fetches an entire row of matrix B, which consists of multiple elements, while in Widx’s case, it is a single node. ii) The number of blocks to be cached depends on the number of non-zeros in the row. iii) There could be multiple potential hits (a row split across multiple cache blocks), and all blocks are serially returned to compute datapath.

6 X-Cache Hierarchies

In this section, we consider the composability of X-Cache and its interactions with address-based caches. We present three systems i) Multilevel X-Cache (MX) ii) X-Cache with Address cache (MXA), and iii) X-Cache with streaming (MXS).

![Diagram of X-Cache Hierarchies](image)

Figure 11: Fitting X-Cache in a memory hierarchy or next to a DSA which needs streaming too. Blue regions: Meta is used. Red regions: Global addr is used.

In practice, X-Cache with streaming (MXS) is perhaps the most common. Many DSAs we studied employ this approach. In this case, the DSA explicitly partitions the data based on the access pattern. For instance, in SpArch (Figure 10b), matrix A is streamed in from the DRAM, while matrix B exhibits dynamic accesses and needs X-Cache. DSAs use global addresses for matrix A while using meta-loads (row-col ids) for the latter. Leveraging access pattern knowledge to partition the data structures is more area and energy efficient than using only X-Cache.

Multiple levels of X-Cache can also be hierarchically organized (MX). This is made feasible cause the metadata in a DSA is a global space (just like addresses). Each data element has a unique meta-tag associated with it that is common across the X-Cache hierarchy. The upstream L1’s X-Cache includes no walker. Similar to a conventional cache, it requests a meta-tag at a time from the downstream X-Cache. Only the last-level X-Cache includes a walker and address-translation, since it interfaces with the DRAM.

X-Caches can also compose with address-caches (MXA). In this case, X-Cache is the closest level to the DSA datapath, and address-based caches complete the lower levels. X-Cache will walk and generate addresses at the boundary. The address-cache simply sees a stream of cache line requests. The address cache itself handles a block at a time, while the X-Cache could proactively refill multiple blocks (e.g., entire matrix row). The address-cache is non-inclusive with X-Cache, since they use different namespaces.

7 Methodology

We use TSIM shorturl.at/iBIJ2 to drive cycle-accurate RTL simulation. TSIM is a library that manages communication between a host machine and the DSA RTL model. It provides a simple API to reset, send messages, and load/run programs on the DSA RTL. It also attaches the DSA to the DRAM model. We enclose our DSAs in an AXI shell to communicate with the DRAM. TSIM and Verilator provide a flexible interface that enables the integration of simulation models at multiple levels of abstraction. We implement the controller and X-Cache in Chisel and generate the Verilog. Verilog translates the Verilog into a cycle-accurate simulator and exposes the memory bus ports through the direct-programming interface. This interface permits us to implement the DSA's functional behaviour at a higher level of abstraction (python/c++) and interface only the loads and stores. The DRAM is modelled as another layer attached to the TSIM driver. We use DRAMsim2 to simulate the DRAM. Each of the models is driven using tokens.

7.1 X-Cache Setup

X-Cache is a toolflow for rapidly constructing domain-specific cache controllers. There are two major parts to X-Cache toolflow (Figure 12): i) A configurable hardware generator that lets the architects rapidly create domain-specific tags and expose domain-specific loadstores to interface with compute datapath. ii) A compiler that combines DSA-specific walking and cache management FSMs, and translates them into a microcode binary that runs on a programmable controller. The generator has been implemented as a Chisel library. It hides all the implementation details of the controller microarchitecture while exposing a simple architectural model that the designer can tune. Figure 13 summarizes the top module of X-Cache generator. There are two major segments: i) IOs: X-Cache interfaces with other components through a set of parameterized message bundles, i.e., latency-insensitive queues. X-Cache includes a base set of I/O to interface with the DSA datapath (MetaIO), DRAM bus, and other X-Cache or address caches. Based on system configuration and hierarchy, we instantiate the required I/Os. ii) Modules: X-Cache also exposes the parameters of the individual components from Figure 8.
The parameters in these modules are optimized for specific target patterns.

We now provide intuition on how each of these parameters influences the design: i) #nExe (line 22): This defines the number of action executors and consequently the number of actions that X-Cache can run concurrently in each cycle across all of the walkers. It determines the throughput of the controller under an ideal memory setup. The #nExe should be set to maximize the number of DRAM refills. ii) #nActive (line 23): Routines suspended on long latency events rely on X-registers to maintain the walker state. The number of the X-register determines the number of concurrent walkers. This, in turn, affects the number of refills in-flight and memory-level parallelism. iii) Meta-tag and Data RAM geometry: The meta-tag and data RAMs are decoupled and independently parameterized. Meta-tags are only needed for associative searches and optimized for underlying search patterns. For instance, in the case of the GraphPulse [30], a direct-mapped cache suffices. The cache is preloaded once, and then the accesses happen in arbitrary order. iv) #wlen: We bank and stripe a data entry across multiple sectors based on the number of words to be supplied on each hit. v) The rTable, trigger and microcode RAM (line 15,17,19) sizes are implicitly set based on the walker coroutines. When we compile them down and encode them, we determine the number of entries required. The structures implicitly scale up or down based on walker FSM complexity.

Table 3 lists the parameters we use for each DSA we evaluate. They have been obtained by sweeping different cache configurations and studying the hit rate and average memory access latency. It is not our goal to find the best controller configuration for each DSA. Here we simply want to isolate the performance impact of walker and controller implementation without being influenced by geometry variances.

### 7.2 DSA Workload Setup

Table 3 summarizes the pareto-optimal geometries we evaluate for each DSA. Here, we maintain the same geometry for the address-based cache, X-Cache, and baseline DSA to ensure a fair comparison. X-Cache’s chisel generator is highly parameterized, permitting us to generate different cache geometries. We now describe the setup for each workload.

<table>
<thead>
<tr>
<th>DSA</th>
<th>#Active</th>
<th>#Exe</th>
<th>#Way</th>
<th>#Set</th>
<th>#Word</th>
</tr>
</thead>
<tbody>
<tr>
<td>Widx</td>
<td>16</td>
<td>2</td>
<td>8</td>
<td>1024</td>
<td>4</td>
</tr>
<tr>
<td>DASX(Hash)</td>
<td>16</td>
<td>4</td>
<td>8</td>
<td>1024</td>
<td>4</td>
</tr>
<tr>
<td>SpArch</td>
<td>32</td>
<td>4</td>
<td>8</td>
<td>512</td>
<td>4</td>
</tr>
<tr>
<td>Gamma</td>
<td>32</td>
<td>4</td>
<td>8</td>
<td>512</td>
<td>4</td>
</tr>
<tr>
<td>GraphPulse</td>
<td>16</td>
<td>4</td>
<td>1</td>
<td>131072</td>
<td>8</td>
</tr>
</tbody>
</table>

**Table 3: X-Cache design parameters per DSA.**

the hash-index table. The meta-loads to X-Cache include the keys needed to be searched in the index table. If the key is not found in the meta-tag, X-Cache hashes and walks to search for the RID (primary keys) in the corresponding index tables.

GraphPulse [30]. Workload: PageRank. Inputs: p2p-Gnutella08: \(N = 6.3K, \ NNZ = 21K\), Web-Google: \(N = 916K, \ NNZ = 5.1M\). The event queue is substituted by the X-Cache. The incoming events are generated by the PEs. These events hold an id (row, bin, column) and a payload. The id probes the meta-tag memory. If the id does not exist, we insert the event by putting the payload in the data memory. Otherwise, we merge the payload of the incoming event and the existing entry using an add operation.

DASX [22]. Workload: MonetDB and TPC-H. DASX [22] is a data structure iterator. The DSA references the keys for each object while hardwiring the address generation, not unlike a TLB walker. The execution proceeds in refill-compute-update rounds. DASX’s collector runs ahead of the compute unit to refill multiple objects into a hardwired object cache. Subsequent accesses are cache hits. The cache is reloaded once a round is complete. Here, we study the hash-table, which uses preloads a set of hash keys. We use the same dataset from MonetDB as Widx.

SpGEMM: SpArch [37] and gamma [35]. Input: p2p-Gnutella31. \(N = 67K, \ NNZ = 147K\). The input to the X-Cache would be the row number of matrix B. The walking is comprised of accessing the B.row_pt array to determine which elements from the B.value array should be loaded and bringing them to the cache. Gamma uses the Gustavson algorithm for sparse GEMM.

8 Evaluation

We answer the following questions: i) How does X-Cache compare in performance to address-based caches and the baseline DSAs that employ a customized on-chip RAM? ii) What is the power overhead of caching compared to an ideal scratchpad? Note that these DSAs cannot use scratchpads. Hence, we isolate the cost of data RAMs only (which even a scratchpad would require). iii) What is the power overhead of a programmable cache controller? What is the overhead for the different controller components when synthesized on an FPGA and ASIC? A summary of our findings:

- X-Cache can be ported and reused across multiple DSA families. We create caches for four different DSA families: Sparse GEMM [35, 37], GraphPulse [30], DASX [22], and Widx [18]. We are the first to provide a common storage idiom for these DSAs.
- Our RTL cycle-accurate simulations reveal that X-Cache is competitive with the hardwired DSAs; no performance loss and up to 50% gain (compared to specific DSAs). X-Cache outperforms sized address-based cache by 1.7×.
- An address-based cache consumes 26 — 79% more power than X-Cache. The cache controller itself requires ≃ 24% of the total cache power (including the walking logic). Note that the walking logic is accounted for within the datapath in the baseline DSAs.
- The programmable controller that enables X-Cache to be portable across DSAs requires between 1.4% — 8% of on-chip power. The tags on the cache require between 1.4–6.5% of the data SRAMs.
- We synthesized our design on the FPGA and ASIC, all the way to GDS. It required less than 7% of a moderate-sized FPGA, making it applicable to DSAs mapped to FPGA as well. At 45nm, the controller occupies 0.1mm² (a 256K cache requires 1.1mm² just for the data RAM and tags).

8.1 Performance Evaluation

**Result:** X-Cache delivers 1.7× improvement over address-based caches. X-Cache short-circuits the walks with meta-tags and reduces the number of nested memory accesses by 2–8×. **Result:** On DSAs with expensive hash calculations, X-Cache delivers a 1.2× performance improvement relative to the baseline DSA. Meta-tags directly cache the keys in a hash table. On hits, they eliminate the need for hashing to find the buckets entirely.

Figure 14 compares the runtime of X-Cache against different DSAs under two scenarios. First, we compare the speedup of X-Cache against the baseline DSA that uses a hardwired custom on-chip RAM (black bar). We also compare X-Cache address-based cache of the same size. We assume the address-based cache includes an ideal walker, i.e., the walker makes orchestration decisions similar to X-Cache or DSA but incurs zero cost, i.e., all performance differences are a consequence of meta-tags, not orchestration logic.

![Figure 14: Left Axis: X-Cache Runtime vs Other DSAs (Lower is better). Right Axis: # of memory accesses of address cache (normalized to X-Cache). Higher means address cache makes more accesses. N/A: Graphpulse cannot be supported by address cache, and we only compare against baseline DSA.](image-url)
**X-Cache**’s performance is competitive with baseline DSAs. In the case of Widx, it can even achieve a 1.54× speedup. **X-Cache** achieves speedup in all the target database queries compared to Widx. On TPC-H-19 and TPC-H-20, **X-Cache** exhibits an even higher speedup. This is because of the cycles required for bucket index calculations. TPC-H-19 and 20 include *string-based* keys, their indexing stage, which is hashes this string and would require 60 cycles. In **X-Cache** on a hit, the load-to-use latency is 10× lower than Widx; since it does not require any hashing. **DASX** is similar to the Widx, except the hashing is coupled with walking, so **X-Cache**’s gains are higher.

Compared to address-based caches, **X-Cache** improves performance by 1.7X on average. Address tags encounter a significant increase in the number of DRAM accesses resulting from nested walks; ≃ 6.5× more than **X-Cache** (see Figure 14: Memory Accesses Y-axis). The extra accesses result from nested walks, which increase the footprint of the DSA and cache miss rate. Address-caches walk even when the data is already in the cache. In the case of the Widx and DASX, the root node of a bucket has to be loaded to find any element. On a miss, **X-Cache** will be limited by memory bandwidth and latency. However, it will minimize the number of DRAM accesses and achieve higher bandwidth utilization. In **Gamma** and **SpArch**, even though the meta-data itself is more regular (an array of index pointers), an extra DRAM access is required to load the start pointer of the Row.

### 8.2 Power Breakdown

**Result:** Address-based caches require 26—79% more power than **X-Cache**. **Result:** The **X-Cache** controller spends 24% of the cache power (including walking and tags). **Result:** The programmable RAM within the controller only require 6%, the meta-tags ≃ 7%. The remaining 11% is required by the walking logic, which would be accounted for within the datapath of the baseline DSA.

In this section, we study the power consumption of **X-Cache**. To calculate the power usage, we split up **X-Cache** into logic, registers and RAM components. The primary RAM components in **X-Cache** are the data array, meta-tag array, and the routine ROMs. For these components, we used a modified version of CACTI that models RAM arrays accurately [https://github.com/bespoke-silicon-group/bsg_fakeram][24]. For modelling logic, routines and actions, we used numbers from validated logic synthesis. We use an event-driven model to estimate the power of the controller datapath. The activity factors were obtained from the RTL simulation. The details of the modelled logic using an open-source 40nm standard cell library are listed in Table 4. The L1 cache power was modelled using CACTI 6.5; we use serial mode to ensure fair comparison against **X-Cache**.

We also compare the power consumption of **X-Cache** against an address-based cache. As shown in Figure 15, address-based caches consume 26—79% more power than **X-Cache**. The main reason is that we eliminate the walking and reduce the number of on-chip accesses. Figure 15 compares the power usage of on-chip RAMs against the controller and address generator. We find this to be 2—8% in the DSAs we study. This effectively measures the overheads of a reusable idiom since the programmable controller is a central requirement for portability across DSA (each requiring a specialized walker). This is a conservative estimate since we are assuming the cost of a hardwired walker to be zero.

Figure 16 breaks down the power consumption of the RAM components and the controller in **X-Cache**. As it shows, the central portion of the power is being used with on-chip data storage. Also, on average, 24% of the power consumption of **X-Cache** is consumed by the controller. Moreover, the cost of the Routine RAM, which is the primary difference between a programmable controller and a hardwired cache, is less than 4.2%.

### 8.3 Performance Exploration

**Result:** As the hit rate increases, **X-Cache** will achieve higher performance relative to the DSAs by eliminating address-generation...
and walking. Also, design exploration shows that access pattern influences whether a DSA can take benefit of a larger X-Cache.

In Figure 17, the runtime of X-Cache has been compared against the Widx for TPC-H:22 for different percentages of the on-chip data. As it depicts, as the percentage and, consequently, hit rate, goes higher, the benefit of using meta-tag against address-tag would be more evident. A higher hit rate reduces the latency for the accesses to the DRAM, which comprises the dominant portion of the runtime. Besides, using meta-tag reduces the load-to-use latency in comparison with the address-tag.

![Figure 17: X-Cache Runtime Vs Widx (runtime normalized to data in DRAM)](image1)

Figure 17: X-Cache Runtime Vs Widx (runtime normalized to data in DRAM)

Apart from the memory-related parameters, the number of ways and the number of sets, X-Cache has two primary parameters that should be set for each benchmark. Figure 18 demonstrates the sweeping of these two parameters for two different datasets of GraphPulse and Widx. For GraphPulse: p2p-Gnutella08, increasing the #Active and #Exe could diminish the runtime by half compared to the baseline design. However, for Widx: TPC-H-22, increasing mentioned parameters results in at most 10 percent of speedup. This observation could be explained by the behaviour of these two DSAs. As it has been elaborated in §5, the bottleneck of the performance of Widx is DRAM access. In addition to this, even for higher hit rates, meta hit is being used for returning the requested data. Consequently, increasing the parallelism in X-Cache would not have a significant impact on the overall runtime. On the other hand, GraphPulse takes benefit of higher #Exe in X-Cache. Although the DRAM access is still the dominant part of the latency, increasing the parallelism in X-Cache brings down the latency in the event generating stage of GraphPulse.

8.4 Synthesis Results

Result: Synthesis results are extracted from Quartus II V.13 for #Exe=4 and #Active=8.

We synthesized the generated X-Cache controller using Quartus II version 13.0. We set the #Exe for this synthesis on 4 and #Active on 8. In Figure 19, we have breakdown the register utilization and logic utilization of the generated RTL. As shown, X-Reg uses the most register, and Action-Executor units use the majority of the logic in X-Cache. We also used OpenROAD [2], an open-source RTL-to-GDS tool for ASIC synthesis X-Cache. Figure 20 depicts the controller with #Exe=4 and #Active=8. Under 45nm technology, the total area required is 0.11mm² and 65K cells. A 256K RAM under 45nm technology require 0.8mm².

![Figure 18: Sweeping Design Parameters](image2)

Figure 18: Sweeping Design Parameters

![Figure 19: FPGA Synthesis](image3)

Figure 19: FPGA Synthesis

![Figure 20: ASIC Layout. X-Cache controller (no RAMs)](image4)

Figure 20: ASIC Layout. X-Cache controller (no RAMs).

9 Related Work

There have been multiple proposals rethinking address-based caches. Stash [21] carves out portions of the address space. Hits to the scratchpad portion directly access the data RAMs; misses
implicitly fetch the data. Jenga [33] and Hotpads [34] organize the hierarchy of caches as a collection of SRAM banks. GPUs [16] and many cores use software-managed scratchpads. These approaches target CPUs or GPUs that role in address-translation and walking into the software. Also, prior approaches need to translate metadata to addresses (either local or global). Our observation is that if the on-chip data tags can be explicitly set to DSA-specific metadata, we can eliminate address generation (similar to a TLB).

There has been extensive work in combining the benefits of DMAs and scratchpads. Most recently, buffets [28] and Ax-DAE [5] developed a scratchpad+DMA storage idiom that can be reused and ported across DSAs. Both can achieve high performance when the access pattern is static, and the order of accesses is known up-front. They studied DSAs that work on tile-based data structures. The underlying orchestration pattern is hardwired. Ax-DAE is also limited by high-level-synthesis tools that do not support parallel accesses to indirectly indexed data. Finally, prior proposals are closely coupled and target static access patterns, affine types, and regular computation.

Finally, Widx [18] and DASX [22] accelerated the walk over data structures. Widx created an enhanced classic RISC pipeline and compiled down walkers to imperative binary. Ax-DAE leverage FPGA high-level-synthesis. Widx stored data in an address-based cache, and DAE stored the data in a scratchpad. In both cases, the storage is decoupled separately from the walker logic. The walker interfaces with the storage through an address-based interface, either explicitly (Ax-DAE) or implicitly (Widx). In either case, this leads to loss of locality for metadata accesses and multiple address generations are required during the walking phase. Patch memory [7] targeted image processing pipelines that need tiling. The DSA-expert has to define the loop order, and the patch runs ahead in a decoupled manner. It cannot be employed for the DSAs we explore. LEAP [1] and CoRAM [6] targeted BRAMs on an FPGA. Leap unified logically separate scratchpads into a tagged cache-like structure with implicit accesses. CoRAM created a framework for defining custom refill engines. Both do not include support for decoupled refill engines to prefetch data. They do not incorporate DSA-specific tags and require multiple accesses to satisfy a load/store.

10 Conclusion

We develop X-Cache, a reusable caching idiom for domain-specific architectures. There are two key ideas in X-Cache: i) Meta-tags: DSA-specific tags implicitly cache the elements in the indirectly-indexed data structure. Meta-tags reuse the elements and short-circuit the data structure traversals. ii) Routines: a programmable microcode engine that runs the walkers and orchestration state machine. We are the first to create a high-performance cache controller, targeting DSAs, with support for implicit accesses, decoupled miss walkers, parallel refills, and pipelined data and tag management. We will be open-sourcing the chisel generator, a compiler to translate walkers to microcode, and cache designs for five DSAs.

References


