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

by

Ali Sedaghati

B.Sc., University of Tehran, 2019

Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of
Master of Science

in the
School of Computing Science
Faculty of Applied Sciences

SIMON FRASER UNIVERSITY
Spring 2022

Declaration of Committee

Name: **Ali Sedaghati**

Degree: **Master of Science**

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

Committee: **Chair:** Keval Vora
Assistant Professor, Computing Science

Arrvindh Shriraman
Supervisor
Associate Professor, Computing Science

Alaa Alameldeen
Committee Member
Associate Professor, Computing Science

Zhenman Fang
Examiner
Assistant Professor, Engineering Science

Abstract

With Dennard scaling ending, architects are turning to domain-specific accelerators (DSAs). Emerging DSAs work with sparse data [40] and indirectly-indexed data structures [20, 32]. They introduce non-affine and dynamic memory accesses [8, 38], and require domain-specific caches. DSA caches need to support custom tags, data-structure walks, multiple refills, and preload for the datapath. Prior works include ad-hoc caches, but they're inseparable from the underlying DSA and do not implement the cache controller.

This research proposes X-Cache, a reusable caching idiom for DSAs. There are three key ideas: i) *DSA-specific Tags:* The designer can use any combination of fields from the DSA-metadata for the tag. Meta-tag eliminates the overhead of walking and translating metadata to global addresses and improves load-to-use latency. ii) *DSA-programmable walkers:* It has been observed that a common set of microcode actions can be used to implement the DSA-specific walking, data block, and tag management. A programmable microcode engine has been developed to execute the data orchestration efficiently. iii) *DSA-portable controller:* 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. *X-Cache* outperforms address-based caches by 1.7 times and remains competitive with hardwired DSAs. Also, meta-tags save 26—79% energy compared to address-tags.

Keywords: Domain-specific accelerators, Domain-specific cache, Modular architecture, Coroutine execution model

Dedication

To
Hanieh
and
Elahe

Acknowledgements

My sincere thanks go to my supervisor, Dr. Arrvindh Shriraman, for his extensive support and guidance during my studies.

I would like to thank Milad Hakimi, my extraordinary friend and colleague, for all of his efforts and patience during this work.

I would especially like to thank Reza, Parmida, Amirali, Minh, Aditya, and Anagha for creating such a professional research environment. It was a great pleasure to work with you.

Many thanks to my amazing friends and family for all of their support whom, without, none of this would have been possible

And at the end, my special thanks to Hanieh, for being patient and supportive during all of the challenges we faced during past years. Thanks for being such a fantastic person!

Table of Contents

List of Tables

List of Figures

Chapter 1

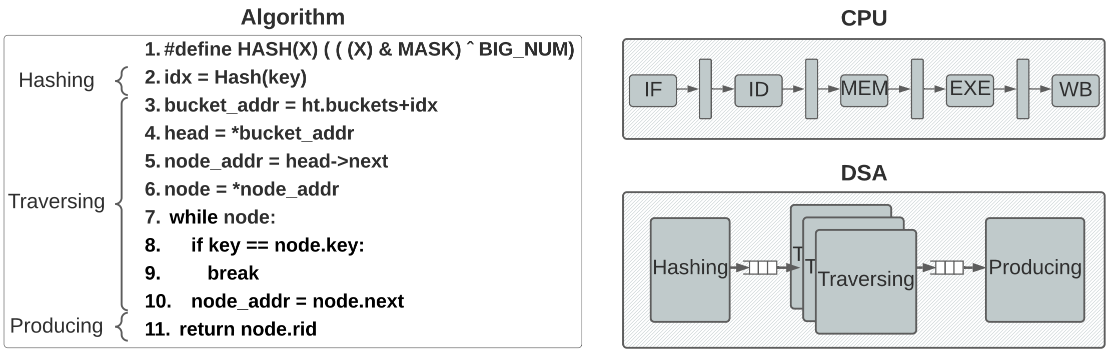
Introduction

Before the degradation of Moore's law and Dennard scaling, the rate of the increase in the performance efficiency of the general-purpose processing units was promising. The performance of the emerging processors has been escalating while exploiting the new technologies has been lowering power consumption. However, this exponential growth was hard to be maintained. After 2005, the end of the Dennard scaling, and also with the recent sharp decline in the performance increment rate, architects sought new approaches to solve the problem. Instruction-level and data-level parallelism, done by increasing the number of cores in the processors and vectorization of the execution stage, could not postpone the degradation for more than a few years. So, specialization became the only path to cope with the slow rate of performance efficiency improvement [15]. The main idea was designing a domain-specific architecture (DSAs) for a single application instead of general-purpose designs. It helps the architects remove the overhead of generalization while designing hardware focuses on efficiently accelerating the execution of just one task. The mentioned efficiency is caused by: i) exploiting the domain-specific knowledge for designing architecture with an optimized resource allocation and utilization. ii) optimizing the data management on the memory side of the DSA, as they tend to lower the memory traffic by finding the optimal storage idiom while maximizing usage of the bandwidth of the DRAM.

This chapter will discuss the concept of domain-specific architecture and the characteristics of the emerging DSAs (§1.1). Then, *X-Cache* would be introduced, and its modularity will be elaborated (§1.2). And finally, we will discuss the contributions of this dissertation (§1.3).

1.1 Domain-Specific Architectures

General-purpose processors have to endure the cost of the generality. Though their architecture allows being extremely programmable to cover a broad range of applications, the generalization would expose such an architecture to some overheads to support this characteristic. The cost is mainly caused by the stages of fetching the instruction, decom-



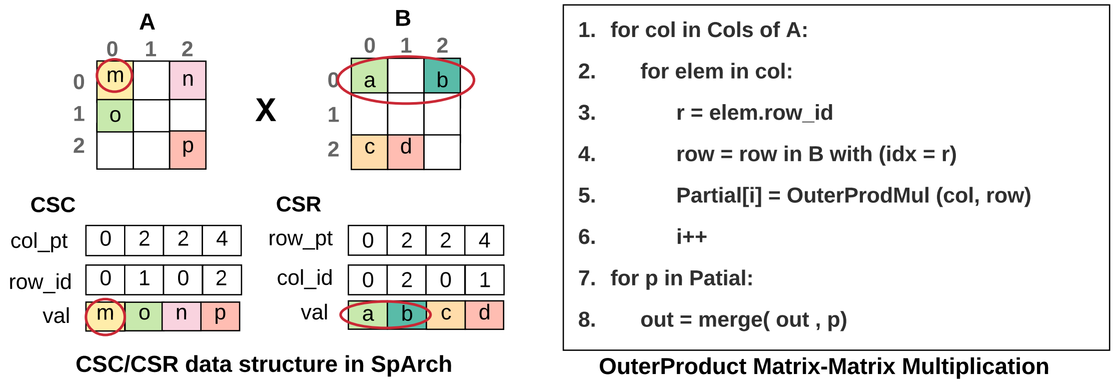
```
#define HASH(X) ( ( (X) & MASK) ^ BIG_NUM)
idx = Hash(key)
bucket_addr = ht.buckets+idx
head = *bucket_addr
node_addr = head->next
node = *node_addr
while node:
    if key == node.key:
        break
    node_addr = node.next
return node.rid
```

Figure 1.1: CPU VS DSA

posing the instruction, and extracting the information in it. All of these overheads have been imposed on general-purpose processors because of their generality. On the other hand, domain-specific architectures specifically focus only on one application. The general idea is to use knowledge related to one domain and exploit it to design efficient and to-the-point accelerators. Targeting one domain and taking benefit from prior knowledge of it directly affects resource allocation, resource utilization, and optimizations in the design process. With this in mind, not only DSAs completely eliminate all of the overheads imposed on the design because of the generality, but also, they are much more performance efficient. Moreover, architects tend to break down the application into smaller pieces and design a specialized architecture for each of these pieces. Figure 1.1 shows an example to elaborate on this idea. As depicted, an algorithm applies the hashing function on a single key and then traverses the hash-table to find the corresponding node. This algorithm will be compiled down to the binary. Then the binary code would be fed into the CPU and be run one instruction at a time. On the other hand, a specialized architecture could be designed for this application. The algorithm will be divided into three main stages, similar to the approach implemented in the [20], and one specific architecture will be designed to execute each of these pieces.

Furthermore, Dally et al. [9] and Hennessy-Patterson [16] 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 [7, 22] predominantly work with dense data and organize them into scratchpads with explicit DMA. However, emerging DSAs target a broader 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 [40], indirect-indices [32], or hash tables [20] for minimizing the footprint. In this light, getting an element's global address could not be done by simple addition, as it would have been possible if the data structure was dense. So, the DSA requires data structure traversal for calculating the global



```
for col in Cols of A:
    for elem in col:
        r = elem.row_id
        row = row in B with (idx = r)
        Partial[i] = OuterProdMul (col, row)
        i++
for p in Patial:
    out = merge( out , p)
```

CSC/CSR data structure in SpArch

OuterProduct Matrix-Matrix Multiplication

Figure 1.2: Dynamic data structure in emerging DSAs

Prefetch a row of B with $Row\_id_B = Col\_id_A$
(Col_ind_A: Column ind in A)

```
R_B = Row_id_B
row_addr = B.r_ptr + R_B
end_addr = row_addr +1
st   = *row_addr
len  = *end_addr-*row_addr
val_addr = B.val + st
for i in len
  b_vals[i] = *(val_addr + i)
return B.val[st:st+len]
```

Figure 1.3: Walker in SpArch

address. Figure 1.2 illustrates the CSR/CSC format used for storing the sparse matrices. As shown, for accessing the value of one single element, reading from *val* array, the two other arrays should be read first to get the correct index of *val*. While using this form of storage inflicts the mentioned overhead on the DSAs, it is vital for the algorithm. In Figure 1.2, the algorithm of outer-product matrix-matrix multiplication is depicted. As it shows, using the CSR/CSC format is necessary to take benefit from this algorithm. As if the data is stored in the dense format, multiple index matching, proportionate to the dimension of the matrix, is required to produce one single partial matrix. **ii) Dynamic accesses:** Both of the data structure and loop pattern cause emerging DSAs to have dynamic (i.e., indirectly addressed) and irregular non-linear accesses. Figure 1.2 elaborates on this matter. As the algorithm shows, the row index that DSA should fetch from matrix B depends on the element's column index in matrix A. So the process of address calculation should be done *dynamically* as the elements of matrix A are streaming in. For capturing the reuse, emerging DSAs, which have dynamic access patterns, use cache. [38] **iii) Walkers**: Since data is stored in non-linear data structures, a walker is required. Walker is a data-structure-dependent logic dedicated to traversing the data structure to find the needed data. The logic of the walker could be simply an addition if the data structure is an array. Alternatively, when the data structure is a linked-list, the logic traverses the linked-list nodes until it gets to the requested node. The walker shown in Figure 1.3 traverses the data stored in CSR/CSC format. Also, in some cases, the walker should preload the cache [32], or even execute multiple nested preloads [38, 40]. **iv) Data orchestration**: Finally, DSAs need to orchestrate cache replacement and refill with the compute datapaths explicitly.

1.2 Overview of X-Cache

In this dissertation, we present *X-Cache*, an architectural template for domain-specific caches and a toolflow to achieve DSA-specialization through the cache controllers. *X-Cache* deconstructed the caches into some primary aspects. It allows the toolflow to remain DSA-agnostic. At the same time, it covers all of the characteristics listed previously to be adaptable to target different DSAs with caches designed specifically for that domain. Some prior DSAs include cache-like structures [8, 38, 40] that are inextricably tied to the underlying DSA and do not implement the cache controller.

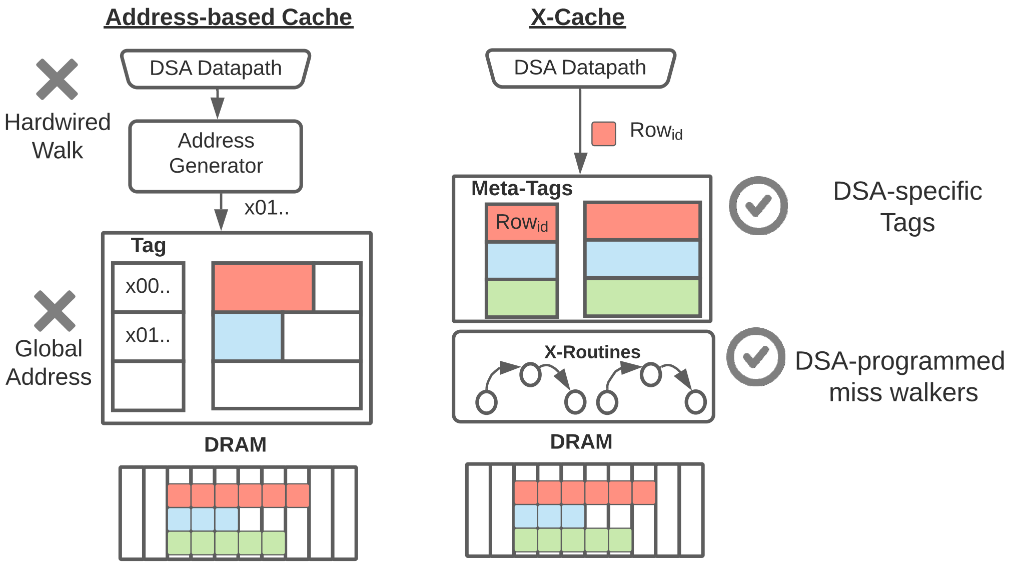


Figure 1.4: Address-based Cache vs. *X-Cache*

Figure 1.4 illustrates the challenges of incorporating a conventional cache in a state-of-the-art sparse matrix DSA: i) *Metadata to address translation (Wasted energy)*: The domain-specific notions that DSAs use are usually more abstract than the global addresses. For example, the sparse GEMM DSA works with row/col indices, while the address-based caches are based on the global addresses. So, for converting the abstract notions, row/col in this example, to addresses, logic would be required to take extra steps; in sparse matrices, this would require processing CSR/CSC metadata. ii) *Address-based tags (High load-to-use latency)*: Conventional caches are tagged by blocks' addresses. So, even when the requested data is residing in the cache, or in other words, it is a hit, it could not be determined immediately. Since the DSA works with domain-specific notions, the process of converting the abstract notion to address might take several cycles. E.g., even if the required row/col is cached, the request cannot be determined to be a hit until the address corresponding to the row/col is found out. It means that some extra accesses to the cache or even the DRAMs are required to traverse the vectors stored in CSR/CSC format to get the addresses of the elements. This would lead to much higher load-to-use latency. iii) *Walker logic (High NRE)*: Finally, significant design effort is required for creating custom walkers and implementing them for each DSA, which leads to high non-recurring expenses (NRE). Walkers also might need to handle parallel refills, nested walks, and pipelining. Additionally, the common attributes of walking, tag management, and data orchestration between different DSAs could not be easily exploited since each logic is tightly coupled with the targeted DSA.

X-Cache removes the memory layout and address-generation concerns from the DSA. It dispenses with the flat address space abstraction and prevents DSA from reading on manipulating 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 the cache tags. In contrast to the conventional caches, which strictly use a part of the global address for tagging an entry, *X-Cache* allows the user to use the domain-specific knowledge to find the proper sets of the domain-specific notions to be used as the *meta-tag* of the entry. For example, in sparse GEMM, the (Row, Col) in CSR/CSC representations will serve as the meta-tag. So, the compute datapath can use meta loads/stores and put the onus of locating the data on-chip on *X-Cache*. In this light, *X-Cache* is able to use global addresses only on misses, which requires walking the DRAM for the requested data. While prior DSAs [32, 38, 40] include fields that mirror the functionality of meta-tags, this dissertation is the first to define and generalize the concept. **DSA-programmable walking (X-Routine):** On a miss, *X-Cache* implicitly walks the data structure and finds the relevant data. As the targeted data structure of each DSA varies, each DSA probably requires a different walking logic. For supporting this characteristic, in *X-Cache*, each DSA can specify a custom walker. The key observation was that the walker could be implemented using a common set of microcode *actions* for walking and also for data and tag management. **DSA-agnostic controller architecture:** *X-Cache* exploits the observation that the walking and data orchestration of non-overlapping requests could be done simultaneously and as short-lived, non-blocking tasks. This observation has been captured by defining each task as a coroutine. Coroutines conveniently process multiple meta-tags in parallel (like cache blocks), are lockup-free, and schedule away controller conflicts. Also, the underlying controller and the abstraction used for defining the coroutines could be designed DSA-agnostic, which offers the opportunity of having a portable design across different DSAs.

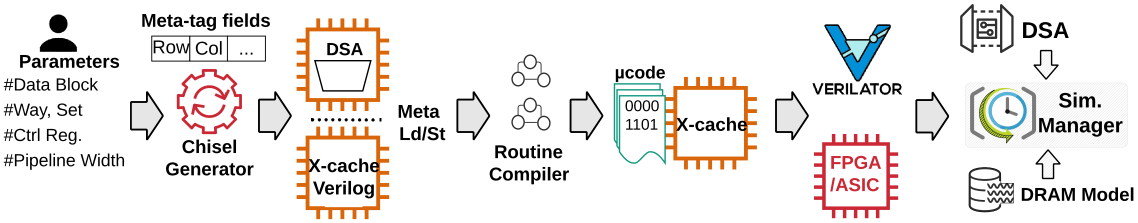


Figure 1.5: *X-Cache* Toolflow

1.3 Contributions

X-Cache is a toolflow for rapidly constructing domain-specific cache controllers. There are three parts to *X-Cache* (Figure 1.5): i) A configurable hardware generator that lets the architects rapidly create domain-specific tags and expose domain-specific loads/stores 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. iii) A coroutine model that helps designers specify the data orchestration in DSAs. Coroutines conveniently capture the underlying parallelism across meta-tags and help schedule operations in the controller while minimizing occupancy.

X-Cache has been evaluated by creating domain-specific caches for five different DSAs, SpArch [40], GraphPulse [32], Widx [20], DASX [24], and Gamma [38]. Both SpArch and Gamma can use the same *X-Cache* microarchitecture. And to achieve specialization, the controller was the only component that had to be changed, and it is possible by reprogramming it. It has been demonstrated that the meta-tags across different DSA families, vertex ids (GraphPulse), database keys (Widx), data structure indices (DASX), Compressed-matrices (SpArch and Gamma), could be auto-generated. Overall, *X-Cache*'s performance was comparable to the hard-coded caches in existing DSAs. In the case of Widx, *X-Cache* even improved performance by 50% by reducing the load-to-use latency. Compared to the best-performing address-based cache for each DSA, *X-Cache* improved performance by 1.7X. 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., the minimal penalty for being reusable compared to a hardwired design. Dissertation's contributions:

- A reusable caching idiom for domain-specific architectures supports dynamic access patterns and irregular data structures in emerging DSAs.

- A generalized concept of domain-specific tags (Meta-tags) and creating a high-performance programmable cache controller. It has been demonstrated that DSA walkers and orchestration can be expressed as a common set of microcode actions.

- A domain-specific cache targeting five different DSAs from three domains sparse-matrix computation, databases, and graph processing.

- Open-sourcing the Chisel generator, microcode compiler, and X-cache designs.
- Domain-specific caches improve performance 1.7× by short-circuiting data structure walks and reducing memory accesses by 2—8×. The cache controller requires 7% of on-chip energy, and tags only require 1.5–6.5% of data RAM energy.

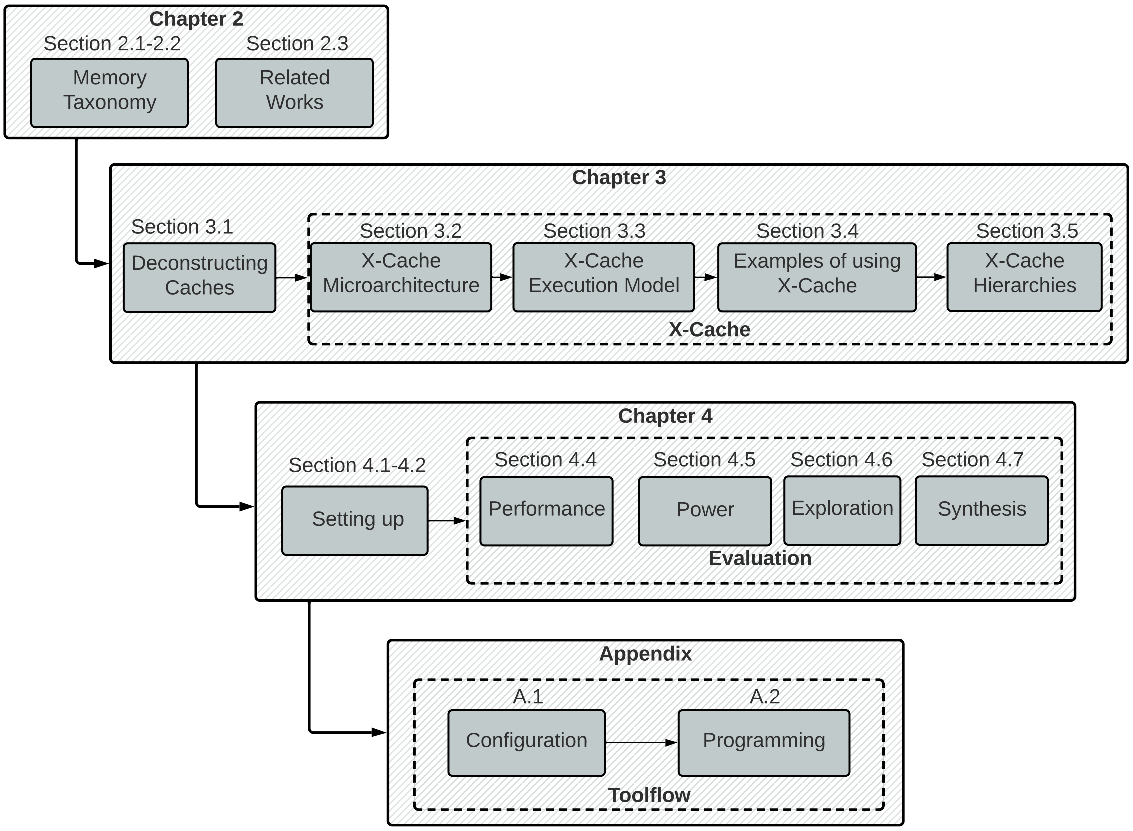


Figure 1.6: The layout of the Dissertation

1.4 Publications

This dissertation also includes our published paper at a peer-reviewed conference. For this paper, I have collaborated with my supervisor Dr.Shriraman and my colleagues, Milad Hakimi and Reza Hojabr.

- **ISCA 2022 — X-Cache: A Modular Architecture for Domain-Specific Caches [33]. Ali Sedaghati, Milad Hakimi, Reza Hojabr, Arrvindh Shriraman**

Construction of the toolflow, designing an architecture to support the reusable caching idiom, and implementation of the five DSAs using our toolflow, have been done by me. Also, Milad Hakimi, contributed to setting up the workloads and collaborated with me to evaluate them.

1.5 Dissertation Outline

The dissertation is organized as follows: Chapter 2 describes the related works and the motivation behind the work. Chapter 3 represents *X-Cache* and goes over a few examples to elaborate on the architecture and its usage. The evaluation of the work and describing the toolflow will be presented in Chapter 4. And Chapter 5 will outline the conclusion and future works. Figure 1.6 elaborates on the materials covered by each section.

Chapter 2

Scope and Related Works

In this section, we first discuss the vital criteria which should be paid attention to by the storage idioms targeting emerging DSAs. Then *X-Cache* would be compared against popular storage idioms in literature (§2.1). Also, we discuss why existing idioms do not support the dynamic access patterns exhibited by emerging DSAs, while that is the approach which *X-Cache* took (§2.2). Additionally, we elaborate on the related storage idioms discussed in the literature (§2.3)

2.1 Taxonomy

State-of-the-art approaches have sought to create storage idioms for static access patterns. Figure 2.1 captures the most popular storage idioms, scratchpads with decoupled DMAs, scratchpads with access-execute, and FIFOs. There are two fundamental limitations of other idioms i) they require metadata to address translation since they introduce additional address spaces. ii) they do not support dynamic access patterns and indirectly-indexed data structures. iii) they include a fixed-function or a hardwired walker. The walkers vary across DSA families since the underlying data structure is different, and even within family depending on computation order. The DSAs we study in this paper cannot incorporate these storage idioms; they require domain-specific caching. We also include address-based caches to help contrast.

We use the following classification: 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 work support decoupled fetches for static access patterns. However, emerging DSAs require support for dynamic-decoupled accesses. iii) **Dynamic vs. Static** In DSAs with static access patterns, the data access order is fixed, 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 data distribution in the input.

Moreover, Table 2.1 thoroughly compares different approaches proposed for the data storage of the DSAs, against each other and *X-Cache*. Various criteria have been categorized into two main categories, the targeting domain, and the design perspective. The criteria are listed as follows:

Granularity indicates the smallest size of the data which the storage idiom manages. While other storage idiom works with a fixed granularity, *X-Cache*'s granularity could be tuned by a domain expert to increase the on-chip memory utilization. **Meta-to-Address Translation** points out if the storage idiom is required to translate the domain-specific notions into the global addresses to respond to a load request. *X-Cache* eliminates it by exploiting the domain-specific tag for storing the data on-chip. This allows us to directly use the mentioned tag for responding to a hit request when the data is already available on-chip. **Behavior** is related to the access pattern of the targeted DSA. Similar to the storage idioms of caches, *X-Cache* targets dynamic access patterns. **Domain** shows how the data with which the DSA works is accessed. In contrast to the other storage idioms, *X-Cache* flexibly adapts itself to support different types of accessing the data, based on the requirements of the targeted DSA. **Coupling** indicates if the storage idiom is tangled with the DSA. While conventional caches are tightly coupled with DSA, e.g. the onus of address generation is completely on the DSA, *X-Cache* works in a decoupled fashion. So, the stages of generating the address, traversing the data structure, and fetching the data are managed by *X-Cache*. In this light, we can take benefits of being decoupled, similar to the scratchpad-based storage idioms, like Buffet [30], and also at the same time, target the DSAs they cannot support efficiently.

Trigger states how the DSA triggers the storage idiom. *X-Cache* is able to support different types of communication with DSAs. The concept of the *event* explained in §3, allows *X-Cache* to be initiated with various forms of triggers. **Walker** shows if the storage idiom is responsible for walking the data structure. *Coroutines* explained in Chapter 3.3, make it possible to implement walking logics as a part of the *X-Cache*'s controller. **Control** is related to the level of flexibility for implementing the control logic. Most of the existing storage idioms, use a hardwired and non-changeable controller in their design. For targeting a broader range of DSAs, some emerging storage idioms, moved toward allowing limited flexibility, using Fixed FSMs as their controllers. In contrast, *X-Cache*'s controller is programmable by a set of pre-defined **actions**, mentioned in §3, to support a broad range of DSAs while remaining efficient. **Energy** points out the energy usage of the storage idioms. As explained in §4.5, the energy usage of *X-Cache* is significantly lower than a fixed address-based cache. Also, the programmability of *X-Cache* imposes an extremely small overhead on the power usage. **Preload** shows if the controller supports the preloading before the data is needed for the DSA. Conventional caches need a separate controller for managing the preloading. Also, Scratchpad-based and FIFO-based approaches, support a limited form of preloading,

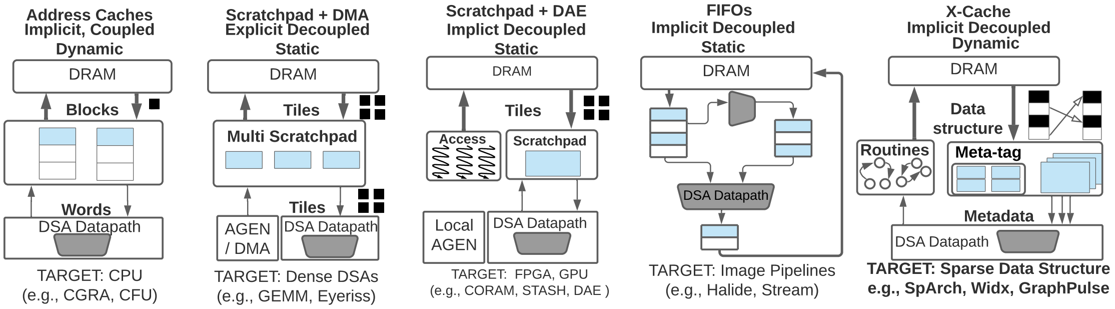


Figure 2.1: *X-Cache* vs. state-of-the-art storage idioms

Table 2.1: X-cache vs. state-of-the-art storage idioms (shaded cells indicate limitations)

		Caches [3, 12, 25, 28, 29]	**Scratch+DMA** Buffets [30] Stream [13, 27]	**Scratch+DAE** CoRAM [7] DAE [6] Stash [23]	**FIFOs** Spatial [21, 22] Pipeline [10, 17]	***X-Cache***
Target	Granularity	Blocks	Tiles	Word	Elements	DSA-specific
	Meta-to-Addr	Yes. Walking and translation always required.				No (only on misses)
	Behavior	Dynamic	Static pattern (affine)			Dynamic (DSA-spec.)
	Domain	—	Linear data structure		Stream	Flexible
	Addressing	Implicit	Explicit	Implicit	Implicit	Implicit
	Coupling	Coupled (load/store)	Decoupled	Coupled	Decoupled	Decoupled
Design	Trigger	Implicit (load-/store)	Explicit (datapath)		Implicit (push/pop)	DSA-specific
	Walker	No. DSA has to walk metadata.			Only FIFO	Yes (coroutine)
	Control	Hardwired	Fixed FSM	Hardwired	Hardwired	Programmable
	Energy	High	Low	High (threads)	Low	Low
	Preload	— (separate)	Limited (credit)		Limited (credits)	Yes (FSM driven)
	Orch.	Simple (load-to-use)	Ready/Valid. Fill or gather			Simple (load-to-use)

mainly for the linear data structure. On the other hand, *X-Cache* can take benefit of an FSM-driven preloading logic, defined as a *routine*, which can target a broad range of DSAs which need preloading. In the end, **Orchestration** focuses on when the data would be accessed. *X-Cache*, similar to the address-based caches, uses a simple load-to-use approach. Basically, one single event, mainly load, triggers a sequence of events required for responding to the request.

The shaded cells indicate the limited ability of different storage idioms to cover the corresponding criterion.

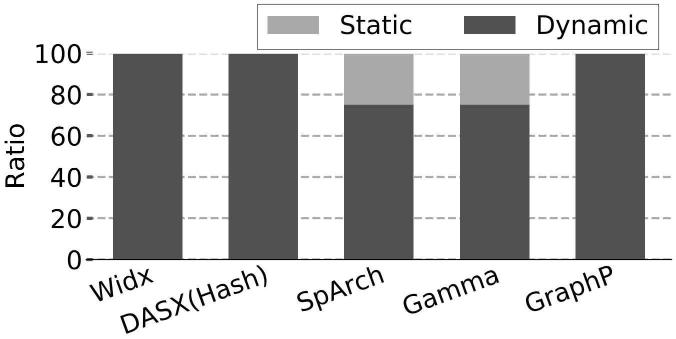


Figure 2.2: The percentage of the static and dynamic memory accesses in a DSA. Y-axis: Percentage of the ratio of static accesses vs. dynamic accesses. Result: Dynamic accesses dominate.

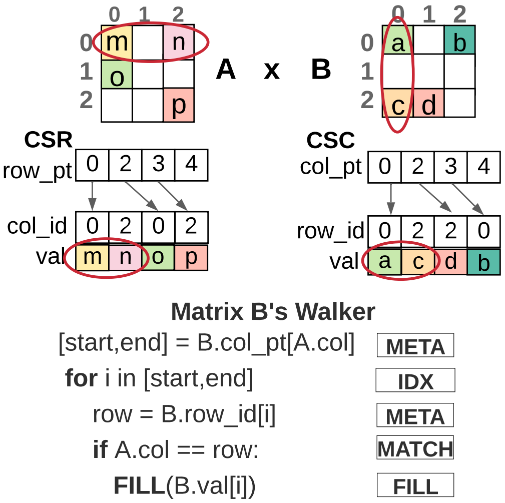
2.2 *X-Cache*'s storage idiom

In this section, we elaborate on our observation that for our targeted DSAs, which exploit all of the characteristics mentioned in §1.1, an implicit-decoupled storage idiom is the best approach. Also, we justify using the domain-specific notions instead of the global address for storing in *X-Cache*.

Why caches and not scratchpads

Summary: *Multiple DSAs in recent literature exhibit dynamic access patterns, require domain-specific caches, and cannot use scratchpads.*
State-of-the-art DSAs that we explore (e.g., SpArch, GraphPulse, Widx, Gamma) exhibit dynamic access patterns. We are not aware of any work supporting dynamic access patterns on scratchpads. Emerging DSAs address a broader scope of algorithms and work with sparse data (e.g., graphs, sparse matrices. Thus the computation's spatial-temporal mapping varies based on the inputs. The order in which data is fetched from the DRAM, the order in which the datapath accesses them, and the address access pattern all vary during runtime. Figure 2.2 plots the ratio of static to dynamic accesses in the DSAs we study. State-of-art scratchpad controllers ease the burden for static patterns [30]. A decoupled DMA engine refills data from DRAM in rounds or tiles and is responsible for pushing data into the compute units. Unfortunately, they only support dense data, which are iterated over with affine functions. Past proposals also include support for implicitly gathering and packing in data from the DRAM. Unfortunately, it is not clear how to support dynamic access patterns. Further, they require the DSA to use local addresses.

Figure 2.3 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 column 0, but for `A[0,1]` we refill B's column 1. Since B is sparse, DSA traverses indices (IDX) that store 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.



```
[start,end] = B.col_pt[A.col]      META
for i in [start,end]               IDX
    row = B.row_id[i]              META
    if A.col == row:               MATCH
        FILL(B.val[i])             FILL
```

Figure 2.3: Walkers in Sparse GEMM DSA (Inner Product)

Why not addresses?

Summary: *Caches and scratchpads are challenging to incorporate in emerging DSAs since they require metadata to address translation.*

Both conventional caches and scratchpads work with addresses. Caches require global addresses for the tag. Scratchpads [1,7] augment the address space with a local address range and eliminate the tags. Unfortunately, the use of addresses (global or local) wastes energy on multiple layers of translation from DSA-specific metadata and look-ups. The data-structure walks may trigger nested translations between metadata and data and may require multiple table look-ups. Caches also tie in data orchestration (e.g., what to fetch from DRAM?), the on-chip storage, and tags (how to look for the data). In the case of scratchpads, the DSA also needs to manage the data between multiple address regions. While support exists for static patterns [30], no solution exists for dynamic access patterns.

2.3 Related Works

There has been a long history of work on decoupled engines for shuttling dense tensors from DRAM onto on-chip SRAMs. This includes scatter/gather engines [11, 14, 19, 34], memory controllers [5], and tiled DMAs [1, 7, 30]. All designs introduce additional address spaces, either local [19, 30], carved from the global DRAM [34] or virtual [5] address. Thus, DSAs [38, 40] that include unordered and dynamic accesses would need to implement address-translation from meta-tags to these local addresses. Further, they can only support access patterns driven by affine loops and strides. We focus on indirect indexes, sparse tensors, and algorithms with conditional accesses. Also, the optimizations proposed by some of these works, cannot be applied to our explored DSAs, e.g., in [14] they assume all of the addresses are already available, but in some DSAs, i.e. SpArch, FiberCache, they have to be generated from the streamed data.

There have also been works on augmenting scratchpads in GPUs [23] and FPGAs [6, 7]. 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 heavyweight 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 walks elements in a specific order, they cannot accommodate dynamic input-dependent reuse behaviour (e.g., indirect indexes).

There has been extensive work in combining the benefits of DMAs and scratchpads. Most recently, buffets [30] and Ax-DAE [6] developed a merged 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 upfront. While they lack good performance, when targeting applications dependent on the dynamic address generation, they studied DSAs that work only on tile-based data structures. Additionally, 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.

Stash [23] attempts to merge the scratchpads and caches while exploiting the benefits of both of them. It manages it by carving out portions of the address space. All of the hit to the scratchpad portion directly accesses the data RAMs; misses implicitly fetch the data. Jenga [36] and Hotpads [37] organize the hierarchy of caches as a collection of SRAM banks in a hierarchical fashion. And manages orchestration of the data between different levels by adding some extra bits, or entries.

GPUs [18] 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).

In addition to these, some of the emerging DSAs mainly focused on walking the data structure. For instance, Widx [20] decoupled the hashing stage from the walking stage to reduce the critical path of their design significantly. And then they created an enhanced 2-stage RISC pipeline and compiled down hashing and walking logic to imperative binary. Doing so, allows them to target a wide range of hash-tables, while their design still remains specialized for this domain. Moreover, DASX removes the overhead of the walking from the DSA and manages it in its controller. Basically, they *collect* all of the required data ahead of time. So, during the computation phase, all of the DSA accesses could be responded with the on-chip data. They have Ax-DAE leverage FPGA high-level-synthesis. Widx stored data in an address-based cache, and DAE kept the data in a scratchpad. In both cases, the storage is decoupled separately from the walker logic. The walkers interface 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.

Furthermore, some new approaches explored replacing the global address for tagging cache lines, with more flexible and domain-specialized notions. Patch memory [8] targeted image processing pipelines that need tiling, and it specialized the storage idiom for 2-D and 3-D tiles. Yet they lack a high-level approach for walking the targeted data structures. In their work, 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, as the loop pattern of these DSAs is highly data-dependent. LEAP [1] and CoRAM [7] 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.

Also, the coherency controller is another aspect supported by the cache storage idioms. GEM5 [4] and their programmable coherency controller, using SLICC format [35], is an example of it. While the concept of the coroutine in *X-Cache* and defining walker within it, and SLICC, share some common characteristics, e.g., both rely on state-machine-based controllers and both exploit quite the same level of programmability, they're different in some major grounds. Not only they target two thoroughly different aspects of cache storage idioms, *X-Cache* focuses on walking and data orchestration whereas SLICC focuses on keeping the caches coherent, but also, some optimization is required for implementing the hardware-controller, which are not the main concern of SLICC, e.g. parallelism.

Chapter 3

X-Cache: Microarchitecture and Execution Model

In this section, the design choices we made while implementing *X-Cache* will be discussed (§3.1). Then, we go over the architecture of the *X-Cache* and elaborate on it(§3.2). In §3.3, the execution model of the *X-Cache* will be explained. Finally, we thoroughly explain the usage of *X-Cache*, using two examples on Widx [20] and SpArch [40] in §3.4.

3.1 X-Cache Overview: Deconstructing DSA Caches

We deconstruct the architecture of domain-specific caches and discuss the design decisions behind *X-Cache*. Emerging DSAs have exhibited considerable interest om caches [6,32,38,40], but are understandably cautious. Our thesis is that we should facilitate the adoption of domain-specific caches by deconstructing them into modular components. Deconstructing permits us: i) to be selective about which components in the cache should we make fixed-function (e.g., tag checks) and ii) achieve portability by enabling programmability in specific modules (e.g., walking).

3.1.1 Design choice 1: Address Tags vs. Meta Tags

In *X-Cache*, the DSA uses meta loads/stores that directly reference the elements in 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 3.1 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 TLBs tags; they short circuit the metadata-to-address translation and eliminate the nested data-structure walks. In contrast, the address-tags require many times more access for walking, even when the required element is cache resident.

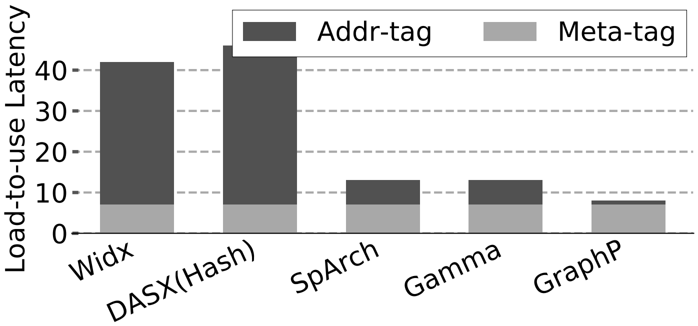
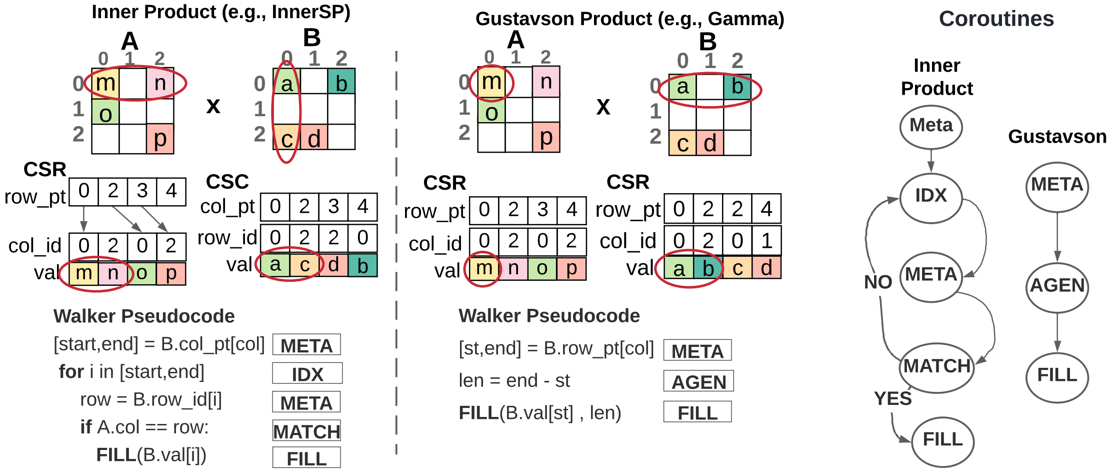


Figure 3.1: Load-to-use-latency. Address Tags vs. Meta-tags. Y-axis: Ratio of the load-to-use latency, in case of a hit, when using Addr-tag vs. using Meta-tag.



Walker Pseudocode

```
[start,end] = B.col_pt[col]    META
for i in [start,end]           IDX
    row = B.row_id[i]          META
    if A.col == row:           MATCH
        FILL(B.val[i])         FILL
```

Walker Pseudocode

```
[st,end] = B.row_pt[col]    META
len = end - st              AGEN
FILL(B.val[st] , len)       FILL
```

Figure 3.2: Walkers for two DSAs in Sparse GEMM Family

Implication: We need to create a framework that can rapidly create cache tags supporting the metadata specific to each DSA family. A DSA-family is a set of DSAs that share the same data structure e.g., sparse GEMM accelerators [31, 38, 40].

3.1.2 Design choice 2: Fixed vs. Programmable Walker

A central design consideration in *X-Cache*'s controller is whether to hardwire the walkers. To understand this, we consider two DSAs from the same family, sparse GEMM: inner-product and Gustavson product [38] (Figure 3.2). Both DSAs work with the same data structures (CSR/CSC compressed matrices) and use the same meta-tag, *(row,col)* of the non-zero element. 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 compute loop order.

The inner product performs dot product between matrix A and a tile of columns from matrix B. For each row in matrix A, the walker finds non-zero elements from corresponding col e.g., Row 0 in matrix A has non-zero at `A[0,0]` and `A[0,2]`. The walker traverses column

0 finding B[0,0]:a and B[2,0]:c. They are reused when the datapath works on subsequent rows of matrix A, e.g., B[0,0]:a when considering matrix A's row 1. Reuse depends on the sparsity pattern within matrices A and B. In the Gustavson product DSA [38] 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,:]. The walker pre-loads all the non-zeros in B[0,:]. There is minimal reuse.

Implication: Specialization across a DSA family can be achieved by varying only the walker. A programmable cache controller will enable design reuse and portability. Consequently, we make it feasible for the designer to compile down the state machines to controller microcode.

3.1.3 Design choice 3: Threads vs. Coroutines

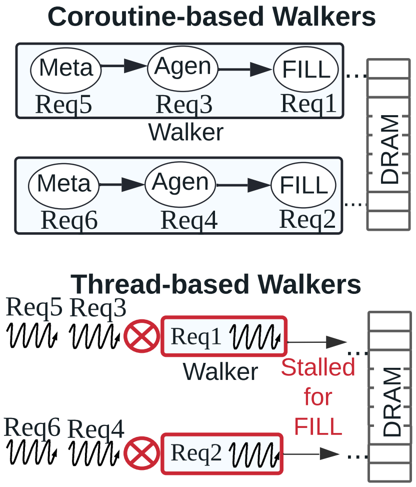


Figure 3.3: Walker Implementation.

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 [6, 7, 20]. 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) in the walker logic (see Figure 3.2 and Figure 3.3). They lock up the pipeline, reduce the number of active walkers, and minimize memory-level parallelism.

This paper expresses the walkers as a coroutine and multiplexes them on a pipeline. (see FSM shown in Figure 3.2). 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 (AGEN), and data-dependent branches (MATCH). When the long latency event

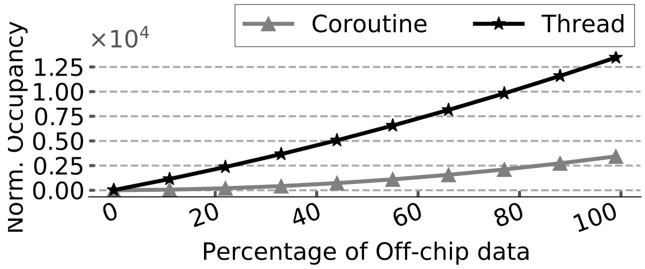
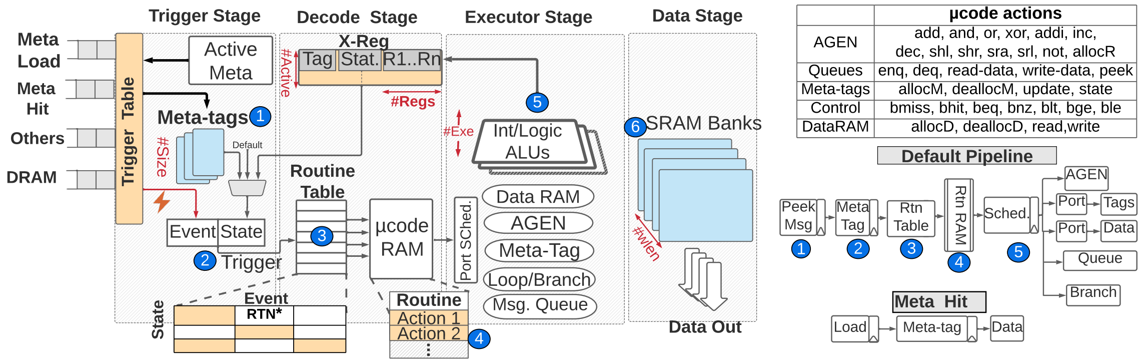


Figure 3.4: Occupancy. Y-axis: Occupancy of coroutines vs. threads, normalized to the occupancy of coroutine when all of the data are on-chip.



	µcode actions
AGEN	add, and, or, xor, addi, inc, dec, shl, shr, sra, srl, not, allocR
Queues	enq, deq, read-data, write-data, peek
Meta-tags	allocM, deallocM, update, state
Control	bmiss, bhit, beq, bnz, blt, bge, ble
DataRAM	allocD, deallocD, read,write

Figure 3.5: *X-Cache* Architecture

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 3.4 compares the occupancy of controller when walker implemented with coroutine vs. threads. Occupancy is measured as $\#active\text{-}reg \times size(bytes) \times lifetime(cycles)$. 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.

Implication: Walking and data orchestration need to be implemented as a coroutine to minimize controller occupancy and maximize parallel refills from the DRAM. Coroutines are a portable abstraction for representing different walkers, and we need a compiler to translate it into controller microcode.

3.2 *X-Cache*'s Microarchitecture Modules

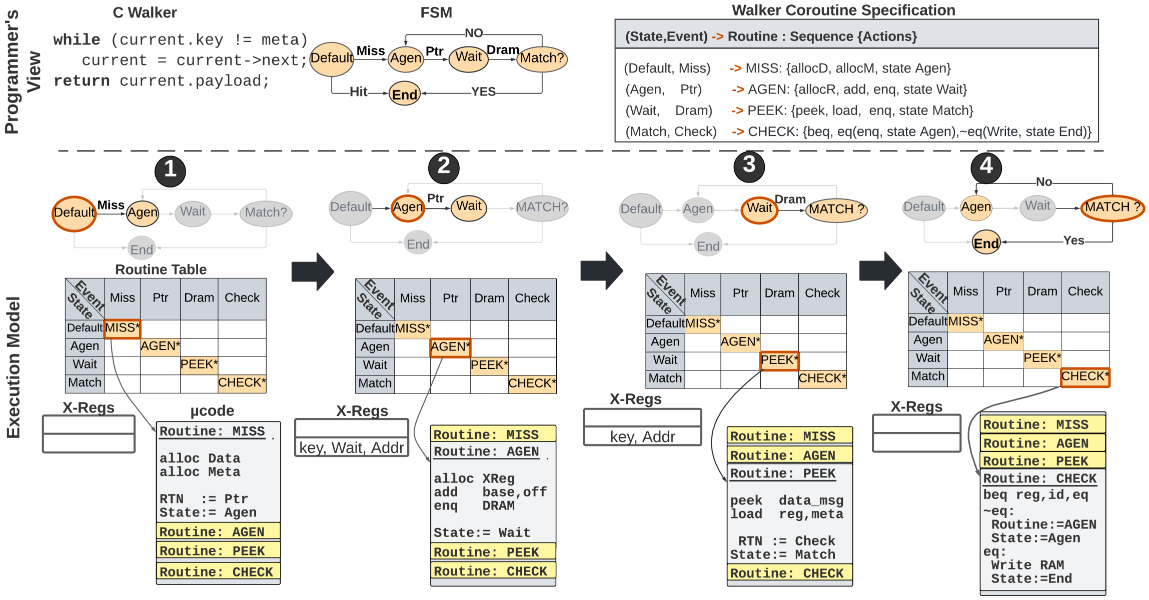
Figure 3.5 illustrates the modules in *X-Cache*. ❶ **Meta-Tag** It is created by our Chisel generator based on the parameters set by the DSA architect. The architect defines the metadata fields that the controller should write explicitly to the tag entry when a refill is completed. These fields will 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).

❷ **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.* ❸ **Routine Table** Similar to cache coherence protocols [35], 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. ❹ **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.

❺ **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 3.5 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).

❻ **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



```
while (current.key != meta)
   current = current->next;
return current.payload;
```

(State,Event) -> Routine : Sequence {Actions}	
(Default, Miss)	-> MISS: {allocD, allocM, state Agen}
(Agen, Ptr)	-> AGEN: {allocR, add, enq, state Wait}
(Wait, Dram)	-> PEEK: {peek, load, enq, state Match}
(Match, Check)	-> CHECK: {beq, eq(enq, state Agen),~eq(Write, state End)}

State \ Event	Miss	Ptr	Dram	Check
Default	MISS*			
Agen		AGEN*		
Wait			PEEK*	
Match				CHECK*

```
Routine: MISS

alloc Data
alloc Meta

RTN  := Ptr
State:= Agen
Routine: AGEN
Routine: PEEK
Routine: CHECK
```

```
Routine: MISS
Routine: AGEN

alloc XReg
add   base,off
enq   DRAM

State:= Wait
Routine: PEEK
Routine: CHECK
```

```
Routine: MISS
Routine: AGEN
Routine: PEEK

peek  data_msg
load  reg,meta

 RTN := Check
State:= Match
Routine: CHECK
```

```
Routine: MISS
Routine: AGEN
Routine: PEEK
Routine: CHECK
beq reg,id,eq
~eq:
 Routine:=AGEN
 State:=Agen
eq:
 Write RAM
 State:=End
```

Figure 3.6: *X-Cache* Execution Model.

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.

3.3 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 3.6) of the walker.

3.3.1 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 3.6:FSM). The main purpose of this step is to convert stalling 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 3.6: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.

3.3.2 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 3.5: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 wake-up, 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. ❶ 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 ❷ 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 intermediary 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 ❸, when `Current` block arrives from the DRAM, the walker peeks and extracts the block's key. In ❹, 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.

3.4 DSA Walker Examples

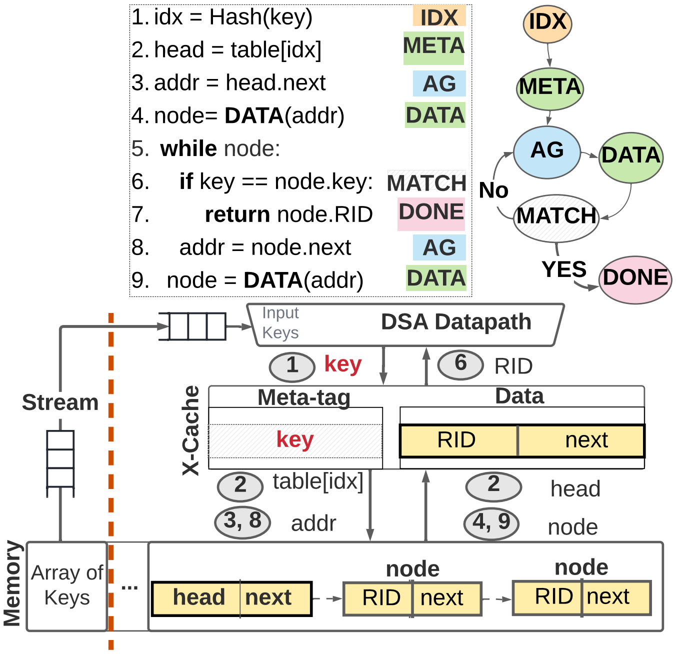
In this section, we provide an overview of how the walkers are specified. Table 3.1 summarizes the features of X-cache that are most relevant to each DSA. Each DSA we study uses an

Table 3.1: *X-Cache* features benefiting DSAs

DSA	Meta-tag	Preload	Coupling	Data	DS
Widx [20]	Key	No	Coupled	Rid	Hash Table
DASX-Hash [24]	Key	Yes	Decoupled	Rid	Hash Table
GraphPulse [32]	Node Idx	No	Decoupled	Event	Graph
SpArch [40]	Col Idx	Yes	Decoupled	B.Row	CSR
Gamma [39]	Col Idx	Yes	Decoupled	B.Row	CSR

indirectly accessed data structure that uses explicit metadata to identify the element. We make the key observation that the metadata provides a namespace on which loads/stores can be performed and clearly separates out non-overlapping elements. In this section we present the details for the walkers of two families: i) Widx [20]: a DSA for relation-database hash indices. and ii) SpArch [40]: a DSA for outer-product sparse GEMM. We walk through their data orchestration pipeline and discuss their access patterns. We then identify common attributes that motivate the need for *X-Cache*.

3.4.1 Widx: Meet the Walkers [20]



```
1. idx = Hash(key)                 IDX
2. head = table[idx]               META
3. addr = head.next                AG
4. node= DATA(addr)                DATA
5.  while node:
6.    if key == node.key:          MATCH
7.        return node.RID          DONE
8.    addr = node.next             AG
9.   node = DATA(addr)             DATA
```

Figure 3.7: Demonstration of Widx

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 3.7); 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.

3.4.2 SpArch: Outer-product sparse GEMM [40]

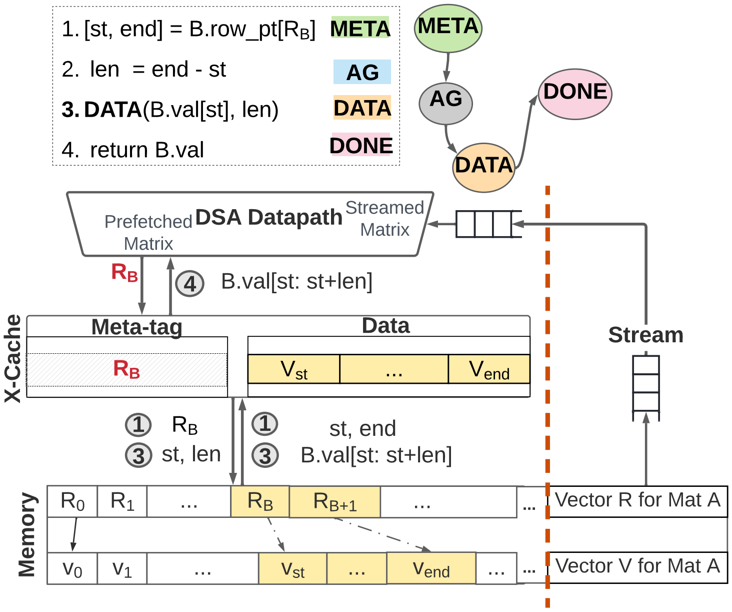


Figure 3.8: Demonstration of SpArch

SpArch 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.

3.5 *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).

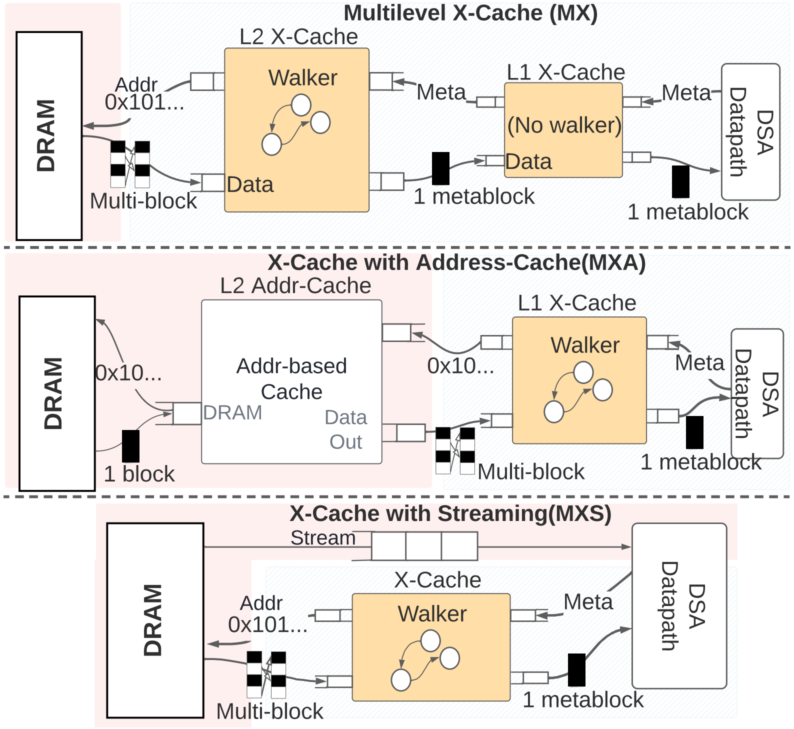


Figure 3.9: Fitting *X-Cache* in a memory hierarchy or next to a DSA which needs streaming too. Blue regions: Meta is used. Red regions: Global address is used.

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 3.8), 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-Cache*s 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.

Chapter 4

Evaluation

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 DRAMsim 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. W implement the controller and X-cache in Chisel and generate the Verilog. Verilator 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 an another layer that is attached to the TSIM driver.

4.1 *X-Cache* Setup

X-Cache is a toolflow for constructing domain-specific caches, which exploit domain-specific meta-tags, walkers and data orchestration. Three major parts in the toolflow of the *X-Cache* are the generator, programmable coroutine model, and the compiler. 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. Also, we use a table-based approach for programming the walker. The programmer uses our model for mapping different pairs of (event,state) to start the pointer of routines. Also, the compiler has been implemented in Scala. It compiles down the table-based definition to microcodes so it could be loaded to the programmable RAMS. The details of the configuring and programming *X-Cache* would be discussed in the Appendix A.

X-Cache targetting different DSA families (§4.2) and compare against the hardwired memory system of each DSA. Table 4.1 summarizes the pareto-optimal cache geometries we evaluate for each DSA. They have been obtained by sweeping different cache configurations and studying the hit rate and average memory access latency. Figure 4.1 depicts sweeping

DSA	#Active	#Exe	#Way	#Set	#Word
DASX(Hash)	16	4	8	1024	4
SpArch	32	4	8	512	4
Gamma	32	4	8	512	4
GraphPulse	16	4	1	131072	8

Table 4.1: *X-Cache* design parameters per DSA.

cache size for observing changes in the hit-rate and the read latency of the on-chip memory. Since, Widx and DASX, and SpArch and Gamma share the same data structure, we only ran the analysis for one of them and used the result for the other one as well. Also, since we're replacing the event queue in the GraphPulse design, we're using the exact configuration as they mentioned. Note that 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. 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.

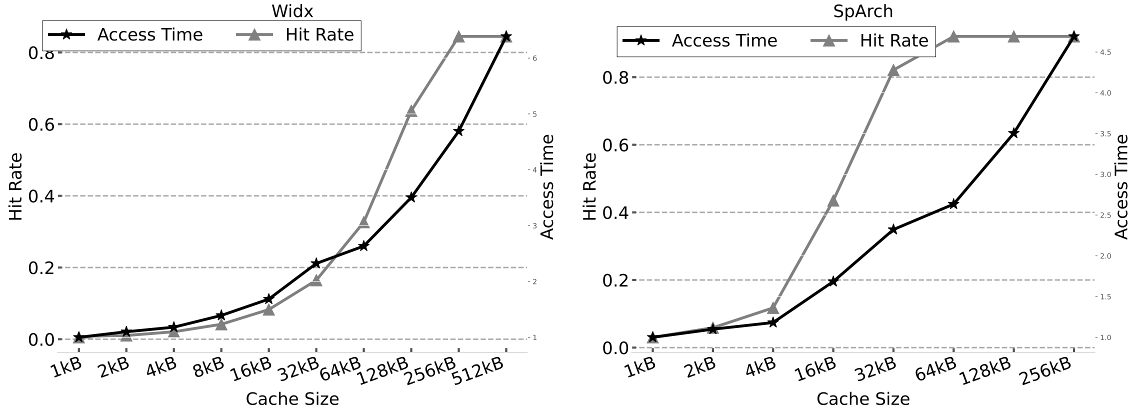


Figure 4.1: Sweeping Cache Size for different DSAs. Left Y-axis: Hit rate for different cache sizes. Right Y-axis: Read access-time of the on-chip SRAMs, normalized to the access-time of the 1 kB SRAM.

4.2 DSA Workload Setup

Widx [20]. Workload: MonetDB and TPC-H. Queries TPC-H 19/20/22. Dataset: 100GB *Widx* is a DSA for in-memory DBMSs hash index lookups. In hash index look-ups, the critical path is comprised of a key hashing followed by a series of pointer chasing in the key's corresponding bucket. To achieve high performance, Widx decouples key hashing from the linked-list traversal and processes multiple keys in parallel. We run DSS queries

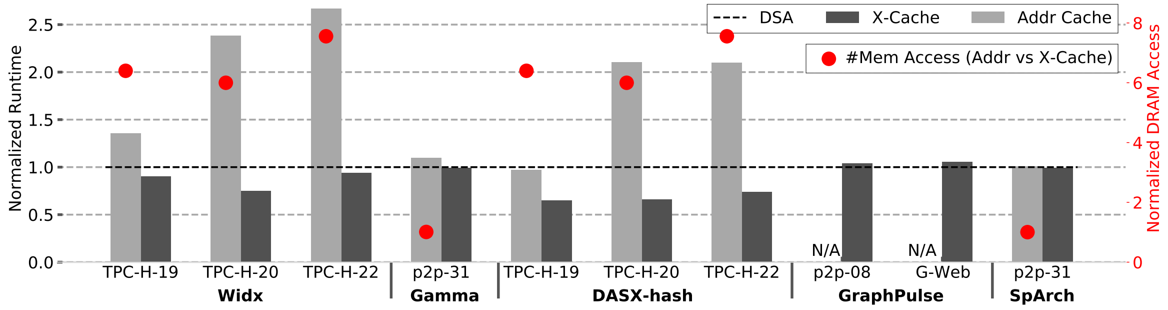


Figure 4.2: **Left Axis:** *X-Cache* Runtime vs Other DSAs (Lower is better). **Right Axis:** # of memory accesses of address cache (normalized to *X-Cache*). Higher means address cache makes more accesses. N/A: Graphpulse cannot be supported by address cache and we only compare against baseline DSA.

from TPC-H benchmarks on MonetDB and hijack the hash-joins. *X-Cache* replaces 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 [32]. Workload: PageRank. Inputs: p2p-Gnutella08: $N = 6.3K$, $NNZ = 21K$, **Web-Google:** $N = 916K$, $NNZ = 5.1M$ This is an event-driven accelerator for asynchronous graph algorithms. It uses a cache for maintaining the event queue. Each event contains a node id and a payload. The *event queue* keeps track of the unprocessed events. The event queue makes sure that for each destination node, there is only one event in the queue. If an incoming event violates this rule, the cache coalesces the conflicting event. A scheduler walks through the events in the cache and feeds them to the arithmetic units that perform a series of reductions. The result of these reductions updates the cache entry. The event Queue is substituted by the *X-Cache*. The incoming events are generated by the PEs from outside. 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 [24]. Workload: MonetDB and TPC-H DASX [24] is a data structure iterator. The DSA references the keys for each object while hardwiring the address generation. 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. We study *the hash table* which uses the collector for key searches. We evaluate hash using the dataset from MonetDB. The rest of the methodology is similar to the method we used for Widx.

Sparse GEMM: SpArch [40] and gamma [38]. 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 B.value array should be loaded and bringing them to the cache. Gamma uses the Gustavson algorithm to calculate the matrix-matrix multiplication, the pre-loading and walking Matrix B is very similar to the SpArch.

4.3 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 [38, 40], GraphPulse [32], DASX [24], and Widx [20]. 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 $\simeq$ 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 0.1mm^2 (a 256K cache requires $1.1mm^2$ just for the data RAM and tags).

4.4 Performance Evaluation

Result: *X-Cache delivers* $1.7X$ *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 4.2 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.

X-Cache's performance is competitive with baseline DSAs. In the case of Widx, it can even achieve a $1.54X$ speed up. *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.

In comparison 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 as a result of nested walks; $\simeq 6.5\times$ more than *X-Cache*(see Figure 4.2: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*.

4.5 Power Breakdown

Result: *Address-based caches require 26—79% more power than X-Cache.* **Result:** *The X-Cache controller expends 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:`

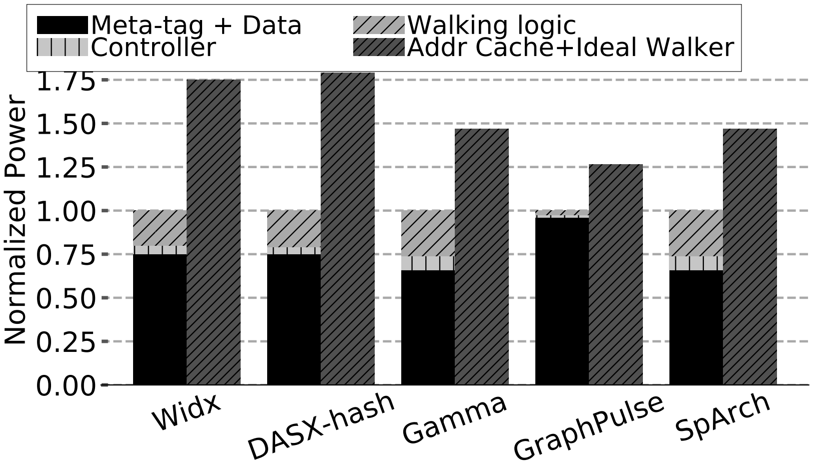


Figure 4.3: Total Power Breakdown in *X-Cache* (Lower is better)

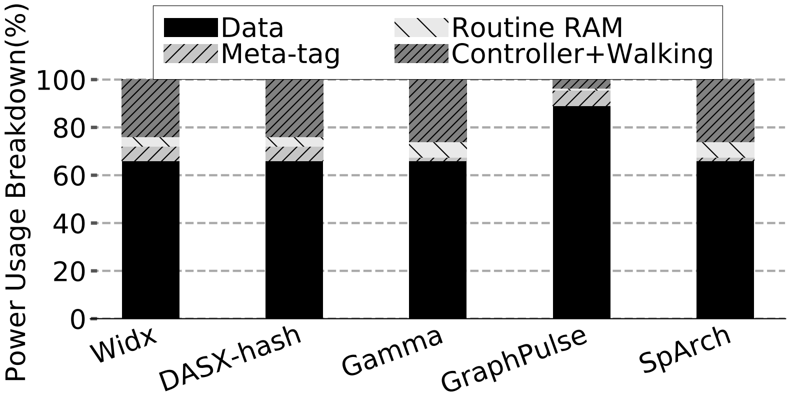


Figure 4.4: Breakdown of RAM power in *X-Cache*

`//github.com/bespoke-silicon-group/bsg_fakeram` [26]. 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 4.2. 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 it shows in Figure 4.3, 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 4.3 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 4.4 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 Rom, which is the primary difference between a programmable controller and a hardwired cache, is less than 4.2%.

Parameters	**Energy usage per bit[pj]**
Register	8.9e-03
Add	2.1e-01
Mul	12.6
Bitwise Op	1.8e-02
Shift	4.1e-01

Memory	**Energy usage per access to Byte [nj]**
Tag	6.4e-01
L1 Cache	7.7e-02

Table 4.2: Energy parameters (timing: 1GHz)

4.6 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 4.5, 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.

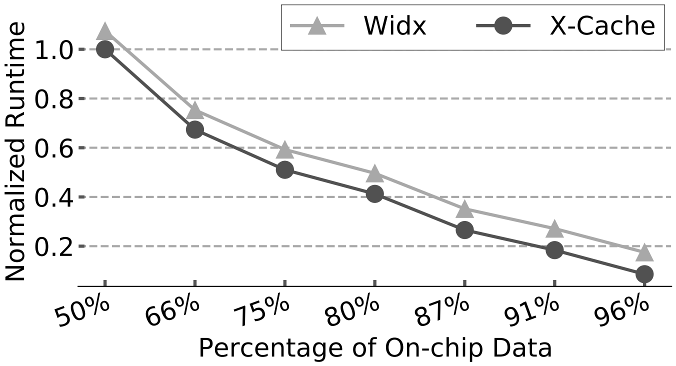


Figure 4.5: *X-Cache* Runtime Vs Widx (runtime normalized to data in DRAM)

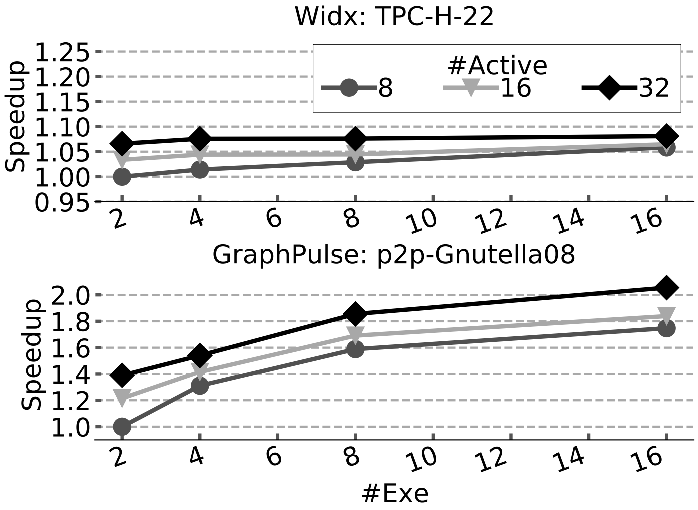
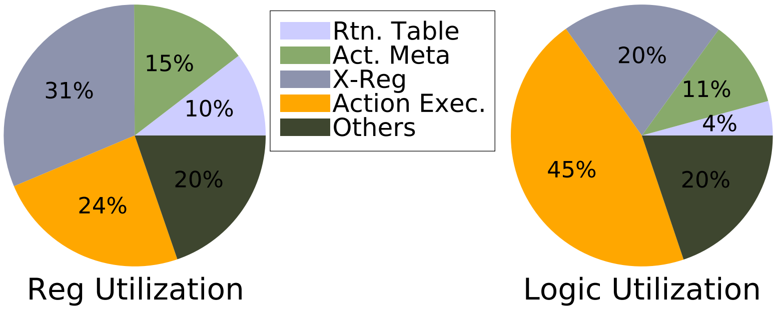


Figure 4.6: Sweeping Design Parameters. Y-Axis: Speedup of different configuration, normalized to the #Active=8 and #Exe=2



FPGA	Altera Cyclone IV GX
Model	EP4CGX150DF31C8
Total Logics	6985 (6% utlilized)
Total Combinational Functions	5766 (5% utilized)
Total Registers	3457 (2% utilized)

Figure 4.7: FPGA Synthesis

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 4.6 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 in comparison with 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 3.4, 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

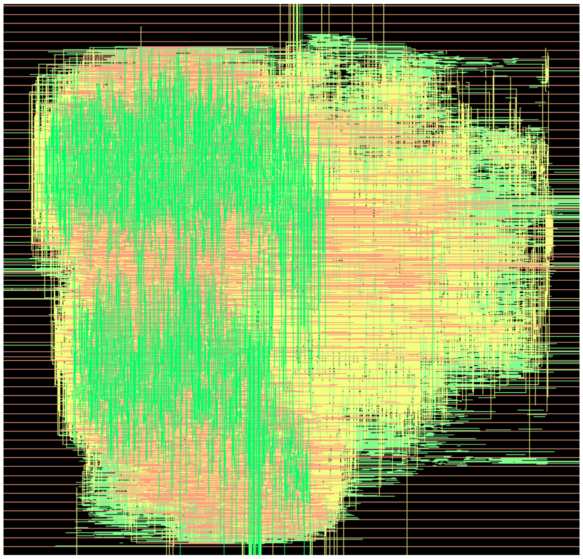


Figure 4.8: ASIC Layout. *X-Cache* controller (no RAMs).

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.

4.7 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. For this synthesis, we set the #Exe on 4 and #Active on 8. In Figure 4.7, we have breakdown the register utilization and logic utilization of the generated RTL. As it shows, 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 4.8 depicts the controller with #Exe=4 and #Active=8. Under 45nm technology, the total area required is 0.11mm^2 and 65K cells. A 256K RAM under 45nm technology require 0.8mm^2.

Chapter 5

Conclusion

5.1 Conclusions

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 machines. We are the first to create a high-performance cache controller targetting 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.

5.2 Future Works

Further Optimization

Although *X-Cache* focuses on supporting walking and orchestration for DSAs with dynamic access patterns and reducing the load-to-use latency, we believe that an area of research could focus on optimizing other aspects of our design. i) Replacement policy: Optimizing the replacement policy based on the targeted DSA could be explored for further optimizing *X-Cache*. While we provide a decoupled replacement-policy logic for facilitating further optimization, our research does not focus on this aspect. *X-Cache* uses LRU replacement policy by default, and our decoupled logic will allow programmers to further optimize this logic without interfering with the rest of the design. ii) Design parameters: An opportunity exists for further optimizing *X-Cache* by analysis on the targeted DSA and configuring it with the best set of parameters.

Mutation

X-Cache provides a modular-programmable framework for supporting a wide range of DSAs, exploiting data structure with complex access patterns. However, our modular architecture gives the DSA's developers the opportunity to fine-tune our architecture based on their needs. The fact that *X-Cache* leverage a modular architecture and it would be available as an open-source tool, facilitate further optimization, or even mutation, based on the DSA's requirements. Take binary-tree as an example. If a hypothetical DSA works with binary-tree as its primary data structure, a special entry in the X-Registers could be manually allocated by the developer for storing the *root*. This allows the *X-Cache* to reduce a significant amount of DRAM accesses. Note that *X-Cache*'s architecture allows the developer to add this feature with minimum overhead.

Bibliography

[1] Michael Adler, Kermin E. Fleming, Angshuman Parashar, Michael Pellauer, and Joel Emer. Leap scratchpads: Automatic memory and cache management for reconfigurable logic. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '11, pages 25–28, New York, NY, USA, February 2011. Association for Computing Machinery.

[2] T Ajayi, D Blaauw, TB Chan, CK Cheng, VA Chhabria, DK Choo, M Coltella, S Dobre, R Dreslinski, M Fogaça, et al. Openroad: Toward a self-driving, open-source digital layout implementation tool chain. *Proc. GOMACTECH*, pages 1105–1110, 2019.

[3] Bahar Asgari, Ramyad Hadidi, and Hyesoon Kim. Ascella: Accelerating sparse computation by enabling stream accesses to memory. In *Proc. of Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2020.

[4] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, aug 2011.

[5] J. Carter, W. Hsieh, L. Stoller, M. Swanson, Lixin Zhang, E. Brunvand, A. Davis, Chen-Chi Kuo, R. Kuramkote, M. Parker, L. Schaelicke, and T. Tateyama. Impulse: building a smarter memory controller. In *Proceedings Fifth International Symposium on High-Performance Computer Architecture*, pages 70–79, 1999.

[6] Tao Chen and G Edward Suh. Efficient data supply for hardware accelerators with prefetching and access/execute decoupling. In *Proc. of the 49th MICRO*, pages 1–12, 2016.

[7] Eric S Chung, James C Hoe, and Ken Mai. CoRAM: an in-fabric memory architecture for FPGA-based computing. In *PROC of the 19th FPGA*. ACM Request Permissions, February 2011.

[8] Jason Clemons, Chih-Chi Cheng, Iuri Frosio, Daniel R Johnson, and Stephen W Keckler. A patch memory system for image processing and computer vision. In *Proc. of the 49th MICRO*, pages 1–13, 2016.

[9] William J. Dally, Yatish Turakhia, and Song Han. Domain-specific hardware accelerators. *Communications of the ACM*, 63(7):48–57, June 2020.

[10] David Durst, Matthew Feldman, Dillon Huff, David Akeley, Ross G Daly, Gilbert Louis Bernstein, Marco Patrignani, Kayvon Fatahalian, and Pat Hanrahan. Type-directed scheduling of streaming accelerators. In *Proc. of the 41st PLDI*, pages 408–422, 2020.

[11] Daichi Fujiki, Niladrish Chatterjee, Donghyuk Lee, and Mike O'Connor. Near-memory data transformation for efficient sparse matrix multi-vector multiplication. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '19, New York, NY, USA, 2019. Association for Computing Machinery.

[12] Graham Gobieski, Amolak Nagi, Nathan Serafin, Mehmet Meric Isgenc, Nathan Beckmann, and Brandon Lucia. Manic: A vector-dataflow architecture for ultra-low-power embedded systems. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, page 670–684, New York, NY, USA, 2019. Association for Computing Machinery.

[13] Tae Jun Ham, Juan L Aragn, and Margaret Martonosi. DeSC: decoupled supply-compute communication management for heterogeneous architectures. In *Proc. of the 48th MICRO*, pages 191–203, 2015.

[14] Bingsheng He, Naga K. Govindaraju, Qiong Luo, and Burton Smith. Efficient gather and scatter operations on graphics processors. In *SC '07: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, pages 1–12, 2007.

[15] John L. Hennessy and David A. Patterson. *Computer Architecture, Sixth Edition: A Quantitative Approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 6th edition, 2017.

[16] John L. Hennessy and David A. Patterson. A new golden age for computer architecture. *Communications of the ACM*, 62(2):48–60, January 2019.

[17] Chen-Han Ho, Sung Jin Kim, and Karthikeyan Sankaralingam. Efficient execution of memory access phases using dataflow specialization. In *Proc. of the 42nd ISCA*, pages 118–130, 2015.

[18] Vishwesh Jatala, Jayvant Anantpur, and Amey Karkare. Scratchpad Sharing in GPUs. *ACM Transactions on Architecture and Code Optimization*, 14(2):15:1–15:29, May 2017.

[19] Ujval J. Kapasi, Scott Rixner, William J. Dally, Brucek Khailany, Jung Ho Ahn, Peter Mattson, and John D. Owens. Programmable stream processors. *Computer*, 36(8):54–62, aug 2003.

[20] Onur Kocberber, Boris Grot, Javier Picorel, Babak Falsafi, Kevin T Lim, and Parthasarathy Ranganathan. Meet the walkers: accelerating index traversals for in-memory databases. In *Proc. of the 46th MICRO*, pages 468–479, 2013.

[21] David Koeplinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Ruben Fiszel, Tian Zhao, Luigi Nardi, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. *Spatial: a language and compiler for application accelerators.* ACM, New York, New York, USA, June 2018.

[22] David Koeplinger, Raghu Prabhakar, Yaqi Zhang, Christina Delimitrou, Christos Kozyrakis, and Kunle Olukotun. Automatic Generation of Efficient Accelerators for Reconfigurable Hardware. In *Proc. of the 43rd ISCA*, 2016.

[23] Rakesh Komuravelli, Matthew D Sinclair, Johnathan Alsop, Muhammad Huzaifa, Maria Kotsifakou, Prakalp Srivastava, Sarita V Adve, and Vikram S Adve. Stash: have your scratchpad and cache it too. In *Proc. of the 42nd ISCA*, pages 707–719, 2015.

[24] Snehasish Kumar, Naveen Vedula, Arrvindh Shriraman, and Vijayalakshmi Srinivasan. DASX: Hardware accelerator for software data structures. *Proceedings of the International Conference on Supercomputing*, 2015-June:361–371, 2015.

[25] L. Lu, J. Xie, R. Huang, J. Zhang, W. Lin, and Y. Liang. An efficient hardware accelerator for sparse convolutional neural networks on fpgas. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 17–25, 2019.

[26] Naveen Muralimanohar, Rajeev Balasubramonian, and Norm Jouppi. Optimizing NUCA organizations and wiring alternatives for large caches with CACTI 6.0. In *PROC of the 40th MICRO*, 2007.

[27] Tony Nowatzki, Vinay Gangadhar, Newsha Ardalani, and Karthikeyan Sankaralingam. Stream-Dataflow Acceleration. In *Proc. of the 44th ISCA*, pages 416–429, 2017.

[28] Eriko Nurvitadhi, Asit Mishra, and Debbie Marr. A sparse matrix vector multiply accelerator for support vector machine. In *Proceedings of the 2015 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, CASES '15, page 109–116. IEEE Press, 2015.

[29] Subhankar Pal, Jonathan Beaumont, Dong-Hyeon Park, Aporva Amarnath, Siying Feng, Chaitali Chakrabarti, Hun-Seok Kim, David Blaauw, Trevor Mudge, and Ronald Dreslinski. Outerspace: An outer product based sparse matrix multiplication accelerator. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 724–736. IEEE, 2018.

[30] Michael Pellauer, Yakun Sophia Shao, Jason Clemons, Neal Clayton Crago, Kartik Hegde, Rangharajan Venkatesan, Stephen W Keckler, Christopher W Fletcher, and Joel S Emer. Buffets: An efficient and composable storage idiom for explicit decoupled data orchestration. In *Proc. of the 24th ASPLOS*, pages 137–151, 2019.

[31] Eric Qin, Ananda Samajdar, Hyoukjun Kwon, Vineet Nadella, Sudarshan Srinivasan, Dipankar Das, Bharat Kaul, and Tushar Krishna. Sigma: A sparse and irregular gemm accelerator with flexible interconnects for dnn training. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020.

[32] Shafiur Rahman, Nael B Abu-Ghazaleh, and Rajiv Gupta. GraphPulse: An event-driven hardware accelerator for asynchronous graph processing. In *Proc. of the 53rd MICRO*, pages 908–921, 2020.

[33] Ali Sedaghati, Milad Hakimi, Reza Hojabr, and Arrvindh Shriraman. X-cache: A modular architecture for domain-specific caches. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, page 396–409, New York, NY, USA, 2022. Association for Computing Machinery.

[34] Vivek Seshadri, Thomas Mullins, Amirali Boroumand, Onur Mutlu, Phillip B. Gibbons, Michael A. Kozuch, and Todd C. Mowry. Gather-scatter dram: In-dram address translation to improve the spatial locality of non-unit strided accesses. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, page 267–280, New York, NY, USA, 2015. Association for Computing Machinery.

[35] Daniel J. Sorin, Mark D. Hill, and David A. Wood. A Primer on Memory Consistency and Cache Coherence. *Synthesis Lectures on Computer Architecture*, 6(3):1–212, November 2011. Publisher: Morgan & Claypool Publishers.

[36] Po-An Tsai, Nathan Beckmann, and Daniel Sanchez. Jenga: Software-defined cache hierarchies. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 652–665, June 2017.

[37] Po An Tsai, Yee Ling Gan, and Daniel Sanchez. Rethinking the memory hierarchy for modern languages. *Proceedings of the Annual International Symposium on Microarchitecture, MICRO*, 2018-Octob:203–216, 2018.

[38] Guowei Zhang, Nithya Attaluri, Joel S. Emer, and Daniel Sánchez. Gamma: leveraging gustavson's algorithm to accelerate sparse matrix multiplication. In *Proc. of the 26th ASPLOS*, 2021.

[39] Guowei Zhang, Nithya Attaluri, Joel S. Emer, and Daniel Sanchez. Gamma: Leveraging gustavson's algorithm to accelerate sparse matrix multiplication. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2021, page 687–701, New York, NY, USA, 2021. Association for Computing Machinery.

[40] Zhekai Zhang, Hanrui Wang, Song Han, and William J Dally. Sparch: Efficient architecture for sparse matrix multiplication. In *26th Int'l. Symposium on High Performance Computer Architecture*, 2020.

Appendix A

Toolflow

A.1 Configuration

Figure A.1 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 3.5. 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 [32], 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.

```
class XCacheIO () extends Bundle with MsgParams{
    val meta = MetaIO()
    val fill = DRAMIO()
    val msg =  MsgIO()
    val address = AddressIO()
}
class XCache (triggers,transitions,routines) {

  /*** IO Ports ***/
  val io = IO( new XCacheIO())
   ...
  /*** Data ***/
  val dataRAM = Memory([dbanks,dsets,dways],wlen)
  /*** Tags ***/
  val metaRAM = MetaTag([tbanks,tsets,tways],tag,width)
  /*** Programmable RAM ***/
  val rTable = RAM(transitions.compile())
  val microcode   = RAM(routines.compile())
  /*** Modules ***/
  val exe = Executor(nExe)
  val xReg = XRegister(nActive,nWidth,tag)
   ....
}
```

Figure A.1: *X-Cache* parameterized design

```
object RoutineTable{

val transitions = Array[Transition](
    //Transition ( Routine(DST_Routine), Trigger(Event, State))

    Transition ( Routine(MISS), Trigger(Miss, Default)),
    Transition ( Routine(AGEN), Trigger(Ptr , Agen)),
    ...
    )
}
```

Figure A.2: Programming the Routine Table

A.2 Programming

Apart from the configurability, *X-Cache* allows the developer to program several parts of the generated design, in order to provide a different walking and orchestration logic. There are two main components in the programmability of *X-Cache*: i) Transition: For creating a table-based transition in *X-Cache*, the programmer should define states, events, and valid pairs of them which lead to routine triggering. As Figure 3.5 showed, the *Routine Table* is in charge of holding these definitions. In Figure A.2 these definitions are depicted. ii) Routine RAM: In Figure A.3, the definition of the routines and actions within them has been shown.

```
object microcodeRoutine{

val routines = Array[Routine](

    // Routine (Name , Seq(Actions))

    Routine ("MISS", Seq( allocD, allocM, enq "Ptr", state "Agen"))
    Routine ("AGEN", Seq( allocR, add, enq DRAM, state "Wait"))
        ...
    )
}
```

Figure A.3: Programming the Microcode RAM

For each entry, a string, containing a name the same as the one used in transition-definition, would be passed, along with the actions forming it. The allowed actions are the same as the Table in Figure 3.5.

Compilation

Then, the *objects* defined in Figure A.2 and Figure A.3 would be compiled down to microcodes and filled the routine table and μ-coded RAM respectively. Basically, the compiler maps the events and states to binary values and concatenates them in order to form an address for the routine table. Also, the routines and their actions would be translated to an array first, and the index of the beginning of each routine fills the entries inside the routine table.