Accelerating Pointer Chasing in 3D-Stacked Memory: Challenges, Mechanisms, Evaluation

Kevin Hsieh† Samira Khan‡ Nandita Vijaykumar†
Kevin K. Chang† Amirali Boroumand† Saugata Ghose† Onur Mutlu§†
†*Carnegie Mellon University* ‡*University of Virginia* §*ETH Zürich*

***Abstract*—Pointer chasing is a fundamental operation, used by many important data-intensive applications (e.g., databases, key-value stores, graph processing workloads) to traverse linked data structures. This operation is both memory bound and latency sensitive, as it (1) exhibits irregular access patterns that cause frequent cache and TLB misses, and (2) requires the data from *every memory access* to be sent back to the CPU to determine the next pointer to access. Our goal is to accelerate pointer chasing by performing it *inside main memory*, thereby avoiding inefficient and high-latency data transfers between main memory and the CPU. To this end, we propose the *In-Memory PoInter Chasing Accelerator* (IMPICA), which leverages the logic layer within 3D-stacked memory for linked data structure traversal.**

This paper identifies the key design challenges of designing a pointer chasing accelerator in memory, describes new mechanisms employed within IMPICA to solve these challenges, and evaluates the performance and energy benefits of our accelerator. IMPICA addresses the key challenges of (1) how to achieve high parallelism in the presence of serial accesses in pointer chasing, and (2) how to effectively perform virtual-to-physical address translation on the memory side without requiring expensive accesses to the CPU's memory management unit. We show that the solutions to these challenges, *address-access decoupling* and a *region-based page table*, respectively, are simple and low-cost. We believe these solutions are also applicable to many other in-memory accelerators, which are likely to also face the two challenges.

Our evaluations on a quad-core system show that IMPICA improves the performance of pointer chasing operations in three commonly-used linked data structures (linked lists, hash tables, and B-trees) by 92%, 29%, and 18%, respectively. This leads to a significant performance improvement in applications that utilize linked data structures — on a real database application, DBx1000, IMPICA improves transaction throughput and response time by 16% and 13%, respectively. IMPICA also significantly reduces overall system energy consumption (by 41%, 23%, and 10% for the three commonly-used data structures, and by 6% for DBx1000).

1. Introduction

Linked data structures, such as trees, hash tables, and linked lists are commonly used in many important applications [21, 25, 28, 33, 60, 61, 89]. For example, many databases use B/B$^+$-trees to efficiently index large data sets [21, 28], key-value stores use linked lists to handle collisions in hash tables [25, 60], and graph processing workloads [1, 2, 81] use pointers to represent graph edges. These structures link nodes using pointers, where each node points to at least one other node by storing its address. Traversing the link requires serially accessing consecutive nodes by retrieving the address(es) of the next node(s) from the pointer(s) stored in the current node. This fundamental operation is called *pointer chasing* in linked data structures.

Pointer chasing is currently performed by the CPU cores, as part of an application thread. While this approach eases the integration of pointer chasing into larger programs, pointer chasing can be inefficient within the CPU, as it introduces several sources of performance degradation: (1) dependencies exist between memory requests to the linked nodes, resulting in serialized memory accesses and limiting the available instruction-level and memory-level parallelism [33, 61, 62, 67, 75]; (2) irregular allocation or rearrangement of the connected nodes leads to access pattern irregularity [18, 43, 45, 61, 93], causing frequent cache and TLB misses; and (3) link traversals in data structures that diverge at each node (e.g., hash tables, B-trees) frequently go down different paths during different iterations, resulting in little reuse, further limiting cache effectiveness [59]. Due to these inefficiencies, a significant *memory bottleneck* arises when executing pointer chasing operations in the CPU, which stalls on a large number of memory requests that suffer from the long round-trip latency between the CPU and the memory.

Many prior works (e.g., [14–16, 18, 36, 37, 43, 45, 55, 58, 59, 61, 75, 76, 83, 92, 93, 95, 99]) proposed mechanisms to predict and prefetch the next node(s) of a linked data structure early enough to hide the memory latency. Unfortunately, prefetchers for linked data structures suffer from several shortcomings: (1) they usually do not provide significant benefit for data structures that diverge at each node [45], due to low prefetcher accuracy and low miss coverage; (2) aggressive prefetchers can consume too much of the limited off-chip memory bandwidth and, as a result, slow down the system [18, 43, 84]; and (3) a prefetcher that works well for some pointer-based data structure(s) and access patterns (e.g., a Markov prefetcher designed for mostly-static linked lists [43]) usually does not work efficiently for different data structures and/or access patterns. Thus, it is important to explore new solution directions to alleviate performance and efficiency loss due to pointer chasing.

Our goal in this work is to accelerate pointer chasing by *directly minimizing the memory bottleneck* caused by pointer chasing operations. To this end, we propose to perform pointer chasing *inside main memory* by leveraging processing-in-memory (PIM) mechanisms, *avoiding the need to move data to the CPU*. In-memory pointer chasing greatly reduces (1) the latency of the operation, as an address does not need to be brought all the way into the CPU before it can be dereferenced; and (2) the reliance on caching and prefetching in the CPU, which are largely ineffective for pointer chasing.

Early work on PIM proposed to embed general-purpose logic in main memory [20, 30, 44, 48, 69, 70, 80, 85], but was not commercialized due to the difficulty of fabricating logic and memory on the same die. The emergence of 3D die-stacked memory, where memory layers are stacked on top of a logic layer [38, 39, 41, 42, 50], provides a unique opportunity to embed simple accelerators or cores within the logic layer. Several recent works recognized and explored this opportunity (e.g., [1–3, 6, 7, 10, 12, 22, 27, 31, 34, 46, 51, 56, 71, 72, 96, 97]) for various purposes. For the first time, in this work, we *propose an in-memory accelerator for chasing pointers* in any linked data structure, called the *In-Memory PoInter Chasing Accelerator* (IMPICA). IMPICA leverages the low memory access latency at the logic layer of 3D-stacked memory to speed up pointer chasing operations.

We identify *two fundamental challenges that we believe exist for a wide range of in-memory accelerators*, and evaluate them as part of a case study in designing a pointer chasing accelerator in memory. These fundamental challenges are (1) how to achieve high parallelism in the accelerator (in the presence of serial accesses in pointer chasing), and (2) how to effectively perform virtual-to-physical address translation on the memory side without performing costly accesses to the CPU's memory management unit. We call these, respectively, the *parallelism challenge* and the *address translation challenge*.

The Parallelism Challenge. Parallelism is challenging to exploit in an in-memory accelerator even with the reduced latency and higher bandwidth available within 3D-stacked memory, as the performance of pointer chasing is limited by *dependent sequential accesses*. The serialization problem can be exacerbated when the accelerator traverses multiple streams of links: while traditional out-of-order or multicore CPUs can service memory requests from multiple streams in parallel due to their ability to exploit high levels of instruction- and memory-level parallelism [29, 33, 62, 64–67, 86], simple accelerators (e.g., [1, 22, 72, 97]) are unable to exploit such parallelism unless they are carefully designed.

We observe that accelerator-based pointer chasing is primarily bottlenecked by memory access latency, and that the address generation computation for link traversal takes only a small fraction of the total traversal time, leaving the accelerator idle for a majority of the traversal time. In IMPICA, we exploit this idle time by *decoupling* link address generation from the issuing and servicing of a memory request, which allows the accelerator to generate addresses for one link traversal stream while waiting on the request associated with a different link traversal stream to return from memory. We call this design *address-access decoupling*. Note that this form of decoupling bears resemblance to the decoupled access/execute architecture [82], and we in fact take inspiration from past works [17, 49, 82], except our design is specialized for building a pointer chasing accelerator in 3D-stacked memory, and this paper solves specific challenges within the context of pointer chasing acceleration.

The Address Translation Challenge. An in-memory pointer chasing accelerator must be able to perform address translation, as each pointer in a linked data structure node stores the *virtual* address of the next node, even though main memory is *physically* addressed. To determine the next address in the pointer chasing sequence, the accelerator must resolve the virtual-to-physical address mapping. If the accelerator relies on existing CPU-side address translation mechanisms, any performance gains from performing pointer chasing in memory could easily be nullified, as the accelerator needs to send a long-latency translation request to the CPU via the off-chip channel for each memory access. The translation can sometimes require a page table walk, where the CPU must issue multiple memory requests to read the page table, which further increases traffic on the memory channel. While a naive solution is to simply duplicate the TLB and page walker within memory, this is prohibitively difficult for three reasons: (1) coherence would have to be maintained between the CPU and memory-side TLBs, introducing extra complexity and off-chip requests; (2) the duplication is very costly in terms of hardware; and (3) a memory module can be used in conjunction with many different processor architectures, which use different page table implementations and formats, and ensuring compatibility between the in-memory TLB/page walker and all of these designs is difficult.

We observe that traditional address translation techniques do not need to be employed for pointer chasing, as link traversals are (1) limited to linked data structures, and (2) touch only certain data structures in memory. We exploit this in IMPICA by allocating data structures accessed by IMPICA into contiguous *regions* within the virtual memory space, and designing a new translation mechanism, the *region-based page table*, which is optimized for in-memory acceleration. Our approach provides translation within memory at low latency and low cost, while minimizing the cost of maintaining TLB coherence.

Evaluation. By solving both key challenges, IMPICA provides significant performance and energy benefits for pointer chasing operations and applications that use such operations. First we examine three microbenchmarks, each of which performs pointer chasing on a widely used data structure (linked list, hash table, B-tree), and find that IMPICA improves their performance by 92%, 29%, and 18%, respectively, on a quad-core system over a state-of-the-art baseline. Second, we evaluate IMPICA on a real database workload, DBx1000 [94], on a quad-core system, and show that IMPICA increases *overall* database transaction throughput by 16% and reduces transaction latency by 13%. Third, IMPICA reduces *overall* system energy, by by 41%, 23%, and 10% for the three microbenchmarks and by 6% for DBx1000. These benefits come at a very small hardware cost: our evaluations show that IMPICA comprises only 7.6% of the area of a small embedded core (the ARM Cortex-A57 [4]).

We make the following major contributions in this paper:

- This is the first work to propose an in-memory accelerator for pointer chasing. Our proposal, IMPICA, accelerates linked data structure traversal by chasing pointers inside the logic layer of 3D-stacked memory, thereby eliminating inefficient, high-latency serialized data transfers between the CPU and main memory.
- We identify two fundamental challenges in designing an efficient in-memory pointer chasing accelerator (Section 3). These challenges can greatly hamper performance if the accelerator is not designed *carefully* to overcome them. First, multiple streams of link traversal can unnecessarily get serialized at the accelerator, degrading performance (the *parallelism challenge*). Second, an in-memory accelerator needs to perform virtual-to-physical address translation for each pointer, but this critical functionality does not exist on the memory side (the *address translation challenge*).
- IMPICA solves the *parallelism challenge* by decoupling link address generation from memory accesses, and utilizes the idle time during memory accesses to service *multiple* pointer chasing streams simultaneously. We call this approach *address-access decoupling* (Section 4.1).
- IMPICA solves the *address translation challenge* by allocating data structures it accesses into contiguous virtual memory regions, and using an optimized and low-cost *region-based page table* structure for address translation (Section 4.2).
- We evaluate IMPICA extensively using both microbenchmarks and a real database workload. Our results (Section 7) show that IMPICA improves both system performance and energy efficiency for all of these workloads, while requiring only very modest hardware overhead in the logic layer of 3D-stacked DRAM.

2. Motivation

To motivate the need for a pointer chasing accelerator, we first examine the usage of pointer chasing in contemporary workloads. We then discuss opportunities for acceleration within 3D-stacked memory.

2.1. Pointer Chasing in Modern Workloads

Pointers are ubiquitous in fundamental data structures such as linked lists, trees, and hash tables, where the nodes of the data structure are linked together by storing the addresses (i.e., pointers) of neighboring nodes. Pointers make it easy to dynamically add/delete nodes in these data structures, but link traversal is often serialized, as the address of the next node can be known only after the current node is fetched. The serialized link traversal is commonly referred to as *pointer chasing*.

Due to the flexibility of insertion/deletion, pointer-based data structures and link traversal algorithms are essential building blocks in programming, and they enable a very wide range of workloads. For instance, at least six different types of modern data-intensive applications rely *heavily* on linked data structures: (1) **databases and file systems** use B/B$^+$-trees for indexing tables or metadata [21, 28]; (2) **in-memory caching** applications based on key-value stores, such as Memcached [25] and Masstree [60], use linked lists to resolve hash table collisions and trie-like B$^+$-trees as their main data structures; (3) **graph processing workloads** use pointers to represent the edges that connect the vertex data structures together [1, 81]; (4) **garbage collectors** in high level languages typically maintain reference relations using trees [89]; (5) **3D video games** use binary space partitioning trees to determine the objects that need to be rendered [68]; and (6) **dynamic routing tables** in networks employ balanced search trees for high-performance IP address lookups [88].

While linked data structures are widely used in many modern applications, chasing pointers is very inefficient in general-purpose processors. There are three major reasons behind the inefficiency. First, the inherent serialization that occurs when accessing consecutive nodes limits the available instruction-level and memory-level parallelism [43, 55, 59, 61–67, 75, 76]. As a result, out-of-order execution provides only limited performance benefit when chasing pointers [61, 62, 64]. Second, as nodes can be inserted and removed dynamically, they can get allocated to different regions of memory. The irregular allocation causes pointer chasing to exhibit irregular access patterns, which lead to frequent cache and TLB misses [18, 43, 45, 61, 93]. Third, for data structures that diverge at each node, such as B-trees, link traversals often go down different paths during different iterations, as the inputs to the traversal function change. As a result, lower-level nodes that were recently referenced during a link traversal are unlikely to be reused in subsequent traversals, limiting the effectiveness of many caching policies [47, 55, 59], such as LRU replacement.

To quantify the performance impact of chasing pointers in real-world workloads, we profile two popular applications that heavily depend on linked data structures, using a state-of-art Intel Xeon system:[1] (1) *Memcached* [25], using a real Twitter dataset [23] as its input; and (2) *DBx1000* [94], an in-memory database system, using the TPC-C benchmark [87] as its input. We profile the pointer chasing code within the application separately from other parts of the application code. Figure 1 shows how pointer chasing compares to the rest of the application in terms of execution time, cycles per instruction (CPI), and the ratio of last-level cache (LLC) miss cycles to the total cycles.

We make three major observations. First, both Memcached and DBx1000 spend a significant fraction of their total execution time (7%

[1]We use the Intel® VTune^{TM} profiling tool on a machine with a Xeon® W3550 processor (3GHz, 8-core, 8 MB LLC) [40] and 18 GB memory. We profile each application for 10 minutes after it reaches steady state.

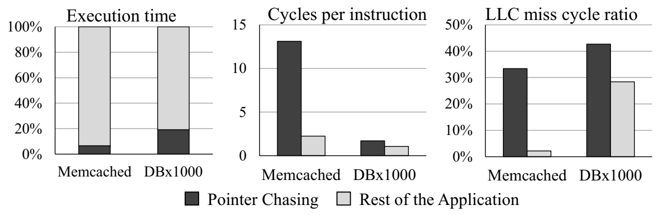


Fig. 1. Profiling results of pointer chasing portions of code vs. the rest of the application code in Memcached and DBx1000.

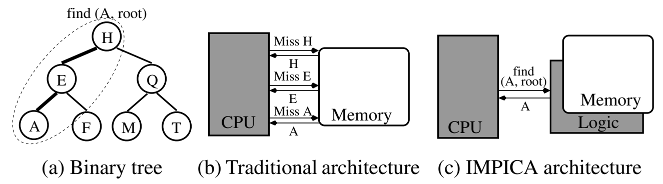


Fig. 2. Pointer chasing (a) in a traditional architecture (b) and in IMPICA with 3D-stacked memory (c).

and 19%, respectively) on pointer chasing, as a result of dependent cache misses [33,61,75]. Though these percentages might sound small, real software often does not have a single type of operation that consumes this significant a fraction of the total time. Second, we find that pointer chasing is significantly more inefficient than the rest of the application, as it requires much higher cycles per instruction (6× in Memcached, and 1.6× in DBx1000). Third, pointer chasing is largely memory-bound, as it exhibits much higher cache miss rates than the rest of the application and as a result spends a much larger fraction of cycles waiting for LLC misses (16× in Memcached, and 1.5× in DBx1000). From these observations, we conclude that (1) pointer chasing consumes a significant fraction of execution time in two important and complicated applications, (2) pointer chasing operations are bound by memory, and (3) executing pointer chasing code in a modern general-purpose processor is very inefficient and thus can lead to a large performance overhead. Other works made similar observations for different workloads [33, 61, 75].

Prior works (e.g., [14–16,18,36,37,43,45,55,58,59,61,75,76,83,92, 93, 95, 99]) proposed specialized prefetchers that predict and prefetch the next node of a linked data structure to hide memory latency. While prefetching can mitigate part of the memory latency problem, it has three major shortcomings. First, the efficiency of prefetchers degrades significantly when the traversal of linked data structures diverges into multiple paths and the access order is irregular [45]. Second, prefetchers can sometimes slow down the entire system due to contention caused by inaccurate prefetch requests [18, 19, 43, 84]. Third, these hardware prefetchers are usually designed for specific data structure implementations, and tend to be very inefficient when dealing with other data structures. It is difficult to design a prefetcher that is efficient and effective for *all* types of linked data structures. **Our goal** in this work is to improve the performance of pointer chasing applications *without* relying on prefetchers, regardless of the types of linked data structures used in an application.

2.2. Accelerating Pointer Chasing in 3D-Stacked Memory

We propose to improve the performance of pointer chasing by leveraging processing-in-memory (PIM) to alleviate the memory bottleneck. Instead of sequentially fetching *each node* from memory and sending it to the CPU when an application is looking for a particular node, PIM-based pointer chasing consists of (1) traversing the linked data structures *in memory*, and (2) returning only the final node found to the CPU.

Unlike prior works that proposed general architectural models for in-memory computation by embedding logic in main memory [20, 30, 44, 48, 69, 70, 80, 85], we propose to design a *specialized In-Memory PoInter Chasing Accelerator* (IMPICA) that exploits the logic layer of 3D-stacked memory [38, 39, 41, 42, 50]. 3D die-stacked memory achieves low latency (and high bandwidth) by stacking memory dies on top of a logic die, and interconnecting the layers using through-silicon vias (TSVs). Figure 2 shows a binary tree traversal using IMPICA, compared to a traditional architecture where the CPU traverses the binary tree. The traversal sequentially accesses the nodes from the root to a particular node (e.g., **H**→**E**→**A** in Figure 2a). In a traditional architecture (Figure 2b), these serialized accesses to the nodes miss in the caches and three memory requests are sent to memory serially across a high-latency off-chip channel. In contrast, IMPICA traverses the tree inside the logic layer of 3D-stacked memory, and as Figure 2c shows, only the final node (**A**) is sent from the memory to the host CPU in response to the traversal request. Doing the traversal in memory minimizes both traversal latency (as queuing delays in the on-chip interconnect and the CPU-to-memory bus are eliminated) and off-chip bandwidth consumption, as shown in Figure 2c.

Our accelerator architecture has three major advantages. First, it improves performance and reduces bandwidth consumption by eliminating the round trips required for memory accesses over the CPU-to-memory bus. Second, it frees the CPU to execute other work than linked data structure traversal, increasing system throughput. Third, it minimizes the cache contention caused by pointer chasing operations.

3. Design Challenges

We identify and describe two new challenges that are crucial to the performance and functionality of our new pointer chasing accelerator in memory: (1) the *parallelism challenge*, and (2) the *address translation challenge*. Section 4 describes our IMPICA architecture, which centers around two key ideas that solve these two challenges.

3.1. Challenge 1: Parallelism in the Accelerator

A pointer chasing accelerator supporting a multicore system needs to handle *multiple* link traversals (from different cores) in parallel at low cost. A simple accelerator that can handle only one request at a time (which we call a *non-parallel accelerator*) would serialize the requests and could potentially be slower than using multiple CPU cores to perform the multiple traversals. As depicted in Figure 3a, while a non-parallel accelerator speeds up each *individual* pointer chasing operation done by one of the CPU cores due to its shorter memory latency, the accelerator is slower *overall* for two pointer chasing operations, as *multiple cores* can operate in *parallel* on independent pointer chasing operations.

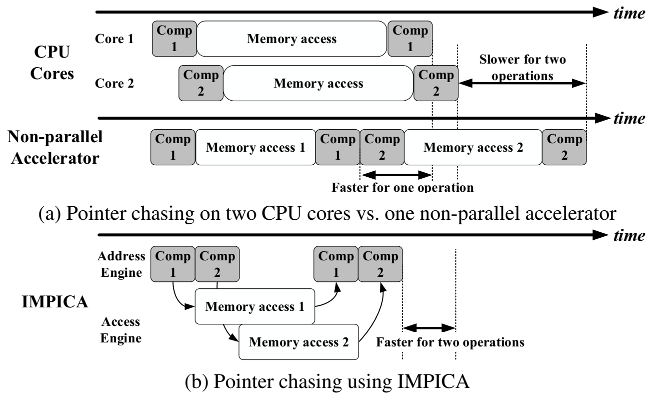


Fig. 3. Execution time of two independent pointer chasing operations, broken down into address computation time (*Comp*) and memory access time.

To overcome this deficiency, an in-memory accelerator needs to exploit parallelism when it services requests. However, the accelerator must do this *at low cost*, due to its placement within the logic layer of 3D-stacked memory, where complex logic, such as out of order execution circuitry, is not currently feasible. The straightforward solution of adding multiple accelerators to service independent pointer chasing operations (e.g., [47]) does not scale well, and also can lead to excessive energy dissipation and die area usage in the logic layer.

A key observation we make is that pointer chasing operations are bottlenecked by memory stalls, as shown in Figure 1. In our evaluation, the memory access time is 10–15× the computation time. As a result, the accelerator spends a significant amount of time waiting for memory, causing its compute resources to sit idle. This makes typical in-order or out-of-order execution engines inefficient for an in-memory pointer-chasing accelerator. If we utilize the hardware resources in a more efficient manner, we can enable parallelism by handling multiple pointer chasing operations *within a single accelerator*.

Based on our observation, we *decouple* address generation from memory accesses in IMPICA using two engines (address engine and access engine), allowing the accelerator to generate addresses from one pointer chasing operation while it concurrently performs memory accesses for a different pointer chasing operation (as shown in Figure 3b). We describe the details of our decoupled accelerator design in Section 4.

3.2. Challenge 2: Virtual Address Translation

A second challenge arises when pointer chasing is moved out of the CPU cores, which are equipped with facilities for address translation. Within the program data structures, each pointer is stored as a virtual address, and requires *translation* to a physical address before its memory access can be performed. This is a challenging task for an in-memory accelerator, which has no easy access to the virtual address translation engine that sits in the CPU core. While sequential array operations could potentially be constrained to work within page boundaries or directly in physical memory, indirect memory accesses that come with pointer-based data structures require some support for virtual memory translation, as they might touch many parts of the virtual address space.

There are two major issues when designing a virtual address translation mechanism for an in-memory accelerator. First, different processor architectures have different page table implementations and formats. This lack of compatibility makes it very expensive to simply replicate the CPU page table walker in the in-memory accelerator as this approach requires replicating TLBs and page walkers for many architecture formats. Second, a page table walk tends to be a high-latency operation involving multiple memory accesses due to the heavily layered formats of a conventional page table. As a result, TLB misses are a major performance bottleneck in data-intensive applications [8]. If the accelerator requires many page table walks that are supported by the CPU's address translation mechanisms, which require high-latency off-chip accesses for the accelerator, its performance can degrade greatly.

To address these issues, we *completely decouple* the page table of IMPICA from that of the CPUs, obviating the need for compatibility between the two tables. This presents us with an opportunity to develop a new page table design that is much more efficient for our in-memory accelerator. We make two key observations about the behavior of a pointer chasing accelerator. First, the accelerator operates only on certain data structures that can be mapped to *contiguous regions* in the virtual address space, which we refer to as *IMPICA regions*. As a result, it is possible to map contiguous IMPICA regions with a *smaller, region-based* page table without needing to duplicate the page table mappings for the *entire* address space. Second, we observe that if we need to map *only* IMPICA regions, we can collapse the hierarchy present in conventional page tables, allowing us to limit the overhead of the IMPICA page table. We describe the IMPICA page table in detail in Section 4.2.

4. IMPICA Architecture

We propose a new in-memory accelerator, IMPICA, that addresses the two design challenges facing accelerators for pointer chasing. The IMPICA architecture consists of a single specialized core designed to decouple address generation from memory accesses. Our approach, which we call *address-access decoupling*, allows us to *efficiently* overcome the parallelism challenge (Section 4.1). The IMPICA core uses a novel *region-based page table* design to perform efficient address translation locally in the accelerator, allowing us to overcome the address translation challenge (Section 4.2).

4.1. IMPICA Core Architecture

Our IMPICA core uses what we call address-access decoupling, where we separate the core into two parts: (1) an *address engine*, which generates the address specified by the pointer; and (2) an *access engine*, which performs memory access operations using addresses generated by the address engine. The key advantage of this design is that the address engine supports fast context switching between multiple pointer chasing operations, allowing it to utilize the idle time during memory access(es) to compute addresses from a different pointer chasing operation. As Figure 3b depicts, an IMPICA core can process multiple pointer chasing operations faster than multiple cores because it has the ability to overlap address generation with memory accesses.

Our address-access decoupling has similarities to, and is in fact inspired by, the decoupled access-execute (DAE) architecture [82], with two key differences. First, the goal of DAE is to exploit instruction-level parallelism (ILP) within a single thread, whereas our goal is to exploit thread-level parallelism (TLP) across pointer chasing operations from multiple threads. Second, unlike DAE, the decoupling in IMPICA does not require any programmer or compiler effort. Our approach is much simpler than both general-purpose DAE and out-of-order execution, as it can switch between different independent execution streams, without the need for dependency checking [35].

Figure 4 shows the architecture of the IMPICA core. The host CPU initializes a pointer chasing operation by moving its code to main memory, and then enqueuing the request in the *request queue* (① in Figure 4). Section 5.1 describes the details of the CPU interface.

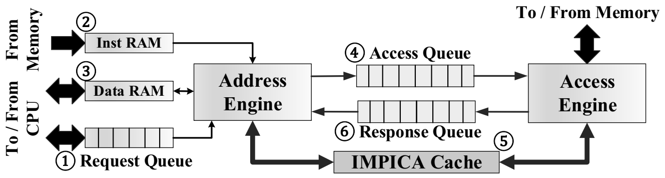


Fig. 4. IMPICA core architecture.

The *address engine* services the enqueued request by loading the pointer chasing code into its *instruction RAM* (②). This engine contains all of IMPICA's functional units, and executes the code in its instruction RAM while using its *data RAM* (③) as a stack. All instructions that do not involve memory accesses, such as ALU operations and control flow, are performed by the address engine. The number of pointer chasing operations that can be processed in parallel is limited by the size of the stack in the data RAM [35].

When the address engine encounters a memory instruction, it enqueues the address (along with the data RAM stack pointer) into the *access queue* (④), and then performs a *context switch* to an independent stream. For the switch, the engine pushes the hardware context (i.e., architectural registers and the program counter) into the data RAM stack. When this is done, the address engine can work on a different pointer chasing operation.

The *access engine* services requests waiting in the access queue. This engine translates the enqueued address from a virtual address to a physical address, using the IMPICA page table (see Section 4.2). It then sends the physical address to the memory controller, which performs the memory access. Since the memory controller handles data retrieval, the access engine can issue multiple requests to the controller without waiting on the data, just as the CPU does today, thus quickly servicing queued requests [35]. Note that the access engine does not contain any functional units.

When the access engine receives data back from the memory controller, it stores this data in the *IMPICA cache* (⑤), a small cache that contains data destined for the address engine. The access queue entry corresponding to the returned data is moved from the access queue to the *response queue* (⑥).

The address engine monitors the response queue. When a response queue entry is ready, the address engine reads it, and uses the stack pointer to access and reload the registers and PC that were pushed onto the data RAM stack. It then resumes execution for the pointer chasing operation, continuing until it encounters the next memory instruction.

4.2. IMPICA Page Table

IMPICA uses a *region-based* page table (RPT) design optimized for in-memory pointer chasing, leveraging the continuous ranges of accesses (*IMPICA regions*) discussed in Section 3.2. Figure 5 shows the structure of the RPT in IMPICA. The RPT is split into three levels: (1) a first-level *region table*, which needs to map only a small number of the contiguously-allocated IMPICA regions; (2) a second-level *flat page table* for each region with a larger (e.g., 2MB) page size; and (3) third-level *small page tables* that use conventional small (e.g., 4KB) pages. In the example in Figure 5, when a 48-bit virtual memory address arrives for translation, bits 47–41 of the address are used to index the region table (① in Figure 5) to find the corresponding flat page table. Bits 40–21 are used to index the flat page table (②), providing the location of the small page table, which is indexed using bits 20–12 (③). The entry in the small page table provides the physical

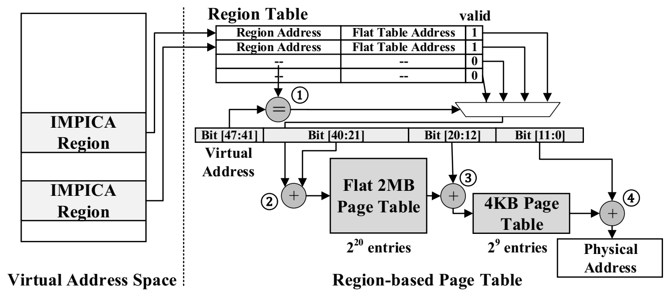


Fig. 5. IMPICA virtual memory architecture.

page number of the page, and bits 11–0 specify the offset within the physical page (④).

The RPT is optimized to take advantage of the properties of pointer chasing. The region table is almost always cached in the IMPICA cache, as the total number of IMPICA regions is small, requiring small storage (e.g., a 4-entry region table needs only 68B of cache space). We employ a flat table with large (e.g., 2MB) pages at the second level in order to reduce the number of page misses, though this requires more memory capacity than the conventional 4-level page table structure. As the number of regions touched by the accelerator is limited, this additional capacity overhead remains constrained. Our page table can optionally use traditional smaller page sizes to maximize memory management flexibility. The OS can freely choose large (2MB) pages or small (4KB) pages at the last level. Thanks to this design, a page walk in the RPT usually results in only two misses, one for the flat page table and another for the last-level small page table. This represents a $2\times$ improvement over a conventional four-level page table, while our flattened page table still provides coverage for a 2TB memory range. The size of the IMPICA region is configurable and can be increased to cover more virtual address space [35]. We believe that our RPT design is general enough for use in a variety of in-memory accelerators that operate on a specific range of memory regions.

We discuss how the OS manages the IMPICA RPT in Section 5.2.

5. Interface and Design Considerations

In this section, we discuss how we expose IMPICA to the CPU and the OS. Section 5.1 describes the communication interface between the CPU and IMPICA. Section 5.2 discusses how the OS manages the page tables in IMPICA. In Section 5.3, we discuss how cache coherence is maintained between the CPU and IMPICA caches.

5.1. CPU Interface and Programming Model

We use a packet-based interface between the CPU and IMPICA. Instead of communicating individual operations or operands, the packet-based interface buffers requests and sends them in a burst to minimize the communication overhead. Executing a function in IMPICA consists of four steps on the interface. (1) The CPU sends to memory a packet comprising the function call and parameters. (2) This packet is written to a specific location in memory, which is memory-mapped to the *data RAM* in IMPICA and triggers IMPICA execution. (3) IMPICA loads the specific function into the *inst RAM* with appropriate parameters, by reading the values from predefined memory locations. (4) Once IMPICA finishes the function execution, it writes the return value back to the memory-mapped locations in the *data RAM*. The CPU periodically polls these locations and receives the IMPICA output. Note that the IMPICA interface is similar to the interface proposed for the Hybrid Memory Cube (HMC) [38,39].

The programming model for IMPICA is similar to the CPU programming model. An IMPICA program can be written as a function in the application code with a compiler directive. The compiler compiles these functions into IMPICA instructions and wraps the function calls with communication codes that utilize the CPU–IMPICA interface.

5.2. Page Table Management

In order for the RPT to identify IMPICA regions, the regions must be tagged by the application. For this, the application uses a special API to allocate pointer-based data structures. This API allocates memory to a contiguous virtual address space. To ensure that all API allocations are contiguous, the OS reserves a portion of the unused virtual address space for IMPICA, and always allocates memory for IMPICA regions from this portion. The use of such a special API requires minimal changes to applications, and it allows the system to provide more efficient virtual address translation. This also allows us to ensure that when multiple memory stacks are present within the system, the OS can allocate all IMPICA regions belonging to a single application (along with the associated IMPICA page table) into one memory stack, thereby avoiding the need for the accelerator to communicate with a remote memory stack.

The OS maintains coherence between the IMPICA RPT and the CPU page table. When memory is allocated in the IMPICA region, the OS allocates the IMPICA page table. The OS also shoots down TLB entries in IMPICA if the CPU performs any updates to IMPICA regions. While this makes the OS page fault handler more complex, the additional complexity does not cause a noticeable performance impact, as page faults occur rarely and take a long time to service in the CPU.

5.3. Cache Coherence

Coherence must be maintained between the CPU and IMPICA caches, and with memory, to avoid using stale data and thus ensure correct execution. We maintain coherence by executing *every* function that operates on the IMPICA regions in the accelerator. This solution guarantees that no data is shared between the CPU and IMPICA, and that IMPICA always works on up-to-date data. Other PIM coherence solutions (e.g., [2,10,26]) can also be used to allow CPU to update the linked data structures, but we choose not to employ these solutions in our evaluation, as our workloads do not perform any such updates.

6. Methodology

We use the gem5 [9] full-system simulator with DRAMSim2 [74] to evaluate our proposed design. We choose the 64-bit ARMv8 architecture, the accuracy of which has been validated against real hardware [32]. We model the internal memory bandwidth of the memory stack to be $4\times$ that of the external bandwidth, similar to the configuration used in prior works [22, 96]. Our simulation parameters are summarized in Table 1. Our technical report [35] provides more detail on the IMPICA configuration.

Table 1. Major simulation parameters.

Processor	
ISA	ARMv8 (64-bits)
Core Configuration	4 OoO cores, 2 GHz, 8 wide, 128-entry ROB
Operating System	64-bit Linux from Linaro [54]
L1 I/D Cache	32KB/2-way each, 2-cycle
L2 Cache	1MB/8-way, shared, 20-cycle
DRAM Parameters	
Memory Configuration	DDR3-1600, 8 banks/device, FR-FCFS scheduler
DRAM Bus Bandwidth	12.8 GB/s for CPU, 51.2 GB/s for IMPICA
IMPICA Configuration	
Accelerator Core	500 MHz, 16 entries for each queue
Cache	32KB[2]/ 2-way
Address Translator	32 TLB entries with region-based page table
RAM	16KB Data RAM and 16KB Inst RAM

6.1. Workloads

We use three data-intensive microbenchmarks, which are essential building blocks in a wide range of workloads, to evaluate the native performance of performance chasing operations: linked lists, hash tables, and B-trees. We also evaluate the performance improvement in a real data-intensive workload, measuring the transaction latency and throughout of DBx1000 [94], an in-memory OLTP database. We modify all four workloads to offload each pointer chasing request to IMPICA. To minimize communication overhead, we map the IMPICA registers to user mode address space, thereby avoiding the need for costly kernel code intervention.

Linked list. We use the linked list traversal microbenchmark [98] derived from the *health* workload in the Olden benchmark suite [73]. The parameters are configured to approximate the performance of the *health* workload. We measure the performance of the linked list traversal after 30,000 iterations.

Hash table. We create a microbenchmark from the hash table implementation of *Memcached* [25]. The hash table in Memcached resolves hash collisions using chaining via linked lists. When there are more than $1.5n$ items in a table of n buckets, it doubles the number of

[2]We sweep the size of the IMPICA cache from 32KB to 128KB, and find that it has negligible effect on our results [35].

buckets. We follow this rule by inserting 1.5×2^{20} random keys into a hash table with 2^{20} buckets. We run evaluations for 100,000 random key look-ups.

B-tree. We use the B-tree implementation of DBx1000 for our B-tree microbenchmark. It is a 16-way B-tree that uses a 64-bit integer as the key of each node. We randomly generate 3,000,000 keys and insert them into the B-tree. After the insertions, we measure the performance of the B-tree traversal with 100,000 random keys. This is the most time consuming operation in the database index lookup.

DBx1000. We run DBx1000 [94] with the TPC-C benchmark [87]. We set up the TPC-C tables with 2,000 customers and 100,000 items. For each run, we spawn 4 threads and bind them to 4 different CPUs to achieve maximum throughput. We run each thread for a warm-up period for the duration of 2,000 transactions, and then record the software and hardware statistics for the next 5,000 transactions per thread,[3] which takes 300–500 million CPU cycles.

6.2. Die Area and Energy Estimation

We estimate the die area of the IMPICA processing logic at the 40nm process node based on recently-published work [57]. We include the most important components: processor cores, L1/L2 caches, and the memory controller. We use the area of ARM Cortex-A57 [4, 24], a small embedded processor, for the main CPU. We *conservatively* estimate the die area of IMPICA using the area of the Cortex-R4 [5], an 8-stage dual issue RISC processor with 32 KB I/D caches. Table 2 lists the area estimate of each component.

Table 2. Die area estimates using a 40nm process.

Baseline CPU (Cortex-A57)	5.85 mm^2 per core
L2 Cache	5 mm^2 per MB
Memory Controller	10 mm^2
IMPICA Core (including 32 KB I/D caches)	0.45 mm^2

IMPICA comprises only 7.6% the area of a single baseline CPU core, or only 1.2% the total area of the baseline chip (which includes four CPU cores, 1MB L2 cache, and one memory controller). Note that we conservatively model IMPICA as a RISC core. A much more specialized engine can be designed for IMPICA to solely execute pointer chasing code. Doing so would reduce the area and energy overheads of IMPICA greatly, but can reduce the generality of the pointer chasing access patterns that IMPICA can accelerate. We leave this for future work.

We use McPAT [52] to estimate the energy consumption of the CPU, caches, memory controllers, and IMPICA. We conservatively use the configuration of the Cortex-R4 to estimate the energy consumed by IMPICA. We use DRAMSim2 [74] to analyze DRAM energy.

7. Evaluation

We first evaluate the effect of IMPICA on system performance, using both our microbenchmarks (Section 7.1) and the DBx1000 database (Section 7.2). We investigate the impact of different IMPICA page table designs in Section 7.3, and examine system energy consumption in Section 7.4. We compare a system containing IMPICA to an accelerator-free baseline that includes an additional 128KB of L2 cache (which is equivalent to the area of IMPICA) to ensure area-equivalence across evaluated systems.

7.1. Microbenchmark Performance

Figure 6 shows the speedup of IMPICA and the baseline with extra 128KB of L2 cache over the baseline for each microbenchmark. IMPICA achieves significant speedups across all three data structures — 1.92× for the linked list, 1.29× for the hash table, and 1.18× for the B-tree. In contrast, the extra 128KB of L2 cache provides very small speedup (1.03×, 1.01×, and 1.02×, respectively). We conclude that IMPICA is much more effective than the area-equivalent additional L2 cache for pointer chasing operations.

To provide insight into why IMPICA improves performance, we present total (CPU and IMPICA) TLB misses per kilo instructions (MPKI), cache miss latency, and total memory bandwidth usage for these microbenchmarks in Figure 7. We make three observations.

[3]Based on our experiments on a real Intel Xeon machine, we find that this is large enough to satisfactorily represent the behavior of 1,000,000 transactions [35].

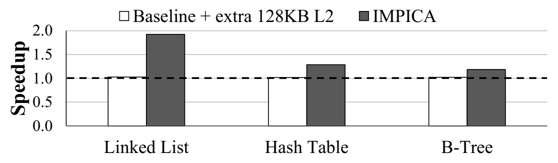
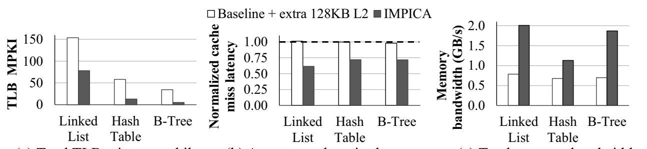


Fig. 6. Microbenchmark performance with IMPICA.

First, a major factor contributing to the performance improvement is the reduction in TLB misses. The TLB MPKI in Figure 7a depicts the total (i.e., combined CPU and IMPICA) TLB misses in both the baseline system and IMPICA. The pointer chasing operations have low locality and pollute the CPU TLB. This leads to a higher overall TLB miss rate in the application. With IMPICA, the pointer chasing operations are offloaded to the accelerator. This reduces the pollution and contention at the CPU TLB, reducing the overall number of TLB misses. The linked list has a significantly higher TLB MPKI than the other data structures because linked list traversal requires far fewer instructions in an iteration. It simply accesses the next pointer, while a hash table or a B-tree traversal needs to compare the keys in the node to determine the next step.



(a) Total TLB misses per kilo instructions (MPKI) (b) Average cache miss latency (c) Total memory bandwidth utilization (GB/s)

Fig. 7. Key architectural statistics for the microbenchmarks.

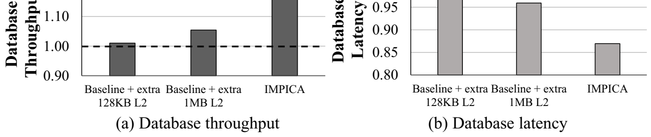
Second, we observe a significant reduction in cache miss latency with IMPICA. Figure 7b compares the average cache miss latency between the baseline last-level cache and the IMPICA cache. On average, the cache miss latency of IMPICA is only 60–70% of the baseline cache miss latency. This is because IMPICA leverages the faster and wider TSVs in 3D-stacked memory as opposed to the high latency and narrow DRAM interface used by the CPU.

Third, as Figure 7c shows, IMPICA effectively utilizes the internal memory bandwidth in 3D-stacked memory, which is cheap and abundant. There are two reasons for high bandwidth utilization: (1) IMPICA runs much faster than the baseline so it generates more traffic within the same amount time; and (2) IMPICA always accesses memory at a larger granularity, retrieving each full node in a linked data structure with a single memory request, while a CPU issues multiple requests for each node as it can fetch only one cache line at a time. The CPU can avoid using some of its limited memory bandwidth by skipping some fields in the data structure that are not needed for the current loop iteration. For example, some keys and pointers in a B-tree node can be skipped whenever a match is found. In contrast, IMPICA utilizes the wide internal bandwidth of 3D-stacked memory to retrieve a full node on each access (more detail is in our tech report [35]).

We conclude that IMPICA is effective at significantly improving the performance of important linked data structures.

7.2. Real Database Throughput and Latency

Figure 8 presents two key performance metrics for our evaluation of DBx1000: *database throughput* and *database latency*. *Database throughput* represents how many transactions are completed within a certain period, while *database latency* is the average time to complete a transaction. We normalize the results of three configurations to the baseline. As mentioned earlier, the die area increase of IMPICA is similar to a 128KB cache. To understand the effect of additional LLC space better, we also show the results of adding 1MB of cache, which takes about 8× the area of IMPICA, to the baseline. We make two observations from our analysis of DBx1000.



(a) Database throughput (b) Database latency

Fig. 8. Performance results for DBx1000, normalized to the baseline.

First, IMPICA improves the overall database throughput by 16% and reduces the average database transaction latency by 13%. The performance improvement is due to three reasons: (1) database indexing becomes faster with IMPICA, (2) offloading database indexing to IMPICA reduces the TLB and cache contention due to pointer chasing in the CPU, and (3) the CPU can do other useful tasks in parallel while waiting for IMPICA. Note that our profiling results in Figure 1 show that DBx1000 spends 19% of its time on pointer chasing. Therefore, a 16% overall improvement is very close to the upper bound that *any* pointer chasing accelerator can achieve for this database.

Second, IMPICA yields much higher database throughput than simply providing additional cache capacity. IMPICA improves the database throughput by 16%, while an extra 128KB of cache (with a similar area overhead as IMPICA) does so by only 2%, and an extra 1MB of cache (8× the area of IMPICA) by only 5%.

We conclude that by accelerating the fundamental pointer chasing operation, IMPICA can efficiently improve the performance of a sophisticated real workload.

7.3. Sensitivity to the IMPICA TLB Size & Page Table Design

To understand the effect of different TLB sizes and page table designs in IMPICA, we evaluate the speedup in the amount of time spent on address translation for IMPICA when different IMPICA TLB sizes (32 and 64 entries) and accelerator page table structures (the baseline 4-level page table; and the region-based page table, or RPT) are used inside the accelerator. Figure 9 shows the speedup in address translation time relative to IMPICA with a 32-entry TLB and the conventional 4-level page table. Two observations are in order.

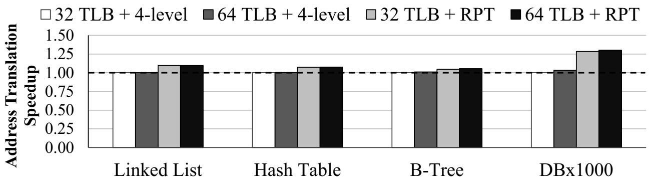


Fig. 9. Speedup of address translation with different designs.

First, the performance of IMPICA is largely unaffected by increasing the IMPICA TLB size. Doubling the IMPICA TLB entries from 32 to 64 barely improves the address translation time. This observation reflects the irregular nature of pointer chasing. Second, the benefit of the RPT is much more significant in a sophisticated workload (DBx1000) than in microbenchmarks. This is because the working set size of the microbenchmarks is much smaller than that of the database system. When the working set is small, the operating system needs only a small number of page table entries in the first and second levels of a traditional page table. These entries are used frequently, so they stay in the IMPICA cache much longer, reducing the address translation overhead. This caching benefit goes away with a larger working set, which would require a significantly larger TLB and IMPICA cache to reap locality benefits. The benefit of RPT is more significant in such a case because RPT does not rely on this caching effect. Its region table is *always* small irrespective of the workload size and it has fewer page table levels.

7.4. Energy Efficiency

Figure 10 shows the system energy consumption for the microbenchmarks and DBx1000. We observe that the overall system *power* increases by 5.6% on average (not shown), due to the addition of IMPICA and higher utilization of internal memory bandwidth. However, as IMPICA significantly reduces the execution time of the evaluated workloads, the overall system *energy* consumption reduces by 41%, 24%, and 10% for the microbenchmarks, and by 6% for DBx1000. We conclude that IMPICA is an energy-efficient accelerator for pointer chasing.

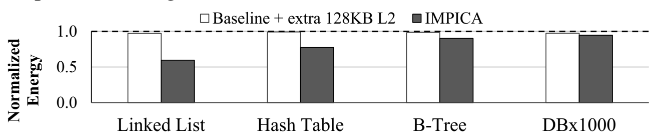


Fig. 10. Effect of IMPICA on energy consumption.

8. Related Work

To our knowledge, this is the first work to (1) leverage the logic layer in 3D-stacked memory to accelerate pointer chasing in linked data structures, (2) propose an address-access decoupled architecture to tackle the *parallelism challenge* for in-memory accelerators, and (3) propose a general and efficient page table mechanism to tackle the *address translation challenge* for in-memory accelerators.

Many prior works investigate the pointer chasing problem and propose solutions using software or hardware prefetching mechanisms (e.g., [14–16, 18, 36, 37, 43, 45, 55, 58, 59, 61, 75, 76, 83, 92, 93, 95, 99]). The effectiveness of this approach is fundamentally dependent on the accuracy and generality of address prediction. In this work, we improve the performance of pointer chasing operation with an in-memory accelerator, inspired by the concept of processing-in-memory (PIM) (e.g., [20, 30, 44, 48, 69, 70, 80, 85]). There are many recent proposals that aim to improve the performance of data intensive workloads using accelerators in CPUs or in memory (e.g., [1–3, 6, 7, 11–13, 22, 27, 31, 34, 46, 47, 51, 53, 72, 77–79, 90, 91, 96, 97]). Here, we briefly discuss these related works. None of these works provide an accelerator for pointer chasing in memory.

Prefetching for Linked Data Structures. Many works propose mechanisms to prefetch data in linked data structures to hide memory latency. These proposals are hardware-based (e.g., [14,16,36,37,43,61, 76,95]), software-based (e.g., [55,59,75,92,93]), pre-execution-based (e.g., [15, 58, 83, 99]), or software/hardware-cooperative (e.g., [18, 63, 76]) mechanisms. These approaches have two major drawbacks. First, they rely on predictable traversal sequences to prefetch accurately. These mechanisms can become very inefficient if the linked data structure is complex or when access patterns are less regular. Second, the pointer chasing is performed at the CPU cores or at the memory controller, which likely leads to pollution of the CPU caches and TLBs by these irregular memory accesses.

Processing-in-Memory (PIM). The idea of PIM can be traced back to 1970 [85]. A number of works have attempted to realize the idea of placing a general-purpose processor in memory with different approaches since the 1980s (e.g., [20,30,44,48,69,70,80]). Our work is inspired by the PIM concept, but our focus is on the design challenges of an in-memory pointer chasing accelerator (and not on the design of a general-purpose processor in memory).

Accelerators in 3D-Stacked Memory. The rapid advances in 3D-stacked memory technology have revived the idea of PIM. There are several in-memory accelerator proposals that leverage 3D-stacked memory designs for data-intensive applications. These include accelerators for MapReduce [72], matrix multiplication [97], data reorganization [3], graph processing [1], databases [7], data-intensive processing [31], and machine learning workloads [12, 46, 51]. Some works propose more generic architectures by adding PIM-enabled instructions [2], GPGPUs [34, 71, 96], or reconfigurable hardware [22, 27] to memory. Our proposal has three major contributions over these works. First, none of these works focus on pointer chasing, a key operation in many important workloads. Second, we expose two new design challenges for in-memory accelerators (parallelism and address translation). Third, we propose an efficient hardware architecture and page table structure to tackle these two challenges, which we believe can also be employed in other in-memory accelerators.

Accelerators in CPUs. There have been various CPU-side accelerator proposals for database systems (e.g., [13,47,90,91]) and key-value stores [53]. Among them, Widx [47], a database indexing accelerator, is the closest to our work. Widx is a set of custom RISC cores in the CPU to accelerate hash index lookups. While a hash table is one of our data structures of interest, IMPICA differs from Widx in three ways. First, it is an *in-memory* (as opposed to CPU-side) accelerator, which poses very different design challenges. Second, we solve the address translation challenge for in-memory accelerators, while Widx uses the CPU address translation structures. Third, we enable parallelism within a single accelerator core, while Widx achieves parallelism by replicating several RISC cores.

9. Conclusion

We introduce the design and evaluation of an *in-memory accelerator*, called IMPICA, for performing pointer chasing operations in 3D-stacked memory. We identify two major challenges in the design of such an in-memory accelerator: (1) the *parallelism challenge* and (2) the *address translation challenge*. We provide new solutions to these two challenges: (1) *address-access decoupling* solves the parallelism challenge by decoupling the address generation from memory accesses in pointer chasing operations and exploiting the idle time during memory accesses to execute multiple pointer chasing operations in parallel, and (2) the *region-based page table* in 3D-stacked memory solves the address translation challenge by tracking only those limited set of virtual memory regions that are accessed by pointer chasing operations. Our evaluations show that for both commonly-used linked data structures and a real database application, IMPICA significantly improves both performance and energy efficiency. We conclude that IMPICA is an efficient and effective accelerator design for pointer chasing. We also believe that the two challenges we identify (parallelism and address translation) exist in various forms in other in-memory accelerators (e.g., for graph processing), and, therefore, our solutions to these challenges can be adapted for use by a broad class of (in-memory) accelerators.

Acknowledgments

We thank the anonymous reviewers and the SAFARI Research Group for their feedback. Special thanks to David G. Andersen and Vivek Seshadri for feedback. We acknowledge the support of our industrial partners: Google, Intel, NVIDIA, Samsung and Seagate. This research was partially supported by NSF (grants 1212962, 1320531, 1409723), ISTC-CC, SRC, and DSSC.

References

[1] J. Ahn *et al.*, “A scalable processing-in-memory accelerator for parallel graph processing,” in *ISCA*, 2015.
[2] J. Ahn *et al.*, “PIM-Enabled Instructions: A low-overhead, locality-aware processing-in-memory architecture,” in *ISCA*, 2015.
[3] B. Akin *et al.*, “Data reorganization in memory using 3D-stacked DRAM,” in *ISCA*, 2015.
[4] ARM, “ARM Cortex-A57,” http://www.arm.com/products/processors/cortex-a/cortex-a57-processor.php.
[5] ARM, “ARM Cortex-R4,” http://www.arm.com/products/processors/cortex-r/cortex-r4.php.
[6] H. Asghari-Moghaddam *et al.*, “Chameleon: Versatile and practical near-DRAM acceleration architecture for large memory systems,” in *MICRO*, 2016.
[7] O. O. Babarinsa and S. Idreos, “JAFAR: Near-data processing for databases,” in *SIGMOD*, 2015.
[8] A. Basu *et al.*, “Efficient virtual memory for big memory servers,” in *ISCA*, 2013.
[9] N. Binkert *et al.*, “The gem5 simulator,” *CAN*, 2011.
[10] A. Boroumand *et al.*, “LazyPIM: An efficient cache coherence mechanism for processing-in-memory,” *CAL*, 2016.
[11] K. K. Chang *et al.*, “Low-Cost Inter-Linked Subarrays (LISA): Enabling fast inter-subarray data movement in dram”,” in *HPCA*, 2016.
[12] P. Chi *et al.*, “A novel processing-in-memory architecture for neural network computation in ReRAM-based main memory,” in *ISCA*, 2016.
[13] E. S. Chung *et al.*, “LINQits: Big data on little clients,” in *ISCA*, 2013.
[14] J. D. Collins *et al.*, “Pointer cache assisted prefetching,” in *MICRO*, 2002.
[15] J. D. Collins *et al.*, “Speculative precomputation: Long-range prefetching of delinquent loads,” in *ISCA*, 2001.
[16] R. Cooksey *et al.*, “A stateless, content-directed data prefetching mechanism,” in *ASPLOS*, 2002.
[17] N. C. Crago and S. J. Patel, “OUTRIDER: Efficient memory latency tolerance with decoupled strands,” in *ISCA*, 2011.
[18] E. Ebrahimi *et al.*, “Techniques for bandwidth-efficient prefetching of linked data structures in hybrid prefetching systems,” in *HPCA*, 2009.
[19] E. Ebrahimi *et al.*, “Coordinated control of multiple prefetchers in multi-core systems,” in *MICRO*, 2009.
[20] D. G. Elliott *et al.*, “Computational RAM: A memory-SIMD hybrid and its application to DSP,” in *CICC*, 1992.
[21] R. Elmasri, *Fundamentals of database systems*. Pearson, 2007.
[22] A. Farmahini-Farahani *et al.*, “NDA: Near-DRAM acceleration architecture leveraging commodity DRAM devices and standard memory modules,” in *HPCA*, 2015.
[23] M. Ferdman *et al.*, “Clearing the clouds: A study of emerging scale-out workloads on modern hardware,” in *ASPLOS*, 2012.
[24] M. Filippo, “Technology preview: ARM next generation processing,” *ARM Techcon*, 2012.
[25] B. Fitzpatrick, “Distributed caching with Memcached,” *Linux journal*, 2004.
[26] M. Gao *et al.*, “Practical near-data processing for in-memory analytics frameworks,” in *PACT*, 2015.
[27] M. Gao and C. Kozyrakis, “HRL: Efficient and flexible reconfigurable logic for near-data processing,” in *HPCA*, 2016.
[28] D. Giampaolo, *Practical file system design with the BE file system*. Morgan Kaufmann Publishers Inc., 1998.
[29] A. Glew, “MLP yes! ILP no!” in *ASPLOS WACI*, 1998.
[30] M. Gokhale *et al.*, “Processing in memory: The Terasys massively parallel PIM array,” *IEEE Computer*, 1995.
[31] B. Gu *et al.*, “Biscuit: A framework for near-data processing of big data workloads,” in *ISCA*, 2016.
[32] A. Gutierrez *et al.*, “Sources of error in full-system simulation,” in *ISPASS*, 2014.
[33] M. Hashemi *et al.*, “Acclerating dependent cache misses with an enhanced memory controller,” in *ISCA*, 2016.
[34] K. Hsieh *et al.*, “Transparent offloading and mapping (TOM): Enabling programmer-transparent near-data processing in GPU systems,” in *ISCA*, 2016.
[35] K. Hsieh *et al.*, “A pointer chasing accelerator for 3D-stacked memory,” *CMU SAFARI Technical Report No. 2016-007*, 2016.
[36] Z. Hu *et al.*, “TCP: Tag correlating prefetchers,” in *HPCA*, 2003.
[37] C. J. Hughes and S. V. Adve, “Memory-side prefetching for linked data structures for processor-in-memory systems,” *JPDC*, 2005.
[38] Hybrid Memory Cube Consortium, “HMC Specification 1.1,” 2013.
[39] Hybrid Memory Cube Consortium, “HMC Specification 2.0,” 2014.
[40] Intel, “Intel Xeon Processor W3550,” 2009.
[41] J. Jeddeloh and B. Keeth, “Hybrid memory cube: New DRAM architecture increases density and performance,” in *VLSIT*, 2012.
[42] JEDEC, “High Bandwidth Memory (HBM) DRAM,” Standard No. JESD235, 2013.
[43] D. Joseph and D. Grunwald, “Prefetching using Markov predictors,” in *ISCA*, 1997.
[44] Y. Kang *et al.*, “FlexRAM: Toward an advanced intelligent memory system,” in *ICCD*, 1999.
[45] M. Karlsson *et al.*, “A prefetching technique for irregular accesses to linked data structures,” in *HPCA*, 2000.
[46] D. Kim *et al.*, “Neurocube: A programmable digital neuromorphic architecture with high-density 3D memory,” in *ISCA*, 2016.
[47] Y. O. Koçberber *et al.*, “Meet the walkers: Accelerating index traversals for in-memory databases,” in *MICRO*, 2013.
[48] P. M. Kogge, “EXECUBE–A new architecture for scaleable MPPs,” in *ICPP*, 1994.
[49] L. Kurian *et al.*, “Memory latency effects in decoupled architectures with a single data memory module,” in *ISCA*, 1992.
[50] D. Lee *et al.*, “Simultaneous Multi-Layer Access: Improving 3D-stacked memory bandwidth at low cost,” *TACO*, 2016.
[51] J. H. Lee *et al.*, “BSSync: Processing near memory for machine learning workloads with bounded staleness consistency models,” in *PACT*, 2015.
[52] S. Li *et al.*, “The McPAT framework for multicore and manycore architectures: Simultaneously modeling power, area, and timing,” *TACO*, 2013.
[53] K. T. Lim *et al.*, “Thin servers with smart pipes: Designing SoC accelerators for Memcached,” in *ISCA*, 2013.
[54] Linaro, “Linux gem5 support,” 2014.
[55] M. H. Lipasti *et al.*, “SPAID: Software prefetching in pointer- and call-intensive environments,” in *MICRO*, 1995.
[56] G. H. Loh *et al.*, “A processing in memory taxonomy and a case for studying fixed-function PIM,” in *WoNDP*, 2013.
[57] P. Lotfi-Kamran *et al.*, “Scale-out processors,” in *ISCA*, 2012.
[58] C. Luk, “Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors,” in *ISCA*, 2001.
[59] C. Luk and T. C. Mowry, “Compiler-based prefetching for recursive data structures,” in *ASPLOS*, 1996.
[60] Y. Mao *et al.*, “Cache craftiness for fast multicore key-value storage,” in *EuroSys*, 2012.
[61] O. Mutlu *et al.*, “Address-value delta (AVD) prediction: Increasing the effectiveness of runahead execution by exploiting regular memory allocation patterns,” in *MICRO*, 2005.
[62] O. Mutlu *et al.*, “Techniques for efficient processing in runahead execution engines,” in *ISCA*, 2005.
[63] O. Mutlu *et al.*, “Address-value delta (AVD) prediction: A hardware technique for efficiently parallelizing dependent cache misses,” *IEEE TC*, 2006.
[64] O. Mutlu *et al.*, “Efficient runahead execution: Power-efficient memory latency tolerance,” *IEEE Micro*, 2006.
[65] O. Mutlu and T. Moscibroda, “Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared DRAM systems,” in *ISCA*, 2008.
[66] O. Mutlu *et al.*, “Runahead execution: An alternative to very large instruction windows for out-of-order processors,” in *HPCA*, 2003.
[67] O. Mutlu *et al.*, “Runahead execution: An effective alternative to large instruction windows,” *IEEE Micro*, 2003.
[68] B. Naylor *et al.*, “Merging BSP trees yields polyhedral set operations,” in *SIGGRAPH*, 1990.
[69] M. Oskin *et al.*, “Active pages: A computation model for intelligent memory,” in *ISCA*, 1998.
[70] D. Patterson *et al.*, “A case for intelligent RAM,” *IEEE Micro*, 1997.
[71] A. Pattnaik *et al.*, “Kernel scheduling techniques for PIM-assisted GPU architectures,” in *PACT*, 2016.
[72] S. H. Pugsley *et al.*, “NDC: Analyzing the impact of 3D-stacked memory+logic devices on MapReduce workloads,” in *ISPASS*, 2014.
[73] A. Rogers *et al.*, “Supporting dynamic data structures on distributed-memory machines,” *TOPLAS*, 1995.
[74] P. Rosenfeld *et al.*, “DRAMSim2: A cycle accurate memory system simulator,” *CAL*, 2011.
[75] A. Roth *et al.*, “Dependence based prefetching for linked data structures,” in *ASPLOS*, 1998.
[76] A. Roth and G. S. Sohi, “Effective jump-pointer prefetching for linked data structures,” in *ISCA*, 1999.
[77] V. Seshadri *et al.*, “Gather-Scatter DRAM: In-DRAM address translation to improve the spatial locality of non-unit strided accesses,” in *MICRO*, 2015.
[78] V. Seshadri *et al.*, “Fast bulk bitwise AND and OR in DRAM,” *CAL*, 2015.
[79] V. Seshadri *et al.*, “RowClone: Fast and energy-efficient in-DRAM bulk data copy and initialization,” in *MICRO*, 2013.
[80] D. E. Shaw *et al.*, “The NON-VON database machine: A brief overview,” *IEEE Database Eng. Bull.*, 1981.
[81] J. Shun and G. E. Blelloch, “Ligra: A lightweight graph processing framework for shared memory,” in *PPoPP*, 2013.
[82] J. E. Smith, “Decoupled access/execute computer architectures,” in *ISCA*, 1982.
[83] Y. Solihin *et al.*, “Using a user-level memory thread for correlation prefetching,” in *ISCA*, 2002.
[84] S. Srinath *et al.*, “Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers,” in *HPCA*, 2007.
[85] H. S. Stone, “A logic-in-memory computer,” *IEEE TC*, 1970.
[86] R. M. Tomasulo, “An efficient algorithm for exploiting multiple arithmetic units,” *IBM JRD*, 1967.
[87] TPC, “Transaction processing performance council,” http://www.tpc.org.
[88] M. Waldvogel *et al.*, “Scalable high speed IP routing lookups,” in *SIGCOMM*, 1997.
[89] P. R. Wilson, “Uniprocessor garbage collection techniques,” in *IWMM*, 1992.
[90] L. Wu *et al.*, “Navigating big data with high-throughput, energy-efficient data partitioning,” in *ISCA*, 2013.
[91] L. Wu *et al.*, “Q100: The architecture and design of a database processing unit,” in *ASPLOS*, 2014.
[92] Y. Wu, “Efficient discovery of regular stride patterns in irregular programs,” in *PLDI*, 2002.
[93] C. Yang and A. R. Lebeck, “Push vs. pull: Data movement for linked data structures,” in *ICS*, 2000.
[94] X. Yu *et al.*, “Staring into the abyss: An evaluation of concurrency control with one thousand cores,” *VLDB*, 2014.
[95] X. Yu *et al.*, “IMP: Indirect memory prefetcher,” in *MICRO*, 2015.
[96] D. P. Zhang *et al.*, “TOP-PIM: Throughput-oriented programmable processing in memory,” in *HPDC*, 2014.
[97] Q. Zhu *et al.*, “Accelerating sparse matrix-matrix multiplication with 3D-stacked logic-in-memory hardware,” in *HPEC*, 2013.
[98] C. B. Zilles, “Benchmark health considered harmful,” *CAN*, 2001.
[99] C. B. Zilles and G. S. Sohi, “Execution-based prediction using speculative slices,” in *ISCA*, 2001.