Tiny but Mighty: Designing and Realizing Scalable Latency Tolerance for Manycore SoCs

Marcelo Orenes-Vera
Princeton University

Aninda Manocha
Princeton University

Jonathan Balkind
UC Santa Barbara

Fei Gao
Princeton University

Juan L. Aragón
University of Murcia

David Wentzlaff
Princeton University

Margaret Martonosi
Princeton University

ABSTRACT

Modern computing systems employ significant heterogeneity and specialization to meet performance targets at manageable power. However, memory latency bottlenecks remain problematic, particularly for sparse neural network and graph analytic applications where indirect memory accesses (IMAs) challenge the memory hierarchy.

Decades of prior art have proposed hardware and software mechanisms to mitigate IMA latency, but they fail to analyze real-chip considerations, especially when used in SoCs and manycores. In this paper, we revisit many of these techniques while taking into account manycore integration and verification.

We present the first system implementation of latency tolerance hardware that provides significant speedups without requiring any memory hierarchy or processor tile modifications. This is achieved through a Memory Access Parallel-Load Engine (MAPLE), integrated through the Network-on-Chip (NoC) in a scalable manner. Our hardware-software co-design allows programs to perform long-latency memory accesses asynchronously from the core, avoiding pipeline stalls, and enabling greater memory parallelism (MLP).

In April 2021 we taped out a manycore chip that includes tens of MAPLE instances for efficient data supply. MAPLE demonstrates a full RTL implementation of out-of-core latency-mitigation hardware, with virtual memory support and automated compilation targetting it. This paper evaluates MAPLE integrated with a dual-core FPGA prototype running applications with full SMP Linux, and demonstrates geomean speedups of 2.35× and 2.27× over software-based prefetching and decoupling, respectively. Compared to state-of-the-art hardware, it provides geomean speedups of 1.82× and 1.72× over prefetching and decoupling techniques.

CCS CONCEPTS

• **Computer systems organization → Multicore architectures**; *Reconfigurable computing*; Heterogeneous (hybrid) systems.

ISCA '22, June 18–22, 2022, New York, NY, USA

ACM ISBN 978-1-4503-8610-4/22/06.
https://doi.org/10.1145/3470496.3527400

KEYWORDS

memory, latency tolerance, decoupling, modular RTL

ACM Reference Format:
Marcelo Orenes-Vera, Aninda Manocha, Jonathan Balkind, Fei Gao, Juan L. Aragón, David Wentzlaff, and Margaret Martonosi. 2022. Tiny but Mighty: Designing and Realizing Scalable Latency Tolerance for Manycore SoCs. In *The 49th Annual International Symposium on Computer Architecture (ISCA '22), June 18–22, 2022, New York, NY, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3470496.3527400

1 INTRODUCTION

The "memory wall" becomes even more challenging in accelerator-rich systems. From the perspective of Amdahl's Law, as specialized accelerators speed up computation, memory operations that supply data represent a bigger portion of the runtime [53]. Workloads with cache-unfriendly irregular memory access patterns are particularly bottlenecked, such as those in the domains of graph analytics and sparse linear algebra. Their irregularity arises from Indirect Memory Accesses (IMAs) that require many off-chip, long-latency accesses to DRAM. Software optimizations to reduce memory latency often require increased code complexity and reduced portability, and can incur overheads that limit performance gains [31]. Thus, hardware innovations are necessary.

Table 1 shows much of the 40 years of prior work in latency mitigation of IMAs. One might think that these hardware innovations are easy to include in real chips, but that is often not the case due to complex core or cache modifications [9, 41, 54, 56], the need for new ISA instructions [15, 43, 45, 55, 59], or excessive area overheads per core [22, 62]. Moreover, deep microarchitecture changes are hard to make in practice because of the verification burden. For SoC integration, it is often easier to incorporate off-the-shelf cores.

These observations are not abstract for us; our exploration into the prior work in latency tolerance started with the goal of fabricating a chip to efficiently process sparse algebra and graph analytic workloads. Prior work has leveraged SMT and beefy OoO to hoist accesses and thus mitigate the latency of IMAs [35, 45, 58]. However, we choose to use many slim in-order cores instead of a few out-of-order (OoO) cores, because latter are generally not effective for irregular memory accesses without additional specialization. In-order cores with specialized hardware to handle irregular accesses offer better performance density for our application domain [56].

Previously proposed latency tolerance techniques fall short analyzing trade-offs that arise from manycore integration or real-chip

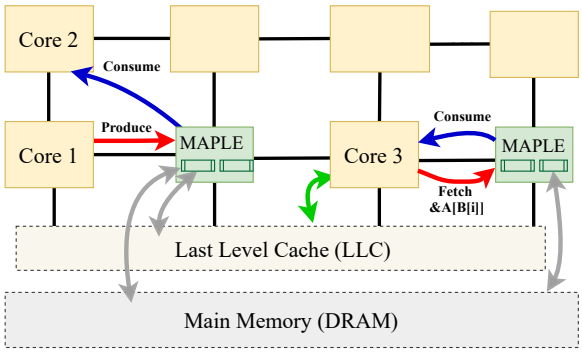

Figure 1: MAPLE is an area-efficient alternative to fetch irregular memory patterns. Each MAPLE can supply data for up to 8 cores in parallel. For clarity we depict the two scenarios of cores using MAPLE separately. The arrows are MAPLE API operations (off-the-shelf cores can target MAPLE using Memory-Mapped loads and stores). In decoupling mode, *Core 1* runs ahead of time producing pointers to irregular memory locations for MAPLE to fetch and store in one of its scratchpad hardware queues; *Core 2* is consuming already fetch data from MAPLE queue; For prefetching, *Core 3* is using MAPLE as a prefetching engine, scheduling in advance a series of indirect accesses of the shape A[B[i]]; *Core 3* can thus fetch cache-averse patterns using MAPLE, and fetch regular patterns using the memory hierarchy. MAPLE operations can fetch directly from DRAM or from the LLC, as desired.

implementations, such as precise per-core area overheads or engineering effort needed to verify such core modifications.

Our Approach: This paper introduces a Memory Access Parallel-Load Engine (MAPLE), the world's first taped-out NoC-connected hardware that mitigates memory latency and improves performance without requiring processor tile or memory hierarchy modifications. In this paper, we implement, verify, and evaluate MAPLE's RTL, integrated into an open-source manycore framework through the NoC in a scalable, tiled, manner. Figure 1 highlights two scenarios that leverage MAPLE's specialization for memory latency tolerance and timely supply of data to processing units. MAPLE supports decoupling and prefetching techniques through its API. These are not custom ISA instructions but regular load and store instructions from user mode to read and write to a MAPLE instance through simple Memory-Mapped IO (MMIO) (see Section 3).

MAPLE offers a flexible programming model that extends far beyond scheduling a task to an engine that subsequently raises an interrupt upon completion (e.g. DMA engines). Utilizing MAPLE's hardware queues enables decoupling of data-produce and compute operations for latency tolerance. Previously, this fine-grained supply capability has only been supported through new ISA instructions and deep microarchitectural changes [22, 49], which made it difficult to adopt in practice.

Off-the-shelf cores can produce (store) data into MAPLE, and consume (load) from it as if they were interacting with a software queue, but with the advantage that MAPLE can transform the data in between. Figure 1 shows how MAPLE can be invoked to fetch irregular or IMAs and place the data into its FIFO queues for cores or accelerators to consume from them and perform dense computation.

MAPLE's scratchpad offers hardware queues implemented as circular FIFOs. MAPLE performs hundreds of long-latency IMAs in parallel, utilizing many slots in the FIFO queues, whose indices are used to reorder memory responses. This provides memory level parallelism (MLP) without the area overhead of IMA-dedicated hardware on every core.

Our main innovation is the exploitation of the software optimizations of decoupling and prefetching, while leveraging **specialized memory-access hardware, without modifying the core, ISA, or memory hierarchy**, demonstrated with a real implementation. This work enables MLP in systems with area-efficient cores (e.g. with small instruction windows or in-order execution), where software-only approaches are ineffective. MAPLE provides hardware assistance through an API and does not require modifications of processor cores. Our API and hardware-software co-design are compliant with Virtual Memory (VM) and SMP Linux, and support scaling the number of MAPLE instances, as done in our chip tapeout. Each MAPLE is individually protected through the core-level, standard, virtual memory protection.

MAPLE's API can be targeted by automated compiler passes in which the original program is transformed through LLVM [30] passes to offload IMAs to MAPLE. Alternatively, the API could be targeted from the backend of a Domain-Specific Language (DSL), e.g. GraphIt [67] or TACO [28], to overlap memory latency with computation.

Unlike much of the prior work shown in Table 1, MAPLE can be adopted in practice with little engineering effort. We demonstrate this with its integration into an open-source manycore SoC (OpenPiton+Ariane) [6] on its own tile. We measured MAPLE's effectiveness by evaluating prominent latency-bound workloads on an FPGA prototype.

Our main technical contributions are:

- A HW-SW co-design that mitigates long-latency memory accesses through scalable specialized units that are integrated into an SoC without core or memory hierarchy modifications.
- An API to program MAPLE units to timely supply data to cores for various access patterns, which can be automatically used by existing compiler passes.
- A full RTL implementation of MAPLE that supports virtual memory and is programmed from SMP Linux. This implementation is formally verified and it is made open-source with the publication of this paper.

We evaluate MAPLE and demonstrate that:

- Latency tolerance is possible without core changes; geomean speedups of 1.82× and 1.72× over prior work in hardware prefetching and decoupling, for latency-bound workloads widely used for sparse algebra and graphs analytics.
- SW techniques alone are ineffective to mitigate IMAs with slim cores; geomean speedup of 2.27× on FPGA emulation using MAPLE over software decoupling.
- MAPLE is easy to integrate and area efficienct; over 6-stage in-order RISC-V cores, MAPLE incurs just 1.1% area overhead for each of the cores it can supply.

Table 1: Classification of the hardware-assisted prior work on IMA latency mitigation, based on the key features that make the adoption of a hardware technique practical for SoCs.

Amenable for / Proposed Technique	Unmodif. Cores	Unmodif. ISA	Simple Cores	HW-SW Co-design
HW DAE [21, 36, 49]	✗	✗	✓	✓
DeSC/MTDCAE[22, 55]	✗	✗	✓	✓
SW Pre-execution [35]	✗	✗	✗	✓
Triggered inst.[43]	✗	✗	✓	✓
Slipstream [52, 54]	✗	✓	✓	✗
HW Prefetching[9]	✗	✓	✓	✗
Graph Pref, IMP.[1, 62]	✗	✓	✓	✗
Programmable Pref. [3]	✗	✗	✓	✓
DSWP [45]	✗	✗	✗	✓
Outrider [15]	✗	✗	✗	✓
Clairvoyance [58]	✓	✓	✗	✗
SWOOP [59]	✗	✓	✓	✓
MAD [24]	✗	✓	✓	✓
Pipette [41]	✗	✗	✗	✓
Prodigy [56]	✗	✓	✓	✓
MAPLE	✓	✓	✓	✓

2 BACKGROUND AND MOTIVATION

Over the last 30 years, many works have proposed techniques for memory latency tolerance. With the increasing importance of graph analytic and sparse neural network (SNN) applications, recent techniques have focused on mitigating the latency of IMAs. These can be coarsely divided into prefetching-based [1–3, 26, 62], streaming multi-core / multi thread [41, 52, 57], and decouple access-execute (DAE) [15, 22, 45, 49]. We accommodate these techniques in software, with the transparent usage of our out-of-the-core specialized hardware to increase performance.

This paper attempts to leverage the range of techniques proposed in this field, both in hardware and software. It identifies four **key limitations to be overcome** to democratize the access to their benefits in modern heterogeneous systems: **(1)** Prior hardware techniques modify the core microarchitecture, sometimes even reducing its generality. Adopting such techniques also increases the verification burden of already overloaded hardware designers [20]. This is exacerbated in the context of SoC generator frameworks [5, 8, 13, 60], where modifications to third-party cores' RTL can be very challenging and limit their reusability. **(2)** Some modern software techniques assume special capabilities from the core, like OoO or SMT. This limits their usage, e.g. area-constrained environments tend to use simple in-order cores. **(3)** Techniques that rely on ISA extensions [23] or ISA-specific instructions have limited applicability and portability problems, especially in the context of heterogeneous-ISA architectures [7, 34]. **(4)** Hardware-only techniques like Slipstream [52, 54] or hardware prefetching [1, 26] often require costly structures for book-keeping, detection, and prediction. Data movement is decisive to meet performance goals, and often these patterns are known in software [10]. Leveraging program knowledge, either extracted by a compiler pass or explicitly written (in the backend of a DSL) is key to delivering high performance at a low area and complexity cost.

Table 1 classifies prior software and hardware approaches in the extensive literature on latency tolerance of IMAs, based on the four identified features for a technique to achieve ease of adoption, software programmability, and performance in low-power, area-efficient systems.

Decoupled Access/Execute (DAE): The DAE [49] paradigm was introduced decades ago to overlap memory accesses and computation without relying either on out-of-order execution or on prefetching unpredictable access patterns. DAE slices a program into two parallel threads, the Access thread handles memory access and address computation, and the Execute thread does computations. Ideally, the Access runs ahead of the Execute by issuing memory requests and enqueuing their data. In the meantime, the Execute consumes the data from the communication queue to perform value computations. If the access slice can run ahead of the execute slice and produce all of the data required for computation, it can act as a non-speculative *perfect prefetcher*. Among other decoupling proposals [15, 36, 55], DeSC [22] builds upon DAE, introducing compiler and hardware optimizations, so that loads with no further dependencies on the Access side (whose values are used exclusively by Execute) can be loaded in a side structure of the Access core without stalling the pipeline. The biggest drawback of DAE, DeSC, and all the prior-art in hardware decoupling is that it requires specific hardware changes for Access and Execute cores, limiting their usage to those roles.

Our hardware mechanism is amenable to the decoupling programming model. MAPLE provides a software API for decoupling *without modifying the core altogether*.

In our case, any number of cores can be programmed to behave as Access or Execute at runtime. MAPLE, as a memory access engine, can handle many data loads in parallel by utilizing hardware queues to track data for completed requests. This prevents stalls in the Access thread if it is not capable of hiding latency (e.g. short instruction window).

An important and substantial accomplishment of MAPLE is its ability to support DeSC-style decoupling, but with off-the-shelf cores. This increases the programmability of the system to allow other latency-tolerance techniques while providing comparable performance (see Section 5).

Figure 2 showcases how our hardware-software co-design provides the API programmability of software decoupling while being assisted by specialized hardware, to achieve MLP even with simple cores. Since the Access thread is running on a core with a small instruction window, shared-memory decoupling loses runahead due to long-latency stalls of IMAs, and so the Execute thread stalls waiting for the data to be produced. With MAPLE, the Access thread only produces IMA pointers, which MAPLE will load asynchronously to the core—in a highly parallel manner—and supply data to the Execute in time. The performance gain of MAPLE for decoupling is demonstrated in Section 5.1 against software and hardware decoupling approaches.

Prefetching IMAs: This paradigm includes changes in hardware and/or software. Recent hardware proposals [1, 26, 40, 62] have

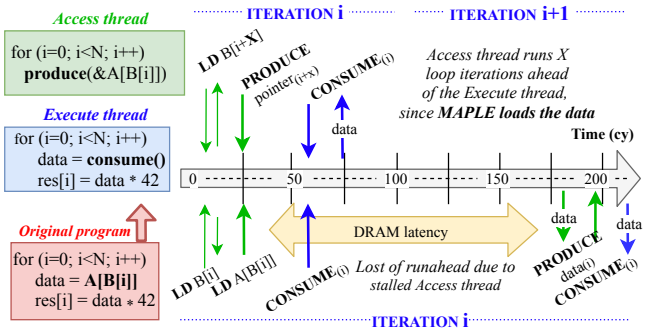

Figure 2: Memory transactions timeline of a decoupled program running on a thin core baseline. The original program (in red) has been sliced into Access (green) and Execute (blue) threads, using a software API for decoupling. The figure shows two executions targeting our decoupling API: using MAPLE's implementation (above the timeline), and using a shared-memory implementation (below).

the drawback of core-microarchitecture modifications, and the often large and per-core area overhead of the structures needed to predict access. Software techniques are a tempting proposition since they keep the core untouched and can leverage compiler knowledge. However, software prefetching incurs overheads due to code-bloating—up to 8.5× the instruction count in innerloops [2]. Prefetching might thrash the L1 too with large blocks or untimely data. To overcome these limitations, we use MAPLE as a programmable prefetcher too. State-of-the-art software prefetching techniques can use our API to issue prefetch commands to MAPLE, which can also decide the granularity and where to place the loaded data (into MAPLE queues or LLC). Moreover, MAPLE has specialized logic for IMAs which occur in loops, avoiding the extra instructions needed for address calculation of prefetches. **This mechanism provides the advantages of software techniques, with the enhanced performance brought by specialized hardware for memory accesses.**

Core Models: Prior art has already characterized that OoO without specialized hardware for memory accesses is not effective for our domain [37, 41, 56]. Because MAPLE units are effective for irregular, long-latency data (through decoupling or prefetching), this allows the use of simpler and more area-efficient cores. Since cores either consume from MAPLE or load regular-pattern data from the cache, these cores do not require expensive reordering units (in OoO) nor core-coupled IMA prefetchers. This pairing of slim cores + MAPLE saves per-core area, which allows for larger core counts and higher parallelism. We have seen the trend of manycores made out of slim cores recently in academia [6, 17, 65] and industry, e.g. Cerebras Systems [33] and Esperanto [19].

3 THE HARDWARE-SOFTWARE APPROACH

Our co-design offers a software interface to leverage MAPLE hardware specialization. Its communication mechanism is amenable for software pipelining and its programming model is easily extensible to incorporate domain-specific access patterns or more memory operations, e.g. data structure reshaping or Read-Modify-Write atomic operations. In the scope of this paper, we showcase how our co-design accommodates both software prefetching and decoupling optimizations, with enhanced performance due to specialized hardware assistance.

MAPLE can achieve speedups similar to that of latency tolerant DAE architectures, without requiring modifications to cores to designate them as Access or Execute. Instead, the DAE programming model is supported via the API. When using decoupling API operations, MAPLE provides the data communication queue, where data for memory requests from the Access thread are enqueued in order to serve the Execute thread. This allows us to offer the latency tolerance of DAE through hardware that is outside the core, unlike many prior hardware DAE approaches, which significantly modify the cores to support decoupling.

Additionally, MAPLE's connection to the interconnection network eases its scalable integration, where possibly hundreds of units could be connected to the SoC, each one supporting several queues. The concept of queues in the API is a software abstraction detached from the hardware queues. Internally, the API implementation can map hardware queues across multiple MAPLE instances, if needed. A thread can communicate with any MAPLE instance from user mode by having the OS map MAPLE's associated page (address range) into virtual memory, through Memory-Mapped IO (MMIO). This provides access protection and transparent allocation.

Sections 3.1- 3.3 provide examples of how the API can be used for different memory optimizations. Section 3.4 explains the details of MAPLE's implementation and how its hardware-software mechanism is fully compliant with virtual memory and requires no ISA-dependent instructions.

3.1 Using MAPLE for Decoupled Programs

The following list presents the API operations that emulate DAE techniques. `PRODUCE_PTR` utilizes MAPLE to load data and thus reduce the Access thread burden, especially on accesses with poor cache locality.

- **INIT**(*queues*): Initializes the queues for a program.
- **OPEN/CLOSE**(*id*): Opens exclusive communication with a queue, or closes such a connection.
- **PRODUCE** (*id, data*): Pushes data into a queue.
- **CONSUME** (*id*): Pops data from a queue.
- **PRODUCE_PTR** (*id, pointer*): pushes (stores) a pointer into MAPLE, which will fetch its data from memory and write the response into a queue in program order.

Besides these main operations, the API also contains functions to collect performance counters and debugging.

Figure 2 shows an example of a decoupled code targeting MAPLE, and the depiction of its runtime memory transactions. Figure 3 now shows the hardware components of MAPLE that are involved in this program and their interactions with the rest of the system. MAPLE is connected to the Network-on-Chip (NoC) through protocol decoders and encoders, and thus it can receive/send operations from/to the cores and make requests to DRAM and/or to the LLC. MAPLE also manages hardware queues for data communication between threads, implemented as circular FIFOs, using its scratchpad.

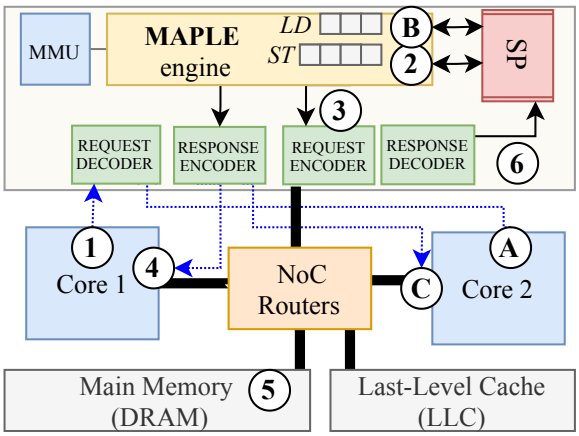

Figure 3: A high-level overview of MAPLE components including the scratchpad (SP) storage that queues are sharing. Numbers represent the steps of a pointer-produce operation and the letters the steps of data-consume.

Figure 3 depicts the software-hardware timeline of a pointer-produce (steps 1-6) and consume operation (steps A-C).

The Produce path works as follows: (1) It starts by doing a store instruction where the stored data is the pointer to fetch. This store is targeted to an address composed of MAPLE's instance base address, queue ID, and operation code; (2) The decoder identifies the operation as a pointer-produce, and routes it to the produce pipeline where it will reserve an entry in the corresponding queue; (3) The pointer (virtual address) is first translated into a physical address in MAPLE's MMU, and the data associated with that address is requested from DRAM, using as the transaction ID the index of the allocated entry in the queue; (4) The initial store request is acknowledged to the Access thread which considers the produce as finished and retires the store instruction; (5) The memory request reaches DRAM which responds to MAPLE; (6) The response is decoded and stored in the corresponding queue entry.

Consumes occur later in time than the data production provided that the Access thread has enough runahead.This should be the norm when using MAPLE, since the Access is not stalled and the hardware queues are big enough to hold the data fetched in advance.

The Consume path works as follows: (A) The execute thread generates a consume operation—implemented in the API as a load request to MAPLE. Once the load reaches MAPLE, it is decoded and routed to the consume pipeline; (B) If the queue specified on the request parameters is not empty, it would pop the entry in the head of the queue and return it as a response to the load instruction; (C) The response reaches the core that is running the Execute thread and the consume operation finishes.

Using MAPLE for decoupling brings software flexibility over the original hardware DAE approach or state-of-the-art DeSC architecture. In our approach, Access or Execute are conceptual "roles" taken by software threads rather than a hardwired core type, and they can be determined at runtime. This enables dynamic reconfigurability for applications with different data supply and computation demands. Some might benefit from having multiple Execute threads being supplied from the same Access thread, generating an *asymmetric decoupling* relation. This is possible with MAPLE (see Section 3.6), unlike with previous architectures for DAE, which only scale in pairs of Access-Execute cores [22, 49, 55].

```
for ( i = 0; i < N; i++ ){
    // D is distance in number of iterations
    LIMA (A, B, ptr[i+D], ptr[i+1+D]);
    for ( j = ptr[i];  j < ptr[i+1];  j++ ){
        res[j] = C[j] * A[B[j]];
    }
}
```

Figure 4: Code example using MAPLE for prefetches of tight Loops of IMAs (LIMA). This avoids the code bloat problem of software prefetching. MAPLE can issue prefetches that place the data in the LLC (speculative, shown here) or into MAPLE queues (non-speculative). The IMA is marked in red and the cache-friendly access is marked in green.

3.2 Using MAPLE for Prefetching

MAPLE's API can also be used for software prefetching. Non-speculative prefetching can leverage the aforementioned queue management functions and `PRODUCE_PTR` to place all the prefetched data into a queue within MAPLE. This is especially desirable in the context of IMAs like `A[B[i]]`. Placing the data of irregular, cache-averse accesses into MAPLE has a two-fold advantage over placing it in the memory hierarchy: it prevents data from being replaced if fetched too early with respect to its usage, and it avoids thrashing the L1 cache with low-reuse data. Additionally, MAPLE can prefetch into the shared LLC to support speculative prefetching (`PREFETCH(ptr)`).

Software prefetching of IMAs within inner-loops incurs an instruction overhead to calculate the address of the target prefetch and other book-keeping [2]. To remove that overhead, MAPLE can prefetch Loops of Indirect Memory Accesses (LIMA). This is targeted through API operations:

- **LIMA** (*A,B,begin,end*): It speculatively prefetches in hardware `A[B[i]]` (or `B[i]` if A is 0) in the range between *begin* and *end*, into the shared-cache.
- **LIMA_PRODUCE** (*qid,A,B,begin,end*): LIMA version for non-speculative prefetching, where the data is produced into MAPLE queues, to later be consumed.
- **PREFETCH** (*pointer*): It speculative prefetches a pointer into the Last-level Cache (LLC).

Figure 4 shows a code example of injecting LIMA speculative prefetching. A single software operation provides prefetches for a whole loop of accesses (details in Section 3.4).

3.3 Targeting MAPLE Automatically

Although one could use our API directly, we nevertheless believe programmers need not directly code data movement. Instead, compiler passes or domain-specific languages such as TACO [28] (sparse algebra) or GraphIt [67] (graph analytics) could use the API, as they have knowledge about data structures and coherence. Recent automatic compiler techniques already target software prefetching [2]

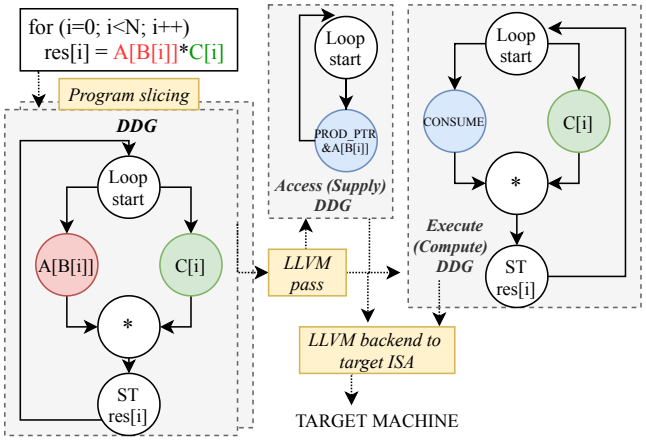

Figure 5: Simplified compiler flow for decoupling. First, the program is sliced into Access and Execute, then a LLVM pass converts the IMA (in red) into `PRODUCE_PTR` and `CONSUME` API operations targeting MAPLE. Finally, both slices are compiled down to assembly.

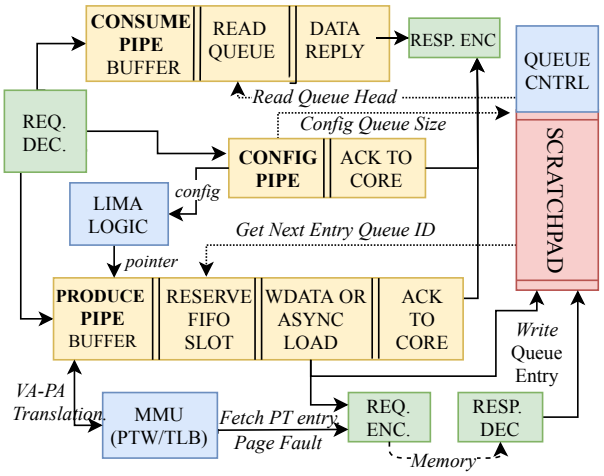

Figure 6: Microarchitecture of MAPLE: designed to maximize MLP and area efficiency. The pipelines allow several concurrent operations, one per pipeline stage. The design has separate pipelines to avoid deadlock situations (formally verified). The LIMA logic loads chunks of adjacent data (`B[i]`) and performs pointer indirection for each word by internally feeding pointers (`A[B[i]]`) into the Produce path.

and slice decoupled programs [22]. Therefore, they could be adapted to target API operations instead of ISA-specific instructions.

Figure 5 shows our adaptation of the compilation flow of DeSC [22]. This flow slices the program into Access and Execute threads; loads are transformed into `PRODUCE` and `CONSUME` operations. After the program slicing, some loads no longer have dependencies on the Access code (only on the Execute), and so the Access can produce the pointer for MAPLE to load, `PRODUCE_PTR`. We evaluate using this LLVM-based [30] automatic compiler pass in Section 5.2, showing that by simply utilizing established compiler techniques, MAPLE can be leveraged to yield significant performance improvements.

Automatic compiler techniques for prefetching could potentially target the LIMA operation, thus reducing the instruction overhead of software prefetching IMAs in tight-inner loops, but this is out of the scope of this paper.

3.4 MAPLE Hardware Implementation

Figure 6 presents the microarchitecture of MAPLE. There we can observe the breakdown of the aforementioned engine of MAPLE into three pipelines and a queue controller.

The *Configuration pipeline* is used to create logical queues and bind them to software threads at runtime. These queues are implemented as circular FIFOs using a local scratchpad. Depending on the program's needs, one can configure to have fewer, larger, queues, or many but smaller. There is an upper limit on the number of queues per MAPLE unit, which is set as an RTL parameter at tape-out, along with the scratchpad size. This pipeline receives read operations when the configuration requires a response (e.g. queue binding), and write operations when the configuration needs to specify a payload (e.g. for the LIMA unit). This pipeline is non-blocking as it needs to be available for software configuration of the MMU and debug operations.

The *Consume pipeline* is solely used for cores to read data from the queues.

The *Produce pipeline* receives store instruction from the cores where the payload contains either data or a pointer to fetch. Produces are processed in several stages: First, the transaction is buffered. In the case of a pointer, the virtual address is translated in the MMU; second, a slot in the queue is reserved; third, either the data is written into the reserved slot (data-produce) or the memory request is issued to DRAM (pointer-produce) using as transaction ID the queue slot index. Memory responses come in any order. We ensure data is written in program order with the transaction ID.

The reason to have separate pipelines is to avoid deadlocks. When a specific queue is full, the produce operation is buffered (no overflow) in the first stage until an entry is consumed. Meanwhile, operations to other queues can proceed without stalls. Consumes work similarly, a load into an empty queue is buffered (no polling) until new data is available to be returned to the core. Each pipeline has a final stage to respond to the issuing core. This design was verified with industry-level formal tools (Section 3.9).

LIMA operations fetch `A[B[i]]` for a given range of `i`. Once the base pointers for arrays A and B are configured (virtual addresses), LIMA performs TLB translation and fetches array B in chunks of 64B that are stored in the scratchpad. As soon as the chunks start arriving, LIMA iterates over them word by word utilizing an offset into array A to calculate the final address. Finally, depending on whether the prefetch is speculative or non-speculative, it inserts into the Produce pipe the equivalence of a pointer-produce or a prefetch operation. Because MAPLE is ISA-agnostic, the prefetch operations are not using ISA prefetch instructions. Instead, it sends a network request to the shared cache, similar to how a private cache would do.

3.5 Virtual Memory Support

When a core requests a queue through the API, the OS maps a free MAPLE instance into a page (MMIO). Thus, the core performs address translation to load or store into the MAPLE address-space, accessing that MAPLE context (control registers) in a protected manner. Since this is a single page, the translation hits in the TLB with no overhead. Because the data that is delivered to MAPLE can be a pointer, i.e. a virtual address, it needs translation. MAPLE fully supports virtual memory through its local MMU and TLB, to be able to access any regularly allocated memory. MAPLE's TLB is fully associative and has 16 entries, the same as the cores' TLB. Because IMAs are irregular and often span different pages, TLB misses add latency to IMAs. The total latency is mitigated by MAPLE with runahead execution and memory parallelism.

Upon a TLB miss, MAPLE's hardware Page Table Walker (PTW) fetches the corresponding entry from the memory hierarchy. If the PTW encounters a page fault (e.g. if the page is invalid), an interrupt is raised, and the kernel invokes the MAPLE driver. This driver reads the virtual address that caused the page fault (using the Configuration pipeline) and maps it into the page table if valid access. The device driver implements Linux's callback function for shootdowns, which are communicated to the MAPLE-MMU to prevent stale entries.

3.6 Communicating with MAPLE units

Portable: General-purpose cores can communicate with MAPLE from user mode through MMIO. This allows operations like Produce and Consume to, under the hood, use existing store and load instructions, respectively. These are synchronous (not polling) with MAPLE, i.e. memory instructions return once MAPLE acknowledges them. The round-trip path is depicted and latency-characterized in Figure 14.

Scalable: Since many MAPLE instances can co-exist in an SoC (e.g. a tiled architecture), each one is accessed via a different physical page. We leverage virtual memory translation to provide process-exclusive access to MAPLE's hardware resources and provide data protection. This also allows a process to decide at runtime which MAPLE unit to target. As we introduced before, previous approaches [22, 23, 49] do not offer this software programmability of decoupling-hardware resources, as these are tightly connected to specific cores.

Extensible: The fact that each unit's control registers are mapped to a page allows MAPLE to re-purpose the index of a word within the page to distinguish operation codes based on bits 3 to 8. This gives the API up to 128 operation codes, i.e. 64 for loads and 64 for stores so that more operations can be included.

Core-Agnostic: The only capability MAPLE needs from a core is having load and store instructions. Thus, it can communicate with any off-the-shelf core and is not limited to non-speculative in-order cores. Although not included in our chip tapeout, MAPLE could handle speculative queue/dequeues from cores using transaction IDs encoded in the lower address bits. A chip for different workloads, where OoO cores are desired, could also integrate MAPLE units to speed up irregular accesses.

Efficient: MAPLE has full access to the memory hierarchy, thus, it can do cache-coherent loads from the LLC or non-coherent loads directly to main memory (determined by the decoded operation-code). There are advantages and limitations inherent to the idea of offloading memory operations into a specialized unit. MAPLE behaves effectively as a Private Local Memory (PLM) so that the data inside the scratchpad has no coherence guarantees after it is fetched. The compiler technique or DSL using the API should make sure that the arrays loaded by MAPLE have no further writes to them. This condition holds for the irregularly accessed array of most of the graph algorithms studied, since updates often occur only after an epoch barrier. Leveraging conditions known at the software level allows MAPLE to use highly parallel and efficient hardware.

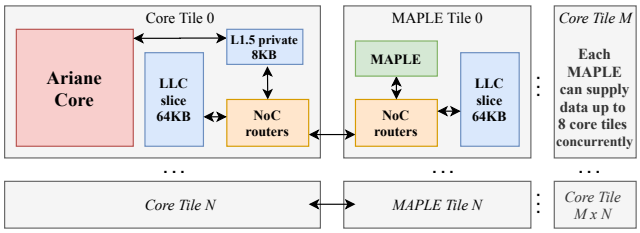

Figure 7: Integration of MAPLE into an OpenPiton tile. MAPLE only needs to be connected to the NoC through its parameterizable encoders and decoders.

We envision MAPLE to be an easy-to-adopt and scalable resource to include in an SoC. MAPLE speeds up workloads that do not leverage traditional cache locality and benefit from a programmable unit accessible from the memory hierarchy.

3.7 MAPLE Integration via NoC

A key feature of MAPLE is that it can be adopted by a system without modifying existing hardware. It can simply be accessed via the on-chip interconnection network (NoC). This procedure has been followed for the 2D P-Mesh protocol of OpenPiton [8], which is an open-source, tile-based SoC framework. Figure 7 depicts this integration of MAPLE on its own tile via the NoC routers. This integration has been evaluated on FPGA (Section 4.2) and the results are reported in Section 5.1.

Ease of adoption: The integration of MAPLE with OpenPiton took around a hundred Verilog RTL lines of code (LoC), which demonstrates that it is easy to adopt. This contrasts with the 5K LoC of MAPLE itself. This demonstrates the advantage of integrating it as a reusable IP block versus building it from scratch. Moreover, the integration does not require details about the underlying system aside from the communication protocol. It is agnostic to ISA and CPU internals.

3.8 Reusing MAPLE in SoCs

Deep microarchitecture changes are hard to take into practice because of the verification burden. Hardware designers are spending about half their time doing verification [20], and trends [47] indicate that the number and diversity of IP blocks per SoC can exacerbate this burden. Several frameworks have emerged to make SoC development agile and multicore [5, 8, 13, 60], by connecting highly parameterized IP blocks to form a complete SoC design. Reusing

IP alleviates the verification burden so that engineers can focus on system-level requirements [4]. However, SoC generator frameworks do not have a reusable hardware solution to the memory latency bottleneck yet. Because MAPLE is agnostic to the ISA and core model, it could even be included in SoC frameworks with heterogeneous cores [4, 8] and hybrid ISAs [7, 34].

3.9 Formal Verification of MAPLE

MAPLE saves verification effort over the prior work in latency tolerance techniques. It is so because the verification burden is shifted from the integration process to the decoupled unit. We invested significant time to verify MAPLE's correctness at the unit-level to remain agnostic of the rest of the system and ease integration. This makes MAPLE reusable without the verification burden of a tightly-coupled integration.

The verification was conducted using industry-standard SystemVerilog Assertions (SVA) [25] and JasperGold [12] and assisting tools [42]. We followed a verification-first approach to save late-stage debugging time and increase the confidence in creating a verifiably correct design. We exhaustively tested the pipelines and MMU interactions.

As a result of this lengthy process, we deliver a verified RTL design for functional correctness and liveness. The quality metrics provided by JasperGold give confidence in the goodness of the assertions—they cover more than 99% of the MAPLE's RTL.

After integrating MAPLE with the final system, we included the SVA properties on the system-level simulation testbench. MAPLE held correct, but our effort uncovered two bugs in the open-source core interacting with MAPLE, which were communicated to the maintainers. The formal testbench will also be included in the open-sourcing of this project

4 EVALUATION METHODOLOGY

This section first describes four widely-used data-analytic benchmarks that exhibit memory latency bottlenecks due to IMAs. Second, it provides details of the SoC prototype emulated on FPGA. At the end, it describes the methodology employed for the evaluation over the prior techniques and sensitivity studies.

4.1 Applications for Data-analytics

Memory latency bottlenecks of Graph and Sparse Algebra applications have been characterized several times in the last couple of years [22, 41, 56] with over 60-70% of the runtime dedicated to memory stalls.

Sparse matrices often contain few non-zero elements and therefore are stored in compact representations. Two of the most popular representations are Compressed Sparse Row (CSR) and Compressed Sparse Column (CSC). They both efficiently represent a sparse matrix using three one-dimensional arrays to store the number of non-zero elements of a row (or column), indices of non-zero matrix elements within that row (or column), and the non-zero matrix elements. Meanwhile, dense matrices are simply stored as one-dimensional arrays, similar to the data arrays in the CSR and CSC formats. Because they are dense, the indices of the elements can be determined by knowing the number of rows and columns and do not require information about where non-zero elements are located.

Sparse Dense Hadamard Product (SDHP): Performs an elementwise operation, e.g. multiplication, between a sparse and a dense matrix. Because the operation is performed elementwise, the dense matrix is sparsely sampled based on the locations of non-zero elements in the sparse matrix. This results in irregular accesses to the dense matrix, as they are not predictable and therefore not amenable to the cache locality. By decoupling the kernel so that the Access can fetch the irregular memory accesses before the Execute core needs their data, this performance bottleneck can be alleviated.

Sparse Matrix-Matrix Multiplication (SPMM): Performs a matrix multiplication between two sparse matrices `A` and `B` in a layer-wise fashion [39] to train a sparse deep neural network. This kernel is parallelized in the columns of `B`, while intermediate results are stored in a dense, temporary matrix.

Sparse Matrix-Vector Multiplication (SPMV): Performs matrix multiplication between a sparse matrix and a dense vector. Similar to SDHP, the dense vector is sparsely sampled according to the non-zero elements of the sparse matrix, providing an improvement opportunity for decoupling.

Breadth First Search (BFS): Determines the distance (number of hops) from a given root vertex in a graph to all other vertices. The traversal starts at the root and in each iteration, examines all vertices in a layer-wise fashion to find neighbors that have not been visited and require an update. Accessing neighbor data requires IMAs.

Datasets: We evaluate these kernels using real-world networks and synthetic datasets. SDHP uses matrices from SuiteSparse [18] and a Kronecker network [32], BFS operates on Wikipedia, YouTube, and LiveJournal graphs, while SPMM and SPMV use synthetic matrices from *riscv-tests* [46].

4.2 FPGA Emulated SoC System

As described in Section 3.7, we have integrated MAPLE's RTL within the OpenPiton framework [6], to characterize its advantages in a real system. OpenPiton is an open-source tile-based manycore architecture, which supports multiple ISAs. We use RISC-V Ariane [63] cores to demonstrate how latency tolerance can be achieved even in simple, non-speculative, in-order cores which are commonly used in area- and power-constrained environments.

Table 2 presents the SoC details as well as the parameters used for MAPLE and the FPGA used for the prototype.

This evaluation demonstrates that our RTL implementation works on a potential SoC, emulated on FPGA, running applications on top of SMP Linux (version v5.6-rc4). We evaluate the applications and datasets described above, running single and multithreaded versions with OpenMP [16] parallelization. The FPGA evaluation highlights the performance speedups obtained by doing prefetching and decoupling through MAPLE, over the baseline of do-all parallelism.

The evaluation compares the same decoupled program with the API, using (a) a shared-memory implementation of decoupling; and (b) an implementation targeting MAPLE to characterize the benefits of our hardware-software co-design. Then, the evaluation compares the latest prefetching techniques over using LIMA to fetch loops of IMAs. For a fair comparison, prefetches are inserted in the code at the best location known to the programmer.

Table 2: SoC configuration for the full-system evaluation booting Linux v5.6-rc4, including MAPLE (top); FPGA board specification and prototype utilization (bottom).

SoC configuration	OpenPiton + MAPLE
MAPLE Instances / Scratchpad Size	1 / 1KB
Core Count / Threads per core	2 / 1
Core Type	RISCV64 Ariane 6-Stage In-O
L1D+L1I per core / Latency	8KB+16KB 4-way / 2-cycle
L2-size per tile (shared) / Latency	64KB 8-way / 30-cycle
FPGA board	**Virtex 7**
Model	XC7VX485T-2FFG1761C
Board	Xilinx VC707
Core Frequency	60MHz
CLB LUTs Utilized	216831(69.9%)
DRAM Device / Size / Latency	DDR3 / 1GB / 300-cycle

Table 3: Core and memory parameters of the simulated system, to compare MAPLE over the prior work.

System Model Parameter	Values
Core Count / Threads per core	2 / 1
Instruction Window / ROB Size	1 / 1, In-Order
L1D (per core) / Latency	8KB / 4-way / 2-cycle
L2-size (shared) / Latency	64KB / 8-way / 30-cycle
DRAM Size / Bandwidth / Latency	4GB / 68GB/s / 300-cycle

The related work has not provided RTL implementations. Since implementing in RTL the related work that we wanted to compare [9, 22] would take months, even for people with years of industry experience, we compared against these in a simulator and compared against software techniques on FPGA emulation. The FPGA could only fit a MAPLE instance and two cores, so we ran with scaling threads on the simulator as well (Figure 13). The simulator-based evaluation of MAPLE over prior decoupling leverages the automatic compiler program-slicing seen in Section 3.3. However, this slicing was done manually for the FPGA runs, since this was not yet incorporated into our FPGA flow.

4.3 Evaluation Against Prior Work

In addition to our real-system evaluations, which demonstrate a significant improvement over the baseline, we evaluate MAPLE over the latest latency-mitigation approaches, including DeSC decoupling [22] and DROPLET hardware prefetching [9], via system simulation. To do this we leverage MosaicSim [38], a simulator for heterogeneous architectures and hardware-software co-designs, and model the communication queues used in MAPLE.

Table 3 shows the core model and memory hierarchy parameters of the simulated system. We tried to match the simulator model with the SoC configuration to prove the same premise, that MAPLE can provide latency tolerance even for single-issue in-order cores. This evaluation leverages DEC++ [51] compiler flow for automatic code transformation of MAPLE-decoupling and DeSC.

4.4 Sensitivity Parameters to Characterize

MAPLE has many interesting parameters worth evaluating, like the size of the queue connecting a pair of threads (determined at runtime). For decoupling, this queue must be big enough to allow the Access thread to run ahead and hide the memory latency so that the Execute thread does not stall waiting for data. However, the smaller the size, the more logical queues can share MAPLE scratchpad memory. This size is closely related to the round-trip latency between MAPLE and any given core (sets the throughput to/from the queue) along with the DRAM latency, since it determines the runahead that is necessary. To evaluate the impact of different queue sizes on the runahead between Access and Execute, we study the performance counters provided by MAPLE through debug operations when running on the FPGA.

It is important to characterize the round-trip latency between cores and MAPLE, since it determines the cost of a data consume. This latency depends on the memory hierarchy, the network, and the placement of MAPLE unit(s). We first break down and characterize this round-trip latency in the OpenPiton framework by analyzing waveforms of an RTL simulation. Then, we study this latency's impact on performance by varying it as a parameter in simulation.

5 RESULTS

This section first presents the performance results obtained from the FPGA evaluation of our SoC prototype: It compares the speedups obtained by decoupled programs (with and without using MAPLE) over the parallel version of the original program; and the performance of a program enhanced with prefetching over no prefetching. The prefetching version is both evaluated using MAPLE's API and prefetching instructions. Second, there is a comparison against prior hardware techniques, both for decoupling and hardware prefetching. Finally, it presents the conclusions from the sensitivity studies and the area analysis of MAPLE's RTL implementation.

5.1 FPGA Emulation of the SoC Prototype

Figure 8 compares the speedups achieved by decoupling Access and Execute threads using MAPLE's API and a shared-memory implementation, over traditional doall parallelism on 2-threads. The rightmost comparison shows the geomean speedup obtained across all applications. Using MAPLE achieves 1.51× speedup over doall and 2.27× over software-only decoupling. This demonstrates that decoupling is not performant by itself in area-constrained systems without MAPLE hardware support.

We also compare MAPLE against software prefetching with a baseline of no prefetching for single-thread execution. We evaluate MAPLE's *LIMA_PRODUCE* operation, which places the prefetched data into its hardware queues, from which the core consumes. Since IMAs have poor cache locality, it is better to consume them from MAPLE (as non-cacheable), and reserve the caches for regular data accesses that exploit locality.

Figure 9 compares the speedups of prefetching IMAs in hardware with MAPLE (using the LIMA operation), and conventional software prefetching. The geomean speedup is 1.73× over no prefetching and up to 2.4× for SPMV. In addition, using MAPLE achieves a

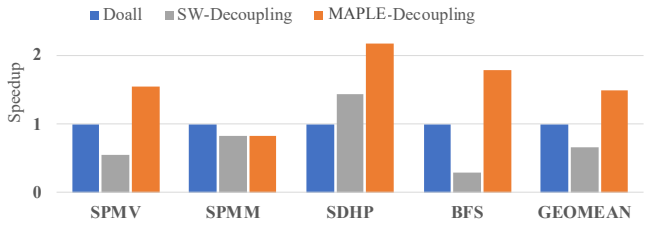

Figure 8: Speedups obtained with decoupling (1 Access and 1 Execute thread), normalized to 2-thread doall parallelism. It showcases that decoupling only in software is not effective on the in-order baseline without hardware support.

Figure 9: Speedups obtained for a single-thread doing non-speculative prefetching with MAPLE (using the LIMA operation) and conventional software prefetching, normalized to no prefetching. It shows that placing the IMA prefetches into MAPLE queues is desirable over prefetching into the L1.

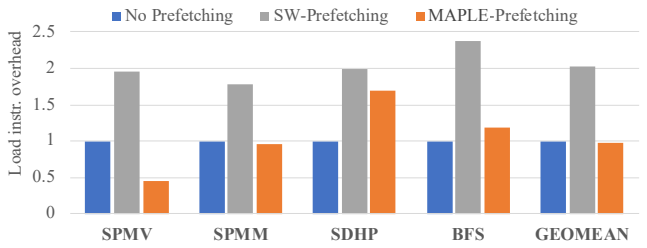

Figure 10: Normalized load-instruction overhead due to prefetching using the MAPLE's LIMA operation and software prefetching, normalized to no prefetching.

geomean speedup of 2.35× over software prefetching, showing the advantage of not bringing highly irregular data into the L1 cache.

Moreover, prefetching using MAPLE reduces the instruction overhead of software prefetching since IMAs in a whole tight inner-loop can be offloaded into MAPLE with a single LIMA operation.

Figure 10 presents the normalized overhead of load instructions due to prefetching relative to the baseline with no prefetching. Software prefetching doubles the number of loads, whereas MAPLE slightly reduces the total number of loads. The reduction occurs because the sparse IMAs are gathered inside MAPLE queues, and if the data type is a 32-bit word (as it happens in SPMV), the core loads two words at a time.

This evaluation has also collected hardware performance counters to measure the average latency of load instructions.

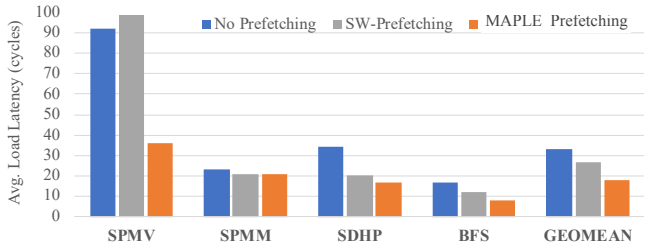

Figure 11: Average clock cycles of load instructions. It compares software prefetching and LIMA operation. It shows that MAPLE prefetches are timely.

Figure 11 shows that using MAPLE's LIMA operation for prefetching significantly decreases the average load latency to nearly half (1.85× geomean reduction), thus demonstrating its effectiveness to hide memory latency of cache-averse accesses. It is significantly more effective than doing software prefetching into the L1 cache, which suffers from cache thrashing due to the low spatial and temporal locality of IMAs. Moreover, consuming data from MAPLE queues avoids the premature replacement of prefetched data in caches. These advantages are shown clearly for SPMV. Although it is not characterized here, **LIMA operations can complement regular prefetching instructions, where MAPLE is targeted for IMAs while regular access patterns are prefetched natively.** Since the compiler can automatically detect which accesses are irregular [50], it could insert adequate prefetches.

5.2 Comparison against Prior Work

Figure 12 compares the runtime performance of MAPLE decoupling, DeSC [22] decoupling, and DROPLET [9] hardware prefetching, as well as that of traditional doall parallelism, for 2 threads. The speedup result for each application is the geomean of the speedups obtained across the datasets evaluated.

DeSC slicing is more restrictive than MAPLE decoupling, since DeSC's Execute (Compute) core does not have visibility into the memory hierarchy, and all data is passed to the Access (Supply) to be stored. This results in a loss of runahead for BFS, and thus DeSC performs poorly compared to MAPLE. Decoupling in general is not effective for the selected SPMM implementation, since the IMAs are Read-Modify-Writes and cannot be decoupled. Unlike DeSC, we do not propose a DAE architecture, but MAPLE supports decoupling; if the compiler pass for program slicing cannot find an IMA, it falls back to doall parallelism. In contrast, SPMV and SDHP kernels—well suited for decoupling—achieve high performance with DeSC. We pay a threshold latency for cores to communicate with MAPLE, which is slightly higher than the architecturally visible, tightly-coupled, DeSC queues. **MAPLE supports a flexible alternative to DeSC for decoupling**, which does not constrain the architecture. Despite no core modifications, MAPLE achieves at least 76% of DeSC 's performance for decoupling-friendly applications, and it presents overall better performance. It achieves a geomean speedup of 1.72× over DeSC and 1.82× over DROPLET hardware prefetching, and up to 3× (geomean 1.96×) over doall for BFS.

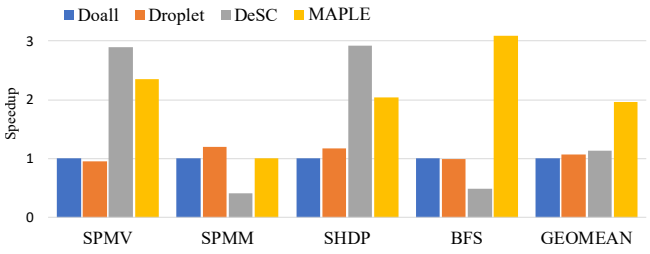

Figure 12: Speedup (y-axis) achieved with MAPLE, DeSC, and DROPLET over the baseline. Decoupling with MAPLE and DeSC use 1-Access and 1-Execute threads, while DROPLET and the baseline perform 2-thread doall.

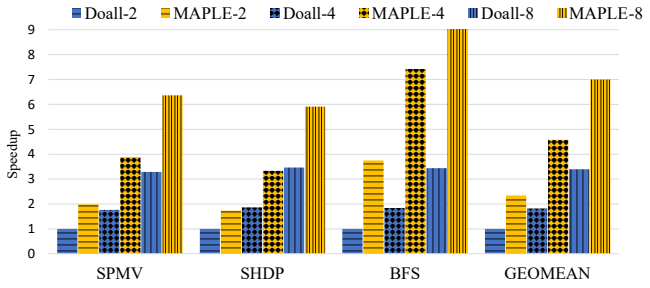

Figure 13: Speedup (y-axis) achieved with decoupling (threads are sharing a single MAPLE unit) over do-all parallelism, with scaling threads: 2, 4, and 8.

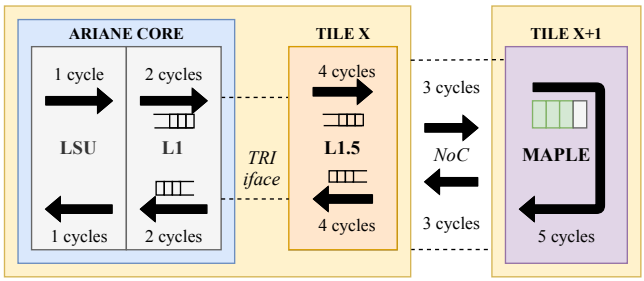

Figure 14: Step by step breakdown of the round trip latency of Core-to-MAPLE communication at the OpenPiton framework. Latency could be lower if L1 requests would not pass through the L1.5 cache. A lower communication latency would incur greater performance benefits (studied in Figure 15).

5.3 Conclusions about the Sensitivity Studies

Figure 13 shows that our co-design scales well with an increasing number of threads, maintaining the speedup achieved over doall parallelism when scaling to 4 and 8 threads sharing the same MAPLE unit for decoupling. More units can be employed for larger thread counts in a tiled manner, conforming to a scalable system, only limited ultimately by the chip IO.

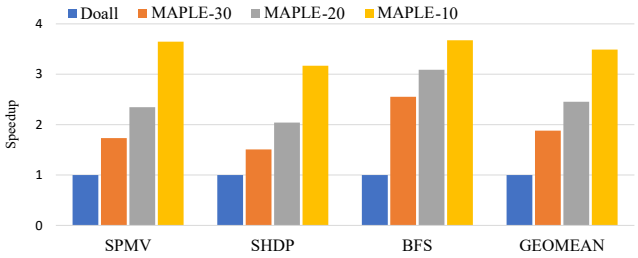

Figure 15: Speedup (y-axis) achieved with different core-to-MAPLE latency values, to study the impact of communication latency on performance.

When fetching data into MAPLE queues (decoupling or non-speculative prefetching), the long latency of IMAs is reduced to the consume round trip between the core and MAPLE. Figure 14 shows the characterization of that latency for the OpenPiton SoC. This latency is similar to the L2 access, 25 cycles plus a cycle per hop, and an order of magnitude smaller than DRAM.

In a manycore mesh scenario, MAPLE instances are often scattered across the X and Y tile axes so that MAPLE are near cores. As explained in Section 3.6, MAPLE instances are mapped into virtual memory, and a process could leverage the OS to minimize the distance between the running core and any available MAPLE instance, to minimize round-trip latency.

Besides evaluating the particular latency of the OpenPiton network, we characterize in Figure 15 how the performance changes with smaller and larger communication latency values. The number next to MAPLE represents the average round-trip latency between cores and MAPLE, in cycles. This demonstrates that speedups are greater with a lower NoC delay.

We studied the performance impact of different queue sizes and observed that performance remains stable while the queues can hold enough data to hide latency. Although it is not shown here, a queue of 32 entries—4 bytes each—was sufficient to provide runahead without losing performance, while 16 entries caused a 5-10% decrease. With 32 entries per queue, MAPLE can supply data for up to 8 cores with just 1KB of storage (256 entries).

5.4 Area analysis of the RTL Implementation

In our design, MAPLE synthesis including 8 circular queues sharing a 1KB scratchpad represents 1.1% of the area of the single-issue in-order Ariane [63, 64] cores it can supply, which are already very area-efficient. Thus, the overhead of MAPLE compared to more beefy cores would be negligible. MAPLE need not be per-core, and thus its area can be amortized over multiple cores that use it. In contrast, tightly-integrated prefetchers can increase logic delay, core area, and cycle latency

Some prefetcher designs claim a low storage overhead (<1KB), but their designs also contain FSMs, muxes, and combinational logic whose area is not accounted for in their bitcount-based (storage) estimates. MAPLE is the only technique implemented in RTL and taped out, to our knowledge. While area overheads for IMP [62], Prodigy [56] and other related works only count storage, MAPLE overhead is calculated from the 12nm synthesis of our chip tapeout.

6 ADDITIONAL RELATED WORK

Since *Decoupled Access-Execute (DAE)* was originally proposed by Smith [48], several hardware implementations have been proposed, where data communication occurs through architectural queues [21, 36, 61]. In these papers, DAE aims to hide memory latency as a simpler alternative to superscalar processors. Later work analyzes the problems that arise from work imbalance between Access and Execute [27] and loss of decoupling (LoD) due to control dependencies [11]. Other work has envisioned Access and Execute cores having multiple physical threads [44, 55], or even having both Access and Execute as physical threads in the same core [15]. DeSC [22] introduces compiler and hardware techniques to avoid LoD and large instruction windows, and a special buffer in the Access core to host early committed load instructions. Memory Access Dataflow (MAD) [24] introduces an engine optimized for dataflow computation that is integrated with cores or accelerators to execute memory-intensive portions of programs.

A limitation of these approaches is they require special ISA instructions to configure and use the communication queues. We overcome this problem by placing the queues within the MAPLE tiles—addressable through MMIO—allowing them to be shared among several cores in a manycore architecture. MAPLE is capable of providing fine-grain data supply with the same programming model as a native decoupling architecture.

Hardware prefetching has long been proposed to avoid cache misses in regular access patterns [26, 40], but traditionally does not work for IMAs. Recent proposals [1, 3, 62] achieve better performance in applications dominated by IMAs, e.g. graph analytics. Notably, Prodigy [56] introduces novel compiler techniques to further assist the hardware. However, these approaches still require modification of the core microarchitecture, which is a considerable engineering effort both in design and verification. Thus, software techniques for latency tolerance are a tempting proposition in terms of ease of adoption.

Other hardware techniques like Slipstream [52, 54] and Triggered Instructions [43], strive to separate data access and usage. Pipette [41] is a hardware-software co-design that aims to generalize decoupling to a stream of stages that a program can go through. However, the deep microarchitecture modifications of these techniques limit their adoption in practice.

Software latency tolerance often uses compiler knowledge to improve performance. *DSWP* [45] does automatic software pipelining without speculation by utilizing a hardware-aided inter-thread communication mechanism; Clairvoyance [58] proposes compiler code separation into Access-Execute phases, to leverage the wide execution engines present in OoO cores. However, these software techniques rely on expensive hardware structures (RoB and LSQ) to maintain large instruction windows. As an alternative to this, SWOOP [59] introduces compiler techniques, with the hardware assistance for context remapping—a novel form of register renaming—to enable dynamic separation of Access and Execute phases in the code. However, SWOOP requires microarchitectural changes of the core, while MAPLE works with off-the-shelf cores. Our design does not need the core to support large instruction windows since it can achieve memory-level-parallelism (MLP) in an area-efficient manner through MAPLE.

Helper threads avoid large instruction windows by using a secondary thread of execution to improve the performance of the main thread [35]. This thread is either programmer [14] or compiler generated [66]. *Software prefetching* has been shown effective for pointer indirection [2], aided by compiler techniques to automatically insert prefetches in the code. Helper threads and prefetching are sensitive to timeliness and can cause cache thrashing if not properly controlled, along with other problems like code-bloating already described in Section 2.

Many of the latency tolerance techniques mentioned here can co-exist or combine with MAPLE. For example, we envision leveraging existing compiler techniques to target its API [2, 29]. This paper combines the advantages of software techniques, i.e. leveraging program knowledge, and hardware specialization while remaining ISA-agnostic so it can be widely adopted and extended.

7 CONCLUSION

This paper has introduced a hardware-software co-design for latency tolerance that offers the best of both worlds: its flexible software interface enables MAPLE to be automatically targeted by compiler techniques for both prefetching and decoupling, and its specialized hardware does not need ISA extensions nor microarchitectural changes to the cores, which is key in today's open-source hardware renaissance.

We have demonstrated MAPLE gains on FPGA emulation by running sparse linear algebra and graph analytic kernels on SMP Linux. MAPLE provides significant performance improvements, 2.35× and 2.27×, over software-only techniques, and 1.82× and 1.72× geomean, over hardware prefetching and decoupling respectively. Moreover, MAPLE provides increased programmability and reusability over hardware-only approaches. The key for performance/area efficiency is to benefit both from compiler-extracted program knowledge and hardware specialization, while the key for usability is to provide a generic, extensible software interface and easy-to-adopt hardware.

We envision MAPLE's VM-capable hardware and programming model to be reused and extended by both the hardware and the software community. For example, to do pipelining, where each program stage is executed in a different off-the-shelf core or accelerator. We are open-sourcing MAPLE with the publication of this paper.

ACKNOWLEDGMENTS

We thank Tyler Sorensen and Esin Tureci for helping to craft MAPLE's software API. We thank the rest of the DECADES team for their helpful feedback. This material is based on research sponsored by the Air Force Research Laboratory (AFRL) and Defense Advanced Research Projects Agency (DARPA) under agreement No. FA8650-18-2-7862 and the National Science Foundation (NSF) under Grant No. CNS-1823222. [1] Prof. Aragón was also supported by Fundación Séneca-Agencia de Ciencia y Tecnología, Región de Murcia, Programa Jiménez de la Espada (21508/EE/21).

[1]The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of AFRL, DARPA, NSF, or the U.S. Government.

REFERENCES

[1] Sam Ainsworth and Timothy M. Jones. 2016. Graph Prefetching Using Data Structure Knowledge. In *Proceedings of the 2016 International Conference on Supercomputing* (Istanbul, Turkey) *(ICS '16)*. Association for Computing Machinery, New York, NY, USA, Article 39, 11 pages. https://doi.org/10.1145/2925426.2926254

[2] Sam Ainsworth and Timothy M Jones. 2017. Software prefetching for indirect memory accesses. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 305–317.

[3] Sam Ainsworth and Timothy M Jones. 2018. An event-triggered programmable prefetcher for irregular workloads. *ACM SIGPLAN Notices* 53, 2 (2018), 578–592.

[4] A. Amid, D. Biancolin, A. Gonzalez, D. Grubb, S. Karandikar, H. Liew, A. Magyar, H. Mao, A. Ou, N. Pemberton, P. Rigge, C. Schmidt, J. Wright, J. Zhao, Y. S. Shao, K. Asanović, and B. Nikolić. 2020. Chipyard: Integrated Design, Simulation, and Implementation Framework for Custom SoCs. *IEEE Micro* 40, 4 (2020), 10–21.

[5] Krste Asanovic, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, et al. 2016. The Rocket chip generator. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17* (2016).

[6] Jonathan Balkind, Katie Lim, Fei Gao, Jinzheng Tu, David Wentzlaff, Michael Schaffner, Florian Zaruba, and Luca Benini. 2019. OpenPiton+Ariane: The First Open-Source, SMP Linux-booting RISC-V System Scaling From One to Many Cores. In *Third Workshop on Computer Architecture Research with RISC-V, CARRV*, Vol. 19.

[7] Jonathan Balkind, Katie Lim, Michael Schaffner, Fei Gao, Grigory Chirkov, Ang Li, Alexey Lavrov, Tri M. Nguyen, Yaosheng Fu, Florian Zaruba, Kunal Gulati, Luca Benini, and David Wentzlaff. 2020. BYOC: A "Bring Your Own Core" Framework for Heterogeneous-ISA Research. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 699–714. https://doi.org/10.1145/3373376.3378479

[8] Jonathan Balkind, Michael McKeown, Yaosheng Fu, Tri Nguyen, Yanqi Zhou, Alexey Lavrov, Mohammad Shahrad, Adi Fuchs, Samuel Payne, Xiaohua Liang, Matthew Matl, and David Wentzlaff. 2016. OpenPiton: An Open Source Manycore Research Framework. In *ASPLOS*. ACM, 217–232.

[9] A. Basak, S. Li, X. Hu, S. M. Oh, X. Xie, L. Zhao, X. Jiang, and Y. Xie. 2019. Analysis and Optimization of the Memory Hierarchy for Graph Processing Workloads. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 373–386.

[10] Nathan Beckmann. 2021. The Case for a Programmable Memory Hierarchy. https://www.sigarch.org/the-case-for-a-programmable-memory-hierarchy/.

[11] Peter L Bird, Alasdair Rawsthorne, and Nigel P Topham. 1993. The effectiveness of decoupling. In *Proceedings of the 7th international conference on Supercomputing*. 47–56.

[12] Cadence Design Systems. 2015. JasperGold Apps User's Guide.

[13] Luca P. Carloni. 2016. The Case for Embedded Scalable Platforms. In *Proceedings of the 53rd Design Automation Conference (DAC)*. 17:1–17:6.

[14] Jamison D Collins, Hong Wang, Dean M Tullsen, Christopher Hughes, Yong-Fong Lee, Dan Lavery, and John P Shen. 2001. Speculative precomputation: Long-range prefetching of delinquent loads. In *Proceedings 28th Annual International Symposium on Computer Architecture*. IEEE, 14–25.

[15] Neal Clayton Crago and Sanjay Jeram Patel. 2011. OUTRIDER: Efficient memory latency tolerance with decoupled strands. In *Proceedings of the 38th annual international symposium on Computer architecture*. 117–128.

[16] Leonardo Dagum and Ramesh Menon. 1998. OpenMP: An Industry-Standard API for Shared-Memory Programming. *IEEE Computational Science and Engineering* 5, 1 (Jan. 1998), 10 pages. https://doi.org/10.1109/99.660313

[17] Scott Davidson, Shaolin Xie, Christopher Torng, Khalid Al-Hawai, Austin Rovinski, Tutu Ajayi, Luis Vega, Chun Zhao, Ritchie Zhao, Steve Dai, Aporva Amarnath, Bandhav Veluri, Paul Gao, Anuj Rao, Gai Liu, Rajesh K. Gupta, Zhiru Zhang, Ronald Dreslinski, Christopher Batten, and Michael Bedford Taylor. 2018. The Celerity Open-Source 511-Core RISC-V Tiered Accelerator Fabric: Fast Architectures and Design Methodologies for Fast Chips. *IEEE Micro* 38, 2 (2018), 30–41. https://doi.org/10.1109/MM.2018.022071133

[18] Timothy A Davis. 2015. SuiteSparse: A suite of sparse matrix software. *URL http://faculty. cse. tamu. edu/davis/suitesparse. html* (2015).

[19] Esperanto Technologies. 2021. Esperanto's ET-Minion on-chip RISC-V cores. https://www.esperanto.ai/technology/.

[20] Harry D. Foster. 2015. Trends in Functional Verification: A 2014 Industry Study. In *Proceedings of the 52nd Annual Design Automation Conference* (San Francisco, California) *(DAC '15)*. Association for Computing Machinery, New York, NY, USA, Article 48, 6 pages. https://doi.org/10.1145/2744769.2744921

[21] James R Goodman, Jian-tu Hsieh, Koujuch Liou, Andrew R Pleszkun, PB Schechter, and Honesty C Young. 1985. PIPE: a VLSI decoupled architecture. *ACM SIGARCH Computer Architecture News* 13, 3 (1985), 20–27.

[22] Tae Jun Ham, Juan L. Aragón, and Margaret Martonosi. 2015. DeSC: Decoupled Supply-compute Communication Management for Heterogeneous Architectures. In *MICRO*. ACM.

[23] Tae Jun Ham, Juan L. Aragón, and Margaret Martonosi. 2019. Efficient Data Supply for Parallel Heterogeneous Architectures. *ACM TACO* 16, 2, Article 9 (2019), 23 pages. https://doi.org/10.1145/3310332

[24] Chen-Han Ho, Sung Jin Kim, and Karthikeyan Sankaralingam. 2015. Efficient execution of memory access phases using dataflow specialization. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*. 118–130.

[25] IEEE. 2013. Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language. (2013), 1–1315. https://doi.org/10.1109/IEEESTD.2013.6469140

[26] Yasuo Ishii, Mary Inaba, and Kei Hiraki. 2011. Access map pattern matching for high performance data cache prefetch. *Journal of Instruction-Level Parallelism* 13, 2011 (2011), 1–24.

[27] Lizy Kurian John, Vinod Reddy, Paul T Hulina, and Lee D Coraor. 1995. Program balance and its impact on high performance RISC architectures. In *Proceedings of 1995 1st IEEE Symposium on High Performance Computer Architecture*. IEEE, 370–379.

[28] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. 2017. The tensor algebra compiler. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–29.

[29] Manjunath Kudlur and Scott Mahlke. 2008. Orchestrating the execution of stream programs on multicore platforms. *ACM SIGPLAN Notices* 43, 6 (2008), 114–124.

[30] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO*. IEEE Press.

[31] Jaekyu Lee, Hyesoon Kim, and Richard Vuduc. 2012. When Prefetching Works, When It Doesn't, and Why. *ACM Trans. Archit. Code Optim.* 9, 1, Article 2 (March 2012), 29 pages. https://doi.org/10.1145/2133382.2133384

[32] Jure Leskovec, Deepayan Chakrabarti, Jon Kleinberg, Christos Faloutsos, and Zoubin Ghahramani. 2010. Kronecker Graphs: An Approach to Modeling Networks. *Journal of Machine Learning Reseach (JMLR)* 11 (March 2010), 985–1042.

[33] Sean Lie. 2021. Multi-Million Core, Multi-Wafer AI Cluster. In *2021 IEEE Hot Chips 33 Symposium (HCS)*. IEEE Computer Society, 1–41.

[34] Katie Lim, Jonathan Balkind, and David Wentzlaff. 2019. JuxtaPiton: Enabling Heterogeneous-ISA Research with RISC-V and SPARC FPGA Soft-Cores. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Seaside, CA, USA) *(FPGA '19)*. Association for Computing Machinery, New York, NY, USA, 184. https://doi.org/10.1145/3289602.3293958

[35] Chi-Keung Luk. 2001. Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors. In *Proceedings 28th Annual International Symposium on Computer Architecture*. IEEE, 40–51.

[36] William Mangione-Smith, Santosh G Abraham, and Edward S Davidson. 1990. The effects of memory latency and fine-grain parallelism on astronautics ZS-1 performance. In *Twenty-Third Annual Hawaii International Conference on System Sciences*, Vol. 1. IEEE, 288–296.

[37] Aninda Manocha, Tyler Sorensen, Esin Tureci, Opeoluwa Matthews, Juan L Aragón, and Margaret Martonosi. 2021. GraphAttack: Optimizing Data Supply for Graph Applications on In-Order Multicore Architectures. *ACM Transactions on Architecture and Code Optimization (TACO)* 18, 4 (2021), 1–26.

[38] Opeoluwa Matthews, Aninda Manocha, Davide Giri, Marcelo Orenes-Vera, Esin Tureci, Tyler Sorensen, Tae Jun Ham, Juan L Aragón, Luca P Carloni, and Margaret Martonosi. 2020. MosaicSim: A Lightweight, Modular Simulator for Heterogeneous Systems. In *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 136–148.

[39] Mohammad Hasanzadeh Mofrad, Rami Melhem, Yousuf Ahmad, and Mohammad Hammoud. 2019. Multithreaded Layer-wise Training of Sparse Deep Neural Networks using Compressed Sparse Column. In *2019 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–6.

[40] K. J. Nesbit and J. E. Smith. 2004. Data Cache Prefetching Using a Global History Buffer. In *10th International Symposium on High Performance Computer Architecture (HPCA'04)*. 96–96.

[41] Quan M Nguyen and Daniel Sanchez. 2020. Pipette: Improving Core Utilization on Irregular Applications through Intra-Core Pipeline Parallelism. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 596–608.

[42] Marcelo Orenes-Vera, Aninda Manocha, David Wentzlaff, and Margaret Martonosi. 2021. AutoSVA: Democratizing Formal Verification of RTL Module Interactions. In *Proceedings of the 58th Design Automation Conference (DAC)*.

[43] A. Parashar, M. Pellauer, M. Adler, B. Ahsan, N. Crago, D. Lustig, V. Pavlov, A. Zhai, M. Gambhir, A. Jaleel, R. Allmon, R. Rayess, and J. Emer. 2013. Triggered Instructions: A Control Paradigm for Spatially-Programmed Architectures. In *ISCA*.

[44] J-M Parcerisa and Antonio Gonzalez. 1999. The synergy of multithreading and access/execute decoupling. In *Proceedings Fifth International Symposium on High-Performance Computer Architecture*. IEEE, 59–63.

[45] Ram Rangan, Neil Vachharajani, Manish Vachharajani, and David I. August. 2004. Decoupled Software Pipelining with the Synchronization Array. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. http://dx.doi.org/10.1109/PACT.2004.14

[46] RISC-V Foundation. 2019. Riscv-tests. https://github.com/riscv/riscv-tests..
[47] Karl Rupp. 2018. 42 Years of Microprocessor Trend Data. https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/.
[48] James Smith. 1982. Decoupled Access/Execute Computer Architectures. In *Proceedings of the 9th Annual Symposium on Computer Architecture* (Austin, Texas, USA) *(ISCA)*. 8 pages. http://dl.acm.org/citation.cfm?id=800048.801719
[49] James E Smith. 1982. Decoupled access/execute computer architectures. In *ACM SIGARCH Computer Architecture News*, Vol. 10. IEEE Press.
[50] Tyler Sorensen, Aninda Manocha, Esin Tureci, Marcelo Orenes-Vera, Juan L Aragón, and Margaret Martonosi. 2020. A simulator and compiler framework for agile hardware-software co-design evaluation and exploration. In *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. IEEE, 1–9.
[51] Tyler Sorensen, Aninda Manocha, Esin Tureci, Marcelo Orenes-Vera, Juan L. Aragón, and Margaret Martonosi. 2020. A Simulator and Compiler Framework for Agile Hardware-Software Co-design Evaluation and Exploration. In *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. 1–9.
[52] V. Srinivasan, R. B. R. Chowdhury, and E. Rotenberg. 2020. Slipstream Processors Revisited: Exploiting Branch Sets. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 105–117.
[53] Xian-He Sun and Yong Chen. 2010. Reevaluating Amdahl's law in the multicore era. *Journal of Parallel and distributed Computing* 70, 2 (2010), 183–188.
[54] Karthik Sundaramoorthy, Zach Purser, and Eric Rotenberg. 2000. Slipstream processors: Improving both performance and fault tolerance. *ACM SIGPLAN Notices* 35, 11 (2000), 257–268.
[55] Michael Sung, Ronny Krashinsky, and Krste Asanović. 2001. Multithreading decoupled architectures for complexity-effective general purpose computing. *ACM SIGARCH Computer Architecture News* 29, 5 (2001), 56–61.
[56] Nishil Talati, Kyle May, Armand Behroozi, Yichen Yang, Kuba Kaszyk, Christos Vasiladiotis, Tarunesh Verma, Lu Li, Brandon Nguyen, Jiawen Sun, et al. 2021. Prodigy: Improving the Memory Latency of Data-Indirect Irregular Workloads Using Hardware-Software Co-Design. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 654–667.
[57] Michael Bedford Taylor, Jason Kim, Jason Miller, David Wentzlaff, Fae Ghodrat, Ben Greenwald, Henry Hoffman, Paul Johnson, Jae-Wook Lee, Walter Lee, et al. 2002. The raw microprocessor: A computational fabric for software circuits and general-purpose programs. *IEEE micro* 22, 2 (2002), 25–35.
[58] Kim-Anh Tran, Trevor E Carlson, Konstantinos Koukos, Magnus Själander, Vasileios Spiliopoulos, Stefanos Kaxiras, and Alexandra Jimborean. 2017. Clairvoyance: look-ahead compile-time scheduling. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 171–184.
[59] Kim-Anh Tran, Alexandra Jimborean, Trevor E Carlson, Konstantinos Koukos, Magnus Själander, and Stefanos Kaxiras. 2018. SWOOP: software-hardware co-design for non-speculative, execute-ahead, in-order cores. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 328–343.
[60] P. N. Whatmough, M. Donato, G. G. Ko, S. K. Lee, D. Brooks, and G. Wei. 2020. CHIPKIT: An Agile, Reusable Open-Source Framework for Rapid Test Chip Development. *IEEE Micro* 40, 4 (2020), 32–40.
[61] Wm A Wulf. 1992. Evaluation of the WM Architecture. In *Proceedings of the 19th annual international symposium on Computer architecture*. 382–390.
[62] Xiangyao Yu, Christopher J Hughes, Nadathur Satish, and Srinivas Devadas. 2015. IMP: Indirect memory prefetcher. In *Proceedings of the 48th International Symposium on Microarchitecture*. 178–190.
[63] Florian Zaruba and Luca Benini. 2018. Ariane: An Open-Source 64-bit RISC-V Application Class Processor and latest Improvements. Technical talk at the RISC-V Workshop https://www.youtube.com/watch?v=8HpvRNh0ux4.
[64] F. Zaruba and L. Benini. 2019. The Cost of Application-Class Processing: Energy and Performance Analysis of a Linux-Ready 1.7-GHz 64-Bit RISC-V Core in 22-nm FDSOI Technology. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 27, 11 (Nov 2019), 2629–2640. https://doi.org/10.1109/TVLSI.2019.2926114 https://github.com/openhwgroup/cva6.
[65] Florian Zaruba, Fabian Schuiki, and Luca Benini. 2020. Manticore: A 4096-Core RISC-V Chiplet Architecture for Ultraefficient Floating-Point Computing. *IEEE Micro* 41, 2 (2020), 36–42.
[66] Weifeng Zhang, Dean M Tullsen, and Brad Calder. 2007. Accelerating and adapting precomputation threads for efficient prefetching. In *2007 IEEE 13th International Symposium on High Performance Computer Architecture*. IEEE, 85–95.
[67] Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman Amarasinghe. 2018. Graphit: A high-performance graph dsl. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–30.