UpDown: A Novel Architecture for Unlimited Memory Parallelism

Andronicus Rajasukumar*
andronicus@uchicago.edu
University of Chicago
Chicago, Illinois, USA

Tianchi Zhang*
tonyztc@uchicago.edu
University of Chicago
Chicago, Illinois, USA

Ruiqi Xu
ruiqix@uchicago.edu
University of Chicago
Chicago, Illinois, USA

Andrew A. Chien
aachien@uchicago.edu
University of Chicago
Argonne National Laboratory
Chicago, Illinois, USA

ABSTRACT

The emergence of HBM as a high-volume memory product has made memory bandwidths of 1.2TB/s (1 stack) to 4.8TB/s (4 stacks) feasible. Exploiting such bandwidths requires high memory level parallelism, but the memory access mechanisms in today's CPUs are ill-suited. We define the Memory Parallelism Abstract Machine (MPAM) that characterizes limits of a variety of various commercial and research designs.

We propose the UpDown architecture that generates unlimited, cost-efficient memory parallelism using split-transaction accesses and a large compute-namespace to synchronize memory responses. Using MPAM, we show that UpDown can generate unlimited memory parallelism constrained only by the memory technology servicing the system and memory reference issue rate.

Our evaluation shows that the smallest compute element of UpDown , a single lane can generate up to **3.5x** more memory parallelism compared to a modern out-of-order CPU core, despite its much smaller area (<1%). We also show that 64 lanes of the UpDown architecture can sustain **1,673** outstanding memory references to nearly saturate the full bandwidth of 1 HBM3e stack (1.2TB/s). Finally, we also show that UpDown is much more energy and power efficient.

CCS CONCEPTS

• **Computer systems organization → Multicore architectures**; **Parallel architectures**.

KEYWORDS

Memory Level Parallelism, High Bandwidth Memory, Accelerator, Parallel Architecture

1 INTRODUCTION

For decades, CPU-based computer systems have been designed to minimize their memory bandwidth needs because it was expensive and accessing memory was a long latency operation. Over time, as CPU clock rates increased, we saw the rise of deep memory hierarchies that filter programs' memory traffic, producing low amortized memory access time (AMAT) and minimizing the number of program memory requests that need to go all the way to DRAM. Directly, they minimized the need to access memory (memory bandwidth).

*Both authors contributed equally to this research

One reason for this is that DRAM memories have historically used inexpensive packages with low-pin-count interfaces (e.g., DDR1-5, LPDDR, etc.). This packaging meant that increasing memory bandwidth required many memory packages, and in recent times, many DDR interfaces to reach a few hundred GB/s of memory bandwidth [12, 61].

Recently, the emergence of wide memory interfaces with 1024 IO's [30, 50] has changed the situation. With wide interfaces, a small number of DRAM chips can provide bandwidths as high as 1,200GB/s, exposing the bank-level/rank-level parallelism of the DRAM chips. In today's commercial products, a small number of stacks (6) can deliver > 7 TB/s in a single package [62]. Directly, with these stacked DRAM technologies, memory bandwidth has become plentiful and inexpensive.

Because of the long latency of DRAM, exploiting the high memory bandwidth requires compute elements to generate high memory-level parallelism in the form of large numbers of outstanding memory requests. This is a radically different objective than traditional memory hierarchy design [26, 51] which focuses on latency avoidance and bandwidth reduction. For this new world of plentiful memory bandwidth, *we describe and evaluate a range of approaches for generating memory parallelism.*

To structure evaluation, we first define the *Memory Parallelism Abstract Machine (MPAM)* model that captures the fundamental limits and resources used by conventional architectures to generate memory parallelism. We use MPAM to study four commercial systems. Next, we propose the UpDown accelerator, describing its novel memory access mechanisms of split-transaction and flexible software-defined synchronization. Using the MPAM model, we compare UpDown to conventional approaches. Finally, we use varied applications to compare both memory parallelism achieved (performance) and power/area (cost) for a range of approaches.

Specific contributions of the paper include:

- Definition and description of the *memory parallelism abstract machine (MPAM)* to model the limits of memory parallelism in computer architectures
- Application of *MPAM* to four commercial CPUs, using it to accurately characterize their design and performance
- Design of UpDown, a novel architecture that achieves unlimited memory parallelism using software-managed split-transaction memory access and compute-name synchronization to achieve efficient memory parallelism scaling

- Evaluation that shows UpDown saturates full HBM3e stack bandwidth (1.2TB/s) by sustaining **1,673** outstanding memory references. The smallest UpDown element, a single lane can generate **3.5x** the memory parallelism of an out-of-order (OOO) core. UpDown is also much more energy and power efficient.

In Section 2 and Section 3, we describe the background and motivation for our work. We present the Memory Parallelism Abstract Machine (MPAM) model in Section 4, showing how various architectures map to the model. In Section 5, we present the design of the UpDown architecture. In Section 6, we describe the methodology used for the evaluation in Section 7. Related work is discussed in Section 8. Finally, in Section 9, we summarize the results and discuss future research directions.

2 BACKGROUND

We briefly present techniques that have been adopted in various architectures to extract higher Memory Level Parallelism (MLP).

2.1 Out-of-Order Execution

Traditional CPUs use aggressive out-of-order (OOO) techniques to issue a high number of outstanding memory requests. Multiple-issue superscalar cores support out-of-order execution using wide instruction windows to perform static or dynamic scheduling [23], supported with large Reorder-Buffers [59] to hold the state of out-of-order uncommitted instructions. Register-renaming is used to avoid anti-dependencies between instructions and is implemented with separate reservation stations or reuses the reorder buffer [63, 68]. Deep load/store queues are additionally used to book-keep and track outstanding memory references. These queues implement store-forwarding and memory ordering checks to maintain an architecture-specific consistency model [22, 55]. While these schemes enable high single-threaded performance on compute-bound applications, the number of in-flight memory accesses is limited by internal namespaces imposed by these structures on the architecture. Further, book-keeping is usually done using Content Addressable Memories that are very expensive to scale for higher memory parallelism as required by new high bandwidth memories.

2.2 Mechanisms for Memory Parallelism in Caches

Caches are designed to avoid long-latency DRAM accesses. But on applications with low data reuse, caches are ineffective with high miss rates. Miss Handling Architectures (MHA) in lock-up free/non-blocking caches [18, 32] support multiple outstanding misses using specialized structures - Miss Status Holding Registers (MSHR) for book-keeping. These registers create another multi-level namespace for memory parallelism, controlling the number of outstanding memory requests.

Hardware Prefetching in caches is a popular technique to lower cache miss rates. A number of sophisticated prefetchers have been proposed and implemented [11, 27, 69]. However, prefetchers compete for the same resources as demand misses - cachelines and MSHRs (memory bandwidth). Increasing prefetch to increase memory parallelism can have a negative impact on performance due to this [39]. Additionally, effective prefetching for irregular applications is challenging and is an area of active research [46, 54, 73].

2.3 High Bandwidth Memories

Data-intensive workloads like AI, large-scale graph processing, etc., fueled the development and rapid adoption of Stacked DRAMs like HBMs. These memories have increased DRAM bandwidths many-fold by increasing the interface width to 1024 bits and beyond. Table 1 captures the current and future HBM memories and their bandwidths, with a projected 2+ TB/s of bandwidth per stack of HBM. Saturating these high bandwidths requires generating and sustaining ~ 3200 outstanding memory requests. Current architectures are incapable of achieving this or require prohibitively expensive scaling of existing mechanisms to do so.

Table 1: High bandwidth DRAM technologies [13, 56, 57]

DRAM technology	Max Bandwidth per pin	Max Bandwidth per stack	Max Capacity
HBM2e	3.6 GT/s	460 GB/s	16GB
HBM3	6.4 GT/s	819.2 GB/s	24GB
HBM3e	9.6 GT/s	1.28 TB/s	48GB
HBMNext	> 10 GT/s	> 2 TB/s	36-64GB

2.4 The UpDown System and Project

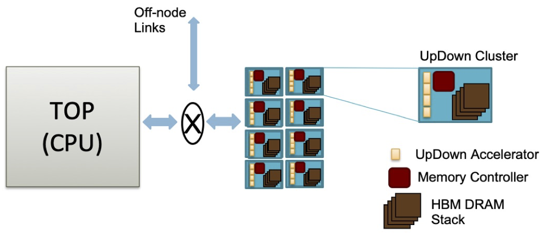
The novel memory access mechanisms for UpDown, described in this paper, are part of a larger system design project funded as part of IARPA's Advanced Graphic Intelligence Logical Computing Environment (AGILE) program [2]. The objective of the overall program is to create breakthrough performance on irregular applications and graphs with extreme skew and low data reuse.

The UpDown system is an ambitious design under study and development as part of the AGILE program, driven by a team of researchers from the University of Chicago, Purdue University, and Tactical Computing Laboratories. The UpDown system is a collection of UpDown accelerators, each with 64 lanes, employing the memory access mechanisms for unlimited parallelism described in this paper. A full description of the UpDown instruction set architecture can be found here [14].

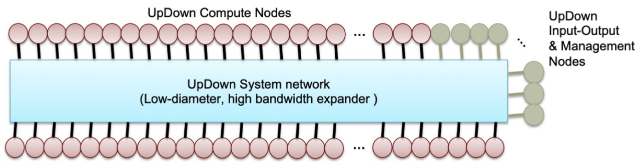
Each UpDown node consists of 32 UpDown accelerators and 8 HBM DRAM stacks, connected to a CPU as illustrated in Figure 1a [52]. The DRAM is shared amongst all of the accelerators and the CPU on the node, and can be globally addressed by all of the nodes in the system. The Updown system is 16,384 Updown nodes with 512GB DRAM each connected by a high-bandwidth, low-diameter network [33], with latency of 0.5 microseconds and > 50 petabytes/second bisection – 50x greater than today's largest supercomputers. The overall system is depicted in Figure 1b.

The innovative memory access interface discussed in this paper is one novel element in the UpDown architecture. This feature gives UpDown the ability to exploit high memory bandwidth, a key for irregular graph applications.

The UpDown architecture builds on previous research architectures namely the Unified Automata Processor (UAP) [16] and the Unstructured Data Processor (UDP) [17]. UpDown extends the fast



(a) An UpDown node


UpDown Compute Nodes
UpDown Input-Output & Management Nodes
UpDown System network
(Low-diameter, high bandwidth expander)

(b) The UpDown System

Figure 1: The UpDown node and system

symbol processing and multi-way dispatch capabilities in UAP and UDP to generic event-driven execution.

3 MOTIVATION

Irregular applications with structures like graphs, hash-maps, and trees have low data reuse and therefore become memory latency bound. High memory level parallelism is required to achieve high performance on these applications. Current architectures require

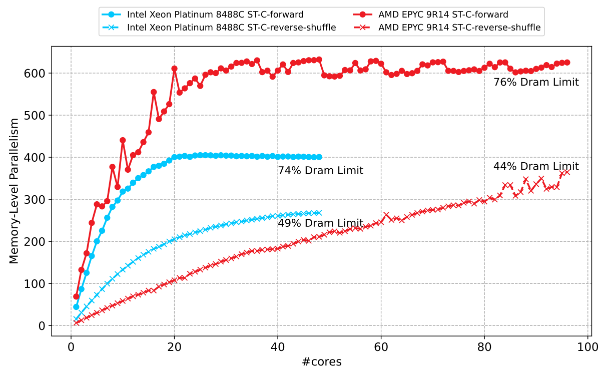


Figure 2: Memory Parallelism achieved on commercial multi-core CPUs using STREAM-Copy[42]

expensive memory access mechanisms and hardware scaling to generate high enough memory parallelism for good performance. We present the results of the STREAM benchmark [42] on various commercial systems. We modified the STREAM benchmark to also do a reverse-shuffle scan of addresses (Figure 2) to minimize the effect of prefetch. Figure 2 shows the achieved memory bandwidth with increasing hardware thread count. It shows that > 20 sophisticated OOO cores are required to saturate > 70% of the system's 300 GB/s memory bandwidth on Intel Xeon Platinum 8488C. On AMD EPYC 9R14, nearly 60 cores are required to reach 90% of system's 412.5 GB/s bandwidth. The results further show that when prefetching is ineffective, even lower memory parallelism and bandwidth is achieved. From a scaling point of view, fewer GB/s are achieved per core.

These experiments highlight the inherent limits to memory parallelism in modern CPU architectures. Today's architectures (and micro-architectures) create local namespaces for book-keeping and synchronization and to support memory ordering and coherence protocols. These namespaces are further bound to physical resources, making generating and scaling memory parallelism expensive.

We define the *Memory Parallelism Abstract Machine (MPAM)* that models these fundamental limits to memory parallelism. Building on this model, we propose UpDown: a novel architecture that enables very high memory parallelism. The UpDown approach involves a key set of memory access mechanisms - **software-managed split transaction DRAM access, compute-name synchronization** coupled with **efficient memory parallelism scaling** that seek to overcome the limits imposed by these local namespaces.

4 MEMORY PARALLELISM ABSTRACT MACHINE (MPAM)

4.1 Memory Parallelism Limits

Memory parallelism in current architectures is fundamentally limited by three internal namespaces used for book-keeping and synchronizing in-flight memory requests. These namespaces are bound to architectural resources (renamed registers, load/store buffers, MSHRs, shared queues, etc.) that either limit or make them prohibitively expensive to scale. We describe these limiting namespaces below and use them to define the **Memory Parallelism Abstract Machine (MPAM)** to capture the limits to memory parallelism in any given architecture.

4.1.1 ***Synchronization Namespace (N_{sync}):*** Processor architectures create names to allow memory responses to be synchronized into thread execution. Traditional CPU architectures typically map the architecture ISA register names to a larger physical register namespace to facilitate renaming and retain true dependencies between instructions. They further bind this physical register namespace to either a large physical register file, reorder buffers, reservation stations, etc., depending on specific implementations. Execution of load/store instructions allocates one of these names to be used for synchronization with memory responses (store commits or load data). Thus, a finite, resource-bound Synchronization Namespace N_{sync} exists that imposes a fundamental limit on the number of memory references that the architecture can issue.

4.1.2 ***Outstanding Request Namespace (N_{out}):*** Processor backends create a second local namespace to book-keep and track in-flight memory requests. This namespace is shared by two types of in-flight memory requests.

(1) Synchronous Request namespace (N_{out_sync}): Synchronous memory requests are those generated by load/store instructions. These memory responses are synchronized back into the thread execution.

(2) Asynchronous Request namespace (N_{out_async}): Asynchronous memory requests are memory requests generated independent of instruction execution by mechanisms like prefetch / direct memory access (DMA). These memory responses are not synchronized into thread execution but fill up caches / scratchpads in an effort to increase hit rates and hide memory latencies.

The total Outstanding request namespace is the sum of these two sets of names, so that

$$N_{out} = N_{out\_sync} + N_{out\_async} \tag{1}$$

The outstanding request namespace is often bound to content addressable memories like MSHRs in caches, where the names are used to perform an associative search over all the entries. Just like the Synchronization namespace, the Outstanding Request namespace also imposes a fundamental limit on the number of memory references the architecture can generate and sustain.

4.1.3 ***Shared Request Namespace (N_{sh}):*** Finally, architectures with multiple cores typically share resources in the memory path (L3 / LLC / System Level Caches / Shared FIFO queues), adding a further limit to the total outstanding requests. N_{sh} imposes a restrictive limit on N_{synch} and N_{out}. In the absence of these shared resources, N_{sh} falls back to the entire address space.

4.2 Model Description

Figure 3 depicts an Abstract Machine that captures the namespaces described above. The machine has L FrontEnds (AM_{FE}) which contribute L copies of the Synchronization namespace N_{sync}. K backends AM_{BE} track the references generated by AM_{FE} (N_{out_sync}) in addition to generating their own memory requests (N_{out_async}). Finally, J shared resources AM_{SH} impose an overarching limit on all requests generated cumulatively from AM_{FE} and AM_{BE}. With this description of the Abstract Machine AM, we present the total limit to the memory level parallelism of this machine (MLP), as

$$\begin{aligned} MLP =& (1/cl)\, min(\, min(L \cdot N_{sync} \cdot s_s, K \cdot N_{out\_sync} \cdot s_o) \\ & + K \cdot N_{out\_async} \cdot s_o, J \cdot N_{sh} \cdot s_{sh}) \end{aligned} \tag{2}$$

where s_s, s_o, s_{sh} are the request sizes (in words) of the three namespaces N_{sync}, N_{out} and N_{sh} respectively and cl is the cacheline size (in words). Therefore, the MLP generated by **MPAM** gives the limit on the total number of outstanding cachelines possible in a given architecture.

The exact values of N_{sync}, N_{out_sync} and N_{out_async} depend on the implementation of AM_{FE} and AM_{BE} in specific architectures.

MPAM provides a model to reason about the memory level parallelism capabilities of an architecture being designed. It allows architects to reason about the size of the namespaces and the number and type of resources to be used in order to achieve a desired MLP capability. A number of important ideas are immediately apparent from the model as stated below.

(1) Varying L, K and J leads to various configurations of single-core/multi-core architectures. For example, When L is a multiple of K, we get an SMT/Hyperthreaded AM_{FE} with (L/K) threads per core. As another example, $L = K = n_c$ (number of cores) leads to a multi-core architecture where each AM_{BE} is private to an AM_{FE} as in the case of private $l1$ and $l2$ caches.

(2) N_{sync} and N_{out_sync} are closely co-ordinated namespaces. Thus increase in N_{out_sync} will be limited by N_{sync} and further increase will require both to be increased.

(3) N_{out_async}, as the name suggests, can be increased independently of program execution and is only limited by the resources the namespace is bound to.

(4) The N_{sh} namespace, as mentioned earlier, is an overarching limit on the total outstanding requests. Architectures that use, for example, a shared inclusive LLC / System Level Cache will be limited by the MSHRs in this cache. Other architectures like Intel's core microarch (post-Skylake), where the LLC is a non-inclusive victim cache, are not limited by this namespace.

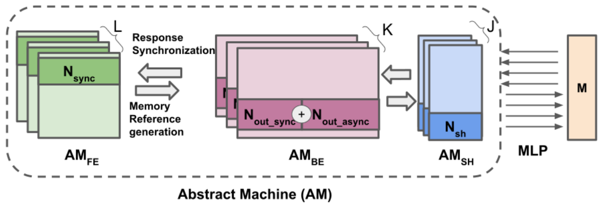


Figure 3: Memory Parallelism Abstract Machine captures the fundamental limits to memory parallelism in three separate resource bound namespaces - Synchronization Namespace (N_{sync}), Outstanding Request Namespace(N_{out}) and Shared Request Namespace(N_{sh})

In the next section, we show how MPAM can be used to model the memory parallelism in specific implementations of AM - a trivial in-order machine and a complex realistic out-of-order machine. Subsequently, we validate this model with measurements on 2 commercial systems whose architectural parameters / resources are available. Finally, we present an application of MPAM in deriving some of these architectural parameters that limit memory level parallelism on 2 commercial systems.

4.3 Modeling Microarchitectures with MPAM

We start with a trivial single in-order core implementation (without prefetch) and proceed to a more complex out-of-order core with prefetch example.

4.3.1 In-order Load/Store Architectures. For a single In-order load/store architecture, $L = K = 1$. $N_{sync} = R$, the ISA register namespace; $N_{out\_sync} = 1$, $N_{out\_async} = 0$, since an in-order core can only sustain 1 outstanding request and there is no prefetch. Additionally, $s_s = 1$, $s_o = 1$, Using Eqn:2, we get,

$$MLP = (1/cl)\, min(R, 1) \tag{3}$$

4.3.2 Out-of-Order Load/Store Architectures. For a single out-of-order core, $L = K = 1$. AM_{FE} is implemented using a number of sophisticated mechanisms like speculative execution, dynamic scheduling, etc. These mechanisms are supported using a large number of physical registers $R_{phy} > R$, using register renaming and reorder buffers. OOO cores use R_{phy} to synchronize outstanding requests. This gives $N_{sync} = R_{phy}/cl$, $s_s = 1$.

The AM_{BE} uses multiple levels of caches and implements bookkeeping using MSHRs at each level. MSHRs track outstanding references at the cache-line granularity. Hardware Prefetching adds an asynchronous component N_{out_async} and can increase the number of outstanding requests to memory. If we assume prefetch is implemented in $l2$ (as in most Intel architectures), and use α to indicate the effectiveness of prefetching we get

$$N_{out\_sync} = MSHR_{l1} \tag{4}$$
$$N_{out\_async} = \alpha(MSHR_{l2} - MSHR_{l1}) \tag{5}$$

So the MLP for a single OOO core is given by

$$\begin{aligned} MLP =& min(R_{phy}/cl, MSHR_{l1}) \\ &+ \alpha \cdot (MSHR_{l2} - MSHR_{l1}) \end{aligned} \tag{6}$$

4.3.3 Multi-core Out-of-Order Load/Store Architectures. With multiple cores (n_c) of the type in 4.3.2 sharing a single $l3$ cache, we have $L = K = n_c$, $J = 1$. N_{sync} is replicated $L = n_c$ times and so are N_{out_sync} and N_{out_async}, by $K = n_c$ times. N_{sh} in this multi-core architectures is $N_{sh} = MSHR_{l3}$. Then we have

$$N_{sync} = n_c \cdot (1/cl)(R_{phy}) \tag{7}$$
$$N_{out\_sync} = n_c \cdot MSHR_{l1}$$
$$N_{out\_async} = n_c \cdot \alpha(MSHR_{l2} - MSHR_{l1})$$
$$N_{sh} = MSHR_{l3} \tag{8}$$
$$\begin{aligned} MLP =& min(n_c \cdot min(R_{phy}/cl, MSHR_{l1}) \\ &+ n_c \cdot \alpha(MSHR_{l2} - MSHR_{l1}), MSHR_{l3}) \end{aligned} \tag{9}$$

It is easy to verify that when $n_c = 1$ and $MSHR_{l3}$ is set to a large value, Eqn:9 reduces to Eqn:6 for a single OOO core (with no $l3$).

Appendix A contains an extensive set of commercial and research architectures with mapping to MPAM to show the robustness and flexibility of MPAM in understanding the limits of memory parallelism in the system. Next, we corroborate this model with measurements taken on 2 commercial platforms.

4.4 Validating MPAM's ability to Model Commercial Systems

4.4.1 Intel Platinum 8375C. : We first corroborate MPAM on Intel Platinum 8375c, based on the Icelake architecture. For this and other commercial designs we summarize the published parameters in Table 2). For this architecture, $MSHR_{l1} = 12$, $MSHR_{l2} = 48$, $R_{phy} = 224$ and $cl = 8$ based on [35]. [4] suggests that the $l3$ cache is a non-inclusive victim cache and N_{sh} for the $l3$ cache falls back to the entire memory address space (we do not show this below).

Figure 4 plots the various namespaces with increasing number of cores (up to 32 cores). We plot the following limits based on Eqn:9 to show the interaction of the various limits in the architecture.

$$N_{sync} = 28n_c \tag{10}$$
$$N_{out\_sync} = 12n_c$$
$$N_{out\_async} = 36\alpha n_c$$
$$MLP = 12n_c + 36\alpha n_c \text{ for } 0 \leq \alpha \leq 1 \tag{11}$$

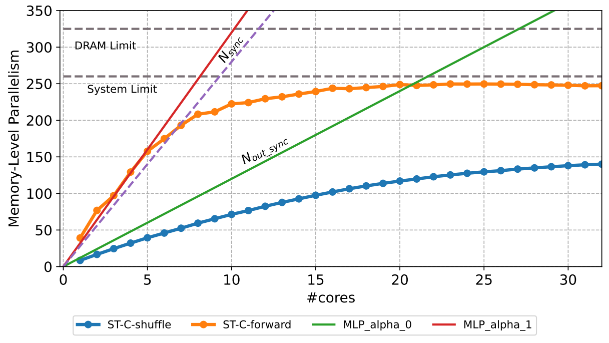


Figure 4: MPAM Corroboration with Intel Platinum 8375C Platform using published numbers

We also plot STREAM [42] results for forward (ST-C-forward) and shuffled (ST-C-shuffle). As can be seen, *MLP_alpha_0* provides a lower bound for when the hardware prefetching is ineffective ($N_{out\_async} = 0$), and *MLP_alpha_1* provides an upper bound for effective prefetching ($N_{out\_async} = 36n_c$). Finally, we plot the DRAM limit of the platform (4-channels, DDR4-3200).

Table 2: Systems for MPAM demonstration and validation [12, 21, 34, 35]

Systems	# Cores	Cache	Memory

Intel Platinum 8375C	32	L1d: 48KB per core, 1.4ns;
		L2: 1.25MB per core, 4.0ns;
		L3: 54MB on chip, 27.9ns
			8 Channel
			DDR4-3200
			103.3ns

AMD EPYC 9R14	96	L1d: 32KB per core, 0.7ns;
		L2: 1MB per core, 2.4ns;
		L3: 32MB per CCD, 10.9ns;
		386MB on chip, 191.0ns
			12 Channel
			DDR5-4400
			129.4ns

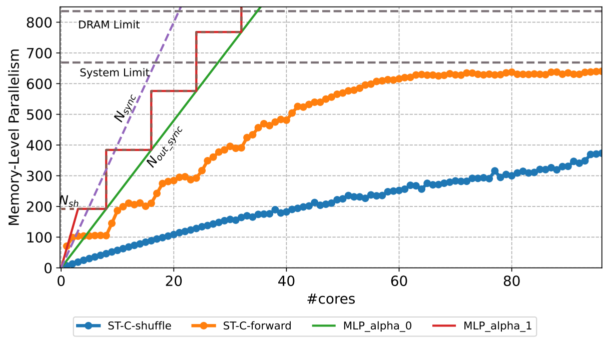
Intel Platinum 8488C	48	L1d: 48KB per core, 1.3ns;
		L2: 2MB per core, 4.2ns;
		L3: 105MB on chip, 33.0ns
			8 Channel
			DDR5-4800
			127.9ns

AWS Graviton 3	64	L1d: 64KB per core, 1.6ns;
		L2: 1MB per core, 4.3ns;
		L3: 32 MB on chip, 30.6ns
			8 Channel
			DDR5-4400
			102.2ns

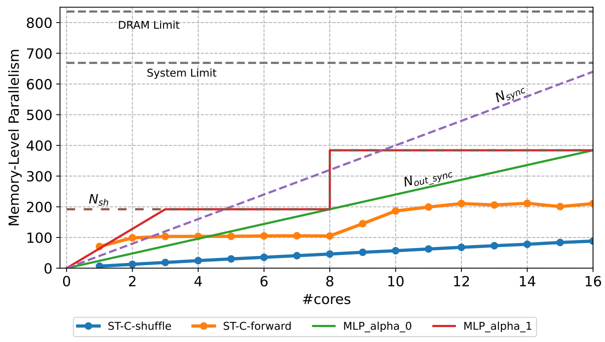
4.4.2 AMD EPYC 9R14. : We repeat similar experiments on AMD EPYC 9R14 (based on Zen4 architecture). Using publicly available parameters [53] ($MSHR_{l1} = 24$, $MSHR_{l2} = 64$, $MSHR_{l3} = 192$, $R_{phy} = 320$, and $cl = 8$), we plot MPAM limits alongside STREAM results as before in Figure 5a. This EPYC platform has an $l3$ per CCD (8 cores) which adds an $N_{sh} = 192\lceil n_c/8 \rceil$ namespace as below,

$$MLP = min(24n_c + 64\alpha n_c, 192\lceil n_c/8 \rceil) \text{ for } 0 \leq \alpha \leq 1 \tag{12}$$

We show the first 16 cores in Figure 5b to highlight the effect of N_{sh} and how MPAM predicts the effect of shared namespaces at



(a) MPAM with all 96 cores on AMD EPYC 9R14



(b) Expanded graph for 16 cores to highlight effect of N_{sh} due to per-CCD L3

Figure 5: MPAM Corroboration with AMD EPYC 9R14 Platform

various hierarchies. While we used 192 as $MSHR_{l3}$, the results show that the actual limit is probably lower $\sim$ 100, which needs further investigation to see if it is a limit by design or an inefficiency.

Thus, MPAM effectively predicts the memory parallelism limits for a given architecture. Designers can use this model to predict the effectiveness of their architectures, the memory parallelism scaling trends with increasing cores, and the cost of achieving a given level of memory parallelism required by applications.

4.5 Using MPAM to derive architectural parameters

In this section, we demonstrate MPAM's applicability in deriving architectural parameters in two commercial systems. We run STREAM [42] on each system with increasing numbers of hardware cores (without hyper-threading) and measure the memory parallelism achieved in each system. Using a least squares estimate from the measurements of the first few (up to three cores), we predict the architectural parameters using **MPAM**. We also run a modified STREAM version with a reverse shuffled scan of addresses to reduce the effect of prefetching. We show the detailed parameter extraction for the first system as an example and summarize the remaining two in Table 3.

4.5.1 Intel Platinum 8488C. : Intel Platinum 8488C is based on the Sapphire Rapids architecture. The forward and shuffle scan STREAM[42] results are shown in Figure 6 (top) and Figure 6 (bottom) respectively. The least squares fit (3 cores) for these measurements are shown in Eqn 13, 14. We set the y-intercept of the fit to 0 to ensure there is no residual parallelism for 0 cores. The STREAM benchmark flattens at $\sim$ 70% of the maximum parallelism of the DRAM system (8 channels, DDR5-4800).

$$MLP_f = 44n_c \tag{13}$$

$$MLP_s = 16n_c \tag{14}$$

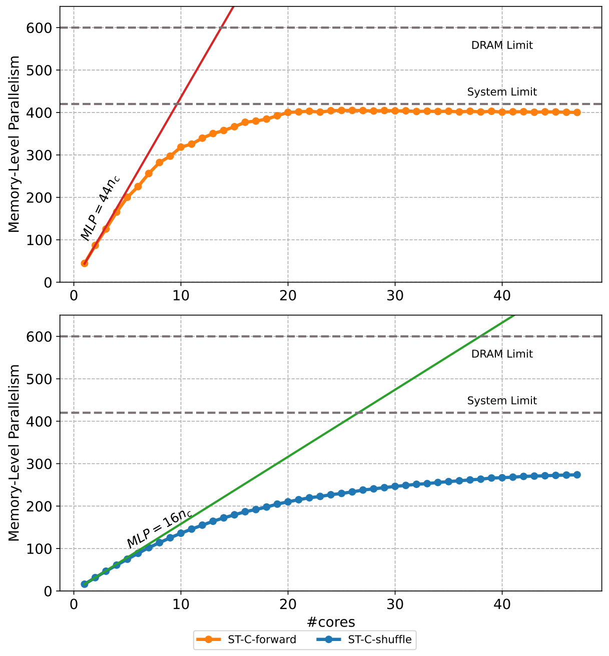


Figure 6: Memory Parallelism in Intel(R) Xeon(R) Platinum 8488C with - Forward Scan (top), Shuffle Scan (bottom)

We set $n_c = 1$, $\alpha = 0$ in Eqn:14 (no prefetch), and comparing with Eqn:9 (removing $MSHR_{l3}$ as in Section 4.4) we get

$$min(R_{phy}/8, MSHR_{l1}) = 16 \tag{15}$$

Given prior Intel microarchitectures have always had $R_{phy} > 128$ and assuming the same here, we get

$$MSHR_{l1} = 16 \tag{16}$$

Again setting $n_c = 1$ in Eqn:13 (with prefetch), using the computed $MSHR_{l1}$ and comparing with Eqn:9 we get

$$16 + \alpha(MSHR_{l2} - 16) = 44 \tag{17}$$

Solving for this and using $\alpha = 1$, we get

$$MSHR_{l2} = 44 \tag{18}$$

Comparing this with published numbers for these parameters [34], $R_{phy} = 152, MSHR_{l1} = 16, MSHR_{l2} = 48$, we see that MPAM predicts $MSHR_{l1}$ with good accuracy. The gap between the published $MSHR_{l2}$ and the derived $MSHR_{l2}$ is the actual effectiveness of the hardware prefetchers on STREAM giving $\alpha = 28/32 = 0.88$.

4.5.2 *AWS Graviton 3.* : We use a similar process as for Intel 8488C on the AWS Graviton 3 platform, which is based on the ARM Neoverse V1 architecture. The STREAM[42] results for forward and shuffle scan are shown in Figure 7. We list the least square fit equations and the extracted parameters below. The STREAM benchmark flattens at $\sim 85\%$ of the maximum parallelism of the DRAM system (8 channels, DDR5-4400).

$$MLP_f = 70n_c \tag{19}$$
$$MLP_s = 27n_c \tag{20}$$

The derived parameters are,

$$MSHR_{l1} = 27 \tag{21}$$
$$MSHR_{l2} = 70 \tag{22}$$

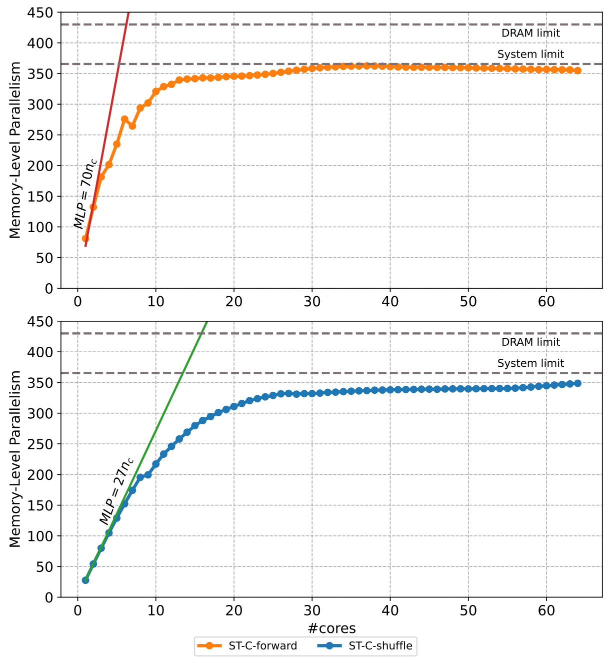


Figure 7: Memory Parallelism in AWS Graviton 3

The exact values of these parameters are not public information since these are cores customized for AWS. However, MPAM is able to provide a model for how to estimate these parameters to decide which platforms might be best suited for the memory parallelism needs of the target applications.

5 UPDOWN ARCHITECTURE

Next, we present UpDown: a novel architecture for unlimited memory parallelism. Unlimited Memory Parallelism refers to UpDown's ability to generate very high (virtually unlimited) outstanding memory requests, constrained only by the memory system that services these requests. The UpDown architecture implements the UpDown Instruction Set Architecture (UpDown-ISA [14]) in a single UpDown lane. Multiple lanes (up to 64) are combined to create the UpDown accelerator. Each UpDown lane in the UpDown accelerator enables a high memory parallelism with a set of key architectural memory access mechanisms - **Split-transaction DRAM memory requests** and **Compute-Name Synchronization of memory responses** ensure that a single thread is able to exploit memory level parallelism using the large namespaces described in Section 5.3. **Efficient memory parallelism scaling** is achieved within a single UpDown lane using multiple light-weight thread contexts and scale-out across UpDown lanes using hardware parallelism. We describe these mechanisms in detail in Section 5.2 and use MPAM to showcase UpDown's memory performance.

Table 3: MPAM derived architectural parameters for commercial systems [21, 34]

Platform	MPAM derived parameters		Published parameters		
	$MSHR_{l1}$	$MSHR_{l2}$	R_{phy}	$MSHR_{l1}$	$MSHR_{l2}$
Intel Xeon Platinum 8848C	16	44	512	16	48
AWS Graviton 3	27	70	256	-	-

5.1 UpDown Design

The UpDown accelerator connects directly to the DRAM, without going through a deep cache hierarchy as in traditional CPUs, enabling UpDown to access the DRAM as shown in Figure 8. Multiple accelerators can be integrated per HBM stack / DRAM system as illustrated in Figure 8b. A simple controller core is used to schedule and offload work on UpDown .

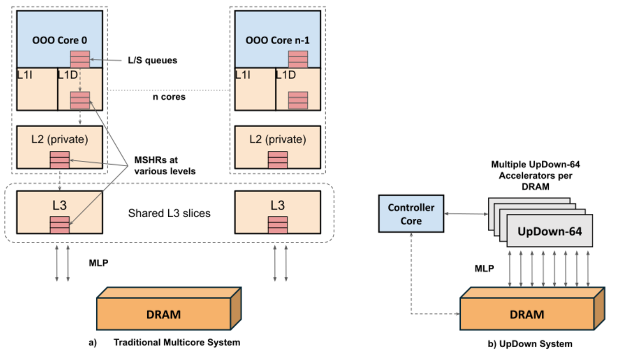


Figure 8: Varying memory access paths between a) Traditional Multicore CPU and b) UpDown System

Each UpDown accelerator contains 64 UpDown lanes (Figure 9). Each UpDown lane is an event-driven programmable accelerator that generates unlimited memory parallelism under software control. Events are first-class primitives in the UpDown ISA. It combines traditional hardware events like memory_return, write_acks, and custom software events with a unified programmable event framework.

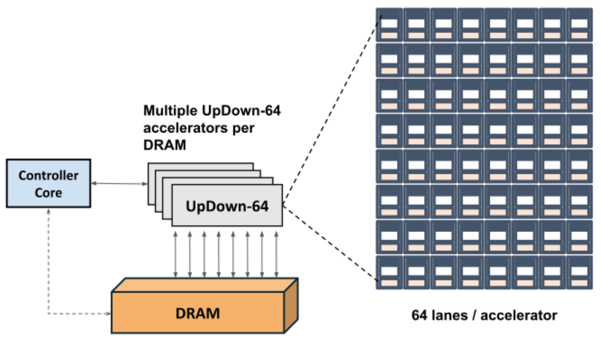


Figure 9: Multiple UpDown lanes are integrated into the memory path controlled by a simple controller core that offloads work on UpDown lanes

Execution in the UpDown lane proceeds through the invocation of events, scheduled using event words in an Event Queue (see Figure 10). Each event will 1) create / activate a thread context, 2) integrate data payload (up to 8 words) using Operand Buffer into the thread context (source of payload could be another lane or CPU core or memory read returns) and 3) execute the instructions corresponding to the event until a `yield, yieldt` instruction is executed. For a full discussion of the instruction set architecture and related event mechanisms, we refer the reader to the the UpDown ISA manual [14].

5.2 Key Mechanisms for Memory Parallelism in UpDown

UpDown enables unlimited memory parallelism through the following key mechanisms:

- **Split Transaction DRAM memory requests:** Unlimited outstanding memory request namespace supported by novel `send` instructions
- **Explicit Compute-Name Synchronization of memory responses:** Unlimited memory response synchronization namespace supported by event-driven execution
- **Efficient Memory Parallelism Scaling:** Scale-up parallelism with many light-weight thread contexts per lane and scale-out parallelism with multiple lanes per accelerator and multiple accelerators per DRAM

These mechanisms build on a number of previous research machine designs [9, 15–17, 47, 66] and are described in detail below.

5.2.1 Split Transaction DRAM memory requests: The UpDown ISA includes a novel set of messaging instructions for split-transaction DRAM memory accesses. The instructions `sendm`, `sendmr`, and `sendmops` (See Table 4) format messages using data from one of three sources 1. registers, 2. scratchpad, or 3. operand buffer to create the payload (1-8 words). Each message also carries a *continuation word* (see Figure 11) that holds synchronization information for memory responses, described in detail below in Section 5.2.2.

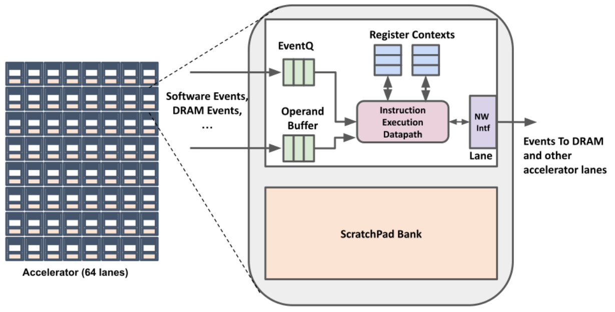

Software Events, DRAM Events, ...
EventQ
Register Contexts
Operand Buffer
Instruction Execution Datapath
NW Intf
Lane
Events To DRAM and other accelerator lanes
ScratchPad Bank
Accelerator (64 lanes)

Figure 10: UpDown Lane Architecture

These messages are pushed out through the network interface (see Figure 10) asynchronously, and instruction execution can proceed beyond the `sendm, sendmr`, or `sendmops` instruction.

Table 4: Novel instructions in UpDown

Instruction(s)	Description
`sendm`	Asynchronous Memory access instruction using buffer in scratchpad
`sendmops`	Asynchronous Memory access instruction using operand buffer
`sendmr`	Asynchronous Memory access instructions using register values
`ev, evi, evr`	Continuation creation instructions
`yield, yieldt`	Thread Management Instructions

As explained in Section 2, traditional architectures require expensive book-keeping of outstanding memory requests. This book-keeping typically limits the outstanding request namespace to a narrow local namespace (e.g., MSHR entry IDs in cache-based architectures, FIFO entry IDs in decoupled access execute architectures, etc.). In contrast, in UpDown, the inclusion of the continuation-word in the message obviates the need for any book-keeping of outstanding memory requests within the lane, resulting in an unlimited outstanding request namespace. Memory ordering and data coherence are handled in software, allowing custom implementations based on individual applications.

Finally, the software control over memory access allows applications to implement custom data movement efficiently. Whereas traditional cache-based architectures rely on hardwired cache policies and prefetch mechanisms to manage locality, these mechanisms prove ineffective for applications with low data reuse. UpDown can manage the locality of data between registers, scratchpad, and DRAM effectively in software.

5.2.2 Explicit Compute-Name Synchronization of memory responses: Event-driven execution in UpDown enables explicit software synchronization of memory responses. UpDown unifies hardware events like memory_response, cache_invalidate, etc. with custom software events and are represented in software using an event word as shown in Figure 11. The event word carries the following information

1. **eventLabel:** The target instruction offset for the thread invocation.
2. **threadID:** Name of the thread context to be used for execution.

(3) **numOperands:** Number of words in the current event's data payload in the operand buffer
(4) **networkID:** Destination target's (UpDown lane) name, where the event should be sent and executed.

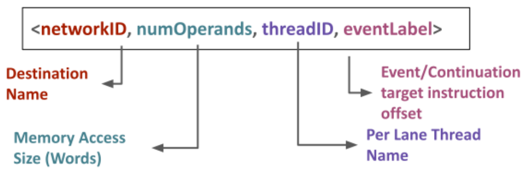

<networkID, numOperands, threadID, eventLabel>
Destination Name
Event/Continuation target instruction offset
Memory Access Size (Words)
Per Lane Thread Name

Figure 11: Event / Continuation Word Fields

While this encoding of events allows a number of powerful thread invocation and synchronization possibilities, we focus on the memory response synchronization. Figure 12 shows the steps to synchronize a memory response. ① Memory response continuation words are enqueued in the EventQ of a lane specified by the **networkID**. ②**eventLabel** is used to compute the next instruction address for execution as an offset from the thread program's base instruction address (Progbase). This multi-way dispatch builds on mechanisms developed and illustrated in [16, 17] ③**threadID** integrates the corresponding register state from the physical registers into the datapath and finally, ④**numOperands** integrates the top numOperands words in the Operand Buffer into the datapath. Following this, execution proceeds until a `yield, yieldt` instruction is executed, at which point the next scheduled event_word is processed.

As shown in Section 2, traditional architectures synchronize memory responses with physical registers in ROB, Reservation Stations, FIFO registers, etc. In contrast, UpDown synchronizes memory responses to a compute name (threadID, eventLabel), allowing an unlimited synchronization namespace. However, explicit software synchronization must be added to programs wherever ordering is needed for correctness. Fortunately, UpDown's synchronization operations are very inexpensive (1 cycle), and in many fine-grained parallel programs there are many unordered memory references.

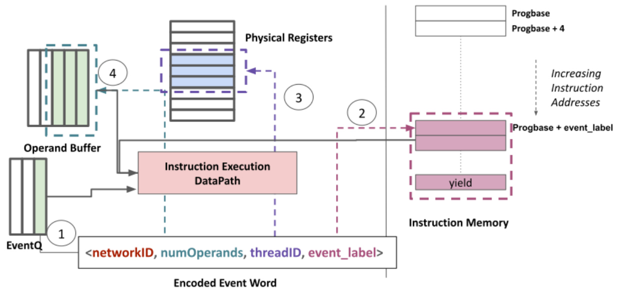


Figure 12: Synchronization using Compute-name (continuation-word) in UpDown .

5.2.3 Efficient Memory Parallelism Scaling: UpDown supports multiple lightweight thread contexts (16 general purpose registers + 8 special registers per thread and up to 128 threads) per lane. These threads are explicitly managed by software control through `yield` and `yieldt` instructions. A `yield` instruction suspends the execution of the current thread context while persisting its register state. A `yieldt` instruction, on the other hand, terminates the execution of the thread and frees up its register state. A new thread context is created when an event_word with special **threadID** 0xFF is processed. While split-transaction memory accesses with compute name synchronization allow unlimited memory parallelism to be generated from a single thread, multiple-thread contexts provide further potential to scale up memory parallelism in applications that have higher data-dependent control flows.

Further scale-out parallelism is provided by hardware parallelism, as shown in Figure 9. At $\sim 0.06mm^2$ per lane, 64 lanes per accelerator and multiple accelerators per DRAM are still only a fraction of the area (~ 4%) of a large OOO multi-core CPU[45] enabling low-cost scale-out memory level parallelism in a system. This scale-out parallelism, as shown in the following section, enables UpDown to achieve memory level parallelism sufficient to saturate an HBM3e stack's bandwidth of 1.2TB/s.

5.3 Memory Level Parallelism analysis on UpDown using MPAM

Using **MPAM**, we describe how we modify the traditional limits on memory level parallelism by using Compute-Names for the Synchronization Namespace (N_{sync}), removing the conventional architectural limits on the Outstanding Namespace N_{out} and getting rid of any shared namespaces N_{sh}. We also present the effectiveness of these mechanisms using a simple scan read-only micro-benchmark.

5.3.1 Synchronization Namespace (N_{sync}): Traditional architectures synchronize memory responses with register names (architectural or physical). Some research machines allocate FIFO names [60], scratchpad locations [71] to synchronize memory responses. By contrast, UpDown uses **Compute Names** represented by event_words as described in Section 5.2.2. Figure 11 shows the representation of the compute name that allows synchronization into a specific instruction address (eventLabel), thread (threadID), and compute resource (networkID). This gives us N_{sync} as follows

$$N_{sync} = 2^{b_{tid}+b_{nwid}+b_{elabel}} \tag{23}$$

where b_{tid}, b_{nwid} and b_{elabel} are the number of bits used to represent the threadID, networkID, and eventLabel respectively. Additionally, since there can be any number of invocations of a compute structure (events in UpDown), repetitions of these names are possible in the in-flight memory requests, leading to the possibility of a very large synchronization namespace.

5.3.2 Outstanding Request Namespace (N_{out_sync}, N_{out_async}): In UpDown, all outstanding requests are software controlled and hence $N_{out\_async} = 0$. In traditional architectures, book-keeping for in-flight memory requests is done using structures like load/store buffers, MSHRs in caches, etc. However, using split-transaction memory accesses (described in Section 5.2.1) and allowing software-controlled consistency, UpDown avoids the need to bind resources to outstanding requests for book-keeping. This allows the entire memory address space to be used as the outstanding request namespace.

$$N_{out\_sync} = 2^{b_{mem}} \quad (24)$$

where b_{mem} represents the number of bits used to represent the address space. Additionally, as with N_{sync} there can be any number of in-flight requests to the same address leading to the possibility of a very large outstanding namespace.

5.3.3 Shared Request Namespace (N_{sh}): Finally, in UpDown the memory path from each compute resource (UpDown lane) does not include any shared namespace restrictions.

Using Eqns:24, 23, and substituting in Eqn:2, we get

$$MLP = (1/cl) \cdot L \cdot n_{rep} \cdot s_{op}(min(2^{b_{tid}+b_{nwid}+b_{elabel}}, 2^{b_{mem}})) \quad (25)$$

where L is the number of UpDown lanes, n_{rep} is the average number of repetitions to the same address, s_{op} is the average size of a memory request. As can be seen, Eqn:25 can be virtually unlimited. When connected to a specific DRAM system, the queues in the DRAM system and the issue rate of memory references per lane will determine the total memory parallelism that the whole system can support.

We evaluate the impact of the UpDown's memory access mechanisms using a simple read-only microbenchmark that does a scan read of a contiguous block of memory in DRAM.

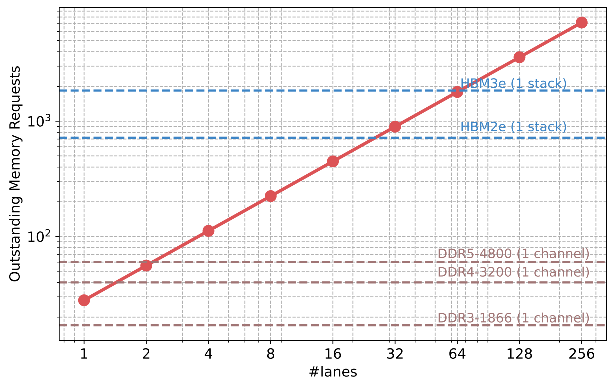


Figure 13: Memory Parallelism with scan read microbenchmark running on multiple UpDown lanes. Horizontal lines show memory parallelism required for various memory systems assuming 100ns latency

We perform this experiment with the setup mentioned in Section 6 running each lane at 2GHz. We also plot the memory parallelism required (see Figure 13) to saturate the bandwidth available on various DDR and HBM memory systems using a uniform 100ns memory latency (using [38]). As can be seen, 1 lane can saturate a single DDR3-1866 channel (14.9GB/s), and 64 lanes (1 UpDown accelerator) can saturate a HBM3e stack (1.2TB/s / 1,673 requests). In contrast, a Sapphire Rapids system with 56 cores achieves about 240GB/s or 590GB/s with a DDR5-4800 or HBM2e memory system [43].

6 METHODOLOGY

6.1 Workloads and Datasets

For evaluation of the UpDown's mechanisms, we use a number of memory-intensive workloads, as shown in Table 5.

Table 5: Workloads

Workloads	Dataset
STREAM microbenchmark [42] (ST-C) (ST-S) (ST-A) (ST-T)	Three arrays with 16Mi 64-bit floating-point entries STREAM-Copy STREAM-Scale STREAM-Add STREAM-Triad
Top-k (TK)	One array with 16Mi 128-bit (64-bit unsigned integer, 64-bit floating point) key-value pairs; $k = 8$
Dot Product (DP)	Two arrays with 16Mi 64-bit floating point entries each
Image Processing (HIST)	One array with 32Mi 64-bit unsigned integer entries
Sparse Matrix Vector Product (SpMV)	Square matrix of dimension 12,288 and 4% uniformly distributed nonzeros in CSR format (64-bit unsigned integer, 64-bit floating point)
Fast Fourier Transform (FFT)	One array with 16Mi 128-bit (64-bit floating point, 64-bit floating point) complex numbers
Image Filtering (CNN)	One 3074×3074 2-D array with 9.45Mi 64-bit floating points; One 3×3 convolution filter with nine 64-bit floating points

6.2 Simulation Model and Configurations

We use gem5 [40] and DRAMsim3 [37] for cycle-level simulations to evaluate UpDown's performance and memory level parallelism. We developed a cycle-accurate model of UpDown and integrated it into the gem5 infrastructure. To compare UpDown against traditional cores, we use the In-Order and Out-of-Order implementations available in gem5 with the key configurations of the cores listed in Table 6. The OOO-core settings are tuned to match STREAM measurements on Sapphire Rapids GoldenCove cores. The configurations of the caches used and the MSHR entries are also listed in Table 6. For configurations with prefetch enabled, the prefetcher is enabled in the L2 [28]. These latencies were measured on commercial platforms using [58, 70].

Table 6: System Components

Component	Specification
In-Order Core	Simple InOrder core (gem5 MinorCPU) @2GHz
OOO Core	x86 OOO core (GoldenCove configuration using gem5 O3CPU) @2GHz 280 Physical Registers, 512 ROB entries 192 LoadBuff entries, 114 StoreBuff entries
Caches	L1: 32KB ICache, 64KB DCache per core (16 MSHR entries, 8 slots per entry), L2: 1MB unified per core (private) (32 MSHR entries, 8 slots per entry) L3: 2MB per core (shared) (32 MSHR entries, 8 slots per entry)
UpDown Accelerator	64 UpDown Lanes + 4MB scratchpad (64KB per lane) @2GHz
DRAM Memory	HBM2e @460GB/s (1 stack, 8x channels), 100ns latency

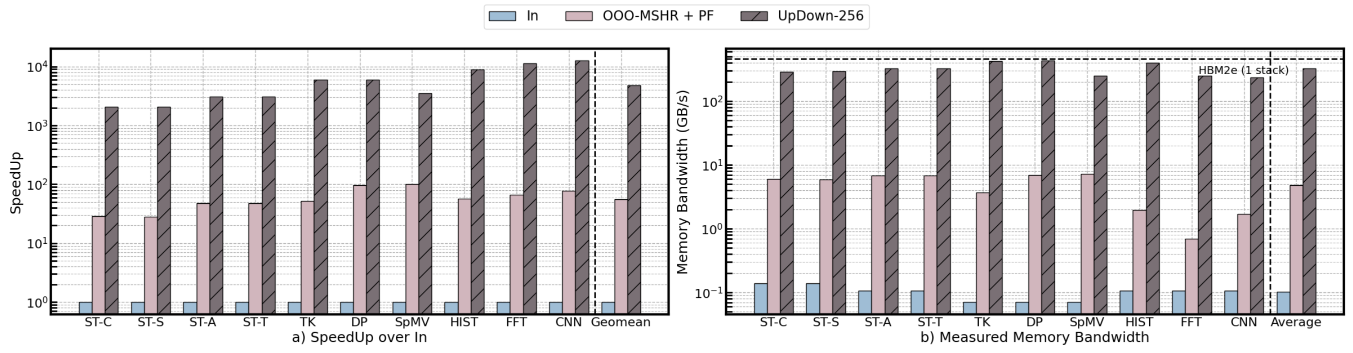


Figure 14: a) SpeedUp of OOO-MSHR+PF, UpDown-256 over In, b) Measured Memory BW of In, OOO-MSHR+PF and UpDown-256

Table 7: Experiment Configurations and Metrics

Configurations	Description
IN	In-Order Load/Store Core, No Caches, Word-Sized Memory Accesses
OOO	OOO Load/Store Core, No Caches, Word-Sized Memory Accesses
OOO-MSHR	OOO + L1/L2 Caches, Cache-Block sized Memory Accesses
OOO-MSHR + PF	OOO-MSHR + Prefetch enabled
OOO-MSHR + PF-32	32 OOO-MSHR + PF cores, private L1/L2 caches, shared L3 cache
UpDown-1	UpDown lane - CPU (Core+caches) + 1 UpDown Lane
UpDown-64	UpDown accelerator - CPU (Core+caches) + 64 UpDown Lanes
UpDown-256	Scale-out UpDown - CPU (Core+caches) + 256 UpDown Lanes
Metric	**Definition**
Runtime (seconds)	Execution time
SpeedUp	Relative Performance
Memory Bandwidth (GB/s)	Memory Traffic (GB) / Runtime (s)
Memory Parallelism	No of outstanding memory requests (Memory Bandwidth (GB/s) / access_size (B) * latency (ns))

7 EVALUATION

We evaluate the memory access mechanisms presenting the overall performance of the workloads in Table 5 on the various configurations in Table 7. We relate this performance to the memory level parallelism in each system and the resultant memory bandwidth utilization, showing how UpDown is able to achieve extremely high memory parallelism at low costs.

We use data from [31] and [45] for area costs of the In-Order and OOO cores. For UpDown, synthesis using Synopsys [1] tools on 28nm libraries and projecting it to 7nm using [10, 64, 65], gives $0.06mm^2$ per UpDown lane. Given this, UpDown-1 is comparable in area to a single In-Order core, and UpDown-256 is comparable to a single Out-of-order core.

7.1 Performance relative to In-Order core

We first present the overall runtime speedup over an In-Order Core and the corresponding memory bandwidths in Figure 14a and Figure 14b respectively. UpDown-256 achieves **4,804x** geomean speedup over a single In-Order core. In contrast, the OOO-core with $l1$, $l2$, and $l3$ caches and prefetch enabled achieves 55x geomean speedup over the In-Order core (with a maximum of 98x). UpDown's novel memory access mechanisms enable it to achieve high speedups, demonstrating that UpDown-1 is **18x** more efficient, on average, in memory parallelism than the In-Order core.

Figure 14b reveals the close relationship between performance and achieved memory bandwidths. On applications like **TK** and **DP**, UpDown-256 nearly saturates the HBM2e bandwidth at $\sim$ 428 GB/s. On average, UpDown-256 achieves 322 GB/s across all applications.

7.2 Performance relative to OOO cores and cache mechanisms

Next, we compare the performance of UpDown-1 and UpDown-256 to OOO, OOO-MSHR, and OOO-MSHR + PF configurations. Figure 15a shows that a single UpDown lane (UpDown-1) achieves **1.7x** geomean speedup over OOO and **2.4x** geomean speedup over OOO-MSHR+PF. With hardware parallelism, UpDown-256 achieves **51x** geomean speedup over OOO and **74x** over OOO-MSHR + PF.

Figure 15a emphasizes the fact that applications with low data reuse, like TK and DP, see minimal benefit from the addition of a cache or enabling prefetch (OOO $\rightarrow$ OOO-MSHR $\rightarrow$ OOO-MSHR + PF). In contrast, UpDown's memory access mechanisms enable even a single lane UpDown-1 to achieve performance comparable to OOO-MSHR + PF across all applications irrespective of the level of data reuse. Figure 15b confirms the performance benefits being directly related to the higher bandwidths achieved in these applications. It is instructive to note that even on applications like FFT and CNN that benefit from caching in traditional systems, UpDown-256 achieves high performance by offsetting cache benefits with higher memory parallelism. This highlights the alternate path to high performance - utilizing higher memory bandwidths more effectively (like in HBMs) as opposed to using hardwired caching mechanisms to reduce latency.

7.3 Scalability

We next analyze memory-parallelism scalability with an increasing number of cores. Figure 16 shows the impact of increasing the number of cores (lanes in UpDown) on Memory Parallelism using STREAM [42] (COPY). With OOO-MSHR+PF, the memory parallelism quickly saturates beyond 20 cores due to the namespace limits in the shared $MSHR_{l3}$. UpDown on the other hand (with 64 lanes) is able to hit $\sim$ 438 outstanding requests on HBM2e, which

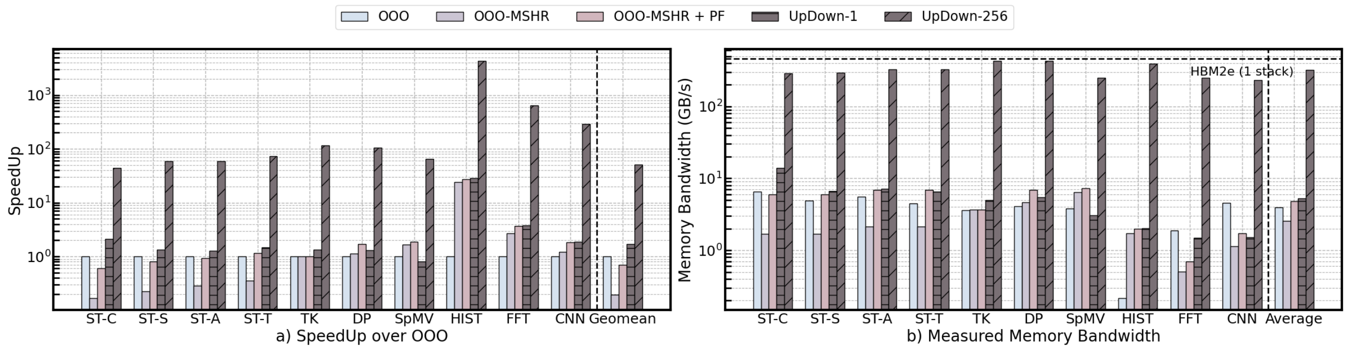


Figure 15: a) SpeedUp of UpDown-1 and UpDown-256 over OOO, OOO-MSHR, OOO-MSHR+PF, b) Measured Memory BW on OOO, OOO-MSHR, OOO-MSHR+PF, UpDown-1 and UpDown-256

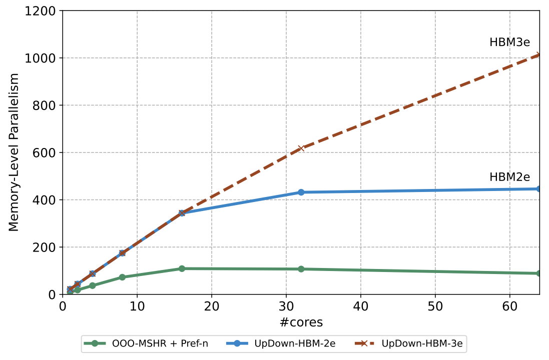


Figure 16: Memory Parallelism vs #cores on ST-C

is > 63% of the maximum required to saturate it. With HBM3e, on ST-C, UpDown is able to achieve ~ 1100 outstanding requests, which is > 59% of the maximum. In both cases, unlike the scan-read benchmark in Figure 13, STREAM is limited by the switching between reads and writes and the write bandwidth limits.

7.4 Correlation between Performance and Memory Parallelism

The correlation between performance and memory parallelism on all configurations under evaluation is shown in Figure 17. We also plot the MPAM-derived limits for the baseline systems (MLP_{IN}, $MLP_{OOO-MSHR+PF}$, $MLP_{OOO-MSHR+PF-32}$). These limits divide the space into 4 illustrative bands.

The first band, occupied by IN, is a low memory parallelism region. The memory parallelism achievable on a single In-order core does not exceed 1. The second band is occupied by UpDown-1 and OOO-MSHR+PF. This clearly shows the impact of the memory access mechanisms in UpDown and the fact that removing the namespace constraints in the architecture provides the opportunity to achieve extremely high memory parallelism at low costs. UpDown-1 is able to extract memory parallelism comparable to and up to **3.5x** more than a sophisticated OOO core (nearly **22** outstanding requests in ST-C), while being ~ 250x smaller in area. The third band shows the effect of hardware parallelism with OOO-MSHR+PF

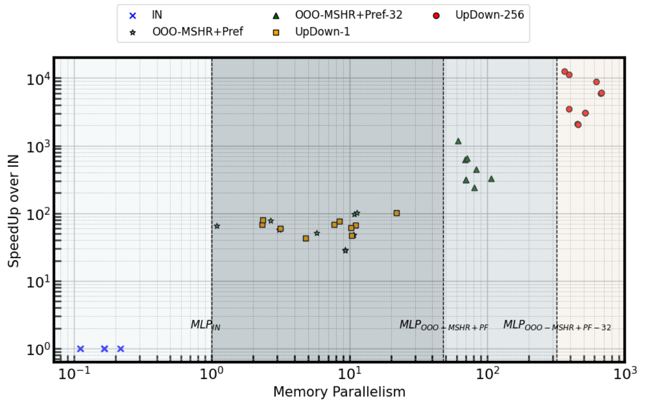


Figure 17: SpeedUp vs Memory Parallelism on In, OOO-MSHR+Pref, OOO-32, UpDown-1 and UpDown-256

cores. For a high data-reuse application like CNN, increasing the number of cores results in a nearly linear performance benefit of 31x. However, for other applications, performance uplifts are limited (< 19x) by the memory parallelism limits of the shared L3 cache (N_{sh}) and its finite resource-bound miss-handling mechanisms.

The fourth band in Figure 17 shows the high performance and corresponding memory parallelism achieved in UpDown-256 generating up to **670** outstanding references. UpDown-256, comparable in area to OOO-MSHR+PF, achieves **26-79x** higher memory parallelism with the potential to scale-out more (Figure 13).

7.5 Cost of Memory Parallelism

Finally, we compare the cost of different approaches to memory parallelism. Book-keeping and tracking of in-flight memory requests are done using Content-Addressable Memories (CAM) [24] at various levels in the memory hierarchy - the Load/Store queues, MSHRs across the cache hierarchy, write buffers, and so on. For every request sent to the next level in the memory hierarchy, an associative check is performed against all the buffer/memory entries in the current level, typically using the physical memory address. When responses arrive at a particular level of memory, a similar associative check is repeated to update the state and data. Thus, a CAM

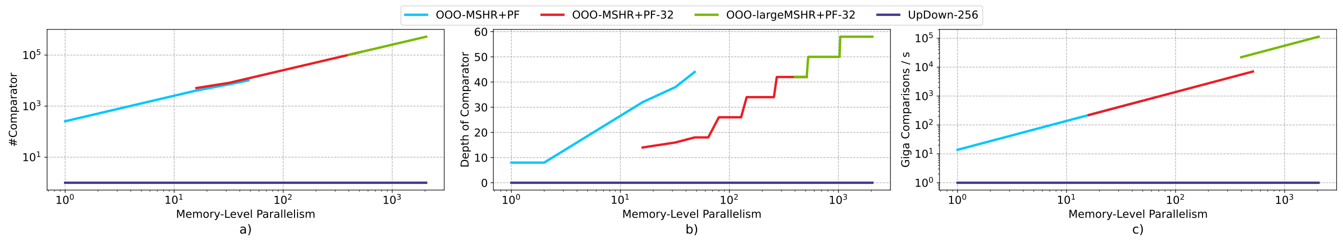


Figure 18: Comparing the Cost of Memory Parallelism of various OOO cores and UpDown (the lower the better) a) Number of Comparators vs Memory Parallelism ; b) Depth of Comparators vs Memory-Level Parallelism ; c) GigaComparisons/second vs Memory Parallelism

with n words requires n-way comparators with $log_2 n$ depth match address encoder(depth)[24]. We use this idea for a notional area and power comparison between OOO-MSHR+PF, OOO-MSHR+PF-32, and UpDown for memory parallelism. We also compare against a hypothetical configuration (OOO-largeMSHR+PF-32) with large enough CAMs to meet the required memory parallelism. [1]

Figure 18a and 18b highlight the trend in the number of comparators (cumulative over all structures - Load/Store queue, MSHR entries in l1, l2, and l3) while increasing the system's memory parallelism. Figure 18b, similarly, shows an increase in the depth of the comparator (encoder depth) against memory parallelism [24, 67, 75]. The two figures show that the cost of the comparator increases linearly in area to support the increase in outstanding memory requests. Increasing the number of comparators has the additional effect of increasing latency in producing the match result. Nominal inverter delays at 7nm are $\sim$ 20ps ([49]), leading to about 10 levels of logic per cycle (3.5GHz freq). Figure 18b shows that having very large MSHRs can increase the depths $>$ 50 levels, adding $\sim$ 12 additional cycles to memory latency, clearly demonstrating the limitation in scaling CAM structures. In contrast, because UpDown does not track or book-keep the outstanding requests and implements consistency in software, no dedicated hardware comparators are needed, and increasing memory parallelism has no additional cost.

Beyond the quantity of hardware, these comparators need to be run at a high rate - we estimate the giga-comparisons/second required to achieve a given memory bandwidth (see Figure 18c). These giga-comparisons (and associated data movement) reflect the power consumption required to sustain that memory parallelism. The results show that $\sim$ 10,000 giga-comparison per second (10 tera-comparisons/second) are required to reach HBM2e bandwidths. Again, no such operation rates are needed in UpDown .

Current OOO cores cannot saturate a single HBM2e stack. 32 or more OOO-MSHR+PF cores with an overhead $>$ 10,000x comparators in trees with depth $>$ 40 are required to match the requirement for HBM2e bandwidths. We also show a hypothetical OOO core with an extremely large MSHR (OOO_largeMSHR+PF-32) to estimate overhead to achieve HBM3e (1.2TB/s) bandwidths. As can be seen $>$ 1,000,000 comparators arranged in depths of $>$ 60 would be required to achieve and sustain HBM3e bandwidths.

[1]We ignore the match line sense amplifier and search word register area and power as an approximation compared to the area and power of the CAM cells and the encoder

8 RELATED WORK

8.1 Hardware Prefetchers

Hardware prefetching mechanisms provide additional memory parallelism in an asynchronous manner (N_{out_async}) by predicting the access patterns. A number of custom prefetchers [46, 54, 73, 74] have been proposed for irregular applications as well that are customized to specific applications. This is in addition to the large body of research that exists in this space [20, 28, 29]. The lack of uniform data access patterns narrows the effectiveness of prefetchers to a subset of applications that benefit from the predictive cacheline fills. Additionally, prefetchers compete for the same resources as the load/store instructions - namely memory bandwidth and cache storage. Previous studies [36, 39] have shown the negative impact on performance due to competitive thrashing between demand miss fills and prefetch fills and potential reduction in usable memory bandwidth. UpDown, on the other hand, provides applications the ability to achieve the benefits of prefetching using software and lightweight threads. As our experiments have shown, UpDown can almost fully utilize any given memory system, fetching only useful data.

8.2 Multi-Threaded Architectures

A number of processors with various forms of hardware multithreading have been proposed and built over the last couple of decades [5, 6, 8, 19]. From having hardware threads switching at every cycle as in barrel processors [44] to SIMT-style threads in GPUs [41] that switch out a warp of 32 threads at long-latency memory accesses, these mechanisms attempt to generate sufficient memory requests to saturate the available bandwidth using a large number of threads. All the thread-switching is controlled by hardware. To support this, the size of the N_{sync} is multiplied by the number of thread contexts required. For GPUs, N_{sync} is further expanded by allowing the shared local memory also to be used as a synchronization space. However, this increase in N_{sync} comes at the cost of excessive registers ($>$ 2K per thread [3]). The outstanding and shared request namespaces remain the same. UpDown with their lightweight threads that are software-controlled allow runahead execution inherently, beyond memory reference instructions until an explicit `yield, yieldt` instruction is encountered. This gives greater control over the memory accesses per thread, enabling higher memory parallelism per thread.

8.3 Decoupled Access Execute Architectures

Modern incarnations of the original Decoupled Access Execute architectures [60] have been proposed in an effort to overcome some of the limitations in the original idea [48, 71, 72]. These architectures need sophisticated compilers that can sufficiently decouple the access and execute programs to achieve good performance on the decoupled version. Further, recent proposals have suggested the use of OOO cores to replace the in-order cores in [60]. This leads to complex and expensive mechanisms to seamlessly support mis-prediction, mis-speculation rollback mechanisms requiring synchronization between the 2 processors. Finally, as an approach to achieve higher memory parallelism shown in Section 4, they are still limited by the sizes of the queues used to book-keep outstanding requests and synchronize responses. Scaling up to the higher memory bandwidths of HBM2e/3e will require allocating and managing larger queues or more instances to support higher memory parallelism. UpDown, on the other hand, is only limited by the issue rate of memory references and can scale up to meet the requirements of higher bandwidth systems.

9 CONCLUSION

We defined the Memory Parallelism Abstract Machine (MPAM) to model the limits of memory parallelism in different architectures and characterized the limits on key architectures. To the best of our knowledge, this is the first work that highlights the multiple local namespaces as limits to memory-level parallelism in a concise model. We applied MPAM to 4 commercial CPUs to characterize their design and performance. MPAM can be used by architects to either understand the various limits in a system for memory parallelism or to design future systems with specific memory parallelism requirements. We described the UpDown: a novel architecture for unlimited parallelism building on insights from MPAM. A single UpDown lane achieves **3.5x** more memory parallelism than a sophisticated OOO core. We also showed that UpDown can also scale-up and scale-out cost-effectively to saturate an HBM3e stack's memory bandwidth $\sim$ 1.2TB/s.

ACKNOWLEDGEMENTS

This research is based upon work supported by the Office of the Director of National Intelligence (ODNI), Intelligence Advanced Research Projects Activity (IARPA), through the Advanced Graphical Intelligence Logical Computing Environment (AGILE) research program, under Army Research Office (ARO) contract number W911NF22C0082. 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 the ODNI, IARPA, or the U.S. Government.

This work is also supported in part by NSF Grant CNS-1907863 and a Computation Innovation Fellows Award. Thanks also to the entire UChicago UpDown team, and also our collaborators at Purdue University and Tactical Computing Laboratories.

REFERENCES

[1] [n. d.]. Synopsys teaching resources. https://www.synopsys.com/community/university-program/teaching-resources.html.

[2] 2022. AGILE: ADVANCED GRAPHIC INTELLIGENCE LOGICAL COMPUTING ENVIRONMENT Program. https://www.iarpa.gov/research-programs/agile.

[3] 2023. NVIDIA H100 Tensor Core GPU Architecture. *NVIDIA* (2023). https://resources.nvidia.com/en-us-tensor-core

[4] 2024. Intel® 64 and IA-32 Architectures Optimization Reference Manual Volume 1. *Intel* (2024). https://cdrdv2.intel.com/v1/dl/getContent/671488

[5] Sriram Aananthakrishnan, Nesreen K. Ahmed, Vincent Cave, Marcelo Cintra, Yigit Demir, Kristof Du Bois, Stijn Eyerman, Joshua B. Fryman, Ivan Ganev, Wim Heirman, Hans-Christian Hoppe, Jason Howard, Ibrahim Hur, MidhunChandra Kodiyath, Samkit Jain, Daniel S. Klowden, Marek M. Landowski, Laurent Montigny, Ankit More, Przemyslaw Ossowski, Robert Pawlowski, Nick Pepperling, Fabrizio Petrini, Mariusz Sikora, Balasubramanian Seshasayee, Shaden Smith, Sebastian Szkoda, Sanjaya Tayal, Jesmin Jahan Tithi, Yves Vandriessche, and Izajasz P. Wrosz. 2020. PIUMA: Programmable Integrated Unified Memory Architecture. http://arxiv.org/abs/2010.06277 arXiv:2010.06277 [cs].

[6] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith. 1990. The Tera computer system. In *Proceedings of the 4th International Conference on Supercomputing* (Amsterdam, The Netherlands) *(ICS '90)*. Association for Computing Machinery, New York, NY, USA, 1–6. https://doi.org/10.1145/77726.255132

[7] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith. 1990. The Tera computer system. In *Proceedings of the 4th International Conference on Supercomputing*. 1–6.

[8] Wendell Anderson, Robert Rosenberg, and Marco Lanzagorta. 2003. Experiences Using the Cray Multi-Threaded Architecture (MTA-2). In *Proceedings of the 2003 DoD User Group Conference (DOD_UGC '03)*. IEEE Computer Society, USA, 378.

[9] Shekhar Borkar, Robert Cohn, George Cox, Thomas Gross, H. T. Kung, Monica Lam, Margie Levine, Brian Moore, Wire Moore, Craig Peterson, Jim Susman, Jim Sutton, John Urbanski, and Jon Webb. 1990. Supporting Systolic and Memory Communication in IWarp. In *Proceedings of the 17th Annual International Symposium on Computer Architecture* (Seattle, Washington, USA) *(ISCA '90)*. Association for Computing Machinery, New York, NY, USA, 70–81. https://doi.org/10.1145/325164.325116

[10] David Brooks. 2018. What's the future of technology scaling? https://www.sigarch.org/whats-the-future-of-technology-scaling/

[11] Tien-Fu Chen and Jean-Loup Baer. 1995. Effective hardware-based data prefetching for high-performance processors. *IEEE Trans. Comput.* 44, 5 (1995), 609–623. https://doi.org/10.1109/12.381947

[12] George Cozma Chester Lam. 2022. AMD's Zen 4, Part 2: Memory Subsystem and Conclusion. *Chips and Cheese* (November 2022). https://chipsandcheese.com/2022/11/08/amds-zen-4-part-2-memory-subsystem-and-conclusion/

[13] Daniel Chiang. 2024. SK Hynix to reveal 16-layer HBM3E to main the lead. *DigiTimesAsia* (February 2024). https://www.digitimes.com/news/a20240222PD215/sk-hynix-hbm-dram-2024-production.html

[14] Andrew Chien, Andronicus Rajasukumar, Marziyeh Nourian, Yuqing Wang, Tianshuo Su, Chen Zou, and Yuanwei Fang. 2024. *UpDown Accelerator Instruction Set Architecture (ISA) v2.4*. Technical Report TR-2024-03. University of Chicago. https://newtraell.cs.uchicago.edu/research/publications/techreports/TR-2024-03

[15] W. J. Dally, L. Chao, A. Chien, S. Hassoun, W. Horwat, J. Kaplan, P. Song, B. Totty, and S. Wills. 1987. Architecture of a Message-Driven Processor. In *Proceedings of the 14th Annual International Symposium on Computer Architecture* (Pittsburgh, Pennsylvania, USA) *(ISCA '87)*. Association for Computing Machinery, New York, NY, USA, 189–196. https://doi.org/10.1145/30350.30372

[16] Yuanwei Fang, Tung T. Hoang, Michela Becchi, and Andrew A. Chien. 2015. Fast Support for Unstructured Data Processing: The Unified Automata Processor. In *Proceedings of the 48th International Symposium on Microarchitecture* (Waikiki, Hawaii) *(MICRO-48)*. Association for Computing Machinery, New York, NY, USA, 533–545. https://doi.org/10.1145/2830772.2830809

[17] Yuanwei Fang, Chen Zou, Aaron J. Elmore, and Andrew A. Chien. 2017. UDP: a programmable accelerator for extract-transform-load workloads and more. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-50 '17)*. Association for Computing Machinery, New York, NY, USA, 55–68. https://doi.org/10.1145/3123939.3123983

[18] K. I. Farkas and N. P. Jouppi. 1994. Complexity/performance tradeoffs with non-blocking loads. In *Proceedings of the 21st Annual International Symposium on Computer Architecture* (Chicago, Illinois, USA) *(ISCA '94)*. IEEE Computer Society Press, Washington, DC, USA, 211–222. https://doi.org/10.1145/191995.192029

[19] John Feo, David Harper, Simon Kahan, and Petr Konecny. 2005. ELDORADO. In *Proceedings of the 2nd Conference on Computing Frontiers* (Ischia, Italy) *(CF '05)*. Association for Computing Machinery, New York, NY, USA, 28–34. https://doi.org/10.1145/1062261.1062268

[20] John WC Fu, Janak H Patel, and Bob L Janssens. 1992. Stride directed prefetching in scalar processors. *ACM SIGMICRO Newsletter* 23, 1-2 (1992), 102–110.

[21] Chester Lam George Cozma. 2022. Graviton 3: First Impressions. *Chips and Cheese* (May 2022). https://chipsandcheese.com/2022/05/29/graviton-3-first-impressions/

[22] Kourosh Gharachorloo. [n. d.]. MEMORY CONSISTENCY MODELS FOR SHARED-MEMORY MULTIPROCESSORS. ([n. d.]).

[23] Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. 1992. Hiding memory latency using dynamic scheduling in shared-memory multiprocessors. *ACM SIGARCH Computer Architecture News* 20, 2 (May 1992), 22–33. https://doi.org/10.1145/146628.139678

[24] Karl-Erwin Grosspietsch. 1992. Associative processors and memories: A survey. *IEEE Micro* 12, 3 (1992), 12–19.

[25] Tae Jun Ham, Juan L. Aragón, and Margaret Martonosi. 2015. DeSC: Decoupled supply-compute communication management for heterogeneous architectures. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 191–203. https://doi.org/10.1145/2830772.2830800

[26] John L. Hennessy and David A. Patterson. 2012. *Computer Architecture: A Quantitative Approach* (5 ed.). Morgan Kaufmann, Amsterdam.

[27] Zhigang Hu, Margaret Martonosi, and Stefanos Kaxiras. 2003. TCP: Tag Correlating Prefetchers. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture (HPCA '03)*. IEEE Computer Society, USA, 317.

[28] Z. Hu, M. Martonosi, and S. Kaxiras. 2003. TCP: tag correlating prefetchers. In *The Ninth International Symposium on High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings*. 317–326. https://doi.org/10.1109/HPCA.2003.1183549

[29] Sorin Iacobovici, Lawrence Spracklen, Sudarshan Kadambi, Yuan Chou, and Santosh G. Abraham. 2004. Effective stream-based and execution-based data prefetching. In *Proceedings of the 18th Annual International Conference on Supercomputing* (Malo, France) *(ICS '04)*. Association for Computing Machinery, New York, NY, USA, 1–11. https://doi.org/10.1145/1006209.1006211

[30] Hongshin Jun, Jinhee Cho, Kangseol Lee, Ho-Young Son, Kwiwook Kim, Hanho Jin, and Keith Kim. 2017. HBM (High Bandwidth Memory) DRAM Technology and Architecture. In *2017 IEEE International Memory Workshop (IMW)*. 1–4. https://doi.org/10.1109/IMW.2017.7939084

[31] Jack Kang. 2017. SiFive U54-MC Coreplex: Multicore, 64-bit Application Processor class RISC-V CPU. https://static.dev.sifive.com/slides/SiFive-U54-MC.pdf

[32] David Kroft. 1998. Lockup-free instruction fetch/prefetch cache organization. In *25 years of the international symposia on Computer architecture (selected papers)*. ACM, Barcelona Spain, 195–201. https://doi.org/10.1145/285930.285979

[33] Kartik Lakhotia, Laura Monroe, Kelly Isham, Maciej Besta, Nils Blach, Torsten Hoefler, and Fabrizio Petrini. 2024. PolarStar: Expanding the Horizon of Diameter-3 Networks. In *Proceedings of the 36th ACM Symposium on Parallelism in Algorithms and Architectures* (Nantes, France) *(SPAA '24)*. Association for Computing Machinery, New York, NY, USA, 345–357. https://doi.org/10.1145/3626183.3659975

[34] Chester Lam. 2021. Popping the Hood on Golden Cove. *Chips and Cheese* (December 2021). https://chipsandcheese.com/2021/12/02/popping-the-hood-on-golden-cove/

[35] Chester Lam. 2022. Sunny Cove: Intel's Lost Generation. *Chips and Cheese* (June 2022). https://chipsandcheese.com/2022/06/07/sunny-cove-intels-lost-generation/

[36] 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 (mar 2012), 29 pages. https://doi.org/10.1145/2133382.2133384

[37] Shang Li, Zhiyuan Yang, Dhiraj Reddy, Ankur Srivastava, and Bruce Jacob. 2020. DRAMsim3: A Cycle-Accurate, Thermal-Capable DRAM Simulator. *IEEE Computer Architecture Letters* 19, 2 (2020), 106–109. https://doi.org/10.1109/LCA.2020.2973991

[38] John D. C. Little. 1961. A Proof for the Queuing Formula: L = λW. *Oper. Res.* 9, 3 (jun 1961), 383–387. https://doi.org/10.1287/opre.9.3.383

[39] Gabriel H. Loh. 2008. 3D-Stacked Memory Architectures for Multi-core Processors. In *Proceedings of the 35th Annual International Symposium on Computer Architecture (ISCA '08)*. IEEE Computer Society, USA, 453–464. https://doi.org/10.1109/ISCA.2008.15

[40] Jason Lowe-Power, Abdul Mutaal Ahmad, Ayaz Akram, Mohammad Alian, Rico Amslinger, Matteo Andreozzi, Adrià Armejach, Nils Asmussen, Brad Beckmann, Srikant Bharadwaj, Gabe Black, Gedare Bloom, Bobby R. Bruce, Daniel Rodrigues Carvalho, Jeronimo Castrillon, Lizhong Chen, Nicolas Derumigny, Stephan Diestelhorst, Wendy Elsasser, Carlos Escuin, Marjan Fariborz, Amin Farmahini-Farahani, Pouya Fotouhi, Ryan Gambord, Jayneel Gandhi, Dibakar Gope, Thomas Grass, Anthony Gutierrez, Bagus Hanindhito, Andreas Hansson, Swapnil Haria, Austin Harris, Timothy Hayes, Adrian Herrera, Matthew Horsnell, Syed Ali Raza Jafri, Radhika Jagtap, Hanhwi Jang, Reiley Jeyapaul, Timothy M. Jones, Matthias Jung, Subash Kannoth, Hamidreza Khaleghzadeh, Yuetsu Kodama, Tushar Krishna, Tommaso Marinelli, Christian Menard, Andrea Mondelli, Miquel Moreto, Tiago Mück, Omar Naji, Krishnendra Nathella, Hoa Nguyen, Nikos Nikoleris, Lena E. Olson, Marc Orr, Binh Pham, Pablo Prieto, Trivikram Reddy, Alec Roelke, Mahyar Samani, Andreas Sandberg, Javier Setoain, Boris Shingarov, Matthew D. Sinclair, Tuan Ta, Rahul Thakur, Giacomo Travaglini, Michael Upton, Nilay Vaish, Ilias Vougioukas, William Wang, Zhengrong Wang, Norbert Wehn, Christian Weis, David A. Wood, Hongil Yoon, and Éder F. Zulian. 2020. The gem5 Simulator: Version 20.0+. https://doi.org/10.48550/arXiv.2007.03152 arXiv:2007.03152 [cs].

[41] David Luebke. 2008. CUDA: Scalable parallel programming for high-performance scientific computing. In *2008 5th IEEE International Symposium on Biomedical Imaging: From Nano to Macro*. 836–838. https://doi.org/10.1109/ISBI.2008.4541126

[42] John D. McCalpin. 1991-2007. *STREAM: Sustainable Memory Bandwidth in High Performance Computers*. Technical Report. University of Virginia, Charlottesville, Virginia. http://www.cs.virginia.edu/stream/ A continually updated technical report. http://www.cs.virginia.edu/stream/.

[43] John D. McCalpin. 2023. Bandwidth Limits in the Intel Xeon Max (Sapphire Rapids with HBM) Processors. In *High Performance Computing*, Amanda Bienz, Michèle Weiland, Marc Baboulin, and Carola Kruse (Eds.). Springer Nature Switzerland, Cham, 403–413.

[44] David Mizell and Kristyn Maschhoff. 2009. Early experiences with large-scale Cray XMT systems. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing (IPDPS '09)*. IEEE Computer Society, USA, 1–9. https://doi.org/10.1109/IPDPS.2009.5161108

[45] Hassan Mujtaba. 2022. Intel Sapphire Rapids '4th Gen Xeon' CPU Delidded By Der8auer, Unveils Extreme Core Count Die With 56 Golden Cove Cores. *WCCFTech* (January 2022). https://wccf.tech/189cd

[46] Anurag Mukkara, Nathan Beckmann, Maleen Abeydeera, Xiaosong Ma, and Daniel Sanchez. 2018. Exploiting Locality in Graph Analytics through Hardware-Accelerated Traversal Scheduling. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–14. https://doi.org/10.1109/MICRO.2018.00010

[47] Michael D. Noakes, Deborah A. Wallach, and William J. Dally. 1993. The J-Machine Multicomputer: An Architectural Evaluation. In *Proceedings of the 20th Annual International Symposium on Computer Architecture* (San Diego, California, USA) *(ISCA '93)*. Association for Computing Machinery, New York, NY, USA, 224–235. https://doi.org/10.1145/165123.165158

[48] 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 *Proceedings of the 49th Annual International Symposium on Computer Architecture*. ACM, New York New York, 817–830. https://doi.org/10.1145/3470496.3527400

[49] Rajeev Kumar Pandey and Sanjeev Kumar Pandey. 2020. Analyzing the Performance of 7nm FinFET Based Logic Circuit for the Signal Processing in Neural Network. In *2020 IEEE Recent Advances in Intelligent Computational Systems (RAICS)*. 136–140. https://doi.org/10.1109/RAICS51191.2020.9332500

[50] J. Thomas Pawlowski. 2011. Hybrid memory cube (HMC). In *2011 IEEE Hot Chips 23 Symposium (HCS)*. 1–24. https://doi.org/10.1109/HOTCHIPS.2011.7477494

[51] Steven A. Przybylski. 1990. Cache and memory hierarchy design. (1 1990). https://doi.org/10.1016/B978-0-08-050059-1.50006-2

[52] Andronicus Rajasukumar, Jiya Su, Yuqing, Wang, Tianshuo Su, Marziyeh Nourian, Jose M Monsalve Diaz, Tianchi Zhang, Jianru Ding, Wenyi Wang, Ziyi Zhang, Moubarak Jeje, Henry Hoffmann, Yanjing Li, and Andrew A. Chien. 2024. UpDown: Programmable fine-grained Events for Scalable Performance on Irregular Applications. arXiv:2407.20773 [cs.AR] https://arxiv.org/abs/2407.20773

[53] Cliff Robinson. 2021. AMD Zen 3 at Hot Chips 33. https://www.servethehome.com/amd-zen-3-at-hot-chips-33/

[54] Brian C. Schwedock, Piratach Yoovidhya, Jennifer Seibert, and Nathan Beckmann. 2022. tākō: a polymorphic cache hierarchy for general-purpose optimization of data movement. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*. ACM, New York New York, 42–58. https://doi.org/10.1145/3470496.3527379

[55] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. 2010. x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM* 53, 7 (July 2010), 89–97. https://doi.org/10.1145/1785414.1785443

[56] Anton Shilov. 2023. Micron Publishes Updated DRAM Roadmap: 32 Gb DDR5 DRAMs, GDDR7, HBMNext. *AnandTech* (July 2023). https://www.anandtech.com/show/18982/micron-publishes-updated-dram-roadmap-32-gb-ddr5-drams-gddr7-hbmnext

[57] Anton Shilov. 2023. Micron Unveils HBM3 Gen2 Memory: 1.2 TB/sec Memory Stacks For HPC and AI Processors. *AnandTech* (July 2023). https://www.anandtech.com/show/18981/micron-unveils-hbm3-gen2-12-tbs-per-stack-at-92-gts-speed

[58] Siarhei Siamashka. 2017. tinymembench. https://github.com/ssvb/tinymembench.

[59] J.E. Smith and A.R. Pleszkun. 1988. Implementing precise interrupts in pipelined processors. *IEEE Trans. Comput.* 37, 5 (1988), 562–573. https://doi.org/10.1109/12.4607

[60] James E Smith. 1982. Decoupled access/execute computer architectures. *ACM SIGARCH Computer Architecture News* 10, 3 (1982), 112–119.

[61] Ryan Smith. 2023. AMD: EPYC "Genoa-X" CPUs With 1.1GB of L3 Cache Now Available. *AnandTech* (June 2023). https://www.anandtech.com/show/18914/amd-epyc-genoax-cpus-with-11gb-of-l3-cache-shipping-now

[62] Ryan Smith. 2024. NVIDIA Blackwell Architecture and B200/B100 Accelerators Announced: Going Bigger With Smaller Data. *AnandTech* (March 2024). https://www.anandtech.com/show/21310/nvidia-blackwell-architecture-

and-b200b100-accelerators-announced-going-bigger-with-smaller-data

[63] Jared Stark, Paul Racunas, and Yale N. Patt. 1997. Reducing the performance impact of instruction cache misses by writing instructions into the reservation stations out-of-order. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture* (Research Triangle Park, North Carolina, USA) *(MICRO 30)*. IEEE Computer Society, USA, 34–43.

[64] Aaron Stillmaker and Bevan Baas. 2017. Scaling equations for the accurate prediction of CMOS device performance from 180nm to 7nm. *Integration* 58 (2017), 74–81. https://doi.org/10.1016/j.vlsi.2017.02.002

[65] Aaron Stillmaker and Bevan Baas. 2019. Corrigendum to "Scaling Equations for the Accurate Prediction of CMOS Device Performance from 180nm to 7nm" [Integr. VLSI J. 58 (2017) 74–81]. *Integr. VLSI J.* 67, C (jul 2019), 170. https://doi.org/10.1016/j.vlsi.2019.04.006

[66] M.B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, Jae-Wook Lee, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal. 2002. The Raw microprocessor: a computational fabric for software circuits and general-purpose programs. *IEEE Micro* 22, 2 (2002), 25–35. https://doi.org/10.1109/MM.2002.997877

[67] G. Thirugnanam, N. Vijaykrishnan, and M.J. Irwin. 2001. A novel low power CAM design. In *Proceedings 14th Annual IEEE International ASIC/SOC Conference (IEEE Cat. No.01TH8558)*. 198–202. https://doi.org/10.1109/ASIC.2001.954697

[68] R. M. Tomasulo. 1967. An Efficient Algorithm for Exploiting Multiple Arithmetic Units. *IBM Journal of Research and Development* 11, 1 (1967), 25–33. https://doi.org/10.1147/rd.111.0025

[69] Steve VanderWiel and David J Lilja. 1996. A survey of data prefetching techniques. In *Procs. of the 23rd International Symposium on Computer Architecture*. Citeseer.

[70] Krishnaswamy Viswanathan. 2024. Intel® memory latency Checker v3.11. https://www.intel.com/content/www/us/en/developer/articles/tool/intelr-memory-latency-checker.html

[71] Luming Wang, Xu Zhang, Songyue Wang, Zhuolun Jiang, Tianyue Lu, Mingyu Chen, Siwei Luo, and Keji Huang. 2024. Asynchronous Memory Access Unit: Exploiting Massive Parallelism for Far Memory Access. http://arxiv.org/abs/2404.11044 arXiv:2404.11044 [cs].

[72] Zhengrong Wang and Tony Nowatzki. 2019. Stream-based memory access specialization for general purpose processors. In *Proceedings of the 46th International Symposium on Computer Architecture* (, Phoenix, Arizona,) *(ISCA '19)*. Association for Computing Machinery, New York, NY, USA, 736–749. https://doi.org/10.1145/3307650.3322229

[73] Xiangyao Yu, Christopher J. Hughes, Nadathur Satish, and Srinivas Devadas. 2015. IMP: indirect memory prefetcher. In *Proceedings of the 48th International Symposium on Microarchitecture* (Waikiki, Hawaii) *(MICRO-48)*. Association for Computing Machinery, New York, NY, USA, 178–190. https://doi.org/10.1145/2830772.2830807

[74] Dan Zhang, Xiaoyu Ma, Michael Thomson, and Derek Chiou. 2018. Minnow: Lightweight Offload Engines for Worklist Management and Worklist-Directed Prefetching. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems* (Williamsburg, VA, USA) *(ASPLOS '18)*. Association for Computing Machinery, New York, NY, USA, 593–607. https://doi.org/10.1145/3173162.3173197

[75] C.A. Zukowski and Shao-Yi Wang. 1997. Use of selective precharge for low-power content-addressable memories. In *1997 IEEE International Symposium on Circuits and Systems (ISCAS)*, Vol. 3. 1788–1791 vol.3. https://doi.org/10.1109/ISCAS.1997.621492

A MAPPING MPAM TO OTHER ARCHITECTURES

A.1 Decoupled Access and Execute Architectures:

In Decoupled Access Execute(DAE) Architecture [60], AM_{FE} and AM_{BE} are implemented as two separate in-order cores - Execute (E) and Access (A) respectively. Each processor runs a separate instruction stream. These two instruction streams are carefully derived such that the Access processor performs all the memory operations, and the Execute processor only runs non-memory instructions. Both these processors are executed in-order. The two processors communicate using two queues 1. AEQ-Access to Execute Queue to forward loads from A to E and 2. EAQ-Execute to Access Queue to forward stores from E to A. For stores, the A processor allocates computed addresses in an internal queue WAQ-Write address queue that is synchronized with the arrival of data on EAQ. For loads, an entry in the AEQ is allocated to synchronize memory responses. N_{sync} and N_{out} are the same value in these architectures and given by

$$MLP = N_{sync} = N_{out} = (1/cl) \cdot (FIFO_{WAQ} \cdot s_q + FIFO_{AEQ} \cdot s_q)$$

where $FIFO_X$ is the number of entries in Queue X and s_q is the size of each entry in words.

A.1.1 Decoupled Supply-Compute (DeSC)[25]: Decoupled Supply-Compute (DeSC) develops the DAE idea further. In DeSC AM_{FE} and AM_{BE} are implemented as two specialized devices, SuppD and CompD. SuppD 'supplies' CompD with data to compute on. These devices can be programmable processors or specialized accelerators. The two devices are coupled with CommQ, a hardware FIFO queue that feeds into CommBuff in CompD. A Store Value Buffer is also used for synchronizing store values. However, memory accesses and memory parallelism are controlled and limited by mechanisms in SuppD. If SuppD is an OOO core (as evaluated in [25]), the memory parallelism matches that of an OOO core as below

$$\begin{aligned} N_{out\_sync} &= MSHR_{l1} \\ N_{out\_async} &= \alpha(MSHR_{l2} - MSHR_{l1}) \\ MLP &= min(R_{phy}/cl, MSHR_{l1}) \\ &\quad + \alpha \cdot (MSHR_{l2} - MSHR_{l1}) \end{aligned} \tag{26}$$

The design does allow SuppD to be a specialized accelerator, in which case the memory access mechanism of that accelerator will dictate the MLP of the system. The performance benefit of the system lies in the runahead execution of SuppD over CompD that fills up CommQ and CommBuff with operands for CompD to lookup.

A.1.2 MAPLE[48]: MAPLE implements AM_{FE} and AM_{BE} as execute and access processes running on separate cores, coupled circular communication queues in a special hardware unit (MAPLE Engine) using its scratchpad. The processes communicate with MAPLE using MMIO, and MAPLE queue slots are used for synchronization and outstanding namespaces. Thus, the MAPLE queue size determines the MLP per process. Multiple cores can share a single MAPLE unit. MAPLE itself goes through the LLC to access the DRAM memory.

$$\begin{aligned} N_{sync} &= N_{out} = n_{MAPLE} \cdot (1/cl) \cdot FIFO_{MAPLE} \\ N_{sh} &= n_{MAPLE} \cdot MSHR_{llc} \\ MLP &= min(n_{MAPLE} \cdot (1/cl) \cdot FIFO_{MAPLE} \cdot s_q, \\ &\quad n_{MAPLE} \cdot MSHR_{llc}) \end{aligned} \tag{27}$$

where n_{MAPLE} is the number of MAPLE units, $FIFO_{MAPLE}$ is the number of entries per MAPLE unit, and s_q is the size of each entry in words.

A.1.3 AMU[71]: Finally, a more recent research proposal, the Asynchronous Memory Access Unit, partitions the L2 cache to create a scratchpad for book-keeping of outstanding requests to support higher memory parallelism with an OOO core. N_{sync} and N_{out} for this system are again equal and limited by the number of entries in its memory access request table (or request ID namespace). If b_{req} represents the number of bits in the request ID,

$$MLP = N_{sync} = N_{out} = (1/cl) \cdot 2^{b_{req}} \cdot s_q \tag{28}$$

where s_q is the average size of each memory access.

A.2 Multi-Threaded Architectures:

A.2.1 MTA based Architectures [7, 44]: Barrel Process Architectures like [7, 8, 44] used multiple thread contexts to create larger synchronization namespace and removed data caches. Each thread could issue up to 8 outstanding requests. So for these systems,

$$\begin{aligned} N_{sync} &= n_{threads} \cdot R \\ N_{out} &= n_{threads} \cdot 8 \\ MLP &= (1/cl)\, min(n_{threads} \cdot R \cdot s_s, n_{threads} \cdot 8 \cdot s_o) \end{aligned} \tag{29}$$

for a single processor. R is the number of registers available per thread, s_s is the word-length of each register and s_o is the size of each outstanding request.

A.2.2 GPUs: GPU threads execute in warps. The synchronization namespace is enlarged to the number of threads supported as in [7]. However, all threads in a GPU do not use the same number of registers, and the synchronization namespace is limited by the physical registers available $R_{phy'}$. This is typically an order of magnitude higher than the physical registers in a CPU to support the large number of threads.

$$\begin{aligned} N_{sync} =& min(n_{threads} \cdot R, R_{phy'}) \\ N_{out\_sync} =& MSHR_{l2} \\ MLP_{SM} =& (1/cl)\, min(\, min(n_{threads} \cdot R, R_{phy'}) \cdot s_s, \\ & MSHR_{l2}) \end{aligned} \tag{30}$$

where MLP_{SM} is the MLP per stream processor, and s_s is the register word size. With newer designs [3], accelerators have been added to asynchronously fill the shared memory with data from memory. In such cases, an additional component N_{out_async} is added as below

$$\begin{aligned} MLP_{SM} =& (1/cl)(\, min(\, min(n_{threads} \cdot R, R_{phy'}) \cdot s_s, \\ & MSHR_{l2}) + Sh_{mem}) \end{aligned} \tag{31}$$

where Sh_{mem} is the maximum shared memory that can be configured to be loaded by the accelerator.