Architecture and RISC-V ISA Extension Supporting Asynchronous and Flexible Parallel Far Memory Access

Songyue Wang, Luming Wang, Tianyue Lu, Mingyu Chen
Center for Advanced Computer Systems, ICT, CAS
University of Chinese Academy of Sciences
Beijing, China
wangsongyue18@mails.ucas.ac.cn,{wangluming,lutianyue,cmy}@ict.ac.cn

ABSTRACT

Nowadays memory access speed has significantly lagged behind the improvement of CPU performance. As far memory techniques (e.g. heterogeneous memory and memory pooling) grow with ever-increasing datasets, the "memory wall" problem becomes even worse. Far memory access latency is further increased than traditional DRAM but it provides higher capacity. However, modern CPUs are latency-sensitive and cannot fully utilize far memory resources efficiently.

We propose an architecture that supports explicitly manipulating massively parallel asynchronous memory accesses and its RISC-V ISA extension (AME) under far memory scenarios. This work decouples the request and response of memory access operation, and provides asynchronous load/store instructions in non-blocking mode with variable granularity. Software can use AME to issue massive overlapping memory access requests and focus on other work while waiting for responses to improve performance.

We implement AME on a RISC-V CPU and build an FPGA-based prototype system. The evaluation shows that for memory-sensitive benchmarks, our approach achieves on average 11× speedup when the additional latency introduced by far memory is equivalent to 2μs faced with a 2GHz CPU, with the average number of in-flight memory requests reaches 238 for HPCC Random Access. Improvements are also shown in other practical memory-bound applications such as graph computing and concurrent data structures.

KEYWORDS

memory level parallelism, asynchronous memory access

1 INTRODUCTION

Nowadays, big data applications have an increasing demand for memory capacity (such as in-memory databases, online analysis, etc.). Various emerging memories are being introduced into data centers to meet this demand. One current solution uses new interconnect technologies and protocols to access memory on other machines or memory pools, such as CXL[2], OpenCAPI[1] and GenZ[3]. Another example is Non-Volatile Main Memory (NVMM), which offers higher memory density than DRAM and still can be accessed at cache-line granularity[4]. We call all the above **far memory** because it is usually loosely connected and its access latency is higher than DRAM[24]. They mainly have the following characteristics:

- **Direct accessible via load/store.** This work focuses on the scenario that applications access far memory with load/store instructions rather than using them as a swap device.
- **Widely distributed latency.** Applications can access memory on a remote node or a new medium (e.g. NVMM), and the access latency can be distributed over a wide range.
- **Potentially large aggregated capacity and bandwidth.** As memory resources can come from multiple nodes, the large number of total memory channels makes the aggregated memory capacity and bandwidth show significant superiority compared to local memory, leaving it a challenge for processors to fully utilize the abundant memory resources.

However, modern CPUs are very sensitive to memory access latency. Research shows that the average slowdown of memory-bound workloads (such as Redis and GAPBS) is more than 30% when the additional latency introduced is only 64ns using CXL in Azure[21]. Modern CPU hides long access latency by implicitly issuing parallel memory requests, it dynamically schedules multiple load/store instructions to execute in parallel by its out-of-order mechanism. Furthermore, the non-blocking cache serves multiple memory requests simultaneously by miss status handling registers (MSHR). In such cases, memory access latency is hidden by memory-level parallelism (MLP).

The longer the latency, the more MLP is needed. Typical DRAM latency is lower than 200ns, while far memory latency is around 300ns-5μs[4]. Therefore, the number of pending memory requests on a far memory system could be much larger than those on local DRAM. However, to issue and manage in-flight memory requests, various hardware resources in a CPU are required, such as re-order buffer, physical register files, load/store queue and MSHR. These hardware resources limit the MLP of a CPU by lacking its capability to issue a sufficient number of requests to hide far memory access latency. For example, a single Intel Skylake core can only support no more than 12 in-flight DRAM memory accesses[25]. Besides, the granularity of existing memory access strategy is fixed at the size of cache line, which is excessive for random access to small blocks and insufficient for sequential and large blocks. So there are two challenges to improve MLP in far memory scenario:

How to make CPU fully and more flexibly utilize high memory bandwidth? We propose our Asynchronous Memory Access Unit (AMU)[27] inside CPU cores. AMU decouples memory access requests and responses logic and uses much larger storage than MSHR to record in-flight request status, making it support issuing massive requests in parallel. In addition, the size of the load-/store request is also variable from 8B up to 1KB decided by software. AMU can help CPU make full use of bandwidth by massive overlapping requests and accessing exact data size on demand.

How to reduce the occupation of critical resources by long-latency load/store instructions? We propose our RISC-V Asynchronous Memory Access ISA Extension (AME) to solve it.

Long occupation is caused by the synchronous load/store instructions waiting for the response in CPU pipeline and hold resources at the same time. As the software interface of AMU, AME conceptually provides asynchronous load/store instructions (called *aload* and *astore*) from CPU microarchitecture. *Aload/astore* instructions will commit without waiting for the data responses. Applications can use AME to issue requests in batches and then do other computations while waiting for responses to improve performance.

Both AMU and AME are designed inside RISC-V architecture, because RISC-V is an open source and extensible ISA, and the architecture of RISC-V CPU is more concise and modern. We use Chisel HDL[7] to implement AMU and AME based on NutShell[5], an in-order open source RISC-V64(IMACSU) SoC that has been verified through tape-out. Based on this, we build a FPGA-based prototype system on Xilinx ZYNQ UltraScale+ ZU19EG MPSoC board, where we also implement an emulated far memory access path that supports high bandwidth and variable latency. For programming support, we provide an agile coroutine-based programming framework, and evaluate performance on a series of memory-sensitive benchmarks. Results show when the additional latency introduced by far memory is 2μs faced with a 2GHz CPU, our approach achieves on average 11× speedup with the average number of in-flight memory requests reaching 238 for HPCC Random Access[22], 63.48× for sequence memory access and 9.04× for random. And we find that applications using AME are less sensitive to latency changes.

In conclusion, this paper makes the following contributions:

(1) Propose a RISC-V ISA Extension AME and design its architecture AMU to improve MLP of far memory by asynchronous access with variable granularity.
(2) Design a programming paradigm for exploiting MLP using AME.
(3) Implement AMU and AME on a RISC-V CPU, and build an FPGA-based prototype system. Then evaluate it with memory-bound workloads.

2 BACKGROUND

In this section, we first introduce existing approaches to mask long memory access latency and analyze their effects. Then we outline the key ideas of our design.

2.1 Software Latency-tolerant Techniques

The relatively high latency of far memory brings performance challenges for latency-sensitive applications. Interleaved execution[20] is a universal technique that runs multiple coroutines (or light-weight threads) to hide latency. Since existing CPU do not have asynchronous access instructions, software implements an asynchronous-access-like effect by invoking prefetch instructions. Software first issues a prefetch instruction, and then executes another coroutine. When the prefetch is estimated to be complete, it switches back to execute the load/store instruction to access the memory.

Several paradigms such as Group Prefetching (GP), Software-Pipelined Prefetching (SPP)[12] and Asynchronous Memory Access Chaining (AMAC)[18] are proposed to exploit parallelism of lookup-like operations. However, transforming an existing code to apply these paradigms is painful[16]. To implement such paradigms easily, researchers have tried to use C++20 standard concurrent libraries[14, 16] or have designed domain-specific languages[17]. But the critical problem of these methods is applications cannot know if the data has been fetched to the local cache, which is neither smart nor portable. The number of prefetch requests that CPUs can handle at the same time is also limited.

2.2 Hardware Latency-tolerant Techniques

Based on out-of-order (OoO) execution and non-blocking Cache, researchers have tried to improve the latency tolerance of processors (or improve MLP) in many different ways.

In practical, the MSHR is implemented as a fully-associative array, which is hard to increase. To address this problem, replacing the fully associative MSHRs array with hash arrays was proposed [6, 26]. Another idea is adding additional spaces (re-order buffer (ROB), instruction queue (IQ), etc.) to handle the long load latency. For example, the Load Slice Core[10] and Forward Slice Core[19] add extra IQ for issuing another load/store and its address calculation instructions by IBDA (a dependency analysis) supports, and this process is transparent to software. By this way, these instructions no longer occupy critical resources in the processor.

The idea of all these techniques is to allow CPUs to dynamically schedule and issue more memory access requests to cover the long latency caused by waiting for data responses. However, none of them fundamentally solve the bottleneck, which is the limited number of critical hardware resources (e.g. RF, ROB, IQ and MSHR). Assuming that the CPU running at 2GHz, 1μs access latency of far memory is as high as 2000 cycles. After a load instruction targeted to far memory was issued, ROB requires thousands-level entries to prevent the CPU pipeline from stalling and thousands of physical registers for writing back results. Things will be even worse when there are also many load instructions in subsequence, possibly requiring hundreds of MSHRs. These resource requirements are clearly beyond the number of resources available with current technology. Therefore, we argue that the current mechanism for implicitly issuing parallel memory requests is insufficient for far memory latency.

2.3 Key Idea of AME

AME is a novel solution to the problems mentioned in Section 1 with three groundbreaking key ideas. The key idea **A** solves the problem that traditional *load* and *store* instructions occupy critical hardware resources for a long time. Idea **B** allows CPUs to issue a large batch of memory access requests to make full use of bandwidth. And idea **C** solves the dilemma of insufficient critical resources in CPU pipeline and Cache in a new way.

A. Asynchronous memory access instructions with variable granularity. *Aload* and *astore* instructions commit without waiting for the memory responses, but just write the request to an **asynchronous memory request queue (AMQ)** in L2 Cache data array. Each request is marked by an ID. Each ID is reusable, but requires software to guarantee its uniqueness during transmission. For far memory access path, we adopt memory access protocols that can support identification for transferring this ID, which is supported by most protocols (e.g. ID signal in AXI4). We also provide an instruction *getfin* to query the completion of requests,

which will return an ID of a completed request. This is the way to inform software whether the request is complete. The asynchronous design of AME instructions can greatly reduce the pressure on ROB and IQ.

In addition, *aload* and *astore* also support variable granularity memory access from 8B up to 1KB. The granularity is set at a control register of the AMU, which will be mentioned in Section 3.3. This allows software to be more flexible with small-grained random access on demand instead of fitting to cache line size. Also a single request to load or store a large amount of data can make better use of bandwidth. In Section 6 we will experimentally demonstrate the benefit of variable granularity support for AME.

B. Decouples memory requests and responses. This work separates the processing logic of the memory request channel and response channel which are sequentially coupled before. The asynchronous memory request logic sends out asynchronous memory requests in AMQ in turn. The asynchronous memory response logic also maintains an **asynchronous memory request finished ID queue (AFQ)**, this logic processes the data response and writes the completion ID to AFQ.

C. Configurable ScratchPad Memory (SPM). To solve the problem of critical resource shortage, AMU configures a region of L2 Cache as SPM to partially replace the functions of the MSHR and physical register files (RF), so AMQ and AFQ are placed in SPM. Besides, software can adjust the ratio of Cache and SPM regions. This design has two advantages: First, the capacity of L2 Cache (typically larger than 128KB) is much larger than size of MSHR and RF, sufficient for maintaining a large number of in-flight memory access requests. In addition, SPM can be used to store data to reduce the pressure of the RF. For the large-grained memory access (such as 512B) supported by AMU, RF does not have enough capacity to store the response data but SPM can accommodate it easily. Second, this design is compatible with original architecture: for applications not using AME, SPM can be fully configured back to Cache space. In this way, the AMU proposed in this paper will not have any performance impact on traditional applications. Original structure of pipeline need not be significantly changed, and no large area of storage and logic will be introduced.

3 ISA AND MICROARCHITECTURE

In this section, we first introduce the specification of RISC-V AME extension shown in Table 1, using custom-0 (0001011) as its opcode. Then we present the microarchitecture of AMU.

3.1 ISA Extension

0000000	rs2	rs1	000	00000	0001011	ALOAD
0000000	rs2	rs1	001	00000	0001011	ASTORE
0000000	00000	00000	010	rd	0001011	GETFIN
0000000	00000	rs1	100	rd	0001011	ASETID
0000000	00000	rs1	101	rd	0001011	ACFGRD
0000000	rs2	rs1	110	00000	0001011	ACFGWR
funct7	rs2	rs1	funct3	rd	AME	

Table 1: Specification of RISCV64-AME ISA Extension

- `asetid id` Set the request ID. The reason for designing a separate ID setting instruction is that there are only 2 source operands in RISC-V instructions. ID is numbered from 1.
- `aload spm_addr, mem_addr` Issue an asynchronous load request with the set request ID (value of the latest *asetid* setting), read the data from memory address at *rs2* to SPM address at *rs1*.
- `astore spm_addr, mem_addr` Asynchronous store request instruction, write the data from SPM address at *rs1* to memory address at *rs2*.
- `getfin id` Get an ID of a completed request from AFQ. Return 0 if there is no completed request.
- `acfgrd value, offset` Read AMU Config Registers (Table 2).
- `acfgwr value, offset` Write AMU Config Registers.

The programming model will be described in Section 4. Here we first focus on the microarchitecture of AMU. Figure 1 presents an overview of the AMU logic in the CPU core, which is mainly distributed in the CPU pipeline and L2 Cache.

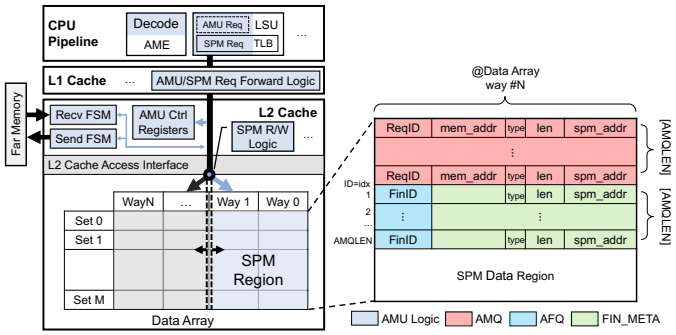

Figure 1: Overview of AMU microarchitecture

3.2 New Functions in Pipeline

In addition to decoding logic, the load store unit (LSU) needs to forward *aload* and *astore* requests directly to L2 Cache. And *aload* and *astore* will be committed after the requests are written to the AMQ in SPM. The *getfin* retrieves an ID from AFQ in SPM, similar to standard load instructions. The *acfgrd* and *acfgwr* also need to be forwarded to L2 Cache where AMU control registers locate in. For normal load/store instructions, TLB and memory access path should allow them directly access SPM in L2 Cache.

3.3 AMU Logic inside L2 Cache

Here we first introduce the division of SPM. According to the `SPMWAY` in AMU control registers (see Table 2), SPM occupies consecutive ways in data array and its address is encoded by the concatenate of way number and set number. Software uses standard load/store to access SPM and can adjust its size through *acfgwr*. Also, `SPMWAY` can be set to 0 for keeping all cache ways.

The metadata is placed at the beginning of the SPM, including the two circular queues AMQ and AFQ mentioned above (see Section 2.3), and a metadata table FIN_META for querying when responses arrive. When an *aload* and *astore* request forwarded by CPU pipeline reaches L2 Cache, AMU records {spm_addr, mem_addr, ID, request type (aload or astore), and granularity in `RWLEN` (see Table 2)} as an item in AMQ. The FIN_META table is designed to support out-of-order responses from far memory, indexed by request ID rather than a circular queue. When a request is issued by AMU, information (spm address, data length and request type) required for process of responses is recorded to

FIN_META[request ID]. This avoids new requests overwrites metadata of previous uncompleted requests when responses arrive out of order.

Ctrl Reg	Description
SPMWAY	The number of ways SPM consecutive occupied (from 0th) in data array
AMQLEN	The length of AMQ, AFQ and FIN_META
RWLEN	The data length of an *aload* or *astore* access (8-byte alignment required)

Table 2: AMU Control Registers

Finally we introduce decoupled asynchronous memory request and response logic, which are implemented using state machines (*Send FSM* and *Recv FSM* in Figure 1). Send FSM sequentially reads the request items in AMQ and issues far memory access requests with its ID in turn, then records them into FIN_META[request ID]. Recv FSM starts working when a far memory response arrives. First, metadata in FIN_META according to response ID is found. Then Recv FSM writes the response data to SPM if the type of request is aload , and puts the response ID into AFQ finally.

4 PROGRAMMING MODEL

Traditional memory accesses are all serial and synchronous at the code level, so a novel programming paradigm is needed for asynchronous memory access provided in AME. In this section, we first show the basic approach to programming with AME. Then a coroutine-based agile programming framework **CAP** (Coroutines for AME Programming) will be introduced, which can greatly simplify the code for applying AME. For ease of use, we wrap the AME instructions into C/C++ function interfaces (see Table 3).

Instructions	Function Interface
aload	aload (int id, uintptr_t spm_addr, uintptr_t mem_addr)
astore	astore (int id, uintptr_t spm_addr, uintptr_t mem_addr)
getfin	int id = getfin ()
acfgrd	uint64_t value = acfgrd (int offset)
acfgwr	void acfgwr (uint64_t value, int offset)

Table 3: AME Programming Interface

4.1 Basic Paradigm

List 1 shows the basic programming paradigm of AME. It first configures AMU, and executes *aload* without data returns. Then it handles other work when this request is unfinished, and uses *getfin* to check for finished requests on demand. After finished, data can be accessed via standard load/store instructions.

```
#define MAX_PARALLELISM 128
int *far_mem_to_access; // assume on far memory
//  AMU configuration
acfgwr(MAX_PARALLELISM, AMQLEN);
acfgwr(sizeof(int), RWLEN);
int *spm_data_area = (int *)alloc_spm_addr(sizeof(int));
int id = 1;  // alloc a request ID
// invoke an asynchronous memory access requests
aload(id, spm_data_area, &far_mem_to_access);
while(id != getfin()) { /* do something else  */ }
// access via standard load/store instruction
printf("%d\n", *spm_space);
```

List 1: Basic usage of AME

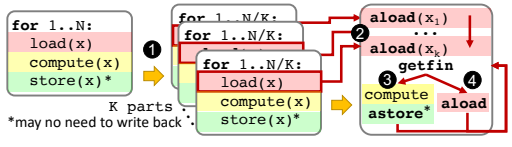

Figure 2: Steps to issuing massive request with AME

For memory access intensive programs, we can use AME to issue massive memory access requests in this way: First, divide the program into several parts ❶. Each part usually loads a remote element, then calculates on it and repeats this process. Second, convert these *load* to *aload* requests, issue them in batches and record the request ID for each part ❷. When a completed request ID is retrieved by *getfin*, programs perform post-memory access calculation of the part that corresponds to this ID ❸, and continue to execute *aload/astore* after completion for this part ❹.

```
/* --------------------- Original Version --------------------- */
for (int i = 0; i < n; ++i)  L[i] ^= i;
/* -------------- AME Version with State Machine -------------- */
#define NUM_PARTS 256
struct {
  int idx;              // now index of this part
  int num_finish;       // finished update number
  int stage;            // 0: need aload, 1: already aload
} update[NUM_PARTS + 1]; // status record struct
// ❶ division init: part 'j' (0..255) holds id 'j + 1' (1..256)
int *spm_addr = alloc_spm_addr (NUM_PARTS * sizeof(int));
for (int j=0; j<NUM_PARTS; j++) {
  update[j].idx = j * NUM_PARTS / n
  update[j].num_finish = 0, update[j].stage = 1;
}
// ❷ first round: aload in batch by exploiting MLP
for (int j=0; j<NUM_PARTS; j++)
  aload(j + 1, &spm_addr[j], &L[update[j].idx++]);
// aload/astore State Machine
int remain_parts = NUM_PARTS;
while (remain_parts > 0) {
  if ((j = getfin()) != 0){ // one response retrieved
    switch(update[j - 1].stage) {
    case 0:  // ❹ issue next aload
      if (update[j - 1].num_finish < n / NUM_PARTS) {
        aload(j, &spm_addr[j - 1], &L[update[j - 1].idx++]);
        update[j - 1].stage = 1;
      } else remain_parts--;
      break;
    case 1:  // ❸ calculate and astore
      spm_addr[j - 1] ^= update[j - 1].idx;
      astore(j, &spm_addr[j - 1], &L[update[j - 1].idx]);
      update[j - 1].stage = 0, update[j - 1].num_finish++;
      break;
    }
  }
}
```

List 2: Exploiting MLP using AME: an example of update

We take a sequential update program as an example (first 2 lines in List 2). It cyclistically loads elements of the array `L` allocated in far memory and updates them after `xor`. We first modify it by ❶ dividing the update task into `NUM_PARTS` parts. The execution process of each part can be described by a state machine: ❷ sends batched *aload* requests → ❸ one request complete, calculate and send *astore* request → ❹ *astore* complete and continue *aload*. To support this idea, the state of each part, index and number of finished updates need to be recorded in local memory. The modification with state machine idea using AME is in the rest of List 2.

4.2 CAP Coroutines Programming Framework

Using the paradigm above (state machine) can issue massive asynchronous memory requests, but it requires elaborate state machine design and heavy modifications for complex programs.

Thus we implement an agile programming framework CAP based on the coroutine library in C++20. CAP adopts AME to implement the switching and scheduling methods of coroutines (such as *suspend* after *aload* and *astore*, *resume* after *getfin*). And CAP provides an implementation of local locks based on address hashing, so that it can protect concurrency consistency. This makes modifications using AME become easier and similar to multi-threaded

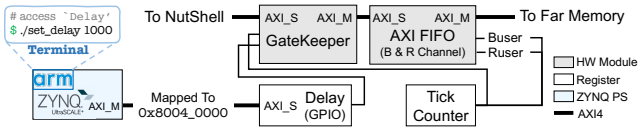

Figure 4: Design of Delayer in emulated far memory path

programs, without explicitly modifying with step ❷, ❸ and ❹. Programmers have no need to pay attention to the details of life-cycle of IDs, *getfin* and scheduling (see List 3).

```
template<typename Scheduler>
coro::task<void> update (int idx, int *L, int eachNUPDATE,
      Scheduler &sched){
  int *spm_addr = sched.alloc_spm_addr();
  for (int i = idx; i < idx + eachNUPDATE; ++i) {
    co_await aload_coro(spm_addr, &L[i], sched);
    *spm_addr ^= i;
    co_await astore_coro(spm_addr, &L[i], sched);
  }
  sched.release_spm(spm_addr);
}
```

List 3: Example update program using CAP

5 FPGA-BASED PROTOTYPE SYSTEM

We extended NutShell[5], an in-order pipeline RISC-V CPU verified through taped out, to support AME. And we built our FPGA-based prototype system on a Xilinx ZYNQ UltraScale+ ZU19EG MPSoC board for evaluation. It has quad ARM Cortex A53 processors on the Processing System (PS), which can help us access GPIO or monitor datastream in Programmable Logic (PL).

Our FPGA prototype system comprises two memory paths for the NutShell with AME implemented (Figure 3). They are mapped to different address spaces, and characteristics are as follows:

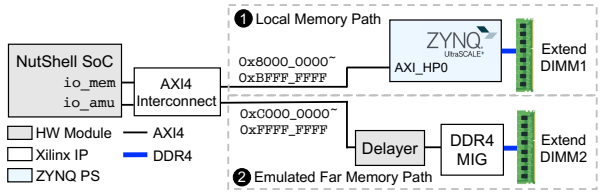

Figure 3: Memory paths on FPGA-based Prototype System

Local Memory Path (❶ in Figure 3) We use the AXI_HP interface on the PS side of ZYNQ. This HP port only supports 64 simultaneous requests, but its memory controller is hard logic with high clock frequency. Therefore, path ❶ is in line with the characteristics of local memory with low latency and low concurrency.

Emulated Far Memory Path (❷ in Figure 3) We use extrapolated DDR4 memory as far memory. A soft Memory Interface Generator (MIG) is instantiated as a memory controller, and its AXI AxID width is 32-bit which supporting high concurrency. In addition, far memory responses will go through the module *Delayer*, which adds an equal and adjustable delay to all overlapping responses in AXI B and R channel. Figure 4 depicts the structure and access of *Delayer*. It contains an AXI FIFO with 256 entries (the maximum number of concurrent requests in our system) for temporarily holding responses, and record the timestamp when each response entered the FIFO via the xUSER signal for precisely controlling the latency added to it. The module "GateKeeper" is responsible for releasing requests with sufficient waiting time at the exit of FIFO by controlling handshake signals of AXI. In addition, we made the register "Delay" (latency value we set) as a AXI GPIO, and mapped it to the address space of the PS-side ARM core. Therefore, this value can be adjusted at any time by software on the ARM core through accessing this address.

In order to observe the number of in-flight requests in real time from the hardware, we add a counter to AMU, which records the difference of the number of requests which have already sent and received. We also make this counter an AXI GPIO and map it to the address space of ARM core for observing.

Item	Origin	AME	+/-	Item	Origin	AME	+/-
LUT	14040	16130	+14.9%	BRAM + URAM	7.5+12	6+14	+2.5%
FF	12073	14656	+21.4%	LUTRAM	539	587	+8.9%

Note: Tools: Vivado 2019.2, core_clk at 200MHz and uncore_clk at 100MHz.

Table 4: Post-implementation utilization comparison

Table 4 shows the utilization comparison on SoC hierarchy in FPGA project before and after implementing AME. It can be seen that AME has little impact on the original SoC especilally on storage because of the reuse of SPM in L2 Cache.

6 EVALUATION

6.1 Benchmarks

We use 6 benchmarks to verify the performance of AME under different memory access latency scenarios on our prototype system for evaluation. The descriptions and workloads are as follows:

Random Access (GUPS) We adopt the single-node version of HPCC Random Access benchmark[22]. It continuously selects elements of the table in memory to update at random. The program gives GUPS (Giga Updates Per Second) as the performance evaluation. The table is allocated in far memory with a size of 256MB (32M uint64_t elements in total), and the number of updates is 2^{20}.

Sequence Access (SA) SA is a sequential memory access benchmark modified from GUPS. It sequentially updates elements in the table instead of random access. The workload is the same as GUPS.

Hash Join (HJ) We adopt the multi-core Hash Join benchmark from ETH System Group[9]. It first builds a hash table based on a relation with a set of tuples, and then uses another relation for detection. HJ allocates hash tables and relations in far memory, each relation contains 512,000 tuples (7.812MB) at random. Performance is measured in the number of tuples processed per second.

Integer Sort (IS) IS is a parallel performance benchmark from the NAS Parallel Benchmark (NPB) v3.3.0[8]. It uses the radix sort algorithm, first pre-sorts the high order of elements with a medium bucket then sorts all elements to reduce random access. We choose workload W: the size of array to sort is 2^{20} (8MB, uint64_t type), the size of bucket is 2^{10} and the maximum element is 2^{16}, while the array is placed in far memory and the bucket is placed in local memory. The performance is measured by its running time.

Hash Table with Hand-over-hand Linked List (HL) We adopt the chained hash table from ASYCLIB[13], a concurrent data structure protected by locks. Each hash table bucket points to a hand-over-hand locking linked list[15], which contains collided items. Hash table is allocated in local memory with 32768 buckets, while the linked lists are allocated in far memory with 102400 keys (2MB). HL launches 256 coroutines and each operates with 100 random keys, 70% of which are lookup and 30% are insert. The performance is measured in number of operations per second (op/s).

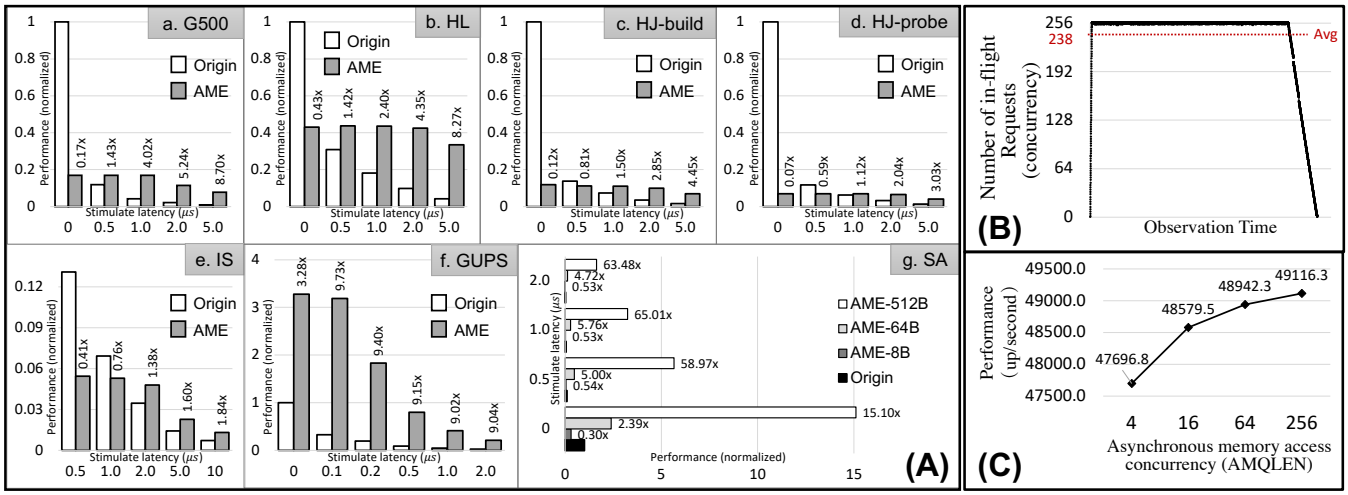

Figure (A) shows the performance comparison for all benchmarks with the maximum number of concurrency is 256. The x-axis represents the simulate latency (μs), and y-axis represents the normalized value of performance. The labels on columns are the speedup ratio of AME relative to the original at this certain latency. In particular, the different columns in **Figure (A).g** represent the performance of AME with different granularity of aload and astore.
Figure (B) shows the change of the number of in-flight requests over time when GUPS has a maximum concurrency at 256 with 2μs additional simulate latency.
Figure (C) shows GUPS performance with different maximum concurrency settings.

Figure 5: Evaluation results

Graph500 (G500)[23] We construct a graph with 2^{16} nodes (16MB) using the Kronecker algorithm, where each node has a degree of 16. G500 builds 8 BFS trees with a seq-list as its data structure. The graph and vertex list are allocated in far memory. We measure performance by its running time.

6.2 Evaluation Setup

We use our FPGA-based prototype system to simulate the performance of a 2GHz processor faced with different far memory access latencies. In our system, NutShell runs at 200MHz, whose clock period is 10 times that of 2GHz. The frequency of both AXI4 memory paths is 100MHz, which is a reference to some real processors whose frequency of last-level cache and out parts (uncore) is reduced to half the frequency of the core.

In our experiments, the additional latency introduced by emulated far memory is between 0 and 10,000 clock cycles, which is equivalent to a 2GHz processor facing a situation of 0 to 10μs. In practice, access latency of NVMM is around 0.5μs, while machines with network latency or further may reach 10μs. The stimulate additional latencies shown on the x-axis below are all converted to this 2GHz processor.

For comparison, we deploy both the original (labeled "Origin") and NutShell with AME-implemented (labeled "AME") to our FPGA-based prototype system. To simulate in a more realistic runtime environment, all benchmarks were run on Debian 11.

6.3 Results

For clarity, we normalize the results to take the performance of origin NutShell with 0 additional latency in each benchmark as 1.

Basic Case. We first focus on pure far memory access performance, where SA and GUPS focus on random and sequential access respectively. The performance of SA on AME is inferior to the original design with Cache that utilizes locality when granularity is small as 8B. But a large-granularity access such as once 512KB can make more full use of bandwidth, so that AME can improve the performance by ~60× when the latency is greater than 1μs (Figure 5 (A).g). As for random access, figure 5 (A).f illustrates that using AME at 8B small-granularity is more flexible and efficient than traditional methods even when there is no additional latency, and the performance can be improved by ~9× with additional latency.

The concurrency of AME. Applications with supports of AME can sustain far more requests than traditional architectures shown in Figure 5 (B), where the average number of in-flight requests reaches 238. And larger memory access concurrency can greatly improve performance (see Figure 5 (C)).

Practical Benchmarks. We find that applications using AME are less sensitive to latency changes by just taking a glance at Figure 5 (A). G500 shows an obvious acceleration effect of AME on graph computing, which has big data sets and contains massive memory access. HL illustrates that AME can increase the throughput of concurrent data structures through asynchronous memory accesses. With variable granularity, even for applications that have a lot of sequential access like IS, AME can help improve performance.

7 CONCLUSION AND FUTURE WORK

This paper presents a study supporting explicit asynchronous memory access with variable granularity to hide the latency of far memory and improve MLP. We propose our Asynchronous Memory Access Unit (AMU) along with its ISA extension AME. Evaluation results show that, with asynchronous programming paradigm, memory-bound applications are able to exploit more power of far memory.

This work only presents the basic instructions and structures to enable asynchronous memory access. There are more works to be done in OS and compiler for real-world applications. Furthermore, this asynchronous design can be easily extended to support more memory extensions, such as complex access patterns and Processing-In-Memory (PIM) mechanism, given that the far memory subsystem supports more rich memory semantics[11].

REFERENCES

[1] 2017. OpenCAPI Specification. http://opencapi.org [Online; accessed: Febrary 2022].

[2] 2018. Compute Express Link. https://www.computeexpresslink.org/ [Online; accessed: Febrary 2022].

[3] 2018. Gen-Z Specification. https://genzconsortium.org/specifications [Online; accessed: Febrary 2022].

[4] 2019. Intel optane persistent memory. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html [Online; accessed: Febrary 2022].

[5] 2020. NutShell: Open Source Chip Project by University. https://github.com/OSCPU/NutShell [Online; accessed: Febrary 2022].

[6] Mikhail Asiatici and Paolo Ienne. 2019. Stop crying over your cache miss rate: Handling efficiently thousands of outstanding misses in fpgas. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 310–319.

[7] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. 2012. Chisel: constructing hardware in a scala embedded language. In *DAC Design automation conference 2012*. IEEE, 1212–1221.

[8] David Bailey, Tim Harris, William Saphir, Rob Van Der Wijngaart, Alex Woo, and Maurice Yarrow. 1995. *The NAS parallel benchmarks 2.0.* Technical Report. Technical Report NAS-95-020, NASA Ames Research Center.

[9] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M Tamer Özsu. 2013. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE, 362–373.

[10] Trevor E Carlson, Wim Heirman, Osman Allam, Stefanos Kaxiras, and Lieven Eeckhout. 2015. The load slice core microarchitecture. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 272–284.

[11] Li-Cheng Chen, Ming-Yu Chen, Yuan Ruan, Yong-Bing Huang, Ze-Han Cui, Tian-Yue Lu, and Yun-Gang Bao. 2014. MIMS: Towards a Message Interface Based Memory System. *Journal of Computer Science and Technology* 29, 2 (March 2014), 255–272. https://doi.org/10.1007/s11390-014-1428-7

[12] Shimin Chen, Anastassia Ailamaki, Phillip B Gibbons, and Todd C Mowry. 2007. Improving hash join performance through prefetching. *ACM Transactions on Database Systems (TODS)* 32, 3 (2007), 17–es.

[13] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. 2015. Asynchronized concurrency: The secret to scaling concurrent search data structures. *ACM SIGARCH Computer Architecture News* 43, 1 (2015), 631–644.

[14] Yongjun He, Jiacheng Lu, and Tianzheng Wang. 2020. CoroBase: coroutine-oriented main-memory database engine. *Proceedings of the VLDB Endowment* 14, 3 (2020), 431–444.

[15] Maurice Herlihy and Nir Shavit. 2012. *The Art of Multiprocessor Programming, revised first edition.* Morgan Kaufmann.

[16] Christopher Jonathan, Umar Farooq Minhas, James Hunter, Justin Levandoski, and Gor Nishanov. 2018. Exploiting coroutines to attack the" killer nanoseconds". *Proceedings of the VLDB Endowment* 11, 11 (2018), 1702–1714.

[17] Vladimir Kiriansky, Haoran Xu, Martin Rinard, and Saman Amarasinghe. 2018. Cimple: Instruction and memory level parallelism: A dsl for uncovering ilp and mlp. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*. 1–16.

[18] Onur Kocberber, Babak Falsafi, and Boris Grot. 2015. Asynchronous memory access chaining. *Proceedings of the VLDB Endowment* 9, 4 (2015), 252–263.

[19] Kartik Lakshminarasimhan, Ajeya Naithani, Josué Feliu, and Lieven Eeckhout. 2020. The Forward Slice Core Microarchitecture. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques* (Virtual Event, GA, USA) *(PACT '20)*. Association for Computing Machinery, New York, NY, USA, 361–372. https://doi.org/10.1145/3410463.3414629

[20] J. Laudon, A. Gupta, and M. Horowitz. 1994. Interleaving: A Multithreading Technique Targeting Multiprocessors and Workstations. In *International Conference on Architectural Support for Programming Languages & Operating Systems*.

[21] Huaicheng Li, Daniel S Berger, Stanko Novakovic, Lisa Hsu, Dan Ernst, Pantea Zardoshti, Monish Shah, Ishwar Agarwal, Mark Hill, Marcus Fontoura, et al. 2022. First-generation Memory Disaggregation for Cloud Platforms. *arXiv preprint arXiv:2203.00241* (2022).

[22] Piotr R Luszczek, David H Bailey, Jack J Dongarra, Jeremy Kepner, Robert F Lucas, Rolf Rabenseifner, and Daisuke Takahashi. 2006. The HPC Challenge (HPCC) benchmark suite. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, Vol. 213. 1188455–1188677.

[23] Richard C Murphy, Kyle B Wheeler, Brian W Barrett, and James A Ang. 2010. Introducing the graph 500. *Cray Users Group (CUG)* 19 (2010), 45–74.

[24] Christian Pinto, Dimitris Syrivelis, Michele Gazzetti, Panos Koutsovasilis, Andrea Reale, Kostas Katrinis, and H. Peter Hofstee. 2020. ThymesisFlow: A Software-Defined, HW/SW co-Designed Interconnect Stack for Rack-Scale Memory Disaggregation. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 868–880. https://doi.org/10.1109/MICRO50266.2020.00075

[25] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. 2019. ZombieLoad: Cross-privilege-boundary data sampling. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 753–768.

[26] James Tuck, Luis Ceze, and Josep Torrellas. 2006. Scalable cache miss handling for high memory-level parallelism. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*. IEEE, 409–422.

[27] Luming Wang, Xu Zhang, Tianyue Lu, and Mingyu Chen. 2022. Asynchronous memory access unit for general purpose processors. *BenchCouncil Transactions on Benchmarks, Standards and Evaluations* 2, 2 (2022), 100061. https://doi.org/10.1016/j.tbench.2022.100061