BenchCouncil Transactions on Benchmarks, Standards and Evaluations 2 (2022) 100061

Contents lists available at ScienceDirect

BenchCouncil Transactions on Benchmarks, Standards and Evaluations

journal homepage: www.keaipublishing.com/en/journals/benchcouncil-transactions-on-benchmarks-standards-and-evaluations/

Research article

Asynchronous memory access unit for general purpose processors

Luming Wang, Xu Zhang, Tianyue Lu, Mingyu Chen *

Institute of Computing Technology, Chinese Academy of Sciences, Beijing, 100190, China
School of Computer Science and Technology, University of Chinese Academy of Sciences, Beijing, 100049, China

ARTICLE INFO

Keywords:
Asynchronous memory access
Far memory
Micro-architecture

ABSTRACT

In future data centers, applications will make heavy use of far memory (including disaggregated memory pools and NVM). The access latency of far memory is more widely distributed than that of local memory accesses. This makes the efficiency of traditional out-of-order load/store mechanism in most general-purpose processors decrease in this scenario. Therefore, this work proposes an in-core asynchronous memory access unit to fully utilize the far memory resources.

1. Introduction

In recent years, to improve the utilization of resources in cloud data centers, more and more resources are organized into resources pools. Memory resources trends to be the next resources organized as a pool. However, nowadays remote memory pools usually use software interfaces (such as key-value, RDMA, files, etc.) rather than direct load/store [1]. Recently, new interconnect technologies and protocols (such as OpenCAPI [2], Gen-Z [3] and CXL [4]) enable the construction of load/store interface based disaggregated memory pool that contains multiple nodes. Prototypes of such systems have already been constructed by researchers [5]. It is foreseeable that complex disaggregated memory pools using direct load/store interface will emerge soon.

On the other hand, Non-volatile Main Memory (NVMM) start to emerge, offering higher memory density and lower standby power consumption. However, it faces similar challenges as remote memory systems. Compared to traditional DRAM, NVMM has higher latency and a wide range of latency variation (6x-30x higher write latency and 5x-10x higher read latency) [6]. Currently, there is no efficient and standard interface for accessing NVMM. Commercial products, such as Intel's Optane DC [7], still use a modified synchronous DRAM interface.

Both remote memory and non-volatile main memory are sometimes called "far memory" as their access latency is larger than local DRAM [8]. There are two main differences in accessing far memory compared to accessing traditional DRAM-based local memory.

Widely distributed latency The memory allocated from a disaggregated memory pool can locate on some faraway remote nodes. Furthermore, the memory device can be DRAM, NVM, or other emerging memory devices. As a result, memory access latency becomes uncertain. Latency may distribute over a wide range.

Potential large aggregated bandwidth As memory resources may come from multiple different machines, the maximum aggregated access bandwidth may increase significantly compared to local memory which is limited by physical channels, making it a challenge to make use of the abundant bandwidth.

Access latency in traditional memory systems is also uncertain due to the multi-level cache hierarchy. However, the distribution of latency is relatively narrow. The latency of a single access memory request is around 1 ns (when L1 hits) to 100 ns (when accesses local DRAM). Modern processors can tolerate this difference in latency by out-of-order execution and non-blocking cache. Fig. 1 shows the limitations of current Out-of-Order processors in far memory scenarios. The range of latency they can tolerate is limited by the number of entries in the instruction queue, ROB, MSHRs, etc. Once one of these resources is exhausted, the OoO processor cannot issue more memory access requests any longer. The resource insufficiency even occurs in the local memory scenario. For example, due to the limited number of MSHRs, a single core of Intel Skylake processor can only reach a memory bandwidth of about 15 GB/S, which is even lower than that of a single DDR4-2400 DIMM. It is difficult for modern processors to tolerate the access latency fluctuations (300 ns-10 μs) of far memory.

Although improving the out-of-order execution capability of traditional general-purpose processor cores [9–12] (e.g., increasing the number of entries of MSHRs and ROB, using multi-level MSHRs and ROB, etc.) can also improve the performance of load/store in this scenario. But such an improvement, even if it is feasible, requires significant hardware resources. The key issue is that traditional load/store instructions are synchronous, every outstanding memory access needs to hold at least one hardware resource until the operation is completed. The more parallelism the more hardware resources will be needed.

One approach to address this problem is asynchronous memory access. A similar predicament had already existed in network programming, where applications call blocking socket interfaces made program

* Corresponding author at: Institute of Computing Technology, Chinese Academy of Sciences, Beijing, 100190, China.
E-mail addresses: wangluming@ict.ac.cn (L. Wang), zhangxu19s@ict.ac.cn (X. Zhang), lutianyue@ict.ac.cn (T. Lu), cmy@ict.ac.cn (M. Chen).

https://doi.org/10.1016/j.tbench.2022.100061
Received 21 March 2022; Received in revised form 7 May 2022; Accepted 7 May 2022
Available online 24 May 2022

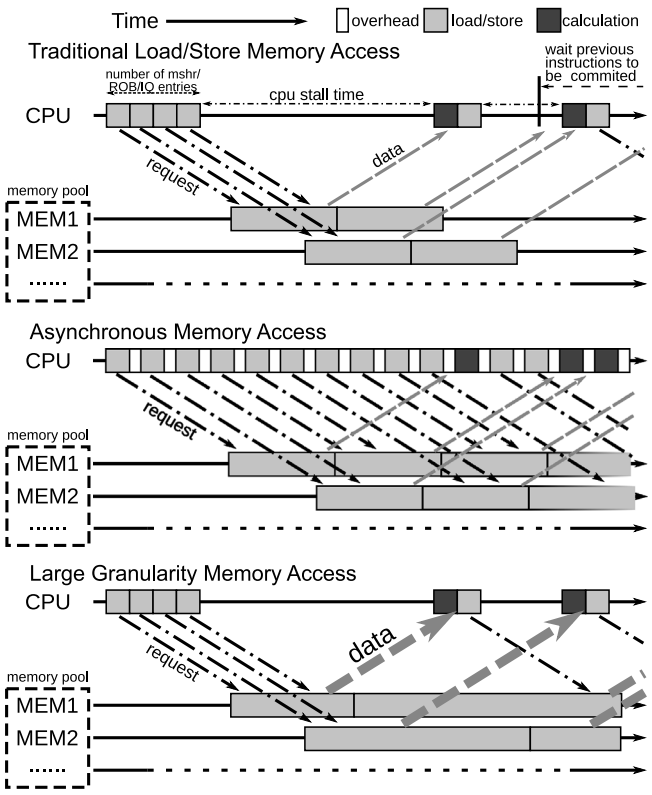

Fig. 1. Ways to improve memory bandwidth utilization.

performance suffer from network latency. To solve this problem, asynchronous non-blocking interfaces such as select() and epoll() had been built. Just as network programming has evolved from the early synchronous blocking model to today's asynchronous non-blocking model, we suggest that an asynchronous non-blocking model for memory access is also needed. Therefore, there should be some mechanisms for invoking asynchronous memory access efficiently in a general purpose processor.

Another approach is supporting memory access with variable granularity. As shown in Fig. 1, to improve performance, applications can initiate memory requests with a large granularity to hide the latency and fully utilize the bandwidth. Furthermore, applications can adjust the granularity based on data semantics to access memory flexibly and efficiently.

Concerning the above approaches, we propose an in-core Asynchronous Memory access Unit (AMU) to support asynchronous access in a general purpose processor. AMU enables applications to asynchronously initiate many variable granularity memory access requests by simple instructions, such as asynchronous load/store. AMU also enables applications to start various complex memory access requests with additional configuration registers. Processors can still use traditional synchronous load/store instructions for compatibility while the data from both sources can be consumed by computation instructions transparently. We argue that this is a more practical and efficient way than designing an un-core or off-chip asynchronous memory accelerator.

The following chapters outline the main features of the asynchronous memory access unit.

2. Asynchronous memory access unit

Traditionally, mostly used load/store instructions are implemented as synchronous mode. Each pending memory operation will hold certain hardware resources such as GPR, ROB and MSHR entry. The hardware resources will not be released until the memory operations have finished.

We propose a new class of asynchronous memory access instructions that will not hold hardware resources during the operation. An asynchronous instruction that invokes a memory access request will be committed immediately once it is accepted by functional unit and sent out. Thus, applications can continue to execute other operations rather than waiting for the memory access operation to finish. Then, applications can poll whether there is a completed request later. Moreover, even if an outstanding request did not finish for a long time, there is no pipeline stall due to the shortage of ROB or other hardware resources.

Asynchronous memory access instructions are decoded and issued as normal instructions. The functional unit to process asynchronous instructions is called Asynchronous Memory access Unit (AMU). AMU is inspired by the Vector Processing Unit (VPU) of many modern processors. The VPU is a separate functional unit in the CPU. Applications use the VPU through a standalone instruction set (i.e., vector instruction set), which contains a set of extra registers (i.e., vector registers) to hold the wide data to be processed. Besides, vector instructions are scheduled together with scalar instructions. Vector registers and scalar registers can exchange data efficiently. Just as VPU, AMU can coexist with synchronous load/store Unit and can be ignored when compatibility comes first.

AMU is responsible for processing asynchronous instructions. However, it cannot depend on internal hardware registers or queues to keep the status of outstanding memory operations, which will become another potential bottleneck for parallelism. In fact, the status of pending requests are stored in SPM. Each processor core is equipped with a ScratchPad Memory (SPM), which acts as vector registers in VPU but has a larger capacity and flexible data structure. Data is moved asynchronously and automatically between SPM and main memory by AMU. To initiate asynchronous memory access requests, applications can prepare data in SPM and then execute asynchronous memory access instructions. After receiving the request, AMU will move the data between memory and SPM in background.

From the view of an application, the SPM is a stand-alone memory space. Applications can use synchronous load/store instructions to access the data in the SPM and process them with other regular instructions. In addition, applications can copy data from main memory to the SPM and vice versa. The SPM is fully compatible with the processor's original data access and processing mechanisms.

By asynchronous access, AMU can support as many requests as the capacity of SPM can support in theory. However, AMU does not assure the consistency among all outgoing memory operations. The overhead of hardware consistency checking is one of the reasons that limit the capacity of traditional load/store queue and MSHRs. In the AMU design, we leave the consistency issue to software. We argue that software and hardware cooperation is the right way to exploit the memory parallelism over large latency.

2.1. Instructions

There are three core instructions of AMU. These instructions enable the most basic asynchronous memory access.

Asynchronous load/store instructions In AMU, *aload/astore* instruction invokes a data movement request between SPM and memory. An SPM address and a memory address are passed to AMU by registers. AMU will move data between the provided SPM address and off-core memory address. Then a request id, which is used for tracking the request, is stored in the destination register.

Instruction for getting an id of finished request We propose *getfin* instruction for getting an id of any completed request. If there is no finished request, the instruction returns a failure code. This instruction does not block execution regardless of whether there is a completed request or not.

2.2. Registers

Due to the limited field space of instructions, some complex memory access settings cannot be encoded in a single instruction. To solve this problem, we designed several configuration registers, which contain advanced parameters.

Memory Access Configuration Registers These registers contain advanced memory access configurations, including address format, granularity, priority, etc. Settings in the configuration will be combined with each access instruction to form a rich semantic memory request. Application can keep several different configuration registers for different data regions.

Default Configuration Register Due to the limited encoding space of some instructions, it is even not possible to specify all configuration registers. For such case, the system automatically chooses the configuration register specified in this register.

Access Pattern Registers The access pattern registers are used to initiate complex asynchronous memory access. They contain the access pattern(such as stride, neighbor, stream, etc.) of a class of complex memory access requests.

2.3. Programming model

Listing 1 shows a basic example of asynchronous memory accessing. The code initiates an asynchronous memory access request with the *aload* instruction. The code then keeps retrying the *getfin* instruction to get the id of a completed request and can do other work while the request is still pending. After the request completes, the code then reads the data from the SPM with the *load* instruction.

Listing 1: Asynchronous Memory Access Basic Example

```
int memory_need_to_be_accessed;
int *spm_space = (int*) A_SPM_ADDR;
// Invoke an asynchronous memory access
// requests. The request's id is ignored.
aload(spm_space,
  &memory_need_to_be_accessed);
while ((rd = getfin()) != 0) {
  // Do something else
}
// Access data from SPM via load/store
printf("%d\n", *spm_space);
```

AMU instructions can support a variety of programming paradigms.

- **Vector Model** Vector instructions and vector processors are mature techniques for exploiting data-level parallelism. As a technique to improve data-level parallelism, AMU instructions have many similarities to vector instructions. Thus, it is possible to combine AMU instructions with vector instructions efficiently.
- **Event-Driven Model** The event-driven model is a common paradigm in single-thread non-blocking network programming. Furthermore, the *aload/astore* instructions are like non-blocking socket *read()/write()*. *getfin* instructions are like the *select()* in network programming. Thus, the event-driven model can be naturally applied to asynchronous memory accesses especially for out of order scenarios.
- **Coroutine Model** For asynchronous access requests with complex access patterns, coroutines or lightweight software threads are more suitable programming paradigms. Coroutines can easily work with high performance concurrent data structures and enable more interactions between software and AMU.

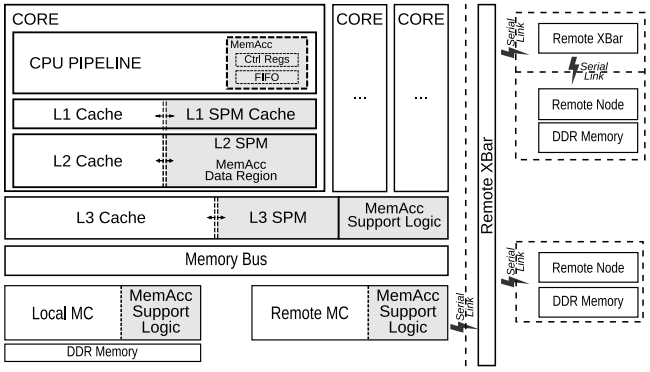

Fig. 2. Architecture design.

3. Architecture design

Fig. 2 shows the architecture design of AMU. There are three key design choices:

CPU Pipeline Integration To support efficient asynchronous memory access instructions, many state control registers are integrated into CPU core pipeline. Some of these registers indicate the AMU's status, which allows programs to rapidly get the status of outstanding instructions. In addition, speculative execution of asynchronous memory access instructions brings new challenges since some states are stored in SPM. This requires the pipeline of the processor core to be designed carefully.

Re-configurable Cache/SPM Space We propose to dynamically configure part of the CPU Cache as SPM. So there are no proprietary SPM resources and interface needed. This design also allows more flexibility for the software to decide how to use the SPM. Applications can adjust the size of Cache and SPM themselves based on the workload. For example, Random-Access benchmark needs only about 12 KB to support up to 512 in-flight 8B memory requests.

Integration with L2 Controller We propose to integrate the AMU logic with the L2 cache controller. Since the size of L2 is large enough compared to register file and reserve part of L2 will not affect much performance as L1. The logic implements the management and execution of asynchronous access requests and the engine to move data between SPM and memory controller.

Because the main metadata of memory requests maintained by AMU are stored in the SPM, the extra storage overhead is only about a few KB and does not vary when the required MLP increases.

AMU can work with a standard memory controller for local memory or far memory access. However to better support various asynchronous memory instructions with rich semantics and various far memory resources, there should also be newly designed memory controllers that can do the transformation between local bus requests and network packets, such as packing, unpacking, filtering, compressing, routing, etc.

We do not specify the server-side of memory resources. For asynchronous memory access supported by AMU, there are no latency or granularity limitations for memory servers. Different kinds of memory resources can be accessed through a unified interface. That is the separation of memory organization and memory access.

4. Early evaluation

To demonstrate the concepts of AMU, an early simulator prototype has been built. We modified GEM5 to implement a cycle-accurate model of AMU and evaluated it by running Random-Access benchmark

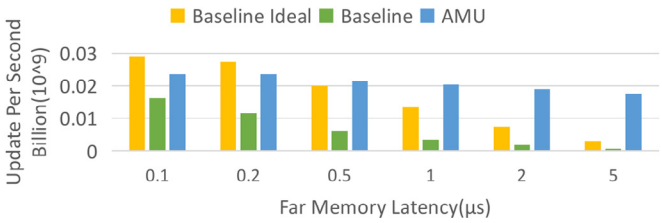

Fig. 3. Performance of random access benchmark.

from HPCC (shown in Fig. 3). The far memory latency is set to several different values (from 0.1 μs to 5 μs) respectively. 64 KB of the 256 KB Cache Capacity are reserved for SPM. AMU performs similarly across different configurations, while the performance of baseline drops rapidly when access latency increases. The results show that AMU can better tolerate a large range of access latency.

5. Discussion

5.1. Efficient ID management

When initiating an asynchronous access request, an id needs to be assigned to the request and be released after finishing. Since the id is needed every time an asynchronous access is initiated, the overhead of id management must be as low as possible. For this goal, we chose to design an efficient hardware id management mechanism instead of leaving id management to software. Also, this mechanism should be integrated with out-of-order and speculation mechanisms smoothly.

5.2. Consistency

As mentioned in Section 2, this work relies on software to handle the consistency. For many data-parallel programs (such Key-Value databases and graph processing [13], etc.), they easily apply coroutine model mentioned above. As each interleaved coroutines processes independent data, thus naturally avoiding data consistency problems.

For other programs that need strong consistency, it is possible to use a combination of hardware and software solution to handle the consistency problem of asynchronous access to far memory. For example, consistency checking can be implemented in local memory or local cache. Applications can check the consistency of requests locally before invoking asynchronous accesses to far memory. In addition, some explicit and efficient locking mechanisms might also be provided by hardware to ensure consistency. Furthermore, software can deal with locking asynchronously to avoid blocking. Although these approaches introduce extra complexity, we argue that the overhead is acceptable comparing with the benefit.

5.3. Comparison with prefetching

There are three major differences between asynchronous memory access and prefetching. First, prefetching mechanisms do not provide any method to query whether prefetch requests have been completed. Thus, applications cannot know if the data has been transferred to local cache. This may impact the efficiency of the application as the access latency is distributed in a wide range. Second, modern processors' prefetching mechanisms are also limited by the number of MSHR entries, while the proposed AMU fully bypass the MSHRs. Third, the memory space for prefetching results is not reserved so prefetched data might be lost before access.

6. Future work

In this paper, only the basic instructions and structures of AMU are presented. Further design and evaluation are undertaking. The AMU design can be easily extended support more memory access protocols. For example, we can add configuration registers and instructions for issuing processing-in-memory related requests. The AMU will enable more programming flexibility if the underlying memory system support more richer semantics, such as message interface based memory systems [14].

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work is partially supported by Strategic Priority Research Program of Chinese Academy of Sciences (Grant No. XDC05030400), and Huawei Technologies Company, Ltd.

References

[1] M.K. Aguilera, N. Amit, I. Calciu, X. Deguillard, J. Gandhi, P. Subrahmanyam, L. Suresh, K. Tati, R. Venkatasubramanian, M. Wei, Remote memory in the age of fast networks, in: Proceedings of the 2017 Symposium on Cloud Computing, in: SoCC '17, Association for Computing Machinery, New York, NY, USA, 2017, pp. 121–127, http://dx.doi.org/10.1145/3127479.3131612.

[2] openCAPI specification, 2017, Online URL http://opencapi.org. (Accessed Febrary 2022).

[3] Gen-Z specification, 2018, Online URL https://genzconsortium.org/specifications. (Accessed Febrary 2022).

[4] Compute express link, 2022, Online URL https://www.computeexpresslink.org/. (Accessed Febrary 2022).

[5] C. Pinto, D. Syrivelis, M. Gazzetti, P. Koutsovasilis, A. Reale, K. Katrinis, H.P. Hofstee, 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, 2020, pp. 868–880, http://dx.doi.org/10.1109/MICRO50266.2020.00075.

[6] H.-K. Liu, D. Chen, H. Jin, X.-F. Liao, B. He, K. Hu, Y. Zhang, A survey of non-volatile main memory technologies: State-of-the-arts, practices, and future directions, J. Comput. Sci. Tech. 36 (1) (2021) 4–32, http://dx.doi.org/10.1007/s11390-020-0780-z.

[7] Intel optane persistent memory, 2022, Online URL https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html. (Accessed Febrary 2022).

[8] A. Lagar-Cavilla, J. Ahn, S. Souhlal, N. Agarwal, R. Burny, S. Butt, J. Chang, A. Chaugule, N. Deng, J. Shahid, G. Thelen, K.A. Yurtsever, Y. Zhao, P. Ranganathan, Software-defined far memory in warehouse-scale computers, in: Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, in: ASPLOS '19, Association for Computing Machinery, New York, NY, USA, 2019, pp. 317–330, http://dx.doi.org/10.1145/3297858.3304053.

[9] J. Tuck, L. Ceze, J. Torrellas, Scalable cache miss handling for high memory-level parallelism, in: 2006 39th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO'06, IEEE, 2006, pp. 409–422.

[10] M. Asiatici, P. Ienne, 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, 2019, pp. 310–319.

[11] S.T. Srinivasan, R. Rajwar, H. Akkary, A. Gandhi, M. Upton, Continual flow pipelines: achieving resource-efficient latency tolerance, IEEE Micro 24 (6) (2004) 62–73.

[12] A. Hilton, S. Nagarakatte, A. Roth, iCFP: Tolerating all-level cache misses in in-order processors, in: 2009 IEEE 15th International Symposium on High Performance Computer Architecture, IEEE, 2009, pp. 431–442.

[13] T.J. Ham, L. Wu, N. Sundaram, N. Satish, M. Martonosi, Graphicionado: A high-performance and energy-efficient accelerator for graph analytics, in: 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO, IEEE, 2016, pp. 1–13.

[14] L.-C. Chen, M.-Y. Chen, Y. Ruan, Y.-B. Huang, Z.-H. Cui, T.-Y. Lu, Y.-G. Bao, MIMS: Towards a message interface based memory system, J. Comput. Sci. Tech. 29 (2) (2014) 255–272, http://dx.doi.org/10.1007/s11390-014-1428-7.