

Active Access: A Mechanism for High-Performance Distributed Data-Centric Computations

Maciej Besta
Department of Computer Science
ETH Zurich
maciej.best@inf.ethz.ch

Torsten Hoefler
Department of Computer Science
ETH Zurich
htor@inf.ethz.ch

ABSTRACT

Remote memory access (RMA) is an emerging high-performance programming model that uses RDMA hardware directly. Yet, accessing remote memories cannot invoke activities at the target which complicates implementation and limits performance of data-centric algorithms. We propose Active Access (AA), a mechanism that integrates well-known active messaging (AM) semantics with RMA to enable high-performance distributed data-centric computations. AA supports a new programming model where the user specifies handlers that are triggered when incoming puts and gets reference designated addresses. AA is based on a set of extensions to the Input/Output Memory Management Unit (IOMMU), a unit that provides high-performance hardware support for remapping I/O accesses to memory. We illustrate that AA outperforms existing AM and RMA designs, accelerates various codes such as distributed hash tables or logging schemes, and enables new protocols such as incremental checkpointing for RMA. We also discuss how extended IOMMUs can support a virtualized global address space in a distributed system that offers features known from on-node memory virtualization. We expect that AA and other IOMMU features can enhance the design of HPC operating and runtime systems in large computing centers.

1. INTRODUCTION

Scaling on-chip parallelism alone cannot satisfy growing computational demands of datacenters and HPC centers with tens of thousands of nodes [15]. Remote direct memory access (RDMA) [35], a technology that completely removes the CPU and the OS from the messaging path, enhances performance in such systems. RDMA networking hardware gave rise to a new class of Remote Memory Access (RMA) programming models that offer a Partitioned Global Address Space (PGAS) abstraction to the programmer. Languages such as Unified Parallel C (UPC) [38] or Fortran 2008 [23], and libraries such as MPI-3 [26] or SHMEM implement the RMA principles and enable direct *one-sided* low-overhead

put and *get* access to the memories of remote nodes, outperforming Message Passing (MP) routines [19].

Active Messages (AMs) [40] are another scheme for improving performance in distributed environments. An active message invokes a handler at the receiver's side and thus AMs can be viewed as lightweight remote procedure calls (RPC). AMs are widely used in a number of different areas (example libraries include IBM's DCMF, IBM's PAMI, Myrinet Express (MX), GASNet [11], and AM++ [41]). Unfortunately, AMs are limited to message passing and cannot be directly used in RMA programming.

In this work we propose *Active Access* (AA), a mechanism that enhances RMA with AM semantics. The core idea is that a remote memory access triggers a user-definable CPU handler at the target. As we explain in § 1.1, AA eliminates some of the performance problems specific to RMA.

Intercepting and processing puts or gets requires control logic to identify memory accesses, to decide when to run a handler, and to buffer necessary data. To preserve all RDMA benefits (OS-bypass, zero-copy, etc.) in AA, we propose a hardware-based design that extends the input/output memory management unit (IOMMU), a hardware unit that supports I/O virtualization. IOMMUs evolved from simple DMA remapping devices to units offering advanced hardware virtualization [8]. Still, AA shows that many potential benefits of IOMMUs are yet to be explored. For example, as we will later show (§ 3.5), moving the notification functionality from the NIC to the IOMMU enables high performance communication with the CPU for AA. Moreover, AA based on IOMMUs can generalize the concept of virtual memory and enable hardware-supported virtualization of networked memories with enhanced data-centric paging capabilities.

In summary, our key contributions are as follows:

- We propose *Active Access* (AA), a mechanism that combines active messages and RMA to improve the performance of RMA-based applications and systems.
- We illustrate a detailed hardware design of simple extensions to IOMMUs to construct AA.
- We show that AA enables a new data-centric programming model that facilitates developing RMA applications.
- We evaluate AA using microbenchmarks and four large-scale use-cases (a distributed hashtable, an access counter, a logging system, and fault-tolerant parallel sort). We show that AA outperforms other communication schemes.
- We discuss how the IOMMU could enable hardware-based virtualization of remote memories.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ICST'15, June 8–11, 2015, Newport Beach, CA, USA.

Copyright is held by the owner/author(s). Publication rights licensed to ACM.

ACM 978-1-4503-3559-1/15/06 ...\$15.00.

<http://dx.doi.org/10.1145/2751205.2751219>.

1.1 Motivation

Consider a distributed hashtable (DHT): RMA programming improves its performance in comparison to MP 2-10 times [19]. Yet, hash collisions impact performance as handling them requires to issue many expensive remote atomics (see § 4). Figure 1 shows how the performance varies by a factor of ≈ 10 with different collision rates.

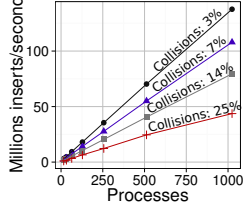


Figure 1: Inserts/s in our RMA hashtable (§ 1.1)

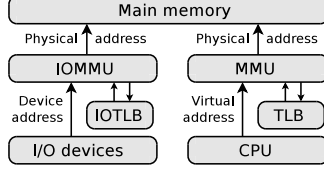


Figure 2: Comparison of the IOMMU and the MMU (§ 2.2)

We will show later (§ 4) how AA reduces the number of remote accesses from six to *one*. Intuitively, the design of AA, based on IOMMU remapping logic, intercepts memory requests and passes them for direct processing to the local CPU. Thus, AA combines the benefits of AMs and OS-bypass in RMA communications.

2. BACKGROUND

We now briefly outline RMA programming. Then, we discuss the parts of the IOMMU design (DMA remapping, IOMMU paging) that we later use to design Active Access.

2.1 RMA Programming Models

RMA is a programming model in which processes communicate by directly accessing one another’s memories. RMA is typically built on OS-bypass RDMA hardware to achieve highest performance. Thus, RMA *put* (writes to remote memories) and *get* (reads from remote memories) have very low latencies, and significantly improve performance over MP [19]. RDMA is available in virtually all modern networks (e.g., IBM’s Cell on-chip network, InfiniBand [37], IBM PERCS, iWARP, and RoCE). In addition, numerous existing languages and libraries based on RMA such as UPC, Titanium, Fortran 2008, X10, Chapel, or MPI-3.0 RMA are actively developed and offer unique features for parallel programming. Consequently, the number of applications in the RMA model is growing rapidly.

Here, we use *source* or *target* to refer to a process that issues or is targeted by an RMA access. We always use *sender* and *receiver* to refer to processes that exchange messages.

2.2 IOMMUs

IOMMUs are located between peripheral devices and main memory and can thus intercept any I/O traffic. Like the well-known memory management units (MMUs), they can be programmed to translate device addresses to physical host addresses. Figure 2 shows the similarities between MMUs and IOMMUs. An IOMMU can virtualize the view of I/O devices and control access rights to memory pages.

All major hardware vendors such as IBM, Intel, AMD, Sun, and ARM offer IOMMU implementations to support virtualized environments; Table 1 provides an overview. We conclude that IOMMUs are a standard part of modern computer architecture and the recent growth in virtualization for cloud computing ensures that they will remain important in

Vendor	IOMMU and its application
AMD	GART [1]: address translation for the use by AGP
	DEV [8]: memory protection
	AMD IOMMU [2]: address translation & memory protection
IBM	Calgary PCI-X bridge [8]: address translation, isolation
	DART [8]: address translation, validity tracking
	IOMMU in Cell processors [8]: address translation, isolation
	IOMMU in POWER5 [5]: hardware enhanced I/O virtualization
	TCE [21]: enhancing I/O virtualization in pSeries 690 servers
Intel	VT-d [22]: memory protection, address translation
ARM	CoreLink SMMU [25]: memory management in System-on-Chip (SoC) bus masters, memory protection, address translation
PCI-SIG	IOV & ATS [33]: address translation, memory protection
Sun	IOMMU in SPARC [29]: address translation, memory protection
SolarFlare	IOMMU in SF NICs [34]: address translation, memory protection

Table 1: An overview of existing IOMMUs (§ 2.2).

the future. However, IOMMUs are a relatively new concept with many unexplored opportunities. Their ability to intercept any memory access and provide full address space virtualization can be the basis for many novel mechanisms for managing global (RDMA) address spaces.

To be as specific as possible, we selected Intel’s IOMMU technology [22] to explain concepts of generic IOMMUs. Other implementations vary in some details but share the core features (per-page protection, DMA remapping, etc.).

DMA Remapping DMA remapping is the IOMMU function that we use extensively to design AA. The IOMMU remapping logic allows any I/O device to be assigned to its own private subset of host physical memory that is isolated from accesses by other devices. To achieve this, IOMMUs utilize three types of remapping structures (all located in main memory): *root-entry* tables, *context-entry* tables, and IOMMU *page tables*. The first two are used to map I/O devices to device-specific page tables. To improve the access time, the remapping hardware maintains several caches such as the *context-cache* (device-to-page-table mappings) and the *I/O Translation Lookaside Buffer* (IOTLB) (translations from device addresses to host physical addresses).

Page Tables & Page Faults IOMMU page tables allow to manage host physical memory in a hierarchical way; they are similar to standard MMU page tables (still, MMU and IOMMU page tables are set up independently). A 4-level table allows 4KB page granularity on 64 bit machines (superpages of various sizes are also supported).

IOMMU page tables implement a page fault mechanism similar to MMUs. Every page table entry (PTE) contains two protection bits, W and R, which indicate whether the page is writable and readable, respectively. Any access that violates the protection conditions is blocked by the hardware and a *page fault* is generated, logged, and reported to the OS. The IOMMU logs the fault information using special registers and in-memory *fault logs*. The OS is notified using Message Signal Interrupts (MSI). Every page fault is logged as a fixed-sized *fault entry* that contains the fault metadata (the address of the targeted page, etc.); the data being transferred is discarded. We will later extend this mechanism to log active accesses and their data and to bypass the OS.

3. THE ACTIVE ACCESS MECHANISM

Active Access combines the benefits of RMA and AMs. AMs enhance the message passing model by allowing messages to actively integrate into the computation on the re-

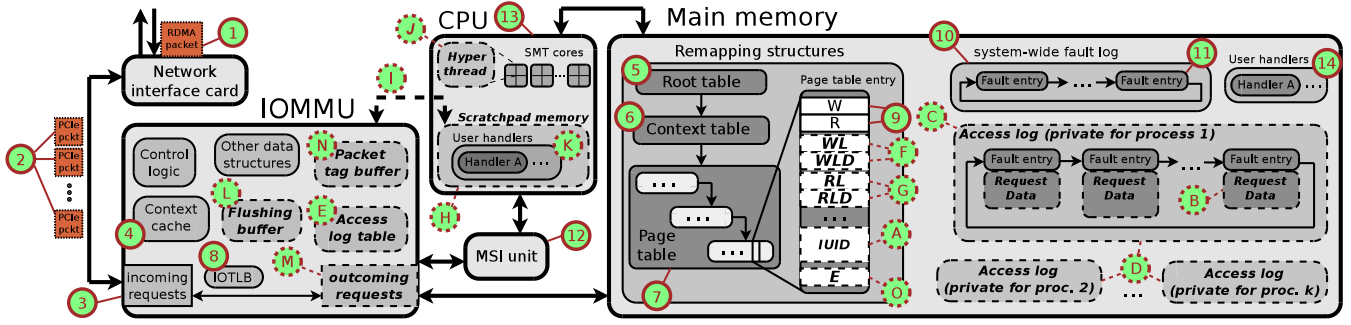


Figure 3: The overview of the IOMMU and the cooperating devices. The proposed extensions are marked with dashed edges and bold-italic text. Solid circles with numbers (1 - 14) indicate the specific steps discussed in detail in § 3.1. Dashed circles (15 - 18) are extensions pointed out in § 3.1-§ 3.7.

ceiver side. In RMA, processes communicate by accessing remote memories instead of sending messages. Thus, an analogous scheme for RMA has to provide the *active* semantics for both types of remote operations; puts and gets become *active puts* (AP) and *active gets* (AG), respectively.

Listing 1 shows the interface of AM and AA. An active message sent to a process receiver_id carries arguments and payload that will be used by a handler identified by a pointer hlr_addr. In AA, the user issues puts and gets at trgt_addr in the address space of a process trgt_id. No handler address is specified. Instead, we enable the user to associate an arbitrary page of data with a selected handler and with a set of additional actions (discussed in § 3.3 and § 3.4). When a put or a get touches such a page, it becomes *active*: first, it may or may not finalize its default memory effects (depending on the specified actions); second, *both* its meta-data *and* data are ultimately processed by the associated handler. AA is fully transparent to RMA and, as Listing 1 shows, it entails no changes to the traditional interface.

```

1 /***** interface of AM *****/
2 void send_active_message(ptr hlr_addr, void* arguments,
   void* payload, int receiver_id) { ... }
3 /***** interface of AA *****/
4 void put(void* trgt_addr, void* data, int trgt_id) {
5   /* Attempt to copy data from the local memory into
6    the memory location trgt_addr of a process trgt_id.*/
7 }
8 void get(void* trgt_addr, void* l_addr, int trgt_id) {
9   /* Attempt to fetch the data from the memory location
10  trgt_addr of a process trgt_id to l_addr.*/
11 }
12 void assoc_page(void* addr, void* act, int hlr_id) {
13   /* Associate a page at addr with actions act and
14  with a handler identified by the id hlr_id. */

```

Listing 1: Interface of Active Messages and Active Access (§ 3)

We now show how to extend IOMMUs to implement the above AA interface and to enable active puts and gets. From now on, we will focus on designs based on PCI Express (PCIe) [32]. We first describe the interactions between an RDMA request and current IOMMUs. The numbers in circles (1 - 14) refer to the corresponding numbers in Figure 3.

3.1 State-of-the-art IOMMU Processing Path

Consider an RDMA put or a get that is issued by a remote process. First, the local NIC receives the RDMA packet (1). The NIC attempts to access the main memory with DMA and thus it generates appropriate PCIe packets (one or more depending on the type of the PCIe transaction [32]) (2). Each packet is intercepted by the IOMMU (3). First, the IOMMU resolves the mapping from the device to its page table (using the packet header [22]). Here, the IOMMU uses the context cache (4) or, in case of a cache miss, it walks the remapping tables (5-6). Finally, the IOMMU obtains the

location of the specific page-table (7). The IOMMU then resolves the mapping from a device address to a physical address using the IOTLB (8) or, in the event of a miss, the page-table (7). When it finds the target PTE, it checks its protection bits W and R (9). The next steps depend on the request type. For puts, if W=1, the value is simply written to the target location. If W=0, the IOMMU raises a page fault and does not modify the page. For gets, if R=1, the request returns the accessed value to the NIC. If R=0, the IOMMU raises a page fault and does not return the value.

Upon a page fault, the IOMMU tries to record the fault information (fault entry) in the system-wide fault log (10) (implemented as an in-memory ring buffer). In case of an overflow (e.g., if the OS does not process the recorded entries fast enough) the fault entry is not recorded. If the fault entry is logged (11), the IOMMU interrupts the CPU (13) with MSI (12) to run one of the specified handlers (14).

We now analyze the extensions that enable active puts/gets (symbols 15 - 18 refer to the related symbols in Figure 3). Our goal is to enable the IOMMU to multiplex intercepted accesses among processes, buffer them in designated memory locations, and pass them for processing to a CPU.

3.2 Processing the Intercepted Data

In the original IOMMU design the fault log (10) is shared by all the processes running on the node where the IOMMU resides; a potential performance bottleneck. In addition, the IOMMU does not enable multiplexing the data coming from the NIC across the processes and handlers, limiting performance in multi/manycore environments. Finally, the data of a blocked RDMA put is lost as the fault log entry only records the address (see steps 9-10). To alleviate these issues, we propose to enhance the design of the IOMMU and its page tables to enable a data-centric multiplexing mechanism in which *the PTEs themselves guide the incoming requests to be recorded in the specified logging data structures and processed by the designated user-space handlers*.

We first add a programmable field *IOMMU User Domain ID* (IUID) to every IOMMU PTE (15). This field enables *associating* pages with user domains. The OS and the NIC can ensure that one IUID is associated with at most one local process, similarly to Protection Domains in RDMA [35]. To add IUID we use bits 52-61 of IOMMU PTEs (ignored by the current IOMMU hardware [22]). We can store 2^{10} domains on each node; enough to fully utilize, e.g., BlueGene/Q (64 hardware threads/node) or Intel Xeon Phi (256 hardware threads/chip). Second, our extended IOMMU logs *both* the generated fault entry *and* the carried data (16) to the *access log*, a new in-memory circular ring buffer (17). A process can have multiple private IUIDs/access logs located in its address space (18). Third, the IOMMU maintains the *access log*

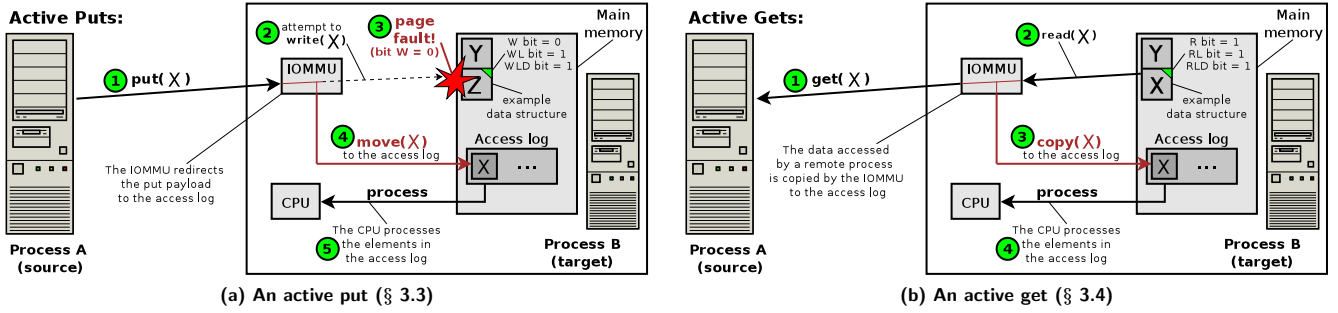





Figure 4: Active puts and gets. Here, the numbers in circles are independent of the numbering in Figure 3.

table , a simple internal associative data structure with tuples (IUID, base, head, tail, size). One entry maps an IUID to three physical addresses (the base, the head, and the tail pointer) and the size of the respective access log ring buffer. The access log table is implemented as content addressable memory (CAM) for rapid access and it can be programmed in the same way as other Intel IOMMU structures [22].

3.3 Controlling Active Puts (APs)

Active puts enable redirecting data coming from the NIC and/or related metadata to a specified access log. Figure 4a illustrates active puts in more detail. Two additional PTE bits control the logging of fault entry and data: WL (Write Log) and WLD (Write Log Data) . If WL=1 then the IOMMU logs the fault entry for the written page and if WLD=1 then the IOMMU logs both the fault entry and the data. The flags W, WL and WLD are independent. For example, an active put page is marked as W=0, WL=1, WLD=1. The standard way, in which IOMMUs manage faults triggered by writes, is defined by the values W=0, WL=1, WLD=0.

3.4 Controlling Active Gets (AGs)

Active gets enable the IOMMU to log a copy of the remotely accessed data locally. When a get succeeds and the returned data is flowing from the main memory to the NIC, it is replicated by the IOMMU and saved in the access log (see Figure 4b). Similar to active puts, two additional PTE bits control the logging behavior of such accesses: RL (Read Log) and RLD (Read Log Data) . If RL=1 then the IOMMU logs the fault entry for the read page and if RLD=1 then the IOMMU logs the fault entry and the returned data.

The proposed control bit extensions for active puts and active gets can easily be implemented in practice. For example, bits 7-10 in the Intel IOMMU PTEs are ignored [22]. These bits can be used to store WL, WLD, RL, and RLD.






3.5 Interactions with the Local CPU

Finally, the IOMMU has to notify the CPU to run a handler to process the logs. Here, we discuss interrupts/polling and we propose a new scheme where the IOMMU directly accesses the CPU, bypassing the main memory.

Interrupts Here, one could use a high-performance MSI wakeup mechanism analogous to the scheme in InfiniBand [37]. The developer specifies conditions for triggering interrupts (when the amount of free space in an access log is below a certain threshold, or at pre-determined intervals). If the access log is sufficiently large and messages are pipelined then the interrupt latency may not influence the overall performance significantly (cf. § 5.2).

Polling As the IOMMU inserts data directly into a user address space, processes can monitor the access log head/-tail pointers and begin processing the data when required.


Polling can be done either directly by the user, or by a run-time system that runs the handlers transparently to the user.

Direct CPU Access This mechanism is motivated by the architectural trends to place scratchpad memories on processing units, a common practice in today's NVIDIA GPUs [31] and several multicore architectures [24]. One could add a scratchpad to the CPU , connect it directly with the IOMMU , and place the head/tail pointers in it. A dedicated hyperthread  polls the pointers and runs the handlers if a free entry is available . If the size of the handler code is small, it can also be placed in the scratchpad, further reducing the number of memory accesses .

The IOMMU and the CPU also have to synchronize while processing the access log. This can be done with a simple lock for mutual access. The IOMMU and the CPU could also synchronize with the pointers from the access log table.

3.6 Consistency Model

We now enhance AA to enable a weak consistency model similar to MPI-3 RMA [20]. In RMA, a blocking flush synchronizes nonblocking puts/gets. In AA, we use an *active flush* (flush(int target_id)) to enforce the completion of active accesses issued by the calling process and targeted at target_id. One way to implement active flushes could be to issue an active get targeted at a special designated *flushing page* in the address space of target_id. The IOMMU, upon intercepting this get, would wait until the CPU processes the related access log and then it would finish the get to notify the source that the accesses have been committed.

Extending the IOMMU We add an IOMMU internal data structure called the *flushing buffer*  to store tuples (address, IUID, active, requester-ID, tag), where address is the address of the flushing page, active is a binary value initially set to false, and requester-ID, tag are values of two PCIe packet fields with identical names; they are initially zeroed and we discuss them later in this section. The flushing buffer is implemented as CAM for rapid access.

Selecting a Flushing Page Here, the system could reserve a high virtual address for this purpose. We then add a respective entry (with the selected address and the related IUID) to the flushing buffer.

Finalizing an Active Flush The IOMMU intercepts the issued get, finds the matching entry in the flushing buffer, sets active=true, copies the values of the tag and requester-ID PCIe fields to the matching entry, and discards the get. Processing of the targeted access log is then initiated with any mechanism from § 3.5, depending on user's choice.

Alternative Mechanism The proposed consistency mechanism sacrifices one page from the user virtual address space. To alleviate this issue, we propose a second scheme similar to the semantics offered by, e.g., GASNet [11]. Here, AA



does not guarantee any consistency. Instead, it allows the user to develop the necessary consistency by issuing a *reply* (implemented as an active put) from within the handler. This reply informs which elements from the access log have been processed. To save bandwidth, replies can be batched. The reply would be targeted at a designated page (with bits $W=0$, $WL=1$, $WLD=1$) with an IUID pointing to a designated access log. The user would poll the log and use the replies to enforce an arbitrary consistency.

Mixing AA/RMA Accesses At times, mixing AA and RMA puts/gets may be desirable (see § 4.1). The consistency of such a mixed scheme can be managed with active and traditional RMA flushes as these two calls are orthogonal. The completion of pending AA/RMA accesses is enforced with AA/RMA flushes, respectively.

3.7 Hardware Implementation Issues

We now describe solutions to several PCIe and RDMA control flow, ordering, and backward compatibility issues in the proposed extensions. If the reader is not interested in these details then they may skip this part and proceed to § 4. Numbers and capitals in circles refer to Figure 3.

Logging Data of PCIe Write Requests Every RDMA put is translated into one or more PCIe write requests flowing from NIC to main memory. The ordering rules for *Posted Requests* from the PCIe specification (§ 2.4.1 in [32], entry A2a) ensure that the packets for the same request arrive in order and may thus be simply appended to the log.



Logging Data of PCIe Read Requests A PCIe read transaction consists of one read request (issued by the NIC) and one or more read replies (issued by the memory controller). The IOMMU has to properly match the incoming and outgoing PCIe packets. For this, we first enable the IOMMU to intercept PCIe packets flowing back to the NIC  (standard IOMMUs process only incoming memory accesses). Second, we add the *packet tag buffer*  (implemented as CAM) to the IOMMU to temporarily maintain information about PCIe packets. The IOMMU would add the transaction tags of incoming PCIe read requests that access a page where $RLD=1$ to the tag buffer. PCIe read reply packets are then matched against the buffer and logged if needed.

We require the tag buffer as PCIe read replies only contain seven lower bits of the address of the accessed memory region (see § 2.2.9 in [32]), preventing the IOMMU from matching incoming requests with replies. The PCIe standard also ensures ordering of read replies (see § 2.3.1.1 in [32]).

Order of PCIe Packets from Multiple Devices The final ordering issue concerns multiple RMA puts or gets concurrently targeted at the same IOMMU. If several multi-packet accesses originate from different devices then the IOMMU may observe an arbitrary interleaving of PCIe packets. To correctly reassemble the packets in the access log, we extend the tag buffer so that it also stores pointers into the access log. Upon intercepting the first PCIe packet of a new PCIe transaction, the IOMMU inserts a tuple (*transaction-tag*, *tail*) into the tag buffer, records the packet in the access log, and adds the size of the *whole* PCIe transaction to the tail pointer. Thus, if some transactions interleave, the IOMMU leaves “holes” in the access log and fill these holes when appropriate PCIe packets arrive¹. The IOMMU removes an

entry from the buffer after processing the last transaction packet. To ensure that the CPU only processes packets with no holes, the IOMMU increments *tail* or sets appropriate synchronization variables only when each PCIe packet of the next transaction is recorded in the respective access log.

Control Flow IOMMUs, unlike MMUs, cannot suspend a remote process and thus buffers may overflow if they are not emptied fast enough. To avoid data loss, we utilize the back-pressure mechanism of the PCIe transaction layer protocol (TLP) as described in § 2.6.1. in [32]. This will eventually propagate through a reliable network and block the sending process(es). Issues such as head of line blocking and deadlocks are similar to existing reliable network technologies and require efficient programming at the application layer (regular emptying of the queues). Head of line blocking can also be avoided by dropping packets and retransmission [6].

Support for Legacy Codes Some codes may rely on the default IOMMU behavior to buffer the metadata in the default fault log . To cover such cases, we add the E bit  to IOMMU PTEs to determine if the page fault is recorded in the fault log ($E=0$) or in one of the access logs ($E=1$).

4. ACTIVE ACCESS PROGRAMMING

We now discuss example RMA-based codes that leverage AA. AA improves the application performance by reducing the amount of communication and remote synchronization. First, it reduces the number of puts, gets, and remote atomic operations in distributed data structures and other codes that perform complex remote memory accesses. For example, enqueueing an element into a remote queue costs at least two remote accesses (atomically get and increment the tail pointer and put the element). With AA, this would be a simple put to the list address and a handler that inserts the element; our DHT example is very similar. Second, since handlers are executed by local cores, the usual on-node synchronization mechanisms are utilized with no need to issue expensive remote synchronization calls such as remote locks.

4.1 Designing Distributed Hash Tables

DHTs are basic data structures that are used to construct distributed key-value stores such as Memcached [17]. In our design, the DHT is open and each process manages its part called the local *volume*. The volume consists of a table of elements and an overflow heap for elements with hash collisions. Both the table and the heap are implemented as fixed-size arrays. To avoid costly array traversals, pointers to most recently inserted items and to the next free cells are stored along with the remaining data in each local volume.

Due to space constraints we discuss inserts and only briefly lookups and deletes. In RMA, inserts are based on atomics (Compare-and-Swap and Fetch-and-Op, denoted as *cas* and *fao*), RMA puts (*rma_put*) and RMA flushes (*rma_flush*); see Listing 2. For simplicity we assume that atomics are blocking. The semantics of CAS are as follows: `int cas(elem, compare, target, owner)`; if `compare == target` then `target` is changed to `elem` and its previous value is returned. For FAO we have `int fao(op, value, target, owner)`; it applies an atomic operation `op` to `target` using a parameter `value`, and returns `target`’s previous value. In both *cas* and *fao*, *owner* is the id of the process that owns the targeted address space. The semantics for *rma_put* and *rma_flush* are the same as for AA puts and flushes (cf. § 3, § 3.6). \emptyset indicates that the specific array cell is empty. To insert *elem* we first issue a *cas* (line 9). Upon a

¹PCI Express 3.0 Specification limits the PCIe transaction size to 4KB. Thus, the maximum size of an active put or get also amounts to 4 KB. This limitation can be easily overcome in future PCIe systems.

```

1 /* Volume is a structure that contains the fields:
2 owner: the id of the volume owner; vol_size: volume size,
3 elems[]: the table + the overflow heap; each cell contains
   two subfields: elem (the actual value) and ptr (the
   pointer to the next element),
4 next_free_cell: a ptr to the next free cell in the heap,
5 last_ptr[]: pointers to the most recent elements */
6
7 void insert(int elem, Volume v) { //put elem into volume v
8   int pos = hash(elem); //get the position of elem in v
9   if(cas(elem,0,v.elems[pos].elem,v.owner) != 0) {
10     int free_cell = fao(SUM,1,v.next_free_cell,v.owner);
11     if(free_cell >= v.vol_size) { /*an overflow - resize*/
12       rma_put(elem,v.elems[free_cell].elem,v.owner);
13       rma_flush(v.owner);
14       int prev_ptr=fao(REPLACE,free_cell,v.last_ptr[pos],
15         v.owner);
16       if(cas(free_cell,0,v.elems[pos].ptr,v.owner) != 0) {
17         rma_put(free_cell,v.elems[prev_ptr].ptr,v.owner);
18         rma_flush(v.owner); } } }

```

Listing 2: Insert in the traditional RMA-based DHT

collision we acquire a new element in the overflow heap (line 10). We then insert `elem` into the new position (lines 12-13), update the respective last pointer and the next pointer of the previous element in the heap (lines 14-17).

Implementation of Inserts with Active Puts We now accelerate inserts with AA. We present the multi-threaded code in Listing 3. The inserting process calls `insert` (lines 1-3). The PTEs of the hash table data are marked with `W=0`, `WL=1`, `WLD=1`; thus, the metadata and the data from the put is placed in the access log. The CPU then (after being interrupted or by polling the memory/scratchpad) executes `insert_handler` to insert the elements into the local volume (lines 5-10). Here, we assume that a thread owns one access log and that the size of the access log is divisible by the size of `int`. Elements are inserted with `local_insert`, a function similar to `insert` from Listing 2. The difference is that each call is local (we thus skip the `lv.owner` argument).

Synchronization AA handlers are executed by the local CPU, thus, `local_insert` requires no synchronization with remote processes. In our code we use local atomics, however, other simple local synchronization mechanisms (e.g., locks or hardware transactional memory) may also be utilized.

Consistency The proposed DHT is loosely consistent. For implementing any other consistency (e.g., sequential consistency) one can use either active flushes or enforce the required consistency using replies from within the handler.

Lookups Contrary to inserts, lookups do not generate hash collisions that entail multiple memory accesses. Thus, we propose to implement a lookup as a single *traditional* RMA get, similarly to [14]. For this, we mark the PTEs associated with the hashtable data as `R=1`, `RL=0`, `RLD=0`. Here, we assume that DMA is cache coherent (true on, e.g., Intel x86 [14]) and that RMA gets are aligned. As the DHT elements are word-size integers, a get is atomic with respect to any concurrent accesses from Listing 3. Consistency with other lookups and with inserts can be achieved with RMA and active flushes, respectively. More complicated schemes that fetch the data from the overflow heap are possible; the details are outside the scope of the paper.

Deletes A simple protocol built over active puts performs deletes. Here, we use a designated page P marked as `W=0`, `WL=1`, `WLD=1`. The implementation of the delete issues an active put. This put is targeted at P and it contains a key of the element(s) to be deleted. The IOMMU moves the keys to a designated access log and a specified handler uses them to remove the elements from the local volume.

```

1 void insert(int elem, Volume v) {
2   put(elem, v.elems[hash(elem)].elem, v.owner);
3 }
4
5 void insert_handler(Access_log log) {
6   while(log.tail != log.head) {
7     local_insert(*log.tail); log.tail += sizeof(int);
8     if(log.tail == log.base + log.size) {
9       log.tail = log.base;
10    } } }
11
12 void local_insert(int elem) { //lv is the local DHT volume
13   int pos = hash(elem); //get the position of elem in lv
14   if(cas(elem,0,lv.elems[pos].elem) != 0) {
15     int free_cell = fao(SUM,1,lv.next_free_cell);
16     if(free_cell >= lv.vol_size) { /*an overflow - resize*/
17       lv.elems[free_cell].elem = elem;
18       int prev_ptr=fao(REPLACE,free_cell,lv.last_ptr[pos]);
19       if(cas(free_cell,0,lv.elems[pos].ptr) != 0) {
20         lv.elems[prev_ptr].ptr = free_cell; } } }

```

Listing 3: Insert in the AA-based DHT

4.2 Collecting Statistics on Memory Accesses

Recent work [18] presents an *active* key-value store “Comet”, where automated gathering of statistics is one of the key functionalities. Such systems are usually implemented in the application layer, which significantly limits their performance. Architectures based on traditional RMA suffer from issues similar to the ones described in § 4.1.

AA enables hardware-based gathering statistics. For example, to count the number of puts or gets to a data structure, one has to appropriately set the control bits in the PTEs that point to the memory region where this structure is located: `W=1`, `WL=1`, `WLD=0` (for puts), and `R=1`, `RL=1`, `RLD=0` (for gets). Thus, the IOMMU ignores the data and logs only metadata that is later processed in a handler to generate statistics; the processing can be enforced with active flushes. This mechanism would also improve the performance of cache eviction schemes in memcached applications.

AA enables gathering separate statistics for each page of data. Yet, sometimes a finer granularity could be required to count accesses to elements of smaller sizes. In such cases one could place the respective elements in separate pages.

4.3 Enabling Incremental Checkpointing

Recent predictions about mean time between failures (MTBF) of large-scale systems indicate failures every few hours [9]. Fault tolerance can be achieved with various mechanisms. In *checkpoint/restart* [9] all processes synchronize and record their state to memories or disks. Traditional checkpointing schemes record the same amount of data during every checkpoint. However, often only a small subset of the application state changes between two consecutive checkpoints [39]. Thus, saving all the data wastes time, energy, and bandwidth. In *incremental checkpointing* only the modified data is recorded. A popular scheme [39] tracks data modifications at the granularity of pages and uses the dirty bit (DB) to detect if a given page requires checkpointing. This scheme cannot be directly applied to RMA because memory accesses performed by remote processes are not tracked by the MMU paging hierarchy [35].

AA enables incremental checkpointing in RMA codes. The combination `W=1`, `WL=1`, `WLD=0` (set to the data that requires checkpointing) enables tracking the modified pages. To take a checkpoint all the processes synchronize and process the access logs to find and record the modified data. Our incremental checkpointing mechanism for tracking data modifications is orthogonal to the details of synchronization and recording; one could use any available scheme [9].

Most often both remote and local accesses modify the memory. The latter can be tracked by the MMU and any existing method (e.g., the DB scheme [39]) can be used. While checkpointing, every process parses both the access log and the MMU page table to track both types of memory writes.

4.4 Reducing the Overheads of Logging

Another fault tolerance mechanism for RMA is uncoordinated checkpointing combined with *logging of puts and gets* where the crashed processes repeat their work and replay puts and gets that modified their state before the failure; these puts and gets are logged during the application runtime [9]. While logging puts is simple and does not impact performance, logging gets wastes network bandwidth because it requires transferring additional data [9]. We now describe this issue and propose a solution based on AA.

Logging Gets in Traditional RMA A get issued by process A and targeted at B fetches data from B’s memory and impacts the state of A. Thus, if A fails and begins recovery, it has to replay this get. Still, A cannot log this get locally because the contents of its memory are lost after the crash (see Fig. 5, part 1). Thus, B can log the get [9].

The core problem in RMA is that B knows nothing of gets issued by A, and cannot actively perform any logging action. It means that A has to wait for the data to be fetched from B and only then can it *send this data back to B*. This naive scheme comes from the fundamental rules of one-sided RMA communication: B is completely oblivious to any remote accesses to its memory [9] (cf. Fig. 5, part 2).

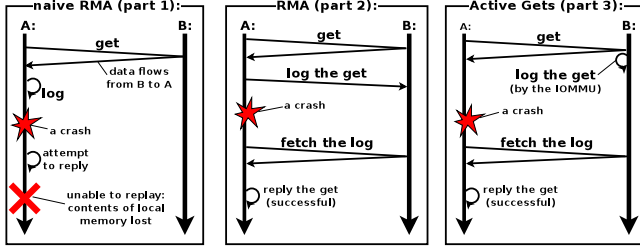


Figure 5: Logging and replaying issued operations in RMA and AA.

Improving the Performance with Active Gets In AA, the IOMMU can intercept incoming gets and log the accessed data locally. First, we set up PTEs in the IOMMU page table that point to the part of memory that is targeted by gets. We set the control bits ($R=1, RL=1, RLD=1$) in these PTEs to make each get touching this page active. Every such get triggers the IOMMU to copy the accessed data into the access log annihilating the need for the source to send the same data back (see Figure 5, part 3). During the recovery a crashed process fetches the logs and then uses them to recover to the state before the failure. We omit further details of this scheme as this is outside the scope of this paper; example protocols (e.g., for clearing the logs or replaying puts and gets preserving the RMA consistency order) can be found in the literature [9].

5. EVALUATION

To evaluate AA we first conduct cycle-accurate simulations to cover the details of the interaction between the NIC, the IOMMU, the CPU, and the memory system. Second, we perform simplified large-scale simulations to illustrate how AA impacts the performance of large-scale codes.

5.1 Microbenchmarks

We first perform cycle-accurate microbenchmarks that evaluate the performance of data transfer between two machines connected with an Ethernet link. We compare system configurations without the IOMMU (no-iommu) and with the extended IOMMU presented in this paper (e-iommu). We use the gem5 cycle-accurate full-system simulator [10] and a standard testbed that allows for modeling two networked machines with in-order CPUs, Intel 8254x 1GbE NICs with Intel e1000 driver, a full operating system, TCP/IP stack, and PCIe buses. The utilized OS is Ubuntu 11.04 with pre-compiled 3.3.0-rc3 Linux kernel that supports 2047MB memory. We modify the simulated system by splitting the PCIe bus and inserting an IOMMU in between the two parts. We model the IOMMU as a bridge with an attached PTE cache (IOTLB). The bridge provides buffering and a fixed delay for passing packets; we set the delay to be $70ns$ for each additional memory access. We also use a $5ns$ delay for simulating IOMMU internal processing. We base these values on the L1/memory latencies of the simulated system.

We first measure the performance of data transfer with PktGen [28] (a high-speed packet generator) enhanced with netmap [36] (a framework for fast packet I/O). Second, we evaluate the performance of a TCP and a UDP stream with netperf, a popular benchmark for measuring network performance. We show the results in Figures 6a-6b. The IOMMU presence only marginally affects the data transfer bandwidth (the difference between the no-iommu and e-iommu is 1-5% with no-iommu, as expected, being marginally faster).

We also simulate a hashtable workload of one process inserting new elements at full link bandwidth into the memory of the remote machine; the results are presented in Figure 6c. Here, we compare AA with a traditional RMA design of the DHT. As the collision rate increases, the performance of both designs drops due to a higher number of memory accesses. Still, AA is ≈ 3 times more performant than RMA.

5.2 Evaluation of Large-Scale Applications

The second performance-related question is how the AA semantics, implemented using the proposed IOMMU design, impacts the performance of *large-scale* codes. To be able to run large-scale benchmarks on a real supercomputer we simplify the simulation infrastructure. We simulate one-sided RMA calls with MPI point-to-point messages. We replace one-directional RMA puts with a single message and two-directional RMA calls (gets and atomics) with a pair of messages exchanged by the source and target, just like packets in hardware. We then emulate extended IOMMUs by appropriately stalling message handlers. As the IOMMU performs data replication and redirection bypassing the CPU, there are four possible sources of such overheads: interrupts, memory accesses due to the logging, IOMMU page table lookups, and accesses to the scratchpads on the CPU.

First, we determine the interrupt and memory access latencies on our system to be $3\mu s$ and $\approx 70ns$, respectively. Second, we simulate the IOTLB and page table lookups varying several parameters (PTE size, associativity, eviction policy). Finally, we assume that an access to the scratchpad to notify a polling hyperthread is equal to the cost of an L3 access and we evaluate it to $\approx 15ns$.

All the experiments are executed on the CSCS Monte Rosa Cray XE6 system. Each node contains four 8-core AMD processors (Opteron 6276 Interlagos 2.3 GHz) and is

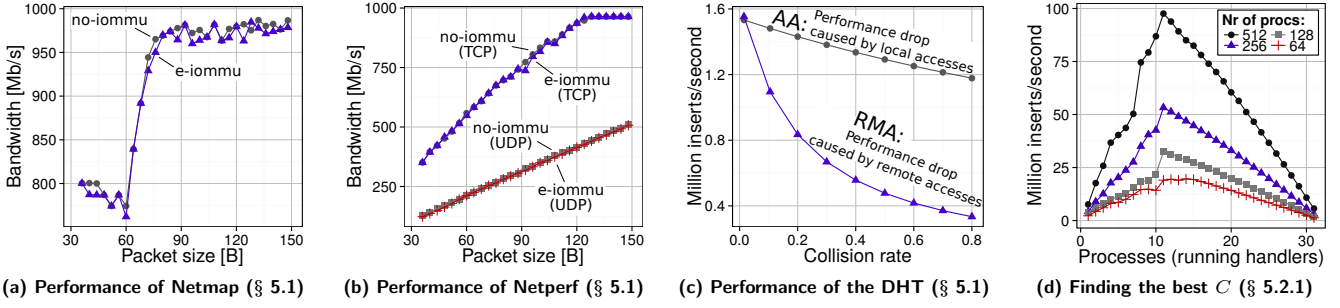


Figure 6: Microbenchmarks (Figures 6a-6c) and finding optimum configuration for AA-Onload (Figure 6d).

connected to a 3D-Torus Gemini network. We use 32 processes/node and the GNU Environment 4.1.46 for compiling.

We compare the following communication schemes:

AA-Int, AA-Poll, AA-SP: AA based on the IOMMU communicating with the CPU using: interrupts, polling the main memory, and accessing the scratchpad, respectively.

RMA: traditional RMA representing RDMA architectures.

AM: an AM scheme in which processes poll at regular intervals to check for messages. Note that this protocol is equivalent to traditional message passing.

AM-Exp: an AM variant based on exponential backoff to reduce polling overhead. If there is no incoming message, we double the interval after which a process will poll.

AM-Onload: an AM scheme where several cores are only dedicated to running AM handlers and constantly poll on flags that indicate whether new AMs have to be processed.

AM-Ints: an AM mechanism based on interrupts generated by the NIC that signal to the CPU it has to run the handler.

5.2.1 Distributed Hashtable

We implement eight hashtable variants using each of the schemes above. Here, processes insert random elements with random keys (delete evaluation gives similar performance and is skipped due to space constraints). Each DHT volume can contain 2^{21} elements. We vary different parameters to cover a broad spectrum of possible scenarios. First, we study the scalability by changing the number of inserting processes P . Second, we evaluate benchmarks with different numbers of hash collisions (R_{cols} , the ratio between the number of hash collisions and the total number of inserts). Third, we simulate different applications by varying computation ratios (R_{comp} , the ratio between the time spent on local computation and the total experiment runtime). We also vary the IOTLB parameters: IOTLB size, associativity, and the eviction policy. Finally, we analyzed two variants of AA-Ints/AM-Ints in which an interrupt is issued every 10^1 and 10^2 inserts (the differences in performance were negligible ($<5\%$); we only report numbers for the former).

AM-Onload depends on the number of cores (C) per node that are dedicated to processing AM requests. Thus, to make the comparison fair, we run AM-Onload for every C between 1 and 31 in order to find the most advantageous configuration for every experiment. Figure 6d shows that $C = 11$ delivers maximum performance. If $C < 11$ the cores become congested and the performance decreases. $C > 11$ limits performance as receiving cores become underutilized.

Varying P and R_{cols} Figure 7a shows the results for $R_{cols} = 5\%$ and Figure 7b for $R_{cols} = 25\%$. Here, both AA-SC and AA-Poll outperform all other schemes by a factor of ≈ 2 . As expected, AA-SC is slightly ($\approx 1\%$) more performant than AA-Poll. AA-Ints is comparable to AM-Ints;

both mechanisms suffer from interrupt latency overheads. The reasons for performance differences in the remaining schemes are as follows: in AM-Exp and AM the computing processes have to poll on the receive buffer and, upon active message arrival, extract the payload. In AA this part is managed by the IOMMU and the computing processes only have to insert the elements into the local hashtable. AM-Onload devotes smaller number of processes to compute and thus, even in its best configuration, cannot outperform AA. RMA issues costly atomics for every insert and 6 more remote operations for every collision, degrading performance.

Varying R_{comp} Increasing R_{comp} from 0% to 95% did not significantly influence the performance patterns between evaluated schemes. The only noticeable effect was that the differences between the results of respective schemes were smaller (which is expected as scaling R_{comp} reduces appropriately the amount of communication per time unit).

Varying the IOTLB Parameters We now analyze the influence of various IOTLB parameters on the performance of DHT; the results are presented in Figure 7c. The name of each plot encodes the used eviction policy (lru: least recently used, rnd: random) and associativity (a1: direct-mapped, a2: 2-way, a4: 4-way, af: fully-mapped). For plot clarity we only analyze AA-Poll; both AA-SP and AA-Ints follow similar performance patterns. For a given associativity, lru is always better than rnd as it entails fewer IOTLB misses. Increasing associativity and IOTLB size improves the performance, for example, using lru_af instead of lru_a4 allows for an up to 16% higher insert rate.

5.2.2 Access Counter

We now test the performance of a simple tool that counts accesses to an arbitrary data structure. We compare AA-Poll (counting done by the IOMMU), RMA (increasing counters with remote atomics), and two additional designs: an approach based on the “active key-value” store [18] (A-KV), and a scheme where counting is done at the source and the final sums are computed with the Allreduce collective operation [26] (Allreduce). Finally, we test a scenario with no counting (No-Cnt). The number of accesses per second is presented in Figure 7d. AA-Poll outperforms A-KV (overheads caused by the application-level design), RMA (issuing costly atomics), and All-Reduce (expensive synchronization).

5.2.3 Performant Logging of Gets

In the next step we evaluate the performance of active gets by testing the implementation of the mechanism that logs RMA gets. Here, processes issue remote gets targeted at random processes. Every get transfers one 8-byte integer value. In this benchmark we do not compare to AM-Exp, AM-Onload, and AM-Ints because these schemes were not suitable for implementing this type of application. Instead,

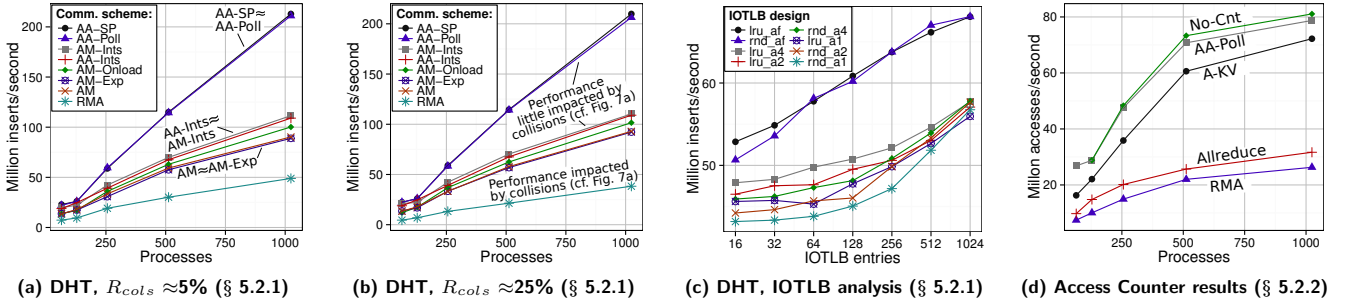


Figure 7: The performance of the DHT (Figures 7a, 7b, 7c) and the access counter (Figure 7d). We use 32 processes/node.

we compare to No-FT: a variant with no logging (no fault-tolerance overhead) that constitutes the best-case baseline. We illustrate the scalability of AA in Figure 8a. AA achieves the best performance, close to No-FT. In all the remaining protocols the data to be logged has to be transferred back to a remote storage using a put (RMA) or a send (AM), which incurs significant overheads. Varying the remaining parameters (R_{comp} , IOTLB parameters) follows the same performance pattern as in the hashtable evaluation.

5.2.4 Fault-Tolerant Performant Sort

To evaluate the performance of active gets we also implemented an RMA-based version of the parallel sort Coral Benchmark [13] that utilizes gets instead of messages, and made it fault-tolerant. We present the total time required to communicate the results of sorting 1GB of data between processes in Figure 8b. Again, AA is close to No-FT (< 1%) and reduces communication time by $\approx 50\%$ and $\approx 80\%$ in comparison to AM and RMA, respectively.

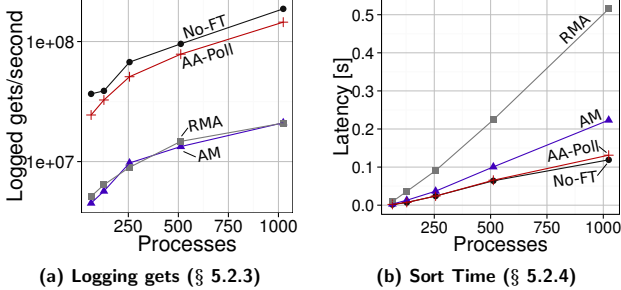


Figure 8: The performance of the AA-based fault tolerance scheme.

6. RELATED WORK AND DISCUSSION

Not all possible use cases for IOMMUs have been studied so far. Ben-Yehuda et al. [8] discuss IOMMUs for virtualization in Linux. Other works target efficient IOMMU emulation [4], reducing IOTLB miss rates [3], isolating Linux device drivers [12], and mitigating IOMMUs' overheads [7]. There are also vendors' white papers and specifications [1, 2, 5, 21, 22, 25, 33]. Our work goes beyond these studies by proposing a new mechanism and a programming model that combines AM with RMA and uses IOMMUs for high-performance distributed data-centric computations.

There are several mechanisms that extend the memory subsystem to improve the performance of various codes. Active Pages [30] enable the memory to perform some simple operations allowing the CPU to operate at peak computational bandwidth. Active Memory [16] and in-memory computing [42] add simple compute logic to the memory controller and the memory itself, respectively. AA differs from these schemes as it targets distributed RMA computations

and its implementation does not modify the memory subsystem requiring minor extensions to the commodity IOMMUs.

Scale-Out NUMA [27] is an architecture, programming model, and communication protocol that offers low latency of remote memory accesses. It differs from AA as it does not provide the active semantics for both puts and gets and it introduces significant changes to the memory subsystem.

Active messages were introduced by von Eicken et al [40]. Scalable programming for RMA was discussed by Gerstenberger et al [19]. Some of AA's functionalities could be achieved using existing RMA/AM interfaces such as Portals [6], InfiniBand [37], or GASNet [11]. However, Portals would introduce additional memory overheads per NIC because it requires descriptors for every memory region. These overheads may grow prohibitively for multiple NICs. Contrarily, AA uses a single centralized IOMMU with existing paging structures, ensuring no additional memory overheads. Furthermore, AA offers notifications on gets and it enables various novel schemes such as incremental checkpointing for RMA and performant logging of gets.

AA could also be implemented in the NIC. Still, using IOMMUs provides several advantages. Modern IOMMUs are integrated with the memory controller/CPU and thus can be directly connected with CPU stratchpads for a high-performance notification mechanism (see § 3.5). This way, all I/O devices could take advantage of this functionality (e.g., Ethernet RoCE NICs). Moreover, we envision other future mechanisms that would enable even further integration with the CPU. For example, the IOMMU could be directly connected to the CPU instruction pipeline to directly feed the CPU with handler code.

Finally, AA's potential can be further explored to provide hardware virtualization of remote memories. There are three major advantages of virtual memory: it enables an OS to swap memory blocks into disk, it facilitates the application development by providing processes with separate address spaces, and it enables useful features such as memory protection or dirty bits. Some schemes (e.g., PGAS languages) emulate a part of these functionalities for networked memories. Extending AA with features specific to MMU PTEs (e.g., invalid bits) would enable a hardware-based *virtual global address space* (V-GAS) with novel enhanced paging capabilities and data-centric handlers running transparently to any code accessing the memory; see Fig. 9.

The IOMMUs could also become the basis of V-GAS for Ethernet. All the described IOMMU extensions are generic and do not rely on any specific NIC features, leaving the possibility of moving the V-GAS potential into commodity machines that do not provide native RDMA support. For example, by utilizing Single Root I/O Virtualization (SR/IOV), a standard support for hardware virtualization

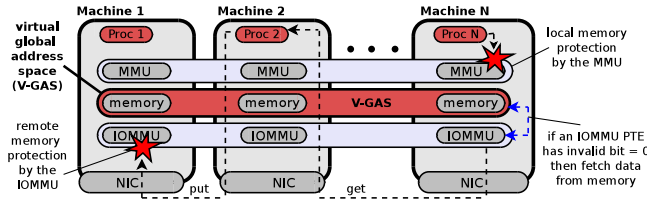


Figure 9: V-GAS together with some example features.

combined with multiple receive and transmit rings, one can utilize IOMMUs to safely divert traffic right into userspace.

7. CONCLUSION

RMA is becoming more popular for programming datacenters and HPC computers. However, its traditional one-sided model of communication may incur performance penalties in several types of applications such as DHT.

To alleviate this issue we propose the Active Access scheme that utilizes IOMMUs to provide hardware support for *active* semantics in RMA. For example, our AA-based DHT implementation offers a speedup of two over optimized AMs. The novel AA-based fault tolerance protocol enables performant logging of gets and adds negligible (1-5%) overheads to the application runtime. Furthermore, AA enables new schemes such as incremental checkpointing in RMA. Finally, our design bypasses the OS and enables more effective programming of datacenters and HPC centers.

AA enables a new programming model that combines the benefits of one-sided communication and active messages. AA is *data-centric* as it enables triggering handlers when certain data is accessed. Thus, it could be useful for future data processing and analysis schemes and protocols.

The proposed AA design, based on IOMMUs, shows the potential behind currently available off-the-shelf hardware for developing novel mechanisms. By moving the notification functionality from the NIC to the IOMMU we adopt the existing IOMMU paging structures and we eliminate the need for expensive memory descriptors present in, e.g., Portals, thus reducing memory overheads. The IOMMU-based design may enable even more performant notification mechanisms such as direct access from the IOMMU to the CPU pipeline. Thus, AA may play an important role in designing efficient codes and OS/runtime systems in large datacenters, HPC computers, and highly parallel manycore environments which are becoming commonplace even in commodity off-the-shelf computers.

Acknowledgements

We thank the CSCS team granting access to the Monte Rosa machine, and for their excellent technical support. We thank Greg Bronevetsky (LLNL) for inspiring comments and Ali Saidi for help with the gem5 simulator. MB is supported by the 2013 Google European Doctoral Fellowship in Parallel Computing.

8. REFERENCES

- [1] AMD. Software Optimization Guide for the AMD64 Processors, 2005.
- [2] AMD. AMD I/O Virtualization Technology (IOMMU) Spec., 2011.
- [3] N. Amit, M. Ben-Yehuda, and B.-A. Yassour. IOMMU: strategies for mitigating the IOTLB bottleneck. In *Proc. of Intl. Conf. on Comp. Arch.*, ISCA'10, pages 256–274, 2010.
- [4] N. Amit et al. vIOMMU: efficient IOMMU emulation. In *USENIX Ann. Tech. Conf.*, USENIXATC'11, pages 6–6, 2011.
- [5] W. J. Armstrong et al. Advanced virtualization capabilities of POWER5 systems. *IBM J. Res. Dev.*, 49(4/5):523–532, 2005.

- [6] B. W. Barrett et al. The Portals 4.0 network programming interface, 2012. Sandia National Laboratories.
- [7] Ben-Yehuda et al. The price of safety: Evaluating IOMMU performance. In *Ottawa Linux Symp.(OLS)*, pages 9–20, 2007.
- [8] M. Ben-Yehuda et al. Utilizing IOMMUs for virtualization in Linux and Xen. In *In Proc. of the Linux Symp.*, 2006.
- [9] M. Besta and T. Hoefer. Fault Tolerance for Remote Memory Access Programming Models. In *Proc. of the 23rd Intl Symp. on High-perf. Par. and Dist. Comp.*, HPDC '14, pages 37–48, 2014.
- [10] N. Binkert et al. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, Aug. 2011.
- [11] D. Bonachea. GASNet Spec., v1. *Tech. Rep. UCB/CSD-02-1207*, 2002.
- [12] S. Boyd-Wickizer and N. Zeldovich. Tolerating malicious device drivers in Linux. In *USENIX Ann. Tech. Conf.*, USENIXATC'10, pages 9–9, 2010.
- [13] Coral Collaboration. Coral Procurement Benchmarks. In *Coral Vendor Meeting*, 2013.
- [14] A. Dragojević et al. FaRM: fast remote memory. In *Proc. of the 11th USENIX Symp. on Net. Syst. Des. and Impl. (NSDI 14)*. USENIX, 2014.
- [15] H. Esmailzadeh et al. Dark silicon and the end of multicore scaling. In *Proc. of Intl. Symp. Comp. Arch.*, ISCA '11, pages 365–376, 2011.
- [16] Z. Fang et al. Active Memory Operations. In *Proc. of the 21st Ann. Intl Conf. on Supercomp.*, ICS '07, pages 232–241, 2007.
- [17] B. Fitzpatrick. Distributed caching with memcached. *Linux journal*, 2004(124):5, 2004.
- [18] R. Geambasu et al. Comet: An Active Distributed Key-value Store. In *USENIX Conf. on Op. Sys. Des. and Impl.*, OSDI'10, pages 1–13, 2010.
- [19] R. Gerstenberger, M. Besta, and T. Hoefer. Enabling Highly-scalable Remote Memory Access Programming with MPI-3 One Sided. In *Proc. of ACM/IEEE Supercomputing*, SC '13, pages 53:1–53:12, 2013.
- [20] T. Hoefer et al. Remote Memory Access Programming in MPI-3. *ACM Trans. Par. Comp. (TOPC)*, 2015. accepted for publication on Dec. 4th.
- [21] IBM. *Logical Partition Security in the IBM eServer pSeries 690*. 2002.
- [22] Intel. Intel Virtualization Technology for Directed I/O (VT-d) Architecture Specification, September 2013.
- [23] ISO Fortran Committee. Fortran 2008 Standard (ISO/IEC 1539-1:2010). 2010.
- [24] Y. Kim, D. Broman, J. Cai, and A. Shrivastava. WCET-aware dynamic code management on scratchpads for software-managed multicores. In *IEEE Real-Time and Emb. Tech. and App. Symp. (RTAS)*, 2014.
- [25] R. Mijat and A. Nightingale. The ARM Architecture Virtualization Extensions and the importance of System MMU for virtualized solutions and beyond, 2011. ARM White Paper.
- [26] MPI Forum. MPI: A Message-Passing Interface Standard. Ver. 3, 2012.
- [27] S. Novakovic et al. Scale-out NUMA. In *Intl. Conf. on Arch. Sup. for Prog. Lang. and Op. Sys.*, ASPLOS '14, pages 3–18, 2014.
- [28] R. Olsson. PktGen the linux packet generator. In *Proc. of the Linux Symp., Ottawa, Canada*, volume 2, pages 11–24, 2005.
- [29] Oracle. *UltraSPARC Virtual Machine Spec.* 2010.
- [30] M. Oskin, F. T. Chong, and T. Sherwood. Active Pages: A Computation Model for Intelligent Memory. In *Proc. of the 25th Ann. Intl Symp. on Comp. Arch.*, ISCA '98, pages 192–203, 1998.
- [31] D. Patterson. The top 10 innovations in the new NVIDIA Fermi architecture, and the top 3 next challenges. *NVIDIA Whitepaper*, 2009.
- [32] PCI-SIG. *PCI Express Base Spec. Rev. 3.0*. 2010.
- [33] PCI-SIG. PCI-SIG I/O Virtualization (IOV) Specifications, 2013.
- [34] S. Pope and D. Riddoch. Introduction to OpenOnload, 2011. SolarFlare White Paper.
- [35] R. Recio, B. Metzler, P. Culley, J. Hilland, and D. Garcia. A remote direct memory access protocol specification, Oct 2007. RFC 5040.
- [36] L. Rizzo. netmap: A novel framework for fast packet i/o. In *USENIX Annual Technical Conference*, pages 101–112, 2012.
- [37] The InfiniBand Trade Association. *Infiniband Architecture Spec. Vol. 1-2, Rel. 1.3*. InfiniBand Trade Association, 2004.
- [38] UPC Consortium. UPC language spec., v1.2. Technical report, Lawrence Berkeley National Laboratory, 2005. LBNL-59208.
- [39] M. Vasavada, F. Mueller, P. H. Hargrove, and E. Roman. Comparing different approaches for incremental checkpointing: The showdown. In *Linux'11: The 13th Annual Linux Symposium*, pages 69–79, 2011.
- [40] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauer. Active messages: a mechanism for integrated communication and computation. In *Proc. of Intl. Symp. Comp. Arch.*, ISCA '92, pages 256–266, 1992.
- [41] J. Willcock et al. AM++: A Generalized Active Message Framework. In *Intl. Conf. on Par. Arch. and Comp. Tech.*, pages 401–410, 2010.
- [42] Q. Zhu et al. Accelerating sparse matrix-matrix multiplication with 3D-stacked logic-in-memory hardware. In *High Perf. Ext. Comp. Conf. (HPEC)*, pages 1–6. IEEE, 2013.