Scale-Out NUMA

Stanko Novaković Alexandros Daglis Edouard Bugnion Babak Falsafi Boris Grot†
EcoCloud, EPFL
†University of Edinburgh

Abstract

Emerging datacenter applications operate on vast datasets that are kept in DRAM to minimize latency. The large number of servers needed to accommodate this massive memory footprint requires frequent server-to-server communication in applications such as key-value stores and graph-based applications that rely on large irregular data structures. The fine-grained nature of the accesses is a poor match to commodity networking technologies, including RDMA, which incur delays of 10-1000x over local DRAM operations.

We introduce Scale-Out NUMA (soNUMA) – an architecture, programming model, and communication protocol for low-latency, distributed in-memory processing. soNUMA layers an RDMA-inspired programming model directly on top of a NUMA memory fabric via a stateless messaging protocol. To facilitate interactions between the application, OS, and the fabric, soNUMA relies on the *remote memory controller* – a new architecturally-exposed hardware block integrated into the node's local coherence hierarchy. Our results based on cycle-accurate full-system simulation show that soNUMA performs remote reads at latencies that are within 4x of local DRAM, can fully utilize the available memory bandwidth, and can issue up to 10M remote memory operations per second per core.

Categories and Subject Descriptors C.1.4 [*Computer System Organization*]: Parallel Architectures—Distributed Architectures; C.5.5 [*Computer System Organization*]: Computer System Implementation—Servers

Keywords RDMA, NUMA, System-on-Chips

1. Introduction

Datacenter applications are rapidly evolving from simple data-serving tasks to sophisticated analytics operating over enormous datasets in response to real-time queries. To minimize the response latency, datacenter operators keep the data in memory. As dataset sizes push into the petabyte range, the number of servers required to house them in memory can easily reach into hundreds or even thousands.

Because of the distributed memory, applications that traverse large data structures (e.g., graph algorithms) or frequently access disparate pieces of data (e.g., key-value stores) must do so over the datacenter network. As today's datacenters are built with commodity networking technology running on top of commodity servers and operating systems, node-to-node communication delays can exceed 100μs [50]. In contrast, accesses to local memory incur delays of around 60ns – a factor of 1000 less. The irony is rich: moving the data from disk to main memory yields a 100,000x reduction in latency (10ms vs. 100ns), but distributing the memory eliminates 1000x of the benefit.

The reasons for the high communication latency are well known and include deep network stacks, complex network interface cards (NIC), and slow chip-to-NIC interfaces [21, 50]. RDMA reduces end-to-end latency by enabling memory-to-memory data transfers over InfiniBand [26] and Converged Ethernet [25] fabrics. By exposing remote memory at user-level and offloading network processing to the adapter, RDMA enables remote memory read latencies as low as 1.19μs [14]; however, that still represents a $>$10x latency increase over local DRAM.

We introduce Scale-Out NUMA (soNUMA), an architecture, programming model, and communication protocol for distributed, in-memory applications that reduces remote memory access latency to within a small factor ($\sim$4x) of local memory. soNUMA leverages two simple ideas to minimize latency. The first is to use a stateless request/reply protocol running over a NUMA memory fabric to drastically reduce or eliminate the network stack, complex NIC, and switch gear delays. The second is to integrate the protocol controller into the node's local coherence hierarchy, thus avoiding state replication and data movement across the slow PCI Express (PCIe) interface.

soNUMA exposes the abstraction of a partitioned global virtual address space, which is useful for big-data applications with irregular data structures such as graphs. The programming model is inspired by RDMA [37], with applica-

Permission to make digital or hard copies of part or all 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 third-party components of this work must be honored. For all other uses, contact the owner/author(s).
ASPLOS '14, March 1–4, 2014, Salt Lake City, Utah, USA.
Copyright is held by the owner/author(s).
ACM 978-1-4503-2305-5/14/03.
http://dx.doi.org/10.1145/2541940.2541965

tion threads making explicit remote memory read and write requests with copy semantics. The model is supported by an architecturally-exposed hardware block, called the *remote memory controller* (RMC), that safely exposes the global address space to applications. The RMC is integrated into each node's coherence hierarchy, providing for a frictionless, low-latency interface between the processor, memory, and the interconnect fabric.

Our primary contributions are:

- the RMC – a simple, hardwired, on-chip architectural block that services remote memory requests through locally cache-coherent interactions and interfaces directly with an on-die network interface. Each operation handled by the RMC is converted into a set of stateless request/reply exchanges between two nodes;
- a minimal programming model with architectural support, provided by the RMC, for one-sided memory operations that access a partitioned global address space. The model is exposed through lightweight libraries, which also implement communication and synchronization primitives in software;
- a preliminary evaluation of soNUMA using cycle-accurate full-system simulation demonstrating that the approach can achieve latencies within a small factor of local DRAM and saturate the available bandwidth;
- an soNUMA emulation platform built using a hypervisor that runs applications at normal wall-clock speeds and features remote latencies within 5x of what a hardware-assisted RMC should provide.

The rest of the paper is organized as follows: we motivate soNUMA (§2). We then describe the essential elements of the soNUMA architecture (§3), followed by a description of the design and implementation of the RMC (§4), the software support (§5), and the proposed communication protocol (§6). We evaluate our design (§7) and discuss additional aspects of the work (§8). Finally, we place soNUMA in the context of prior work (§9) and conclude (§10).

2. Why Scale-Out NUMA?

In this section, we discuss key trends in datacenter applications and servers, and identify specific pain points that affect the latency of such deployments.

2.1 Datacenter Trends

Applications. Today's massive web-scale applications, such as search or analytics, require thousands of computers and petabytes of storage [60]. Increasingly, the trend has been toward deeper analysis and understanding of data in response to real-time queries. To minimize the latency, datacenter operators have shifted hot datasets from disk to DRAM, necessitating terabytes, if not petabytes, of DRAM distributed across a large number of servers.

The distributed nature of the data leads to frequent server-to-server interactions within the context of a given computation, e.g., Amazon reported that the rendering of a single page typically requires access to over 150 services [17]. These interactions introduce significant latency overheads that constrain the practical extent of sharding and the complexity of deployed algorithms. For instance, latency considerations force Facebook to restrict the number of sequential data accesses to fewer than 150 per rendered web page [50].

Recent work examining sources of network latency overhead in datacenters found that a typical deployment based on commodity technologies may incur over $100\mu s$ in round-trip latency between a pair of servers [50]. According to the study, principal sources of latency overhead include the operating system stack, NIC, and intermediate network switches. While $100\mu s$ may seem insignificant, we observe that many applications, including graph-based applications and those that rely on key-value stores, perform minimal computation per data item loaded. For example, read operations dominate key-value store traffic, and simply return the object in memory. With 1000x difference in data access latency between local DRAM (100ns) and remote memory ($100\mu s$), distributing the dataset, although necessary, incurs a dramatic performance overhead.

Server architectures. Today's datacenters employ commodity technologies due to their favorable cost-performance characteristics. The end result is a *scale-out* architecture characterized by a large number of commodity servers connected via commodity networking equipment. Two architectural trends are emerging in scale-out designs.

First, System-on-Chips (SoC) provide high chip-level integration and are a major trend in servers. Current server SoCs combine many processing cores, memory interfaces, and I/O to reduce cost and improve overall efficiency by eliminating extra system components, e.g., Calxeda's ECX-1000 SoC [9] combines four ARM Cortex-A9 cores, memory controller, SATA interface, and a fabric switch [8] into a compact die with a 5W typical power draw.

Second, system integrators are starting to offer *glueless fabrics* that can seamlessly interconnect hundreds of server nodes into fat-tree or torus topologies [18]. For instance, Calxeda's on-chip fabric router encapsulates Ethernet frames while energy-efficient processors run the standard TCP/IP and UDP/IP protocols as if they had a standard Ethernet NIC [16]. The tight integration of NIC, routers and fabric leads to a reduction in the number of components in the system (thus lowering cost) and improves energy efficiency by minimizing the number of chip crossings. However, such glueless fabrics alone do not substantially reduce latency because of the high cost of protocol processing at the end points.

Remote DMA. RDMA enables memory-to-memory data transfers across the network without processor involvement on the destination side. By exposing remote memory and re-

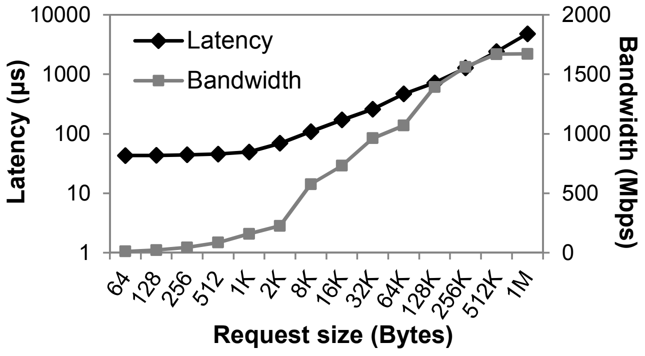


Figure 1: Netpipe benchmark on a Calxeda microserver.

liable connections directly to user-level applications, RDMA eliminates all kernel overheads. Furthermore, one-sided remote memory operations are handled entirely by the adapter without interrupting the destination core. RDMA is supported on lossless fabrics such as InfiniBand [26] and Converged Ethernet [25] that scale to thousands of nodes and can offer remote memory read latency as low as 1.19μs [14].

Although historically associated with the high-performance computing market, RDMA is now making inroads into web-scale data centers, such as Microsoft Bing [54]. Latency-sensitive key-value stores such as RAMCloud [43] and Pilaf [38] are using RDMA fabrics to achieve object access latencies of as low as 5μs.

2.2 Obstacles to Low-Latency Distributed Memory

As datasets grow, the trend is toward more sophisticated algorithms at ever-tightening latency bounds. While SoCs, glueless fabrics, and RDMA technologies help lower network latencies, the network delay per byte loaded remains high. Here, we discuss principal reasons behind the difficulty of further reducing the latency for in-memory applications.

Node scalability is power-limited. As voltage scaling grinds to a halt, future improvements in compute density at the chip level will be limited. Power limitations will extend beyond the processor and impact the amount of DRAM that can be integrated in a given unit of volume (which governs the limits of power delivery and heat dissipation). Together, power constraints at the processor and DRAM levels will limit the server industry's ability to improve the performance and memory capacity of scale-up configurations, thus accelerating the trend toward distributed memory systems.

Deep network stacks are costly. Distributed systems rely on networks to communicate. Unfortunately, today's deep network stacks require a significant amount of processing per network packet which factors considerably into end-to-end latency. Figure 1 shows the network performance between two directly-connected Calxeda EnergyCore ECX-1000 SoCs, measured using the standard `netpipe` benchmark [55]. The fabric and the integrated NICs provide 10Gbps worth of bandwidth.

Despite the immediate proximity of the nodes and the lack of intermediate switches, we observe high latency (in excess of 40μs) for small packet sizes and poor bandwidth scalability (under 2 Gbps) with large packets. These bottlenecks exist due to the high processing requirements of TCP/IP and are aggravated by the limited performance offered by ARM cores.

Large-scale shared memory is prohibitive. One way to bypass complex network stacks is through direct access to shared physical memory. Unfortunately, large-scale sharing of physical memory is challenging for two reasons. First is the sheer cost and complexity of scaling up hardware coherence protocols. Chief bottlenecks here include state overhead, high bandwidth requirements, and verification complexity. The second is the fault-containment challenge of a single operating system instance managing a massive physical address space, whereby the failure of any one node can take down the entire system by corrupting shared state [11]. Sharing caches even within the same socket can be expensive. Indeed, recent work shows that partitioning a single many-core socket into multiple coherence domains improves the execution efficiency of scale-out workloads that do not have shared datasets [33].

PCIe/DMA latencies limit performance. I/O bypass architectures have successfully removed most sources of latency except the PCIe bus. Studies have shown that it takes 400-500ns to communicate short bursts over the PCIe bus [21], making such transfers 7-8x more expensive, in terms of latency, than local DRAM accesses. Furthermore, PCIe does not allow for the cache-coherent sharing of control structures between the system and the I/O device, leading to the need of replicating system state such as page tables into the device and system memory. In the latter case, the device memory serves as a cache, resulting in additional DMA transactions to access the state. SoC integration alone does not eliminate these overheads, since IP blocks often use DMA internally to communicate with the main processor [5].

Distance matters. Both latency and cost of high-speed communication within a datacenter are severely impacted by distance. Latency is insignificant and bandwidth is cheap within a rack, enabling low-dimensional topologies (e.g., 3-D torus) with wide links and small signal propagation delays (e.g., 20ns for a printed circuit board trace spanning a 44U rack). Beyond a few meters, however, expensive optical transceivers must be used, and non-negotiable propagation delays (limited by the speed of light) quickly exceed DRAM access time. The combination of cost and delay puts a natural limit to the size of tightly interconnected systems.

3. Scale-Out NUMA

This work introduces soNUMA, an architecture and programming model for low-latency distributed memory.

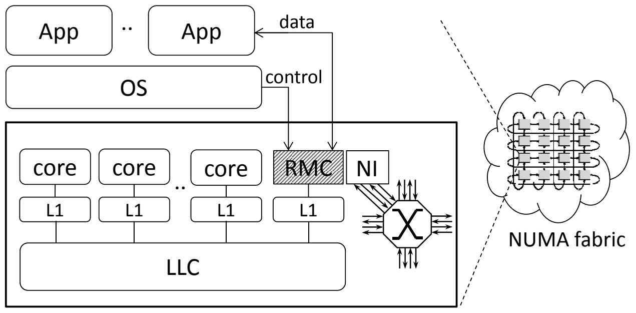


Figure 2: soNUMA overview.

soNUMA addresses each of the obstacles to low-latency described in §2.2. soNUMA is designed for a scale-out model with physically distributed processing and memory: (i) it replaces deep network stacks with a lean memory fabric; (ii) eschews system-wide coherence in favor of a global partitioned virtual address space accessible via RMDA-like remote memory operations with copy semantics; (iii) replaces transfers over the slow PCIe bus with cheap cache-to-cache transfers; and (iv) is optimized for rack-scale deployments, where distance is minuscule. In effect, our design goal is to borrow the desirable qualities of ccNUMA and RDMA without their respective drawbacks.

Fig. 2 identifies the essential components of soNUMA. At a high level, soNUMA combines a lean memory fabric with an RDMA-like programming model in a rack-scale system. Applications access remote portions of the global virtual address space through remote memory operations. A new architecturally-exposed block, the *remote memory controller* (RMC), converts these operations into network transactions and directly performs the memory accesses. Applications directly communicate with the RMC, bypassing the operating system, which gets involved only in setting up the necessary in-memory control data structures.

Unlike traditional implementations of RDMA, which operate over the PCI bus, the RMC benefits from a tight integration into the processor's cache coherence hierarchy. In particular, the processor and the RMC share all data structures via the cache hierarchy. The implementation of the RMC is further simplified by limiting the architectural support to one-sided remote memory read, write, and atomic operations, and by unrolling multi-line requests at the source RMC. As a result, the protocol can be implemented in a stateless manner by the destination node.

The RMC converts application commands into remote requests that are sent to the *network interface* (NI). The NI is connected to an on-chip low-radix router with reliable, point-to-point links to other soNUMA nodes. The notion of fast low-radix routers borrows from supercomputer interconnects; for instance, the mesh fabric of the Alpha 21364 connected 128 nodes in a 2D torus using an on-chip router with a pin-to-pin delay of just 11ns [39].

soNUMA's memory fabric bears semblance (at the link and network layer, but not at the protocol layer) to the QPI and HTX solutions that interconnect sockets together into multiple NUMA domains. In such fabrics, parallel transfers over traces minimize pin-to-pin delays, short messages (header + a payload of a single cache line) minimize buffering requirements, topology-based routing eliminates costly CAM or TCAM lookups, and virtual lanes ensure deadlock freedom. Although Fig. 2 illustrates a 2D-torus, the design is not restricted to any particular topology.

4. Remote Memory Controller

The foundational component of soNUMA is the RMC, an architectural block that services remote memory accesses originating at the local node, as well as incoming requests from remote nodes. The RMC integrates into the processor's coherence hierarchy via a private L1 cache and communicates with the application threads via memory-mapped queues. We first describe the software interface (§4.1), provide a functional overview of the RMC (§4.2), and describe its microarchitecture (§4.3).

4.1 Hardware/Software Interface

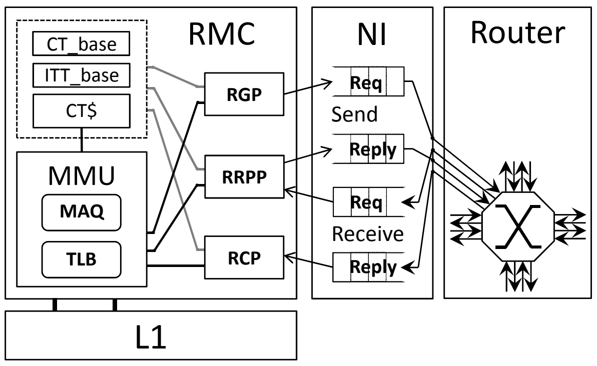
soNUMA provides application nodes with the abstraction of globally addressable, virtual address spaces that can be accessed via explicit memory operations. The RMC exposes this abstraction to applications, allowing them to safely and directly copy data to/from global memory into a local buffer using remote write, read, and atomic operations, without kernel intervention. The interface offers atomicity guarantees at the cache-line granularity, and no ordering guarantees within or across requests.

soNUMA's hardware/software interface is centered around four main abstractions directly exposed by the RMC: (i) the context identifier (`ctx_id`), which is used by all nodes participating in the same application to create a global address space; (ii) the context segment, a range of the node's address space which is globally accessible by others; (iii) the queue pair (QP), used by applications to schedule remote memory operations and get notified of their completion; and (iv) local buffers, which can be used as the source or destination of remote operations.

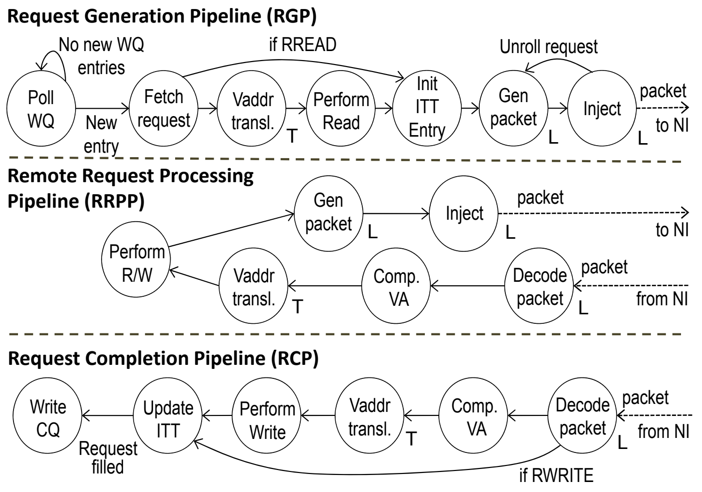
The QP model consists of a work queue (WQ), a bounded buffer written exclusively by the application, and a completion queue (CQ), a bounded buffer of the same size written exclusively by the RMC. The CQ entry contains the index of the completed WQ request. Both are stored in main memory and coherently cached by the cores and the RMC alike. In each operation, the remote address is specified by the combination of <`node_id, ctx_id, offset`>. Other parameters include the length and the local buffer address.

4.2 RMC Overview

The RMC consists of three hardwired pipelines that interact with the queues exposed by the hardware/software in-



(a) RMC's internals: all memory requests of the three pipelines access the cache via the MMU. The CT_base register, the ITT_base register, and the CT$ offer fast access to the basic control structures.



(b) Functionality of the RMC pipelines. States with an '*L*' next to them indicate local processing in combinational logic; '*T*' indicates a TLB access; the rest of the states access memory via the MMU.

Figure 3: RMC internal architecture and functional overview of the three pipelines.

terface and with the NI. These pipelines are responsible for request generation, remote request processing, and request completion, respectively. They are controlled by a configuration data structure, the *Context Table* (CT), and leverage an internal structure, the *Inflight Transaction Table (ITT)*.

The CT is maintained in memory and is initialized by system software (see §5.1). The CT keeps track of all registered context segments, queue pairs, and page table root addresses. Each CT entry, indexed by its `ctx_id`, specifies the address space and a list of registered QPs (WQ, CQ) for that context. Multi-threaded processes can register multiple QPs for the same address space and `ctx_id`. Meanwhile, the ITT is used exclusively by the RMC and keeps track of the progress of each WQ request.

Fig. 3a shows the high-level internal organization of the RMC and its NI. The three pipelines are connected to distinct queues of the NI block, which is itself connected to a low-radix router block with support for two virtual lanes. While each of the three pipelines implements its own datapath and control logic, all three share some common data structures and hardware components. For example, they arbitrate for access to the common L1 cache via the MMU.

Fig. 3b highlights the main states and transitions for the three independent pipelines. Each pipeline can have multiple transactions in flight. Most transitions require an MMU access, which may be retired in any order. Therefore, transactions will be reordered as they flow through a pipeline.

Request Generation Pipeline (RGP). The RMC initiates remote memory access transactions in response to an application's remote memory requests (reads, writes, atomics). To detect such requests, the RMC polls on each registered WQ. Upon a new WQ request, the RMC generates one or more network packets using the information in the WQ entry. For remote writes and atomic operations, the RMC accesses the local node's memory to read the required data, which it then encapsulates into the generated packet(s). For each request, the RMC generates a transfer identifier (`tid`) that allows the source RMC to associate replies with requests.

Remote transactions in soNUMA operate at cache line granularity. Coarser granularities, in cache-line-sized multiples, can be specified by the application via the `length` field in the WQ request. The RMC unrolls multi-line requests in hardware, generating a sequence of line-sized read or write transactions. To perform unrolling, the RMC uses the ITT, which tracks the number of completed cache-line transactions for each WQ request and is indexed by the request's `tid`.

Remote Request Processing Pipeline (RRPP). This pipeline handles incoming requests originating from remote RMCs. The soNUMA protocol is stateless, which means that the RRPP can process remote requests using only the values in the header and the local configuration state. Specifically, the RRPP uses the `ctx_id` to access the CT, computes the virtual address, translates it to the corresponding physical address, and then performs a read, write, or atomic operation as specified in the request. The RRPP always completes by generating a reply message, which is sent to the source. Virtual addresses that fall outside of the range of the specified security context are signaled through an error message, which is propagated to the offending thread in a special reply packet and delivered to the application via the CQ.

Request Completion Pipeline (RCP). This pipeline handles incoming message replies. The RMC extracts the `tid` and uses it to identify the originating WQ entry. For reads and atomics, the RMC then stores the payload into the application's memory at the virtual address specified in the request's WQ entry. For multi-line requests, the RMC computes the target virtual address based on the buffer base address specified in the WQ entry and the offset specified in the reply message.

The ITT keeps track of the number of completed cache-line requests. Once the last reply is processed, the RMC signals the request's completion by writing the index of the completed WQ entry into the corresponding CQ and moving the CQ head pointer. Requests can therefore complete out of order and, when they do, are processed out of order by the application. Remote write acknowledgments are processed similarly to read completions, although remote writes naturally do not require an update of the application's memory at the source node.

4.3 Microarchitectural Support

The RMC implements the logic described above using a set of completely decoupled pipelines, affording concurrency in the handling of different functions at low area and design cost. The RMC features two separate interfaces: a coherent memory interface to a private L1 cache and a network interface to the on-die router providing system-level connectivity. The memory interface block (MMU) contains a TLB for fast access to recent address translations, required for all accesses to application data. TLB entries are tagged with address space identifiers corresponding to the application context. TLB misses are serviced by a hardware page walker. The RMC provides two interfaces to the L1 cache – a conventional word-wide interface as well as a cache-line-wide interface. The former is used to interact with the application and to perform atomic memory operations. The latter enables efficient atomic reads and writes of entire cache lines, which is the granularity of remote memory accesses in soNUMA.

The RMC's integration into the node's coherence hierarchy is a critical feature of soNUMA that eliminates wasteful data copying of control structures, and of page tables in particular. It also reduces the latency of the application/RMC interface by eliminating the need to set up DMA transfers of ring buffer fragments. To further ensure high throughput and low latency at high load, the RMC allows multiple concurrent memory accesses in flight via a *Memory Access Queue* (MAQ). The MAQ handles all memory read and write operations, including accesses to application data, WQ and CQ interactions, page table walks, as well as ITT and CT accesses. The number of outstanding operations is limited by the number of miss status handling registers at the RMC's L1 cache. The MAQ supports out-of-order completion of memory accesses and provides store-to-load forwarding.

Each pipeline has its own arbiter that serializes the memory access requests from the pipeline's several stages and forwards the requests to the MAQ. The latter keeps track of each request's originating arbiter, and responds to that once the memory access is completed. Upon such a response, the arbiter feeds the data to the corresponding pipeline stage.

Finally, the RMC dedicates two registers for the CT and ITT base addresses, as well as a small lookaside structure, the *CT cache (CT$)* that caches recently accessed CT entries to reduce pressure on the MAQ. The CT$ includes the context segment base addresses and bounds, PT roots, and the queue addresses, including the queues' head and tail indices. The base address registers and the CT$ are read-only-shared by the various RMC pipeline stages.

5. Software Support

We now describe the system and application software support required to expose the RMC to applications and enable the soNUMA programming model. §5.1 describes the operating system device driver. §5.2 describes the lower-level wrappers that efficiently expose the hardware/software interface to applications. Finally, §5.3 describes higher-level routines that implement unsolicited communication and synchronization without additional architectural support.

5.1 Device Driver

The role of the operating system on an soNUMA node is to establish the global virtual address spaces. This includes the management of the context namespace, virtual memory, QP registration, etc. The RMC device driver manages the RMC itself, responds to application requests, and interacts with the virtual memory subsystem to allocate and pin pages in physical memory. The RMC device driver is also responsible for allocating the CT and ITT on behalf of the RMC.

Unlike a traditional RDMA NIC, the RMC has direct access to the page tables managed by the operating system, leveraging the ability to share cache-coherent data structures. As a result, the RMC and the application both operate using virtual addresses of the application's process once the data structures have been initialized.

The RMC device driver implements a simple security model in which access control is granted on a per `ctx_id` basis. To join a global address space `<ctx_id>`, a process first opens the device `/dev/rmc_contexts/<ctx_id>`, which requires the user to have appropriate permissions. All subsequent interactions with the operating system are done by issuing `ioctl` calls via the previously-opened file descriptor. In effect, soNUMA relies on the built-in operating system mechanism for access control when opening the context, and further assumes that all operating system instances of an soNUMA fabric are under a single administrative domain.

Finally, the RMC notifies the driver of failures within the soNUMA fabric, including the loss of links and nodes. Such transitions typically require a reset of the RMC's state, and may require a restart of the applications.

5.2 Access Library

The QPs are accessed via a lightweight API, a set of C/C++ inline functions that issue remote memory commands and synchronize by polling the completion queue.

We expose a synchronous (blocking) and an asynchronous (non-blocking) set of functions for both reads and writes. The asynchronous API is comparable in terms of functionality to the Split-C programming model [15].

Fig. 4 illustrates the use of the asynchronous API for the implementation of the classic PageRank graph algorithm [45]. `rmc_wait_for_slot` processes CQ events (calling `pagerank_async` for all completed slots) until the head of the WQ is free. It then returns the freed slot where the next entry will be scheduled. `rmc_read_async` (similar to Split-C's `get`) requests a copy of a remote vertex into a local buffer. Finally, `rmc_drain_cq` waits until all outstanding remote operations have completed while performing the remaining callbacks.

This programming model is efficient as: (i) the callback (`pagerank_async`) does not require a dedicated execution context, but instead is called directly within the main thread; (ii) when the callback is an inline function, it is passed as an argument to another inline function (`rmc_wait_for_slot`), thereby enabling compilers to generate optimized code without any function calls in the inner loop; (iii) when the algorithm has no read dependencies (as is the case here), asynchronous remote memory accesses can be fully pipelined to hide their latency.

To summarize, soNUMA's programming model combines true shared memory (by the threads running within a cache-coherent node) with explicit remote memory operations (when accessing data across nodes). In the PageRank example, the `is_local` flag determines the appropriate course of action to separate intra-node accesses (where the memory hierarchy ensures cache coherence) from inter-node accesses (which are explicit).

Finally, the RMC access library exposes atomic operations such as compare-and-swap and fetch-and-add as inline functions. These operations are executed atomically within the local cache coherence hierarchy of the destination node.

5.3 Messaging and Synchronization Library

By providing architectural support for only read, write and atomic operations, soNUMA reduces hardware cost and complexity. The minimal set of architecturally-supported operations is not a limitation, however, as many standard communication and synchronization primitives can be built in software on top of these three basic primitives. In contrast, RDMA provides hardware support (in adapters) for unsolicited send and receive messages on top of reliable connections, thus introducing significant complexity (e.g., per-connection state) into the design [47].

Unsolicited communication. To communicate using `send` and `receive` operations, two application instances must first each allocate a bounded buffer from their own portion of the global virtual address space. The sender always writes to the peer's buffer using `rmc_write` operations, and the content is read locally from cached memory by the receiver. Each

```
float *async_dest_addr[MAX_WQ_SIZE];
Vertex lbuf[MAX_WQ_SIZE];

inline void pagerank_async(int slot, void *arg) {
  *async_dest_addr[slot] += 0.85 *
    lbuf[slot].rank[superstep%2] / lbuf[slot].out_degree;
}

void pagerank_superstep(QP *qp) {
  int evenodd = (superstep+1) % 2;
  for(int v=first_vertex; v<=last_vertex; v++) {
    vertices[v].rank[evenodd] = 0.15 / total_num_vertices;
    for(int e=vertices[v].start; e<vertices[v].end; e++) {
      if(edges[e].is_local) {
        // shared memory model
        Vertex *v2 = (Vertex *)edges[e].v;
        vertices[v].rank[evenodd] += 0.85 *
              v2->rank[superstep%2] / v2->out_degree;
      } else {
        // flow control
        int slot = rmc_wait_for_slot(qp, pagerank_async);
        // setup callback arguments
        async_dest_addr[slot] = &vertices[v].rank[evenodd];
        // issue split operation
        rmc_read_async(qp, slot,
              edges[e].nid,      //remote node ID
              edges[e].offset,   //offset
              &lbuf[slot],       //local buffer
              sizeof(Vertex));   //len
      }
    }
  }
  rmc_drain_cq(qp, pagerank_async);
  superstep++;
}
```

Figure 4: Computing a PageRank superstep in soNUMA through a combination of remote memory accesses (via the asynchronous API) and local shared memory.

buffer is an array of cache-line sized structures that contain header information (such as the length, memory location, and flow-control acknowledgements), as well as an optional payload. Flow-control is implemented via a credit scheme that piggybacks existing communication.

For small messages, the sender creates packets of predefined size, each carrying a portion of the message content as part of the payload. It then *pushes* the packets into the peer's buffer. To receive a message, the receiver polls on the local buffer. In the common case, the send operation requires a single `rmc_write`, and it returns without requiring any implicit synchronization between the peers. A similar messaging approach based on remote writes outperforms the default send/receive primitives of InfiniBand [32].

For large messages stored within a registered global address space, the sender only provides the base address and size to the receiver's bounded buffer. The receiver then *pulls* the content using a single `rmc_read` and acknowledges the completion by writing a zero-length message into the sender's bounded buffer. This approach delivers a direct memory-to-memory communication solution, but requires synchronization between the peers.

At compile time, the user can define the boundary between the two mechanisms by setting a minimal message-size threshold: push has lower latency since small messages complete through a single `rmc_write` operation and also allows for decoupled operations. The pull mechanism leads to higher bandwidth since it eliminates the intermediate packetization and copy step.

Barrier synchronization. We have also implemented a simple barrier primitive such that nodes sharing a `ctx_id` can synchronize. Each participating node broadcasts the arrival at a barrier by issuing a `write` to an agreed upon offset on each of its peers. The nodes then poll locally until all of them reach the barrier.

6. Communication Protocol

soNUMA's communication protocol naturally follows the design choices of the three RMC pipelines at the protocol layer. At the link and routing layers, our design borrows from existing memory fabric architectures (e.g., QPI or HTX) to minimize pin-to-pin delays.

Link layer. The memory fabric delivers messages reliably over high-speed point-to-point links with credit-based flow control. The message MTU is large enough to support a fixed-size header and an optional cache-line-sized payload. Each point-to-point physical link has two virtual lanes to support deadlock-free request/reply protocols.

Routing layer. The routing-layer header contains the destination and source address of the nodes in the fabric (<`dst_nid, src_nid`>). `dst_nid` is used for routing, and `src_nid` to generate the reply packet.

The router's forwarding logic directly maps destination addresses to outgoing router ports, eliminating expensive CAM or TCAM lookups found in networking fabrics. While the actual choice of topology depends on system specifics, low-dimensional k-ary n-cubes (e.g., 3D torii) seem well-matched to rack-scale deployments [18].

Protocol layer. The RMC protocol is a simple request-reply protocol, with exactly one reply message generated for each request. The WQ entry specifies the `dst_nid`, the command (e.g., `read`, `write`, or `atomic`), the `offset`, the length and the local buffer address. The RMC copies the `dst_nid` into the routing header, determines the `ctx_id` associated with the WQ, and generates the `tid`. The `tid` serves as an index into the ITT and allows the source RMC to map each reply message to a WQ and the corresponding WQ entry. The `tid` is opaque to the destination node, but is transferred from the request to the associated reply packet.

Fig. 5 illustrates the actions taken by the RMCs for a remote read of a single cache line. The RGP in the requesting side's RMC first assigns a `tid` for the WQ entry and the

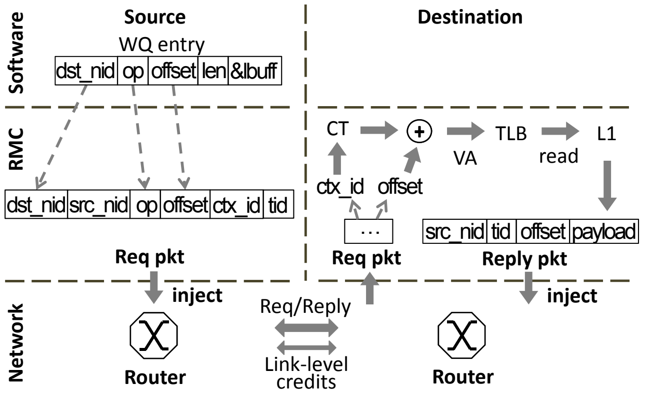


Figure 5: Communication protocol for a remote read.

`ctx_id` corresponding to that WQ. The RMC specifies the destination node via a `dst_nid` field. The request packet is then injected into the fabric and the packet is delivered to the target node's RMC. The receiving RMC's RRPP decodes the packet, computes the local virtual address using the `ctx_id` and the `offset` found in it and translates that virtual address to a physical address. This stateless handling does not require any software interaction on the destination node. As soon as the request is completed in the remote node's memory hierarchy, its RMC creates a reply packet and sends it back to the requesting node. Once the reply arrives to the original requester, the RMC's RCP completes the transaction by writing the payload into the corresponding local buffer and by notifying the application via a CQ entry (not shown in Fig. 5).

7. Evaluation

7.1 Methodology

To evaluate soNUMA, we designed and implemented two platforms: (i) development platform – a software prototype of soNUMA based on virtual machines used to debug the protocol stack, formalize the API, and develop large-scale applications; and (ii) cycle-accurate model – a full-system simulation platform modeling the proposed RMC.

Development platform. Our software soNUMA prototype is based on the Xen hypervisor [3] and a conventional cc-NUMA server, on top of which we map (pin) multiple virtual machines to distinct NUMA domains. This includes both virtual CPUs and memory page frames. The server we use for the prototype is a modern AMD Opteron server with 4 CPU sockets (12 cores each, three-level cache hierarchy, 16MB LLC) and 256GB of RAM. The memory subsystem provides us with 8 NUMA domains (2 per socket).

Fig. 6 shows our setup. Each individual VM represents an independent soNUMA node, running an instance of the full software stack. The stack includes all user-space libraries, applications, the OS kernel, as well as the complete RMC device driver inside it. The driver is a Linux kernel mod-

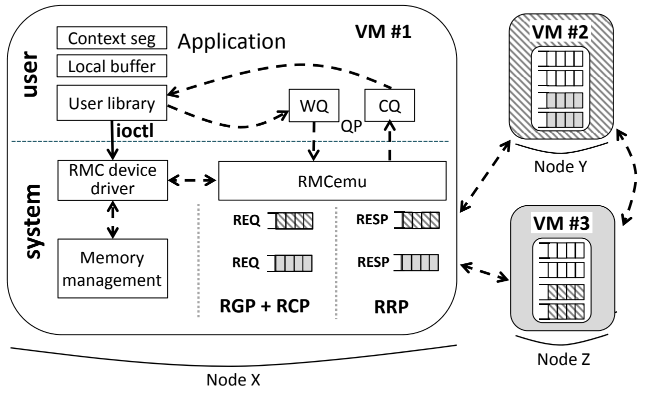


Figure 6: soNUMA development platform. Each node is implemented by a different VM. `RMCemu` runs on dedicated virtual CPUs and communicates with peers via shared memory.

ule that responds to user library commands through `ioctl`, enabling WQ/CQ registration, buffer management, and security context registration.

In this platform, we have implemented an RMC emulation module (RMCemu), which runs in kernel space. RMCemu implements the RMC logic and the soNUMA wire protocol (for a total of 3100LOC). The module exposes the hardware/software interface described in §4.1 to the RMC device driver and applications. RMCemu runs as a pair of kernel threads pinned to dedicated virtual CPUs, one running RGP and RCP, the other RRPP of Fig. 3b. All of the user-level data structures and buffers get memory-mapped by the device driver into the kernel virtual address space at registration time, and thus become visible to the RMCemu threads.

We emulate a full crossbar and run the protocol described in §6. Each pair of nodes exchanges protocol request/reply messages via a set of queues, mapped via the hypervisor into the guest physical address spaces of the VMs (there are two queue pairs per VM pair, emulating virtual lanes). To model the distributed nature of an soNUMA system, we pin each emulated node to a distinct NUMA domain such that every message traverses one of the server's chip-to-chip links. However, for the 16-node configuration, we collocate two VMs per NUMA domain.

Core	ARM Cortex-A15-like; 64-bit, 2GHz, OoO, 3-wide dispatch/retirement, 60-entry ROB
L1 Caches	split I/D, 32KB 2-way, 64-byte blocks, 2 ports, 32 MSHRs, 3-cycle latency (tag+data)
L2 Cache	4MB, 2 banks, 16-way, 6-cycle latency
Memory	cycle-accurate model using DRAMSim2 [49]. 4GB, 8KB pages, single DDR3-1600 channel. DRAM latency: 60ns; bandwidth: 12GBps
RMC	3 independent pipelines (RGP, RCP, RRPP). 32-entry MAQ, 32-entry TLB
Fabric	Inter-node delay: 50ns

Table 1: System parameters for simulation on Flexus.

Cycle-accurate model. To assess the performance implications of the RMC, we use the Flexus full-system simulator [59]. Flexus extends the Virtutech Simics functional simulator with timing models of cores, caches, on-chip protocol controllers, and interconnect. Flexus models the SPARC v9 ISA and is able to run unmodified operating systems and applications. In its detailed OoO timing mode with the RMCs implemented, Flexus simulates "only" 5000 instructions per second, a slowdown of about six orders of magnitude compared to real hardware.

We model simple nodes, each featuring a 64-bit ARM Cortex-A15-like core and an RMC. The system parameters are summarized in Table 1. We extend Flexus by adding a detailed timing model of the RMC based on the microarchitectural description in §4. The RMC and its private L1 cache are fully integrated into the node's coherence domain. Like the cores, the RMC supports 32 memory accesses in flight. Fig. 3b illustrates how the logic is modeled as a set of finite state machines that operate as pipelines and eliminate the need for any software processing within the RMC. We model a full crossbar with reliable links between RMCs and a flat latency of 50ns, which is conservative when compared to modern NUMA interconnects, such as QPI and HTX.

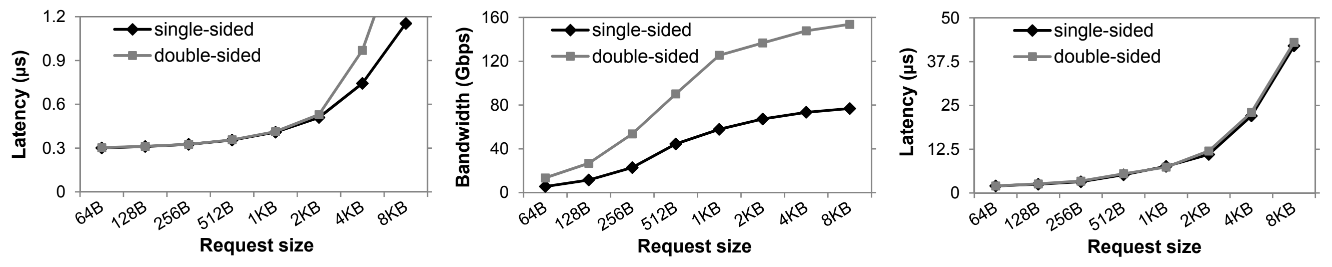
7.2 Microbenchmark: Remote Reads

We first measure the performance of remote read operations between two nodes for both the development platform and the Flexus-based simulated hardware platform. The microbenchmark issues a sequence of read requests of varying size to a preallocated buffer in remote memory. The buffer size exceeds the LLC capacity in both setups. We measure (i) remote read latency with synchronous operations, whereby the issuing core spins after each read request until the reply is received, and (ii) throughput using asynchronous reads, where the issuing core generates a number of non-blocking read requests before processing the replies (similar to Fig. 4).

Fig. 7 plots the latency and bandwidth of remote read operations. Because of space limitations, we only show the latency graph on the emulation side. We run the microbenchmark in both single-sided (only one node reads) and double-sided (both nodes read from each other) mode.

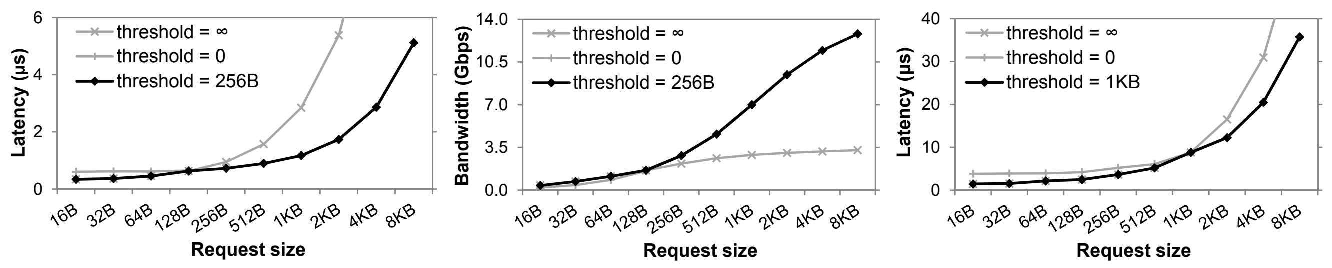
Fig. 7a shows the remote read latency on the simulated hardware as a function of the request size. For small request sizes, the latency is around 300ns, of which 80ns are attributed to accessing the memory (cache hierarchy and DRAM combined) at the remote node and 100ns to round-trip socket-to-socket link latency. The end-to-end latency is within a factor of 4 of the local DRAM access latency. In the double-sided mode, we find that the average latency increases for larger message sizes as compared to the single-sided case. The reason for the drop is cache contention, as each node now has to both service remote read requests and write back the reply data.

Fig. 7b plots the bandwidth between two simulated soNUMA nodes using asynchronous remote reads. For 64B



(a) Remote read latency (sim'd HW). (b) Remote read throughput (sim'd HW). (c) Remote read latency (dev. platform).

Figure 7: Remote read performance on the sim'd HW (latency and bandwidth) and the development platform (latency only).



(a) Send/receive latency (sim'd HW). (b) Send/receive throughput (sim'd HW). (c) Send/receive latency (dev. platform).

Figure 8: Send/receive performance on the sim'd HW (latency and bandwidth) and the development platform (latency only).

requests, we can issue 10M operations per second. For page-sized requests (8KB), we manage to reach 9.6GBps, which is the practical maximum for a DDR3-1600 memory channel. Thanks to the decoupled pipelines of the RMC, the double-sided test delivers twice the single-sided bandwidth.

Fig. 7c shows the latency results on the development platform. The baseline latency is $1.5\mu s$, which is 5x the latency on the simulated hardware. However, we notice that the latency increases substantially with larger request sizes. On the development platform, the RMC emulation module becomes the performance bottleneck as it unrolls large WQ requests into cache-line-sized requests.

7.3 Microbenchmark: Send/Receive

We build a Netpipe [55] microbenchmark to evaluate the performance of the soNUMA unsolicited communication primitives, implemented entirely in software (§5.3). The microbenchmark consists of the following two components: (i) a ping-pong loop that uses the smallest message size to determine the end-to-end one-way latency and (ii) a streaming experiment where one node is sending and the other receiving data to determine bandwidth.

We study the half-duplex latency (Fig. 8a) and bandwidth (Fig. 8b) on our simulation platform. The two methods (`pull`, `push`) expose a performance tradeoff: `push` is optimized for small messages, but has significant processor and packetization overheads. `pull` is optimized for large transfers, but requires additional control traffic at the beginning of each transfer. We experimentally determine the optimal boundary between the two mechanisms by setting the threshold to 0 and ∞ in two separate runs. The black curve shows the performance of our unsolicited primitives with the threshold set to the appropriate value and both mechanisms enabled at the same time. The minimal half-duplex latency is 340ns and the bandwidth exceeds 10Gbps with messages as small as 4KB. For the largest request size evaluated (8KB), the bandwidth achieved is 12.8 Gbps, a 1.6x increase over Quad Data Rate InfiniBand for the same request size [24]. To illustrate the importance of having a combination of `push` and `pull` mechanisms in the user library, we additionally plot in grey their individual performance.

We apply the same methodology on the development platform. The minimal half-duplex latency (see Fig. 8c) is $1.4\mu s$, which is only 4x worse than the simulated hardware. However, the threshold is set to a larger value of 1KB for optimal performance, and the bandwidth is 1/10th of the simulated hardware. Again, we omit the bandwidth graph for the emulation platform due to space limitations. The relatively low bandwidth and a different threshold are due to the overheads of running the fine-grain communication protocol entirely in software (§7.2).

7.4 Comparison with InfiniBand/RDMA

To put our key results in perspective, Table 2 compares the performance of our simulated soNUMA system with an industry-leading commercial solution that combines the Mellanox ConnectX-3 [36] RDMA host channel adapter connected to host Xeon E5-2670 2.60Ghz via a PCIe-Gen3 bus. In the Mellanox system [14], the servers are connected back-to-back via a 56Gbps InfiniBand link. We consider four

Transport	soNUMA		RDMA/IB
	Dev. Plat.	Sim'd HW	[14]
Max BW (Gbps)	1.8	77	50
Read RTT (μs)	1.5	0.3	1.19
Fetch-and-add (μs)	1.5	0.3	1.15
IOPS (Mops/s)	1.97	10.9	35 @ 4 cores

Table 2: A comparison of soNUMA and InfiniBand.

metrics – read bandwidth, read latency, atomic operation latency, and IOPS.

As Table 2 shows, compared to the state-of-the-art RDMA solution, soNUMA reduces the latency to remote memory by a factor of four, in large part by eliminating the PCIe bus overheads. soNUMA is also able to operate at peak memory bandwidth. In contrast, the PCIe-Gen3 bus limits RDMA bandwidth to 50 Gbps, even with 56Gbps InfiniBand. In terms of IOPS, the comparison is complicated by the difference in configuration parameters: the RDMA solution uses four QPs and four cores, whereas the soNUMA configuration uses one of each. Per core, both solutions support approximately 10M remote memory operations.

We also evaluate the performance of atomic operations using fetch-and-add, as measured by the application. For each of the three platforms, the latency of fetch-and-add is approximately the same as that of the remote read operations on that platform. Also, soNUMA provides more desirable semantics than RDMA. In the case of RDMA, fetch-and-add is implemented by the host channel adapter, which requires the adapter to handle all accesses, even from the local node. In contrast, soNUMA's implementation within the node's local cache coherence provides global atomicity guarantees for any combination of local and remote accesses.

7.5 Application Study

Large-scale graph processing engines, key-value stores, and on-line graph query processing are the obvious candidate applications for soNUMA. All of them perform very little work per data item (if any) and operate on large datasets, and hence typically require large scale-out configurations to keep all the data memory resident. Most importantly, they exhibit poor locality as they frequently access non-local data.

For our application study, we picked graph processing and the canonical PageRank algorithm [45], and compared three parallel implementations of it. All three are based on the widely used Bulk Synchronous Processing model [57], in which every node computes its own portion of the dataset (range of vertices) and then synchronizes with other participants, before proceeding with the next iteration (so-called superstep). Our three implementations are:

(i) `SHM(pthreads)`: The baseline is a standard pthreads implementation that assumes cache-coherent memory rather than soNUMA. For the simulated hardware, we model an eight-core multiprocessor with 4MB of LLC per core. We provision the LLC so that the aggregate cache size equals

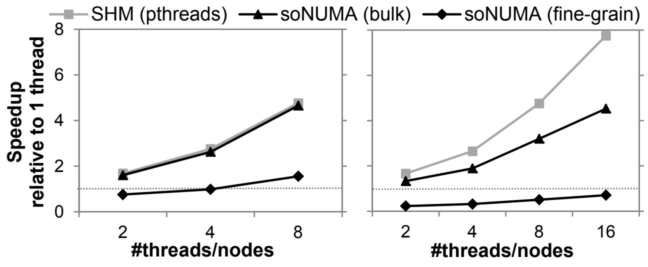


Figure 9: PageRank speedup on simulated HW (left) and on the development platform (right).

that of the eight machines in the soNUMA setting. Thus, no benefits can be attributed to larger cache capacity in the soNUMA comparison. For the development platform, we simply run the application on our ccNUMA server described in §7.1, but without a hypervisor running underneath the host OS. In this implementation, the application stores two rank values for each vertex: the one computed in the previous superstep and the one currently being computed. Barrier synchronization marks the end of each superstep.

(ii) `soNUMA(bulk)`: This implementation leverages aggregation mechanisms and exchanges ranks between nodes at the end of each superstep, after the barrier. Such an approach is supported by Pregel [35] as it amortizes high inter-node latencies and makes use of wide high-capacity links. In this implementation we exploit spatial locality within the global address space by using multi-line requests at the RMC level. At the end of each superstep, every node uses multiple `rmc_read_async` operations (one per peer) to pull the remote vertex information from each of its peers into the local memory. This allows a concurrent shuffle phase limited only by the bisection bandwidth of the system.

(iii) `soNUMA(fine-grain)`: This variant leverages the fine-grain memory sharing capabilities of soNUMA, as shown in Fig. 4. Each node issues one `rmc_read_async` operation for each non-local vertex. This implementation resembles the shared-memory programming model of `SHM(pthreads)`, but has consequences: the number of remote memory operations scales with the number of edges that span two partitions rather than with the number of vertices per partition.

We evaluate the three implementations of PageRank on a subset of the Twitter graph [29] using a naive algorithm that randomly partitions the vertices into sets of equal cardinality. We run 30 supersteps for up to 16 soNUMA nodes on the development platform. On the simulator, we run a single superstep on up to eight nodes because of the high execution time of the cycle-accurate model.

Fig. 9 (left) shows the speedup over the single-threaded baseline of the three implementations on the simulated hardware. Both `SHM(pthreads)` and `soNUMA(bulk)` have near identical speedup. In both cases, the speedup trend is determined primarily by the imbalance resulting from the graph partitioning scheme, and not the hardware. However, `soNUMA(fine-grain)` has noticeably greater overheads, primarily because of the limited per-core remote read rate (due to the software API's overhead on each request) and the fact that each reference to a non-local vertex results in a remote read operation. Indeed, each core can only issue up to 10 million remote operations per second. As shown in Fig. 7b, the bandwidth corresponding to 64B requests is a small fraction of the maximum bandwidth of the system.

Fig. 9 (right) shows the corresponding speedup on the software development platform. We identify the same general trends as on the simulated hardware, with the caveat that the higher latency and lower bandwidth of the development platform limit performance.

8. Discussion

We now discuss the lessons learned so far in developing our soNUMA prototype, known open issues, possible killer applications, and deployment and packaging options.

Lessons learned. During our work, we appreciated the value of having a software development platform capable of running at native speeds. By leveraging hardware virtualization and dedicating processing cores to emulate RMCs, we were able to run an (emulated) soNUMA fabric at wall-clock execution time, and use that platform to develop and validate the protocol, the kernel driver, all user-space libraries, and applications.

Open issues. Our design provides the ability to support many variants of remote memory operations that can be handled in a stateless manner by the peer RMC. This includes read, write, and atomic operations. A complete architecture will probably require extensions such as the ability to issue remote interrupts as part of an RMC command, so that nodes can communicate without polling. This will have a number of implications for system software, e.g., to efficiently convert interrupts into application messages, or to use the mechanisms to build system-level abstractions such as global buffer caches.

Killer applications. Our primary motivation behind soNUMA is the conflicting trend of (i) large dataset applications that require tight and unpredictable sharing of resources; and (ii) manycore designs, which are optimized for throughput rather than resource aggregation. Our soNUMA proposal aims to reconcile these two trends. In our evaluation, we chose a simple graph application because it allows for multiple implementation variants that expose different aspects of the programming model, even though the regular, batch-oriented nature of that application is also a good fit for coarse-grain, scale-out models. Implementing these variants using the RMC API turned out to be almost as easy as using the conventional shared-memory programming abstractions. Many applications such as on-line graph processing algorithms, in-memory transaction processing systems, and key-value stores demand low latency [50] and can take advantage of one-sided read operations [38]. These applications are designed to assume that both client and server have access to a low-latency fabric [38, 43], making them killer applications for soNUMA. Beyond these, we also see deployment opportunities in upper application tiers of the datacenter. For example, Oracle Exalogic today provides a flexible, low-latency platform delivered on a rack-scale InfiniBand cluster [44]. Such deployments are natural candidates for tighter integration using SoC and soNUMA.

System packaging options. The evaluation of the impact of system scale and fabric topologies is outside of the scope of this paper since we focused on the end-behavior of the RMC. To be successful, soNUMA systems will have to be sufficiently large to capture very large datasets within a single cluster, and yet sufficiently small to avoid introducing new classes of problems, such as the need for fault containment [11]. Industry solutions today provide rack-insert solutions that are ideally suited for soNUMA, e.g., HP's Moonshot with 45 cartridges or Calxeda-based Viridis systems with 48 nodes in a 2U chassis. Beyond that, rack-level solutions seem like viable near-term options; a 44U rack of Viridis chassis can thus provide over 1000 nodes within a two-meter diameter, affording both low wire delay and massive memory capacity.

Research directions. This work has demonstrated the benefits on simple microbenchmarks and one application. We plan to evaluate the impact of soNUMA on latency-sensitive applications such as in-memory transaction processing systems and on-line query processing. We also see research questions around system-level resource sharing, e.g., to create a single-system image or a global filesystem buffer cache, or to rethink resource management in hypervisor clusters. Multikernels, such as Barrelfish [4], could be an ideal candidate for soNUMA.

We plan to investigate the micro-architectural aspects of the RMC. This includes further latency-reducing optimizations in the RMC and the processor, a reassessment of the RMC's design for SoC's with high core counts, and investigation of the flow-control, congestion and occupancy issues in larger fabrics.

Our study focused on data sharing within a single soNUMA fabric. For very large datasets, datacenter deployments would likely interconnect multiple rack-scale soNUMA systems using conventional networking technologies. This opens up new systems-level research questions in the areas of resource management (e.g., to maximize locality) and networking (e.g., how to use the soNUMA fabric to run network protocols).

9. Related Work

Many of the concepts found in remote memory architectures today and our soNUMA proposal originate from research done in the '80s and '90s. In this section we look at the relationship between soNUMA and several related concepts. We group prior work into six broad categories.

Partitioned global address space. PGAS relies on compiler and language support to provide the abstraction of a shared address space on top of non-coherent, distributed memory [13]. Languages such as Unified Parallel C [13, 61] and Titanium [61] require the programmer to reason about data partitioning and be aware of data structure non-uniformity. However, the compiler frees the programmer from the burden of ensuring the coherence of the global address space by automatically converting accesses to remote portions into one-sided remote memory operations that correspond to soNUMA's own primitives. PGAS also provides explicit asynchronous remote data operations [7], which also easily map onto soNUMA's asynchronous library primitives. The efficiency of soNUMA remote primitives would allow PGAS implementations to operate faster.

Software distributed shared memory. Software distributed shared memory (DSM) provides global coherence not present in the memory hierarchies of PGAS and soNUMA. Pure software DSM systems such as IVY [31], Munin [10] and Threadmarks [2] expose a global coherent virtual address space and rely on OS mechanisms to "fault in" pages from remote memory on access and propagate changes back, typically using relaxed memory models. Similarly, software DSM can be implemented within a hypervisor to create a cache-coherent global guest-physical address space [12], or entirely in user-space via binary translation [51]. Like software DSM, soNUMA operates at the virtual memory level. Unlike software DSM, soNUMA and PGAS target fine-grained accesses whereas software DSM typically operates at the page level. Shasta [51] and Blizzard [52] offer fine-grain DSM through code instrumentation and hardware assistance, respectively, but in both cases with non-negligible software overheads.

Cache-coherent memory. ccNUMA designs such as Alewife [1], Dash [30], FLASH [28], Fugu [34], Typhoon [48], Sun Wildfire [42], and today's Intel QPI and AMD HTX architectures create a compute fabric of processing elements, each with its own local memory, and provide cache-coherent physical memory sharing across the nodes. FLASH and Sun's Wildfire also provide advanced migration and replication techniques to reduce the synchronization overheads [19].

soNUMA shares the non-uniform aspect of memory with these designs and leverages the lower levels of the ccNUMA protocols, but does not attempt to ensure cache coherence. As a result, soNUMA uses a stateless protocol, whereas ccNUMA requires some global state such as directories to ensure coherence, which limits its scalability. The ccNUMA designs provide a global physical address space, allowing conventional single-image operating systems to run on top. The single-image view, however, makes the system less resilient to faults [11]. In contrast, soNUMA exposes the abstraction of global virtual address spaces on top of multiple OS instances, one per coherence domain.

User-level messaging. User-level messaging eliminates the overheads of kernel transitions by exposing communication directly to applications. Hybrid ccNUMA designs such as FLASH [23], Alewife [1], Fugu [34], and Typhoon [20] provide architectural support for user-level messaging in conjunction with cache-coherent memory. In contrast, soNUMA's minimal design allows for an efficient implementation of message passing entirely in software using one-sided remote memory operations.

SHRIMP [6] uses a specialized NIC to provide user-level messaging by allowing processes to directly write the memory of other processes through hardware support. Cashmere [56] leverages DEC's Memory Channel [22], a remote-write network, to implement a software DSM. Unlike SHRIMP and Cashmere, soNUMA also allows for direct reads from remote memory.

Fast Messages [46] target low latency and high bandwidth for short user-level messages. U-Net [58] removes the entire OS/Kernel off the critical path of messaging. These systems all focus on the efficient implementation of a message send. In soNUMA, the RMC provides architectural support for both one-sided read and write operations; messaging is implemented on top of these basic abstractions.

Remote memory access. Unlike remote write networks, such as SHRIMP and DEC Memory Channel, Remote Memory Access provides for both remote reads and remote writes. Such hardware support for one-sided operations, similar to soNUMA's, was commercialized in supercomputers such as Cray T3D [27] and T3E [53]. Remote memory access can also be implemented efficiently for graph processing on commodity hardware by leveraging aggressive multithreading to compensate for the high access latency [41]. soNUMA also hides latency but uses asynchronous read operations instead of multithreading.

Today, user-level messaging and RDMA is available in commodity clusters with RDMA host channel adapters such as Mellanox ConnectX-3 [36] that connect into InfiniBand or Converged Ethernet switched fabrics [25]. To reduce complexity and enable SoC integration, soNUMA only provides a minimal subset of RDMA operations; in particular, it does not support reliable connections, as they require keeping per-connection state in the adapter.

NI integration. One advantage of soNUMA over prior proposals on fast messaging and remote one-sided primitives is the tight integration of the NI into the coherence domain. The advantage of such an approach was previously demonstrated in Coherent Network Interfaces (CNI) [40], which leverage the coherence mechanism to achieve low-latency communication of the NI with the processors, using cacheable work queues. More recent work showcases the advantage of integration, but in the context of kernel-level TCP/IP optimizations, such as a zero-copy receive [5]. Our RMC is fully integrated into the local cache coherence hierarchy and does not depend on local DMA operations. The simple design of the RMC suggests that integration into the local cache-coherence domain is practical. Our evaluation shows that such integration can lead to substantial benefits by keeping the control data structures, such as the QPs and page tables, in caches. soNUMA also provides global atomicity by implementing atomic operations within a node's cache hierarchy.

10. Conclusion

Scale-Out NUMA (soNUMA) is an architecture, programming model, and communication protocol for low-latency big-data processing. soNUMA eliminates kernel, network stack, and I/O bus overheads by exposing a new hardware block –the remote memory controller– within the cache coherent hierarchy of the processor. The remote memory controller is directly accessible by applications and connects directly into a NUMA fabric. Our results based on cycle-accurate full-system simulation show that soNUMA can achieve remote read latencies that are within 4x of local DRAM access, stream at full memory bandwidth, and issue up to 10M remote memory operations per second per core.

Acknowledgments

The authors thank Dushyanth Narayanan and Aleksandar Dragojevic from Microsoft Research for many inspiring conversations on rack-scale computing requirements and applications early in the project. We thank Amitabha Roy, Ryan Stutsman, David Wood, and the anonymous reviewers for their precious feedback on the manuscript. Patrick Tae ran netpipe between Calxeda nodes.

This work has been partially funded by the *EuroCloud* project of the 7th Framework Program of the European Commission, and by the *Workloads and Server Architectures for Green Datacenters* project of the Swiss National Science Foundation.

References

[1] A. Agarwal, R. Bianchini, D. Chaiken, K. L. Johnson, D. A. Kranz, J. Kubiatowicz, B.-H. Lim, K. Mackenzie, and D. Yeung. The MIT Alewife Machine: Architecture and Performance. In *Proceedings of the 22nd International Symposium on Computer Architecture (ISCA)*, 1995.

[2] C. Amza, A. L. Cox, S. Dwarkadas, P. J. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. Treadmarks: Shared Memory Computing on Networks of Workstations. *IEEE Computer*, 29(2):18–28, 1996.

[3] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. L. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[4] A. Baumann, P. Barham, P.-É. Dagand, T. L. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The Multikernel: a New OS Architecture for Scalable Multicore Systems. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, 2009.

[5] N. L. Binkert, A. G. Saidi, and S. K. Reinhardt. Integrated Network Interfaces for High-Bandwidth TCP/IP. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XII)*, 2006.

[6] M. A. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. W. Felten, and J. Sandberg. Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer. In *Proceedings of the 21st International Symposium on Computer Architecture (ISCA)*, 1994.

[7] D. Bonachea. Proposal for Extending the UPC Memory Copy Library Functions and Supporting Extensions to GASNet, Version 2.0. 2007.

[8] Calxeda Inc. Calxeda Energy Core ECX-1000 Fabric Switch. `http://www.calxeda.com/architecture/fabric/`, 2012.

[9] Calxeda Inc. ECX-1000 Technical Specifications. `http://www.calxeda.com/ecx-1000-techspecs/`, 2012.

[10] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and Performance of Munin. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP)*, 1991.

[11] J. Chapin, M. Rosenblum, S. Devine, T. Lahiri, D. Teodosiu, and A. Gupta. Hive: Fault Containment for Shared-Memory Multiprocessors. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP)*, 1995.

[12] M. Chapman and G. Heiser. vNUMA: A Virtual Shared-Memory Multiprocessor. In *Proceedings of the 2009 conference on USENIX Annual Technical Conference*, 2009.

[13] C. Coarfa, Y. Dotsenko, J. M. Mellor-Crummey, F. Cantonnet, T. A. El-Ghazawi, A. Mohanti, Y. Yao, and D. G. Chavarría-Miranda. An Evaluation of Global Address Space Languages: Co-Array Fortran and Unified Parallel C. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, 2005.

[14] D. Crupnicoff. Personal communication (Mellanox Corp.), 2013.

[15] D. E. Culler, A. C. Arpaci-Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. A. Yelick. Parallel Programming in Split-C. In *Proceedings of the 1993 ACM/IEEE Conference on Supercomputing (SC)*, 1993.

[16] M. Davis and D. Borland. System and Method for High-Performance, Low-Power Data Center Interconnect Fabric. WO Patent 2,011,053,488, 2011.

[17] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's Highly Available Key-Value Store. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP)*, 2007.

[18] A. Dhodapkar, G. Lauterbach, S. Li, D. Mallick, J. Bauman, S. Kanthadai, T. Kuzuhara, G. S. M. Xu, and C. Zhang. SeaMicro SM10000-64 Server: Building Datacenter Servers Using Cell Phone Chips. In *Proceedings of the 23rd IEEE HotChips Symposium*, 2011.

[19] B. Falsafi and D. A. Wood. Reactive NUMA: A Design for Unifying S-COMA and CC-NUMA. In *Proceedings of the 24th International Symposium on Computer Architecture (ISCA)*, 1997.

[20] B. Falsafi, A. R. Lebeck, S. K. Reinhardt, I. Schoinas, M. D. Hill, J. R. Larus, A. Rogers, and D. A. Wood. Application-Specific Protocols for User-Level Shared Memory. In *Proceedings of the 1994 ACM/IEEE Conference on Supercomputing (SC)*, 1994.

[21] M. Flajslik and M. Rosenblum. Network Interface Design for Low Latency Request-Response Protocols. In *Proceedings of the 2013 USENIX Annual Technical Conference*, 2013.

[22] R. Gillett. Memory Channel: An Optimized Cluster Interconnect. *IEEE Micro*, 16(2):12–18, 1996.

[23] J. Heinlein, K. Gharachorloo, S. Dresser, and A. Gupta. Integration of Message Passing and Shared Memory in the Stanford FLASH Multiprocessor. In *Proceedings of the 6th International Conference*

on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI), 1994.

[24] HPC Advisory Council. Interconnect Analysis: 10GigE and InfiniBand in High Performance Computing. `http://www.hpcadvisorycouncil.com/pdf/IB_and_10GigE_in_HPC.pdf`, 2009.

[25] *IEEE 802.1Qbb: Priority-Based Flow Control.* IEEE, 2011.

[26] InfiniBand Trade Association. *InfiniBand Architecture Specification: Release 1.0.* 2000.

[27] R. Kessler and J. Schwarzmeier. Cray T3D: A New Dimension for Cray Research. In *Compcon Spring '93, Digest of Papers*, 1993.

[28] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. L. Hennessy. The Stanford FLASH Multiprocessor. In *Proceedings of the 21st International Symposium on Computer Architecture (ISCA)*, 1994.

[29] H. Kwak, C. Lee, H. Park, and S. B. Moon. What is Twitter, a Social Network or a News Media? In *Proceedings of the 19th International Conference on World Wide Web (WWW)*, 2010.

[30] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. L. Hennessy, M. Horowitz, and M. S. Lam. The Stanford Dash Multiprocessor. *IEEE Computer*, 25(3):63–79, 1992.

[31] K. Li and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Trans. Comput. Syst.*, 7(4):321–359, 1989.

[32] J. Liu, J. Wu, and D. K. Panda. High Performance RDMA-Based MPI Implementation over InfiniBand. *International Journal of Parallel Programming*, 32(3):167–198, 2004.

[33] P. Lotfi-Kamran, B. Grot, M. Ferdman, S. Volos, Y. O. Koçberber, J. Picorel, A. Adileh, D. Jevdjic, S. Idgunji, E. Özer, and B. Falsafi. Scale-Out Processors. In *Proceedings of the 39th International Symposium on Computer Architecture (ISCA)*, 2012.

[34] K. Mackenzie, J. Kubiatowicz, A. Agarwal, and F. Kaashoek. Fugu: Implementing Translation and Protection in a Multiuser, Multimodel Multiprocessor. In *Proceedings of the 1994 Workshop on Shared Memory Multiprocessors*, 1994.

[35] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A System for Large-Scale Graph Processing. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2010.

[36] Mellanox Corp. ConnectX-3 Pro Product Brief. `http://www.mellanox.com/related-docs/prod_adapter_cards/PB_ConnectX-3_Pro_Card_EN.pdf`, 2012.

[37] Mellanox Corp. RDMA Aware Networks Programming User Manual, Rev 1.4. `http://www.mellanox.com/related-docs/prod_software/RDMA_Aware_Programming_user_manual.pdf`, 2013.

[38] C. Mitchell, Y. Geng, and J. Li. Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store. In *Proceedings of the USENIX Annual Technical Conference*, 2013.

[39] S. Mukherjee, P. Bannon, S. Lang, A. Spink, and D. Webb. The Alpha 21364 Network Architecture. In *Hot Interconnects IX*, 2001.

[40] S. S. Mukherjee, B. Falsafi, M. D. Hill, and D. A. Wood. Coherent Network Interfaces for Fine-Grain Communication. In *Proceedings of the 23rd International Symposium on Computer Architecture (ISCA)*, 1996.

[41] J. Nelson, B. Myers, A. H. Hunter, P. Briggs, L. Ceze, C. Ebeling, D. Grossman, S. Kahan, and M. Oskin. Crunching Large Graphs with Commodity Processors. In *Proceedings of the 3rd USENIX Conference on Hot Topics in Parallelism (HotPar)*, 2011.

[42] L. Noordergraaf and R. van der Pas. Performance Experiences on Sun's WildFire Prototype. In *Proceedings of the ACM/IEEE Conference on Supercomputing (SC)*, 1999.

[43] D. Ongaro, S. M. Rumble, R. Stutsman, J. K. Ousterhout, and M. Rosenblum. Fast Crash Recovery in RAMCloud. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[44] Oracle Corp. Oracle Exalogic Elastic Cloud X3-2 (Datasheet). `http://www.oracle.com/us/products/middleware/exalogic/overview/index.html`, 2013.

[45] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank Citation Ranking: Bringing Order to the Web. *Stanford InfoLab Technical Report*, 1999.

[46] S. Pakin, M. Lauria, and A. Chien. High performance messaging on workstations: Illinois Fast Messages (FM) for Myrinet. In *Proceedings of the 1995 ACM/IEEE Conference on Supercomputing*. ACM, 1995.

[47] R. Recio, B. Metzler, P. Culley, J. Hilland, and D. Garcia. A Remote Direct Memory Access Protocol Specification. RFC 5040, 2007.

[48] S. K. Reinhardt, J. R. Larus, and D. A. Wood. Tempest and Typhoon: User-Level Shared Memory. In *Proceedings of the 21st International Symposium on Computer Architecture (ISCA)*, 1994.

[49] P. Rosenfeld, E. Cooper-Balis, and B. Jacob. DRAMSim2: A Cycle Accurate Memory System Simulator. *Computer Architecture Letters*, 10(1):16–19, 2011.

[50] S. M. Rumble, D. Ongaro, R. Stutsman, M. Rosenblum, and J. K. Ousterhout. It's Time for Low Latency. In *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems*, 2011.

[51] D. J. Scales, K. Gharachorloo, and C. A. Thekkath. Shasta: A Low Overhead, Software-Only Approach for Supporting Fine-Grain Shared Memory. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, 1996.

[52] I. Schoinas, B. Falsafi, A. R. Lebeck, S. K. Reinhardt, J. R. Larus, and D. A. Wood. Fine-Grain Access Control for Distributed Shared Memory. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)*, 1994.

[53] S. L. Scott and G. M. Thorson. The Cray T3E Network: Adaptive Routing in a High Performance 3D Torus. In *Hot Interconnects IV*, 1996.

[54] S. Shelach. Mellanox wins $200m Google, Microsoft deals. `http://www.globes.co.il/serveen/globes/docview.asp?did=1000857043&fid=1725`, 2013.

[55] Q. O. Snell, A. R. Mikler, and J. L. Gustafson. Netpipe: A Network Protocol Independent Performance Evaluator. In *IASTED International Conference on Intelligent Information Management and Systems*, volume 6, 1996.

[56] R. Stets, S. Dwarkadas, N. Hardavellas, G. C. Hunt, L. I. Kontothanassis, S. Parthasarathy, and M. L. Scott. Cashmere-2L: Software Coherent Shared Memory on a Clustered Remote-Write Network. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP)*, 1997.

[57] L. G. Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM*, 33(8):103–111, 1990.

[58] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP)*, 1995.

[59] T. Wenisch, R. Wunderlich, M. Ferdman, A. Ailamaki, B. Falsafi, and J. Hoe. SimFlex: Statistical Sampling of Computer System Simulation. *IEEE Micro*, 26:18 –31, 2006.

[60] WinterCorp. Big Data and Data Warehousing. http://www.wintercorp.com/.

[61] K. A. Yelick, D. Bonachea, W.-Y. Chen, P. Colella, K. Datta, J. Duell, S. L. Graham, P. Hargrove, P. N. Hilfinger, P. Husbands, C. Iancu, A. Kamil, R. Nishtala, J. Su, M. L. Welcome, and T. Wen. Productivity and Performance Using Partitioned Global Address Space Languages. In *Workshop on Parallel Symbolic Computation (PASCO)*, 2007.