A Flexible & Efficient Shared Memory Abstraction with Minimal HW Assistance

Nikolaos Kallimanis  Manolis Marazakis  Nikolaos Chrysos

Computer Architecture and VLSI Systems (CARV) Laboratory
FORTH – ICS
Communication mechanisms of the Unimem architecture:

1. Load/Store instructions across remote nodes.
2. Every page of physical memory is cacheable only in a single node.
3. Efficiently copying large amounts of memory from/to remote nodes.
4. Send and receive of small atomic messages in a low latency manner.

Towards exascale.
How to exercise the Unimem remote memory?
Unimem’s APIs

- Apps (Unimem optimized)
- Apps (unmodified)
- Apps (unmodified)
- Apps (unmodified)

Global Shared Address Space

Runtimes (Unimem oriented)

Unimem Interfaces (Remote DMA, etc.)

Unimem Sockets (modified libc)

System Software - Unimem testbed (drivers, etc.)

Remote Memory SWAP

HW - Unimem testbed (RDMA, virtual mailbox, virtual packetizer, remote memory access)
GSAS - Global Shared Address Space

1. Global Shared Address Space across system’s remote nodes.

2. It is mostly implemented based on mechanisms for sending/receiving small messages atomically.
   - No complex hw-coherence protocols.
   - Flexibility.

3. API resembles to shared memory communication.

4. Applications can use this API for synchronization and for utilizing remote memory.

5. Data are cached in the node that reside on → cacheable at single node.
   - This is a Unimem property.
GSAS - SW & HW Stack

SW

<table>
<thead>
<tr>
<th>Node #1</th>
<th>Node #2</th>
<th>Node #3</th>
<th>Node #4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unimem - User level libraries</td>
<td>Unimem - User level libraries</td>
<td>Unimem - User level libraries</td>
<td>Unimem - User level libraries</td>
</tr>
<tr>
<td>Drivers</td>
<td>Drivers</td>
<td>Drivers</td>
<td>Drivers</td>
</tr>
<tr>
<td>mbox</td>
<td>mbox</td>
<td>mbox</td>
<td>mbox</td>
</tr>
<tr>
<td>packetizer</td>
<td>packetizer</td>
<td>packetizer</td>
<td>packetizer</td>
</tr>
</tbody>
</table>

HW

Unimem Interconnect (i.e. Exanet)
Overview of the GSAS environment

The GSAS Address Space is a 64-bit Global Address Space. The first 16 bits contain the routing information, specifically node-id. The remaining 48 bits index the memory of each node.
Applications that use the GSAS API are able to:

- **Allocate** of memory in any remote node.
- **Spawn** new processes on any remote node.
- Execute **atomic operations** (i.e., CAS, FAD, SWAP, etc.) on any remote memory location.

Low latency primitives (Exanet on QFDB prototype).

- $\approx 1 - 2 \ \mu$sec for a remote issued atomic instruction.
- A few nsec for local issued atomic instruction.
## Application Programming Interface

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>allocSharedPage</td>
<td>Allocation of remote/local memory</td>
</tr>
<tr>
<td>freeSharedPage</td>
<td>Free allocated memory</td>
</tr>
<tr>
<td>remoteFork</td>
<td>Spawn of a new process on some remote node</td>
</tr>
<tr>
<td>Read/Write</td>
<td>Read/Write operations</td>
</tr>
<tr>
<td>CAS</td>
<td>Compare&amp;Swap operations</td>
</tr>
<tr>
<td>FAD</td>
<td>Fetch&amp;Add operations</td>
</tr>
<tr>
<td>SWAP</td>
<td>SWAP operations</td>
</tr>
<tr>
<td>BarrierJoin/BarrierDestroy</td>
<td>Functionality for Barriers</td>
</tr>
</tbody>
</table>
There is an atomic service at each node that serves remote requests.

Atomic service is running on core 0 on every node of the system.

Apps and the atomic service communicate through small atomic messages with low latency.

There is a user-space library that handles the requests on the issuer side.
Processes perform atomic operations on the allocated space on any node (local or remote).
- i.e., CAS, FAD, atomic READ, etc.
- Only operations performed by remote process are applied by the atomic service ⇒ Improved Performance.
Experiments on a Unimem testbed (2 QFDB board):

- Each board is equipped with 4 nodes, each of which:
  - Zynq MP Ultrascale+ SoC.
  - 4 Arm A53 cores @ 1.4 GHz.
  - 16 GB of local DDR4.
  - Exanet network interfaces.
### Latency Microbenchmarks

<table>
<thead>
<tr>
<th></th>
<th>Trenz prototype</th>
<th>QFDB (1 hop)</th>
<th>QFDB (2 hop)</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>GSAS Write</td>
<td>4.0 usec</td>
<td>1.0 usec</td>
<td>1.5 usec</td>
<td>64-bit write</td>
</tr>
<tr>
<td>GSAS Read</td>
<td>4.0 usec</td>
<td>1.5 usec</td>
<td>2.0 usec</td>
<td>64-bit read</td>
</tr>
<tr>
<td>GSAS Fetch&amp;Add</td>
<td>4.0 usec</td>
<td>1.5 usec</td>
<td>2.0 usec</td>
<td>64-bit Fetch&amp;Add</td>
</tr>
<tr>
<td>Small Message Transfer</td>
<td>1.9 usec</td>
<td>0.7 usec</td>
<td>1.2 usec</td>
<td>32 bytes - one way</td>
</tr>
</tbody>
</table>
GSAS Performance Evolution

![Graph showing GSAS performance evolution with data points for Juno Prototype (2016), Trenz Prototype (2017), and QFDB Prototype (2018). The graph compares remote and local reference performance.](image-url)
Example App: Distributed Hash Table

- A concurrent Distributed Hash Table is implemented on top of the GSAS environment.
- Any thread that runs on any node of the system is able to access/modify the stored data by using **Put** and **Get** operations.
- Put and Get operations are executed concurrently.
  - There is no dedicated server for serving the requests.
- The data structure is able to use the memory of all available cores.
DHT Performance on GSAS

Workload = 100% DhtGet

Throughput millions operations/sec

# nodes

Trenz prototype
QFDB prototype
DHT Performance on GSAS

Workload = 80% DhtGet + 20% DhtPut

Throughput millions operations/sec

- Trenz Prototype
- QFDB Prototype
Conclusions

- GSAS provides Global Shared Address Space across system’s remote nodes.
- It is mostly implemented based on mechanisms for sending/receiving small atomic messages.
  - No complex hw-coherence protocols.
  - Flexibility.
- API resembles to shared memory communication.
- The latency of remote operations is about 1 – 2 usec (1 or 2 hop distance).
- A GSAS use-case example is considered, i.e. a Distributed HashTable.
Thank You