

# SANDIA REPORT

SAND2018-9199  
Unlimited Release  
Printed August 21, 2018

## Opal: A Centralized Memory Manager for Investigating Disaggregated Memory Systems

V.R. Kommareddy and A. Awad  
Secure and Advanced Computer Architecture Research Group  
University of Central Florida  
Orlando, FL 32816  
vamseereddy8@Knights.ucf.edu, amro.awad@ucf.edu

C. Hughes and S.D. Hammond  
Center for Computing Research  
Sandia National Laboratories  
Albuquerque, NM 87185  
{chughes, sdhammo}@sandia.gov

Prepared by  
Sandia National Laboratories  
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

Approved for public release; further dissemination unlimited.



**Sandia National Laboratories**

Issued by Sandia National Laboratories, operated for the United States Department of Energy by National Technology and Engineering Solutions of Sandia, LLC.

**NOTICE:** This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government, nor any agency thereof, nor any of their employees, nor any of their contractors, subcontractors, or their employees, make any warranty, express or implied, or assume any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represent that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government, any agency thereof, or any of their contractors or subcontractors. The views and opinions expressed herein do not necessarily state or reflect those of the United States Government, any agency thereof, or any of their contractors.

Printed in the United States of America. This report has been reproduced directly from the best available copy.

Available to DOE and DOE contractors from  
U.S. Department of Energy  
Office of Scientific and Technical Information  
P.O. Box 62  
Oak Ridge, TN 37831

Telephone: (865) 576-8401  
Facsimile: (865) 576-5728  
E-Mail: [reports@adonis.osti.gov](mailto:reports@adonis.osti.gov)  
Online ordering: <http://www.osti.gov/bridge>

Available to the public from  
U.S. Department of Commerce  
National Technical Information Service  
5285 Port Royal Rd  
Springfield, VA 22161

Telephone: (800) 553-6847  
Facsimile: (703) 605-6900  
E-Mail: [orders@ntis.fedworld.gov](mailto:orders@ntis.fedworld.gov)  
Online ordering: <http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online>



# Opal: A Centralized Memory Manager for Investigating Disaggregated Memory Systems

V.R. Kommareddy<sup>1</sup>, C. Hughes<sup>2</sup>, S. Hammond<sup>2</sup>, and A. Awad<sup>1</sup>

<sup>1</sup>SACA Research Group, University of Central Florida, Orlando, FL 32816

<sup>2</sup>Center for Computing Research, Sandia National Laboratories, Albuquerque, NM 87185

## Abstract

Many modern applications have memory footprints that are increasingly large, driving system memory capacities higher and higher. Moreover, these systems are often organized where the bulk of the memory is collocated with the compute capability, which necessitates the need for message passing APIs to facilitate information sharing between compute nodes. Due to the diversity of applications that must run on High-Performance Computing (HPC) systems, the memory utilization can fluctuate wildly from one application to another. And, because memory is located in the node, maintenance can become problematic because each node must be taken offline and upgraded individually.

To address these issues, vendors are exploring disaggregated, memory-centric, systems. In this type of organization, there are discrete nodes, reserved solely for memory, which are shared across many compute nodes. Due to their capacity, low-power, and non-volatility, Non-Volatile Memories (NVMs) are ideal candidates for these memory nodes. This report discusses a new component for the Structural Simulation Toolkit (SST), Opal, that can be used to study the impact of using NVMs in a disaggregated system in terms of performance, security, and memory management.

This page intentionally left blank.

# Contents

<b>1</b>	<b>Introduction</b>	<b>7</b>
<b>2</b>	<b>Opal Component</b>	<b>9</b>
	Integrating Opal in Simulated Systems .....	12
	Opal Configuration .....	14
	Example Opal Configuration .....	14
	Opal Requests .....	16
	Memory Allocation Policies .....	17
	Communication Between Nodes .....	17
<b>3</b>	<b>Evaluation</b>	<b>19</b>
<b>4</b>	<b>Conclusion</b>	<b>23</b>
	<b>References</b>	<b>24</b>

# List of Figures

1.1	An example of a disaggregated memory system. The system has several nodes (SoCs) where each node may have its own internal memory but share external memory. ....	7
2.1	A simulated system that uses Opal for centralized memory management. ....	10
2.2	Implementing access control for direct access scheme. ....	11
2.3	Implementing access control for virtualized system memory. ....	12
2.4	Example configuration ....	16
3.1	Disaggregated memory system performance in instructions per cycle for different memory allocation policies by varying number of nodes and number of shared memory pools ....	20
3.2	Comparing performance of disaggregated memory system by varying different memory allocation policies ....	21

# Chapter 1

## Introduction

With the arrival of the big data era, the need for fast processing and access to shared memory structures has never been as crucial as it is today. For better reliability, upgradability and flexibility, major vendors are considering designs that have disaggregated memory systems, which can be accessed by a large number of processing nodes. The fact that such systems are disaggregated allows upgrading memory, isolating malicious or unreliable nodes, and enables easy integration of heterogeneous compute nodes (e.g. GPUs, FPGAs, and custom accelerators). To better understand how a disaggregated memory system is organized, please refer to Figure 1.1, which depicts a sample disaggregated memory system.

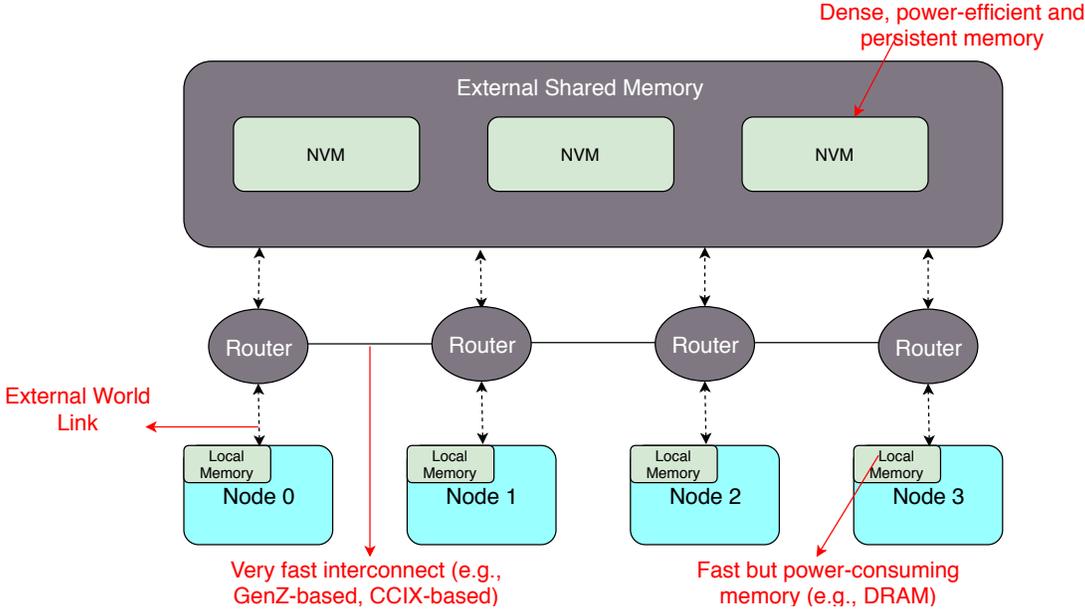


Figure 1.1: An example of a disaggregated memory system. The system has several nodes (SoCs) where each node may have its own internal memory but share external memory.

As shown in Figure 1.1, the nodes must access an off-chip network to access the external memory. Although local updates to external memory locations can be made visible to all other nodes, scaling the coherence protocol is challenging. While using directories could help, there are still inherent design and performance complexities that can arise. One direction that vendors are considering is the use of software to flush updates in local caches to the shared memory and

make it visible to other nodes. One can think of it as having a lock around the shared data, and not releasing it until all of the updates have been flushed to the external memory. Once the lock is released, the other nodes need to make sure they are reading the data from the external memory rather than their internal caches. One way to do that is to use `clflush` after any reads or updates, which guarantees copies of that memory location are invalidated in the cache hierarchy. Other use cases include partitioning the memory carefully between nodes, where each node signals all of its updates and flushes. After which, an aggregator node can read the updated values from the external memory. In much simpler cases, such as a file containing a large social network graph where no updates are expected to that graph (read-only), there is no need for special handling of accesses to the graph.

The Structural Simulation Toolkit (SST) [6] has been proven to be one of the most reliable simulators for large-scale systems due to the scalability and modular design of its components. This makes SST the perfect candidate for simulating disaggregated memory systems at scale. One of the current limitations of SST is the lack of a centralized memory management entity that can correctly model page faults and requests for physical frames from the simulated machine. Such a limitation becomes more relevant when there are a large number of shared resources (*e.g.* memory pools). To address this problem and to facilitate research efforts in disaggregated memory systems, a centralized memory management entity is proposed that can be used to investigate allocation policies, page placement, page migration [5], the impact of TLB shutdown [7, 1, 3], and other important aspects that are related to managing disaggregated memory systems. This report describes Opal, a centralized memory management entity, and shows its efficacy using case studies that can leverage the component.

# Chapter 2

## Opal Component

Opal can be thought of as the Operating System (OS) memory manager, and in the case of a disaggregated memory system, the system memory allocator/manager. In conventional systems with single level memory, once a process tries to access a virtual address, a translation is triggered to map the virtual address to a physical address. If a translation is not found, and the hardware realizes that either there is no mapping to that virtual address or the access permissions would be violated, it triggers a page fault that is handled by the OS. The page fault handler maps the virtual page to a physical page that is chosen from a list of free frames (physical pages). Once a physical page is selected, its address is inserted in the page table along with the corresponding access permissions. Later, any accesses to that virtual address will result in a translation process that concludes with obtaining the physical address of the selected page. Since SST aims for fast simulation of HPC systems, it does not model the OS aspects of this sequence of events. However, the memory allocation process will have a major impact on performance for heterogeneous memory systems and disaggregated memory simply because of the many allocation policies that an OS can select from. Moreover, allocation policies are not well understood on disaggregated memory systems, making it important to investigate them to discover the best algorithm or heuristics to be employed for both performance and energy efficiency. To this end, Opal is proposed to facilitate fast investigation and exploration of allocation policies in heterogeneous and disaggregated memory systems.

As shown in Figure 2.1, the Opal component should be connected to the processing elements in SST and the hardware MMU unit. The main reason to be connected to processing elements is to pass allocation hints. For instance, if a process calls `malloc` or `mmap` with hints to whether the physical frames should be allocated from local or remote memory, these hints should be recorded by Opal. While these calls do not immediately allocate physical pages, when a page is mapped, Opal can use the hints to decide where to allocate the physical page. Similarly, the hardware MMU unit should have links to Opal, so once a TLB miss and page table walk conclude with a page fault (unmapped virtual address), Opal will be sent a request for physical frame allocation, which will be eventually mapped to the corresponding faulting virtual address.

Before diving into the details of Opal, it is useful to understand the different ways a disaggregated memory systems can be managed.

- **Exposing External Memory Directly to Local Nodes**

In this approach a local node OS (or Virtual Machine) sees both the local memory and exter-

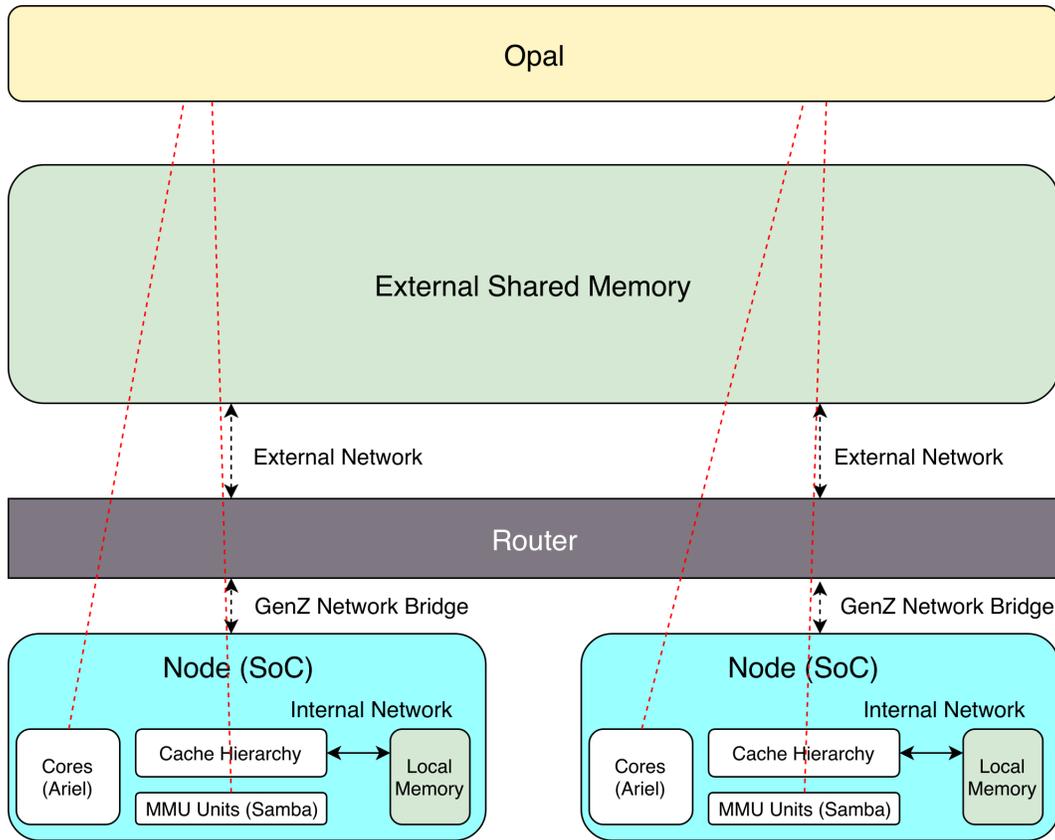


Figure 2.1: A simulated system that uses Opal for centralized memory management.

nal memory. However, it needs to request physical frames from a central memory manager to be able to access external memory. To enforce access permissions and to achieve isolation between data belongs to different nodes/users, the system must provide a mechanism to validate the mappings and the validity of physical addresses being accessed by each node. To better understand the challenges of this scheme, refer to Figure 2.2, which depicts different options to implement access control on shared resources when external memory is directly exposed to local nodes.

In the Figure 2.2, Option ① would be to check if the requesting node is eligible to access the requested address at the memory module level. This implementation requires a bookkeeping mechanism at the memory module level (or in the memory blade) to check the permission of every access. If the request is valid, it is forwarded to the memory device, otherwise either random data is returned or an error packet (access violation) is returned to the requesting core. Since the external memory is shared between nodes, the system memory manager must have a consistent view of allocated pages and their owning nodes. One way to implement requesting external memory is through a device driver (part of local nodes' OS) that can be used to communicate (either through the network or predefined memory regions) with the external memory manager. Option ② is similar but instead of relegating the permission check to the memory module, the router will have a mechanism to check if the accessed physical addresses are granted to the requesting node. Finally, in Option ③, an additional bump-on-the-wire (ASIC or FPGA) can be added by the system integrator to check for the

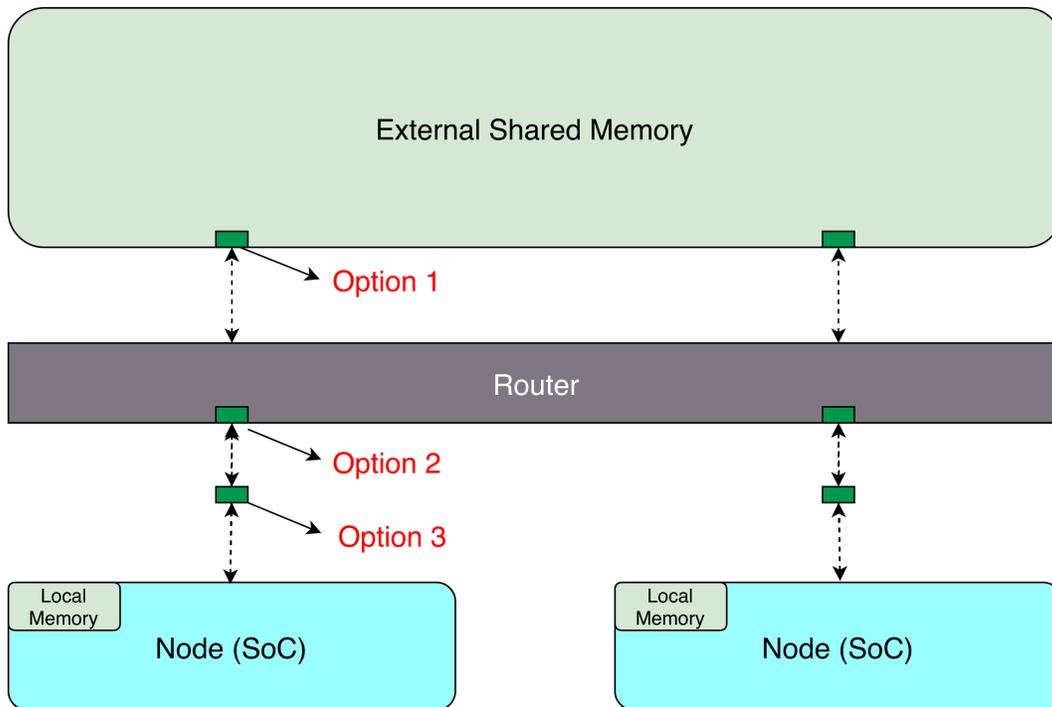


Figure 2.2: Implementing access control for direct access scheme.

permissions of the requests coming out from each node. In all options, nodes will not be able to have direct access to the permission tables; only the system memory manager will have such access. This can be guaranteed by encrypting requests with integrity and freshness verification mechanisms. There are pros and cons of this implementation:

- ✓ Page table walking process is not modified and it is much faster than virtualized environments (4 steps vs. 26 steps).
- ✓ Node-level memory manager optimizations and page migrations are feasible (unlike virtualized environments).
- ✗ The operating system must be patched with a device driver to communicate with external memory manager.
- ✗ The centralized memory manager becomes a bottleneck if not scalable.

### • Virtualizing External Memory

In this approach, each node has the illusion that it owns all the system memory, which means the OS doesn't need to be aware of the current state of the actual system physical memory. Figure 2.3 depicts the virtualized system memory scheme.

As shown in the figure, a system translation unit (STU) must be added to support translation from the *node physical address* to the *system physical address*. The STU can be implemented as an ASIC- or FPGA-based unit that takes a physical address from the node and translates it into the corresponding system physical address. If the address has never been accessed, an on-demand request mechanism is initiated by the STU to request a system physical page. The

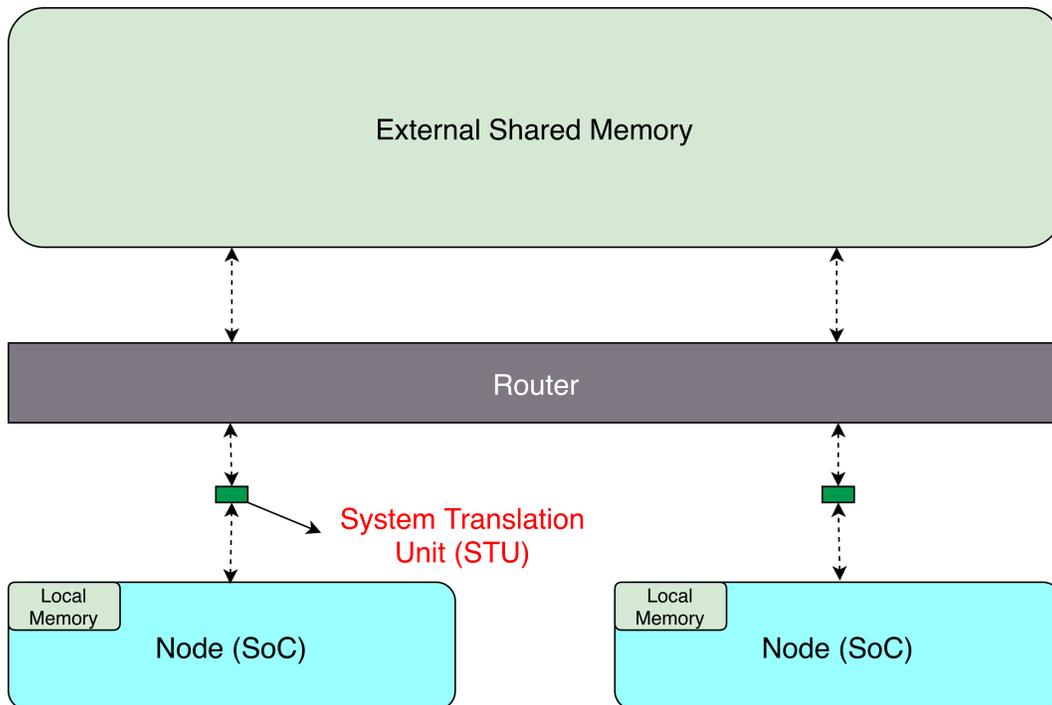


Figure 2.3: Implementing access control for virtualized system memory.

STU might need to do a full system page table walk to obtain the node to system translation. Most importantly, the STU can only be updated through the system memory manager. The advantages and disadvantages of this scheme are:

- ✓ The OS does not need to be changed or patched.
- ✗ In addition to walking the node's page table at the node level, the STU will need to walk the system level page table.
- ✗ There is no guarantee of where the system physical pages that back up the node physical pages exist.

## Integrating Opal in Simulated Systems

As discussed above, Opal must be connected to both a MMU unit (such as SST's Samba) and a Processing Element (such as SST's Ariel). To allow this, any PE core or MMU unit can have a link that connects to their respective ports in Opal – `mmuLink_n` and `coreLink_n`, respectively. For example, port `coreLink_0` can be connected to port `opal_link_0` for Ariel. For Samba, port `mmuLink_0` can be connected to port `ptw_to_opal0`.

Opal expects a minimum of two types of requests to be received through Samba and Ariel links – location hints and allocation requests. Hints originate from processing elements where `mmap` and `malloc` preferences are passed to Opal, which will attempt to satisfy them during on-demand

Table 2.1: SST Modules Used

Module	Description
CPU	Ariel
MMU	Samba [2]
NVM	Messier [4]
Network	Merlin

allocation. This is similar to `libNUMA malloc` hints, which are recorded and used later by the kernel at the time of on-demand paging. Allocation requests come from the page table walker when the accessed virtual page has never been mapped. This resembles minor page faults and on-demand paging on the first access to virtual pages in real systems. Apart from these two requests, Opal also accepts TLB shutdown and shutdown acknowledgment events from Samba units using the Samba to Opal link.

Table 2.2: Opal Parameters

Parameter	Description
<code>clock</code>	frequency of Opal component
<code>max_inst</code>	maximum instructions processed in a cycle.
<code>num_nodes</code>	number of nodes
<code>node_i_cores</code>	number of cores per node
<code>node_i_clock</code>	frequency of each node.
<code>node_i_latency</code>	latency to access Opal component per node
<code>node_i_allocation_policy</code>	memory allocation policy per node
<code>node_i_memory</code>	local memory specific information per node (parameters shown in Table 2.3)
<code>shared_mempools</code>	number of shared memory pools to maintain shared memory
<code>shared_mem.mempool_i</code>	global memory specific information per shared memory pool (parameters shown in Table 2.3)

SST has modular designs for different hardware components. Currently Opal and the disaggregated memory model in SST work with specific modules, shown in Table 2.1. Opal uses Ariel to model CPUs, Samba [2] to simulate memory management units (MMUs), Messier [4] for NVM memory, and Merlin for networking.

Table 2.3: Memory Pool Parameters

Parameter	Description
<code>start</code>	starting address of the memory pool
<code>size</code>	size of the memory pool in KB's
<code>frame_size</code>	size of each frame in memory pool in KB's (equivalent to page size)
<code>mem_tech</code>	memory pool technology (0 : <i>DRAM</i> , 1 : <i>NVM</i> )

## Opal Configuration

Opal should be configured with component-specific, node-specific and shared memory-specific information as shown in Table 2.2. Component-specific information includes clock frequency, maximum instructions per cycle, *etc.*

Node-specific information includes the number of nodes, the number of cores per node, clock frequency, network latency to the Opal component, node memory allocation policy as explained in section 2 and local memory information.

Shared memory-specific information includes the number of shared memory pools and the respective memory pool parameters. Both per-node local memory and per-shared memory pool parameters are related to memory and they are explained separately in Table 2.3. Each of these parameters should be appended with memory related parameters shown in Table 2.2. Table 2.3 describes the memory pool specific parameters. Each memory pool, either shared or local needs a starting address, size of the pool, frame size or page size and memory technology of the pool.

## Example Opal Configuration

---

```
opalParams = {
  "clock"                : "2GHz",
  "max_inst"             : 32,
  "num_nodes"            : 4,
  "shared_mempools"      : 4,
  "node0.cores"          : 8,
  "node0.clock"          : "2GHz",
  "node0.allocation_policy" : 1,
  "node0.latency"        : 2000,      #2us
  "node0.memory.start"   : 0,
  "node0.memory.size"    : 16384,     #in KB
  "node0.memory.frame_size" : 4,
  "node0.memory.mem_tech" : 0,
  "node1.cores"          : 8,
  "node1.clock"          : "2GHz",
  "node1.allocation_policy" : 1,
  "node1.latency"        : 2000,      #2us
  "node1.memory.start"   : 0,
  "node1.memory.size"    : 16384,     #in KB
  "node1.memory.frame_size" : 4,
  "node1.memory.mem_tech" : 0,
  "node2.cores"          : 8,
  "node2.clock"          : "2GHz",
  "node2.allocation_policy" : 1,
  "node2.latency"        : 2000,      #2us
  "node2.memory.start"   : 0,
```

```

"node2.memory.size"           : 16384,      #in KB
"node2.memory.frame_size"     : 4,
"node2.memory.mem_tech"       : 0,
"node3.cores"                 : 8,
"node3.clock"                 : "2GHz",
"node3.allocation_policy"     : 1,
"node3.latency"               : 2000,      #2us
"node3.memory.start"          : 0,
"node3.memory.size"           : 16384,      #in KB
"node3.memory.frame_size"     : 4,
"node3.memory.mem_tech"       : 0,
"shared_mem.mempool0.start"    : 001000000,
"shared_mem.mempool0.size"    : 4194304,    #in KB
"shared_mem.mempool0.frame_size" : 4,
"shared_mem.mempool0.mem_type" : 1,
"shared_mem.mempool1.start"    : 101000000,
"shared_mem.mempool1.size"    : 4194304,    #in KB
"shared_mem.mempool1.frame_size" : 4,
"shared_mem.mempool1.mem_type" : 1,
"shared_mem.mempool2.start"    : 201000000,
"shared_mem.mempool2.size"    : 4194304,    #in KB
"shared_mem.mempool2.frame_size" : 4,
"shared_mem.mempool2.mem_type" : 1,
"shared_mem.mempool3.start"    : 301000000,
"shared_mem.mempool3.size"    : 4194304,    #in KB
"shared_mem.mempool3.frame_size" : 4,
"shared_mem.mempool3.mem_type" : 1,
}

```

---

According to the example configuration, the clock frequency of Opal is  $2GHz$  (“*clock*” : “ $2GHz$ ”). In every cycle Opal can serve up to 32 requests (“*max\_inst*” : 32). The system has 4 nodes (“*num\_nodes*” : 4) with a private memory each and shared global memory is divided into 4 memory pools (“*shared\_mempools*” : 4). Each node has 8 cores (“*node0.cores*” : 8) and clock frequency of  $2GHz$  (“*node0.clock*” : “ $2GHz$ ”). Private memory size is 16MB (“*node0.memory.size*” : 16384) beginning at address 0 (“*node0.memory.start*” : 0). Memory technology of private memories in all the nodes is *DRAM* (“*node0.memory.mem\_tech*” : 0) with a frame size or page size of 4KB (“*node0.memory.frame\_size*” : 4). Network latency to communicate with Opal is 2 micro seconds (“*node3.latency*” : 2000). Total global or shared memory is 16GB, which is divided into 4 memory pools each of 4GB (“*shared\_mem.mempool0.size*” : 4194304). Starting address of shared memory pool 0 is 001000000 (“*shared\_mem.mempool0.start*” : 001000000) which is equivalent to local memory(16MB) + 1, and memory pool 1 starting address is 101000000 (“*shared\_mem.mempool1.start*” : 101000000) which is equal to starting address of shared memory pool 0 + shared memory pool 0 size . Figure 2.4 depicts starting address of each memory pool from which size of each memory pool can be deduced. Each shared memory pool is of *NVM* type (“*shared\_mem.mempool0.mem\_type*” : 1) with 4KB frames (“*shared\_mem.mempool0.frame\_size*” : 4). The memory allocation policy used for all nodes is the alternate memory allocation policy

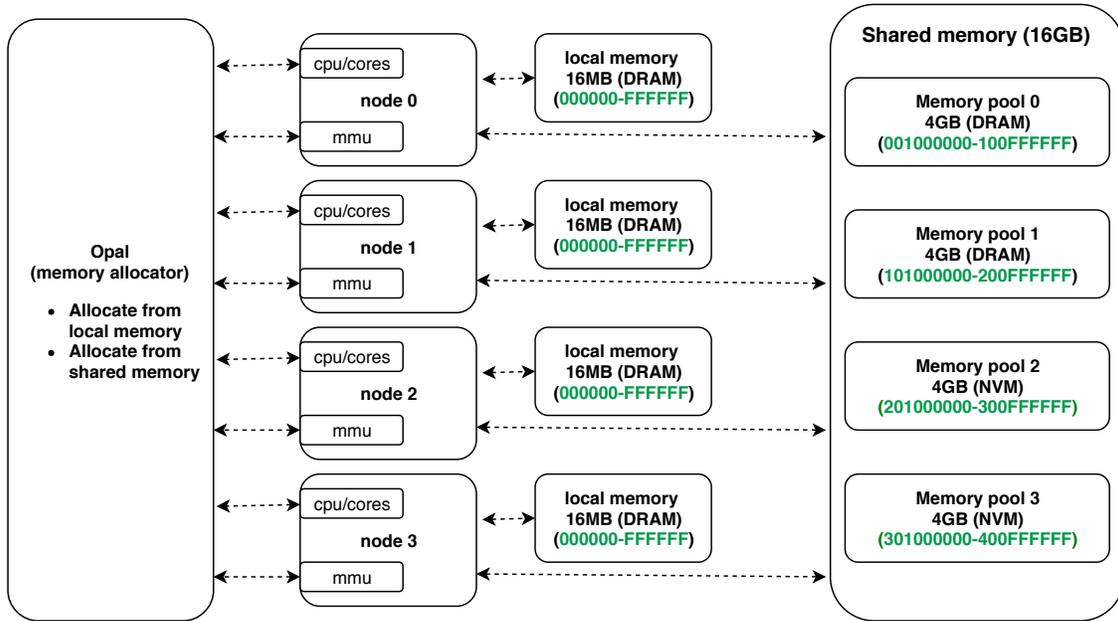


Figure 2.4: Example configuration

("node0.allocation\_policy" : 1), which is explained in Section 2.

## Opal Requests

Several request types are handled and addressed in Opal: hints from the core, page faults from memory management unit, TLB shutdown and shutdown acknowledgement requests.

1. *Hints*: `mmap` and `malloc` requests are used to reserve space in the memory for future use. These requests, are sent to Opal by the core. Opal stores the requests as hints for memory allocation.
2. *Page fault requests*: Page fault requests need to be allocated memory. Allocation of memory from local memory or shared memory is decided based upon the hints provided by the core and the memory allocation policy. For every page fault request, Opal searches for any hints associated with the page. If hints are available, memory is allocated according from the specified memory region. If no hint is found, memory is allocated based on the allocation policies. Memory allocation policies are explained in section 2.
3. *TLB shutdown*: Nodes in disaggregated memory systems can benefit by migrating pages from global memory to local memory. Opal has the capability to migrate pages from local memory to shared memory and vice versa. Whenever pages are migrated, a TLB shutdown is initiated to invalidate the respective pages in all the cores in the nodes.
4. *Shutdown acknowledgement*: The MMU component, which maintains TLB units, sends a shutdown acknowledge event to Opal after invalidating the addresses during a TLB shutdown event.

## Memory Allocation Policies

Various memory allocation policies are implemented in our design and are discussed below.

1. *Local memory first policy*: Local memory is given more priority than shared memory. Local memory is checked first and if there is no spare capacity then shared memory is checked. If shared memory is spread into different memory pools, then the memory pool is chosen randomly from among the available memory pools with enough space. If none of the memory pools have spare capacity, then an error message is thrown. This memory allocation policy can be chosen by setting the *"allocation\_policy"* parameter of a node to 0.
2. *Alternate allocation policy*: Memory allocation alternates between local and shared memory in a round-robin fashion. For example, the first request will be allocated from local memory; the second request from shared memory; the third request from local memory; and so on. If there are multiple shared memory pools then requests alternate between the pools – the first request will be allocated from local memory; the second request from shared memory pool-0; the third request from local memory; the fourth request from shared memory pool-1; the fifth request from local memory; *etc.* This memory allocation policy can be chosen by setting *"allocation\_policy"* parameter of a node to 1.
3. *Round-robin allocation policy*: Similar to the alternate allocation policy, round-robin alternates requests except that it includes all of the available pools in the queue rather than having a nested policy. For example, if there are two shared memory pools then the first request will be allocated from local memory; the second request will be allocated from shared memory pool-0; the third request will be allocated from shared memory pool-1; the fourth from local memory; *etc.* This memory allocation policy can be chosen by setting *"allocation\_policy"* parameter of a node to 2.
4. *Proportional allocation policy*: Memory is allocated proportionate to the fraction of total memory that each memory provides. If local memory size is 2GB and shared memory size is 16GB, then for the 1st memory allocation request, memory is allocated from local memory and then for the next 8 memory requests memory is allocated from shared memory in sequential order. For 10th memory request, memory is allocated from local memory and so on. This memory allocation policy can be chosen by setting *"allocation\_policy"* parameter of a node to 3.

## Communication Between Nodes

Opal also supports communicating between nodes. Nodes can communicate with one another by sending hints with same *fileID* to Opal using Ariel *ariel\_mmap\_mlm* and *ariel\_mlm\_malloc* calls. Opal checks if the received *fileID* is registered with any memory. If it is, then the specific page index is sent to the requesting node. If the *fileID* is not registered with any memory page, then memory is allocated based on the requested size. The allocated memory region is now registered

with the requester *fileID*. Nodes can shared information just by writing information to the specific pages. This reduces costly OpenMPI calls to share information between nodes.

# Chapter 3

## Evaluation

Opal was evaluated by studying the performance of a system with varying number of nodes, amount of shared memory and memory allocation policies. Performance is calculated in terms of instructions per cycle (IPC); IPC of all cores is averaged to get the system IPC.

Table 3.1 describes the configuration of the system used to evaluate the design. We used 2 cores per node, a local memory of 2GB for each node and a shared memory of 16GB. Each core in a node is configured to execute maximum of 100 million instructions. For simplicity we assume that each node executes only one application, XSBench. To increase the size of the load, XSBench is set to have *large* size with 2 threads. It should be noted that for Figures 3.1 and 3.2,  $N$  indicates the number of nodes and  $SM$  indicates number of shared memory pools the shared memory is divided into. For example, N4 with SM2 indicates the disaggregated memory system has 4 nodes and shared memory is divided into 2 shared memory pools. Also, LMF indicates local memory first memory allocation policy, ALT is alternate memory allocation policy, RR is round robin memory allocation policy, and PROP indicates proportional memory allocation policy.

Table 3.1: System Configuration

Parameter	Value
Number of cores in each node	2
maximum instruction count per core	100M
Local memory size	2GB
Shared memory size	16GB
Network Latency	20ns
Application running in each node	XSBench
Application options	-s large -t 2

Contention at shared memory contributes to the performance of disaggregated memory systems. The more the contention at the memory the more will be the delay in getting response from memory. Based on this we explored different memory allocation policies proposed. They are as follows:

1. *Local memory first memory allocation policy*: According to local memory first allocation policy, memory is first allocated from private memory and if it is full, memory is allocated from global memory. XSBench has a memory footprint of approximately 460MB when

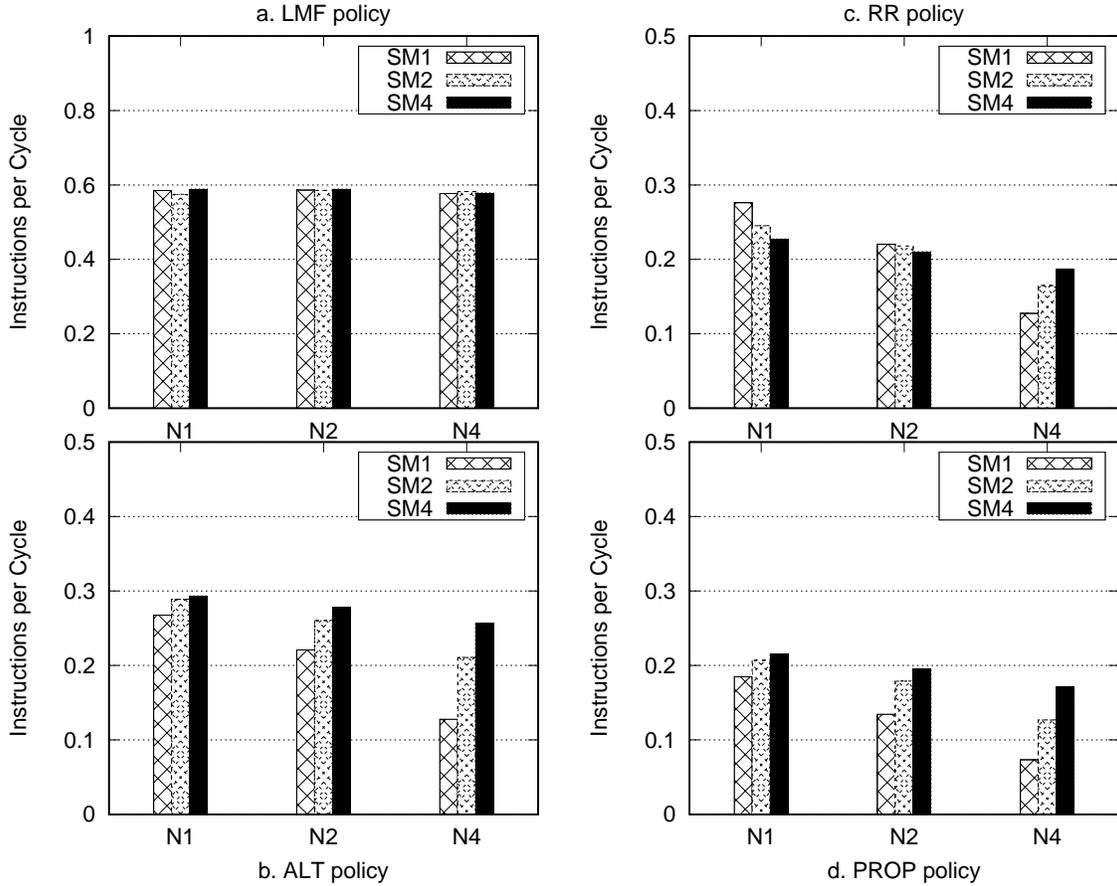


Figure 3.1: Disaggregated memory system performance in instructions per cycle for different memory allocation policies by varying number of nodes and number of shared memory pools

executing 100 million instructions. Accordingly, all the memory can be allocated from the local memory and because each node has its private memory the amount of time to access data from the memory should be same as there is no contention due to other nodes. Our results in Figure 3.1(a) show that the IPC of the system is up to 0.6 with either 1, 2 or 4 nodes as these nodes are not accessing shared memory which is based on the memory requirement of the application and memory allocation policy.

2. *Alternate memory allocation policy:* Every other memory address is allocated from shared memory. So almost half of the memory is from shared memory, that is, among 460MB of memory 230MB is allocated from shared memory. From Figure 3.1(b) it can be seen that the IPC of the nodes with only one node in disaggregated memory system is almost half (0.27) compared to local memory allocation policy. This is because of half of the memory is allocated from shared memory and delay in accessing shared memory is more.

As the number of nodes increases, the system exhibits further slowdowns due to contention at shared memory from multiple nodes. From Figure 3.1(b) we can observe that IPC is 0.27, 0.22 and 0.13 with 1, 2 and 4 nodes in disaggregated memory systems when shared memory is divided into only 1 shared memory. Contention at the memory pool is less when shared

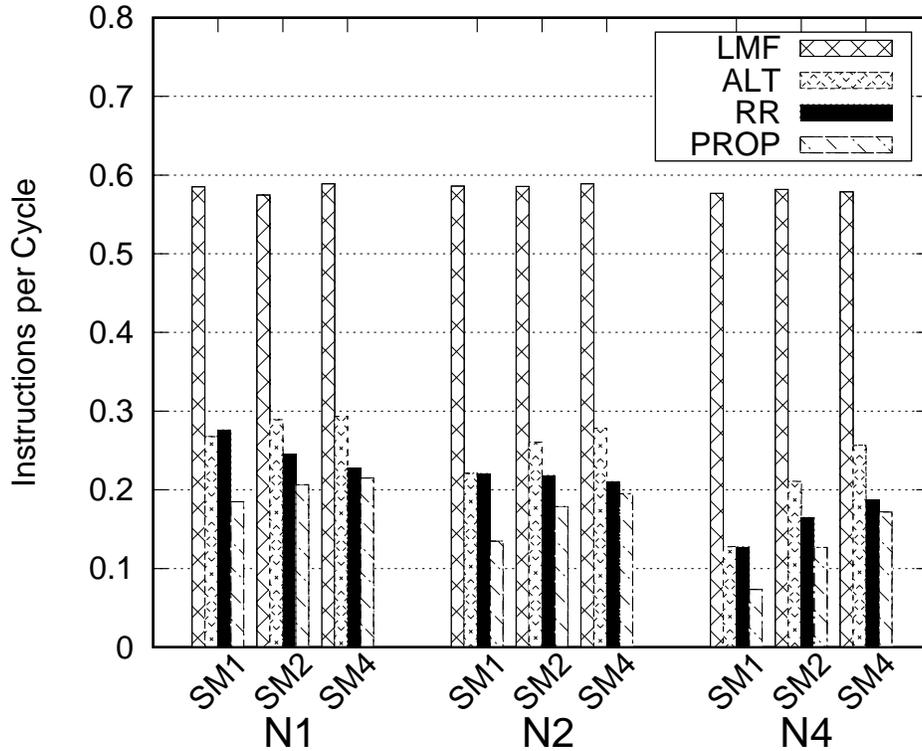


Figure 3.2: Comparing performance of disaggregated memory system by varying different memory allocation policies

memory is maintained in multiple memory pools. Hence, we divided shared memory into 2 and 4 shared memory pools. For instance, from Figure 3.1(b) we can see that if 4 nodes are present in the system, then IPC is 0.13, 0.21 and 0.25 with 1, 2 and 4 shared memory pools respectively.

3. *Round robin memory allocation policy:* In this memory allocation policy, memory addresses are allocated based on the number of memory pools, which includes the local memory pool. So the more shared memory pools, the more total memory that is allocated from shared memory, which degrades system performance. From Figure 3.1(c) it can be observed that when only one node is used with a single shared memory pool, the system has an IPC of 0.27. When the number of memory pools is increased to four, the IPC drops to 0.22. As the number of nodes increases, this effect is not present due to contention at the shared memory from multiple nodes. For instance, from Figure 3.1 it can be observed that if 4 nodes are used, the IPC of the system is around 0.13, when shared memory is maintained in only one shared memory pool, and IPC is around 0.19, when shared memory is divided into 4 shared memory pools.
4. *Proportional memory allocation policy:* In this policy, memory is allocated in proportion with local and shared memory sizes. The local memory size is 2GB and the shared memory size is 16GB, which makes the proportion at which shared memory and local memory is allocated is 8 : 1, that is, for every 9 memory allocations, 8 memory allocations are from shared memory and 1 memory allocation is from local memory. Accordingly, more memory

is allocated from shared memory in comparison with round robin and alternate memory allocation policies. If more memory is allocated from shared memory, the performance decreases. From figure 3.1(d) it can be clearly observed that the performance in proportional memory allocation policy is less compared to the other memory allocation policies. For example, with 4 nodes and with alternate memory allocation policy IPC is 0.13 and with proportional memory allocation policy IPC is 0.07. Like other memory allocation policies, the system IPC decreases as the number of nodes increases. IPC decreases from around 0.2, when 1 nodes is used, to around 0.07, when 4 nodes are used in the system, according to Figure 3.1(d). Also, when shared memory is divided into multiple shared memory pools the IPC of the system increases from 0.07, when shared memory is maintained in one shared memory pool, to 0.18, when shared memory is divided into 4 shared memory pools.

Figure 3.2 compares different memory allocation policies. It can be seen that the performance of the system with the local memory allocation policy is greater when compared to the other memory allocation policies since XSBench does not require more than the total local memory. In a real system, the entire local memory will not be available for one application. Moreover, in a disaggregated memory system, multiple allocation policies are used to allocated memory from both local and shared memory. Alternate memory allocation policy uses less shared memory when there are more available pools, which makes it perform better than RR and PROP. From Figure 3.2, it can be observed that the IPC of the system with alternate memory allocation policy is 0.28, 0.22 with round robin memory allocation policy and 0.20 with proportional memory allocation policy when disaggregated memory system is configured to have only one node and shared memory is divided into 2 shared memory pools. From the same example it should also be noted that the system with performance of proportional memory allocation policy is less compared to all the other memory allocation policies introduced as this policy uses more shared memory compared to other allocation policies.

# Chapter 4

## Conclusion

Opal is a centralized memory manager that can be used to investigate disaggregated memory systems. It can be used to study the effect of page migration policies, page replacement policies, and memory allocation on systems at scale. More generally, it can be used to study dynamic page resizing, dynamic duplication, optimizations for TLB shutdown, acceleration for address translations, and pre-fetching.

This page intentionally left blank.

# References

- [1] Nadav Amit. Optimizing the tlb shutdown algorithm with page access tracking. In *Proc. USENIX Ann. Conf.*, pages 27–39, 2017.
- [2] A. Awad, S.D. Hammond, G.R. Voskuilen, and R.J. Hoekstra. Samba: A Detailed Memory Management Unit (MMU) for the SST Simulation Framework. Technical Report SAND2017-0002, Sandia National Laboratories, Albuquerque, NM, January 2017.
- [3] Amro Awad, Arkaprava Basu, Sergey Blagodurov, Yan Solihin, and Gabriel H Loh. Avoiding tlb shutdowns through self-invalidating tlb entries. In *Parallel Architectures and Compilation Techniques (PACT), 2017 26th International Conference on*, pages 273–287. IEEE, 2017.
- [4] Amro Awad, Gwendolyn Renae Voskuilen, Arun F Rodrigues, Simon David Hammond, Robert J Hoekstra, and Clayton Hughes. Messier: A detailed nvm-based dimm model for the sst simulation framework. Technical report, Technical Report. Sandia National Laboratories (SNL-NM), Albuquerque, NM (United States), 2017.
- [5] Marek Chrobak, Lawrence L Larmore, Nick Reingold, and Jeffery Westbrook. Page migration algorithms using work functions. In *International Symposium on Algorithms and Computation*, pages 406–415. Springer, 1993.
- [6] Arun Rodrigues, Richard Murphy, Peter Kogge, and Keith Underwood. The structural simulation toolkit: A tool for bridging the architectural/microarchitectural evaluation gap. Internal Report SAND2004-6238C, 2004.
- [7] Patricia J. Teller. Translation-lookaside buffer consistency. *Computer*, 23(6):26–36, 1990.
- [8] John R Tramm, Andrew R Siegel, Tanzima Islam, and Martin Schulz. Xsbench-the development and verification of a performance abstraction for monte carlo reactor analysis. *The Role of Reactor Physics toward a Sustainable Future (PHYSOR)*, 2014.

## DISTRIBUTION:

1 MS 1318	Robert J. Hoekstra, 01422
1 MS 1319	Simon D. Hammond, 01422
1 MS 0359	D. Chavez, LDRD Office, 1911
1 MS 0899	Technical Library, 9536 (electronic copy)



