libivy: Distributed Shared Virtual Memory

Suyash Mahar
UC San Diego

Abstract

This paper describes a distributed virtual memory (DSM) based on IVY's fixed manager design. libivy can create an arbitrary-sized distributed shared region, each of which can be accessed by any arbitrary number of pre-specified nodes using a simple JSON-based config. libivy is based on the fixed manager IVY algorithm where one node is the designated manager for the cluster. libivy decides ownership of a page based on who has write-access to it. At any time in the cluster, there can be either one or more readers or exactly one writer. Moreover, it provides a simple shared region allocation API based on C++20 and requires the application to link with the libivy shared object. libivy is designed to automatically set all fault handlers and communication servers/clients for the nodes without any programmer effort. libivy thus acts as a drop-in replacement for multithreaded application that uses shared memory for coordination and processing.

This paper also presents several evaluations of libivy, using a collection of micro-benchmarks and two real workloads: parallel sorting and parallel dot product. Results show that the application's performance improves with diminishing returns as the cluster has more worker nodes.

1 Introduction

Distributed virtual memory allows applications to share part of their address space across networked machines. There exist several different techniques to implement distributed shared memory. Some of the designs are based on the server-client model: IVY [12], synchronization region based: Treadmarks [8, 10], Munin [9], Lazy release consistency [11], and more recently, disaggregated memory: [13, 15].

Other than a distributed shared memory architecture, implementing them requires solving several challenges and making different design choices. These choices include the RPC protocol, memory protection mechanism, programming language, among others. For example, RPC protocol can be based on UDP, TCP, or a higher-level protocol (e.g., HTTP). Similarly, there are currently two different mechanism for protecting a region of memory in Linux, using a combination of mprotect and sigaction or userfaultfd and ioctl. The latter of which is currently not fully implemented in Linux (version 5.11 [1]).

This work makes several tradeoffs that affect performance to keep implementation complexity low. Most important of which is the DSM architecture, libivy uses the fixed-manager model of IVY [12] for its design. Other essential design choices include: (1) Implementing RPC over HTTP. libivy uses GET requests for RPCs. (2) Encoding memory by converting raw byte array to its hex representation before making an RPC.

The contributions of this work are:

1. Implement distributed shared memory for C++ using a shared object.
2. Drop-in replacement for shared memory allocation for multi-threaded workloads.
3. Application transparent handling of page-faults and inter-process/machine communication.

2 Design

To set up a DSM region, the application calls into Ivy to get a shared memory region; Ivy would return a pointer to the region that the application can start using. Ivy on creating the DSM region sets up permission such that any read/write access would call the fault handler. Once the node has access to the page, the page fault handler will be de-registered until some other node requests write access to that page. libivy manages page permission at the granularity of a memory page. In the current implementation, this size is the same as the hardware page size, i.e., 4KiB.

A class-level overview of the implementation is shown in Figure 1.

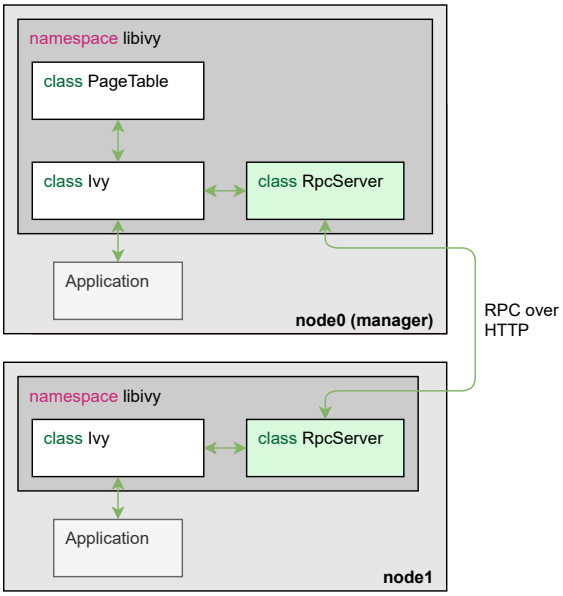


Figure 1: Architectural overview of `libivy`. The implementation separates the actual IVY implementation (**`class Ivy`**) from the communication mechanism (**`class RpcServer`**).

2.1 Manager and worker nodes

Every shared memory process that uses `libivy` has a manager/worker designation. The manager node is specified as part of the `libivy` config; the manager node starts first in the cluster and initially owns all the DSM pages. As different worker nodes get online, they ask the manager for the pages that they need. Each worker node and the manager have a nodeID assigned to them. No two nodes have the same nodeID. The manager node uses these IDs to identify individual nodes for tracking page ownership.

3 Implementation

3.1 RPC

`libivy` implements RPC on top of HTTP using `httplib` [2]. Each node and the manager has its HTTP server that accepts GET requests on different paths that correspond to different functions. Every node uses an HTTP client to communicate with each other's servers.

RPC is implemented using HTTP GET paths with the character '`:`' separating parameters in the request body. All parameters are first encoded to a string which is then transmitted as the body of the GET request.

Algorithm 1: Pseudocode implementing `libivy`'s manager and worker nodes.

Function *Read/Write Fault Handler*:
- mem ← read(manager, addr);
- set_access(addr, read-write);
- memcpy(addr, decode(mem), PAGE_SZ);
- fault_type ← get_fault_type();
- set_access(addr, fault_type);

Function *Read server*:
- **lock_guard** page[addr]
- **if** *I'm the manager* **then**
  - **lock_guard** info[addr];
  - invalidate(info[addr].copyset);
  - info[addr].copyset = ϕ;
  - info[addr].owner ← req_node;
  - mem ← read(owner, addr);
  - **return** encode(mem)
- **else**
  - set_access(addr, read);
  - **return** mem[addr : addr + PAGE_SZ];
- **end**

Function *Write server*:
- **lock_guard** page[addr]
- **if** *I'm the manager* **then**
  - **lock_guard** info[addr];
  - info[addr].copyset = info[addr].copyset ∪ req_node;
  - owner ← info[addr].owner;
  - mem ← read(owner, addr);
  - **return** encode(mem)
- **else**
  - set_access(addr, none);
  - **return** mem[addr : addr + PAGE_SZ];
- **end**

Function *Invalidate*:
- **lock_guard** page[addr];
- set_access(addr, none);
- **return** OK;

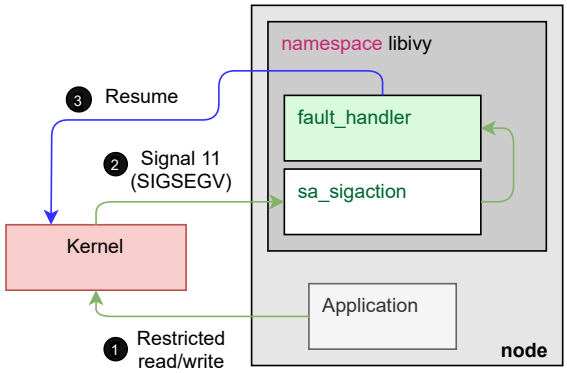


Figure 2: Page faults and permissions in `libivy`. On a page fault, e.g., on a restricted access from the application (❶), the kernel would invoke the handler registered by the application (❷). Next the application processes the page fault (e.g., by fetching the page from the manager) and returns the control to the kernel (❸).

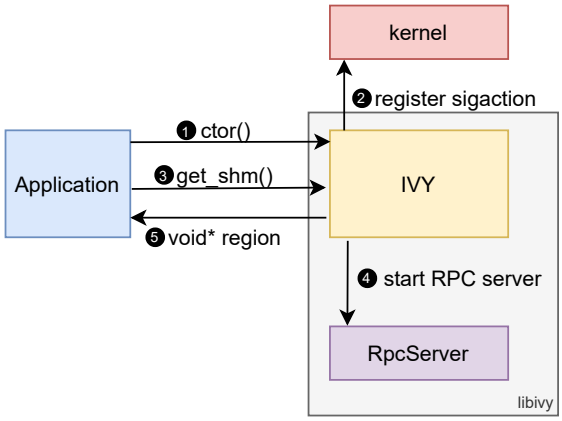


Figure 3: Lifecycle of an application using `libivy`.

List of GET paths implemented for each node:

1. `"get_rd_page_from_manager"`:
   Used by worker nodes to request read access to a page from the manager.

2. `"get_wr_page_from_manager"`:
   Used by worker nodes to request write access to a page from the manager.

3. `"fetch_page"`:
   Used by manager to get contents of a memory page and set new permissions.

4. `"invalidate_pg"`:
   Used by manager to notify the nodes to restrict all accesses to a memory page.

Each GET path on receiving a request, converts the request body to function parameters and calls the appropriate function in `libivy`.

3.2 Memory accesses and page faults

`libivy` manages access to memory using a combination of `mprotect`[3] and `sigaction`[7] system-calls. When `libivy` object is first constructed, it installs a `SIGSEGV` handler that will be invoked by the kernel if the application has a segmentation fault.

To change permission of a memory page, `libivy` calls `mprotect` on an address with appropriate permission. Next, if the application's access falls outside of the specified permission, the kernel will invoke the `libivy` installed `SIGSEGV` handler. The handler will read the address of the fault and the type of fault (read or write fault) and would call the appropriate `libivy` function to service the fault (e.g., ask manager for the page).

Once the read/write fault is serviced (e.g., by fetching a page from some node), `libivy` would set the correct permission on the page, i.e., restrict to read if the page is read only and resume the application.

3.3 Server

The server is internally separated into three classes: `Ivy` (Implements `libivy` API and internal functions), `RpcServer` (Manages RPC server and client), and `IvyPageTable` (Holds locks and access rights for shared memory pages).

3.4 `libivy`'s Lifecycle

Figure 3 describes the lifecycle of `libivy` before it starts to handle page faults for the application. Lifecycle of an application relevant to `libivy` starts with the creation of an `Ivy` object (❶). The application passes in the path to a config file that describes different parameters for `libivy` and the current node's ID. Within the constructor (❷), Ivy registers the page fault handlers with the kernel and returns to the application. Next, the application calls the function `Ivy::get_shm()` to get a pointer to the shared memory region (❸). During this call, `Ivy` initializes the RPC server (❹), calls `mmap` to create an anonymous mapping for the shared region and returns to the application with the pointer to the shared region (❺). All accesses to the shared region at this point are handled as page faults by `libivy`.

3.5 Locks in `libivy`

`libivy` protects all internal functions from data-races and deadlocks using locks for possible concurrent accesses from different `Ivy` threads (e.g., invalidate request while fetching a page from the manager). To do this, when entering a procedure that changes the configuration of the distributed shared memory, `libivy` acquires a per-page lock. If the procedure performs a long blocking operation (e.g., trying to communicate with the manager), the node releases the lock and backs off for a specified period before re-trying the operation. Thus implementation prevents deadlock when two procedures contend for a lock but depend on each other's lock.

`libivy`'s locks are implemented using C++'s `std::mutex` and are stored in an unordered map.

4 Software Artifact and API

`libivy` is implemented as a C++20 shared object. To use `libivy`, user would have to include `libivy.hh` header file and link with `libivy.so`.

To use `libivy`, user creates a new object of the class `Ivy` with path to a config file and a node Id. `libivy` then reads from the config file each node's address, shared region's size, manager's node ID and the base address to mount the shared memory at. Example config file is shown in Listing 2.

`libivy` is available with all dependencies included at https://github.com/suyashmahar/libivy.

4.1 API

`libivy` offers the following API to the applications. Current `libivy` only supports C++20:

- `Ivy(std::string cfg_f, idx_t id)`
  Constructs an object of the type `Ivy`. The constructor takes as input the path to a JSON config file and the current node's id. To keep `libivy`'s complexity low, current implementation restricts each process to one Ivy object. This prevents one `Ivy` object to unregister signal handlers for any existing `Ivy` objects.

- `res_t<void_ptr> Ivy::get_shm()`
  Returns a pointer to the allocated shared memory region on success, otherwise returns an error message.

- `mres_t Ivy::drop_shm()`
  Unmounts the shared memory region and flushes its content.

- `res_t<bool> Ivy::is_manager()`
  Returns true if this node is the manager node for the cluster.

4.2 Dependencies

`libivy` depends on the following packages to implement different functionalities:

- `nlohmann::json` [4]: For parsing JSON config files.

- `httplib` [2]: To implement HTTP server for RPC.

4.3 Using `libivy`

Ivy provides a simple API to allocate a shared memory region and share them among arbitrary number of predefined nodes. Listing 1 shows a simple example that `libivy` to allocate a memory region and then write to it.

Steps to use `libivy`

1. The application would first create an `Ivy` object with the cluster config that it wants to join and the id of the current node.

2. Next the application asks `libivy` for a shared memory region using the `get_shm()` function. This function uses `mmap` to allocate the specified region of memory, sets access to `PROT_NONE` and returns a pointer to the application.

3. The `get_shm()` function can return error if the node experiences an internal failure. The application might check for the error message and retry.

4. Finally, the application starts working on the shared memory. `libivy` does not provide and synchronization guarantee against data races so the application would need to coordinate accesses among its nodes.

5. At the end, when the application has completed working on the shared memory, it will call `drop_shm()` to unmount the shared memory region and free the memory pages from the node's local memory.

```
// 1. Create an object of the class
//    `IVY`with the config file and
//    node ID
Ivy ivy("path/to/config", node_id);

// 2. Automatically sets up the shared
//    memory and all the handlers
auto [shm, err] = ivy.get_shm();

// 3. Handle any error
if (err.has_value())
  ERROR(err.value());

auto arr_ul
  = reinterpret_cast<uint64_t*>(*shm);

// 4. Work on the shared region
invoke_worker(arr_ul, node_id);

// 5. Unmount the shared memory
ivy.drop_shm();
```

Listing 1: Example using `libivy` API.

```
{
    "nodes": [
        "localhost:2000",
        "localhost:2001",
        "localhost:2002"
    ],
    "manager_id": 0,
    "region_sz": 268435456,
    "base_addr": "0x10000000000"
}
```

Listing 2: Example JSON configuration file for `libivy`. `nodes` specifies the list of all nodes that are part of the cluster. `manager_id` is the id of the manager node. `region_sz` is the size of DSM that each node allocates on call to `Ivy::get_shm()`. `base_addr` is used to mount the shared region at the same location in all node's virtual address space, allowing applications to use lean pointers internally.

Table 1: Description of the evaluation platform used.

CPU	Intel(R) Xeon(R) CPU E5-2676
Cores per node	1
Memory per node	1 GiB, 4KiB page size
OS	Ubuntu 20.04
Env.	g++-10, httplib v0.8.9, nlohmann::json v3.9.1

Table 2: Micro-benchmarks used for testing `libivy`.

Microbenchmark	Description
Local Page fault latency §5.2.1	Measures the time it takes to lock and unlock a page locally using `mprotect`.
RPC Page fault latency §5.2.2	Measures time to service a fault (SIGSEGV) using `sigaction`.
DSM Trashing §5.2.3	Uses multiple nodes to contend to write to the same shared memory location.

5 Evaluation

5.1 Evaluation Platform

All experiments were run on AWS EC2 instances, each of which had the specification listed in Table 1. All the nodes were always in the same region and used internal IP addresses to communicate among themselves.

5.2 Micro-benchmarks

To get a better understanding of the implementation, I ran three micro-benchmarks that tested different aspect of the system and implementation before running the actual workloads. These micro-benchmarks are summarized in Table 2.

5.2.1 Local Page Fault Latency

To estimate the overhead of handling a page fault using combination of `mprotect` and `sigaction` handler, I used a micro-benchmark that measured the latency of locking and unlocking a single page using `mprotect` (t_{locking}). Next, the micro-benchmark installs a simple fault handler, and measures the latency of locking a page, writing a single byte to it and unlocking the page in the handler ($t_{\text{fault-on-locked}}$). Now that the micro-benchmark has latency of both locking/unlocking a page and servicing a page fault on locked page, it can calculate the latency of a single page fault ($t_{\text{fault-on-locked}} - t_{\text{locking}}$).

Table 3: Local page fault latency. Cost of handling page fault using `sigaction` and `mprotect`.

Operation	Avg. Latency
`mprotect` lock + unlock	1.052 µs
Handling faults using `sigaction`	3.431 µs

Table 4: Page fault latency for read and write operations using `libivy`.

Fault type	Avg. Latency
Read	1.055 s
Write	1.044 s

The micro-benchmark repeated each operation 2 million times and calculated the average time it takes to service a page fault. Results in Table 3 shows that the average latency of servicing a page fault is in the order of microseconds.

5.2.2 RPC Page Fault Latency

Next to calculate the latency of servicing a page fault over RPC, I used one `libivy` node to fault page from the other and measure the time it takes to perform this operation. With two nodes, node_0 and node_1. node_0 is initialized with a single page in `libivy`'s shared memory region. Next, node_1 would first read the page and then write a single 64 bit integer to the first 8 bytes of the page. Since the pages are transmitted in 4kb chunks, both read and write latency are representative of the corresponding operations.

This experiment was repeated for 10 iterations where the two nodes were located on separate EC2 virtual machines and the average latency is reported in Table 4. The page fault latency for both read and write faults is in the order of seconds.

The page fault latency as measured in this experiment is significantly higher than the local page fault latency (i.e., `mprotect+sigaction` latency). I expect the reason for this to be the additional delay from HTTP server logic and encoding and decoding the memory page.

5.2.3 Shared Memory Trashing

The second workload that I used to microbenchmark the performance of `libivy` is a trashing workload where two nodes alternatively writes to the same address. Each node increments the counter in a round-robin fashion. Every node waits for the $((n-1)\%\text{total nodes})^{\text{th}}$ node to increment the counter, after which the n^{th} node increments the counter. The psuedo code for the trashing workload is shown in Listing 2

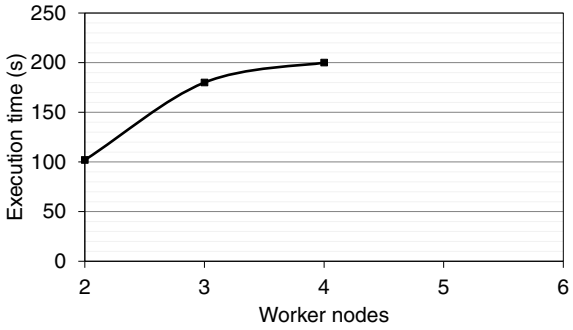


Figure 4: Increase in the time it takes for each node to increment the counter to 50 with increasing number of contending nodes. The microbenchmark timed-out with 5 and 6 worker nodes.

Figure 4 shows the time it takes for each worker to reach the counter value of 50. As the number of workers increase, contention increases resulting in longer execution time. For more than 4 worker nodes, all the nodes trash each other enough that the micro-benchmark never completes. This is because the manager receives conflicting requests for the same address (e.g., invalidate and fetch), so it retries both of them after some delay (Set to 1s in the implementation).

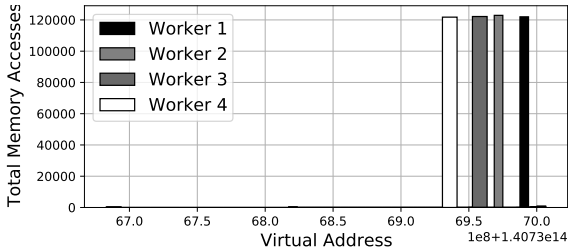
Algorithm 2: Psuedocode for trashing micro-benchmark.

```
Function worker_node(id: size_t):
    while true do
        // Busy waiting for other nodes
        while (counter % total_nodes) ≠ id-1 do
            continue;
        end
        counter ← counter + 1
    end
```

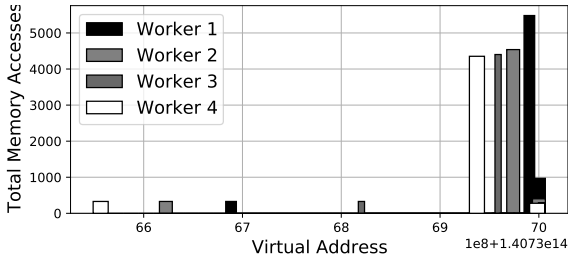
5.3 Primary Workloads

Next, I used two more complex workloads to measure the performance and scaling of libivy. The two workloads are P-Sort [6] and P-Dot [5].

P-Sort workload sorts a list of 64bit integers and writes the result to `stdout`. Each instance of the workload processes $1/n^{\text{th}}$ part of the input list. Since the i^{th} worker



(a) Memory access pattern of P-Sort with 4 Worker threads and input list with 1024 elements.



(b) Memory access pattern of P-Dot with 4 Worker threads and input vectors with 512 elements each.

Figure 5: Distirbution of memory accesses across worker threads for P-Sort and P-Dot. The memory accesses show that majority of processing done by the worker threads are on non-overlapping regions.

works on the i^{th} part of the input, they don't do not access overlapping regions of memory. Once all worker processes have sorted their share of the list, the manager process merges the result and writes the output to `stdout`.

P-Dot workload works in a very similar way to P-Sort and computes the dot product of two vectors using n worker threads. The vectors are laid out in memory one after other and the output is written to `stdout`.

5.4 Workload Behavior

I ran each of the workload with Intel Pin [14] "memtrace" pintool and collected the per thread memory accesses to study their memory access pattern.

Figure 5 shows how the accesses of each worker thread is spread across the virtual address space. Each of the worker performs majority of its IO to non-overlapping region, except for small synchronizing reads and writes.

5.5 Scaling

For scaling experiments, I ran both psort and pdot with increasing number of nodes, keeping the input unchanged.

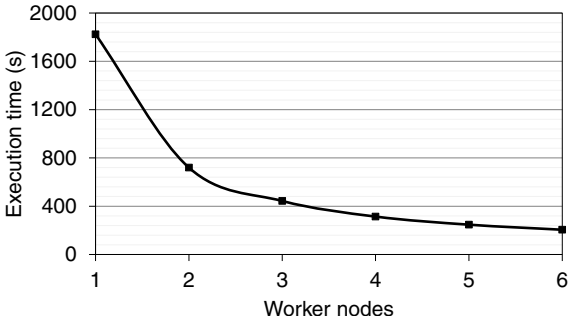


Figure 6: Change in average node execution time for P-Sort with increasing number of worker nodes. Each node in a n worker cluster does $1/n$ work.

For psort, the input was of list with 262144 elements. Each worker node in a cluster of n worker nodes sorted the n^{th} part of list containing $262144/n$ elements. For pdot, the inputs are two 1-D vectors, each of size 32768 elems. Each worker node in a cluster of n nodes calculated the dot product of two $32768/n$ elements vectors.

5.5.1 P-Sort

To explore performance scaling, P-Sort was run with varying number of worker nodes. In each configuration, the list is divided among n workers. Figure 6 shows the average time it takes for each worker to sort its part of the list. The execution time of each worker represents the time it takes to sort the list and any page fault latency and corresponding contention at the manager.

Figure 6 shows a decrease in time it takes to process the list with diminishing returns suggesting that the time it takes to sort the list and communicate with the manager start to dominate as number of nodes increase.

5.5.2 P-Dot

Similar to P-Sort, P-Dot was run on varying number of worker nodes. With n worker nodes, $1/$n of the input vectors A and B were processed by each worker node. Figure 7 shows the average time it takes for each worker to compute the dot product of its part of the vector. Very similar to P-Sort, P-Dots performance improves with increase in number of worker nodes, but with diminishing returns.

6 Conclusion

`libivy` provides a simple to use distributed shared memory implementation in C++. It is based on the design of IVY's fixed manager model and implements it using

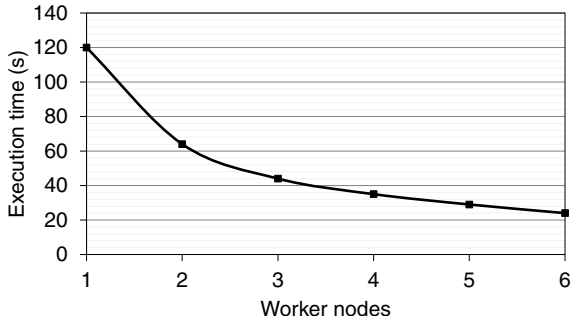


Figure 7: P-Dot execution time with increasing number of nodes.

`mprotect+sigaction` and RPC over HTTP. Evaluation shows that while RPC page fault is expensive, overall the application speeds up with increasing number of node. Possible future work would involve looking into more efficient ways to implement RPC that can speedup inter-node communication and page-fault latency.

References

[1] *DESCRIPTION ⟩⟩ UFFDIO_API ⟩⟩ UFFDIO_REGISTER, ioctl_userfaultfd(2) Linux User's Manual*.

[2] httplib – a c++ header-only http/https server and client library. https://github.com/yhirose/cpp-httplib. Accessed: 2021-05-29.

[3] *mprotect(2) Linux User's Manual*.

[4] nlohmann/json – json for modern c++. https://github.com/nlohmann/json/. Accessed: 2021-05-20.

[5] pdot parallel vector dot product. https://github.com/suyashmahar/IVY-dist-mem/tree/master/src/workloads/pdot. Accessed: 2021-05-14.

[6] psort parallel sort. https://github.com/suyashmahar/IVY-dist-mem/tree/master/src/workloads/psort. Accessed: 2021-05-14.

[7] *sigaction(2) Linux User's Manual*.

[8] AMZA, C., COX, A. L., DWARKADAS, S., KELEHER, P., LU, H., RAJAMONY, R., YU, W., AND ZWAENEPOEL, W. Treadmarks: Shared memory computing on networks of workstations. *Computer 29*, 2 (1996), 18–28.

[9] CARTER, J. B., BENNETT, J. K., AND ZWAENEPOEL, W. Implementation and performance of munin. *ACM SIGOPS Operating Systems Review 25*, 5 (1991), 152–164.

[10] COX, A., DWARKADAS, S., KELEHER, P., AND ZWAENEPOEL, W. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *Proceedings of the Winter 94 Usenix Conference* (1994), no. CONF.

[11] KELEHER, P., COX, A. L., AND ZWAENEPOEL, W. Lazy release consistency for software distributed shared memory. *ACM SIGARCH Computer Architecture News 20*, 2 (1992), 13–21.

[12] LI, K., AND HUDAK, P. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems (TOCS) 7*, 4 (1989), 321–359.

[13] LIM, K., CHANG, J., MUDGE, T., RANGANATHAN, P., REINHARDT, S. K., AND WENISCH, T. F. Disaggregated memory for expansion and sharing in blade servers. *ACM SIGARCH computer architecture news 37*, 3 (2009), 267–278.

[14] LUK, C.-K., COHN, R., MUTH, R., PATIL, H., KLAUSER, A., LOWNEY, G., WALLACE, S., REDDI, V. J., AND HAZELWOOD, K. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2005), PLDI '05, Association for Computing Machinery, p. 190–200.

[15] TSAI, S.-Y., SHAN, Y., AND ZHANG, Y. Disaggregating persistent memory and controlling them remotely: An exploration of passive disaggregated key-value stores. In *2020 {USENIX} Annual Technical Conference ({USENIX}{ATC} 20)* (2020), pp. 33–48.