Memphis on an XT5: Pinpointing Memory Performance Problems on Cray Platforms

Collin McCurdy, Jeffrey Vetter, Patrick H. Worley and Don Maxwell
Oak Ridge National Laboratory

ABSTRACT: *Memphis is a tool that makes use of Instruction Based Sampling (IBS) hardware counters, available in recent AMD processors, to help pinpoint the sources of memory system performance problems. This paper describes our experiences porting Memphis to a test XT5 system at ORNL, including modifications required by Compute Node Linux to the kernel module that interfaces with IBS, and low impact modifications to the batch queue that enable the module's use at runtime. We also include a case study demonstrating the use of Memphis in an iterative process of finding problems and evaluating fixes in the CICE component of the production CESM climate code*

KEYWORDS: Memory system performance, Intra-node performance, NUMA

1. Introduction

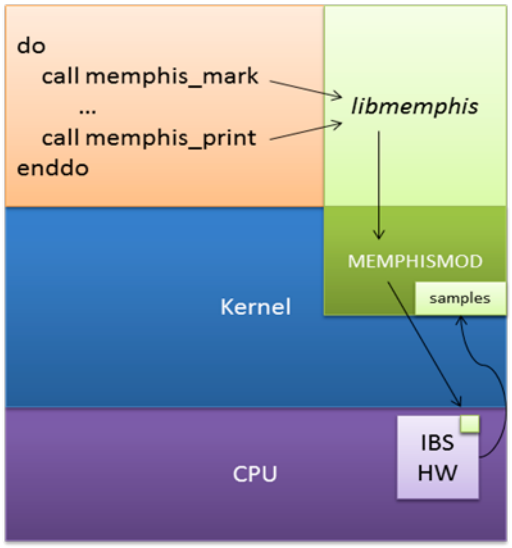
Current forecasts call for each chip in an Exascale system to consist of hundreds to thousands of processing cores. Already, with only on the order of 10 cores per chip, memory limitations and performance considerations are forcing scientific application teams to exploit the node-level single address-space offered by most current large-scale systems through the use of multi-threading programming models.

At the same time, however, trends in micro-processor design are pushing performance problems associated with Non-Uniform Memory Access (NUMA) to ever-smaller scales. Typical performance problems associated with NUMA include hot-spotting and computation/data-partition mismatches. Additionally, NUMA can amplify existing potential problems and turn them into significant real problems. For example, in the presence of contention for locks and other shared variables, NUMA can significantly increase waiting-time, increasing the probability of further contention.

Memphis [1] is a toolset that uses new sampling-based hardware performance monitoring extensions to pinpoint memory performance problems at their source. The toolset takes a data-centric approach: while other sampling-based tools associate information with instructions, *Memphis* associates information with program variables. A key insight behind *Memphis* is that the source of memory performance problem may well not be where the problem is evidenced. For example, while the cause of a hot spot is likely variable initialization, the problem is only evident at the variable's use.

Until recently, a limitation of *Memphis* has been that, because it requires installation of a kernel module, we have only been able to run on very small scale systems. This limitation restricted its use to small-scale runs, potentially unrepresentative in terms of both number of cores and data set size. We have now addressed that problem by porting *Memphis* to *Chester*, a test Cray XT5 system at the Oak Ridge Leadership Computing Facility (OLCF) centered at Oak Ridge National Laboratory.

After a brief description of the components of *Memphis*, this paper details how we got the components running on the XT5, and describes a policy that makes them available to selected users. We then provide a case study demonstrating the use of *Memphis* in an iterative process of finding problems and evaluating fixes in the CICE component of the production CESM climate code.



(a)

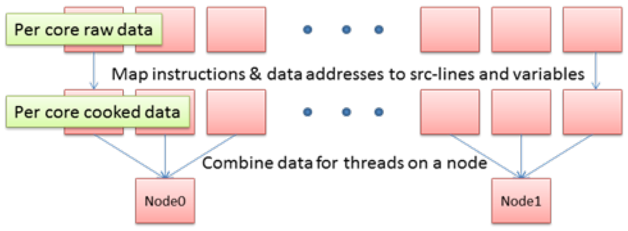

Per core raw data
Map instructions & data addresses to src-lines and variables
Per core cooked data
Combine data for threads on a node
Node0
Node1
(b)

Figure 1: *Memphis* runtime components (a), and post-processing executable (b).

2. IBS and *Memphis*

2.1 Instruction Based Sampling

Available in AMD processor implementations since Barcelona, Instruction Based Sampling (IBS) [2] hardware provides performance monitoring extensions similar in concept to those offered by *ProfileMe* hardware in the DEC Alpha 21264 [3]. More recent Intel chips have extended the performance monitoring unit with roughly the same functionality (*PEBS-LoadLatency*) [4].

Like event-based sampling, IBS is interrupt-driven. However, while event-based interrupts are caused by counter overflows, IBS hardware instead *periodically* interrupts execution, and then proceeds to follow the next instruction through pipeline, keeping track of micro-architectural events that occur due to the execution of that instruction. Upon retirement of the followed instruction, the hardware calls a software interrupt handler, providing the following data useful for finding memory performance problems:

- Precise instruction program counter.
- Virtual address of data referenced by instruction.
- The source of the data: i.e., DRAM, another core's cache.
- Whether the source was on the local node or a remote node.

Data sourcing information indicates potential problems, while instruction and data addresses enable precise attribution to code and variables respectively.

2.2 Memphis Architecture

Memphis enables user-level interaction with IBS hardware via three components, which together facilitate: the specification of regions for sampling, the gathering of sample data, and the aggregation of that sample data into a form useful for finding performance problems. Figure 1 depicts the components, and their interactions:

- *MEMPHISMOD*, a kernel module that interacts with IBS hardware, including: setting it up to gather samples per-thread, turning sampling on and off, and specifying the interrupt handler. The module interfaces with user-level software through a device file in /dev.
- *Libmemphis*, a user-level library whose API enables users to set 'calipers' around interesting regions of code (for instance, the time-step loop) and gather samples into per-thread data files.
- *Memphis-tool*, a post-processing executable that aids in searching for patterns in the sample data files by aggregating: 1) samples originating from the same data and instructions, and 2) samples originating from threads on the same NUMA node.

3. *Memphis* on Cray Platforms

Because Compute Node Linux (CNL), the operating system running on the compute nodes of Cray XT systems, is Linux-based, many components of *Memphis* work on Cray platforms without modification. In

```
chester-login1# mdiag -r memphis.8
Diagnosing Reservations
RsvID                     Type Par   StartTime     EndTime     Duration Node Task Proc
-----                     ---- ---   ---------     -------     -------- ---- ---- ----
memphis.8                 User che   -00:07:30    INFINITY     INFINITY    1    1   12
    Flags: ISACTIVE
    ACL:   RSV==memphis.8= USER==cmccurdy+
    CL:    RSV==memphis.8
    Accounting Creds:  User:root
    Task Resources: PROCS: [ALL]
    SubType: Other
    Attributes (HostExp='^20$')
    Active PH: 0.00/1.51 (0.00%)
    History: 1301595949:PROCS=12
```

Figure 2: Moab reservation on Chester, a test XT5 system at OLCF.

particular, the library and the post-processing executable do not require changes.

The one component that required modification in our initial port was the kernel module. The port of the kernel module was complicated by the black-box nature of CNL. Compilation of a Linux kernel module requires access to kernel source code. While CNL is based on Linux, the code is proprietary and is not available to non-Cray system administrators at the OLCF. Therefore compiling the Memphis kernel module required the help of a patient Cray engineer to perform the first half of each iteration of the compile-install-test-modify loop.

After the kernel module was compiled and running on the system, we then required a mechanism for making it available to jobs that want to use *Memphis*.

3.1 Porting the Kernel Module to CNL

The initial port required two changes to the module. As it turns out, the first was simply the result of porting to a kernel version on which *Memphis* had not run before. While the user-level interface to the kernel is very stable between kernel releases, intra-kernel interfaces tend to fluctuate substantially. Therefore a new kernel version often requires modifications to a kernel module.

In this instance, the kernel used by CNL was older than the kernel for which the kernel module we had borrowed the code that accesses IBS counters had been created. In between the two kernel versions, the mechanism for registering interrupt handlers had been modified. By looking at other drivers with similar functionality we determined that the kernel version used by CNL required *set_nmi_callback* rather than *register_die_notifier*.

The second change was due to an actual difference between CNL and standard Linux and required some understanding of the interface to IBS performance monitoring hardware.

IBS hardware is located in the Northbridge portion of AMD processors. Northbridge configuration registers are accessed and set via *PCI extended configuration space*. The original kernel module code thus uses *pci_get_device* to determine the addresses of Northbridge devices in the node, using predefined kernel constants to name the devices. Because the files that defined this function and constants were not contained in the CNL distribution, we were left with no option but to determine the values we required (using the *lspci* command) and then hard-code these values into calls that set configuration registers.

3.1.1 *Current Status*

After a recent system software upgrade, we found that the *Memphis* kernel module for the standard Linux kernel version used by the new system, worked without further modification.

3.2 Runtime Policy and Configuration

To maximize the availability of *Memphis* for selected users, while minimizing impact of a bleeding-edge kernel module on other users, we arrived at the following policy: the kernel module is always available on a single, dedicated node of the system. Users that want to access *Memphis* have a 'reservation' on that node, such that it is always the first node of their allocation. While this means that only that first node provides sample data for subsequent post-processing, we have found that this is sufficient for our needs, since intra-node performance is typically uniform across nodes.

The policy is implemented on *Chester*, a test XT5 system at OLCF, as follows. On system reboots the kernel module is installed on the dedicated node and a device entry – the library's interface to the module – is created in */dev*. The reservation portion of the policy is implemented through a *standing reservation* in Moab, the workload manager/scheduler on OLCF systems [5].

Figure 2 describes the policy currently in place on *Chester*. In this instance, user *cmccurdy* has a reservation on 12 processors of Node 1. The node allocation of any job run on this user's behalf will start with Node 1 and add nodes as required to fulfil the requested node count.

```
NODE: 0  total: 6591
000) [heap]:tx  [ 0x2a5b1588 - 0x2b017870 ]  1719
  ice_boundary.F90:4106:0x9d4834    [ 0x2a5c1468 - 0x2b017788 ]  1414
  ice_boundary.F90:4106:0x9d4830    [ 0x2a5b1588 - 0x2b017870 ]  279
  ...
001) [heap]:ty  [ 0x2b022808 - 0x2ba83518 ]  1643
  ice_boundary.F90:4106:0x9d4834    [ 0x2b02d190 - 0x2ba83190 ]  1361
  ice_boundary.F90:4106:0x9d4830    [ 0x2b02d8b0 - 0x2ba83518 ]  251
  ...
002) [heap]:tc  [ 0x29b4b158 - 0x2a5abee8 ]  1611
  ice_boundary.F90:4106:0x9d4834    [ 0x29b53d28 - 0x2a5abee8 ]  1377
  ice_boundary.F90:4106:0x9d4830    [ 0x29b4b158 - 0x2a5aae18 ]  205
  ...
003) [heap]:_ice_state_2_  [ 0x172a8dc0 - 0x180b0088 ]  1582
  ice_boundary.F90:4106:0x9d4834    [ 0x176bb2d8 - 0x17e35f48 ]  914
  ice_boundary.F90:2727:0x9cfa64    [ 0x174b1030 - 0x18044610 ]  482
  ice_boundary.F90:4106:0x9d4830    [ 0x176ba888 - 0x17e35930 ]  148
  ...

NODE: 1  total: 506
000) [heap]:<not-found>  [ 0x24b94140 - 0x2c9cdb10 ]  69
  ice_history.F90:2564:0xa4585c    [ 0x29192040 - 0x29b40048 ]  66
  ...
```

Figure 3: CICE initial results: remote DRAM references.

It is not difficult to imagine an alternative, queue-based, policy in which a batch queue would be dedicated to jobs wishing to use *Memphis*. Some number of compute nodes would have the kernel module installed, and one of those nodes would be required to be the initial node in the allocation of any job submitted to the *Memphis* queue.

4. Case Study: CICE

CICE [6] is the sea ice modeling component of the Community Earth System Model (CESM) [7] climate modeling code. In recent large-scale CESM runs on the *Jaguarpf* system at ORNL, we noticed that the ice component was not scaling as well as other components. Though its runtime is not a large fraction of the overall runtime, it is on the critical path of the large atmosphere component and therefore its scalability is critical to overall scalability. We therefore wished to use *Memphis* to investigate improvements in the memory system performance of the ice model that might improve scalability. Having *Memphis* available on the *Chester* system allowed us to measure performance in a realistic setting, with all components active and running a representative data set.

We did make one change to relative to the large-scale configurations, which allowed us to focus on the memory system performance of the ice model. In the large-scale runs the atmosphere, land and ice components were combined on a single processor set. Since CESM allows flexible mapping of components to processor cores, our first step was to isolate the ice model such that it was the only thing running on the node with the *Memphis* kernel module installed.

4.1 Initial Measurements

Figure 3 presents highlights from a file, output by the *Memphis* post-processing tool, describing the remote references to DRAM made during a 12-threaded (42-process, 504 cores total) run on *Chester*.

The tool divides remote memory references by NUMA node. Since an XT5 node consists of two sockets, with six cores and one memory controller per socket, a 12-threaded run that pins one thread to one core runs across two NUMA nodes. By default, the XT runtime policy ensures that threads 0-5 run in NUMA Node 0, while threads 6-11 run in NUMA Node 1.

At the highest level, the results in the figure indicate that the threads in Node 0 suffered many more (some 13X) remote memory references than those in Node 1. The next level of detail describes which variables were referenced remotely. In this run, three heap-allocated variables *tx*, *ty*, and *tc*, and a collection of heap-allocated variables in the *ice_state* module account for nearly all the remote references.

Finally, the lowest level of detail reveals which instructions accessed each variable remotely. Note that almost all remote references are due to the same loopnest (debugging information tends to be accurate only to the level of loopnest in optimized code) in a single source file, *ice_boundary.F90*. This file implements the data packing and communication required by boundary exchanges between processors. In particular, it maintains the halo of non-local data that each process/thread requires around the data it owns.

```
do nmsg=1,halo%numLocalCopies
   iSrc     = halo%srcLocalAddr(1,nmsg)
   jSrc     = halo%srcLocalAddr(2,nmsg)
   srcBlock = halo%srcLocalAddr(3,nmsg)
   iDst     = halo%dstLocalAddr(1,nmsg)
   jDst     = halo%dstLocalAddr(2,nmsg)
   dstBlock = halo%dstLocalAddr(3,nmsg)

   if (srcBlock > 0) then
      if (dstBlock > 0) then
         do l=1,nt
            do k=1,nz
               array(iDst,jDst,k,l,dstBlock) = &
                   array(iSrc,jSrc,k,l,srcBlock)
            end do
         end do
   ...
end do
```

(a)

```
$OMP PARALLEL PRIVATE(myid,...)
myid = omp_get_thread_num()
do nmsg=1,halo%numLocalCopies
   iSrc     = halo%srcLocalAddr(1,nmsg)
   jSrc     = halo%srcLocalAddr(2,nmsg)
   srcBlock = halo%srcLocalAddr(3,nmsg)
   iDst     = halo%dstLocalAddr(1,nmsg)
   jDst     = halo%dstLocalAddr(2,nmsg)
   dstBlock = halo%dstLocalAddr(3,nmsg)

   if (srcBlock > 0) then
      if (dstBlock > 0 .and. &
          block_to_thr(dstBlock).eq.myid) then
         do l=1,nt
            do k=1,nz
               array(iDst,jDst,k,l,dstBlock) = &
                   array(iSrc,jSrc,k,l,srcBlock)
            end do
         end do
   ...
end do
```

(b)

Figure 4: Original single-threaded code that implements 4D halo exchanges between threads (`ice_halo4dr8_lclcpy`) (a), and additions that realize multi-threaded copies (b).

The function containing the loopnest responsible for almost all non-local remote references is specialized for exchanges of four-dimensional data halos. The loopnest itself, shown in Figure 4(a), performs the 4D halo exchange between threads, and is implemented as a simple copy between two regions of the same data array.

As evidence that *Memphis* has pointed out a region of code that would be useful to optimize, the following table provides timing measurements:

Timer	**Count**	**Value**
TimeLoop	240	40.687691
Bound	32410	24.978573
ice_halo4dr8	1700	12.600817
ice_halo4dr8_lclcpy	1700	7.242013

The first built-in timer, `TimeLoop`, describes the time spent in the CICE timestep loop (240 steps), a measure of the total CICE runtime. Another built-in timer, `Bound`, describes the total time spent in the boundary exchanges implemented by *ice_boundary.F90*. We have added two additional timers measuring the time spent in the four-dimensional halo exchange (`ice_halo4dr8`), and that spent in the loopnest pointed out by *Memphis* (`ice_halo4dr8_lclcpy`).

The `ice_halo4dr8_lclcpy` loopnest, responsible for fully 17% of the CICE runtime, is a clear target for optimization.

4.2 *Memphis-directed Modification 1*

Upon inspection of the containing subroutine, and of subroutines that call it, we found that the reason for the large number of remote references by Node 0 in the loopnest is that the boundary code is not threaded. Arrays passed in to the halo exchange routines have been initialized properly, in the NUMA context, so that data usually accessed by a given thread is owned by that thread, i.e., located in the same NUMA node. Therefore, reads and writes by the master thread (typically thread 0, located in Node 0) to data owned by any thread in Node 1, are non-local.

One obvious solution, likely the most non-invasive in terms of code modifications, is to thread the code that implements the `ice_halo4dr8_lclcpy` loopnest. Figure 4(b) illustrates our approach.

In CICE, the unit of parallelism is a ‘block’. Parallel loops are loops over blocks, and threads own the blocks they operate on. Thus, in order to ensure locality of reference, we cannot simply distribute iterations of the *nmsg* do-loop, but instead must allow each thread to iterate over all ‘messages’ to determine the ones that apply to blocks it owns (via a new *block_to_thr* mapping array). Since cache-coherence protocols tend to make non-local writes more expensive than non-local reads, we let the owner of a block read remote data and write the data to its halo.

The following table describes the performance improvements yielded by the new approach:

Timer	**Base**	**Mod1**
TimeLoop	40.69	36.29
Bound	24.98	20.22
ice_halo4dr8	12.60	8.75
ice_halo4dr8_lclcpy	7.24	2.38

```
    NODE: 0  total: 1156
    000) [heap]:_ice_state_2_  [ 0x172d0e80 - 0x180b9018 ]  625
      ice_boundary.F90:2779:0x9cfae4    [ 0x174cfae0 - 0x17fe41e0 ]  465
      ice_boundary.F90:4245:0x9d48e0    [ 0x176ba7f0 - 0x17e35ef0 ]  105
      ...
    001) [heap]:tc  [ 0x29b45cf0 - 0x2a5abe08 ]  231
      ice_boundary.F90:4245:0x9d48e0    [ 0x29b54848 - 0x2a5ab6a0 ]  216
      ...
    002) [heap]:tx  [ 0x2a5b14c0 - 0x2b017ad8 ]  135
      ice_boundary.F90:4245:0x9d48e0    [ 0x2a5b1c50 - 0x2b017ad8 ]  93
      ice_boundary.F90:4164:0x9d4460    [ 0x2a5b14c0 - 0x2b004730 ]  33
      ...
    003) [heap]:ty  [ 0x2b01d348 - 0x2ba83890 ]  110
      ice_boundary.F90:4245:0x9d48e0    [ 0x2b02be70 - 0x2ba837f0 ]  68
      ice_boundary.F90:4164:0x9d4460    [ 0x2b023480 - 0x2ba83490 ]  37
      ...

    NODE: 1  total: 3305
    000) [heap]:ty  [ 0x2b01d348 - 0x2ba83890 ]  708
      ice_boundary.F90:4245:0x9d48e0    [ 0x2b02be70 - 0x2ba837f0 ]  706
      ...
    001) [heap]:tx  [ 0x2a5b14c0 - 0x2b017ad8 ]  678
      ice_boundary.F90:4245:0x9d48e0    [ 0x2a5b1c50 - 0x2b017ad8 ]  675
      ...
    002) [heap]:_ice_state_2_  [ 0x172d0e80 - 0x180b9018 ]  562
      ice_boundary.F90:4245:0x9d48e0    [ 0x176ba7f0 - 0x17e35ef0 ]  494
      ice_boundary.F90:4245:0x9d48e4    [ 0x176c1b08 - 0x17e35fc8 ]  60
      ...
    003) [heap]:tc  [ 0x29b45cf0 - 0x2a5abe08 ]  159
      ice_boundary.F90:4245:0x9d48e0    [ 0x29b54848 - 0x2a5ab6a0 ]  158
      ...
```

Figure 5: Remote DRAM reference results after applying the modification depicted in Figure 4(b).

Threading improves the performance of the `ice_halo4dr8_lclcpy` loopnest by 3X. The resulting savings in time improves CICE performance by 10%. While substantial, 3X is not the perfect speedup one might hope for from 12 threads; is more available?

Figure 5 presents *Memphis* results for the new executable. Note that source code modifications to *ice_boundary.F90* have shifted line numbers such that new line 4245 corresponds to the old line 4106. While the remote references due to the `ice_halo4dr8_lclcpy` loopnest are now more evenly distributed between nodes, the counts are still very high.

4.3 Memphis-Directed Modification 2

Upon further inspection of the code we observe that the high counts are likely due to poor cache behaviour: note that in the loopnest, the *k* and *l* induction variables change faster than all the *i* and *j* induction variables (*iSrc*, *iDst*, *jSrc*, *jDst*), though *k* and *l* index higher, less contiguous, dimensions of the copied array. The result is little reuse from cache lines and therefore significant traffic between caches and memory. Since the code actually copies entire columns or rows of data to effect the copy of a halo, we would like to collect consecutive *i* and *j* references. However, implementing the approach would require substantial modification to both the code that identifies *i* and *j* values, and the many other loopnests that use the identified values.

A more localized approach recognizes that remote cache misses are substantially more expensive than local misses, and therefore collects data requiring communication between threads into a contiguous buffer. The thread that owns the data to be read writes it into a local buffer, so that the thread that owns the halo to be written can read *consecutive,* though remote, elements from the buffer and writes them to its halo. Since more data per cache-line is used, fewer remote cache lines are communicated and expensive communication is reduced.

Figure 6 demonstrates the new approach. The loopnest is essentially replicated. In the first half, the owner of the source data writes into *bufLocal*. In the second half, after a synchronizing barrier, the owner of the destination halo copies data from *bufLocal* into the halo.

Figure 7 indicates a substantial reduction in the number of non-local reference samples. Notably, there are now no samples due to the `ice_halo4dr8_lclcpy` loopnest. The following table indicates the performance improvement due to this second source-code modification:

```
!$OMP PARALLEL PRIVATE(myid,...)
myid = omp_get_thread_num()
do nmsg=1,halo%numLocalCopies
   dstBlock = halo%dstLocalAddr(3,nmsg)
   srcBlock = halo%srcLocalAddr(3,nmsg)
   if (dstBlock > 0 .and. srcBlock > 0) then
      if (block_to_thr(srcBlock).eq.myid) then
         iSrc     = halo%srcLocalAddr(1,nmsg)
         jSrc     = halo%srcLocalAddr(2,nmsg)
         i = 0
         do l=1,nt
            do k=1,nz
               i = i + 1
               bufLocal(i,nmsg) = &
                   array(iSrc,jSrc,k,l,srcBlock)
            end do
         end do
      endif
   ...
end do
```

```
!$OMP BARRIER
do nmsg=1,halo%numLocalCopies
   dstBlock = halo%dstLocalAddr(3,nmsg)
   srcBlock = halo%srcLocalAddr(3,nmsg)
   if (dstBlock > 0 .and. srcBlock > 0) then
      if (block_to_thr(dstBlock).eq.myid) then
         iDst     = halo%dstLocalAddr(1,nmsg)
         jDst     = halo%dstLocalAddr(2,nmsg)
         i = 0
         do l=1,nt
            do k=1,nz
               i = i + 1
               array(iDst,jDst,k,l,dstBlock) = &
                  bufLocal(i,nmsg)
            end do
         end do
      endif
   endif
enddo
```

Figure 6: The `ice_halo4dr8_lclcpy` loopnest after replication of the loopnest to reduce communication.

Timer	**Base**	**Mod1**	**Mod2**
TimeLoop	40.69	36.29	35.90
Bound	24.98	20.22	19.86
ice_halo4dr8	12.60	8.75	8.61
ice_halo4dr8_lclcpy	7.24	2.38	1.95

Although we have doubled the number of instructions executed, the local copy looopnest now executes nearly 4X faster than the base version.

However, not all of the improvement is visible to `ice_halo4dr8`, or to functions higher in the call-tree. The local copy loopnest occurs while a node is in the middle of an asynchronous inter-node communication event. It may be that the inter-node communication now takes longer to complete than the new local copy.

In any event, further improvements to the `ice_halo4dr8_lclcpy` loopnest are unlikely to be fruitful in terms of reducing the overall time to completion. Figure 7 indicates several new sources of remote DRAM references, all from the same *ice_boundary.F90* source file, and originating from loopnests exhibiting similar poor cache behaviour. We are currently investigating a more comprehensive approach that would achieve the goals of the halo-copying routines while better preserving locality of reference.

4.4 Memphis Overhead

The following table presents a measure of *Memphis* overhead, comparing execution time – based on the `TimeLoop` time-step timer – of runs 1) with IBS and instrumentation of allocation statements, and 2) without either IBS or instrumentation:

	IBS Off, No Instrumentation	**IBS On, Instrumented**
Base	40.69	41.18
Mod1	36.29	36.63
Mod2	35.90	36.31

These measurements indicate that the overhead is

```
NODE: 0  total: 638
000) [heap]:_ice_state_2_  [ 0x172a8000 - 0x180b8090 ]  493
  ice_boundary.F90:2779:0x9cfae4    [ 0x174b10b0 - 0x1804ba10 ]  435
  ice_boundary.F90:4164:0x9d44f0    [ 0x176c0f80 - 0x17e33f58 ]  31
  ...
001) [map-anon-23]:tx  [ 0x2aac10a6e5c0 - 0x2aac114d4250 ]  35
  ice_boundary.F90:4164:0x9d44f0    [ 0x2aac10a74768 - 0x2aac114d0958 ]  35
002) [map-anon-23]:ty  [ 0x2aac114da788 - 0x2aac11f406c8 ]  34
  ice_boundary.F90:4164:0x9d44f0    [ 0x2aac114dee80 - 0x2aac11f404a8 ]  33
  ...

NODE: 1  total: 598
000) [heap]:<not-found>  [ 0x24b94140 - 0x2b033088 ]  138
  ice_history.F90:2564:0xa45d9c    [ 0x2918bcc0 - 0x29b3b310 ]  136
```

Figure 7: Remote DRAM reference after applying modification 2, depicted in Figure 6.

consistently only about 1% of runtime, even with instrumentation. Note, however, that we have removed instrumentation from several "single-use" communication buffer allocations, which had been filling instrumentation files with largely useless information.

5. Conclusion

We have described *Memphis* and how we have deployed and used it on *Chester*, a test XT5 system at the OLCF at ORNL. It is our hope that this demonstration of usefulness combined with ease-of-deployment will convince other Cray installation sites that they should consider creating a *Memphis* queue.

6. Acknowledgments

The authors would like to thank John Lewis of Cray, without whose help *Memphis* would never have gotten on an XT5.

7. About the Authors

Collin McCurdy is an R&D associate staff member in the Future Technologies Group at Oak Ridge National Laboratory. His research focuses on memory system designs in current and future processor architectures and their implications for scientific applications. He has a PhD in Computer Science from the University of Wisconsin–Madison and is a member of the Association for Computing Machinery. He can be reached at cmccurdy@ornl.gov.

Jeffrey Vetter is group leader of the Future Technologies Group at Oak Ridge National Laboratory. He can be reached at vetter@ornl.gov.

Patrick H. Worley is a senior R&D staff member in the Computer Science and Mathematics Division of Oak Ridge National Laboratory. His research interests include parallel algorithm design and implementation (especially as applied to simulation models used in climate and fusion energy research) and the performance evaluation of parallel applications and computer systems. He is currently a co-chair of the Community Earth System Model (CESM) Software Engineering Working Group, the principal investigator for the Performance Engineering and Analysis Consortium End Station DOE INCITE project, and is an Associate Editor of the journal Parallel Computing. Worley has a PhD in computer science from Stanford University. He is a member of the Association for Computing Machinery and the Society for Industrial and Applied Mathematics. He can be reached at worleyph@ornl.gov.

Don Maxwell is a Senior System Administrator at Oak Ridge National Laboratory primarily focused on the Cray XT series. He has been a key member of past teams in bringing up new supercomputers for the NCCS. He can be reached at maxwellde@ornl.gov.

8. References

[1] C. McCurdy and J. Vetter, "Memphis: Finding and fixing NUMA-related performance problems on multi-core platforms," in *2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*, White Plains, NY, 2010.

[2] P. J. Drongowski, "Instruction-Based Sampling: A New Performance Analysis Technique for AMD Family 10h Processors," Advanced Micro Devices, Inc.2007.

[3] J. Dean, J. E. Hicks, C. A. Waldspurger, W. E. Weihl, and G. Chrysos, "ProfileMe: hardware support for instruction-level profiling on out-of-order processors," presented at the Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture, Research Triangle Park, North Carolina, United States, 1997.

[4] "Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3B: System Programming Guide, Part 2," Intel Corporation, 2009.

[5] M. Jackson, S. Jackson, and D. Maxwell, "Moab Workload Manager on Cray XT3," in *Cray User Group (CUG)*, Lugano, Italy, 2006.

[6] LANL. (2011, May). *CICE: The Los Alamos Sea Ice Model*. Available: http://climate.lanl.gov/Models/CICE/

[7] UCAR. (2011, May). *Community Earth System Model*. Available: