DMAPP - An API for One-sided Program Models on Baker Systems

Monika ten Bruggencate, Ph.D., *Cray Inc.*
Duncan Roweth, Ph.D., *Cray UK Ltd.*

ABSTRACT: *Cray Baker and follow-on systems will deliver a network with advanced remote memory access capabilities. A new API (DMAPP) has been developed to expose these capabilities to one-sided program models. This paper presents the DMAPP API as well as preliminary performance data for DMAPP on Gemini.*

KEYWORDS: Cray Baker system, one-sided program models

1. Introduction

Since the introduction of Cray T3D, Cray has supported various forms of distributed global memory (DM) program models. Distributed global memory is memory which allows a process to access memory in a remote process without involvement of the remote processor on which the remote process is scheduled.

Support for one-sided program models such as Cray SHMEM[1] and Partitioned Global Address Space (PGAS) languages[2], e.g. Unified Parallel C (UPC) [3] and Co-Array Fortran (CAF) [4], is critical for the future of Cray systems, including Cray Baker and follow-on systems. As a result, a new API, called the Distributed Memory Application (DMAPP) API, was developed to support such one-sided program models on Cray Baker systems. The DMAPP API has two main purposes: to allow higher-level software to realize the hardware performance of the Baker network technology and to allow higher-level software to be portable to future instantiations of this network technology. As such the DMAPP API is intended to be used for library and compiler development rather than directly within end-user application software.

The remainder of this document is organized as follows. Section 2 introduces the DMAPP Program Model and API. Section 3 gives a brief overview of the Cray Baker hardware and software stack, from DMAPP's perspective, and describes the DMAPP implementation. Section 4 discusses preliminary performance data. Future work is discussed in Section 5.

2. DMAPP Program Model and API

2.1 DMAPP Program Model

The DMAPP program model is designed to support both compiler-based (e.g. UPC and CAF) and library based (e.g. Cray SHMEM) one-sided program models in which a distinction exists between local and remote memory references at some level in the program model.

In the DMAPP program model a group of processes execute the same executable in parallel. For the purpose of this discussion, such a group of related processes are called a *job*. Typically the number of processes executing the application does not change over the course of a job. The processes in a job are sometimes called processing elements (PEs). Although each PE executes in its own address space, it can access certain memory segments of other PEs in a one-sided (PUT/GET) manner using PGAS language constructs or by invoking function calls to libraries which support one-sided program models, such as Cray SHMEM. Such memory segments which can be accessed by remote PEs are called *remotely accessible*.
Remotely accessible memory segments in a PE can be classified as either *symmetric* or *non-symmetric*. The address of an object within a symmetric memory segment of a PE has a known relationship (known to some component of the runtime, at least) to the address of this same object in the address space of another PE in the job. This distinction has important implications. Objects within symmetric memory segments can be accessed in a one-sided manner by other PEs in the job using locally generated address information. Objects within non-symmetric memory segments on PE A can only be accessed in a one-sided manner by another PE B using address information generated by PE A and communicated to PE B. See Sections 2.2.2, 2.2.3 and

2.2.7 for more details. In the DMAPP program model, symmetric memory segments are the statically linked data segment and a symmetric heap segment.

For simplicity, for the remainder of this paper, one-sided program models implemented on top of DMAPP will be referred to as DMAPP applications.

2.2 DMAPP API

Since the AMD64 processor cannot be efficiently used to directly load from or store to remotely accessible memory on Baker nodes, DMAPP provides an API for interfacing with the remote memory access (RMA) hardware mechanisms available in Baker systems. The DMAPP RMA functions can be divided into

- one-sided RMA functions
- RMA synchronization functions

where one-sided RMA functions can be further subdivided into three categories:

- blocking
- non-blocking explicit
- non-blocking implicit

A process returns from a blocking function call only after the effects of the remote memory access are globally visible in the system. Blocking functions do not allow overlapping of communication and computation within a single process, although one process' communication may overlap with another's computation. In the case of a non-blocking explicit function call, a synchronization identifier (syncid) is returned to the process and the effects of the remote memory access are only assured to be globally visible in the system after the initiating process has determined via an explicit synchronization call that the syncid has been retired. In the case of a non-blocking implicit function call, no explicit synchronization identifier is returned to the process and the effects of the remote memory access are only assured to be globally visible in the system after a synchronization call by the initiator of the non-blocking implicit function call. See Section 2.2.5 for more details.

All DMAPP functions return a status value indicating success or failure of the function. Note that for non-blocking RMA functions, this status indicates whether the transfer has been issued successfully, but does not indicate whether or not the remote memory access request completed successfully.

2.2.1 Initialization, Termination and Query

Before calling any other DMAPP functions, a DMAPP application must call the initialization function *dmapp_init.* This function sets up NIC specific resources and prepares symmetric memory segments of an DMAPP application for remote memory access. Segments which are exported to be remotely accessible are the static data segment and the symmetric heap.

Similarly, the very last call to the DMAPP library must be a call to *dmapp_finalize* which tears down previously set up resources.

At runtime, DMAPP application software can determine job and RMA attribute specific information via calls to *dmapp_get_jobinfo* and *dmapp_get_rma_attrs*, respectively. Job specific information includes details on the segments of the address space which are exported. RMA specific information includes details on the routing mode used, maximum number of outstanding non-blocking operations supported and the offload threshold (see Sections 3.1 and 3.2).

2.2.2 Point-to-point one-sided RMA functions

Point-to-point one-sided RMA functions allow a PE to access memory on one remote PE at a time. This group of functions share the following characteristics. The remote address is specified by a triplet of virtual remote address, remote segment descriptor and the remote PE. For functions with PUT semantics, the remote address is the target address of the transfer, for functions with GET semantics, it is the source address of the transfer. The number and type of elements to be transferred are specified. The remote memory region defined by the remote address and the amount of data to be accessed must reside within an exported memory segment of the remote PE. These characteristics also apply to all other RMA functions (see Section 2.2.3). Strides and indices are in units of type of the elements. Point-to-point RMA functions allow one-sided access to both types of remotely accessible memory, symmetric as well as non-symmetric.

The PUT functions are *dmapp_put, dmapp_put_nb* and *dmapp_put_nbi.* They store a contiguous block of data starting at a source address in local memory into a contiguous block at a remote address.

The GET functions are *dmapp_get, dmapp_get_nb,* and *dmapp_get_nbi*. They load from a contiguous block of data starting from a remote address and returning the data into a contiguous block starting at a target address in local memory.

The strided PUT functions are *dmapp_iput, dmapp_iput_nb* and *dmapp_iput_nbi*. They deliver data starting at a source address in local memory to a remote address using separate source side and target side strides.

The strided GET functions are *dmapp_iget, dmapp_iget_nb and dmapp_iget_nbi*. They load data starting from a remote source address using a source side stride *sst* and returning the data to a target address in local memory using a separate target stride *tst*.

The indexed PUT functions are *dmapp_ixput_nb, dmapp_ixput_nbi,* and *dmapp_ixput.* They scatter a contiguous block of data starting at a source address in local memory to a remote address using offsets specified by the *tidx* array.

The indexed GET functions are *dmapp_ixget_nb*, *dmapp_ixget_nbi*, and *dmapp_ixget*. These functions gather data starting from a remote source address using offsets specified by the *sidx* array and returning the data into a contiguous block starting at a target address in local memory.

2.2.3 PE-strided One-sided RMA Functions

The DMAPP API provides functions that gather data from and scatter data to remotely accessible addresses across a set of PEs in a DMAPP job. The characteristics discussed in Section 2.2.2 for point-to-point RMA functions also apply to this group of functions. In addition, the remote address must be a symmetric address. Note also that none of these functions are collective operations. They are best used when a small amount of data needs to be scattered to or collected from a set of PEs.

The PUT with indexed PE stride functions are *dmapp_put_ixpe_nb*, *dmapp_put_ixpe_nbi*, and *dmapp_put_ixpe*. These functions deliver data starting at a source address in local memory to a list of target PEs starting at a remote address in their memories. When the transfer is complete, each target PE has a copy of the contents of the original source buffer.

The scatter with indexed PE stride function are *dmapp_scatter_ixpe_nb*, *dmapp_scatter_ixpe_nbi*, and *dmapp_scatter_ixpe*. They deliver data starting at a source address in local memory to a list of target PEs *target_pe_list* starting at a remote address in their memories. Unlike the *dmapp_put_ixpe* function, the source array specifies an array of size *num_target_pes***nelems**sizeof(*type*). A target PE at index *i* into the *target_pe_list* will receive elements *i***nelems* to (*i*+1)**nelems*-1.

The gather with indexed PE stride functions are *dmapp_gather_ixpe_nb*, *dmapp_gather_ixpe_nbi*, and *dmapp_gather_ixpe*. They gather data starting at a remote source address from a list of PEs and concatenate the data in a buffer specified by the target address in local memory.

2.2.4 AMOs

A set of scalar-type Atomic Memory Operation (AMO) functions are provided: *dmapp_<op>_qw_nb*, *dmapp_<op>_qw_nbi* and *dmapp_qw_<op>,* where *op* is either AADD, AAND, AOR, or AXOR or AFADD, AFAND, AFOR, or AFXOR. AMOs only operate on 64-bit (quad-word) entities. For atomic functions with PUT semantics (AADD, AAND, AOR, AXOR) as well as for those with GET semantics (AFADD, AFAND, AFOR, AFXOR) the remote memory location must reside in exported memory of the remote PE.

Additionally, a set of two-operand AMO functions is provided: *dmapp_<op>_qw, dmapp_<op>_qw_nb,* and *dmapp_<op>_qw_nbi*, where *op* is AFAX (atomic fetch and exclusive or) or ACSWAP (atomic compare and swap). The source memory location must reside in exported memory of the remote PE.

2.2.5 Synchronization

DMAPP applications use synchronization functions to determine when locally initiated, non-blocking RMA requests have completed, i.e. the data transferred has arrived in the target memory location.

A process initiating an explicit non-blocking RMA request can determine when the transfer is complete by calling the *syncid* specific synchronization functions *dmapp_syncid_test* or *dmapp_syncid_wait.* The latter is a blocking version of the former and returns only when all remote memory accesses associated with the specified *syncid* are globally visible in the system.

A process can determine when one or more non-blocking implicit RMA requests are globally visible in the system by calling the
dmapp_gsync_test or *dmapp_gsync_wait* functions. The latter is a blocking version of the former and returns only when all remote memory accesses associated with non-blocking implicit requests previously initiated by the caller are globally visible in the system. Note that invoking the gsync style functions does not free resources associated with non-blocking explicit RMA requests. *dmapp_syncid_test* or *dmapp_syncid_wait* calls must be made for each previously issued non-blocking explicit RMA request in order to free up DMAPP internal resources. Further, note that only the initiating process has visibility of completion of a previously issued request.

2.2.6 Symmetric Heap Management

DMAPP provides routines for allocating and releasing symmetric heap memory. DMAPP applications are responsible for preserving the symmetry of this memory. This is achieved by ensuring that all PEs in a job make the same calls to the symmetric heap management functions (including involving the same amount of memory) in the same sequence. The function *dmapp_sheap_malloc* allocates the specified number of bytes of memory from the symmetric heap. The function *dmapp_sheap_realloc* changes the size of a block of memory which was previously allocated by *dmapp_sheap_malloc*. The function *dmapp_sheap_free* frees a block of memory previously allocated by *dmapp_sheap_malloc* or *dmapp_sheap_realloc*. The DMAPP application controls the size of the symmetric heap.

2.2.7 Dynamic Memory Registration and Deregistration

The dynamic memory management functions *dmapp_mem_register* and *dmapp_mem_unregister* allow a DMAPP application to dynamically register or deregister memory which for instance was allocated from the private heap or using *mmap*. Memory registered by a call to dmapp_mem_register becomes remotely accessible and is assumed *non-symmetric.* It cannot be remotely accessed using a locally created address and segment descriptor. Instead, it can only be accessed using a remotely generated address and segment descriptor that have been communicated to the initiating process. In general, the DMAPP application must perform an out-of-bounds exchange of remote address and segment descriptor information between communicating PEs for dynamically registered memory regions.

3. DMAPP Implementation

3.1 Gemini Hardware Overview

A Cray Baker systems consists of AMD Opteron multi-core processors connected via a Cray proprietary interconnection network. The network interface chip (NIC) which provides an interface between the Opteron processors and the Baker interconnection network is called Gemini. Gemini delivers advanced remote memory access features and provides the communication modes and programming model that create the abstraction of a global, shared address space across the entire machine. Additional information on Gemini can be found in [5].

Gemini provides completion queues (CQs) for asynchronous, light-weight event notification. They allow local tracking of progress of FMA and BTE requests. A CQ typically resides in host memory.

Gemini implements two modes for accessing remote memory on another node of a Baker system:

- **Fast Memory Access (FMA)**. This mode provides low overhead, user-level, direct load and store access to remote memory. FMA provides the ability to run short transfers across the network at low latency. FMA translates stores by the AMD64 processor on the local node into fully qualified network requests. Processors that use the FMA mechanism do not directly load and store into memory on other nodes. Instead, stores into an FMA window are used to generate remote memory reference requests and stores to an associated FMA descriptor control how the FMA window is mapped to the job's memory segments.
- **Block Transfer Engine (BTE)**. This mode provides memory-to-memory copies using the block transfer engine. Data is moved asynchronously between local and remote memory, a completion event is generated when the transfer has completed. BTE is primarily intended for large, asynchronous data transfers between nodes.

In addition to the two communication modes, Gemini supports low-latency synchronization using a set of atomic memory operations (AMOs). This set has been modelled on the set of AMOs provided on Cray X2 systems. The AMOs are restricted to 64-bit quantities.

Memory regions must be registered with the local Gemini before they can be accessed from other nodes using FMA or BTE.

3.2 DMAPP Implementation on Gemini

Figure 1 depicts a DMAPP-centric view of the Gemini software stack. A complete discussion of the software stack is beyond the scope of this paper. The Gemini Hardware Abstraction Layer (GHAL) is a set of macros which allows software such as DMAPP to efficiently access the Gemini hardware. The kernel-level component of the Gemini driver is named kGNI, whereas the user-level component is named uGNI (see [6] for details).

DMAPP is used to implement higher-level, more portable APIs such as Cray SHMEM, PGAS compilers and Chapel. This hierarchical design allows such higher-level software to be portable to future instantiations of Gemini while network specific code is hidden within the DMAPP, uGNI, kGNI and GHAL software components.

There are a few phases during the execution of an application when DMAPP interfaces with Gemini through uGNI, and in turn the Linux kernel and kGNI. These phases include initialization and termination. During initialization, the Gemini driver is invoked to prepare Gemini resources such as to create a Communication Domain[1] (CDM) and attach it to the NIC, create a Completion Queue (CQ) and to pre-register symmetric memory regions, i.e. the statically linked data segment and the symmetric heap, with the NIC. Pre-registration of the data segment and the symmetric heap eliminate the need for expensive memory registration of these segments during end-user application execution. During termination, the Gemini driver is invoked to properly

[1] A Communication domain is the set of PEs and associated memory segments that make up a job see [1] for details.

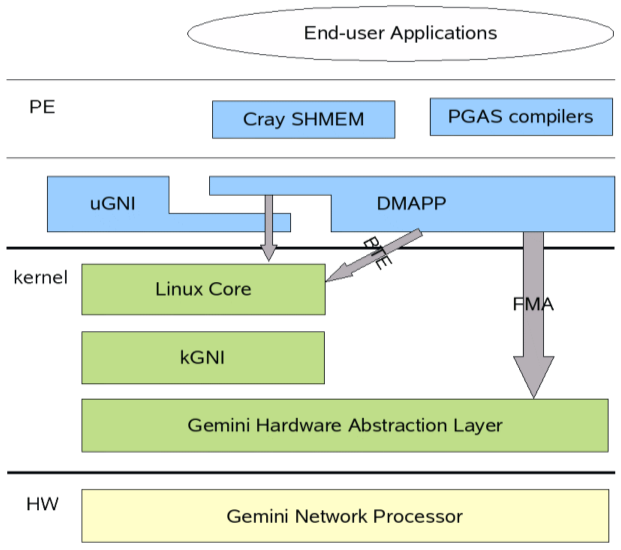

Figure 1: Gemini software stack

release all Gemini resources which were previously held by the job.

To eliminate waste and maximize reuse of memory registration related Gemini resources, DMAPP implements a memory registration cache, which tracks and, if possible, reuses previously registered memory regions. When a memory region to be registered dynamically is found in the registration cache, DMAPP software overhead is minimal. When it is not found in the cache, DMAPP interfaces with the Gemini driver to register the region.

The BTE mechanism is intended to be used for large, contiguous transfers, when the data transfer is best offloaded from the processor. DMAPP takes advantage of this mechanism for large PUTs or GETs, e.g. calls to *dmapp_put_nb* and *dmapp_get,* where the amount of data to be transferred exceeds a user-tuneable threshold. Above this threshold DMAPP executes a call to the BTE to asynchronously move the data. The default threshold value is 4k bytes. The higher level program model calling DMAPP is free to alter this threshold at any point in time, for example setting a high value in a call that is known to be blocking.

The FMA mechanism is intended to be used for small transfers. DMAPP takes advantage of it for all other data transfer functions, such as PUTs and GETs where the amount of data to be transferred is less than the offload threshold, strided and indexed functions, all PE-strided functions and AMOs. DMAPP bypasses the kernel and directly interacts with the FMA hardware on the Gemini NIC via a thin set of GHAL macros. The GHAL macros allow DMAPP to directly store to the Gemini. Information stored to the NIC includes information specifying the remote memory location, details relating to performance, the data to be transferred (for operations with PUT semantics) or details about the amount of data to be gotten (for operations with GET semantics), and more. The stores in turn are translated by the Gemini into fully qualified network requests.

To implement the synchronization functions, DMAPP takes advantage of the light-weight CQ mechanism. DMAPP invokes GHAL to determine if a completion event has arrived locally and then simply analyzes the state of the event to determine success or failure of the transfer associated with the completion event.

Lastly, the symmetric heap management functions are implemented by means of a region allocator within the DMAPP library. The symmetric heap uses large pages, 2Mbytes by default.

The discussion in this section illustrates how DMAPP has been designed and implemented to allow higher-level software such as PGAS compilers and Cray SHMEM to realize much of the performance of the Gemini hardware by keeping the software overhead for operations on the critical path of an end-user application low.

4. Preliminary Performance Measurements

DMAPP performance measurements were made using prototype Gemini hardware and software. We used 2100MHz Opteron CPUs connected to Gemini via a 2400MHz HyperTransport interface. Measurements were taken between processes on nodes connected to neighbouring Gemini routers.

DMAPP provides a low overhead API to the Gemini hardware. In Figure *2* we show PUT and GET latencies as a function of transfer size. We measure the time taken to perform blocking PUTs as seen by the source process as well as the half round trip ping-pong time. The latency as measured at the source includes that for a response to be returned.

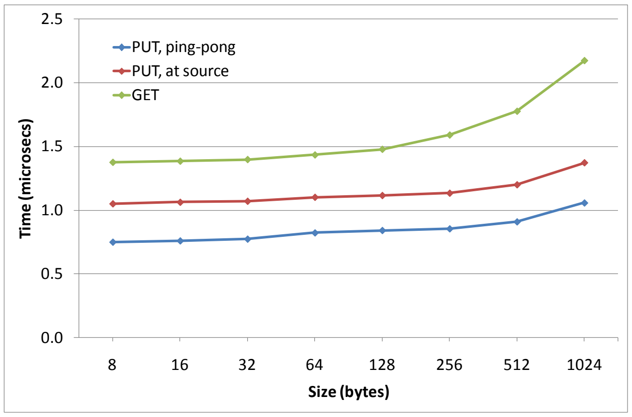

Figure 2: Gemini DMAPP PUT and GET latencies

In Figure *3* we show the PUT bandwidth as a function of increasing element size. Results are shown for one, two and four processes per node. The impact of the switch from FMA to BTE transfer at 4K bytes is clear. DMAPP application writers need to consider how best to set this threshold, trading the increased latency of BTE transfers with the increased CPU availability.

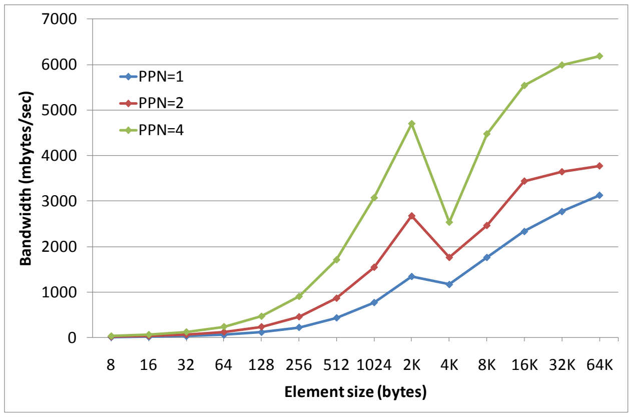

Figure 3: Gemini DMAPP bandwidths for 1, 2 and 4 processes per node

In Figure *4* we show the bandwidth as a function of the number of non-blocking PUTs issued for a range of small transfer sizes. Only small numbers of transfers need be issued to get a marked increase in bandwidth.

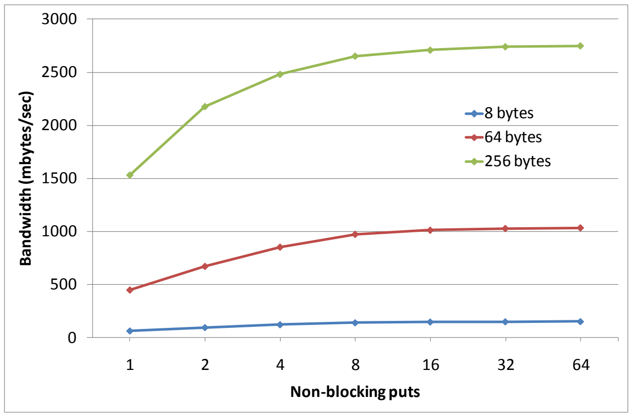

Figure 4: Gemini DMAPP bandwidths for increasing numbers of non-blocking PUTs

In Figure 5 we show the rate at which Gemini can perform 64-bit DMAPP strided PUTs as a function of the target stride for a range of vector lengths. The performance of strided PUTs and GETs is of particular importance to the Co-Array Fortran compiler.

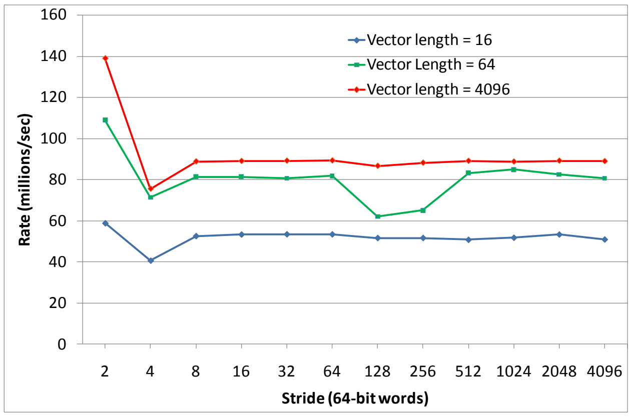

Figure 5: Gemini DMAPP 64-bit strided PUT rate as a function of target stride

In Figure 6 we show the rate of AMOs as a function of job size. In this test all DMAPP AMO calls target atomic variables owned by process 0. We show the aggregate rate for the job when all process target a single AMO and for an AMO chosen at random from a set of 8K, all in process 0. The Gemini NIC caches up to 64 AMOs, the 8K case shows the additional cost of HyperTransport operations to fetch/store the variable.

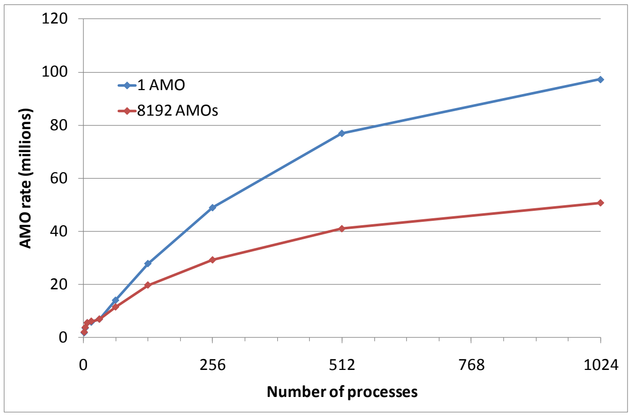

Figure 6: Gemini DMAPP rate of atomic memory operations targeting process 0 as a function of job size.

5. Status and Future Work

DMAPP will be released in July 2010 together with the first Baker systems and the Cray SHMEM, UPC and CAF products that use it. As time and resources permit, optimizations will be implemented. Two areas of particular interest are message rates for small transfers and concurrency.

Initial performance analysis has revealed that FMA descriptor updates (see Section 3.2) are relatively expensive and should be minimized as much as possible. Opportunities to reduce the number of descriptor updates have been identified for non-blocking implicit operations as well as strided and indexed operations. Additional opportunities may be discovered upon further inspection and experimentation. Reducing the number of FMA descriptor updates will increase the small message rate of affected operations in many, but not necessarily all, use cases.

The Gemini NIC provides each process with its own FMA hardware, enabling it to issue RMA operations without synchronizing with other processes. Use of shared memory multi-threaded programming within the node is of increasing interest. Extending DMAPP support to provide multiple FMA descriptors per process would enable threads to issue RMA operations independently.

Acknowledgments

This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0001. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Defense Advanced Research Projects Agency.

References

[1] Cray XT Programming Environment Users Guide (S-2396) and Cray SHMEM manual pages
[2] Productivity and performance using partitioned global address space languages, K. Yelick et al. Proceedings of the 2007 International Workshop on Parallel Symbolic Computation
[3]UPC Language Specification V1.2, for details see www.guu.edu/upc
[4] Co-array Fortran Language Definition, for details see www.co-array.org
[5] Cray Gemini Whitepaper, see www.cray.com
[6] Cray Gemini Network API Specification (S-2446)