

# Implementing an API for Distributed Adaptive Computing Systems

*Mark Jones, Luke Scharf, Jonathan Scott, Chris Twaddle, Matthew Yaconis, Kuan Yao, Peter Athanas*  
*Virginia Tech*  
*and*  
*Brian Schott*  
*USC/ISI-East*

## 1 Introduction

Many applications require the use of multiple, loosely-coupled adaptive computing boards as part of a larger computing system. Two such application classes are embedded systems in which multiple boards are required to physically interface to different sensors/actuators and applications whose computational demands require multiple boards. In addition to the adaptive computing boards, the computing systems for these application classes typically include general-purpose microprocessors and high-speed networks.

The development environment for applications on these large computing systems is not unified. Typically, a developer uses VHDL simulation and synthesis tools to program the FPGAs on the adaptive computing boards. External control for the board, such as downloading new configurations or setting clock speeds, is provided through a vendor-specific API. This API is typically accessed in a C host program that the developer must write in a high-level language environment. Finally, the developer is responsible for writing the networking code that allows interaction between the separate adaptive computing boards and general-purpose microprocessors. No tools are available for either debugging or performance monitoring in this agglomerated system. Development on these systems is time-consuming and platform-specific.

A standard ACS API is proposed to provide a developer with a single API for the control of a distributed system of adaptive computing boards, including the interconnection network. The API:

- is freely available,
- is intended to address a wide range of application requirements including embedded systems and cluster-based adaptive computing systems,
- provides a C-language binding,

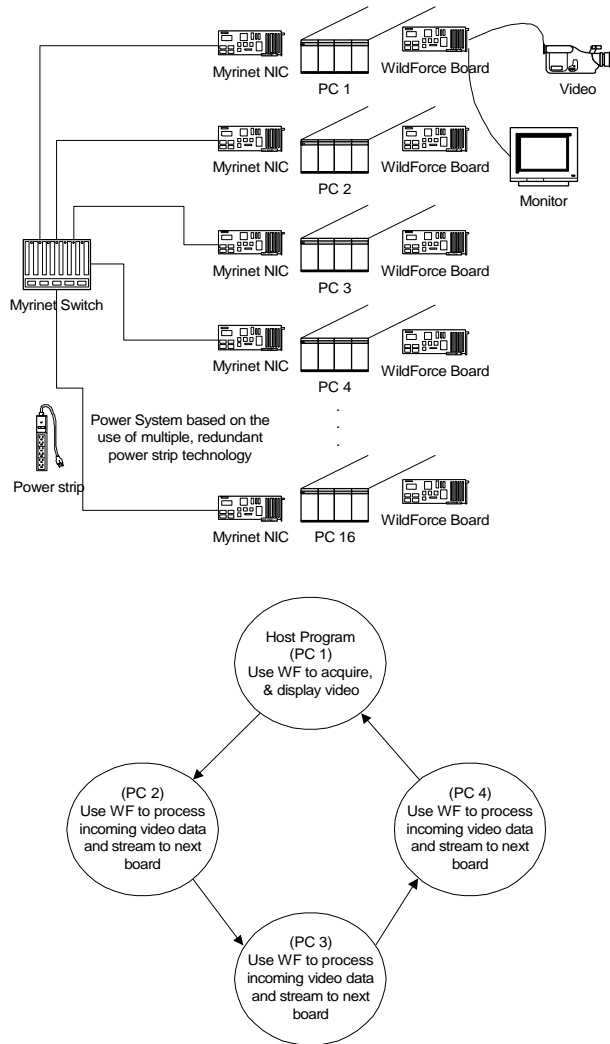
- allows source code porting and scaling from small research platforms to large field-deployable platforms,
- provides a common API for a range of ACS platforms to promote portability,
- allows a developer to write a single host program that controls multiple boards,
- configures the network for communication between multiple boards,
- allows for direct reads/writes to each board's local memory,
- provides for multiple FIFO queues to be configured between boards, and
- is currently targeted to the SLAAC-1, SLAAC-2, and Wildforce platforms [6].

The ACS API is not a programming language for FPGAs; FPGAs are still configured using bitfiles generated by other development environments.

Related projects in this area are Cheops/Magic 7 [7] and Magic 8 [8]. Cheops/Magic 7 is a video accelerator system that includes special-purpose hardware elements. Magic 8 extends Cheops/Magic 7 by providing programming support for networks of workstations with or without special-purpose processors. Both Magic 7 and Magic 8 provide dynamic mapping of applications onto the best available special-purpose processors, while the ACS API provides the user with explicit control of adaptive computing boards. The ACS API could provide Magic 8 with broader support of adaptive computing boards in embedded environments.

This paper presents the pertinent aspects of the API and the design philosophy, along with implementation details. Adaptive computing architectures and applications are discussed in Section 2. The structure of the API as well as short example host programs using the API are given in Section 3. The object-oriented implementation of the API is described in Section 4. Future work,

including high-performance networking, is outlined in Section 5. Finally, the status of the implementation and how to access it are given in Section 6.



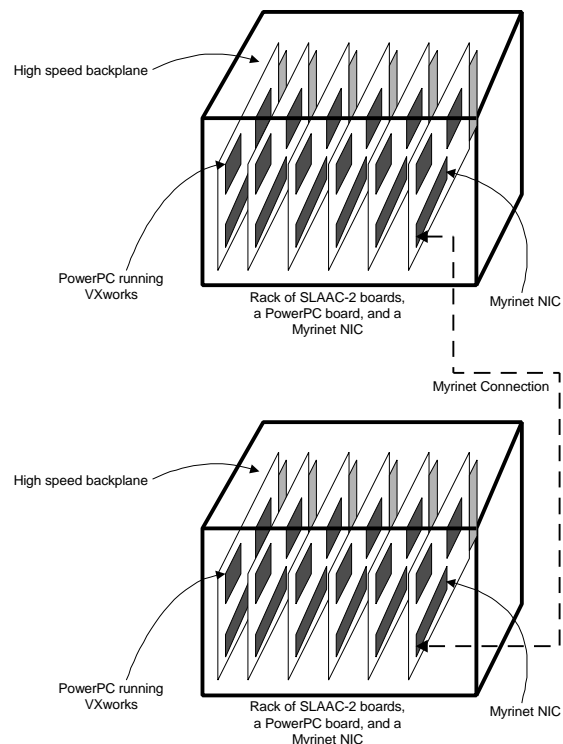
**Figure 1: The Tower of Power and a "ring" based image processing pipeline**

## 2 Distributed Adaptive Computing Architectures and Applications

The cluster computing paradigm is a cost-effective method of constructing small parallel computers using commercial off-the-shelf technology to exploit coarse- and medium-grain parallelism [2]. Adaptive computing systems are a COTS-based approach that exploits fine-grain and, to some extent, coarse-grain parallelism. The Tower of Power is an example of one such cluster. As shown in Figure 1, the Tower of Power has sixteen Pentium II PCs each equipped

with a WildForce board tightly coupled to a Myricom LAN/SAN card; the PCs are connected through a sixteen port Myrinet switch. A total of 80 XC4062XL FPGAs and memory banks are distributed throughout the platform, and are available as computing resources.

Such clusters, in various sizes, are readily available within the research community and suitable for many applications. The physical size of such a cluster, however, is not acceptable for use in embedded systems. A typical embedded system, such as the one shown in Figure 2, may have a single host microprocessor connected to a rack of densely populated ACS boards connected by a Myrinet switch. These large, embedded systems present at least two major difficulties. They present the same programming difficulties as the cluster model, but with the added complexity of a single microprocessor to control all of the nodes and provide all of the debugging information. Second, full-size embedded systems are quite expensive and beyond the means of most research laboratories; porting research codes from affordable laboratory clusters to full-size embedded systems is difficult and time-consuming.



**Figure 2: An embedded system with multiple SLAAC-2 boards**

The ACS API provides an integrated development environment for exploiting clusters and embedded systems. It is the intent that applications developed on clusters using the ACS API can be ported without source code modification to larger, embedded systems. The API provides the structure and communication for coarse-grain parallelism while controlling the adaptive computing boards that provide the fine-grain parallelism. The primary model of coarse-grain parallelism supported by the ACS API is a channel-based model. A channel is a single logical FIFO queue connecting two computational nodes. A user can construct an arbitrary system of channels and nodes. For example, using the API, the TOP, can mimic large systolic arrays such as those provided by multiple Splash-2 boards. The user is freed from the necessity of writing control programs on every computer to pass data manually; the API implementation controls the network and remote boards automatically after the channels are allocated. Furthermore, the API offers a consistent interface for both local and remote resources while preserving system performance.

The minimum system requirements assumed by the SLAAC API are a host CPU running a modern OS and an ACS board with logical or physical FIFOs. The current implementation also requires MPI-based network connectivity between all nodes of the system.

The API could be used to create an image processing application that requires more computational resources that are available on a single board. For example, one PC on an ACS cluster may be equipped with video in and out ports. To harness multiple boards in a deep image processing pipeline, a ring structure could be constructed in which video data is acquired in PC 1, processed in PCs 2-4, and then returned to PC 1 for output to a monitor. The communication in this structure is handled without intervention from the user after the channels are allocated.

### 3 API Description

The API is accessed from a C program called the *host* program. The host program provides for control of the entire system; the programmer need only write one host program no matter how many boards are in the system. The host program can

access several classes of API calls allowing functionality such as system management, memory access, streaming data, and board management. Additional functions to allow for concurrent operations on multiple boards are also part of the API. One of the design goals of the ACS API is provide a simple API for the control of a complex system. A user is required to master a small set of commands to create a working system. This simplicity is provided by a set of reasonable default behaviors. An advanced user can gain more control of the system by altering these default behaviors through the use of a wider range of API functions.

#### 3.1 System Management

The central component of the API is the specification and creation of a *system* object by the programmer. A system object is composed of *nodes* and *channels*. A node is a computational device, for example, an adaptive computing board. A channel is a logical FIFO queue between two nodes.

The first task in a host program is the creation of the system object. The code fragment in Figure 3 will construct the logical system in Figure 1. The program first creates two ACS data structures that describe the desired system object, in this case, a ring of four AMS WildForce boards. Note that these boards could easily be SLAAC-2 or Pamette boards; the API is not specific to any particular architecture. After calling the API initialization routine, the program makes a single call to *ACS\_System\_Create()* to create the system. Following an arbitrary user program that may contain more API calls, routines are called to destroy the ring system object and shutdown the API.

```

/* user structure to describe nodes */
ACS_NODE nodes[4];
/* user structure for channels */
ACS_CHANNEL channels[4];
/* status of ACS API commands */
ACS_STATUS status;
ACS_SYSTEM ring;
/* build a ring of 4 WildForce boards */
for (int i=0;i<4;i++) {
    nodes[i].model = "WF4";
    channels[i].src_node = i;
    channels[i].src_port = 0;
    channels[i].dest_node = (i+1) % 4;
    channels[i].dest_port = 1;
}
/* must be first API call */
ACS_Initialize(argc, argv, &status);
ACS_System_Create(&ring, nodes, 4, channels, 4);
/* user program that accesses "ring" object */
ACS_System_Destroy(ring);
ACS_Finalize(); /*must be last API call */

```

**Figure 3: Code fragment for creating a "ring" system object.**

In addition to the static system creation illustrated in Figure 3, the API also has features for altering a system at runtime. Node and channels may be added or deleted after the creation of a system object through API calls. Note that multiple host processes are also possible in the API, but are not discussed in this paper for the sake of clarity.

### 3.2 Board Management

Once the system object has been created, the boards can be configured and controlled via the API. The code fragment in Figure 4 sends a bitstream to each board as specified in a configuration data structure. This configuration data structure includes information on which processing elements to configure as well as board-level configuration information such as crossbar switch settings. After configuration, the code fragment sets the clock speed, starts the clock, and then sends a reset signal. Finally, as described further in Section 3.3, the API provides calls for writing directly to the memory of a board. The second loop illustrates this call as well as the capability of sending interrupt signals of various types to each board.

Also included for board management are routines to query the board, including functions for readback and querying the clock speed. Current plans include building upon these routines to provide significant debugging capability within the API.

```

for (int i=0;i<4;i++) {
    /* send bitstream for each ACS board */
    ACS_Configure(config[i],i,ring,&status);
    /* set clock speed */
    ACS_Clock_Set(clock,i,ring,&status);
    /* start clock */
    ACS_Run(i,ring,&status);
    /* send reset signal */
    ACS_Reset(i,ring,&status);
}
for (int i=0;i<4;i++) {
    /* write initial data to each board's
    memory */
    ACS_Write(databuf[i],datalen[i],i,
        brd_addr[i],ring,&status);
    /* then send an interrupt (or inta) signal
    #1 to the board */
    ACS_Interrupt(i,1,ring,&status);
}

```

**Figure 4: Code fragment for configuring and writing to ACS boards**

### 3.3 Memory Access

The API contains routines for read and write access to the memories of all boards, local and remote. The use of the *ACS\_Write()* function is illustrated in Figure 4. Also included, to reduce network traffic, is a memory copy command that causes a block of memory to be copied from one node to another node rather than using a read followed by a write. These commands allow data to be sent outside of the channel-based system model; they put the burden of explicitly specifying all data movement solely on the developer. Nevertheless, they can be quite useful for sending initialization data or retrieving accumulated data directly from boards, operations for which the channel model is not naturally suited.

### 3.4 Streaming Data

The channel-based communication model requires the user to explicitly control only the initial entry and final exit of data from the system. Two primary commands, *ACS\_Enqueue()* and *ACS\_Dequeue()*, are required to control communication. The use of these commands is illustrated in Figure 5 where they control the data flow in the ring system that was created by the code fragment in Figure 3. The user can specify the behavior of each of the channels with additional API function calls, but is not required to do so. Such behavior can include the buffer size associated with a channel as well as the underlying communication mechanism.

```

/* use the ring to process the required number
of images */
for (int i=0;i<NUM_IMAGES;i++) {
    /* send image onto channel associated
    with port 0*/
    ACS_Enqueue(image[i],IMAGESIZE,0,
                ring,&status);
    /* get resulting image from channel
    associated with port 1 */
    ACS_Dequeue(result_image[i],
                RESULT_SIZE,1,ring,&status);
}

```

**Figure 5: Code fragment demonstrating channel-based communication**

### 3.5 Non-blocking Commands

An advanced feature of the API is a mechanism for issuing a set of non-blocking commands. Up to this point, the API functions discussed in the paper have all been blocking. For example, the *ACS\_Write()* commands in Figure 4 occur one after the other with the host program blocked during the execution of each write. Through the *ACS\_Request()* function, a user can specify a sequence of API functions to be executed as a set; this sequence is called a request and may include commands to read/write/copy memory, raise a reset line, or send an interrupt (the set of possible commands will be expanded). Once a request has been created, it can be committed to execution using *ACS\_Commit()*. *ACS\_Commit()* issues the commands, creates a handle, and returns control to the user. While those commands are executing, the user may perform other operations. Completion of the set of commands can be checked using *ACS\_Test()*, or can be waited upon in a blocking fashion using *ACS\_Wait()*. Once created, a request may be committed to execution multiple times. Benefits of the request mechanism include improved efficiency by overlapping user task execution with API task execution, combining multiple commands to reduce network overhead, and re-using command sequences to reduce API overhead.

### 3.6 Group Operations

The API also allows for certain commands to result in broadcasts of data rather than simple point-to-point transfers. By specifying *ACS\_ALL*, rather than an individual node number, the *ACS\_Configure()* command can become a broadcast to all nodes, allowing for a single command to configure all the

ACS boards in the system. This broadcast function would be quite useful on a homogeneous system, but would not be appropriate on a system with both SLAAC-1 and SLAAC-2 ACS boards. The group management functions in the API can be used to specify groups of nodes in the system. Group identifiers can be used to transform broadcasts into multicasts. For example, the SLAAC-1 boards could be programmed in one *ACS\_Configure()* call and the SLAAC-2 boards in another. These group operations do not affect channel-based communications.

### 3.7 The Interface on the Processing Elements

To this point, the focus of this paper has been on the host-side programming interface. The interface from the perspective of the processing elements is diverse, but not unlimited. Some obvious prerequisite capabilities on the part of the processing elements are the ability to observe reset lines and manipulate/observe interrupt lines. If memory is present, the processing elements can communicate with the host program by reading/writing that memory. Most importantly, the processing elements on a board (or a subset) must be able to read and write to a set of numbered (perhaps logical) FIFOs to support the channel-based communication model. Unlike the other capabilities, this feature may not be available on all systems and, if present, will not provide an unlimited number of FIFOs. An aspect of porting the API to any new board architecture is the provision of a FIFO mechanism; such a mechanism can, for example, be provided by a combination of memory reads/writes and interrupts. It is the responsibility of the API implementation itself to manage a limited number of physical FIFOs.

## 4 API Implementation

Control of the system across multiple computers is accomplished by using a single process on every computer. The host program serves this purpose on the computer on which it runs. Other computers in the system run a *control* process. A control process is responsible for executing commands initiated by API calls, monitoring the local adaptive computing board, and communicating with other control processes. Each of the control processes is single-threaded for simplicity; the host process is multi-threaded to allow for concurrent execution of the

host process and the control process functions on the computer where the host process is running.

To be of use in the adaptive computing community where the technology is changing rapidly and experimentation is of prime importance, any implementation of this API should be easily extensible. Such extensibility is accomplished by using an object-oriented approach to implementation, as is possible with C++. The identification of objects arises naturally from the specification of the API. As outlined earlier, the *system* object is a collection of *node* and *channel* objects. A channel object represents a FIFO queue connecting one node to another node. A channel can contain a buffer and settings specific to control flow on this FIFO. A node object represents a computational device in the system.

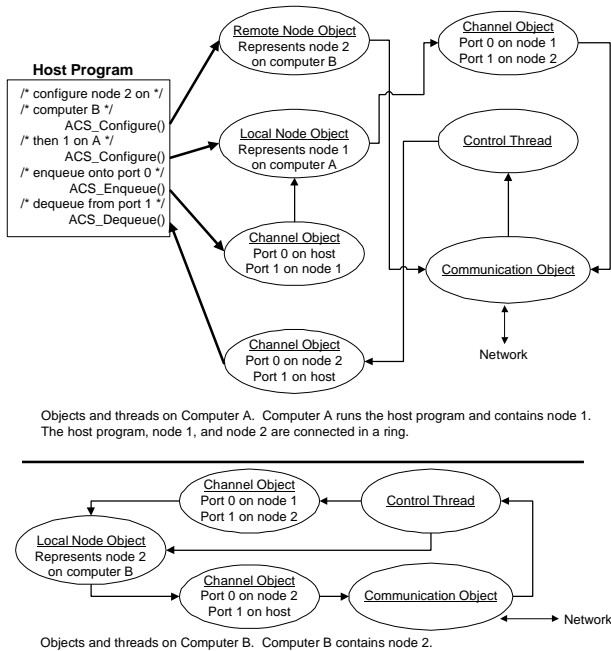
Two objects not directly viewed or manipulated by the user are the *communication* object and the *world* object. The communication object accomplishes all communication between processes on different computers. Different communication objects can be used to allow functionality in a heterogeneous network. The only communication object in the current implementation uses MPI for communication. MPI is itself a standard API for communication between parallel processes [5]. It was chosen as the first communication object in this implementation because of its ubiquitous nature and the availability of high-performance implementations [3]. The world object is used to encapsulate and maintain information about the computing environment in which the API is running. For example, the world object contains a list of all the control processes and host processes running as well as how to communicate with those processes. Further, it contains a list of all the adaptive computing boards managed by each control process. Future extensions to the API include a collection of routines to query the world object so that the user can dynamically create system objects based on which types of boards are available.

The core of the API implementation is written as operations on these objects. The classes associated with these objects, including virtual function definitions, are defined as part of the core implementation. By taking advantage of inheritance and encapsulation, the distinction between local and

remote boards is easily hidden, and new types of boards and communication systems can be seamlessly included. For example, to extend the API to allow control of a new board, a developer just creates a class that inherits the node object and implements all of the virtual functions to allow for control of the new board; the rest of the API implementation remains unchanged.

The need for high performance from the API cannot be lost in the quest for an easily extensible implementation. Of particular importance is the need to avoid introducing unnecessary overhead into the implementation. A potential pitfall in any interprocessor communication system is the introduction of multiple copies of large buffers. The specific method for avoiding buffer copies, particularly ones implicit in calls to an underlying communication system, are specific to the type of communication object used. The API implementation, outside of the communication object, will not introduce extra copies of large buffers. Further savings in overhead can be accomplished by recognizing when commands or data are being sent to a local board as opposed to a board on a remote computer. Fortunately, the object-oriented implementation can accomplish this by simply providing a *remote* node object and a *local* node object that each inherit from the node object. Actions by the API are performed on a node object without regard to local/remote considerations, but the correct node functions are called automatically depending on the whether the node is local or remote. This results in a logically simple implementation that introduces no unnecessary overhead for local operations.

An illustration of the objects in the implementation and their interaction in a typical API operation is given in Figure 6. The objects in this figure represent a host program and node 1 executing on computer A and node 2 executing on Computer B; these objects are connected by three channels to form a ring communication structure. The example host program illustrates the basic communications that occur when the nodes are configured as well as the communications that occur when communications are initiated in the ring.



**Figure 6: Description of objects and threads on two computers.**

## 5 Future Work

Proposed future extensions to the API and its implementation include high-speed networking, support for RTR, support for debugging and performance monitoring, system level management of multiple programs, and integration with systems such as JHDL [1].

The current implementation of the API uses MPI [5] for interprocessor communication; the available high-performance versions of MPI are suitable for most applications [3]. However, some applications require a tighter coupling of the communication board to the computational elements. Some embedded systems designs have communications processors on the same boards as the computational elements. While the API and its implementation are designed such that the host processes will likely require the support of a traditional OS, the control processes do not require the full services of a traditional OS. The next step of the API implementation is to carry out the functions of a control process on two PCI cards, a Myrinet interface card and an ACS card, without the intervention of the CPU or OS.

Also of significant importance is the support for efficient run-time reconfiguration (RTR) of adaptive

computing devices. As currently specified, the API requires that a host process initiate any device reconfiguration via an *ACS\_Configure()* function call. Further, the current implementation requires that every *ACS\_Configure()* function call result in the configuration being sent by the host process to the device to be reconfigured. Future versions will address two modes of efficient RTR. The first mode is host-initiated RTR; in this mode, the host process is always the initiator of reconfiguration. Efficient support for this mode will be provided by allowing control processes to cache several configurations. When *ACS\_Configure()* is called, the cache will be checked for the desired configuration and if the configuration is present, then the network transfer can be avoided. The second mode, data-driven RTR, is more complex to support. In this mode, reconfiguration is driven by the data that is encountered. For example, different filters may be downloaded depending on the light-levels encountered in an image. The host process cannot drive such reconfiguration. In this case, the programmer must be able to describe more complex mechanisms to the control processes, perhaps finite state machines, to manage the reconfiguration. Note that such a mechanism will be required for single board/computer systems as well as the multi-board/computer systems addressed in this paper.

As noted in the integration, no tools exist for performance monitoring and debugging of applications on distributed adaptive computing systems. Board-level tools, such as BoardScope, can provide valuable support for debugging an application on a single board [4]. Extensions to the API to support debugging and performance monitoring can facilitate the task of creating integrated system-level tools. For example, the current API already includes a function that allows for readback of any node on the system. Other types of support could include functions to query the status of the channels, channel throughput statistics, and usage statistics on processing elements.

Also of concern is the management of multiple concurrent programs. For example, if two processes try to configure the same adaptive computing board and no management software intervenes, then both processes may fail in unpredictable ways. The immediate solution to this problem is simply to lock one of the processes out of adding that node to its

system object. A time-sharing feature, however, is likely to be more useful and more interesting. For example, on systems that do not support faster RTR, it would be useful to checkpoint jobs that have been running a long time in favor of new jobs. On systems that support fast RTR, it should be possible to context switch between jobs in a fashion similar to traditional operating systems. The combination of fast RTR and efficient system software would provide a useful virtual adaptive computing system.

The API can serve as the target of higher-level tools as well as a direct implementation language. One such higher-level tool, JHDL, provides a portable, integrated environment for programming a single adaptive computing board as well as the interaction of that board with a host system [1]. A developer can use JHDL to write a host program, program the FPGAs, and debug the combined application. JHDL does not, however, currently provide support for system level applications. Because of its use of an existing programming environment, Java, JHDL could be expanded to include support for the large, distributed systems targeted by the ACS API. Such support can be more easily provided if JHDL can use the ACS API as a target. The use of the ACS API as a target, as well as its extension, can be facilitated by exposing the objects and functions of the underlying implementation in a formal way to developers.

## 6 Concluding Remarks

The API specification, source code, and supporting documents can be accessed on the Virginia Tech Configurable Computing Lab WWW site at <http://www.ee.vt.edu/~ccm>. The first version of the API implementation is complete. The first version uses MPI for interprocessor communication. Near-term plans include porting the ACS API implementation to Linux and adding node objects for the SLAAC-1 and SLAAC-2 ACS boards.

The ACS API and the implementation have applications outside of the control of adaptive computing boards. In particular, they can be used to control a heterogeneous collection of devices in an embedded system. Instead of configuring an adaptive computing device, `ACS_Configure()` could download a configuration to microcontroller or a FFT board. Because of the object-oriented nature of the implementation, this functionality can be

accomplished via the addition of another node object class. To demonstrate this functionality and for use in rapid prototyping, the current implementation has a *dummy* node object class that defines all of the virtual functions for a microprocessor. A developer can insert arbitrary C code in this dummy node so that it can emulate the behavior of any device; a system-level application can be quickly constructed and dummy nodes gradually replaced to increase performance without changing the functional behavior of the system.

## 7 References

1. P. Bellows and B. Hutchings, "JHDL-An HDL for Reconfigurable Systems," *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, April 1998.
2. D. S. Katz, T. Cwik, B. H. Kwan, J. Z. Lou, P. L. Springer, T. L. Sterling, and P. Wang, "An Assessment of a Beowulf System for a Wide Class of Analysis and Design Software," *Advances in Engineering Software*, Vol. 29(3-6), pp. 451-461, 1998.
3. M. Lauria and A. Chien, "MPI-FM: High Performance MPI on Workstation Clusters," *Journal of Parallel and Distributed Computing*, Vol. 40(1), pp. 4-18, 1997.
4. D. Levi and S. A. Guccione, "BoardScope: A Debug Tool for Reconfigurable Systems," In John Schewel, ed., *Configurable Computing Technology and its Use in High Performance Computing, DSP, and Systems Engineering*, Proc. SPIE Photonics East, Bellingham WA, pp. 239-346, 1998.
5. Message Passing Interface Forum, "MPI: A Message-Passing Interface Standard", *International Journal of Supercomputing Applications*, Vol. 8(3/4), 1994.
6. B. Schott, C. Chen, S. Crago, J. Czarnaski, M. French, I. Hom, T. Tho, and T. Valenti, "Architectures for System-Level Applications of Adaptive Computing," *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, 1999.
7. V. Michael Bove, Jr. and John A. Watlington, "Cheops: A Reconfigurable Data-Flow System for Video Processing", *IEEE Transactions on Circuits and Systems for Video Technology*, 5, April 1995, pp. 140-149.
8. John A. Watlington and V. Michael Bove, Jr., "A System for Parallel Media Processing",

presented at the *Workshop on Parallel Processing in Multimedia*, held in conjunction with IPPS '97, April 1, 1997.