Embedded Processor Virtualization for Broadband Grid Computing

Richard Neill Luca P. Carloni
Dept. of Computer Science, Columbia University
New York, NY, 10027
Email: {rich,luca}@cs.columbia.edu

Alexander Shabarshin Valeriy Sigaev Serguei Tcherepanov
Cablevision Systems
Bethpage, NY, 11714
Email: {ashabars,vsigaev,stcherep}@cablevision.com

***Abstract*—We implemented and evaluated a heterogeneous system architecture that combines a traditional computer cluster with a broadband network of embedded set-top box (STB) devices to provide a distributed computing platform for parallel applications. Our prototype system for broadband grid computing leverages the recent dramatic progress in computational power of STBs. It includes a complete head-end cable system based on the Tru2way standard, a DOCSIS-2.0 network, and an implementation of the Open MPI library running on the STB embedded operating system across 128 devices. An important contribution of our work is a novel method for the virtualization of a large collection of embedded processors within a managed broadband network. This enables the embedded processors to transparently inter-operate with servers in the computer cluster using the message-passing model. To evaluate the interoperability, performance, and scalability of our system we completed a set of experiments with the standard IMB MPI benchmark suite as well as two real parallel applications. The experimental results confirm that there is an important convergence trend between traditional computing and embedded computing and that a broadband network of embedded processors is a promising new platform for a variety of computationally-intensive and data-intensive grid applications.**

***Keywords*-message-passing interface (MPI); embedded computing; set-top box; broadband network.**

I. Introduction

The Information Technology industry is experiencing two major trends. On one hand, computation is moving away from traditional desktop and department-level computer centers towards an infrastructural core that consists of many large and distributed data centers with high-performance computer servers and data storage devices. These large-scale centers provide all sorts of computational services to a multiplicity of peripheral clients, through various interconnection networks. On the other hand, the increasing majority of these clients consists of a growing variety of embedded devices, such as smart phones, tablet computers and television set-top boxes (STB), whose capabilities continue to improve. Multiple Service Operators (MSO), such as cable providers, are an example of companies that drive both the rapid growth and evolution of large-scale computational systems and the deployment of an increasing number of increasingly-powerful embedded processors.

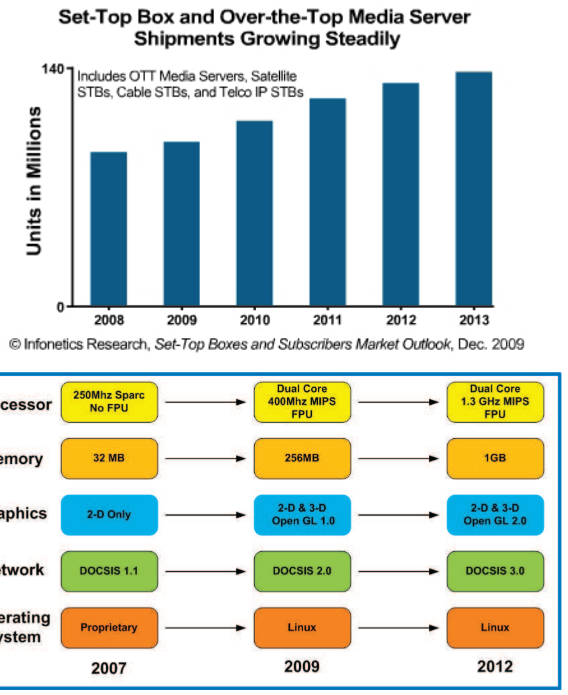


Figure 1. Worldwide STB shipment forecast and STB technology evolution.

The bar chart of Fig. 1 shows the growth in STB shipments between 2008 and 2013 [1], while the bottom diagram illustrates their technology evolution: in the span of less than five years, a STB has evolved from hosting a single 300Mhz processor without floating-point unit, one 2-D graphics unit and just 32MB of memory to a projected multi-core system-on-chip (SoC) with two 1.3-Ghz processing cores, multiple video and 3D graphics accelerators, and 1GB of memory. Indeed, the *heterogeneous multi-core SoC* is the emerging computational platform to implement embedded systems across a variety of consumer-electronics domains from mobile phones to interactive television [2], [3]. Furthermore, the performance gap between embedded SoCs and the high-end processors found in data centers continues to decrease. Meanwhile, MSOs drive also the evolution of certain aspects of grid computing focused on supporting consumer-electronic applications because they can leverage their legacy position in managing a *dedicated* *broadband network* of millions of geographically-distributed embedded devices.

Our work is motivated precisely by the idea that the combination of the technology trends in embedded systems, data centers, and broadband networks opens the way to a new class of computer systems, *broadband grid computing*, whose potential application domains include: ubiquitous on-demand content access, social networking, large-scale data mining and analytics, and even some types of high-performance computing. In particular, we propose a heterogeneous distributed system architecture which combines a traditional computer cluster with a cluster of embedded processors interconnected through a broadband network to offer massive computational potential (and, potentially, energy and cost efficiency). While the discussion of the possible business models is outside the scope of this paper, our technical results offer a promising new direction to augment existing grid computing architectures.

We have implemented a prototype small-scale version of our proposed system where the Computer Cluster features nine high-end blade servers and the Embedded Cluster consists of 128 STBs. The two clusters are interconnected through the broadband network of a complete head-end cable system (as described in Section II). While the cable system remains fully operational in terms of its original function (e.g. by distributing streaming-video content to the STBs which render it to their displays), it is possible to simultaneously and effectively execute other parallel applications by leveraging the additional computation resources that are available in the STB multi-core processors. Specifically, we ported the OPEN MPI software library, i.e. the *de facto* standard for implementing parallel applications with the *message-passing interface* on computer clusters and supercomputers, to our heterogeneous system. As discussed in Section III, this porting posed important challenges in terms of resource management and scalability. We addressed these challenges by performing a virtualization of the embedded processors that allows them to transparently inter-operate with the computer cluster using the message-passing model (Section IV). We also developed an implementation of the complete OPEN MPI runtime environment and software library which is optimized for our embedded devices. Our experimental evaluation includes multiple results (Section V): first, we demonstrated that the system can execute the complete set of Intel MPI IMB benchmarks. Then, in order to gain further insight into the relative performance scaling of the Embedded Cluster versus the Computer Cluster, we run two important parallel applications: ray tracing and multiple sequence alignment. The experimental results confirm the important convergence trend between traditional computing and embedded computing and support the case for broadband grid computing while indicating the avenues for future work to improve our proposed system architecture.

II. THE SYSTEM ARCHITECTURE

Fig. 2 provides a complete view of the system that we designed and implemented. It is composed of four main subsystems.

Computer Cluster. The Computer Cluster consists of a traditional network of nine blade servers and Network Attached Storage (NAS). Each blade has two quad-core 2.0GHz Xeon processors with 32GB of memory and 1Gb/s Ethernet interface. Each processor runs Debian Linux. One of the nine blades acts as Master Host, i.e. is dedicated to the OPEN MPI runtime management and is the master server for the Computer Cluster and the Embedded Cluster host nodes. These use NFS to mount the 2TB Sun storage array which provides a remote common file-system partition to store both applications and data for each of the executing MPI processes across both clusters. The master system also hosts the virtualization software to map the embedded processors into the runtime environment of the Computer Cluster.

Embedded STB Cluster. The Embedded Cluster consists of 128 Samsung SMT-C5320 set-top boxes (STB) that are connected with a radiofrequency (RF) network for data delivery using MPEG and DOCSIS transport mechanisms. The Samsung SMT-C5320 is an advanced (2010-generation) STB featuring a dual-core SoC with a Broadcom MIPS 4000 class processor, a floating-point unit, dedicated video and 2-D/3-D-graphics processors with OpenGL support, 256MB of expandable system memory, 64MB Flash memory, and many network transport interfaces (DOCSIS 2.0, MPEG-2/4 and Ethernet). Indeed an important architectural feature of modern STBs is the multi-core architecture design which allows the MIPS processor, graphics/video processors, and network processors to operate in parallel over independent buses. Hence, user-interface applications (such as the electronic programming guides) can execute in parallel with any real-time video processing. *Indeed, it is the growing parallel-computing capability of the emerging SoC architectures for STBs that enables the execution of applications outside the realm of interactive-TV, thus opening the opportunity for large-scale broadband grid computing that we are pursuing with our work.*

Digital Cable Head-End. This is responsible for controlling the Embedded Cluster devices and providing all interactive television services including: electronic program guide, user-interface, video-on-demand (VOD), and the delivery of MPEG-2 videos. Our digital head-end supports the current generation of STBs based on the Cablelabs Tru2way standard [4] and is a *scaled-down but complete implementation* of a modern digital DOCSIS-based broadband cable system in-use at today's largest MSOs. As shown in Fig. 2, its core components include: 1) the Tru2way Object Carousel for MPEG-2 delivery of Embedded Cluster applications and Tru2way-standard STB signalling; 2) two Linux hosts for TCP/IP DHCP and TFTP network services, which are re-

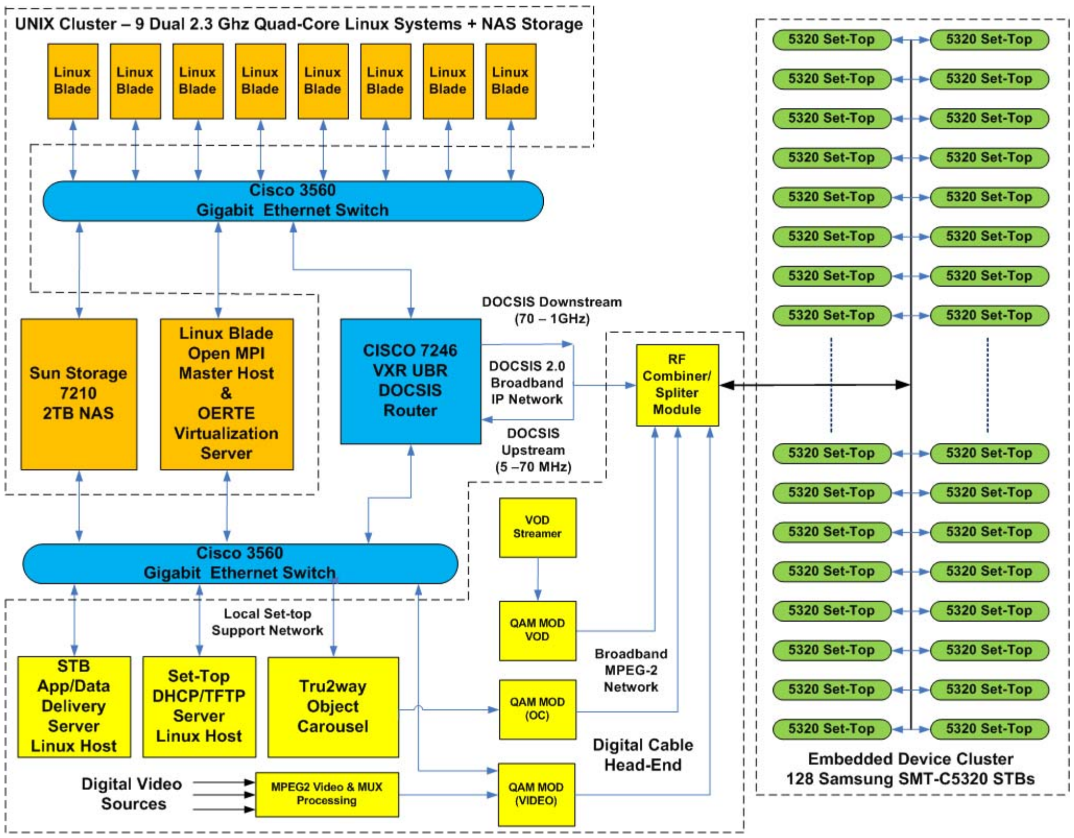

UNIX Cluster – 9 Dual 2.3 Ghz Quad-Core Linux Systems + NAS Storage
Linux Blade
Cisco 3560 Gigabit Ethernet Switch
Sun Storage 7210 2TB NAS
Linux Blade Open MPI Master Host & OERTE Virtualization Server
CISCO 7246 VXR UBR DOCSIS Router
DOCSIS Downstream (70 – 1GHz)
DOCSIS 2.0 Broadband IP Network
DOCSIS Upstream (5 –70 MHz)
RF Combiner/ Spliter Module
VOD Streamer
QAM MOD VOD
QAM MOD (OC)
QAM MOD (VIDEO)
Broadband MPEG-2 Network
Cisco 3560 Gigabit Ethernet Switch
Local Set-top Support Network
STB App/Data Delivery Server Linux Host
Set-Top DHCP/TFTP Server Linux Host
Tru2way Object Carousel
Digital Video Sources
MPEG2 Video & MUX Processing
Digital Cable Head-End
5320 Set-Top
Embedded Device Cluster 128 Samsung SMT-C5320 STBs

Figure 2. The proposed heterogeneous system architecture for broadband grid computing.

quired for assigning system-wide IP addresses and DOCSIS cable-modem configuration data to all Embedded Cluster devices; 3) a HTTP application/data server that supports interactive television services via TCP/IP over the DOCSIS network; 4) support for MPEG-2 video sources that are multiplexed and grouped into digital channels, including a single channel for VOD streams; and 5) a RF distribution and combining network that utilizes a Cisco QAM modulator device to translate digital input signals from the carousel, multiplexed MPEG sources, and VOD server, into modulated QAM256 RF frequencies, which can be combined with the DOCSIS router RF output to feed the broadband network of STBs.

DOCSIS is a standard broadband-network technology for TCP/IP over RF cable [5]. It provides for an inter-operable RF modem, based on TDMA protocols organized in a star topology connecting the central router and the STBs. The SMT-C5320 DOCSIS 2.0 TCP/IP and MPEG-2 transport stream interfaces use quadrature amplitude modulation (QAM) protocols for transmitting and receiving signals on North American digital cable systems. Devices on DOCSIS share access to the network, as arbitrated by the central router, and operate effectively at up to 27Mbps in the downstream direction (towards the STB) and 27Mbps in the upstream direction (towards the cluster). The MPEG-2 interface is primarily used for decoding video programs, but can also receive applications or data delivered via the carousel. This data is sent from the head-end at regular intervals over MPEG-2 directly into a QAM device where it is modulated onto the RF cable plant at a specified frequency for STB reception. Broadcast applications are STB executables or data that are simultaneously available to all STBs connected to the broadband network. A STB device tunes to a specific channel frequency and receives the application/data of interest according to the Tru2way protocol. The carousel may also deliver Tru2way signalling and other forms of data over DOCSIS as multicast group messages following the DOCSIS Set-top Gateway, or DSG protocol [6]. *In our prototype system this data-delivery mechanism is used to control the STB boot-up and user-interface applications.*

Network. The system network is a managed dedicated broadband network which is divided into three IP subnets to isolate the traffic between the DOCSIS-based broadband Embedded Cluster network, the Computer Cluster network, and the digital cable head-end. Its implementation is based on two Cisco 3560 1Gb/s Ethernet switches and one Cisco 7246 DOCSIS broadband router. The upper switch in Fig. 2 interconnects the 8 blades along with the NAS and master host. The lower switch aggregates all the components on the head-end subnetwork. The broadband router has 1Gb/s interfaces for interconnection to the Computer Cluster and head-end networks and a broadband interface for converting between the DOCSIS network and the Ethernet backbone. Each broadband router can support over 16,000 STBs, thus providing large scale fan-out from the Computer Cluster to the Embedded Cluster. While in a normal cable system the Computer Cluster and the digital cable head-end do not necessarily need to share traffic, we connected them over Gigabit Ethernet because this enables, for instance, the execution of MPI collective operations among the Computer Cluster and Embedded Cluster nodes in a seamless way.

In summary, while being representative of a real cable system, our prototype system allows us to execute MPI application processes simultaneously on both the Computer Cluster blades and the Embedded Cluster processors under realistic operations scenarios. For instance, we can execute multiple workloads such as the IMB benchmarks and the MSA and Ray Tracing applications on the embedded processor, while the rest of the components in the STBs, and particularly the MPEG video processing chain, are busy providing streaming-video content. *A key element of our system is the managed broadband network, which not only enables the heterogeneous system implementation, but it offers also a dedicated and massively-scalable infrastructure that can be leveraged for broadband grid computing.*

III. Open MPI: Basics and Challenges

Open MPI is an open-source implementation of the Message Passing Interface (MPI) library for development of parallel applications on distributed memory computer architectures [7]. Open MPI is the result of merging and combining three main previous MPI implementations and is currently among the most popular library for high-performance computing applications. Its design is centered on the Modular Component Architecture (MCA), which provides a flexible and configurable environment for design-time development and run-time installation of various software frameworks [8], [7]. An MCA framework is a construct that is created for a single, specific tasks and provides a public interface. Examples of tasks are the launch of processes on the local host or the execution of collective operations. A framework uses the MCA services to find and load components at run-time. An MCA component is a self-contained implementation of a framework's interface, which can be inserted into the Open MPI code base at run-time and/or compile-time. An MCA module is an instance of a component.

Open MPI is a large project with many different subsystems. Fig. 3 shows the three major ones, which build on each other according to a layered structured. OMPI is the top layer and contains the actual implementation of the MPI application program interface. The *Open Runtime Environment (ORTE)* is responsible for managing the launch and runtime lifecycle of the parallel processes of a given MPI application. Both OMPI and ORTE rely on the underlying

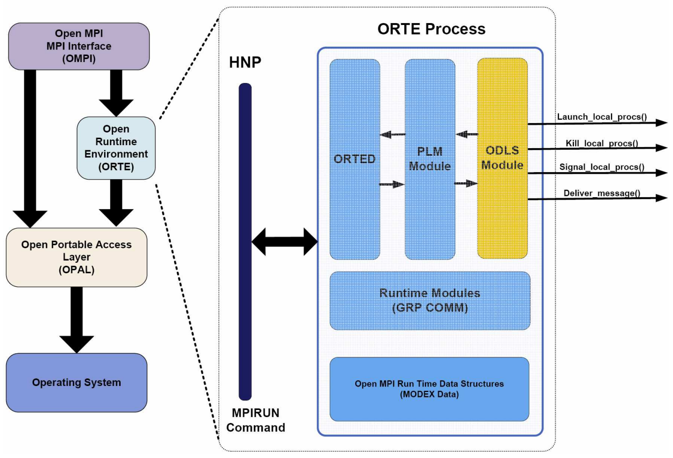

Open MPI MPI Interface (OMPI)
Open Runtime Environment (ORTE)
Open Portable Access Layer (OPAL)
Operating System
ORTE Process
HNP
ORTED
PLM Module
ODLS Module
Launch_local_procs()
Kill_local_procs()
Signal_local_procs()
Deliver_message()
Runtime Modules (GRP COMM)
Open MPI Run Time Data Structures (MODEX Data)
MPIRUN Command

Figure 3. The three Open MPI subsystems and the primary ORTE modules.

Open Portability Layer (OPAL), which contains the utility and "glue" code needed to integrate the higher-layer modules with the native (host) operating system.

The Open Runtime Environment (ORTE). In the Open MPI 1.4.2 release, the ORTE subsystem has 14 distinct frameworks, which offer a flexible and highly-configurable runtime environment by supporting various tasks including: managing process mapping or affinity, launching of MPI processes onto physical processing cores, managing of MPI cluster-wide process lifecycle during execution, error messaging, redirection of process I/O, and process-wide group communications facilities. Thanks to the MCA, ORTE framework components may be replaced with different implementations, all dynamically configurable at runtime. Fig. 3 shows the primary ORTE modules. The execution of a MPI job is initiated by running the `mpirun` command on a computer in the cluster, which therefore becomes the *host node process (HNP)*. As the newly-designated master node, the HNP initiates one or more ORTE Daemon (ORTED) processes on each client host node supporting ORTE, through a remote execution protocol (e.g. RSH or SSH), or a specialized process-launcher communication protocol. Each ORTED process communicates with the Process Lifecycle Management (PLM) module whose functions include controlling the actual mapping of MPI processes to the processing cores and managing their complete execution. For each process, this includes runtime initialization, application launch, signalling, message delivery, and termination. The PLM performs these operations in conjunction with the *ORTE Daemon Local Launch System (ODLS)*, a module which defines an interface contract for each of them and launches local processes on the MPI host node. Finally, the ORTE framework uses `grpcomm`, a group communication module, to distribute message information among the peer ORTE client hosts and the HNP.

Challenges in Porting Open MPI to Embedded Devices. One important message operation required to launch any OPEN MPI application is the sharing of per-process module information, or *module exchanges (modex)*. This is performed by `grpcomm` which executes an `allgather()` operation as follows: each ORTE client gathers local process information and sends it to the HNP, which assembles the information for all processes running on every client and then re-distributes it to all clients. The operation requires fairly high network bandwidth and fairly large memory on each host. Further, these requirements scale up with the number of hosts in the cluster. This represents a significant *scaling challenge* for the effective utilization of OPEN MPI in a heterogeneous computing system that aims at leveraging millions of embedded devices as the one that we envision. For instance, if we assume a typical modex data-structure of 500 bytes, and a cluster with P hosts and N MPI processes per host, then each host must store $P*N*500$ bytes of data. In a large system where P can be of the order of millions, even if we have a small number N of processes, the memory requirements to store modex data could easily exceed 1GB per embedded device. This is an unrealistic requirement for today's embedded SoCs. But even if future SoC architectures were able to accommodate it, it would require to consume a significant data-transfer time simply to copy into the host memories, thus undermining the performance gains from parallelizing the computation. In the next section we discuss how we addressed these challenges. In particular, we leveraged the OPEN MPI MCA architecture to modify the functionality of the ORTE subsystem by replacing the ORTE ODLS component module with a new ODLS module. The new module is interfaced with a newly-developed embedded version of the ORTE framework to support the virtualization of the embedded processors. This allows us to decouple the embedded-process modex management so that all ORTE modex operations are performed at the server side.

IV. EMBEDDED PROCESSOR VIRTUALIZATION AND EMBEDDED SOFTWARE OPTIMIZATION

We first describe the characteristics of the embedded software environment of an STB device and then we present our solutions to port OPEN MPI to a broadband network of STBs. In particular, we completed two new implementations of ORTE, one to support the virtualization of embedded processors to the Computer Cluster and the other as part of the OPEN MPI software optimization for embedded computing.

Embedded Software Environment. The STB software environment is based on an embedded version of Linux with a reduced footprint of only 16MB. This was obtained by minimizing the size of the required kernel, utilities, and associated libraries, which are resident in the Flash memory. For example, the embedded Linux operating system does not include facilities for desktop window systems, development tools, multiple shell environments, or utility packages typical of a full-package Linux distribution. The shell, provided

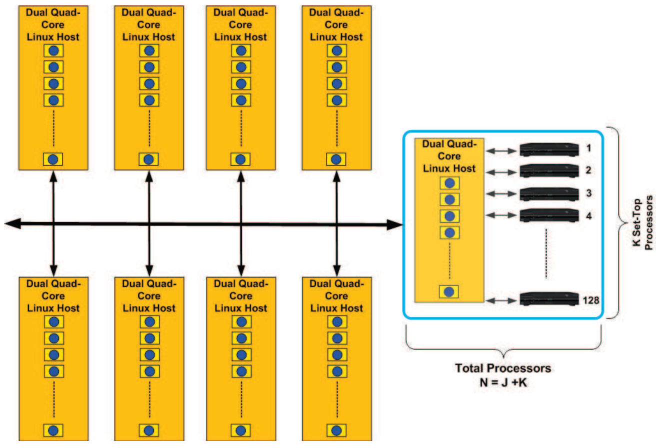


Figure 4. Virtualization of the STB embedded processors.

by BusyBox [9], consists of over 100 common Linux and GNU utilities, whose implementation is highly optimized for embedded devices, requiring only 400KB. The kernel, based on the Linux v2.6 distribution [10], includes support for threads, BSD socket interfaces, network services (NFS, DHCP, Telnet, etc.), and standard GCC libraries. This is sufficient for developing and executing sophisticated multi-threaded Linux applications and MPI applications.

The STB initialization process occurs as follows: during power-on or a reboot, the STB Flash-based boot-loader starts the Linux kernel. This executes all initialization scripts found in the `/etc/init.d` and `/etc/rc.d` directories. All network interfaces are configured using DHCP and the remote file-systems are NFS-mounted from the NAS Storage. To support the development of interactive television applications, the STB also initializes a Java Virtual Machine and a set of Java class instances. After the execution of all STB initialization scripts, a start-up script in `/etc/rc.local` runs a special Linux application which provides a runtime-environment manager required for the STB interoperation with the Computer Cluster Open MPI environment.

Embedded Processor Virtualization. A critical step in the design of our heterogeneous system for broadband grid computing is the virtualization of the STB embedded processors in the context of the OPEN MPI ORTE and the TCP/IP networking environment. First, we mapped the Embedded Cluster network of embedded processors into the processor domain of the Computer Cluster system. Second, we mapped the execution of the OPEN MPI process running on the Embedded Cluster into the Computer Cluster. We did so by implementing those software components which are necessary to support the mapping of the runtime and lifecycle management for these OPEN MPI processes into the standard OPEN MPI runtime software environment running on the Computer Cluster. Fig. 4 illustrates the high-level architecture of the resulting implementation. On the right-end side, K embedded processors are mapped into a

Linux host, which contains J processors. From the external viewpoint of other Linux nodes in the Computer Cluster, the host system becomes a heterogeneous multi-processor system with a total of $N = J+K$ processing cores. The host system is virtualized in the sense that the Computer Cluster nodes are unaware of the Embedded Cluster and simply view the virtualized host as a single N-processor Open MPI compute node. As a result, the overall heterogeneous system equipped with K additional virtualized embedded processors may be utilized in any of three possible configurations: 1) a 129 node heterogeneous cluster consisting of a single Linux master node plus 128 Embedded Cluster processors; 2) a Computer Cluster consisting of the single Linux master node and eight Linux compute nodes; or 3) a heterogeneous cluster consisting of the Linux master node, the eight Computer Cluster nodes, and the 128 Embedded Cluster nodes.

In order to complete the embedded-processor virtualization, we integrated the STB process runtime management environment into the OPEN MPI process runtime environment by implementing a new software framework. This framework provides protocol transformation and adaptation between the two heterogeneous runtime environments. Specifically, we developed a new version of the OPEN MPI ORTE that we called *Open Embedded Runtime Environment (OERTE)*. It consists of four components: 1) a new ODLS module, 2) an OERTE server, 3) an OERTE embedded client that runs on embedded STB devices, and 4) Open MPI Embedded, an optimized Open MPI library for the resource-constrained embedded STB devices.

The New ODLS Module. As shown in Fig. 3, the original ORTE contains a ODLS module, which is responsible for: the launch/termination of an OPEN MPI process on the local host, various signaling, and the managing of modex-entry communications between the launched process and the ORTED during the initial phases of its execution. In particular, to manage a local process the ODLS external interface contract defines four *primary functions*: the process launch is initiated with `Launch_local_procs`, which specifies how many processes (along with their input arguments) must start on the computer host; abnormal termination is obtained by sending `Kill_local_procs` to all executing processes; Linux process signals, such as `SIGSTOP` are passed to all executing MPI processes with `Signal_local_procs`; and, finally, the modex data-exchange operations are performed with `Deliver_message`.

In our implementation we replace the standard ODLS module with a *New ODLS Module* at runtime, as shown in Fig. 5. The new module implements the primary functions by forwarding all requests over a TCP/IP socket interface to a new external *OERTE server*. This is a component which performs the ODLS functions in the context of the distributed embedded system environment and returns all responses in a manner that is equivalent to the original ODLS module function implementation as if these functions

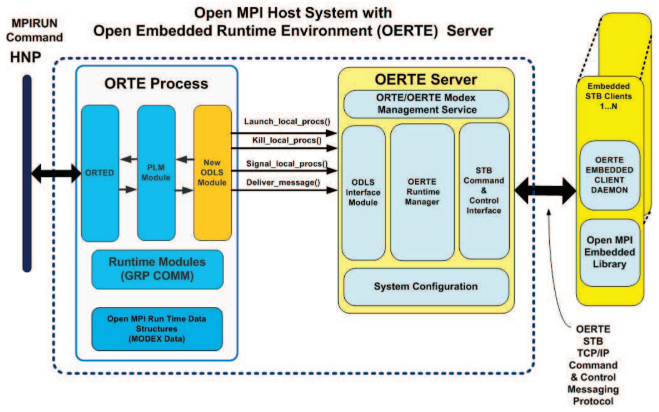

MPIRUN Command
HNP
Open MPI Host System with
Open Embedded Runtime Environment (OERTE) Server
ORTE Process
ORTED
PLM Module
New ODLS Module
Runtime Modules (GRP COMM)
Open MPI Run Time Data Structures (MODEX Data)
Launch_local_procs()
Kill_local_procs()
Signal_local_procs()
Deliver_message()
OERTE Server
ORTE/OERTE Modex Management Service
ODLS Interface Module
OERTE Runtime Manager
STB Command & Control Interface
System Configuration
Embedded STB Clients 1...N
OERTE EMBEDDED CLIENT DAEMON
Open MPI Embedded Library
OERTE STB TCP/IP Command & Control Messaging Protocol

Figure 5. The OERTE Server Architecture.

were executed locally. Response messages are sent from the OERTE server to the New ODLS Module over a second TCP/IP socket. In this manner the New ODLS Module provides a bi-directional bridge between the standard ORTE environment operating on the Computer Cluster and the external OERTE server, which is optimized to manage the Embedded Cluster runtime environment.

Open Embedded Runtime Environment (OERTE) Server. The OERTE server is a stand-alone multi-threaded Java server that acts as a runtime management server for the Embedded Cluster. It executes alongside the ORTE process and transforms ODLS function calls into operations that can be executed on the Embedded Cluster. As shown in Fig. 5, the OERTE server architecture includes an ODLS Interface module, which listens on the sender socket for the command functions from the New ODLS Module. As these are received, they are converted into an internal Java object representation that is passed to the Runtime Manager. This works in conjunction with the Command & Control Interface module to coordinate the sequencing of message deliveries over TCP/IP to the client daemons running on the Embedded Cluster nodes. Responses from these processes are handled in a similar way: a client initiates a TCP/IP connection to the server and delivers response messages to the Command & Control Interface; these are converted back by the Runtime Manager to the appropriate ODLS-response format for transmission to the New ODLS Module for further processing before returning to the PLM module and finally the ORTED process.

The OERTE server has also a subsystem for generation and processing of modex data, which contains all the Internet address and port number information associated with the Computer Cluster and Embedded Cluster devices. This information is required by MPI processes to communicate with one another during execution of point-to-point or collective communication operations. The modex data is shared with all MPI processes through a modex-exchange operation that is coordinated among all cluster ORTED processes and the HNP.

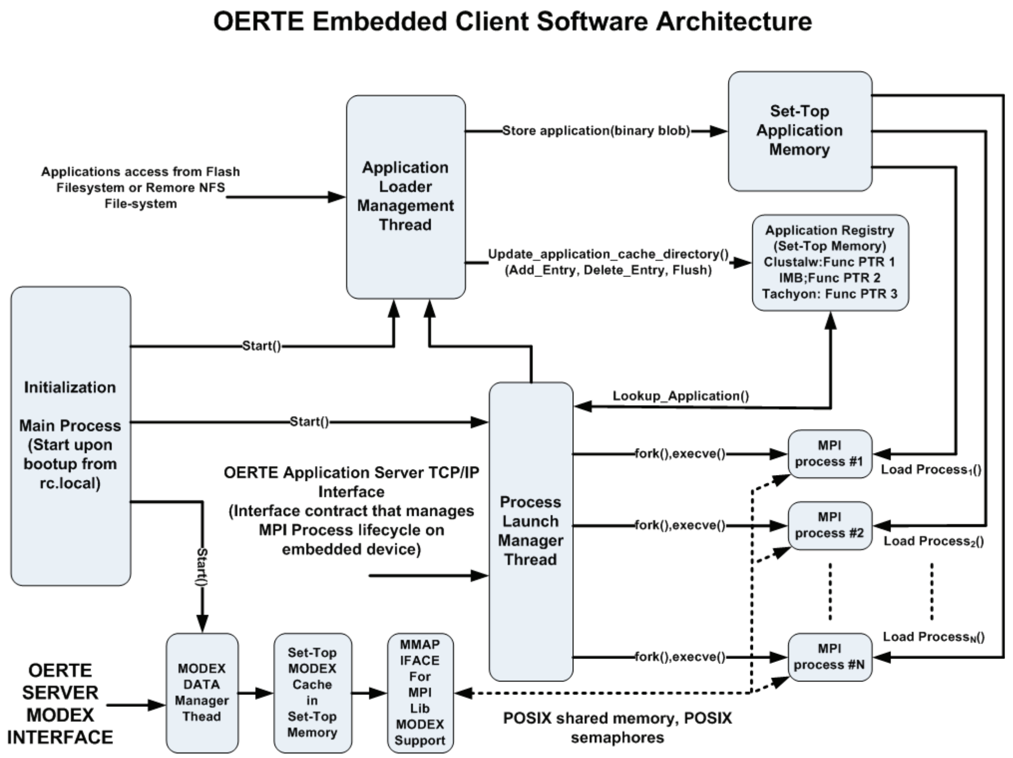

OERTE Embedded Client Software Architecture
Applications access from Flash Filesystem or Remore NFS File-system
Application Loader Management Thread
Store application(binary blob)
Set-Top Application Memory
Update_application_cache_directory() (Add_Entry, Delete_Entry, Flush)
Application Registry (Set-Top Memory) Clustalw:Func PTR 1 IMB;Func PTR 2 Tachyon: Func PTR 3
Initialization Main Process (Start upon bootup from rc.local)
Start()
Start()
Lookup_Application()
OERTE Application Server TCP/IP Interface (Interface contract that manages MPI Process lifecycle on embedded device)
Process Launch Manager Thread
fork(),execve()
MPI process #1
Load Process1()
MPI process #2
Load Process2()
MPI process #N
Load ProcessN()
Start()
OERTE SERVER MODEX INTERFACE
MODEX DATA Manager Thead
Set-Top MODEX Cache in Set-Top Memory
MMAP IFACE For MPI Lib MODEX Support
POSIX shared memory, POSIX semaphores

Figure 6. The software architecture of the OERTE embedded client.

In order for the Embedded Cluster to inter-operate with the Computer Cluster the OERTE server must provide Embedded Cluster modex data to the ORTED process running on the same host in a manner that is transparent to all other Computer Cluster hosts as well as the HNP. This is achieved as follows. First, during OERTE server initialization, a configuration file that defines the Embedded Cluster network parameters is loaded into an in-memory data-structure buffer which is modeled after the original modex data structure. When the modex data associated to all the virtual processes (i.e. the processes running on the embedded devices) is requested by the ORTED process, the New ODLS Module makes a request to the OERTE server, which simply returns the information loaded in the data-structure buffer. This is then forwarded to the ORTED and delivered to the HNP, which aggregates all modex data from all hosts running ORTED in the heterogeneous cluster. The modex exchange process is completed when the HNP sends all the data from all nodes in both the Computer Cluster and the Embedded Cluster to each ORTED process. In the case of the Embedded Cluster however, this information is delivered to the OERTE server, which stores it in a cache memory where it can be accessed by the processes running on the STBs through an on-demand caching mechanism. This mechanism is implemented in the OERTE embedded client process running on the STBs.

OERTE Embedded Client. We developed the OERTE embedded client as a complete new replacement of the OPEN MPI ORTE module optimized for embedded devices. It consists of a application-client daemon that runs as a background process on the embedded Linux operating system. It is responsible for accessing MPI applications locally or from remote services, such as NFS-mounted file systems, and provides the runtime execution environment on the STB. This includes a number of functions: process launch, delivery of modex data and process signals, re-direction of I/O from the executing MPI application to the OERTE server, and process termination. With an approximate size of 32KB, it utilizes a relatively small amount of the STB memory resources.

Fig. 6 shows the software architecture of the OERTE embedded client. The main process is started at STB boot-time from a standard Linux `rc.local` boot script and executes in the background continuously waiting for command and control signalling messages from the OERTE server. This main process forks a sequence of three threads which are executed asynchronously by all sub-systems: 1) a Process Launch Manager; 2) an Application Loader; and 3) a Modex Data Manager.

The Process Launch Manager is responsible for handling command and control protocol communications with the OERTE server. It manages the MPI application lifecycle by processing commands from the OERTE server to launch, deliver signals to, and terminate applications. It also accesses the OPEN MPI application from the STB-resident memory (as stored by the Application Loader) and starts each MPI application by executing a fork and a Linux system call. All applications are child processes of the Process Launch Manager, which may deliver Linux signals and can support redirection of application I/O as required. The Application Loader thread is responsible for accessing and bringing into the STB memory the intended MPI application as determined by the Process Launch Manager thread. The specific method for loading an application is abstracted within the Application Loader, e.g. applications can be accessed independently and concurrently from the STB local Flash memory or retrieved from a remote file system. As future delivery methods become available the Application Loader can be extended. Finally, a separate Modex Data Manager thread manages a small in-memory cache to support the OPEN MPI application modex-data look-up in coordination with the OERTE as shown in Fig. 5.

Open MPI Embedded Library. We developed an optimized, reduced-footprint version of the OPEN MPI library to minimize the use of STB memory resources following an approach similar to the one used by McMahon *et al.* for the MPICH MPI software distribution [11]. Specifically, we removed those frameworks and modules which are not applicable in the STB environment by modifying the Linux build tools appropriately. For example, we removed the Byte-Transfer-Layer (BTL) modules which support delivery of messages over one or more network interfaces other than TCP, such as Infiniband or shared memory. Similarly, we removed modules for parallel I/O and vendor-specific debugging modules. Finally, thanks to the embedded processor virtualization discussed above, we could remove also most of the ORTE subsystem. In terms of size reduction, our optimized embedded version is about 32% smaller than the full OPEN MPI library, which is about 3.1MB. While this represent a significant memory saving for the current STB generation, as the memory capacity of embedded systems continues to grow we expect that this optimization will become less necessary.

In our implementation the ORTE framework was significantly reduced since the embedded processor runtime environment is no longer based on it. Indeed, we replaced ORTE with two new modules. First, the OERTE embedded client replaced the original ORTED. Second, the ORTE module for group communications (`grpcomm`) was refactored to support cache-based modex-lookup operations of the OERTE client instead of the ORTE collective operation method. This is a key optimization to overcome the scaling challenge (discussed in Section III) of deploying OPEN MPI application on a large-scale distributed embedded system. Whereas the standard ORTE implementation relies on a group collective `allgather` operation to exchange modex data among all processes resulting in a memory storage requirements of the order of $O(N * P)$, in our implementation the embedded client requires $O(1)$ memory thanks to the fixed size modex-caching subsystem. In this subsystem, all modex look-up operations are made to a local cache whose entries are co-managed by the OERTE server and embedded client processes.

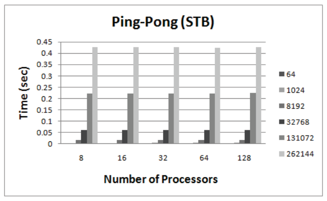
V. EXPERIMENTS

To validate our prototype system we completed three sets of experiments with different workloads. The goal of the experiments with the IMB benchmark suite is to demonstrate that our virtualization approach enables the interoperability between a Computer Cluster and Embedded Cluster to run any OPEN MPI application. The goal of the experiments with Ray Tracing and MSA is to evaluate the scaling potential of our system as a parallel execution platform.

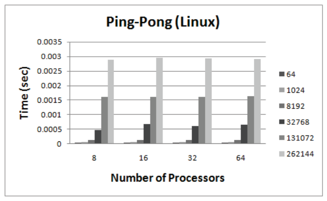
Experimental Setup. Recall from Fig. 2 that the Computer Cluster consists of 9 Linux blades with dual 2Ghz quad-core Xeon processors (one blade acting as master host) while the Embedded Cluster consists of 128 Samsung SMT-C5320 STBs each with a single dual-core 400MHz Broadcom processor. In our experiments, we executed each workload test using all 8 blades plus the master on the Computer Cluster and all 128 embedded STBs on the Embedded Cluster. For each experiment on the Computer Cluster we repeatedly executed a workload by scaling the number of MPI processes (from 8, through 16 and 32, to 64) while evenly distributing them across all Xeon cores. On the Embedded Cluster, we repeatedly executed a workload by scaling the number of MPI processes (from 8 to 128) and distributing them with a one-to-one mapping on the 128 STBs (i.e. each STB runs at most one MPI process).

Intel MPI Benchmarks (IMB). This benchmark suite consists of three parts (IMB-MPI1, IMB-MPI2, and IMB-IO) and provides an efficient way to measure the performance of the main MPI functions [12]. We run the following IMB-MPI1 benchmarks which allow us to test important single-transfer, parallel-transfer, and collective communication operations: `ping-pong`, `send-recv`, `exchange`, `allreduce`, `reduce`, `reduce-scatter`, `allgather`, `gather`, `scatter`, `bcast`, `alltoall`, `barrier`. In particular, `ping-pong`, measures startup latency and throughput for a single-transfer message exchange between two processes. Parallel-transfer benchmarks such as `send-recv` and `exchange`, measure the throughput of concurrence messages sent or received by a particular process in a periodic chain. The collective benchmarks measure the time needed to communicate among a group of processes in different patterns. We run each benchmark with various message sizes (in bytes): 64, 1K, 8K, 32K, 128K, and 256K. While we executed tests with all IMB-MP1 benchmarks, due to lack of space we report the results for only a representative subset. Each bar diagram in Fig. 7 shows the execution time as function of the number of processors and message size.

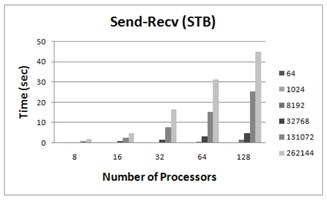
For each test on both clusters, as we increase the number of processors the execution time increases, except for `ping-pong` and `bcast` on the Computer Cluster where



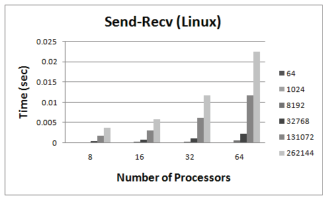
(a)



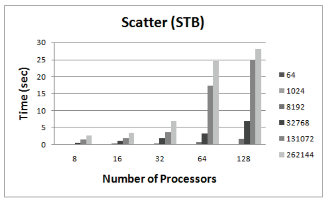
(b)



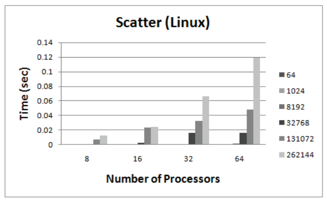
(c)



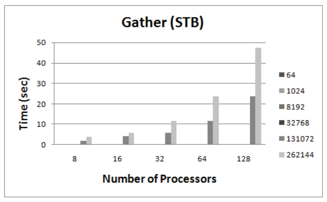
(d)



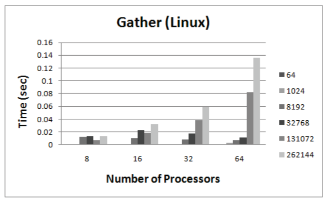
(e)



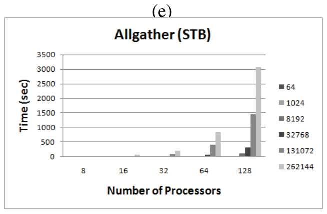
(f)



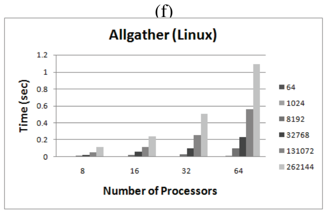
(g)



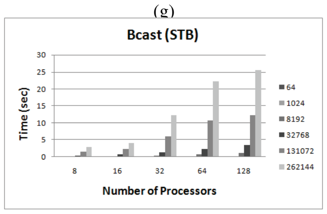
(h)



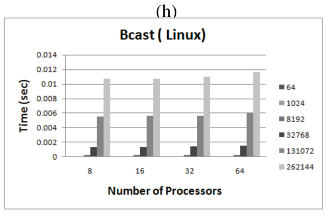
(i)



(j)



(k)



(l)

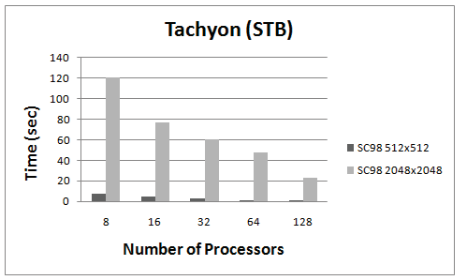
Figure 7. IMB Test Results.

it depends only on the message size. In `ping-pong`, the communication involves only two computer nodes and the average execution times over all message sizes are 122ms and 0.9ms for the Embedded Cluster and the Computer Cluster, respectively. This large difference is due to the performance gap between the Computer Cluster Gigabit Ethernet network and Embedded Cluster DOCSIS network. In fact, the reported bandwidth for this benchmark averages between 45MB/s and 50MB/s for Ethernet but only 0.39MB/s for DOCSIS. The reported execution time of `bcast` (a broadcast communication test) on the Computer Cluster remains approximately constant as we vary the number of computer nodes (averaging 3.7s across all message sizes) and it increases only as we increase the message size: for 64B messages the average execution time is 0.07ms while for larger 256KB message becomes 10ms. In contrast, the execution times of `bcast` for the Embedded Cluster increase as we increase both the number of nodes (from 840ms for 8 STBs to 7.1s for 128 STBs) and the message size (from 200ms for 64B messages to 13.4s for 256KB message, averaged across all node counts). The difference in performance and the sensitivity to the node number between the Computer Cluster and Embedded Cluster is due not only to the lower bandwidth of DOCSIS but also to the highly-optimized implementations of this OPEN MPI collective operations which the Computer Cluster nodes can access.

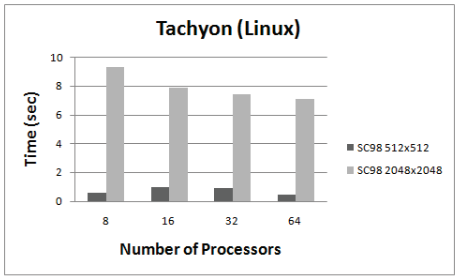
For the rest of the results of Fig. 7, the execution time increases directly proportional to the number of nodes and the message sizes. In all cases it is far less on the Computer Cluster than on the Embedded Cluster (approximately by a factor of 100) and the reasons are similar as above: first, Gigabit Ethernet offers 100 times the bandwidth in comparison to DOCSIS (and significant higher for communications between the Xeon cores that are on the same chip); second, the Computer Cluster can run implementations of such collective OPEN MPI operations as `gather`, `scatter`, and `allgather` that are highly optimized.

Besides these facts, however, the important conclusions of these experiments are: (1) the validation that Embedded OPEN MPI Implementation running on the Embedded Cluster can execute correctly all IMB MPI benchmarks and (2) the demonstration that the performance of the collective MPI operations scales consistently across both the Computer Cluster and the Embedded Cluster environments. The next sets of results show how for those OPEN MPI applications that do not benefit for high-performance implementations of collective operations the performance gap between the two clusters is much smaller and decreases with the scaling of the parallel applications and cluster size.

Parallel Ray Tracing. We use the TACHYON ray-tracer as a workload to evaluate the scaling performance of our system on a parallel image-processing application, Ray tracers are used to render scenes images in games, 3-D modeling/visualization, and virtual reality applications [13]. They are well suited for parallelization thanks to high data parallelism: each pixel in the rendered image can be processed independently and, therefore, different pixels can be assigned by the master host to different computer nodes [14]. We used a scene input file (SC98) from the TACHYON distribution and rendered it in two resolutions (512x512 and 2048x2048) to account for two different computational complexities.



(a)



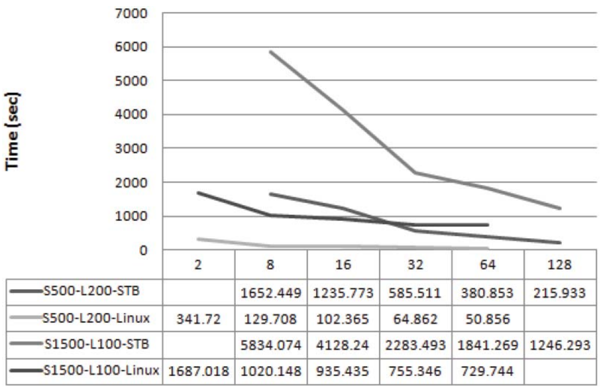
(b)

Figure 8. Ray-Tracing Results: (a) Embedded Cluster; (b) Computer Cluster.

As shown in Fig. 8, both clusters exhibit improved performance as the number of MPI processes grows. For the high-resolution case, as we increase the number of nodes from 8 to 64 the execution time improves from 9.3s to 7.07s (a speedup of 1.3) on the Computer Cluster and from 120.7s to 47.9s on the Embedded Cluster (thus resulting in a higher speedup of 2.5, which becomes 5.3 with 128 STBs). For the 512x512 resolution, as we go from 8 to 64 nodes, the execution time goes from 0.58s down to 0.44s on the Computer Cluster and from 6.9s to 1.27s on the Embedded Cluster (and down to 1.01s with 128 STBs). For the Computer Cluster the performance gain is flatten and the execution time is I/O bounded, thus resulting in overall speedup of just 1.3, with the parallel portion of the application having a speedup of 4.3. Instead, the Embedded Cluster overall speedup is 5.5 (6.9 with 128 STBs), with its parallel portion having a speedup of 7.7 (14.9 with 128 STBs).

In summary, the relative speedup as we increase the number of nodes is higher on the Embedded Cluster than the Computer Cluster for both a large image (requiring more pixel calculations that can be computed in a data-parallel model) and a small image (when relatively more time is spent in I/O operations).

Multiple Sequence Alignment (MSA). This is a fundamental problem in bioinformatics: instances of DNA or protein sequences must be optimally aligned so that the highest possible number of their elements match. Given a scoring scheme to evaluate this matching and penalize the presence of sequence gaps, to solve MSA consists in placing gaps in each sequence to maximize the alignment score [15].



	2	8	16	32	64	128
S500-L200-STB		1652.449	1235.773	585.511	380.853	215.933
S500-L200-Linux	341.72	129.708	102.365	64.862	50.856	
S1500-L100-STB		5834.074	4128.24	2283.493	1841.269	1246.293
S1500-L100-Linux	1687.018	1020.148	935.435	755.346	729.744	

Figure 9. Impact of sequence size/length on performance scaling.

Since MSA is an NP-hard problem, an approximation algorithms such as ClustalW is typically used [16]. We run the MASON parallel ClustalW implementation using MPI [15], [17], which proceeds as follows: the master host partitions the N input sequences among P worker processors; these perform pair-wise alignment on their set of sequences in parallel; alignment scores are sent back to the master which constructs the guide tree and distributes the computed guide order along with associated sequences where the workers then compute a partial MSA; finally, the workers send their partial multiple alignments back to the master, which performs the final stage of progressive alignment. We used ROSE, a tool that produces synthetic sets of DNA sequences which follow an evolutionary model [18], to generate 7 sequences of various length and base pair. In the sequel, the encoding *Sx-Ly* denotes a data set of x DNA sequences with y base pairs (e.g. *S100-L1500* means "100 DNA Sequences with 1500 base pairs").

Table I reports the overall execution times for all sequences for two particular configurations of each cluster (1 and 64 processors for the Computer Cluster, and 8 and 64 STBs for the Embedded Cluster) together with the relative speedups. In most cases the Computer Cluster has higher speedup. This is expected given the benefits of high-performance Xeon processors and Gigabit Ethernet (as verified with the IMB benchmarks). In two cases (sequences S500-L200, S1500-L100), however, the Embedded Cluster outperforms the Computer Cluster in terms of speedup: as shown in Fig. 9, it exhibits higher relative performance gains as we increase the number of sequences, when the average sequence length is relatively short. This is due to the data-parallelism portion (aligning partitioned $\frac{N \cdot (N-1)}{2}$ sequence permutations independently) of the MSA algorithm, which benefits a cluster with a large number of nodes. In contrast, the Computer Cluster gives higher performance gains for longer sequences as their processing requirements has complexity bounded by $O(n^2)$, thus favoring the higher performance blade servers when the number of sequences is small compared to their length. However, Fig. 9 shows that in moving from 64 to 128 STBs the Embedded Cluster actually

Processor		Overall Execution Time (Sec)							
Type	#	S100-L1500	S200-L300	S300-L200	S500-L200	S200-L500	S500-L1100	S1500-L100	Avg.
	1	692	129	62	341	289	4130	1687	
Linux	64	43	15	8	51	21	326	730	
	Speedup	**16.1**	**8.6**	**8.1**	**6.7**	**13.7**	**12.7**	**2.3**	**9.7**
	8	3257	693	319	1652	1522	20854	5834	
C5320 STB	128	391	113	44	216	154	2340	1246	
	Speedup	**8.3**	**6.1**	**7.2**	**7.7**	**9.8**	**8.9**	**4.7**	**7.3**

Table I
EXPERIMENTS WITH MULTIPLE SEQUENCE ALIGNMENT: OVERALL EXECUTION TIMES AND SPEEDUPS OF EACH CLUSTER.

manages to complete the application execution in a time that is shorter that the time taken by the Computer Cluster with only two blades.[1] This suggests that an Embedded Cluster with sufficient processing nodes is suited for a wider range of data-intensive, parallel applications where very large data-sets must be processed.

Discussion. As we evaluate these experimental results (and particularly the IMB ones), if we factor out any performance gain advantage of the Embedded Cluster due to the data-parallelism of the workloads, it is clear that differences in network performance have a significant impact on the overall execution time. As we look to further improve our system, a number of factors including physical network and software architecture must be considered to reduce these differences. In terms of physical network, the Embedded Cluster system was tested on a lab environment (where contention exists) that did not include the use of QoS parameters to manage bandwidth allocations. The DOCSIS standard has facilities for managing and prioritizing bandwidth on a per device or application basis. While this may reduce contention, the physical network is still bounded by a maximum upstream and downstream bandwidth of 27Mb/s. To overcome this limitation, future DOCSIS 3.0 networks will use channel bonding to obtain higher bandwidth up to 340Mb/s. Further improvements, however, will require large investments of capital by service providers. Hence, as a more practical method, we plan to evaluate the use of virtual networks to overlay more efficient communication topologies on top of the existing physical network. These methods are already intrinsic within the Computer Cluster where the OPEN MPI software has highly-optimized MPI collective operations, which utilize tree-based communication algorithms.

VI. RELATED WORK

While the utilization of large scale networks of embedded devices for heterogeneous broadband grid computing within a managed, dedicated system raises new challenges and opportunities in system scalability and performance, the idea of harnessing distributed embedded systems, particularly over the Internet, is not new. A number of initiatives have focused on utilizing volunteer PC hosts or game consoles for solving difficult problems such as those found in Computational Biology [19] or the various projects supported by the Grid Republic organization [20]. In these distributed volunteer-computing initiatives, units of computational work are performed by downloaded agents or screensaver-based software components, which are executed by the subscribing users. This approach is based on an *ad hoc* service model and assumes an unmanaged client environment with no assurance of predictable process execution or participation of client devices. Our work is closer to recent efforts on the virtualization of mobile devices for grid computing [21], [22] from which it differs for the emphasis on combining embedded computing with a *managed broadband network*.

Most related works in the area of integrating message-passing middleware into embedded devices have focused on reducing the size of the MPI stack. Lightweight MPI is based on a thin-client model where the MPI API layer is implemented on top of a thin-client-message protocol which communicates with a proxy server that supports a full MPI stack [23]: client requests are routed through the proxy server, which acts on behalf of one or more MPI thin-client user processes. A key benefit of this approach is the elimination of the need for an operating system on the embedded device. Other approaches are based on a bottom-up implementation of a minimal MPI stack, as in the case of Embedded MPI (eMPI) [11]. Similarly to eMPI, we used a top-down approach to minimize the MPI framework and eliminate unnecessary software modules, resulting in a reduced memory library foot-print. In our system, however, each STB contains a modern real-time operating system that can support a native MPI implementation. Also, while previous work in executing MPI on embedded devices has focused on small test kernels, we can run complete application that use of a rich set of MPI operations, including collectives.

Google designed and deployed a massively-parallel system comprised of commodity dual-core PCs running Linux combined with its custom Map-Reduce framework for parallel computing [24]. This platform is distributed across many data-centers and was estimated in size at over 450,000 systems [25]. A possible future large-scale version of our

[1]As one considers the significance of obtaining the same performance of two blade servers with a cluster of over 64 STBs, it should be kept in mind that each blade features 64-bit processors running at a clock frequency which is five times higher than the clock frequency of the 32-bit processor of the STB!

proposed architecture would have important differences with the Google platform, including the use of a broadband network of embedded devices instead of a network of clusters of PCs and the use of a hybrid MPI and Map-Reduce application model which is today an active area of research.

VII. Conclusions

We presented a heterogeneous distributed system architecture for broadband grid computing. Our contributions include a new method to integrate networks of embedded processors with computer clusters through a software virtualization framework we call OERTE. This enables embedded processors to transparently inter-operate with computer clusters using the message-passing model. We described our prototype implementation and evaluated it with three sets of experiments to validate its operations and scaling potential. The results indicate that the performance of our system is impacted by the existing broadband DOCSIS network which is not optimized for Open MPI collective operations and that future work is required in this area. But they also demonstrates that our system is already able to deliver significant performance gains for some classes of Open MPI applications. This suggests a wealth of opportunity in leveraging broadband grid computing for a number of data-intensive and data-parallel applications.

Acknowledgment

This work is partially supported by Cablevision Systems. The authors gratefully acknowledge the help of Satish Marada for the building of the system prototype.

References

[1] `http://www.infonetics.com`.

[2] C. H. van Berkel, "Multi-core for mobile phones," in *Proc. of the Design Automation Conference*, Apr. 2009, pp. 20–24.

[3] P. Kollig, C. Osborne, and T. Henriksson, "Heterogeneous multi-core platform for consumer multimedia applications," in *Proc. of the Design Automation Conference*, Apr. 2009, pp. 1254–1259.

[4] `http://www.tru2way.com`.

[5] `http://www.cablemodem.com`.

[6] `http://www.cablelabs.com` .

[7] Open MPI, "`http://www.open-mpi.org`."

[8] E. Gabriel *et al.*, "Open MPI: goals, concept, and design of a next generation MPI implementation," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 11th European PVM/MPI Users Group Meeting*, Sep. 2004, pp. 97–104.

[9] `http://www.busybox.net`.

[10] `http://www.kernel.org`.

[11] T. McMahon and A. Skjellum, "eMPI: Embedded MPI," in *MPI Developers Conference*, Jul. 1996, pp. 180–184.

[12] A. Bukhamsin, M. Sindi, and J. Al-Jallal, "Using the Intel MPI benchmarks (IMB) to evaluate MPI implementations on an Infiniband Nehalem Linux cluster," in *Proc. of the Spring Simulation Multiconference*, 2010, pp. 1–4.

[13] M.-L. Li *et al.*, "The ALPBench benchmark suite for complex multimedia applications," *IEEE Workload Characterization Symposium*, vol. 0, pp. 34–45, 2005.

[14] J. Stone, "An efficient library for parallel ray tracing and animation," In Intel Supercomputer Users Group Proceedings, Tech. Rep., 1995.

[15] A. Datta and J. Ebedes, "Multiple sequence alignment in parallel on a workstation cluster," in *Parallel Computing for Bioinformatics and Computational Biology*, ser. Series on Parallel and Distributed Computing, A. Zomaya, Ed. J. Wiley & Sons, 2006, ch. 8, pp. 193–210.

[16] J. Thompson, D. Higgins, and T. Gibson, "ClustalW: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice," *Nucleic Acids Res.*, vol. 22, no. 22, pp. 4673–4680, Nov. 1994.

[17] J. Ebedes and A. Datta, "Multiple sequence alignment in parallel on a workstation cluster," *Bioinformatics*, vol. 20, no. 7, pp. 1193–1195, May 2004.

[18] J. Stoye, D. Evers, and F. Meyer, "Rose: generating sequence families," *Bioinformatics*, vol. 14, no. 2, pp. 157–163, Mar. 1998.

[19] `http://folding.stanford.edu`.

[20] `http://www.gridrepublic.org`.

[21] M. Black and W. Edgar, "Exploring mobile devices as grid resources: Using an x86 virtual machine to run BOINC on an iPhone," in *Proc. of the 10th IEEE/ACM International Conference on Grid Computing*, 2009, pp. 9–16.

[22] G. Huerta-Canepa and D. Lee, "A virtual cloud computing provider for mobile devices," in *Proc. of the 1st ACM Workshop on Mobile Cloud Computing & Services: Social Networks and Beyond*, 2010, pp. 6:1–6:5.

[23] A. Agbaria, D.-I. Kang, and K. Singh, "LMPI: MPI for heterogeneous embedded distributed systems," in *12th International Conference on Parallel and Distributed Systems (ICPADS)*, Jul. 2006, pp. 79–86.

[24] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008.

[25] C. Evans, *Future of Google Earth*. Booksurge Llc., 2008.