Catamount Software Architecture with Dual Core Extensions

Suzanne M. Kelly
Sandia National Laboratories*
Scalable Systems Integration Department
PO BOX 5800
Albuquerque, NM 87185-0817
smkelly@sandia.gov

Ron Brightwell, John Van Dyke
Sandia National Laboratories
Scalable Computing Systems Department
PO Box 5800
Albuquerque, NM 87185-1110
rbbrigh@sandia.gov, jpvandy@sandia.gov

ABSTRACT

Catamount is the light weight kernel operating system running on the compute nodes of Cray XT3 systems. It is designed to be a low overhead operating system for a parallel computing environment. Functionality is limited to the minimum set needed to run a scientific computation. The design choices and implementations will be presented. This paper is a reprise of the CUG 2005 paper, but includes a discussion of how dual-core support was added to the software in the fall/winter of 2005.

Keywords

Operating Systems, MPP, Light Weight Kernel, Dual Core.

1.0 Background

A massively parallel processor (MPP), high performance computing (HPC) system is particularly sensitive to operating system overhead. Traditional, multi-purpose, operating systems are designed to support a wide range of usage models and requirements. To support the range of needs, a large number of system processes are provided and are often inter-dependent on each other. The overhead of these processes leads to an unpredictable amount of processor time available to a parallel application[1, 2]. Except in the case of the most embarrassingly parallel of applications, an MPP application must share interim results with its peers before it can make further progress. These synchronization events are made at specific points in the application code. If one processor takes longer to reach that point than all the other processors, everyone must wait. The overall finish time is increased.

Sandia National Laboratories began addressing this problem more than a decade ago with an architecture based on node specialization [3]. Sets of nodes in an MPP are designated to perform specific tasks, each running an operating system best suited to the specialized function. Sandia chose to not use a multi-purpose operating system for the computational nodes and instead began developing its first light weight operating system, SUNMOS, which ran on the compute nodes on the Intel Paragon system [4]. Based on its viability, the architecture evolved into the PUMA operating system [5]. Intel ported PUMA to the ASCI Red TFLOPS system [6], thus creating the Cougar operating system. Most recently, Cougar has been ported to Cray's XT3 system and renamed to Catamount. As the references indicate, there are a number of descriptions of the predecessor operating systems. While the majority of those discussions still apply to Catamount, this paper takes a fresh look at the architecture as it is currently implemented.

2.0 Description of Catamount

The design goals of catamount remain consistent with its predecessors:

- Targeted at massively parallel environments comprised of thousands of processors with distributed memory and a tightly coupled network.
- Provide *necessary* support for scalable, performance-oriented scientific applications

---

* Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company for the United States Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

- Offer a suitable development environment for parallel applications and libraries.
- Emphasize efficiency over functionality.
- Maximize the amount of resources (e.g. CPU, memory, and network bandwidth) allocated to the application.
- Seek to minimize time to completion for the application.

In achieving these design goals, catamount assumes a functionally partitioned MPP. That is, catamount runs on processors intended for intense computation and relies on other processors within the MPP to perform additional services. These service processors run Linux and provide the interactive development environment, file I/O, and high speed access to external services. Figure 1 depicts the functional partitions.

Figure 1: Functional Partitions of MPP using Catamount

Compute Processors (Catamount)
I/O processors (Linux)
Service Processors (Linux)
Network I/O Processors (Linux)
High Speed External Network

Continue to refer to Figure 1 as we discuss the usage model for Catamount. A user logs into a Linux service processor and performs typical application execution set-up, such as compilation and creation of the problem dataset. The user then requests that the application be run on some number of compute processors. The user-invoked "yod" program (see Section 5.0), executing on a service processor, launches the application on the assigned compute processors. Once started, the application runs under Catamount's control. The application instances on each compute processor pass messages to communicate with each other over the tightly coupled network. File I/O flows between the compute processors and the I/O processors. When the application completes, the user can review the results on the service processors and/or direct that the files be sent to external services for post processing or archival storage.

The Catamount operating system consists of a Quintessential Kernel (QK) and a Process Control Thread (PCT)[7]. The PCT and the QK work together to provide a complete operating system. The PCT will decide what physical memory and virtual addresses a new process is to have and at the behest of the PCT, the QK will set up the virtual addressing structures for the new process that are required by the hardware. The PCT will decide which process is to run next and at the behest of the PCT, the QK will flush caches, set up the hardware registers, and run this process. There is a clear separation between resource management and kernel task execution. The PCT is responsible for setting the policies and the QK is responsible for enforcing them. The basic structure of these components is shown below:

Figure 2: Compute Processor Components

Process Control Thread
Application Process (up to 4; typically 1 for single cores and 2 for dual cores)
Quintessential Kernel

While Figure 2 provides a useful logical view of the components, the physical layout is quite different and is shown in Figure 3.

Figure 3: Catamount Physical Memory Layout (not to scale)

								PCT heap				
QK text	QK data	Stack	Network buffer (qk heap)	Portals memory	PCT text	PCT data	Stack	Persistent	PCT heap	User program text & data	Stack	User heap

Up to 4 instances

When the QK installs the PCT, the remainder of physical memory is included in the PCT's heap. When the PCT loads an application, it forfeits the bulk of its heap space to the application. This is important because it means the user application is within the PCT's address space. This facilitates the PCT's role in supporting an application debugger and performing core dump processing.

While Catamount supports virtual addressing, there is no virtual memory support. This is an important performance, reliability and scalability feature of Catamount. Disks needed to implement virtual memory are very slow in comparison to memory access, have a low mean time to failure, and impede the predictable progress of the application.

Another feature of Catamount's memory management is default support for 2 MB pages. Larger pages can significantly reduce cache misses and TLB flushes as they cover a larger percentage of memory. Smaller, 4K pages are supported as well for applications that require more random access of memory. The user enables small pages with a command line option when starting the parallel application.

3.0 Quintessential Kernel Internals

The QK is the lowest level of the operating system. Logically, it sits closest to the hardware and performs services on behalf of the PCT and user-level processes. The QK supports a small set of tasks that require execution in privileged supervisor mode, including servicing network requests, interrupt handling, and fault handling. If the interrupt or fault is caused by the application, control is turned over to the PCT for handling. The QK also fulfills privileged requests made by the PCT, including running processes, context switching, virtual address translation and validation. However, the QK does not manage the resources on a compute node. It simply provides the necessary mechanisms to enforce policies established by the PCT and to perform specific tasks that must executed in supervisor mode.

The QK provides a trap mechanism for communicating with the PCT or with the application. The following traps are in use:

NULL_TRAP--no trap – a handler is never called (used for timing studies)
SETUID – set user id (PCT only)
LPUTS--print a string to console
QUIT_QUANTUM – quit quantum application request to the PCT
INIT_PROC--initialize a process (PCT only)
GET_CACHE – get cache table entry
RUN_PROCESS--run the indicated process context
INSTALL_PCT--set the start address for the PCT (PCT only)
INIT_REGION--initialize a memory region for a process (PCT only)
MEMLOGCTL—control for the memory log capability
TRAP_NOP – returns CPU ID
TRAP_CPU_MIGRATE – migrate process from cpu0 to cpu1
TRAP_DUAL_PROC – check for second CPU
RCAD_IOCTL—interface to the RAS system
PTL_KERNEL_FWD – dispatch portals system call

4.0 Process Control Thread Internals

The PCT is a privileged user-level process that performs functions traditionally associated with an operating system. It has read/write access to all memory in user-space and is in charge of managing all operating system resources. This involves process loading, job scheduling, and memory management. While QKs do not communicate with each other, the PCTs on the nodes that have been allocated to a parallel application communicate to start, manage, and some times to tear down the job.

The PCT uses a mailbox structure to interface with an application. When needed, the application fills in the mailbox and then gives up the processor using the quit quantum trap. The following is the list of currently supported mailbox requests.

EXIT – process exiting
WAIT – process requests wait
SLEEP – process gives up quantum
SIGNAL – signal another PCT; used in kill processing
SET_QUANTUM – set the application quantum
PERFMON – read or write performance registers
ABORT_LOAD – abort the load of an application
CORE_PROC_INIT – initialize structure for core dump processing
SET_VN_MODE – inform scheduler that a Virtual Node job is running

5.0 Yod Internals

The parallel job launcher component of the runtime system is called yod. Yod is an evolution of the xnc (execute network computer) program used to launch jobs on the nCube supercomputer. (x+1)(n+1)(c+1) = yod. Yod contacts a compute node allocator to obtain a set of compute nodes, and then communicates with the first (base) PCT to move the user's environment and executable out to the compute

nodes. The base PCT works with the secondary PCTs in the job to efficiently distribute this data to all of the compute nodes participating in the job. This fan-out capability is a significant accomplishment. It is an asynchronous protocol, requires no timers, and allows all nodes to be making progress at their own rate. We have empirical data showing that similar attempts to implement a logarithmic job launch have been problematic. There are numerous end cases and reliability issues to overcome.

Once a job has started, yod serves as an I/O proxy for all standard I/O (stdio) functions. Compute node applications are linked with a library (see Libcatamount is Section 7.3) that redefines the standard I/O library routines and some system calls. This library implements a remote procedure call interface to yod, which actually performs the operation locally and then sends the result to the compute node process. Yod also disseminates some UNIX signals that it receives out to the processes running in the parallel job. When yod receives a signal, it sends a message to each PCT in the job who delivers the desired signal to the application process. This feature is used for operations such as user-level checkpointing and killing jobs.

The yod program allows for a large number of arguments on the command line. They are listed below. But in the simplest case, only the yod command and the name of the program are required.

yod [-Account project/task] [-D option] [-help] [{ -size | -sz | -np }{ n | all }] [-VN] [-small_pages] [-stack size] [-tlimit secs] [-list processor-list] [-strace] [-target { catamount | linux }] [-share] [-heap size] [-Priority priority] [-Version] progname [progargs] | -F loadfile

Many of these options are self-explanatory, but a few are worth noting. Since there is no virtual memory, the application stack size does not grow dynamically during execution. It is allocated during application load. The default size is 16 megabytes. This option must be tuned by the user, if necessary. The share option is used to support multiple applications on the same processor. Do not confuse this option with the Dual Core –VN (virtual node) option. The virtual node implementation is described separately in Section 6. The application's heap size defaults to the rest of memory (see Figure 3) after the text, data, and stack sections are allocated. The heap option is used with share mode to reserve space for the subsequent programs sharing the processor.

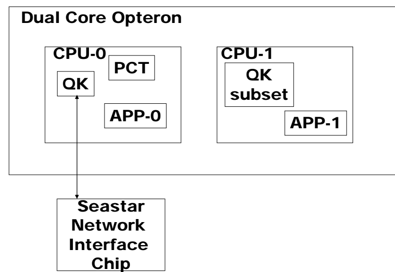
The File or F option allows the user to specify multiple executables. Each executable runs on some portion of the allocated processors. The executables can perform different functions, but are still able to inter-communicate with the other executables. This feature is particularly useful with the target option introduced in Catamount. The target option did not exist in Cougar or its predecessors. Yod can now launch executables on Linux processors as well as processors running Catamount. They must be different executables as the system libraries differ (no dynamic linking on Catamount). One possible usage scenario is to perform intensive computation on the Catamount processors and communicate interim results to the executable on the Linux node. The Linux executable has the fuller set of services to perform such operations as visualization. [Note that a syntax change is being considered, so that the target operating system may be identified in a different way on the command line.]

6.0 Dual Core Extensions

Dual core support has been added to Catamount over the past year. Philosophically it is Virtual Node (VN) mode from Cougar. Cougar was the predecessor operating system to Catamount which ran on the ASCI Red MPP. Each ASCI Red node had two processors.

VN allows the application to use twice as many nodes with no change to the application executable. It must be remembered however that the number of processors is the only resource that has been doubled. The node memory is split between the two processes and the two processes share network access. It should not be thought of as an SMP since the two processes on a node do not share memory. There is only one QK image and one PCT image on a node. The single PCT only runs on the first processor and manages resources for both processes.

Figure 4: Dual Core CPU responsibility assignment



From the application perspective there are two totally independent processes on the node just as if they were on separate nodes. Only one of the processors talks directly to the network. This means that network requests from the second process must be passed (proxied) to the first process. Either process may trap into the kernel and initially that is handled by its own processor. Of the listed traps, the second processor handles three of them itself, forwards three of them to the first processor and rejects the rest. The only case that requires inter-CPU locking is writing to the system console.

As implemented, VN is incompatible with share mode. When running in VN, those two processes must be the only application processes on the node. Share mode was discussed briefly in Section 5. It has proven to be of limited utility and therefore the restriction is not deemed significant.

6.1 QK multi-CPU support

The QK changes for multi-CPU support fall mainly in two categories: adding a dimension to control variables and second CPU startup at boot time. The QK does not have a particular awareness of VN mode, but it does treat the CPUs in a master-slave relationship. Since all PCT execution and scheduling occurs on the first processor, the second processor has a “wait-for-work” loop in the QK. Running an application process on CPU-0 involves a context switch from the PCT. Continuing a process on CPU-1 involves simply clearing the flag that allows the processor to come out of the wait-for-work loop.

6.2 Changes for Job Load

For VN, the load process as seen from yod is hardly changed. A single copy of the executable is fanned out to the PCTs. This does restrict the dual cores to having a single choice of binary on each node. The PCT lays out two copies of the program in memory dividing the heap equally between the two processes.

Both processes on a node then begin independently and the second process migrates to the second CPU by making a system request. It then notifies the PCT to switch schedulers and the VN scheduler lets the processes run as appropriate. There is no true scheduling between them.

7.0 Libraries

By themselves, the QK and PCT that make up the catamount OS are not very useful. System libraries provide the mechanism for an application to use their services. Four key libraries must be linked with each application: libc, libcatamount, libsysio, and libportals.

7.1 Libc

Catamount is the first version of the light weight OS to port and use glibc. The decision to use glibc was difficult to make. It is a very large and complex code base and does not seem in keeping with the light weight nature of the compute node system software. Due to the over-whelming advantage of being able to use an off-the-shelf Linux compiler, the port was done using glibc 2.3.2. Considerable pruning was done to provide only the features supported by the light weight operating system. Although customization is a strong feature of glibc, this port was non-trivial.

The following functional groups are not supported by the lightweight kernel and therefore cannot be implemented by libc:

- No threads support.
- No off-node communication other than via Portals, such as pipes, sockets, rpc's or Internet Protocols.
- No dynamic process creation; for example: no exec(), fork(), popen(), or system().
- No dynamic loading of executable code.
- Limited signals support. See the yod section for detail on signal handling.
- No /proc or ptrace.
- No mmap. A skeleton function is supplied, but returns –1.
- No profil().
- Limited ioctl
- No getpwd family of calls.
- No functions requiring any form of db (e.g. ndb). For example, there is no support for

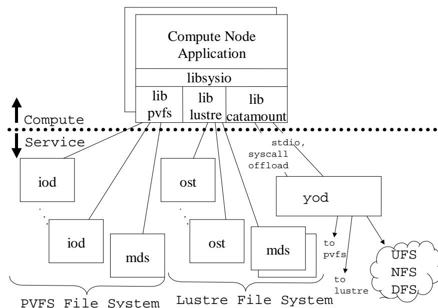
the uid, gid family of queries that are based on the ndb.

- No terminal control
- No functions that require UNIX-style daemons

7.2 Libsysio

The libsysio library multiplexes I/O to the function supporting the particular file system to which the I/O is targeted. Pictorially, the flow is as shown in Figure 5.

Figure 5: Role of Libsysio in File I/O



When a compute node application issues an open() call, libsysio determines the appropriate library to service the call, based on the file type associated with the path name. Once the appropriate library is selected during open, the same library is used for all subsequent I/O to that path name. In the figure, three possible libraries are shown: PVFS, Lustre and Catamount. The libsysio software is not constrained to exactly these three options. More or less libraries are possible. Libcatamount will service all stdio and I/O to the serial (Unix) file systems, currently identified as UFS, NFS, and DFS. Using its RPC mechanism, libcatamount offloads the I/O request to yod. Yod invokes the appropriate Linux system call to perform the I/O operation. Similarly, using the mechanism unique to their implementations, liblustres and libpvfs will forward I/O requests to their appropriate metadata servers and data handlers.

7.3 Libcatamount

Libcatamount provides several services. First, it implements a remote procedure call (RPC) mechanism for communicating between the application and its yod program. Any I/O that is handled by yod uses the RPC mechanism. This feature is sometimes referred to as sys(tem) call offload. While currently only supporting I/O-related functions and exit, it could support other system calls that can only be performed on a full-service operating system. To date, none have been needed.

Libcatamount provides a custom malloc that is tuned to favor large memory allocations. This is the default malloc, although the glibc malloc can be used if explicitly requested on the link line.

Previously, cstart.o was contained in the predecessor versions of libcatamount. The function _cstart2() is called prior to main() on compute node applications. It initializes important structures and communication paths. Due to linkage issues for catamount, cstart.o must remain a separately linked object file. But it is still archived as a member of the library.

Lastly, libcatamount contains some functions that are unique to Catamount. It provides a function for requesting and setting performance registers from/by the PCT. It implements a barrier function for synchronizing all processes on all nodes in the parallel application. This function is invoked in _cstart2 prior to calling main. In this way, applications have a shot-gun style start. The dclock function returns a high-resolution time-since-boot that can be used for timing studies.

7.4 Libportals

The portals library [8] is not part of Catamount, but is a critical and required component. It is a separate, low-level network programming interface. The portals facility is used whenever there is communication between any two nodes—whether they are compute or service nodes. Catamount currently uses version 3.3 of portals. System, application, and service protocols are implemented on top of it.

The portals protocol is connectionless and provides protected, reliable, in-order delivery. It is designed to support multiple communicating processes per node and communication between processes created from different executables.

To support scalability, the portals interface maintains a minimal amount of state. A process is not required to explicitly establish a point-to-point connection with another process in order to communicate. Moreover, all buffers used in the transmission of messages are maintained in user-space. The target process determines how to respond to incoming messages. Messages for

which there are no buffers are discarded. That is, portals are based on expected messages. Higher-level message passing layers that need support for unexpected messages, such as MPI, need to set aside a certain amount of space to receive unexpected messages.

8.0 Future Work

Sandia continues to enhance Catamount. We are currently investigating support for quad core processors. Additionally, we are completing work to use a protocol offload engine in the Network Interface Card.

Sandia is also continuing to research the design, implementation and deployment of operating systems for massively parallel scientific computing platforms. A research project is underway to investigate a framework for building application-specific operating systems [9]. This project is a collaboration between Sandia, the University of New Mexico, and the California Institute of Technology.

9.0 References

[1] F. Petrini, D. J. Kerbyson, and S. Pakin, "The Case of the Missing Supercomputer Performance: Identifying and Eliminating the Performance Variability on the ASCI Q Machine," presented at ACM/IEEE Conference on High Performance Networking and Computing, Phoenix, AZ, 2003.

[2] T. Jones, S. Dawson, R. Neely, W. Tuel, L. Brenner, J. Fier, R. Blackmore, P. Caffrey, B. Maskell, P. Tomlinson, and M. Roberts, "Improving the Scalability of Parallel Jobs by Adding Parallel Awareness to the Operating System," presented at ACM/IEEE Conference on High Performance Networking and Computing, Phoenix, AZ, 2003.

[3] D. S. Greenberg, R. Brightwell, L. A. Fisk, A. B. Maccabe, and R. Riesen, "A system software architecture for high-end computing," presented at SC'97: High Performance Networking and Computing, San Jose, California, 1997.

[4] A. B. Maccabe, K. S. McCurley, R. Riesen, and S. R. Wheat, "SUNMOS for the Intel Paragon: A brief user's guide," presented at Intel Supercomputer Users' Group, 1994.

[5] L. Shuler, C. Jong, R. Riesen, D. W. v. Dresser, A. B. Maccabe, L. A. Fisk, and T. M. Stallcup, "The Puma Operating System for Massively Parallel Computers," presented at Intel Supercomputer Users' Group, Albuquerque, NM, 1995.

[6] T. G. Mattson, D. Scott, and S. R. Wheat, "A TeraFLOP Supercomputer in 1996: The ASCI TFLOP System," presented at International Parallel Processing Symposium, Honolulu, HI, 1996.

[7] R. Brightwell, W. J. Camp, B. Cole, E. DeBenedictis, R. Leland, J. Tomkins, and A. B. Maccabe, "Architectural Specification for Massively Parallel Computers: An Experience and Measurement-Based Approach," *Concurrency and Computation: Practice and Experience*, vol. 17, pp. 1271-1316, 2005.

[8] R. Brightwell, R. Riesen, B. Lawry, and A. B. Maccabe, "Portals 3.0: Protocol Building Blocks for Low Overhead Communication," presented at 2002 Workshop on Communication Architecture for Clusters, 2002.

[9] A. B. Maccabe, P. G. Bridges, R. Brightwell, R. Riesen, and T. B. Hudson, "Highly Configurable Operating Systems for Ultrascale Systems," presented at First International Workshop on Operating Systems, Programming Environments and Management Tools for High-Performance Computing on Clusters, St. Malo, France, 2004.