HP Scalable Computing Architecture

Randy Wright
Arun Kumar
HP Platform Software Architecture Lab

Abstract

The HP V-Class server family provides up to 32 processors and 32 GB of memory in a single cabinet. Scalable Computing Architecture technology allows multiple V-Class cabinets to be interconnected, forming a single cache coherent non-uniform memory architecture (ccNUMA) system providing up to 128 CPUs and 128 GB of memory. This paper discusses some interesting aspects of the Scalable Computing Architecture and the changes made to the HP-UX kernel to operate in this environment.

1. Introduction

The Scalable Computing Architecture (SCA) was proposed to extend the scalability of Hewlett-Packard's high end V-Class machine. The V-Class is a crossbar based symmetric multiprocessor (SMP) machine. The SCA machine connects up to four V-Class nodes using a high-speed high bandwidth interconnect to conform to the ccNUMA architecture.

Normally, Hewlett-Packard's UNIX machines conform to a well-defined architecture as specified in [1] and [2]. The V-Class machine [3, 4] did not fully conform to this architecture. However, by emulating the PA-RISC architecture in the firmware layer, the HP-UX operating system could be supported on the V-Class. The architectural anomalies of the V-Class platform created unique challenges in supporting HP-UX on the SCA platform. This paper describes the modifications made to the HP-UX kernel to address some of these anomalies in the hardware and to extend the capabilities of the V-Class to meet the requirements of the ccNUMA architecture.

At the time we began the SCA operating system work, the V-Class hardware platform was already running HP-UX in a single node configuration. In addition, the SPP-UX operating system, based on the Mach [15] micro-kernel, was running on essentially the same platform (referred to as X-Class) in a multi-node configuration. Leveraging the technology used on the X-Class architecture to expand HP-UX server product line to 128 processors was viewed at that time as a cost effective way to expand HP's UNIX server line.

2. V-Class Hardware Architecture

Figure 1 is a high-level block diagram of a single cabinet V-Class HP computer system.

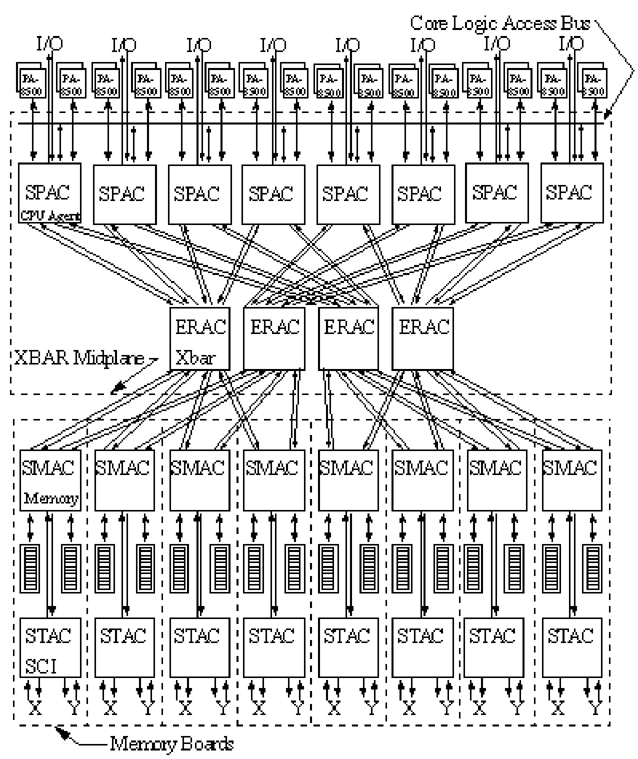
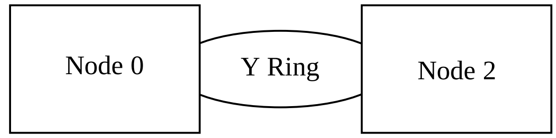


Figure 1: V-2500 Node

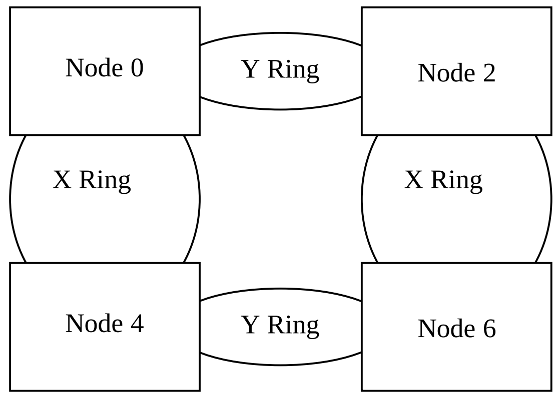
The V-Class system is a crossbar-based machine. On the V-2500, four processors are attached via dedicated busses (termed Runway busses) to a processor agent chip (SPAC). From each SPAC there is attached a PCI interface chip (SAGA) and bus. Memory is interleaved across banks that are managed via memory access controllers (SMAC). A single node V-2500 system is compliant to the PA-RISC platform architecture, implementing a coherent I/O SMP platform. A maximum configuration V-2500 consists of 32 CPUs, 8 SPACs, 8 PCI busses (with 3 slots each), and 8 memory boards (SMACs) providing 4 to 32 GB of memory. Not shown in the diagram is the utilities board. It contains resources used in bootstrapping, reset, configuration, and diagnostic functions. This board contains an RS-232C port used to connect to the system console, an Ethernet connection used for manufacturing and system diagnostic purposes, an LCD display, non-volatile and flash RAM used to hold system firmware and configuration settings, and static RAM used by the system firmware. Each SPAC has access to the utilities board of its containing node.

At the bottom of Figure 1, we see that a Toroidal Access Controller chip (STAC) is attached to each SMAC. This chip implements a variation of the Scalable Coherent Interconnect (SCI) protocol over one to two rings. The STACs and rings are sometimes referred to as the Coherent Toroidal Interconnect (CTI). This interconnect allows V-Class systems to be scaled to multiple nodes (where each node is a cabinet of up to 32 CPUs). Most processor or I/O transactions are conducted through a SPAC, through the crossbar and to a SMAC. If the transaction refers to a component not on the local node, the SMAC forwards the request to its corresponding STAC that in turn traverses the correct ring to a remote node STAC. On the remote node, the STAC sends the transaction to its associated SMAC where it is resolved or forwarded through the remote crossbar port to the appropriate component. Inter-node requests are interleaved across multiple SCI rings in the appropriate ring set (each STAC resides on two rings labeled X and Y). Inter-node memory coherency is managed on a 32-byte line basis. Sample V-Class SCA system configurations are shown in Figure 2. The node numbering contains hardware topological and routing information; this is the reason that nodes in a 4-node system are numbered 0, 2, 4, and 6.

The multi-node V-Class architecture includes a second level inter-node cache, referred to as the CTI cache or the network cache. This is a region of local node memory that is interleaved across memory controllers and is used to cache remote node accesses. The processors cannot access memory used in the CTI cache. Each node typically has 16M to 512M of its local memory used in the CTI network cache. The CTI cache size is



Two Node Configuration



Four Node Configuration

Figure 2: Sample SCA Configurations

configured at boot and is not dynamically configurable; a reset is required to change it.

The precise layout of coherent memory space depends upon the number of memory boards present, size of the DIMMs used on the boards and the amount of memory configured as CTI cache. However, as the node number is encoded into the high order bits of the physical memory address, the memory space of node 0 starts at 0, node 2 starts at 64G, node 4 at 128G, and node 6 at 192G.

All of coherent memory is globally accessible except the memory that is referred to as node private or force node id memory. When the V-Class system was architected, it was anticipated that each node would need some local memory below 4G for firmware and specialized software. To address this need, a special memory access mode known as force node id was introduced. This feature causes the first 128M of each node to be locally mapped, providing firmware with desired local 32 bit-accessible memory on all nodes but resulted in a peculiar physical aliasing property. Consequently, these regions of memory are only accessible by CPUs on the local node.

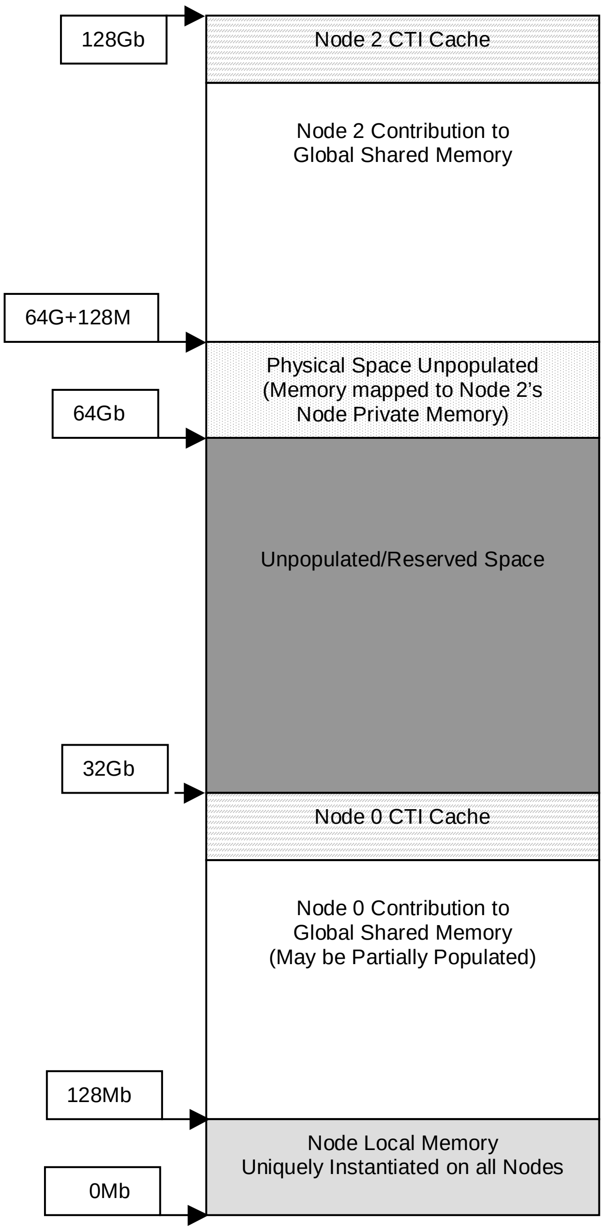


Figure 3: SCA Physical Memory Space

Figure 3 shows the SCA coherent memory address space.

3. SCA Software Architecture

The software development work to support HP-UX on the SCA platform was determined by asking the simple question, "How is a Multi-Node V-Class complex not like a symmetric multi-processor (SMP) system?" and addressing the anomalies so identified. The following discussion summarizes these anomalies and the steps taken to address them.

3.1. Device Management and I/O

One software design goal is to maintain kernel independence from the details of the underlying NUMA system. However, for scalability reasons, it is sometimes necessary to be able to know about the NUMA aspects of the system. To reconcile these conflicting goals, we used an abstraction termed a *locality domain* to provide a rough gauge of "local" versus "remote." A locality domain is a collection of resources including processors, memory, and possibly I/O devices. Equal latency to memory is the characteristic relating resources grouped into a locality domain. On a V-Class SCA system, locality domain boundaries always correspond exactly to the boundaries of a physical node, but the abstraction allows future architectures to define these boundaries more flexibly.

So that this locality domain abstraction could have long-term applicability to other hardware products, it is desirable to prevent exposing hardware-independent code to hardware specific concepts such as the non-sequential node numbers of the V-2500. Therefore, a logical mapping was instituted to insulate high-level software from the hardware-dependent node numbering of V-Class.

The design resulting from these considerations compartmentalizes the hardware node numbering in a low-level hardware-dependent layer, exporting a sequential locality domain or "logical node" numbering to higher-level software when such knowledge is required. This logical node numbering results in a four-node system with logical nodes 0, 1, 2, and 3.

In the following discussion of I/O we will discuss two different methods of identifying I/O devices: Hard Physical Address (HPA) and hardware path. A device's HPA is a unique 64-bit address used internally by system software to identify a device and (in most cases) to actually map it into the processors' address space. We will also discuss hardware paths, which identify devices by their relative location in the system hierarchy of adapters, controllers, and device instances. The hardware path identifies each element in the system with a hardware address that is unique with respect to its superior hardware element; it is not constrained to fit uniquely in a 64-bit address.

Software normally performs I/O on PA-RISC systems by manipulating Control and Status Registers (CSRs), which map into the processors' address space. On a Multi-Node SCA complex, many of the CSRs are not globally accessible across nodes. PCI I/O controller CSRs are accessible only by CPUs on the node containing that PCI controller. Utilities board functions are accessible only from the node containing that utilities board. Most other non-CPU system Core Electronics Complex (CEC) registers are globally accessible. The CPU and most system gate arrays may be addressed globally with unique addresses, as the node number is encoded within the physical address. However, PCI CSRs are mapped node private only; they are not unique complex-wide. For example, all PCI card CSRs for slot 0 of the first PCI bus of each node will have identical physical addresses.

For I/O devices accessed through HP-UX drivers dealing with this aspect of the architecture requires driver software to use process management primitives to migrate its execution to a CPU on the node containing the device to be accessed.

For block devices as accessed via the file system, the solution for binding to the correct node took advantage of existing code that preferred (for cache efficiency reasons) to run the base level code of a device driver on the processor that is designated to handle interrupts for that device. The preference, previously implemented as optional by this I/O forwarding code, simply was made mandatory for the SCA system.

Two sub-cases must be considered to provide direct access to block and character devices, as the kernel supports both legacy uniprocessor drivers as well as multiprocessor-safe drivers. For multiprocessor safe drivers, the I/O system was modified to assure that the interrupt delivery is targeted to a processor on the correct node (that is, the node containing the device) when initializing the device. Base level processing for a multiprocessor-safe driver is bound to a processor on the correct node at the time a request is made; this binding lasts only for the duration of a specific request but has the tendency (unless the requesting process has otherwise specified a processor binding) to migrate processes to the node containing I/O devices they use most heavily.

For uniprocessor drivers, HP-UX provides an emulation wrapper that binds all such drivers to the first processor booted (termed the *monarch* processor). By so doing, this wrapper technique forces traditional uniprocessor serialization when executing base level requests, and by forcing the interrupt handlers to run on the monarch processor as well, provides a very simple paradigm to protect against reentrancy. For the SCA architecture, however, this model had to be improved or uniprocessor driver support could not be provided.

The existing uniprocessor binding mechanism could not simply select the monarch as the emulated uniprocessor target on an SCA system due to the restriction that I/O CSR access is usually not possible across nodes. We attempted to create a generic mechanism that by using a semaphore per device manipulated in a generic wrapper could provide the serialized semantic required by uniprocessor drivers. After providing the serialization required, we could use the process management primitives to bind the executing thread to the appropriate processor. This seems at first to be a simple task, but when a driver sleeps (for instance, awaiting input from a serial port), the generic semaphore is still held by the driver and as a result, no other process can enter the device driver. A semaphore type that is automatically released when the holder sleeps can solve this problem, but an efficient implementation of such a semaphore type could not be provided in the time available. As a result, support for uniprocessor device drivers could not be maintained for SCA systems.

There are restrictions on access to the utilities board functionality that are closely related to those described on CSR accesses. The utilities board resides in non-coherent system CSR space and is accessible only by CPUs on the local node. Therefore, CPUs on one node cannot access the console, NVRAM, or firmware storage areas on another node.

The NVRAM and firmware storage restrictions do not greatly impact the kernel. The kernel retrieves data from firmware only through architected firmware interfaces. The SCA system design calls for all significant firmware configuration data to be stored on node 0 only. By using appropriate scheduling primitives, software assures that the firmware interfaces are invoked from node 0 only, and so this restriction is made a non-issue.

The utilities board also contains the serial port used as the console. Just as is done for other HP-UX device drivers, the console driver uses scheduling primitives to bind to a CPU on the correct node, which for the HP-UX system console is always node 0. (There are serial ports on each node's utilities board, but those on nodes other than 0 are normally unused).

A reliable global reset is required to reboot the system; without a reliable reset, one or more processor may not participate correctly in the next system boot. The V-Class hardware reset mechanism is implemented by writing to a specific CSR on the system utilities board, and it resets only the local node. But hardware does not provide a system-wide atomic reset on a Multi-Node SCA system. No HP-UX kernel change was required to address this anomaly. Instead, the V-Class Processor Dependent Code (PDC) firmware was modified so that a firmware-initiated reset request would reset the entire complex, rather than a single node. The firmware accomplished this by using a diagnostic communication interface between nodes of which the kernel is totally unaware.

The V-Class hardware implements two distinct methods of addressing non-I/O device CSRs, referred to as node-local and global modes. When HP-UX was initially ported to the single-node V-Class platform, a decision was made to use the simple node-local addressing mode for processor HPA (Hard Physical Address) access, as well as other ASIC CSR accesses. This addressing format is useful for accessing CSRs on the current node of execution, but cannot access CSRs on remote nodes.

Inter-node access requires a more general global address format for CSRs. This uses a wide-mode (64-bit) address format that the underlying single-node firmware does not supply.

In addition to correct CSR hardware addressing, each module must be uniquely identified by its HPA. The HPA returned for a memory module is a Coherent Memory address. The HPA returned by firmware for CPU modules during device discovery is the physical address of the CPU's CSR block. These CSRs are located in the non-I/O CSR space. The path through the crossbars is included in the address. Since the kernel does not know how to route an address to a remote node through the crossbars, it relies on the firmware to provide it with an HPA that will work for the platform configuration. Because the path can be different from one node to another, it is required that all nodes of a Multi-Node V-Class platform are fully populated with SPACs, SMACs, and STACs for this to work correctly.

The address space used to access PCI I/O cards and core device I/O CSRs is termed Local I/O Space (core devices are those built into the system, not attached via a peripheral card). Access to this space is available only to CPUs on the same node as the PCI bus adapter or core I/O device. The method used to generate addresses for the local I/O space does not permit the node number to be embedded into a 64-bit or 40-bit physical address. However, the HPA returned by PDC firmware during device discovery for these modules is not a valid CSR address. Therefore, the HPA is used only to identify the module and not to access CSRs. This allows us to treat the address as if it were a coherent memory address for the purposes of generating a unique 64-bit HPA.

To support V-Class SCA I/O, a hardware path must uniquely identify each I/O device. The hardware path is generated from the device path supplied by PDC firmware in response to a PDC_SYSTEM_MAP command during module discovery. On a single-node system, the device path returned by PDC does not contain an indication of the node number containing the module. It is possible - in fact, quite likely - that when two or more nodes are combined to form an SCA system, two or more devices will have the same path, making it impossible to distinguish between them. Therefore, an indication of the physical number of the node containing the I/O module has been encoded in the first component of each device in an SCA system.

For example, the hardware paths for single node V-Class devices have the I/O adapter number - the component called the SAGA on V-2500 systems - as the first component of the hardware path. There may be up to 8 adapters per node, numbered from 0 to 7. Each SAGA controls up to 3 PCI slots, numbered 0 to 2. For instance, 2/1/0.4.0 would refer to a SCSI disk attached to SAGA 2, PCI slot 1, SCSI id 4. If this same SAGA/slot/SCSI-id occurs on node 2, adding the node number times 32 to the I/O adapter number uniquely identifies that adapter within the complex. Extending the previous example, 66/1/0.4.0 would represent SAGA 2 of node 1, slot 1, and SCSI target 4.

The HP-UX kernel typically derives system timing information from an internal control register that implements an interval timer (termed the *itmr*). Each processor contains such an itmr, and in almost all HP PA-RISC systems, these itmrs are synchronized. On an SCA V-Class system, the interval timers (itmrs) within a node are all generated from a single crystal clock source and so are synchronized within that single node. However, the clocks are separate for each node. This means that the itmrs cannot be used to generate synchronized timing information across nodes of the complex, as they can drift with respect to one another over time.

Some code needing timing of events precisely across the system that previously used the processor itmr registers was modified to use a V-Class specific feature. The V-Class hardware provides a 1 microsecond period counter known as the Time of Century counter. This counter is 64 bits wide and globally synchronized in hardware. The maximum inter-node skew is 1 microsecond. This provides system software with a globally consistent view of time without requiring synchronized processor clocks across nodes.

For generic time accounting purposes, we did not want to introduce V-Class specific code throughout the kernel. To support a few other systems with asynchronous clocks, the kernel already had the ability to compensate for itmr drift by maintaining a synchronized per-processor bias tracking the drift of each processor's itmr against the master (or monarch) processor. We were able to take advantage of this existing mechanism and extend the synchronization code to use the Time of Century hardware as a global reference to precisely compensate for the clock drift between nodes.

The PDC firmware of each node had no visibility of the other nodes' resources; each node's PDC initially operated totally independently. It was immediately obvious when starting the SCA kernel design that compensating for this major "non-SMP-ness" explicitly in the kernel would become very intrusive if handled with ad hoc methods. Consequently, a "front-end" or facade layer was designed to produce PDC firmware functionality supporting multi-node SCA in a transparent manner. This component is called Multi-Node PDC layer. It is linked into the HP-UX kernel and mapped specially into the shared memory region. The Multi-Node PDC Front-End provides the following functionality:

a. A unified module table, with some limited exceptions: only node 0 core I/O modules are included; as a result the kernel has no access to console or diagnostic LAN ports of non-zero nodes, and non-zero node memory modules do not include node private memory.

b. The node local CSR address format used by underlying PDC firmware is modified to use the complex-wide global addressing format containing an explicit node number

c. Device paths encoded to include the module's node number.

d. Runtime support for PDC requests that require a remote call to another node.

e. Ability to perform inter-node memory copies from one node's local memory to another node's local memory for specialized initialization and crash dump requirements.

The resulting layer of software should be considered logically as a part of the system's firmware. Whereas it is linked into the kernel image, after an initialization call, it is never directly called again. Instead, it replaces the contents of the location used to vector into the PDC firmware, directing firmware requests to its own entry and then chaining, where appropriate, to the underlying PDC firmware. This design allows the Multi-Node PDC layer to be updated when the kernel is updated. This design choice was made to simplify both development of the firmware layer and field upgrades of existing single-node systems to SCA configuration in the field, and its design succeeded in so doing. However, it requires a logical separation of responsibility so that the Multi-Node PDC layer cannot make calls back into the kernel; if this is violated (for instance, by inadvertent use of a compiler-generated runtime library support function call), the firmware layer may fail during system boot of non-zero nodes. Maintaining this separation has proven to be difficult over time, and since the consequences of violating the required layering are both profound and difficult to diagnose, the design choice is questionable in hindsight as a permanent part of the kernel.

3.2. System Characteristics

One very significant way in which a V-Class SCA system differs from a traditional SMP system is that there exists a special memory region on each node termed *force node-id* or *node private* memory (as discussed above in Section 2) This memory consists of the first 128M of each node, and is accessible only by CPUs on the containing node. The node private memory is unique in its combination of attributes:

a. Spatial locality: In a NUMA sense, access is guaranteed to be efficient because this memory is by definition local to the accessing node.

b. Limited accessibility: only processors on the node may access this memory. This is enforced by the hardware.

c. Uniquely instantiated per node: this refers to the fact that a given address of memory may contain different values on different nodes. This can be very useful for variables that are location dependent; an obvious example is a single location containing the node number for each node. It allows the use of a single variable in the code to be used differently depending upon the node context of the accessing CPU.

The kernel text segment is replicated, one copy per node, in an effort to reduce instruction fetches between nodes. This lowers latency and at the same time does not pollute the CTI caches with kernel text. The node private memory of each node is used to contain that node's copy of the kernel text. (The obvious but much less efficient alternative to this replication would have required loading the kernel at or above the 128Mb boundary, in globally shared memory.)

To accomplish the replication, a new function *rm_ktext_replicate*() is called during system boot. If running on a platform other than Multi-Node V-Class, *rm_ktext_replicate*() performs no action and returns immediately. On Multi-Node V-Class, the function copies the kernel text segment from the node private memory area on node zero to the node private memory area of all other nodes in the system.

The actual replication of kernel text is done near the very end of the initial phase of the boot sequence. The replication is done after the point at which the kernel has performed all platform dependent optimization decisions. In addition to copying the kernel text itself, page zero (which contains data shared between the kernel and PDC firmware) also is replicated to all other nodes. The memory copy operation is done via a firmware interface added specifically for V-Class SCA systems.

The layout of text and data within the SCA kernel is different than that of a traditional HP-UX kernel. As part of exploiting the node private memory architecture to replicate the kernel text, the SCA kernel is linked specially, specifying an alternate memory map to the linker. A mapfile specifies that the text segment remains at a low address in physical memory, and so is in node private memory, but the data segment is placed just above 128M, and so is in global shared memory.

The HP PA-RISC processor performs translations from virtual to absolute addresses using a hardware structure called the Translation Lookaside Buffer (TLB). The purge instruction TLB (*pitlb*) and purge data TLB (*pdtlb*) instructions are used to manage processor TLB entries. On the SCA V-Class platform, these transactions are not broadcast between nodes. Another limitation is that there can only be one outstanding purge per node.

To address these architectural limitations, the kernel implemented an inter-node RPC (Remote Procedure Call) interface that is initiated by a software interrupt when a TLB modification is performed. The interrupt is serviced by dedicated kernel handler code to perform the needed purge operation on each node. Challenges addressed in the design include the need to make the service code very efficient and responsive, yet minimize intrusion of the software on architecturally conforming HP systems.

Processor caches in all multiprocessor systems must be managed with care. For PA-RISC based systems, the flush instruction, flush data, and purge data cache instructions (*fic*, *fdc*, and *pdc*) are used to maintain consistent cache state among CPUs. In the SCA V-Class platform, these flush instructions are not broadcast between nodes. The instruction cache is a particular problem, as there does exist a method to cause a global data cache flush of a line of memory. There is a special host op instruction supported by the PA-8x00 processors that is passed nearly transparently to the system gate arrays. This instruction will globally flush a line of memory from all data and network caches. It is referred to as the NCFG instruction (Network Cache Flush Global). The host op instruction is not precisely interchangeable with the existing architected flush instructions, so special software handling was required.

The instruction cache anomaly required handling user space pages via what we called *write-execute demotion*; a page of user accessible memory could not simultaneously be granted both write and execute permission. Such pages are internally recorded in the virtual memory system as allowing write and execute, but set up in the hardware as only write or execute. The permissions are switched from one to the other in the kernel's trap handler upon an attempt to write an executable page or to execute a writable page. There was concern that this mechanism might be too inefficient for applications that engage in lots of runtime patching. This has not been found to be a problem based on our limited experience, as most such instruction patching or generation occurs only once during a process's lifetime. However, the implementation does produce an inconsistency that the application can detect via the *probe* instruction, which sees the "real" hardware permission – only write or execute - rather than the emulated write and execute permission. Consequently, we created library function interface for use by any application that needed to get the correct result on all architectures. This interface is written so as to query the virtual memory system for the permissions on the SCA V-Class system, rather than trusting the *probe* instruction.

Kernel pages can be handled more simply, as the kernel functions used to flush caches and purge the TLB could be modified and/or patched as required. The modifications to use the RPC mechanism alluded to above were in fact implemented using a single paradigm, sharing data structure definitions and software strategies. When a cross node purge or flush operation is required, the processor generates a pre-designated interrupt type (which differs for purge and flush) to a pre-designated partner processor. The data structure involved records the address range and operation to be affected. The interrupt is serviced at a high priority level; the requestor, in order to maintain the required architectural semantics of the flush or purge operation, must busy wait until the partner completes the requested operation.

There are rare - but quite possible – cases wherein matched pairs of processors on two different notes, each the pre-designated RPC partner of the other, could encounter a deadlock. If one processor holds a spinlock while requesting a flush operation (as sometimes occurs in the page directory manipulation) and its RPC partner processor is in fact waiting for the same spinlock, the result would be a deadlock, since the waiting processor has disabled interrupts in the spinlock acquisition code. To avoid this potential deadlock, we modified the kernel's spinlock routines so that threads waiting for spinlocks periodically check for outstanding remote TLB purge or cache flush requests and service any so detected, thereby avoiding deadlock.

Finally, the virtual memory system was modified to reflect the ccNUMA memory system attributes. Roughly stated, the latency to lines of memory within a node is 500 nanoseconds. This can blossom 2 to 4 times for remote accesses (2.2 microseconds). In addition, access time is not strictly bounded; it can be dependent upon SCI ring contention. The actual coherency state of a line (home, dirty, read shared) and the operation being performed (read, write, flush) on the line also impact the latency.

As more NUMA architectures are anticipated in the future, the virtual memory system was enhanced with knowledge of the locality domain abstraction to allocate memory as efficiently as possible. A first-fault strategy was chosen to instantiate pages for a process from the memory pool of the first locality domain (node in this case) to generate a fault on the process's address space.

The HP-UX process management system was enhanced to efficiently employ the NUMA characteristics of the V-Class SCA architecture. The scheduler was given knowledge of the system's locality domains so that it could effectively place newly created threads as well as appropriately avoid migrating threads across locality domains.

The primitives available for locating threads and processes on specific locality domains and CPUs within the system were enhanced. Particular attention was paid to needs by the I/O system, which may need to migrate threads onto processors appropriate to the node containing a particular physical device.

A new utility *mpsched* was provided to explicitly control thread and process placement within the complex. The policies provided are

a. Round robin launch policy, in which launches of child threads or processes alternate among all localities until all localities have been selected once, then starts over as needed.

b. Least loaded launch policy, under which child threads or processes are launched on the least loaded node in the system at the time of creation.

c. Fill first launch policy, wherein successive processes or threads are launched on the same locality domain as their parent until one has been launched on each processor in the locality domain; at that point, new threads are created on the next locality domain.

d. Packed launch, under which successive threads or processes are launched on the same locality domain as their parent; a different locality domain is never selected.

The default scheduling decisions, at a coarse level, attempt to start new processes on the least loaded node and new threads of an existing process on the node containing the creating thread. In addition, the load is periodically balanced among nodes when the imbalance exceeds some threshold or some node becomes idle.

4. Related Work

One finds a consensus in published literature that approaches beyond traditional SMP are necessary to achieve cost effective scaling of parallel computer systems. The approaches to implement such scaling alternatives vary greatly. Descriptions of hardware implementations in the literature include the directory-based ccNUMA hardware designs of DASH [5] and Alewife [6] as well as Cache Only Memory Architecture (or COMA), exemplified by Kendall Square's KSR-1 and analyzed in [7] and [8]. A hybrid of ccNUMA and COMA, termed Reactive NUMA, is described in [9]. An approach using a Cache Coherent Network of Workstations (ccNOW) is described in [10]. Yet another approach is to omit caches altogether, compensating for the resultant latency by instruction-level parallelism, exemplified by TERA [16]. Studies of memory and process placement in NUMA systems have been extensively described and analyzed in [11], [12], [13], and [14].

Operating systems for NUMA architectures may be monolithic SMP systems modified for NUMA such as our HP-UX port, or micro-kernel based. Micro-kernel based solutions have included Mach [15]. The SPP-UX Mach-based operating system from the HP Convex Division used a message-passing paradigm to communicate between distinct micro-kernel instances running on each node. Other proprietary operating systems are described in many of the above hardware references.

5. Conclusions

In this paper we described the issues encountered and the solutions used for the SCA V-Class project. Some of the areas addressed are platform-specific and unique to the V-Class architecture. However, other areas related to NUMA memory management techniques and process and thread management in a NUMA environment can be leveraged on future HP platforms.

Internally, the V-Class SCA platform is being used to study and improve the performance and the scalability of the HP-UX Operating System. We expect that our experience and learning from the V-Class SCA platform will enable the HP-UX operating system to scale to even larger configurations in future.

6. References

[1] *PA-RISC 2.0 Architecture*, Jerry Kane, ISBN 0-13-182734-0, Prentice Hall; also available online at http://devresource.hp.com/devresource/Docs/Refs/PA2_0/index.html

[2] PA-RISC 2.0 Firmware Architecture, http://devresource.hp.com/devresource/Docs/DocLibrary.html

[3] HP V-2500 Reference Guide, online at http://docs.hp.com/HP-UX/systems

[4] The Evolution of the HP/Convex Exemplar; Brewer, T., Astfalk, G. Convex Div., Hewlett-Packard Co., Richardson, TX, USA, *Proceedings of Compcon 1997*.

[5] The DASH Prototype: Implementation and Performance; Daniel Lenoski, James Laudon, Truman Joe, David Nakahira, Luis Stevens, Anoop Gupta and John Hennessy; *Proceedings of the 19th Annual International Symposium on Computer Architecture*, 1992, Pages 92 – 103.

[6] The MIT Alewife Machine: Architecture and Performance; Anant Agarwal, Ricardo Bianchini, David Chaiken, Kirk L. Johnson, David Kranz, John Kubiatowicz, Beng-Hong Lim, Kenneth Mackenzie and Donald Yeung; *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, 1995, Pages 2 – 13.

[7] Comparative Performance Evaluation of Cache-coherent NUMA and COMA Architectures; Per Stenstrom, Truman Joe and Anoop Gupta; *Proceedings of the 19th Annual International Symposium on Computer Architecture*, 1992, Pages 80-91.

[8] An Empirical Comparison of the Kendall Square Research KSR-1 and Stanford DASH Multiprocessors; J. P. Singh, T. Joe, J. L. Hennessy and A. Gupta; *Proceedings of the Conference on Supercomputing '93*, 1993, Pages 214-225.

[9] Reactive NUMA: A Design for Unifying S-COMA and CC-NUMA; Babak Falsafi and David A. Wood; *Proceedings of the 24th International Symposium on Computer Architecture*, 1997, Pages 229-240.

[10] The S3.mp Architecture: A Local Area Multiprocessor; A. Nowatzyk, M. Monger, M. Parkin, E. Kelly, M. Browne, G. Aybay and D. Lee; *Proceedings of the 5th Annual ACM Symposium on Parallel Algorithms and Architectures*, 1993, Pages 140-141.

[11] NUMA Policies and Their Relation to Memory Architecture; William J. Bolosky, Michael L. Scott, Robert P. Fitzgerald, Robert J. Fowler and Alan L. Cox; *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 1991, Pages 212-221.

[12] Exploiting Operating System Support for Dynamic Page Placement on a NUMA Shared Memory Multiprocessor; Richard P. LaRowe, James T. Wilkes and Carla S. Ellis; *Proceedings of the Third ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, 1991, Pages 122-132.

[13] An Analysis of Dynamic Page Placement on a NUMA Multiprocessor; Richard P. LaRowe, Mark A. Holliday and Carla Schlatter Ellis; *Proceedings of the 1992 ACM SIGMETRICS and PERFORMANCE '92 International Conference on Measurement and Modeling of Computer Systems*, 1992, Pages 23-34.

[14] Dynamic and Static Load Scheduling Performance on a NUMA Shared Memory Multiprocessor; Xiaodong Zhang; *Proceedings of the 1991 International Conference on Supercomputing*, 1991, Pages 128- 135.

[15] An extensive archive of published and unpublished papers on Mach is found at:
http://www.cs.cmu.edu/afs/cs.cmu.edu/project/mach/public/www/mach.html

[16] Exploiting Heterogeneous Parallelism on a Multithreaded Multiprocessor; Gail Alverson, Robert Alverson, David Callahan, Brian Koblenz, Allan Porterfield and Burton Smith; *Proceedings of the 1992 International Conference on Supercomputing*, 1992, Pages 188-197.