555

VLSI Assist For A Multiprocessor

Bob Beck
Bob Kasten
Shreekant Thakkar

Sequent Computer Systems
15450 S. W. Koll Parkway
Beaverton, Oregon 97006-6063

Abstract

Multiprocessors have long been of interest to computer community. They provide the potential for accelerating applications through parallelism and increased throughput for large multi-user system. Three factors have limited the commercial success of multiprocessor systems; entry cost, range of performance, and ease of application. Advances in very large scale integration (VLSI) and in computer aided design (CAD) have removed these limitations, making possible a new class of multiprocessor systems based on VLSI components. A set of requirements for building an efficient shared multiprocessor system are discussed, including: low-level mutual exclusion, interrupt distribution, inter-processor signaling, process dispatching, caching, and system configuration. A system that meets these requirements is described and evaluated.

1. Introduction

The Sequent Balance® system[5] is a shared-memory tightly-coupled multiprocessor system which can contain two to thirty 32-bit microprocessors. Each processor has a private cache as well as a small local memory to hold frequently used kernal routines. The cache coherency is maintained using a protocol based on write-through policy. Application programs run in a single memory shared among all processors, and so have the potential to execute on any processor at any time.

The Balance is a *symmetric* multiprocessor system that makes no distinction amongst its processors. Processors are viewed as another resource that the operating system manages. All processors can run application and operating system code. Processes may be dispatched on any processor in response to process or system needs; a process often executes on several processors during its lifetime, without being aware of the transition. The Balance shares the advantages of *master-slave* systems, but avoid the problems associated with it. All processors can execute system services, no context switch is necessary for system calls or traps and interrupt handling may be distributed among the processors to avoid overloading any single processor. Also, since there is no master processor, the system can run with some loss in performance if at least one of its processors is functional.

® Balance is a trademark of Sequent Computers Systems

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

Building a multiprocessor computer system requires solution to a set of problems not present in the construction of a uniprocessor system. We discuss some of the more interesting of these problems and present our solution.

2. Traditional Multiprocessor Problem Areas

Multiprocessor computer systems introduce some problems not present in uniprocessor systems including low-level mutual exclusion, hardware interrupt distribution, inter-processor signaling, additional system configuration issues, process dispatching among processors, cache policies, and pragmatic concerns such as ability to integrate device drivers into the multiprocessor environment. This section discusses these problems in more detail.

2.1. Low-Level Mutual Exclusion

Uniprocessor systems typically handle this problem by observing that there is no real concurrency in the operating system and the only reentrancy in the operating system is the result of servicing hardware interrupts. Disabling some or all interrupts at appropriate times is enough to avoid *concurrent* (really, reentrant) access to system data structures. This simplification avoids much of the concurrency issue within the operating system. UNIX® on a uniprocessor uses this technique.

A multiprocessor system differs in that it allows more than one process to be executing operating system code at the same time; multiple processes may be attempting to access and change system data structures concurrently. Since there are potentially multiple concurrent threads of

® UNIX is a trademark of AT&T Bell Laboratories.

© 1987 ACM 0-89791-238-1/87/1000-0010 $00.75

execution in the operating system, interrupt-masking techniques fail to fully solve the problem. Explicit control over concurrent accesses to system data structures is required. This control takes the form of **test-and-set** variables and higher-level synchronization primitives built from these.

A test-and-set variable is typically implemented with a hardware mechanism that insures a read/modify/write of the variable is atomic; regardless of the number of concurrent accesses attempted, exactly one at a time will proceed. The variable can be set or cleared; a request to set the variable returns the previous state (set or cleared), and sets it. Using a test-and-set variable to control access to a data structure results in **busy-waiting** i.e., a process spins in a program loop constantly re-testing the variable for *clear* state. If processes obey an appropriate locking protocol, at most one process can access a given data structure at a time.

Since busy-waiting is wasteful of processor cycles, it is desirable to avoid it whenever possible. Processes should hold test-and-set variables for as short a time as possible. Longer term resource allocation and waiting for events require another mechanism, which allows processes to block (i.e., sleep) until access is possible. Counting semaphores can be built from the test-and-set primitive, and provide the necessary semantics.

It is beyond the scope of this paper to discuss the detailed semantics and higher-level uses of these primitives.

2.2. Interrupt Distribution

Uniprocessor systems deliver interrupts to the single processor in the system. When multiple processors are available, it is desirable to distribute the interrupt load among the processors. This distribution can be static or dynamic; static distribution assigns interrupts to fixed processors at hardware and/or software configuration time. Dynamic interrupt distribution allows any processor in the system to handle any interrupt depending on dynamically changing system load characteristics.

Dynamic distribution of interrupts has several advantages over static distribution:

- There is no complex assignment issue (which interrupts to which processor); the interrupt load is automatically distributed.
- Hardware components need not be tailored to a specific environment (e.g., using interrupt-level jumpers).
- No single processor is overloaded with interrupts due to a poor static assignment.
- It is possible to arrange for idle processors (if any) to accept interrupts, thus automatically off-loading interrupt processing from processors doing useful work.
- Interrupts may be fielded by processors executing lower-priority processes.
- Processors may be removed from service without removing the server for any hardware interrupt source.

2.3. Inter-Processor Signaling

It is sometimes necessary for one or more processors in a multiple processor environment to signal another. Examples include:

- Telling a processor to re-dispatch itself, if a higher priority process has become ready to run.
- Delivering a software signal. Delivering the signal may necessitate forcing the signaled process to enter the operating system so it can accept the signal.
- Telling a processor to remove itself from service. The processor must stop executing a process (if necessary) and turn itself off.
- Initiating low-priority interrupt services such as network protocol routines, lower-priority time-of-day clock services, time-slicing, etc.

Another, less likely, occurrence is a system panic (automatic shutdown), induced by a hardware- or software-detected inconsistency in the system's behavior or state information. To shut down the system, the processor that sensed the condition must force other processors to stop executing.

2.4. Process Dispatching

Ideally, in an N processor system the N highest priority processes are executing at any point in time. If fewer than N processes are executable, the remaining processors are idle and immediately available to run processes.

When a process context-switches, it may next execute on a different processor than before the context-switch. This results in the cache of the new processor having little or no useful context for the process. Since filling a processor cache places a burden on the system bus it is important to avoid unnecessary context-switches, as these in general affect system performance. Of course, an unnecessary context-switch also wastes processor cycles.

These goals are difficult to achieve in practice, since dispatching decisions are made using a snapshot of system state information and this state may change before the results of the decisions are realized. A heuristic algorithm keeps the right set of processes running, while avoiding unnecessary context-switches whenever possible.

Since the number of processors can be less than the number of processes that want to run, some form of time-slicing may be used to multiplex processors as in a time-shared uniprocessor system. The time-slice algorithm can distribute the time-slice load among processors running the lowest priority processes. This also helps reduce some context-switches.

2.5. System Configuration

To help reduce complexity and cost of running a multiprocessor system, it is desirable that a single configuration of an operating system be capable of running on as many hardware configurations as possible. This allows a single set of software to run on a variety of machines, simplifying the maintenance problem. Although this problem is not new, the presence of multiple processors in the system aggravates it. Additionally, to avoid the

requirement of source code for configuration (as required in 4.1bsd), the system must be configurable from a binary-only copy.

A mechanism is needed to recognize the various major hardware components in the system (i.e., processors, memory, input/output controllers). Ideally, this mechanism is independent of the particular type of hardware, as this simplifies the system initialization code. Whenever possible, the operating system should allocate data structures to represent these hardware components during system initialization rather than compiling in hard-coded constants. For example, a single operating system binary should boot and run on a system with any number of processors. This simplifies binary configuration and makes the software more independent of particular hardware configurations.

An additional issue is determining which hardware components are statically configured (i.e., at boot-time) and which are dynamic (i.e., at run-time). To assist in debugging and system tuning, processors should be dynamically configurable (i.e., brought in and out of service) during normal system operation. For example, it is often easier to debug a device driver in a single-processor environment and then test the multiprocessor aspects of the driver after it is known that the driver functions properly with the hardware. Dynamically altering the size of system memory and input/output controllers are less well motivated.

2.6. Cache Memory

Even in uniprocessor systems the system bus typically imposes an access protocol that increases the processor's latency to main memory. The main memory may not be fast enough to meet the processors demands even if the bus had no overhead, since memory components fast enough for the processor are too expensive to build the large system memory. As a result, uniprocessor systems often employ a cache to decrease the processor's latency to its more frequently accessed instructions and data. The cache provides a small fast memory, between the processor and the system bus, that can keep up with the processor's demands.

A multiprocessor system increases the need for a cache since there are multiple processors concurrently executing. Often the bus is designed with bandwidth to exceed the needs of an individual processor, but cannot meet the demand that multiple processors create. Additionally, multiprocessor systems frequently employ spin-loops for low-level mutual exclusion primitives and idle-loops; if these loops execute entirely out of processor-local cache, the system bus bandwidth is not degraded.

Many cache policies for multiprocessors systems have been devised[2]. These include write-back (copy-back), write-through, write-once, and hybrids. A related parameter is whether the cache *watches* the system-bus to sense changes in local copies of data. A bus-watching cache simplifies the software and allows low-level spin-loops in the operating system to execute entirely out of cache, relying on the cache to notice the change in value of appropriate state variables.

2.7. Uniprocessor Device Drivers

Throughout its evolution, UNIX has executed on uniprocessor systems. In the UNIX kernel, many algorithms take advantage of the assumption there is only one processor. When moving to a multiprocessor environment, many of these algorithms must be re-evaluated, and changed or rewritten to use more formal techniques (including test-and-set variables and semaphores). All code that is part of the multiprocessor operating system must be evaluated in this manner. Unfortunately, this includes device drivers, which are often written or otherwise supplied by customers of the system as opposed to the developers.

Systems programmers are accustomed to accommodating the hardware dependencies present in a new environment. For example, a driver that ran in a Unibus® environment must be changed to run in a VAX® since the Unibus is accessed differently. The necessary changes for a multiprocessor environment are not as familiar to the UNIX community, however. Thus it is desirable to ease this porting task as much as possible.

One possibility is to insist the system run only a single processor until the driver is fully debugged and tested in the new multiprocessor environment. This is non-optimal in that other processors in the system are not usable during this time, and there may be drivers that do not warrant the investment to bring them to full multiprocessor capability (infrequently used, low throughput, etc.).

A better approach is to accommodate uniprocessor versions of drivers in a multiprocessor environment. This allows the driver to be adapted to the new hardware environment without adding support for multiprocessing, while allowing use of all processors in the system. Ideally, there is no change to the portions of the operating system that call drivers, and no change to the driver beyond hardware dependencies. The operating system must arrange that the driver code run on a particular processor, including the driver's interrupt handlers (thus picking up some aspects of asymmetry). If possible, this processor should be selected at system initialization time to avoid requiring any particular processor being present in the system.

3. System Link and Interrupt Controller

The System Link and Interrupt Controller (SLIC) chip was developed to address some multiprocessor problems. One SLIC chip is coupled with each major component in the system (processor, memory controller, I/O controller), providing support for interrupt distribution, low-level mutual-exclusion, and configuration and error control. A major goal of the SLIC design was to remove many of these concerns from the software, or to provide enough hooks to simplify the problems wherever possible. It was also desired to simplify the system bus, to lower the cost while maintaining high performance and reliability. Figure 2 gives a block view of the SLIC. This section describes the functions of the SLIC. The next section describes how the SLIC addresses several of the multiprocessor problems described previously.

Unibus and VAX are trademarks of Digital Equipment Corporation.

Figure 2: SLIC Block Diagram

3.1. SLIC Messages

SLIC chips in the system communicate with a simple message-passing protocol, using two lines from the system bus for a bit serial, wired-OR medium (the SLIC *bus*). The SLIC bus derives its clock from the system bus but otherwise operates asynchronously; thus, SLIC message traffic is transparent to the system bus, not affecting bus bandwidth. SLIC bus access is arbitrated using wired-OR resolution; in case of contention, the higher priority message automatically wins the arbitration (priority is part of the message encoding). Each SLIC has a system-unique *number*, derived from its host board's position in the system backplane; this number breaks a tie during arbitration. A SLIC message consists of two parts: request and response, as illustrated in figure 3.

s	cmd	srcID	dest	data	parity	accept	rdata	rspID	parity	error

Request Part		Response Part	
s	start bit	accept	interrupt accepted status
cmd	encoded command	rdata	response data
srcID	sender SLIC address	rspID	responder's SLIC address
dest	encoded destination	parity	parity protection
data	8-bit message data	error	SLIC error
parity	parity protection		

Figure 3: SLIC Message Format

A SLIC request message contains a 8-bit command, source address, 8 bits of message data, and a destination address. The response contains a 8-bit data field and status information. The interpretation of the message and response data is a function of the command; several of these are discussed below. The command values are chosen to reflect their relative priority, allowing the wired-OR arbitration to automatically pick the highest priority message if there is contention. The source SLIC address is appended to the command to resolve arbitration ties when two or more SLICs are trying to send the same command message. The SLIC will automatically retry the message if an attempted send looses arbitration on the SLIC bus. Each SLIC sets a status bit to show success or failure in sending the message.

A message addresses one or more SLICs (encoded in the destination field). Each of these SLICs will try to respond to the message if appropriate, by trying to send the response part of the message. It must start responding in the bit-time of the request's parity bit. In the event that more than one SLIC attempts to respond, arbitration is guaranteed by including the SLIC address of the responder in the response. It is possible that no SLIC responds to the message (depending on the command); the sending SLIC records this status in addition to the *message-sent* status. After the message is sent and response received, the software (or other entity controlling the SLIC) uses the status bits to determine the appropriate next action. If the message wasn't received (e.g., because the destination SLIC(s) is ignoring interrupts), the message can be retried. Since proper operation of the SLICs is paramount to system integrity, the system at present panics if any SLIC error is detected.

3.2. SLIC Interrupts

One class of SLIC messages encodes interrupts of various kinds: maskable, non-maskable, and software interrupts can be sent directly to other SLICs or broadcast to a group of SLICs. Interrupts are identified by one of eight **bins**, each corresponding to a processor hardware vector. The bin number is encoded as part of the SLIC command field: bin 7 is the highest arbitration priority, bin 0 is the lowest. The SLIC message data in an interrupt message further subdivides each bin into 256 logical interrupt vectors, allowing interrupt sources to be adequately distinguished. It is stored in a readable SLIC register for the convenience of the operating software.

SLIC provides two basic kinds of interrupt, maskable and non-maskable, corresponding to the interrupt request lines present on most microprocessors. The type of interrupt and bin number are encoded in the command field of the SLIC message. Interrupts are further characterized in how they are directed: to a particular SLIC (**directed** interrupt) or one of a group of SLICs (**group** interrupt; the selection of group is an initialization parameter to the SLIC). All SLICs in a group attempt to accept the interrupt, subject to their current state (more on this below); at most one SLIC will accept the interrupt, using the responding SLIC's address for final arbitration resolution. To allow some software control over arbitration in accepting group interrupts, the SLIC supports an 8-bit register whose upper 5 bits are fully software controlled (the lower 3 bits are randomly generated by the SLIC). This value is placed in the *rdata* field of the response part of the message, as illustrated above (there is no response data returned for an interrupt message, so this field can be used for interrupt acceptance arbitration).

When a SLIC accepts an interrupt, it copies the message data into an internal register, and asserts the appropriate processor interrupt line. The SLIC then refuses to accept another interrupt until the operating software acknowledges it has started its interrupt handler. This also tells the SLIC it has read the message data (the SLIC at present has only one register to hold maskable interrupt message data). Bin 0 differs in that it will always accept an interrupt, logically OR-ing the message-data into the separate ***bin0 message data register***. Software can use this bin to post various kinds of events to selected processors. The current SLIC can be viewed as a subset implementation of a more general model where attributes such as interrupt queuing and message OR-ing are selected on a per-bin basis.

As in uniprocessor systems, there is a need to mask interrupts at various points in time. SLIC supports an 8-bit register, controlled by the processor, in which each bit masks or enables the corresponding interrupt bin for that SLIC. A SLIC with a particular bin masked will not attempt to arbitrate to accept an interrupt in that bin that it might normally accept. Thus, software can arrange that particular sets of interrupts be ignored for specified periods of time. As a result, if a processor or I/O controller attempts to send an interrupt (even a group interrupt), it is possible that the interrupt will not be immediately accepted. This is resolved by having the interrupt source resend the interrupt message. These somewhat non-intuitive semantics make sense in a multiprocessor environment: if a masked SLIC queued the interrupt, it might really take longer to start servicing the interrupt since another processor may unmask its SLIC before the queuing processor starts service. (It also keeps the SLIC implementation simpler.)

Most operating systems accumulate per-process execution statistics, as exemplified by the UNIX notion of *user* and *system* time reported by the /time command. On a multiprocessor system, a single system clock could only support this by broadcasting a periodic interrupt and insuring that all processors accept it. It is more convenient to have each processor generate its own clock. This also has advantages with respect to cache usage: per-processor clocks can be mutually asynchronous, which avoids forcing all processors to change their cache context at the same time (by dropping into the kernel clock handler), thus spreading the system bus load more evenly. To support this, the SLIC implements a programmable internal timer that can be connected to any interrupt bin.

3.3. SLIC Gates

Each SLIC implements a set of 64 binary semaphores, called gates, and supports a set of SLIC commands to atomically test-and-set them. Each SLIC contains a copy of the value of each of the 64 gates at all times. If a processor attempts to *lock* a gate, that processor's SLIC uses its local copy of the gate value to determine if a SLIC-bus message is necessary. If the gate is already locked, the SLIC sets a status bit and sends no message. If the gate is unlocked, the SLIC sends a message to attempt to lock the gate. This message is seen by all SLICs, who set their local copy of the gate to *locked*. The SLIC *lock-gate* message is arbitrated to insure that at most one SLIC sends a lock-gate message at any point in time, thus assuring the atomicity of the operation. Another SLIC message unlocks a particular gate; all SLICs clear their internal copy of the gate. The gate number is encoded in the request data field of the message.

Since gates are used for low-level mutual-exclusion in the operating system, it is important that any *write-behind* data is really in system memory before releasing a gate. For example, data changed while the gate was locked might be in the processor write-buffer on its way to system memory but not yet there. To avoid race conditions that could result, SLIC doesn't send the *unlock-gate* message until the processor write-buffer empties (the write-buffer exports a status bit for this purpose). Thus, when software releases a gate, it has automatically guaranteed the consistency of system memory.

3.4. SLIC Slave Registers

Each SLIC supports a 256 byte local address space, called **slave registers**. These registers are read or written by the SLIC itself or by any other SLIC in the system using two messages directed to a particular SLIC: one to specify the address, the next to read or write the data. Each hardware board in the system implements a subset of these addresses for its SLIC.

3.5. SLIC Implementation

The SLIC has been implemented in a 6000-gate custom CMOS gate array component [4].

The SLIC is accessed via 17 byte-wide locations in its processor's address space. Some of these locations are simple read and/or write registers; for example, local SLIC interrupt mask, arbitration priority register, local SLIC number (provides processor identification as well), timer control registers, etc. Sending a message involves loading several registers with the message data, encoded destination address, and any other necessary data, then loading the command register to start the command. The software then spins looking at a status bit to see when the command completes (very few spins are needed). Once the SLIC request is complete, the status register contains bits describing aspects of the result: was the message sent, was it accepted, were there any errors, etc. Software can retry the command, return a success/fail indication, or whatever makes sense. The programming model relies on spin-waiting for results since each SLIC command completes quickly with its status; servicing an interrupt would impose unreasonable overhead.

The code example in figure 4 illustrates the programming model with a simplified version of the code used to send a maskable interrupt. Code to send a non-maskable interrupt, send a software interrupt, read or write a slave register, or request or release a gate is similar.

```
sendinterrupt(dest, bin, data)
{
        /*
         * Set up target of interrupt (dest),
         * and "vector" number (data).
         */

        slic->sl_dest = dest;
        slic->sl_data = data;

        /*
         * Set up Command and Bin Number,
         * wait for interrupt to be sent,
         * and loop until interrupt accepted.
         */

        do {
                slic->sl_cmd = MINTR | bin;
                while(slic->sl_stat & BUSY)
                        continue;
        } while((slic->sl_stat & OK) == 0);
}
```

Figure 4: Send Interrupt Code Fragment

4. SLIC Solutions to Multiprocessor Problems

The SLIC subsystem provides a low-cost solution to some traditional multiprocessor design problems discussed section 2. The SLIC subsystem supports system-wide interrupt control, mutual exclusion primitives, and a mechanism for inter-module communication. This functionality resolves problems regarding processor synchronization, dynamic interrupt distribution among processors, uniprocessor driver support, inter-processor communication, dynamic load balancing of processes among processors, and dynamic system configuration.

4.1. Processor Synchronization

A basic necessity in a multiprocessor system is a fast, efficient synchronization primitive. The SLIC supports this function with a set of 64 single-bit **gates**. Gates are logically equivalent to a test-and-set primitive and are spin-oriented. That is, the process loops requesting the gate until it is acquired. In many machines the test-and-set function is implemented via an interlock signal on the system bus and/or in the memory controller itself. This imposes extra complexity in the system bus architecture. The use of SLIC gates as the synchronization primitive has two main advantages. First, gate operations are via the SLIC bus. Since the SLIC bus is separate and asynchronous to the system bus, SLIC gate accesses do not use system bus or memory bandwidth. Second, since each SLIC knows the status of all gates, the spinning is done watching the local SLIC status register and not across the SLIC bus. Thus SLIC bus bandwidth is not adversely affected. This technique relieves one performance bottlenecks found in previous multiprocessor systems when under heavy load. Since the mutual exclusion is achieved via the gate mechanism, the systems bus and memory architecture is simpler. This allows the system bus to be built at a lower cost, without compromising performance, and with a higher degree of reliability.

The DYNIX® operating system, Sequent's version of UNIX 4.2bsd, is implemented using three types of mutual exclusion primitives built from gates. These are spin locks, counting semaphores and the direct use of the gates themselves.

The gate is the lowest level primitive and is directly implemented in the SLIC chip. Gate acquisition is the fastest among the three types of mutual exclusion primitives. However, since there are a finite number of gates, they are used only in the most time-critical regions. For example, a gate synchronizes the processors' accesses to the RunQueue. Gates, because of their spin-oriented nature, are held for only a short period of time to minimize contention. However, when there is contention, no system bus or SLIC bus cycles are consumed by a processor waiting for a busy gate.

Locks are defined to multiplex gates. Whereas there are only 64 gates, the number of locks is unlimited. A lock uses a shared memory location to encode the lock state (locked or unlocked) guarded by a gate. The gate is acquired only to manipulate the lock variable (i.e., to set the lock). Therefore, lock operations are also atomic. A single gate may be used to synchronize accesses to many locks. For example, changes to the state and flag variables for a process in the process table must be performed atomically. This is done by first acquiring a per-process lock. If the system is configured for 500 processes, there will be 500 such locks. However, only 10 gates might be assigned to synchronize the manipulation of those locks. Locks, like gates, are spin-oriented and are used to guard short critical regions. Since each processor is equipped with a bus-watching cache, the instructions and data accessed while waiting for a lock are cached. Therefore, no system bus cycles are consumed waiting for a busy lock.

In a uniprocessor, the SPL (set interrupt priority level) mechanism provides the synchronization necessary between interrupt-level and process-level access to shared data structures. Specifically, the process-level code must raise the interrupt priority level of the processor to block out interrupt routines before executing a critical region. In a multiprocessor architecture, a form of interprocessor synchronization (i.e., gates and locks) is necessary in addition to the SPL mechanism. Process-level code must raise the processor interrupt priority level when acquiring a lock to avoid the deadlock that would occur when an interrupt routine on the same processor attempts to acquire the same lock.

The highest-level mutual exclusion primitive used is the counting semaphore. The counting semaphore consists of a counter and a wait queue. A gate guarantees the atomicity of the semaphore manipulations. Semaphores are used to block waiting for an event, or when the critical region guarded by the semaphore is very long. Instead of all waiters being awakened on an event, only one process at a time is awakened and allowed access to the resource.

Semaphores completely replace the conventional UNIX sleep/wakeup mechanism. In the sleep/wakeup model, all waiters are awakened and they all compete for the same resource. These processes proceed to race attempting to acquire the resource. One process wins and gets the resource. The others will have to go back to sleep again. On a uniprocessor, this is not a problem because the first to get scheduled receives the resource. By the time the rest of the processes are scheduled the resource is probably available to them. However, in a multiprocessor the awakened processes might all be scheduled simultaneously. There would be unnecessary context-switching, and context-switches tend to invalidate the processors' caches. This would cause heavy system bus activity while the caches are being refreshed. In general, semaphores are more appropriate in a multiprocessor because they provide more structure and eliminate this type of unnecessary context-switching.

4.2. Interrupt Distribution

Perhaps the most important function of the SLIC is that of interrupt control and distribution. To eliminate the potential overloading of a single processor with the entire system's interrupt load, the SLIC supports dynamic interrupt distribution.

® DYNIX is a trademark of Sequent Computer Systems, Inc.

The SLIC subsystem handles all device interrupts in the system. Interrupts from peripherals on the MULTIBUS® and the Small Computer Systems Interface (SCSI) bus are mapped to SLIC interrupt messages via Multibus-Adapters and SCSI host Adapters, respectively. All such SLIC interrupts are mapped to SLIC bins 1-6. Each bin has a potential for 256 interrupt vectors (specified by unique SLIC messages). Although this provides good flexibility, it is undesirable to build an interrupt vector table for all possible vectors. SLIC interrupts are programmed interrupts, so each module (processor, controller, etc.) on the system bus is told at system initialization time which SLIC bin and message data to send for each interrupt. Interrupt vector tables can then be optimally sized at system initialization time.

Each SLIC on the SLIC bus responds to interrupts directed at its own SLIC number. In addition, each SLIC corresponding to a processor is programmed to respond to the same destination group number. All device interrupts are directed to this group number. When an interrupt is injected into the system, the SLICs in the processor group arbitrate among themselves to determine which accepts the interrupt. Once the interrupt is accepted, the bin on the accepting SLIC is masked until completion of the interrupt handler so that this SLIC will no longer arbitrate for other interrupts in that bin. The acceptance of an interrupt by one SLIC in a group does not inhibit another SLIC in the same group from accepting another interrupt from the same bin or another bin. This allows multiple device interrupts to be serviced simultaneously.

Directing interrupts to a group of processors has several advantages. First, interrupts will be dynamically distributed among the group of processors. Thus no single processor or statically defined group of processors is a bottleneck. Secondly, processors may be brought online or taken offline dynamically without the loss of any interrupts. Finally, this architecture allows improved interrupt latency over that of a uniprocessor.

A SLIC arbitrates for interrupts based on its local priority register. The lower the priority, the more likely the SLIC will win the arbitration. The kernel sets the local priority register to reflect the scheduling priority of the process currently running on the processor. Idle processors set the SLIC priority register to the lowest possible value. The idea is to have the processors running the least important processes handle most of the interrupt load. Thus interrupt load is automatically and dynamically offloaded from the processors doing the most important work.

4.3. Uniprocessor Device Driver Support

To support a uniprocessor-based device driver in a multiprocessor environment, it is necessary to emulate a uniprocessor environment for the driver. The key requirement is to have the uniprocessor's interrupt routine execute on the same processor as the driver's process-level code. If this requirement is met, the SPL mechanism for synchronization will be sufficient. The SLIC supports this requirement with directed interrupts to an individual processor rather than to the normal processor group. There is no restriction on other drivers in the system and their interrupts will continue to be directed to the processor group and therefore distributed.

DYNIX supports uniprocessor drivers by:

- Initializing the device interrupt so it is directed to a single processor M (At present the first processor booted.)
- Emulating the sleep()/wakeup() model. This is implemented via a hashed array of semaphores, where the hashing function translates the wait channel address to a semaphore.
- Binding process-level driver code to run on processor M. A process using the uniprocessor driver is context-switched to processor M and is bound to processor M while executing the uniprocessor driver code.
- Binding timeout routines to processor M, if a uniprocessor driver is configured. This ensures that a uniprocessor driver's timeout routines are properly synchronized with the rest of the driver.

4.4. Inter-processor Signaling

Inter-processor signaling is another multiprocessor issue solved via the SLIC. A processor may need to signal another processor when there is some task for the other processor to perform. An example is a preemptive rescheduling nudge. This signaling is implemented via programmed interrupts through the SLIC. With the SLIC, one processor may send another a normal maskable interrupt (Bin 1-7), a non-maskable interrupt, or a software interrupt (Bin 0).

Software interrupts are most commonly used for interprocessor signaling during normal operations. The main reason is that the sender does not have to wait for the destination to acknowledge the interrupt, as Bin 0 interrupts are always accepted by the destination SLIC. The software ties each of the 8 bits in the software interrupt message data to a particular software interrupt handler, the most common being the rescheduling nudge. In addition, non-maskable interrupts are sent to all processor SLICs when the system panics and needs to shut down rapidly.

4.5. Process Scheduling

The process scheduling technique in a symmetric multiprocessor is conceptually similar to that of a uniprocessor. Whereas the uniprocessor scheduling policy is to always execute the highest priority runnable process, the multiprocessor scheduling policy is to always execute the N highest-priority runnable processes, where N is the number of processors. The multiprocessor dispatching model must:

- Provide for dynamic load balancing.
- Dynamically adapt to N processors, for N >= 1.
- Avoid unnecessary process migration and context-switching, as each dispatch causes heavy bus activity until the processor memory cache is refreshed.
- Allow for the dynamic starting and stopping of processors.

® MULTIBUS is a trademark of Intel Corporation.

The DYNIX scheduler uses a single priority-ordered queue of runnable processes (**RunQueue**). There is no processor-specific distinction made for processes in the RunQueue and there is no static binding of processes to processors. Since all processors are identical, any process (whether in User or Supervisor mode) may run on any processor. The symmetric, shared-memory architecture allows for easy implementation of dynamic load balancing. Since all processors are identical and all process state information, code, and data reside in a common shared memory, process migration is trivial.

The basic dispatching algorithm is simple. Each processor, on entering the dispatcher, merely removes the highest priority process from the RunQueue and executes it. Access to the RunQueue is synchronized via a single SLIC gate. If there is no work to do, the processor executes its idle loop. The idle loop spins testing the RunQueue for runnable processes. Note that this testing does not require any SLIC gate synchronization, since it is sufficient to merely detect change in the RunQueue status. Once change is detected, the idle loop returns to the dispatching loop. The idle loop takes advantage of the bus-watching cache, and does not consume any system bus cycles or SLIC bus cycles. A simplified model of the basic dispatcher is illustrated in Figure 5.

```
swtch() {
    acquire RunQueue gate;
    while( RunQueue empty ) {
        release RunQueue gate;
        idle();
        acquire RunQueue gate;
    }
    remove highest-priority process;
    release RunQueue gate;
    set SLICPRI to that of the process;
    resume( process );
}

idle() {
    set SLICPRI to IDLE;
    while( RunQueue empty )
        continue;
}
```

Figure 5: Basic Dispatcher

Besides rescheduling themselves voluntarily, processes may be preempted either by a time-slice or when a higher-priority process becomes runnable. A time-slicing interrupt is injected into the system periodically to cause the accepting processor to determine whether there is a running process which should be preempted by a higher- or equal-priority process in the RunQueue. If so, the processor to be preempted is nudged to reschedule via a directed software interrupt. A process often becomes runnable as a result of a device interrupt. The processor accepting the device interrupt determines whether the awakened process should preempt a running process. If so, it will nudge the processor running the lowest priority process, telling it to reschedule.

The priority of the process running on a given processor is stored into the SLIC's priority register. This is used by the SLIC for contention resolution when arbitrating to receive an interrupt request. This supports a heuristic that the lowest-priority processor is most likely to handle an interrupt that causes it to be rescheduled. Setting the per-processor redispatch flag (**runrun**) may be the only action necessary to facilitate a reschedule if the interrupt makes a higher priority process runnable. This lowers the overhead in that the more expensive directed software interrupt is avoided and no additional SLIC bus cycles are needed to signal the context-switch.

4.6. System Configuration Control

In addition to its role as an interrupt controller, the SLIC subsystem provides a convenient, simple and reliable communication path among modules (processors, controllers, bus adapters, etc). This communication path determines system configuration, to configure and deconfigure modules, to bring processors online or take them offline, and to exchange error management information.

The support for these diverse functions is obtained via SLIC slave registers. Each SLIC can either read or write any other SLIC's slave registers via SLIC messages. In addition to reporting status, SLIC slave registers also serve as command registers for their respective modules. The functions attached to these registers are module-dependent. For example, slave registers on the memory controller are used to report ECC error information, to identify the memory configuration (64K or 256K chip technology), and to set various configuration attributes of the controller (e.g., base address, interleave factor, etc). Also, processors are brought online and taken offline via remote SLIC messages directed at the desired processors' SLICs.

To help system configuration, there is a subset of SLIC slave registers common to all modules. These include configuration information such as module type (e.g., processor, memory, etc) and revision level. With the SLIC slave register mechanism available to communicate configuration information and to set configuration attributes, there is no need for switches or wire-wrap stakes on the modules on the system bus.

The system's power-up firmware takes advantage of the SLIC bus to probe for the presence of hardware modules. It then builds a complete configuration table in a known place which can be used by diagnostics, the DYNIX operating system, or any other stand-alone program. When another processor board is plugged into the backplane, the power-up firmware will automatically add this resource to the configuration table. No changes to the software are necessary, and a single DYNIX binary is able to control a wide variety of hardware configurations. This simplifies the task of field upgrades and maintenance.

5. Evaluation

The Sequent Balance is an implementation of a shared-memory symmetric multiprocessor, supporting from two to thirty NSC Series 32000 microprocessors as compute engines. Each processor is packaged with a main-memory cache and a SLIC, allowing the operating

system to take advantage of these features. The remainder of this section discusses some of the particular benefits derived and a few problems that result.

5.1. SLIC-Supported Symmetry

A major benefit of the per-processor SLIC and cache, and use of SLIC in all major hardware components of the system, is the resulting symmetry of the hardware as viewed from the software. The primary advantage is that all processors are *identical* in hardware and are so treated by the operating system. This is manifested in a number of ways:

- Any processor can service any interrupt at any time. This automatically distributes the interrupt load, using idle processors when they exist. This also allows processors to be dynamically removed from service, or taken offline, without impact on system operation.
- Any processor can control any piece of hardware, through the SLIC slave registers. In particular, any processor can take another offline or place it back online,
- Any processor can signal another using SLIC interrupt example, there is no central dispatching context switch in another. SLIC messages are often used to start I/O operations, avoiding any special affinity between processors and I/O-controllers.
- Any processor can be the first booted, eliminating the need for special treatment during system initialization.

The result is that no processor is special for any purpose. This allows any process or operating system code to execute on any processor at any time, greatly enhancing the load-balancing aspects of the system. The operating system treats the processors as just another resource to be managed, by arranging that processors dispatch themselves in response to process state changes.

5.2. Bus-Watching Cache

The operating system takes significant advantage of the bus-watching support in the cache. This simplifies low-level mutual exclusion and dispatching primitives without affecting system performance.

5.3. Few Hard-Coded Limits

There are few hard-coded limits on the amount of hardware resource the system will support. There is no notion of the maximum number of processors that may be present in the system, although with enough processors some the low-level dispatching algorithms may be non-optimal. Similarly, the system can support an almost arbitrary size of physical memory and number of I/O bus adapters, subject to hardware limits. Independence of these parameters allows a single operating system binary to run on a large number of different hardware configurations. In particular, the system may be shut down and one or more processor boards (or memory boards, I/O controllers, etc) added or removed; when the same kernel binary is rebooted the operating system automatically uses the new resources or adapts to the lack of a resource previously available.

5.4. Some Performance Data

Since the processors in a tightly-coupled multiprocessor must contend for the bus and memory, it is reasonable to expect that a system containing N processors will provide less than N times the performance of a single processor. In fact, a rule of thumb commonly applied to tightly-coupled multiprocessors predicts that each additional processor will provide at most 80% of the computing power of the previously added processor. Given this rule of thumb, system performance improvement would top out below 5 effective processors no matter how many are added.

The Sequent Balance system design refutes this rule of thumb. The combination of per-processor cache, SLIC, bus, and memory controller design provides a nearly linear increase in performance as the number of processors is increased from two to thirty, depending on the application. Compute-bound applications benefit most from the addition of processors, and are the ones that see the nearly linear improvement. However, the most impressive benefit of the system is in a multi-user environment. It surpasses the uniprocessor in system in response time, throughput and availability. Sequent has run many benchmark programs in single-stream and multi-stream mode to measure the effective throughput of the system as the number of processes and processors are increased. Some of the programs intentionally try to break the cache, to measure a worst case situation. Figures 6 and 7, briefly presents the results; more detailed information is available from Sequent.

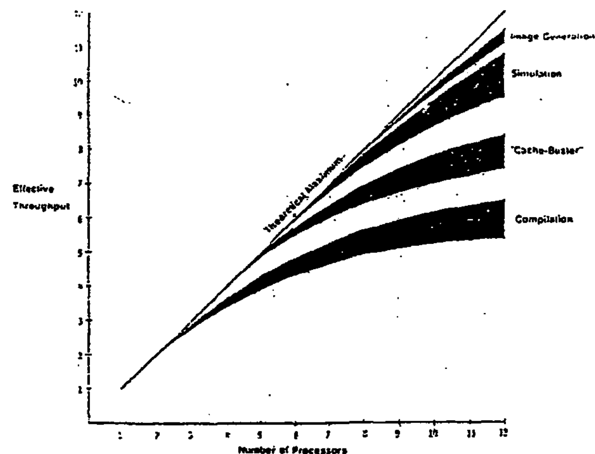

Figure 6: Effective Throughput vs Number of Processors

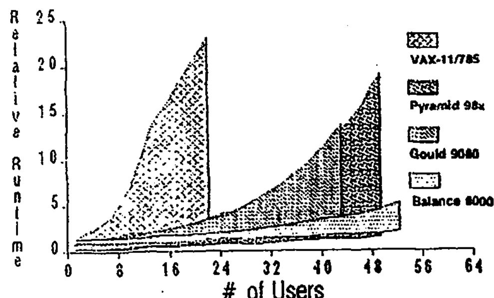

Figure 7: BALANCE MULTI-USER WORKLOAD PERFORMANCE

5.5. Gates vs Memory Test-and-Set

The decision to use SLIC gates instead of test-and-set memory support was based on a desire to keep the system bus, memory controllers, and processor cache implementation simpler and lower cost, while achieving good performance. Separating the SLIC bus from the main system bus allows both to be simpler and more tuned to their respective purpose. Since 64 gates are not enough for all low-level data structure mutual exclusion, the *lock* abstraction of gates was created. The collision rate on the gates is distributed among locks and semaphores is low, typically less than 1% of all gate transactions. There are a few gates that exhibit higher collision rate; in particular, the RunQueue gate and timer-structures gate. But even in a system that supported memory-based test-and-set operations, these data structures would each be controlled by a single test-and-set variable, so using gates creates no additional problem.

A problem with the limited number of gates is managing them as a resource. Currently they are statically allocated, defined mnemonicly in a header file, so the entire system must be recompiled to change the assignment of gates to functions. There also must be a few gates reserved for use by orthogonal parts of the operating system (such as device drivers). Fortunately, the initial distribution of gates seems to work well. This area needs further study.

5.6. User Access To SLIC

SLIC is so central to correct operation that incorrect use can easily crash or deadlock the system. As a result, direct access to the SLIC by user processes is strictly forbidden. The motivation is similar to disallowing direct user access to important kernel data structures. Applications that need access to the SLIC (e.g., to send interrupt messages or access slave registers) typically create a device driver or other code that lives within the operating system for this purpose (all kernel code may access SLIC).

One result of this is that user processes cannot access gates other than by calling the operating system. Applications that want to take explicit advantage of multiple processors running in parallel in a shared memory usually require fast mutual exclusion support, much like the operating system. Calling the operating system for this purpose takes one or two orders of magnitude more time than can be tolerated by many such applications. There are various software solutions to this problem[1,3], but these are awkward to use and don't provide optimum performance. To address this need the Multibus Adapter supports a set of test-and-set variables, called **atomic lock memory**. The Multibus adapter can be mapped into user process address space, to provide convenient and fast access to these variables. Since the bus treats I/O accesses separately from memory accesses, manipulation of atomic lock memory variables does not impact system bus bandwidth.

6. Conclusion

Microprocessors have reached a point of maturity where multiprocessor systems can be built using microprocessors as the computing engines, leveraging VLSI technology to reduce cost, package size and complexity, and increase reliability. It is appropriate that additional VLSI leverage can successfully be applied to address other issues:

Cache Memory. Symmetric, tightly-coupled multiprocessor systems place a high demand on system bus and memory bandwidth. Properly coordinated per-processor cache memories significantly reduce bus traffic and memory contention, avoiding traditional bottlenecks.

Interrupt Distribution. An intelligent interrupt controller can support dynamic distribution of interrupt load among processors. The advantages include:

- Automatic interrupt distribution on a process priority basis.
- Enhanced system availability by not binding interrupt sources to particular servers.
- Improved interrupt latency.
- Transparent assist to dynamic load balancing.
- Integrated inter-processor signaling mechanism.
- Software-assignable interrupt vectors, enhancing configurability.
- Simple accommodation of uniprocessor drivers.

Inter-processor Synchronization. An efficient low-level mutual exclusion primitive can be provided inexpensively, and can support the implementation of classic higher-level synchronization primitives.

System Configuration and Control. Use of a common mechanism to identify and control basic hardware components in the system allows one operating system binary to run on a wide variety of hardware configurations. This reduces the complexity of hardware and software configuration, and simplifies maintenance tasks. System resources can be brought in and out of service dynamically, and error information can be gathered, all using the same mechanism.

Shared-memory symmetric multiprocessor systems have long been of interest because of their potential for high performance and flexibility. In the past, barriers such as entry cost, achievable range of performance, and ease of use have limited their commercial success. Multiprocessors were once thought to be limited to a performance level of 3 to 4 times that of a uniprocessor. However, the

system we have described realizes an almost linear improvement in performance as processors are added. By exploiting VLSI technology, low-cost, high-performance multiprocessors can be built which fully support popular operating system such as UNIX.

References

[1] Dijkstra, E. W., "Solution of a Problem in Concurrent Programming Control," *Communications of the ACM*, vol. 8, 9 (September 1965), .

[2] Goodman, J. R., "Using Cache Memory to Reduce Processor-Memory Traffic," *Proceedings of the 10th International Symposium on Computer Architecture*, June 1983.

[3] Lamport, L., "A New Solution of Dijkstra's Concurrent Programming Problem," *Communications of the ACM*, vol. 17, 8 (November 1974), .

[4] Lovett, T., "A CAE Case History," *VLSI Design*, November 1984.

[5] Thakkar, S. S., Gifford, P. and Fielland, G., "Balance: A Shared Memory Multiprocessor," *Proceedings, 2nd Int. Conf. On Supercomputing, Santa Clara*, May 1987.