

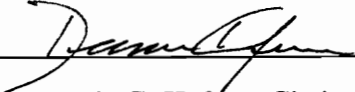
The Porting of the MCC Extensible Software Platform to the Sequent Symmetry

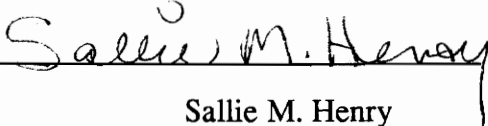
by

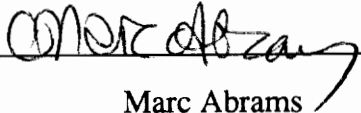
Joel E. Patterson

Project submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements of the degree of
Master of Science
in
Computer Science

APPROVED


Dennis G. Kafura, Chairperson


Sallie M. Henry


Marc Abrams

May 1993
Blacksburg, Virginia

C.2

LD
5655
V851
1993
P377
C.2

The Porting of the MCC Extensible Software Platform to the Sequent Symmetry

by

Joel E. Patterson

Dennis G. Kafura, Chairperson

Computer Science

(ABSTRACT)

The Microelectronic and Computer Technology Corporation (MCC) Extensible Software Platform is an object-oriented system which allows for distributed processing over heterogeneous platforms with a similar processor architecture. A port was proposed for the Sequent Symmetry which would:

- [1] Be the first port of ESP to an Intel 80x86 based system
- [2] Be the first port of ESP to a UNIX-based multi-processor system
- [3] Develop and use a shared-memory mailer between local nodes

In performing the port, the GNU C++ compiler with MCC additions was required to run on the Sequent. The ESP system was then subsequently ported and the shared-memory system implemented. This report describes the process and results thereof.

Acknowledgements

I would like to express my thanks to my advisor, Dr. Dennis G. Kafura, for having conceived the project and aiding me until its completion. I am also obligated to Kim Stuart Smith of MCC for his helpful advice and aid in solving problems. Finally, I would like to thank MCC for having provided financial support throughout the project.

Table of Contents

Introduction	1
1.1 Parallel Programming	1
1.2 Purpose of the Extensible Software Platform	2
1.3 Reasoning Behind the Sequent Port	5
1.3.1 Project Goals	5
1.4 Sequent Architecture	6
ESP Overview	8
2.1 ESP Applications	8
2.2 ESP Classes vs C++ Classes	9
2.3 ESP Methods	10
2.3.1 Method Invocation	10
2.4 Futures	11
2.5 Related Work	14
ESP Software	17
3.1 ESP Software Overview	17
3.2 ESP Processes	17
3.2.1 The Shadow Process	18
3.2.2 The ISSD Process	19
3.2.3 The Mail Daemon Process	19
3.2.4 The Node Kernel	20

3.3	Public Service Objects	20
3.3.1	Application Manager	21
3.3.2	issd_File	22
3.3.3	Local_File	22
3.3.4	Pod/Probe	22
3.3.5	Class Librarian	22
3.3.6	Env	23
The ESP Compiler		24
4.1	Why the G++ Compiler	24
4.2	What Source Was Required	24
4.3	Differences between standard and MCC G++	25
4.3.1	General Differences	25
4.3.2	->() Operator	25
4.3.3	New Operator	25
4.4	Virtual Functions	27
4.5	Making the G++ Compiler	28
4.5.1	Obtaining the Compiler	28
4.5.2	Making the GNU Assembler	29
4.5.3	Making the GNU C Compiler	29
4.5.4	Making the GNU C++ Compiler	30
4.5.5	Making the G++ Library	30
4.6	Problems	31
4.6.1	Include files	31
4.6.1.1	Include File Parameter Lists	31
4.6.1.2	Include File Changes.....	32
4.6.2	Compiler bugs	33

Porting ESP	34
5.1 Overview	34
5.2 ISSD Modifications	35
5.2.1 client_base.h	35
5.2.2 issd.cc	35
5.2.3 issd_File.cc	35
5.2.4 issd_File.h	36
5.3 Public Service Object Modificatio	36
5.4 Mail Daemon Modifications	37
5.4.1 mail_daemon.sequent	37
5.4.2 mail_daemon.cc	37
5.5 Kernel Modifications	37
5.5.1 platform.h	38
5.5.1.1 Semantic changes	38
5.5.1.2 Context switching	38
5.5.1.3 ff1() call	39
5.5.1.4 Miscellaneous	39
5.5.2 mem_mgt.cc	39
5.5.2.1 Shared Memory Message Allocator	40
5.5.3 sys_configuration.cc	40
5.5.3.1 SEQUENTHOST->SUNHOST	40
5.5.3.2 Node_Type()	41
5.5.3.3 sys_configuration()	41
5.5.4 sys_configuration.h	41
5.5.5 symbol_table.cc	41
5.5.6 def.h	41
5.5.6.1 Kernel_Family	42
5.5.6.2 node_types	42

5.5.7	msg_stuff.cc	43
5.5.8	msg_stuff.h	43
5.5.9	remote_base.cc	43
5.5.10	Dynamic Linking	43
5.5.11	ftio_lib.cc	44
5.6	Multiple Change areas	47
5.6.1	utils.h	47
5.6.2	Message Output	48
5.6.3	vprintf() family	48
5.6.4	Byteorder problems	48
	Shared Memory Implementation	50
6.1	Background	50
6.2	Implementation	51
6.2.1	Initialization	51
6.2.2	Shared Memory Manager	51
6.2.2.1	Obtaining Shared Memory	51
6.2.2.2	Initializing Shared memory	53
6.3	Management of the Shared Memory	56
6.3.1	Accessing Shared Memory	56
6.3.1.1	Low-Level locking routines	58
6.3.2	Shared Memory Manager	60
	Shared Memory Mailer	62
7.1	sock_mail.cc	62
7.1.1	mail_system::mail_system()	63
7.1.2	mail_system::wait()	64
7.1.3	mail_system::fill()	65

7.1.4	mail_system::mail()	68
7.2	sock_mail.h	70
7.3	fast_path.cc	70
7.3.1	fast_path::fast_path()	71
7.3.2	fast_path::send()	71
7.3.3	fast_path::receive()	72
7.3.4	fast_path::msgs_avail()	73
7.4	msg_queue.cc	73
7.4.1	msg_queue::msg_queue()	74
7.4.2	msg_queue::insert()	75
7.4.3	msg_queue::remove()	76
7.4.4	msg_queue::msgs_in_queue()	77
Conclusion		78
8.1	Test Programs	78
8.2	Shared Memory Tests	79
8.3	Future Work	81
Appendix		82
References		85
VITA		87

List of Illustrations

Figure	1-1.	ESP Hardware Environment [13, 1-4]
Figure	2-1.	Futures [1, 5]
Figure	2-2.	Example of a class declaration
Figure	3-1.	ESP Processes [6, 40]
Figure	4-1.	Virtual Function Table [13, 5-8]
Figure	5-1.	G++ assembly example
Figure	5-2.	ESP Kernel definitions
Figure	5-3.	Big/Little Endian [14, 178]
Figure	5-4.	FtoA code
Figure	5-5.	DtoA code
Figure	6-1.	<code>shared_mem_manager::shared_mem_manager()</code> code
Figure	6-2.	<code>shared_mem_manager::get_mem()</code> code
Figure	6-3.	<code>spin_lock::acquire()</code> code
Figure	6-4.	<code>m_lock()/m_unlock()</code> code
Figure	6-5.	<code>spin_lock::release()</code> code
Figure	6-6.	<code>sh_mem_all::sh_mem_all()</code> code
Figure	7-1.	<code>mail_system::mail_system()</code> code
Figure	7-2.	<code>mail_system::wait()</code> code
Figure	7-3.	<code>mail_system::fill()</code> code
Figure	7-4.	<code>mail_system::mail()</code> code

Figure	7-5.	<code>fast_path</code> header file
Figure	7-6.	<code>fast_path::fast_path()</code> code
Figure	7-7.	<code>fast_path::send()</code> code
Figure	7-8.	<code>fast_path::receive()</code> code
Figure	7-9.	<code>fast_path::msgs_avail()</code> code
Figure	7-10.	<code>msg_queue</code> header file
Figure	7-11.	<code>msg_queue::msg_queue()</code> code
Figure	7-12.	<code>msg_queue::insert()</code> code
Figure	7-13.	<code>msg_queue::remove()</code> code
Figure	7-14.	<code>msg_queue::msgs_in_queue()</code> code

List of Tables

Table 8-1. Loop_Test test results

Table 8-2. Tree_Top test results

Chapter 1

Introduction

1.1 Parallel Programming

Parallel programming is the subdivision of a program into individual components for execution on separate processors. The goal of parallel processing is to increase the execution speed of a program above that of the program executing on a single processor.

There are two primary methods of implementing parallel processing. The first uses a parallel computer, a computer with multiple processors. A program executing on a parallel computer may have its tasks parcelled between multiple processors. These processors, working in parallel, can usually perform a task faster than a single processor performing the same task.

The second method of implementing parallel processing employs a distributed network. A distributed network of processors is similar to a parallel computer in that there are multiple processors available, but differs in that they do not share a common memory. In fact, in a distributed system, the processors may not even be compatible. Each processor, or group of processors, since parallel machines may be used in a distributed network, is connected to the others through a communications network. The machines, or nodes, in the network communicate with each other through this network.

The disadvantage of a distributed network to a parallel computer is that the network is almost always several orders slower than the communications paths between processors

in a parallel machine. On the other hand, the one of the advantages of a distributed network over a parallel computer is that a network of single processor machines communicating over a network is usually cheaper. Another advantage is that in a distributed network, machines may be added which are capable of performing a specific task while in a parallel environment, one type of processor exists which must perform all tasks.

At the Microelectronics and Computer Technology Corporation (MCC), an effort has been underway to create a distributed system which would allow processes to be divided amongst the nodes in the distributed network. Furthermore, the system would allow for the usage of heterogeneous nodes in the network, allowing the process to take advantage of nodes which specialize in a particular form of computation (i.e. high-speed numeric calculation). This system is called the Extensible Software Platform (ESP). The project outlined in this paper describes the port of ESP to the Sequent Symmetry platform.

Section 1.2 gives a brief overview of the Extensible Software Platform. Section 1.3 identifies the goals of the project, while Section 1.4 describes the hardware system used in the implementation.

Chapter 2 presents a short description of the ESP environment and Chapter 3 describes the various programs/processes used to implement that environment.

Following the overview of ESP, the details of the actual port are reviewed. Chapter 4 reviews the work done to install the compiler required for the compilation of the ESP. Chapter 5 details the actual port of the ESP code.

Additionally, Chapters 6 and 7 describe the enhancements made to the ESP system, namely the addition of a shared memory mailer. The final chapter draws conclusions from the port and discusses possible future work.

1.2 Purpose of the Extensible Software Platform

The Extensible Software Platform was developed between 1987 and 1990 by a group of research scientists and graduate students at the Microelectronics and Computer Technology Corporation; the project was partially funded by the Defense Advance Research Projects Agency (DARPA). [11, 1]

ESP is a group of software modules which may be assembled into experimental parallel-computing systems. The advantage of this approach is that it allows researchers to tailor the system to their own particular area of research; each module may be modified to take maximum advantage of the underlying hardware, software or system environment. A consequence of this decision is that each module must contain a minimal number of modular dependencies as well as being easily reconfigurable. [13, 1-3]

Figure 1-1 shows the model of an ESP hardware environment. [13, 1-4] As the figure shows, heterogeneous hardware configurations are allowed in the ESP environment. The only restriction on adding new hardware configurations is that all machines in the network support the same underlying communications protocol (in ESP's case, TCP). Each node in the network is capable of performing any function within the application so long as it possesses the hardware to do so. For example, the ESP Application Accelerator shown in Figure 1-1 lacks I/O capability and thus may not perform I/O tasks without the appropriate interface system. [13, 1-3]

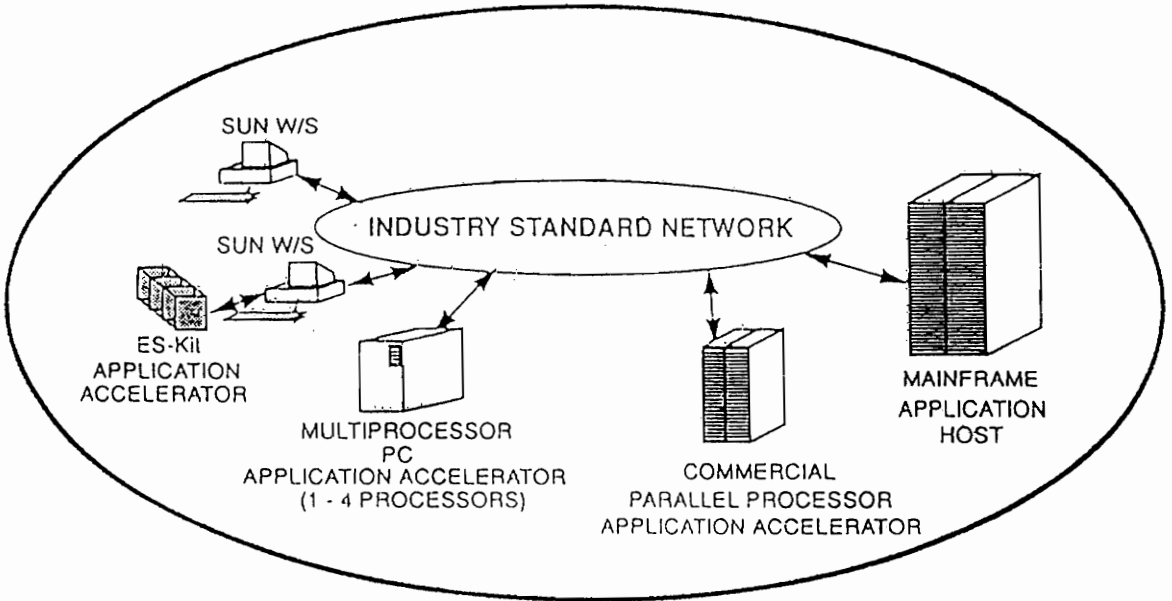


Figure 1-1. ESP Hardware Environment

The software architecture imposes a two-level structure on the system structure. A single system is designated as the "front-end" system where the user may enter commands to ESP. The application and support processes may be executed on remote systems.

The "front-end" system at this time is required to be a UNIX system which supports Internet sockets using TCP. Multiple "front-end" systems may exist in the ESP environment, but only one may exist per application. [13, 1-3]

The application and support processes are not required to execute on remote systems. However, since the purpose of ESP is distributed parallel processing, this is usually the case. It should also be noted that the application itself may execute on multiple, possibly heterogeneous hardware platforms.

1.3 Reasons Behind the Sequent Port

The porting of ESP to the Sequent Symmetry platform was attempted for several reasons. First, although ESP was designed for distributed, parallel processing, it was generally used over a distributed network of single-processor platforms. The Sequent Symmetry ESP port was the first attempt to port ESP to a widely-available multi-processor platform and the first port of ESP to a multi-processor platform with shared memory. A second reason for the port was to add a Sequent platform to the list of computers which run ESP applications. Third, this was the first port to use the Intel 386 architecture. Finally, one of the foremost ESP platforms is the Sun workstation. Both the Sun and Sequent operating systems are based upon BSD (Berkeley Standard Derivation) UNIX 4.3. Therefore, it was felt that aside from assembly language incompatibilities, the port would not have an unacceptable number of operating system differences.

1.3.1 Project Goals

The goals of the ESP Sequent port were as follows:

- [1] Modify for the Sequent the makefiles which compile, link and install the ESP software.
- [2] Develop assembly language interface code for the Sequent Symmetry conforming to the interface of the existing "platform" class.
- [3] Modify the code to perform dynamic loading of object files created and stored on the Sequent system which would conform to the interface of the existing "dynam_ld" class.
- [4] Modify the "post_off" class, the "mail daemon", and any related classes or subsystems to effect the desired efficiency of message-passing between objects co-located on the Sequent and the transparent message-passing to all objects.
- [5] Modify the ISSD subsystem to replace the use of the Sun-specific "on" command by the use of the more generic remote shell facility. [Note: this task was performed by MCC in a release shortly after the start of the project]
- [6] Modify or develop code as necessary to implement the required capabilities of the ported software.
- [7] Document all delivered code within the text files of the code itself and a technical report describing the overall implementation.

1.4 Sequent Architecture

The Sequent Symmetry currently in use at Virginia Tech is a ten-processor machine based upon the Intel 80386 processor. Each processor has an 80387 math co-processor. The Sequent uses DYNIX as its operating system; DYNIX is a version of Berkeley 4.3 BSD adapted for parallel processing.

32 megabytes of memory are available for use by the processors. Processes are assigned to processors by the operating system and may migrate to other processors during their

execution. Since memory is global to all processors, up to the full amount of memory may be accessed as shared memory.

Chapter 2

ESP Overview

2.1 ESP Applications

ESP can be thought of as a network of (possibly heterogeneous) nodes. Each of these nodes contains its own processor(s), local memory and uses a common message protocol. [13, 3-3] Since this is a highly general definition, an equally general definition of an ESP application is used. Thus, ESP applications are defined as a set of objects which are working upon a common problem and can:

- [1] "Include the characteristics of both data structures and procedures"
- [2] "Cause the creation and destruction of other objects, either on the same (local) node or on another (remote) node"
- [3] "Invoke the methods of local and remote objects and receive results from them"
- [4] "Define private data structures and procedures that cannot be directly acted upon by other objects"
- [5] "Inherit and extend the characteristics defined by other objects" [13, 3-3]

Excepting the operations on remote objects, these requirements duplicate the characteristics of class objects in C++. For this reason, C++ was chosen as the language used for ESP. The majority of ESP and all ESP applications are written in C++. For the purposes of this paper, it is assumed that the reader has a basic understanding of the C++ language.

2.2 ESP Classes vs C++ Classes

The concepts of C++ classes, objects and methods are central to the ESP environment. A class is a type definition. An object is an instantiation of a class. The object contains the actual data which was defined in the class; the object also contains pointers to the methods used by the class.

An ESP class is read into memory under three circumstances. "A class may be included as part of the kernel, which would be loaded when the kernel is loaded. A class may be included as part of the application, which would be loaded when the application is loaded. Finally, a class may be loaded when one of the objects of the class is instantiated, or dynamically loaded. No data is associated with a class when it is loaded." [13, 3-4] A specific class is loaded once per node for an application. Note that it is through this loading strategy that an ESP application is able to execute on heterogeneous platforms; the appropriate binary code for the architecture is loaded. If there are multiple similar applications running on a node, each application contains its own copy of the class. [13, 3-5]

Objects can be created in ESP the same as in a normal C++ program. They can be statically declared as in Figure 2-1, which instantiates the object in stack memory or the object can be created dynamically. In the case of dynamic allocation, the object is created in heap memory.

```
class X {  
    ...  
    myclass inst1;  
    ...  
}
```

Figure 2-1. Example of a class declaration

2.3 ESP Methods

A method is an operation (function) which is performed on the objects belonging to a class. However, a method may only be applied to objects of the class in which it is defined.

A method executing on an ESP node can perform the following tasks:

- [1] Invoke a method on a local or remote node
- [2] Reply to an invocation from a local or remote node
- [3] Instantiate an object on a local or remote node
- [4] Wait on a method executing on another node or continue processing without waiting through the use of futures. [13, 1-7]

One of the advantages of C++ is that it allows overloading of methods; multiple methods may have the same method name. The methods are distinguished by the requirement that the combinations of parameter data types must be unique for each method within that group of methods.

2.3.1 Method Invocation

In standard C++, the `->` operator may either select a data element in an object or invoke a method of the object. For example, applying method `f(g)` to instance `x`, can be specified by either `x.f(g)` or `x_ptr->f(g)`, where `x_ptr` is a pointer to `x`. This manner of method selection works in a shared memory environment, where selecting a method is a process of generating an offset address to the method, and then branching to the method. However, this method selection will not work in a distributed processing environment.

Standard method references can not be employed in a distributed environment because the referenced method may exist on another system, in which case no offset address could be generated. If the referenced instance is at a remote node, then the local kernel

must inform the remote node to invoke the method via messages. It may be necessary for the local method to wait until a result is returned from the remote method. When the remote kernel receives the request, the remote kernel invokes the method for the instance and might return a response from the invoked method. Since invoking the remote method may, in turn, cause other remote methods to be invoked, the kernel may block while these other remote methods execute. [13, 3-13]

For these reasons, ESP C++ has redefined how methods are invoked.

The `->()` method operator was created by MCC for remote call invocation. This operator signals to the kernel that the method invoked is a remote method. The syntax of the `->()` operator remains the same whether the method being invoked is local or remote. [13, 3-13]

A local method making a remote method call must block until the remote method completes. The future class is provided in order to allow a program to invoke remote methods without blocking.

2.4 Futures

In order to allow a program to invoke a remote method without having to block, ESP provides the concept of futures. Futures also provide the means through which ESP provides concurrency.

In ESP C++, a future consists of an identifier associated with a request. The general format of a future invocation is:

```
future <identifier> = <class-instance>->function();
```

The `<identifier>` is associated with the future. This identifier represents both the future and the value returned to the future depending on the context. The function invocation, `<class-instance>->function();` is a standard remote method invocation, where `<class-instance>` is an object of the defined class. The message to invoke the remote method is

sent by the kernel when the invocation is executed. However, the future is not evaluated until it is assigned to another type.

Consider the following program sequence:

```
future f = my_instance->func(x);  
...           // perform some processing  
int j = f;
```

In this example, method `func(x)` of instance `my_instance` is executed. The identifier `'f'` represents a future reference. The local kernel generates a request message and sends it to the node containing the requested instance/method, and a future is returned in `'f'`. The local method then performs whatever processing is between the future invocation and the `'int j = f;'` assignment statement.

When the statement `'int j = f;'` is encountered, the future is evaluated as follows. If the remote method has returned, the returned value is assigned to `'j'`. If the remote value has not returned, the local method blocks until the reply is received. [13, 3-16] An example of a future's path is shown in Figure 2-2.

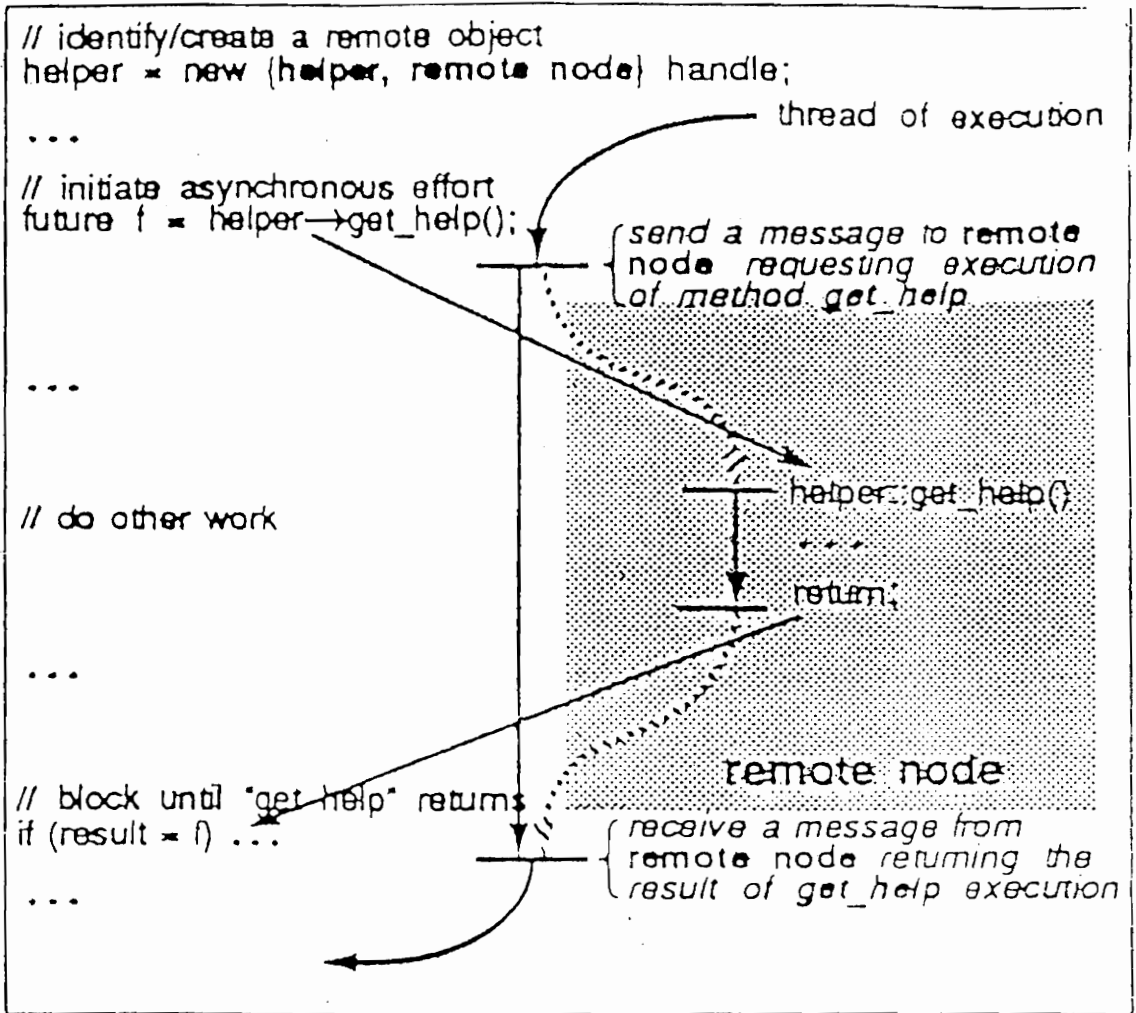


Figure 2-2. Futures

"In the ESP environment, a future is treated the same as any object: it is declared, has associated methods, etc. A future can be passed to other objects, and the receiving object can perform any operations with the future, such as adding to it, evaluating it, etc. There are no syntactic or operational restrictions unique to futures." [13, 3-15]

2.5 Related Work

ESP is not the only system created to run parallel processes on a parallel machine. Indeed, the Sequent DYNIX system itself contains a software library for writing parallel programs. [15, 10] However, the Sequent method only allows the "threadedness" of a parallel program to be equal to the number of physical processors on the system. [15, 10] An alternative method to this approach -- and to ESP, is the PRESTO parallel-programming system, which is described and contrasted to ESP in this section.

PRESTO, like ESP, runs on Sequent hardware, on top of the DYNIX operating system. [15, 10] The goals of the PRESTO system were three-fold: to create a programming environment conducive to expressing concurrent algorithms, to do so efficiently, and to implement it in such a manner that extensions and modifications are easily possible. [15, 1]

The PRESTO system is composed of five main objects: the scheduler, the processor, the thread, the spinlock and the synchronization object. [16, 3] From these objects, a parallel system evolves which includes:

- [1] a preemptive scheduler
- [2] the ability to statically and dynamically create new threads of control
- [3] busy waiting synchronization based on hardware atomic locks
- [4] context switching ability between threads [16, 3]

The ESP and PRESTO systems appear to have much in common; however, when a detailed examination is undertaken, several significant differences arise.

Like ESP, the PRESTO system was also written in C++. The reasoning behind the use of C++ language was similar to that for ESP: the wish for an object-oriented programming language and for portability reasons. [15, 2] However, the C++ compiler used for compiling PRESTO was an unmodified version. This fact has several advantages, primarily being that the PRESTO system is not dependent upon the compiler used, or at the mercy of significant changes to the implementation of the compiler.

Another significant difference between PRESTO and ESP is that in the PRESTO environment, all objects execute in a single address space shared by all the processors. [15, 4] PRESTO's object model (not discussed here) allows for the objects to co-exist without overwriting each other. [15, 4] This method of implementation not only allows for extremely fast communications between objects (similar to the shared-memory mail system implemented under ESP), but also for synchronization of objects via shared memory. The PRESTO memory implementation allows for another advantage over ESP via the matter of object scheduling. In ESP, a kernel is created for every processor in the system. The same is true for PRESTO. However, once an ESP thread is assigned to an ESP kernel, it remains with that kernel for the duration of its life. In PRESTO, since all threads share the same memory space, a thread may be moved from kernel to kernel, depending on the load. [15, 10] Thus, PRESTO is more effective at load balancing than ESP. However, there is nothing in the ESP design which would prevent it from undertaking the same method of memory management and scheduling.

As a counterpoint to this argument, it should be noted that the PRESTO design seems tied to the amount of shared memory on the system and may not work for a system which contains multiple processors, but no shared memory capability.

Although the PRESTO system appears to have several definite advantages over ESP, ESP also has its advantages over PRESTO. The ESP system was designed to work over a heterogeneous network, while PRESTO was designed to work only upon the machine on which it is running. Furthermore, the ESP language appears more robust with regard to the control which the programmer may take over the implementation of the algorithm

to run. This is partially due to MCC having redesigned the G++ compiler; a option which the designers of PRESTO considered and discarded.

Both ESP and PRESTO have numerous advantages for parallel programming. Each contains features and flaws which the other lacks. For parallel computing on a single machine, PRESTO may be the best choice, but ESP is the future.

Chapter 3

ESP Software

3.1 ESP Software Overview

ESP (Extensible Software Platform) is a collection of software modules which can be assembled into experimental platforms for parallel-computing systems. This approach allows researchers to quickly implement and configure an operating system for novel hardware architectures or application structures. [1, 3]

As previously mentioned, ESP software design imposes a two-level structure on the system structure. The first level is comprised of the software front-end (the shadow process) which interfaces with the user and allows for general input and output. Multiple shadow processes may exist at once.

The second level contains the actual application software which executes ESP processes. These applications are usually placed on separate hardware platforms in order to enhance application performance, however they may be all or partially upon the same platform that contains the shadow process.

3.2 ESP Processes

The actual ESP environment is composed of four processes:

- [1] The Shadow process on the front-end system
- [2] An ISSD process, which is attached to both the Shadow process and the Mail Daemon via sockets.
- [3] One or more Mail Daemon processes
- [4] One of more Node operating system kernels

Each of these processes may run on a separate hardware platform, or on the same platform. In the case of the Sequent however, we assume that all processes use the same platform. This is required due to the big-little endian conflict between platforms which prevents the Sequent from communicating with other ESP platforms; the Sequent is little endian while all other ESP platforms are big endian. On a big endian computer, byte 0 is the low-order byte. On a little endian computer, byte 0 is the high-order byte. The Sequent, which uses Intel 80x86 processors, is little endian. Suns, which use the Motorola 68000 family is big endian. An approach to allow big/little endian platforms to interact in the same ESP network is currently being resolved at MCC.

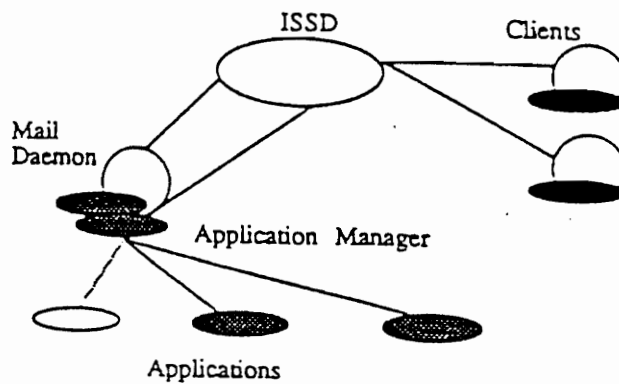


Figure 3-1. ESP Processes

3.2.1 The Shadow Process

The shadow process (shadow) is the application front-end of the ESP environment. This process provides command interpretation and I/O management for the user during an ESP application. [1] The services provided by the shadow are:

- [1] Establishing the connection to the ISSD process in order to start the ESP application
- [2] Shadowing the state of the application to the user (i.e. reporting on the ESP application's status)
- [3] Interpreting local UNIX signals into commands that may affect the operation of the ESP application (i.e. CNTL-C)
- [4] Providing system and I/O services between the front-end system and the ESP application
- [5] Providing facilities for monitoring and debugging the ESP application [13, 1-6]

3.2.2 The ISSD Process

The ISSD process can be considered the communications gateway between the ESP application and the ESP kernel. From the ISSD, the user can monitor, control or change the status of applications or any of the ESP processes. The functions of the ISSD are:

- [1] Establishing the connection between the shadow process(es) and all mail daemon process(es)
- [2] Sending messages to a kernel which invokes the application manager and kernel extensions to the ESP kernel.
- [3] Sending messages to the application manager to invoke subsequent applications.
[13, 1-6]

3.2.3 The Mail Daemon Process

The mail daemon passes messages between the ISSD (and other ESP kernels) and the ESP kernel on its node. There must be a mail daemon process on each node which contains an ESP process. [13, 1-6]

3.2.4 The Node Kernel

The node kernel is responsible for running applications begun by the user with the shadow process. The responsibilities of the kernel are:

- [1] Providing the rudimentary functions of an operating system for each node, such as managing local memory and scheduling local operations.
- [2] Dynamically loading and linking object modules as required for kernel and application execution.
- [3] Passing messages between nodes.
- [4] Keeping track of remote operations and their effect on local tasks.
- [5] Providing local support for configuring, instrumenting and debugging local operations. [13, 1-8]

3.3 Public Service Objects

In order to maintain maximal configurability and portability, the ESP kernel has been kept as small as possible. This was intentionally done so that a single individual could comprehend all the details. [5,32] One of the methods used to accomplish this is the use of Public Service Objects (PSOs) to extend the ESP kernel. "In contrast with the kernel, the PSOs are 'application level' objects, written to interface with the kernel services. PSOs provide traditional operating system facilities such as file I/O, interval timers, exception handling, etc. [Khan 89]" [1,5]

Public Service Objects serve a dual purpose:

- [1] They provide an interface through which an application may gain access to services offered by the kernel (for example, low-level I/O)
- [2] They provide services like environmental facilities and signal handling to applications. In the process, they relieve the kernel from the responsibility of providing support for these, and many other services. Thus, the design of the kernel is smaller, simpler and optimized for the functionality that it has to offer. [6,38]

By placing all but the essential services of an operating system in PSOs, the experimenter gains through:

- [1] Being able to control the kinds of services which are offered by choosing which PSOs to load.
- [2] The ability to configure the algorithms and policies used to implement PSO services. [6,38]

A final advantage of PSOs is that each PSO is a modular unit which contains no machine-dependent code. [1,5] Thus, a researcher can easily modify or configure a PSO and have a source-compatible program across all machine architectures in the ESP environment. [1,5]

The PSOs currently used in the ESP kernel are the Application Manager PSO, the ISSD_File PSO, the Local_File PSO, the Pod PSO, the Class Librarian PSO and the Env(ironment) PSO.

3.3.1 Application Manager

The application manager is a required PSO. It must always be running in the ESP kernel. The tasks of the Application Manager are:

- [1] Creating new applications

- [2] Providing a communications link between the application and the user's front end
- [3] Passing signals to the appropriate application
- [4] Deleting a specified application [6-39]

3.3.2 issd_File

"...applications executing in the ESP environment need to communicate with their front end. At the same time, they should not have to sacrifice speed when involved in I/O intensive computations that manipulate large amounts of data for local calculations."
[6,41] The issd_File PSO is used to relieve the kernel of that responsibility.

3.3.3 Local_File

The Local_File PSO is used to access local files directly instead of having to go through the issd to obtain them. [6,41]

3.3.4 Pod/Probe

The Probe PSO was designed to obtain data about the execution of a program. [6,41]
The Pod Service Object provides the required buffering and sequencing for the data before writing it to a file. [6,41]

3.3.5 Class Librarian

The Class Librarian is another required PSO. The Class Librarian addresses:

- [1] Name space management

- [2] Class search policy
- [3] Performance statistics
- [4] Development environment support [6,42]

3.3.6 Env

"This class is important because it exports the user's environment to the ESP domain. An environment is associated with each application and may be used by it to access the user's front end. This is particularly useful when an application needs to access the user's file system" [6,40]

Chapter 4

The ESP Compiler

4.1 Why the G++ Compiler

In order to compile the ESP system, it was required that a version of GNU G++ no later than version 1.36.3 be compiled for the Sequent. This was necessary since MCC used a modified version of the G++ compiler in compiling ESP programs. The choice of G++ was due to the fact that the GNU Foundation distributes the entire source of G++ freely. A G++ version of at least 1.36.3 was required since it was the earliest version which contained the updates necessary to compile the release of ESP used in this project.

4.2 What Source Was Required

The Sequent version of G++ was only recently released at the time of this project and was still considered a beta version. In order to compile G++, it was necessary that source be obtained for the following: bison, gas, clone, gcc 1.36.3, g++ 1.36.3, gnumake and libg++ 1.36.3. It was also required that a set of patches be obtained from MCC in order to patch the GNU C++ compiler to that it would compile the ESP programs.

Bison is the GNU parser for YACC, gas is the GNU assembler, clone is a program which clones entire directories, gcc is the GNU C compiler, g++ is the GNU C++ compiler, gnumake is the GNU "make" program and libg++ is the GNU C++ library.

4.3 Differences between standard and MCC G++.

The MCC patches when applied to the G++ compiler resulted in several changes in the manner with which a typical 'C++' program would be written. Several of the changes were required due to the fact that ESP kernels do not have access to all memory locations since ESP applications may execute over a distributed network of ESP machines.

4.3.1 General Differences

- [1] There are no global or static variables
- [2] ESP applications do not have a main() statement, and therefore do not support argc and argv[] directly. These are supplied from the env public service object.
- [3] There is no access to public variables via the '->' operator.
- [4] An object cannot use references or pointers to the methods or data of other objects.
[13, 3-23]

4.3.2 ->() Operator

An important difference between methods in ESP G++ and normal G++ is the way which the '->' operator is defined. [3, 3-13]

As discussed previously, the '->' operator has been overloaded so that remote method invocation may be accomplished. The syntax of the '->()' operator is the same when the executed method is local or remote.

4.3.3 New Operator

The 'new' operator is used to allocate memory for an object. This object would normally be instantiated in the memory of the process which made the call. However, in the distributed ESP environment, a new object can potentially be created on any of the ESP nodes. Thus, the 'new' operator was redefined.

The MCC patches to G++ allow the 'new' operator to be used such that objects may be created on a specific node or any node in the ESP environment. The choice of the node to instantiate the object may even be delayed until run-time. Since ESP objects are always created in heap memory, they can be dynamically loaded and linked as the application requires them. Thus, ESP G++ redefines new to have optional arguments.

The first (standard) method of using 'new' is:

```
new object_type(<arguments>);
```

where object_type is the object to be created, and <arguments> is the argument list to be passed to the constructor for that object. This type of object is not addressable across nodes. [13, 3-9]

The second method for using 'new' is:

```
<ident> = (<type>*) new {<relation>} type (<args>);
```

where <ident> is the field which receives the handle of the object. [13, 3-10] A handle is a globally unique identifier for an object in the system. A handle contains four fields:

- [1] The node ID, which identifies the node upon which the object was created.
- [2] The application ID, which identifies an application.
- [3] The class ID, which identifies the class of which the created object is a member.
- [4] The instance ID, which identifies the particular object. [13, 3-11]

<type>* is the class name, used for type casting the handle.

<relation> specifies where the object should be created. This variable has several possible values:

local: create the instance at the node where the call originated

remote: create the instance anywhere but the node where the call originated

anywhere: create the instance on any node

SAMEAS(inst): create the instance on the same node as instance 'inst'

DIFFERENT(inst): create the instance on any node but the node where 'inst' was created.

next: create the instance at the next node after the invoking node

AFTER(node_id): create the instance at the next node after 'node_id'

DIFF_NODE(node_id): create the instance on any node other than 'node_id'

node_id: create the instance on 'node_id'

<args> is the argument list passed to the object constructor.

<type> is the class name. [13, 3-10]

4.4 Virtual Functions

Virtual functions in C++ are the means through which remote method calls are implemented in ESP. A virtual function allows the programmer to define functions in a base class which may be redefined in a derived class. This allows objects derived from a similar base class to be treated uniformly. An object which is an instance of a class with virtual functions contains a "vtable" which is a table of pointers to the object's virtual functions. A virtual function on a remote node can be identified by its index in the vtable. [13, 3-8]

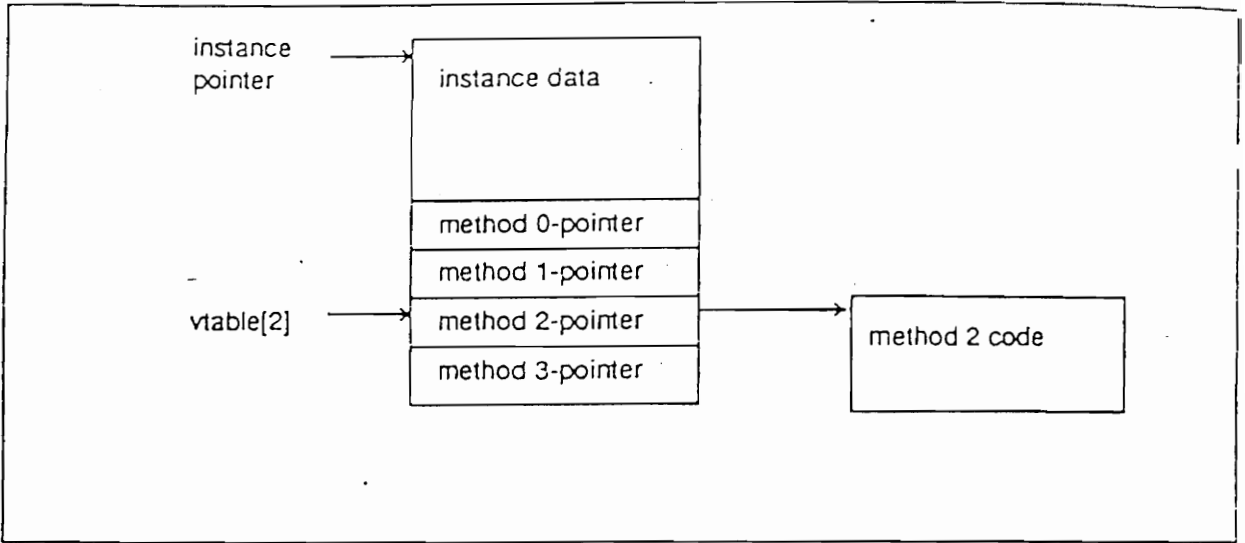


Figure 4-1. Virtual Function Table

Usually, virtual functions are identified by prefacing the keyword 'virtual' before the method declaration in the base class. However, all ESP programs and applications are compiled with the compiler flag '-fall_virtual' which signifies that all functions are virtual. This eliminates the need to declare each method as virtual. [13, 3-8]

4.5 Making the G++ Compiler

Even though G++ had purportedly been ported to the Sequent, making the compiler proved to be the most difficult and time-consuming portion of the project.

4.5.1 Obtaining the Compiler

MCC had source for all the required programs available along with makefiles to compile them. However, the makefiles would not run on a Sequent since they contained embedded "if-then" statements which the Sequent shell refused to handle. Therefore, the source code for the compiler was obtained from uunet.uu.net in order to obtain the standard makefiles which did not contain embedded "if-then" statements. This was deemed more expedient than attempting to unravel and rewrite the MCC makefiles which were exceedingly complex as they attempted to deal with all systems on which ESP ran. Note: due to the problems encountered with the makefiles, MCC is considered a simplification of their makefile logic.

4.5.2 Making the GNU Assembler

The GNU assembler, version 1.34, was the first program compiled. There were no difficulties in the compilation excepting that the 'strchr()' and 'strrchr()' functions were not defined since the Sequent is a BSD based system while those particular calls are System V based. A define statement was inserted which defined 'strchr()' as 'index()' and 'strrchr()' as 'rindex()'. The GNU assembler subsequently fully compiled and ran successfully.

After installation of the GNU assembler as 'gas' it was later noted that compilations appeared to be using the standard compiler, 'as', even though the GNU binary directory was being scanned before the directory containing 'as'. Thus, a symbolic link to gas named 'as' was created and all compiles thereafter successfully used the GNU assembler.

Later in the project, an attempt was made to upgrade to GNU gas 1.38.1. However, although the compile of gas 1.38 went smoothly, the compiler created .o files in the old NS32000 (BBN Butterfly) format which could not be linked. The cause of this problem could not be determined, so gas 1.34 continued to be the standard assembler.

4.5.3 Making the GNU C Compiler

GNU gcc 1.37.0 compiled cleanly without problems. The "bison" program was required to make gcc. It also compiled without difficulty.

4.5.4 Making the GNU C++ Compiler

In order to compile G++ 1.37.1, the procedure to be followed is:

- [1] Obtain the same version of gcc as the version of g++; in this case, gcc 1.37.1. Using a different version of gcc will usually not work. Make sure that the gcc and g++ main directories are named gcc and g++.
- [2] Compile the "patch" program. Apply the MCC patches. In the case of the Sequent Symmetry, use the patch-order.sun3-os4 patches except for the suppress-warnings patches. Use the patches in 1.37.0-1.37.1m.
- [3] Compile the "clone" program. Using the clone program, clone the gcc source directory to a build directory. (i.e. clone gcc build)
- [4] Use the clone program with the "force" option (-f) to clone the g++ source on top of the cloned gcc source. The force option replaces gcc source if a g++ source file has the same name. (i.e. clone -f g++ build)
- [5] Edit the Makefile. Uncomment the line "HAVE_UNISTD_H". This is because the definitions in /usr/include/unistd.h do not have the necessary define statements. Define "prefix" to be the directory where the source is to be installed.
- [6] Edit "cplus-dem.c" and "g++filt.c". Change "#include <string.h>" to "#include <strings.h>"
- [7] Compile and install the compiler using 'gnumake'. NOTE: the compilation will not succeed if 'gnumake' is not used.

4.5.5 Making the G++ Library

In order to compile libg++, g++ must be compiled without the -DNO_AUTO_OVERLOAD option. Thus, g++ should actually be made twice. The first time without the -DNO_AUTO_OVERLOAD option in order to compile libg++. Once libg++ is successfully installed, recompile g++ with the -DNO_AUTO_OVERLOAD option.

The only change to the libg++ source is that ON_EXIT should be undefined in gnulib3.c before reaching the "exit(status)" routine. This is to prevent multiple definitions of _exit when linking with -lc, the standard C library which also defines _exit.

In all compiles of GNU software, gnumake was used. In the case of G++, the use of gnumake was required as the standard Sequent make program would not pass makefile variables correctly and the compiled program would not run correctly.

4.6 Problems

There were several varying problems after the compiles were successfully completed.

4.6.1 Include files

The include files for G++ proved to be one of the more troublesome problems in getting G++ to work correctly.

4.6.1.1 Include file parameter lists

One of the advantages of G++ is that its include files contain prototypes of the parameter lists for the functions they define. However, in the case of ESP, there were often differences between the parameter lists in the include files and the parameter lists used in the ESP function descriptions. For example, often ESP parameter lists used void pointers in their parameter lists, while the parameter lists in the G++ include files would use

integers, long pointers, etc. When compiling, this difference would result in a fatal compilation error.

In order to resolve this problem, one of two routes could be taken. Either the include files could be made to appear similar to the ESP descriptions, or the ESP descriptions would have to be changed to appear like the include files.

In order to avoid any compiler/operating system misinterpretations, it was decided to modify the ESP descriptions to conform to the descriptions in the G++ include files since they represented the actual system definitions.

4.6.1.2 Include file changes

The following include files required changes for them to successfully interface with the ESP modules. In order to determine the Sequent parameter definitions for various functions, the manual page for that function was consulted.

- [1] <sys/ioctl.h> required all variables in macro definitions to be put in single quotes.
- [2] <sys/socket.h> had the parameter definitions for 'select()' to match the Sequent definitions.
- [3] <sys/stat.h> had its functions marked as being external and its parameter definitions changed to match the Sequent definitions.
- [4] <sys/ttychars.h> required all variables in macro definitions to be placed in single quotes.
- [5] <netdb.h> had the parameter definitions for its functions changed to match the Sequent definitions.
- [6] <time.h> had the parameter definitions for its functions changed to match the Sequent definitions.
- [7] <netinet/in.h> did not exist, so it was copied from the standard Sequent include file and its parameter listings modified to coincide with the Sequent parameter

definitions for the relevant function.

4.6.2 Compiler bugs

After using the 1.37.1 compiler for several months, a compiler bug was discovered. This was discovered when the kernel was first being debugged under gdb, the GNU source debugger. When the kernel is being debugged under gdb, the kernel arguments must be passed in through the argument vector list. The kernel uses function "build_configuration_from_argv(**argv)" to make the argv list appear similar to the standard argv list when a kernel is started by the ESP mail daemon.

When the function was used, the argument list was non-existent. However, when the argument list was printed to determine what was occurring, the procedure worked correctly. Through examination of the assembly code it was determined that the compiler kept the result of the calculation on the stack and later erased it without saving the value to memory. Printing out the variable values forced a save to memory. This problem was later confirmed by Kyle Jones at UUnet Communication Systems.

In order to solve this dilemma, we attempted to use a later version of the compiler; version 1.39. However, the MCC patches would not work with the 1.39 compiler and the patches only went up through G++ 1.37.2. However, it was discovered that by compiling G++ 1.37.1 with GCC 1.39, the problem disappeared. Apparently, the problem was a side-effect of using GCC 1.37.0 to compile G++ 1.37.1 based upon GCC 1.37.0. (i.e. using the compiler to compile itself).

Chapter 5

Porting ESP

5.1 Overview

The port of the ESP system was undertaken in several steps. The first step was to successfully compile all the necessary ESP components. Since all the ESP processes were to run on the Sequent, all the required programs needed to be ported: the ISSD, the mail daemon, shadow process, PSOs and the ESP kernel. It was originally planned to run the ISSD on a Sun instead of the Sequent, but the big-little endian conflicts between the systems rendered that option undesirable.

For the most part, the majority of the modules compiled without difficulty; especially the shadow process and PSOs -- which were written with portability in mind. The ISSD and mail daemon required some minor changes, but the majority of modifications were performed on the kernel itself. None of the changes substantially affected the working of any module and were usually cosmetic changes, or were required because of the differences of the Sequent operating system or platform. This section is divided into descriptions of the changes required in the ISSD, mail daemon, kernel and changes which affected more than one area; for example, `utils.h` is included in the kernel, PSOs and ISSD.

The majority of the changes made were apparent quickly. The necessity for them appeared during compilation or linking. Several however, were non-trivial to find and

required substantial amounts of time to locate.

It should also be noted that the kernel was in effect ported twice. A majority of the way through the port, MCC released a new version of ESP. This new version corrected several errors and was semantically a cleaner version of the ESP operating system. At MCC's request, work was switched to the new version of ESP. However, this switch cost several weeks work in that many of the patches previously required to enable ESP to compile on the Sequent were no longer required and some new compile problems now appeared. The changes noted here are for the second port of the ESP system.

5.2 ISSD Modifications

As previously mentioned, the ISSD is the communications gateway for the ESP system. There is only one ISSD process.

5.2.1 client_base.h

This file required the inclusion of `<signal.h>` and required code to enable a byteorder change (see section 5.6.4).

5.2.2 issd.cc

This file required code to enable a byteorder change (see section 5.6.4) and the addition of a section for the Sequent with regard to the location of the PSO path (i.e. `'pso.sequent/')`.

5.2.3 issd_File.cc

Two changes were made to `issd_File.cc`. The first was to move the position of the line `#include "issd_File.h"` so that it could take advantage of the includes done by `file_thing.cc` which was also included in `issd_File.cc`. The second change was to change the line:

```
#include "app_mgt.h"
```

to

```
#ifndef ESKERNEL
#include "app_mgt.h"
#endif
```

so that `"app_mgt.h"` would be included in non-kernel compiles. This was performed since `issd_File.cc` is used in both compiles for the kernel and the ISSD.

5.2.4 `issd_File.h`

The only change required to this file was to modify line:

```
#include "file_thing.h"
```

to

```
#ifdef ESKERNEL
#include "file_thing.h"
#endif
```

so that `"file_thing.h"` is only included in kernel compiles.

5.3 Public Service Object Modifications

The only PSO which required modification was `'klib.cc'` which required a case for `'sequent_node'` to be inserted.

5.4 Mail Daemon Modifications

The mail_daemon is the process responsible for routing messages between the various kernels on its node and the ISSD. There is one mail_daemon per node.

5.4.1 mail_daemon.sequent

The mail_daemon.sequent file is the same as the mail_daemon.sun3 file, only instead of being called connect_SUN3(), it is called connect_SEQUENT().

5.4.2 mail_daemon.cc

This function is similar to the Sun version except that instead of the Sun routines, duplicate Sequent functions have been created. Thus, the module includes 'mail_daemon.sequent' instead of 'mail_daemon.sun3' and in 'connect_member()' calls 'connect_SEQUENT()' instead of 'connect_SUN3()'.

Other modifications to mail_daemon.cc are that the file requires byteorder code (see Section 5.6.4), contains a case for sequent_node, which is a duplication of the sun3_node code, adds "Sequent" to the 'types' array, and had the code for 'vsprintf()' (see Section 5.6.3) added to it.

5.5 Kernel Modifications

As discussed earlier, the vast majority of the modifications made to the ESP source were made to the ESP kernel. The following section is divided into the individual modules which required modification, with subsections if there were several substantial changes to the file. The appendix, at the end of the paper, describes minor changes which were required in order for the code to successfully compile.

5.5.1 platform.h

The platform.h file is where machine-dependent code (e.g. assembly language routines) are kept.

5.5.1.1 Semantic changes

The modifications to platform.h were: the line `'typedef void (*function_pointer) (void *);'` was added and the line `'extern int printf (char*,...);'` was removed due to a conflict with a previous definition.

If `'SEQUENT'` has not been defined, it is defined here.

5.5.1.2 Context Switching

Porting ESP to the Intel 80x86 architecture, involved writing code to both save the current context and restore it. Note: since the GNU compilers were compiled without 80387 support included, they did not use 80387 calls and thus no 80387 support was included in the context switching code. This code was inlined to avoid the procedure call overhead.

Additional code was required to obtain specific values, such as the frame and stack pointers. A problem in writing this code is that no documentation could be located on how to specify a specific register in the G++ asm (assembly) instructions. It was later determined via contacts at MCC that G++ contained an undocumented instruction `'%%'` which allowed for usage of specific registers. An example of one of these routines would be:

```

inline void* get_stack_ptr()
{
    asm("# get_stack_ptr");
    register void *result;
    asm("movl %%esp,%0" : "=d" (result) : );
    return(result);
}

```

Figure 5-1. G++ assembly example

This code segment would move the contents of the esp register into the 'result' variable, and then return the 'result' variable.

5.5.1.3 ff1() call

The ff1() call is a routine which examines an input integer. If the integer is 0, the routine returns -1, otherwise it returns the bit position of the most significant bit. This routine was written in assembly language for the Sun, but was written in a mixture of C++ and assembly language for the Sequent for both clarity and ease of implementation. Both routines were inlined to avoid the procedural call overhead.

5.5.1.4 Miscellaneous

It should be noted that the size of the shared memory space is defined in platform.h. Also, a bug in the function which obtains the initial memory space for the kernel (get_initial_memory()) has been fixed and MCC notified. The function was supposed to return 2**20 bytes of memory but actually returned 2**21 bytes of memory. MCC was made aware of the error and corrected it in their release.

5.5.2 mem_mgt.cc

The memory management class contains the code for managing the memory space of the ESP kernel.

5.5.2.1 Shared Memory Message Allocator

The message allocator is set to access the shared memory message space in this routine. Previously, the message allocator used the standard memory allocator.

```
#ifdef SHMEM
    Kernel_Family[MSG_ALLOC_ID] = (void*)SH_MEM_MGR;
#else
    Kernel_Family[MSG_ALLOC_ID] = (void*)MEM_ALLOC;
#endif
```

Note that since this routine access variables relating to the shared memory manager, 'sh_mem_mgr.h' must be included.

5.5.3 sys_configuration.cc

The sys_configuration file contains some of the initial kernel startup code used to configure the ESP kernel.

5.5.3.1 SEQUENTHOST->SUNHOST

There are several locations in this file where Sun-specific code was written. This code was placed in #ifdef SUNHOST -- #endif sections. Since all the Sun code was relevant to the Sequent as well, a section of code was added which specified that if SEQUENTHOST was defined, SUNHOST was defined as well.

5.5.3.2 Node_Type()

The Node_Type() function returns a string identifier corresponding to the node type. An entry was added to the function for 'sequent_node'.

5.5.3.3 sys_configuration()

In function sys_configuration(), the function has a special case if the node is a Sun3 node or a Sparc node. This case is also true if the node is a Sequent node so the appropriate logic was added to the function.

5.5.4 sys_configuration.h

There are several locations in this file where Sun-specific code was written. This code was placed in #ifdef SUNHOST -- #endif sections. Since all the Sun code was relevant to the Sequent as well, a section of code was added which specified that if SEQUENTHOST was defined, SUNHOST was defined as well.

5.5.5 symbol_table.cc

This function contains a table of the external symbols that are 'exported' by the kernel to the ESP applications. Several of the symbols in this table follow the format of the Motorola 88000 instead of the Sun3 format. These functions are: Abort, Delete, Node_Type and _savest.

5.5.6 def.h

The 'def.h' file contains general definitions which are used throughout all modules of the

ESP kernel.

5.5.6.1 Kernel_Family

The def.h file contains many of the general kernel-wide definitions. One group of these definitions is the kernel object class_id assignments such as the node manager, future manager and memory management manager. The definition for the shared memory manager was also placed here as well as the redefined message allocator.

```
#define MSG_ALLOC_ID    16
#ifdef SHMEM
#define MSG_ALLOC    ((shared_mem_manager*) Kernel_Family[MSG_ALLOC_ID])
#else
#define MSG_ALLOC    ((mem_all*) Kernel_Family[MSG_ALLOC_ID])
#endif
#ifdef SHMEM
#define SH_MEM_MGR_ID  19
#define SH_MEM_MGR    ((shared_mem_manager*) Kernel_Family[SH_MEM_MGR_ID])
#endif
```

Figure 5-2. ESP Kernel definitions

It should be noted that the size of Kernel_Family is limited to 20 entries and that all of them are now filled. If any additional ones are to be entered, the size of the array must be adjusted.

5.5.6.2 node_types

The enumerated type 'node_types' lists the platforms on which ESP runs. The Sequent was added to this list under the name 'sequent_node'.

5.5.7 msg_stuff.cc

There are several locations in this file where Sun-specific code was written. This code was placed in `#ifdef SUNHOST -- #endif` sections. Since all the Sun code was relevant to the Sequent as well, a section of code was added which specified that if `SEQUENTHOST` was defined, `SUNHOST` was defined as well.

5.5.8 msg_stuff.h

`msg_stuff.h` includes `'sh_mem_mgr.h'` in its list of include files. This was done because several modules which reference shared memory code also include `'msg_stuff.h'` and it simplified matters to place the shared memory management include in one file instead of several.

5.5.9 remote_base.cc

`remote_base::delete_instance()` contained a section of code which had one section marked as written for the Sun OS and another default section for all other systems. It was determined the Sun OS version was appropriate for the Sequent. Originally, this was accomplished by an educated guess, however, once the system was firmly established, a changeover was made to the default and the ESP system failed thus confirming the hypothesis.

5.5.10 Dynamic Linking

The dynamic linker is the module which reads in the ESP applications/PSOs, links them with the functions exported by the kernel, and loads them into memory. Both the Sun and Sequent operating system use the same format for binary files (`a.out`) and thus no

modification of this code was required.

Note that the Sequent a.out header files contains entries for shared memory space. These entries will not be correctly interpreted by the ESP loader/linker if they are used. None of the tests performed on the Sequent made use of these entries.

5.5.11 ftio_lib.cc

The ftio_lib file contains code to convert ascii format to binary and binary code to ascii. This code was written for a big endian machine, thus it required modification to work on a little endian platform.

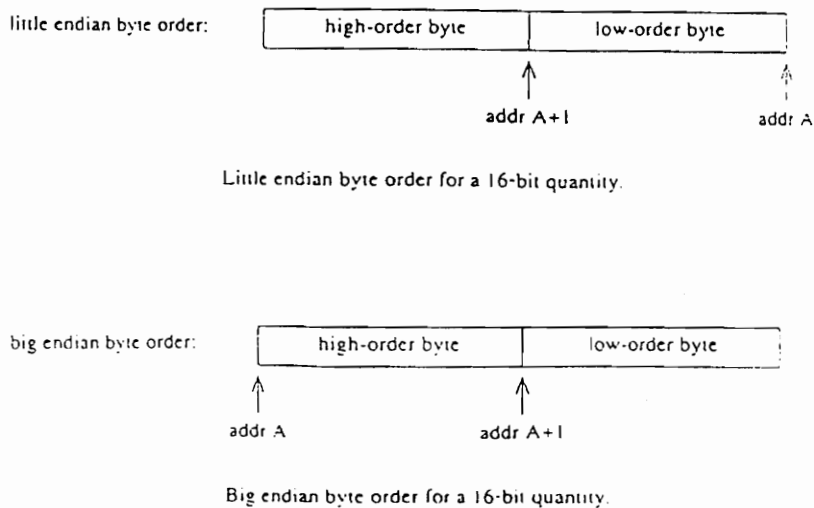


Figure 5-3. Big/Little Endian

In function FtoA, a floating point value is to be converted to an ASCII representation. By increasing the pointer the floating point value by one, the code is able to compensate for the hardware differences between the Sequent and Sun systems.

```

char * FtoA(float fvalue, char * pbuf)
{
    // try to keep about 6 significant digits
    unsigned pfvalue = (unsigned*)&fvalue;
#ifdef SEQUENT
    pfvalue++;
#endif
    int exponent = (*pfvalue >> 23) & 0xff;
    unsigned significand = *pfvalue & 0x7ffff;
    if (exponent == 0xff)
    {
        // then this is an exceptional condition
        if (significand == 0)
            // this is an infinite value
            return("Inf");
        else
            // this represents an illegal value ("Not a Number")
            return("NaN");
    }
    if (exponent == 0)
    {
        // look out here
        if (significand == 0)
            // this is true 0
            return("0.0");
    }
    else
        // there is an implied leading bit
        significand |= 0x800000;
    // 0.significand X 2**exponent
    exponent -= 127;
    // now, shift the implied binary point to the right of the significand
    // significand.0 X 2**exponent
    exponent -= 23;
    return(convert((fvalue < 0), exponent, significand,pbuf));
}

```

Figure 5-4. FtoA code

In function DtoA(), a double precision variable is to be converted to ASCII format. As was done in FtoA(), by increasing the pointer to the double by one, the exponent may be obtained, and by moving it back down the significand can be determined.

```
char *DtoA(double fvalue, char *pbuf)
{
    // we don't get all of the bits, but a few more than float
    unsigned pfvalue = (unsigned*)&fvalue;
#ifdef SEQUENT
    pfvalue++;
#endif
    int exponent = (*pfvalue >> 20) & 0x7ff;
    unsigned significand = *pfvalue & 0xffff;
    significand <<= 7;
#ifdef SEQUENT
    significand |= (*(pfvalue-1) >> 25);
#else
    significand |= (*pfvalue >> 25);
#endif
    if (exponent == 0x7ff)
    {
        // then this is an exceptional condition
        if (significand == 0)
            // this is an infinite value
            return("Inf");
        else
            // this represents an illegal value ("Not a Number")
            return("Nan");
    }
    if (exponent == 0)
    {
        // look out here
        if (significand == 0)
            // this is true 0
            return ("0.0");
    }
}
```

```

else
    // there is an implied leading bit
    significand |= 0x8000000;
    exponent -= 7;          // correct for the shift performed earlier
    exponent -= 1023; // 0. significand X 2**exponent
    // now, shift the implied binary point to the right of the significand
    exponent -= 20;       // significand.0 X 2**exponent
    return(convert((fvalue < 0), exponent, significand, pbuf);
}

```

Figure 5-5. Dtoa code

5.6 Multiple Change areas

The following modules/changes are used in multiple programs.

5.6.1 utils.h

The utils.h file contains code for the basic functions which the ESP processes rely on, such as bcopy, strcmp, and itoa. However, some of this code was written under the assumption it was for a big endian architecture, thus some portion required modification for a little endian platform. Both strcmp() and strcpy() required changes for them to perform correctly on a little endian machine.

Another modification required to utils.h regarded the Clock() routine. The Sun operating system contains the function clock() which allows a process to read the system clock. The Sequent operating system does not contain the clock() system call, so code was inserted to force the code to default to using the gettimeofday() system call instead of clock() to read the internal clock.

5.6.2 Message Output

The ESP kernel and mail daemon were configured to write to `/dev/console`. However, since the console was unavailable, due to its being in a secure room, it was necessary to modify the kernel to write to a specific tty. This was accomplished by having the kernel open a file in the user's home directory called `'devtty'`. This file contained the name of the tty that the user wished the kernel to write to. The kernel then read in that name and used it in opening the terminal. This task was performed in `'kick.cc'` for the kernel and `'mail_daemon.cc'` for the mail daemon.

5.6.3 `vprintf()` family

Several of the kernel calls which the ESP kernel made were to the `vprintf()` family of system calls. These calls were used in formatting messages to be printed by the kernel. However, these calls did not exist for the Sequent operating system. MCC was also unable to provide source code for the `vprintf()` family.

After several days of searching, source code for the calls was located in the BSD archive at node `uunet.uu.net`. This code was freely distributed by the University of California at Berkeley and so could be incorporated into the ESP kernel without difficulty so long as the copyright notice was preserved in the source file.

5.6.4 Byteorder problems

As previously mentioned, the Sequent uses a big endian format whereas the Sun hardware (upon which ESP was designed) uses a little endian format.

The socket ports to which ESP processes connect are kept in a file called calling the `'getservbyname()'` function. However, UNIX convention requires the result of the `'getservbyname()'` function to be returned in big endian format. Thus, it was necessary to

convert the result back to little endian format thru the use of the 'ntohs()' function (network-to-host, short integer). This call was required anywhere the 'getservbyname()' function was used: mail_daemon, issd and kernel.

Chapter 6

Shared Memory Implementation

6.1 Background

ESP objects communicate through messages sent to each other. A message is sent to the `mail_daemon` on the node on which the sending object is located. The `mail_daemon` then examines the address of the receiver object in the message to determine whether the message should be sent to a kernel executing on the same node or a remote kernel.

The standard method of communication between ESP processes is through UNIX sockets. This makes sense in a distributed configuration of nodes in which one kernel is resident on each node; there is no other way for them to communicate. However, the Sequent is a multiple processor shared memory machine. Therefore, it is reasonable to assume that several ESP kernels could be run on this one node. The use of sockets for interprocess communication, while reliable, is still extremely slow compared to memory-memory operations. Furthermore, in a configuration where all processes reside on a single node with shared memory, the need for sockets is superfluous since a message can be transferred directly through memory.

Since the Sequent architecture allows for shared memory segments, it was decided to create a shared memory message passing environment for Sequent kernels on the same node. To achieve this result, a shared memory manager was created to allow the passing of messages to Sequent ESP kernels through shared memory instead of sockets.

Messages to non-kernel processes or ESP kernels not on the Sequent continued to use the socket implementation which was kept intact for this purpose.

6.2 Implementation

The shared memory system allows each Sequent kernel to map a section of the shared memory into its address space. Although the virtual address spaces will differ for each kernel, they will point to the same real address space. Thus, each Sequent kernel has access to a section of memory accessible by all other Sequent kernels.

6.2.1 Initialization

The shared memory manager is instantiated in the file `kick.cc` in function `gen_node()`. This function is responsible for the startup of the ESP kernel. It is important that the shared memory manager be created before the shared memory mailer because the shared memory mailer calls "`get_msg_q_base()`" which is a function of the shared memory manager. Directly after the shared memory manager is created, the shared memory mailer is instantiated.

6.2.2 Shared Memory Manager

The shared memory manager is responsible for allocating the shared memory space, initializing and managing it.

6.2.2.1 Obtaining Shared Memory

There were two different ways to allocate the shared memory. The first method, using `shmalloc(sz)`, returns a shared memory block of size `sz`. The second method, using the

mmap() call, allocates a section of shared memory through the use of a memory-mapped file. For simplicity, the mmap() call was used; an additional factor was that the mmap call can be used for systems without shared memory.

As can be seen in the following code segment in Figure 6-1, the first task is to obtain the page size of memory. This is required since the memory pagesize can be changed in the system configuration and therefore cannot be assumed for every Sequent Symmetry platform.

The second task is to set the top of memory to a page boundary. This is done so that the shared memory segment can start on a page boundary which allows for better system allocation of the memory since it is required that the shared memory be allocated in multiples of the system page size. The page boundary is found by the call:

```
(((int)sbrk(0) + (pgsz-1)) & ~(pgsz-1))
```

where 'sbrk(0)' obtains the current top of memory and pgsz is the page size of memory in bytes.

Once the top of memory has been found and rounded up to a page boundary, it is set there by a 'brk()' call. An 'sbrk()' call is then made to increase the data space of the process by the amount of space which the shared memory is to take. The shared memory is then mapped into this space.

The shared memory segment is mapped into a memory-mapped file obtained by 'sbrk' thru the 'mmap()' call. The 'mmap()' call works by using opening a file which must be at least the size of the space to be mapped (i.e. if we were to map 4096 bytes of shared memory, the file must be at least 4096 bytes in size). To define the association between the main memory and the file space, the process opens a file named 'shmem' with both read and write permissions. This file is then mapped into the process address space via the 'mmap()' call which looks like:

```
mmap((caddr_t)shm_base, size, PROT_READ | PROT_WRITE, MAP_SHARED,  
fd, 0)
```

In this call, 'shm_base' is the top of memory location previously set. This is where the shared memory segment is to start. The parameter 'size' is the size of the shared memory. The bit-coded options, 'PROT_READ | PROT_WRITE' set the pages so that they may be both read and written by the process while 'MAP_SHARED' indicates that the pages are not local to the process and may be accessed by other processes. Finally, 'fd' is a file pointer to the file being mapped and '0' is the offset into the file at which the mapping should be started.

6.2.2.2 Initializing Shared memory

After the mapping of the shared memory is completed, the kernel with `gl_node_id` equal to 1 will initialize the shared memory. This is necessary since the shared memory manager assumes that the shared memory locks are zero to start with.

The sequence of events now turns to formatting the shared memory segment for use by the kernel. Each kernel performs the following tasks:

- * Skip the first 32 bits of shared memory. This space is used as a flag to be set by the kernel with `gl_node_id = 1` when it has finished configuring the shared memory.
- * Set the message queue base to the base of shared memory plus the 32 bit flag.
- * Set the size of the message queue(s) to the size of the message queue structure times the number of Sequent kernels running on that node.
- * Set the shared memory lock to the message queue base address plus the size of the message queues.
- * Set the base of the available shared memory to the shared memory lock address plus the size of the `spin_lock` structure.

After these steps have been accomplished, each kernel will determine the remaining shared memory space through the calculation:

```
rest = size - ((int)mem_all_base - (int)shm_base);
```

Each kernel will then attempt to create a shared memory allocator. Recall, however, that only the kernel with `gl_node_id == 1` initializes the memory. Every other kernel returns without having done any initialization. The actual initialization of the memory is exactly similar to the kernel memory manager.

Finally, if the kernel has `gl_node_id == 1`, it sets the `'mem_init_done'` flag. All other kernels enter a loop which waits until the `mem_init_done` flag has been set. When this occurs, the initialization of the shared memory segment is complete.

```

shared_mem_manager::shared_mem_manager(int size)
{
// attach a shared memory segment using mmap which is aligned on the first
// page beyond the beginning of the current data segment. It is critical
// that all kernels use the same address for this shared segment since
// the segment contains address of entries within the segment itself.

    shared_addr*    mem_all_base;    // space which allocator will use
    shared_addr*    mem_init_done;    // used to synchronize initializing
                                        // kernels
    char*           shm_base;         // virtual address of where shared
                                        // segment will be attached
    int              fd;               // file descriptor

    mutex = 0;
    int pgsz = getpagesize();
    shm_base = (char*) (((int)sbrk(0) + (pgsz-1)) & ~(pgsz-1));
    if ((int)brk(shm_base) < 0)
        Abort(0, "Cannot allocate base of shared memory.0);
    if ((int) sbrk(size) < 0)
        Abort(0, "Cannot allocate space for shared memory.0);
    if ((fd = open("shmem", O_RDWR, 0666)) < 0)
        Abort(0, "Cannot open shmem.0);
    int attach = mmap((caddr_t)shm_base, size, PROT_READ|PROT_WRITE,
        MAP_SHARED, fd, 0);
    if (attach < 0)
        Abort(0, "Cannot attach shared memory segment.0);
    // initialize the shared memory area; only done by kernel 1
    if (gl_node_id == 1)
        bzero((char*) shm_base, size);
}

```

```

// define the template of the shared segment
// use first word for flag
mem_init_done = (shared_addr*) shm_base;
msg_q_base = (shared_addr*)((int)mem_init_done + sizeof(int));
int msg_q_size = sizeof(msg_queue) * SH_MEM_NODES;
mem_lock = (spin_lock*)((int) msg_q_base + msg_q_size);
mem_all_base = (shared_addr*)((int) mem_lock + sizeof(spin_lock));
// establish the shared memory allocator
int rest = size - ((int) mem_all_base - (int) shm_base);
allocator = new sh_mem_all(mem_all_base, rest);
// kernel 1 alone will do the initialization of the shared memory
// (see the mem_all constructor for shared memory). All other kernels
// just hang around. A flag at address mem_init_done is used for
// the synchronization.
if (gl_node_id == 1)
    *((int*) mem_init_done) = 1;
else
{
    while (**int*) mem_init_done == 0);
}
}

```

Figure 6-1. `shared_mem_manager::shared_mem_manager()` code

6.3 Management of the Shared Memory

The shared memory segment is managed in the same way as kernel memory in that the same routines to structure the memory and access it are used. However, since multiple kernels can potentially access this segment, additional synchronization routines have been added to control the access to shared memory.

6.3.1 Accessing Shared Memory

When a kernel wishes to obtain a shared memory data segment, it calls the 'get_mem' method in the class shared_memory_manager. This method then attempts to gain a lock on the shared memory segment. If successful, it then obtains the required shared memory through a call to its shared memory allocator, which was created during the shared memory initialization. After the memory has been obtained, the method then calls another method to release the lock on the shared memory segment. The method for freeing shared memory works on the same principle.

```
shared_addr* shared_mem_manager::get_mem(unsigned bytes)
{
    shared_addr* where;
    mem_lock->acquire();    // exclude other shared memory operations
    mutex++;
    if (mutex > 1)
        Abort(0, "SHARED MEMORY LOCK BROKEN!!!0);
    // set "where" to point to the shared memory address
    where = (shared_addr*) allocator->get_mem(bytes);
    mutex--;
    mem_lock->release();    // release shared memory lock
    return(where);
}
```

```
void shared_mem_manager::free_mem(shared_addr* addr)
{
    mem_lock->acquire();    // exclude other shared memory operations
    mutex++;
    if (mutex > 1)
        Abort(0, "SHARED MEMORY LOCK BROKEN!!!0);
    // free shared memory
    allocator->free_mem(addr);
    mutex--;
    mem_lock->release();    // release shared memory lock
}
```

Figure 6-2. shared_mem_manager::get_mem() code

The method for gaining a lock on the shared memory segment is in the spin_lock class. The acquire method in this class attempts to gain a lock on the shared memory segment. It will not exit until a successful lock is accomplished.

A previous method of locking was attempted using semaphores, however, no atomic semaphore command could be created since the routine could be interrupted at any point by a context switch. Due to this fact, we were forced to use the Sequent locking routines which guaranteed atomic operation.

```
void spin_lock::acquire()
{
    // set lock, using memory location as lock
    m_lock((unsigned char*)&lock_state);
    mutex++;
    if (mutex > 1)
        Abort(0, "LOCKS ON MSG QUEUE BROKEN");
}
```

Figure 6-3. spin_lock::acquire() code

6.3.1.1 Low-Level locking routines

The m_lock() and m_unlock() routines are 'C' procedures used by the Sequent shared memory functions defined in <parallel/parallel.h>. These functions perform the actual locking/unlocking of the shared memory semaphores. They continually loop on the address ('laddr') until they can set it to the correct value. Mutual exclusion is guaranteed by these routines. These functions are used by the shared memory system to lock access

to the shared memory space when required.

```
#include <parallel/parallel.h>

int m_lock(laddr)
// address of the lock for the shared mail queue
unsigned char *laddr;
{
    if (laddr == 0)
        exit(9);
    // call Sequent locking code
    S_LOCK(laddr);
}

int m_unlock(laddr)
// address of the lock for the shared mail queue
unsigned char *laddr;
{
    // call Sequent unlocking code
    S_UNLOCK(laddr);
}
```

Figure 6-4. m_lock()/m_unlock() code

The 'release()' function decreases the mutex value and unlocks the memory space.

```
void spin_lock::release()
{
    mutex--;
    m_unlock((unsigned char*)&lock_state);
}
```

Figure 6-5. spin_lock::release code

6.3.2 Shared Memory Manager

The shared memory manager code was taken directly from the ESP kernel. It works in the same manner as the ESP memory manager excepting that unnecessary methods and kernel-specific sections of the memory manager have been deleted.

The only method which required significant change was the `sh_mem_all::sh_mem_all()` method, which is shown in Figure 6-6. This method is used to initialize the shared memory manager.

The `sh_mem_all()` method creates a fragment list with which to manage the shared memory. This fragment list is created within the shared memory space.

```
sh_mem_all::sh_mem_all(char * base, int size)
{
    int pgsz;
    void *allocation;

    // get the base of the shared memory space
    this = (sh_mem_all *)base;

    if (gl_node_id > 1) return; // the initialization of the shared
                                // memory space is only done by node 1

    pgsz = getpagesize();
    // set up space for the fragment list
    fragment_list = (fragment_desc *)(((int)base + sizeof(sh_mem_all) +
                                     (pgsz - 1) & ~(pgsz-1)));
    memory[0] = (void *)fragment_list;
```

```

fragment_descriptors = fragment_list_ndx
                        = INITIAL_FRAGMENT_DESCRIPTORS;

// initialize the fragment list
for (int flist_ndx = 0; flist_ndx < 33; flist_ndx++)
    Free_Head[flist_ndx] = NULL;
free_fragments = 0;

deallocations = 0;
coalescing = FALSE;

// amount we have allocated for the fragment lists
allocation = (void *) ((int)fragment_list +
    INITIAL_FRAGMENT_DESCRIPTORS*sizeof(fragment_desc));
memory[1] = allocation;
Allocated = size;
segments = 2;
available = size - ((int)allocation - (int)base);

appl_id = KERNEL_APPL_ID;

// this is a special file
Special = TRUE;

// put the free fragments in the free list
append_Free_List((free_fragments*)allocation, available);
last_reported = available;
delta_limit = MAX_STORAGE_CHANGE;
};

```

Figure 6-6. sh_mem_all:sh_mem_all() code

Chapter 7

Shared Memory Mailer

The shared memory mailer was created in order to provide a superior method of communication between Sequent kernels on the same node. These kernels may communicate through shared memory instead of socket communications which are inherently much slower for two reasons. First, the socket calls to send and receive data are really calls to the UNIX kernel, which means an additional delay due to the extra kernel code used to handle the request. Shared memory traffic only requires that the data be read in and out -- without the additional burden of UNIX kernel calls.

The second reason is that the ESP kernels do not communicate with each other directly, but through the mail daemon. Thus, sending a message to another kernel is really sending the message to the mail daemon which then sends it to the appropriate kernel. Through the shared memory system, if the message is to be delivered to a local kernel, the kernel accesses the shared memory mailer directly, instead of sending it to the mail daemon. Thus, for a small increase in kernel overhead, the kernel bypasses the overhead created by the socket call to connect to the mail daemon and also bypasses the mail daemon's overhead time.

7.1 sock_mail.cc

The `sock_mail` module is part of the `mail_system` class. Its job is to handle the mail traffic which is sent and arrives to the kernel from the socket. Since this module was the one which directly used the socket, it was felt that it would be the logical choice for adding the code to handle the shared memory mailer code for reading/sending messages.

There are four functions where the shared mailer code needed to be inserted; `mail_system()`, `wait()`, `fill()` and `mail()`. Also, since this additional code referenced shared memory variables, the `'fast_path.h'` header file was also included. This file contains the definitions for the shared memory methods and their assorted variables. For further discussion on the shared memory methods, reference section 7.4.

7.1.1 `mail_system::mail_system()`

The `mail_system()` call is the constructor for the `mail_system` class. It is used to initialize the mail system. Originally, this function simply sets `'read_lock'` to `FALSE`, indicating that no process was currently reading the sockets. However, to incorporate the shared memory mailer it was necessary to add a call to `fast_path()` which constructed the shared memory mailer pathway.

```
mail_system::mail_system()
{
    read_lock = FALSE;
#ifdef SHMEM
    fax = new fast_path();
#endif
}
```

Figure 7-1. `mail_system::mail_system()` code

7.1.2 mail_system::wait()

The wait() call does exactly what it implies: it waits until a message arrives. The previous method of accomplishing this task was to perform an infinite loop while waiting on a select() call.

The select call works according to the following principle. UNIX file descriptors are numbered 0 .. MAXFILEDESCRIPTORS. In the case of ESP, we assume MAXFILEDESCRIPTORS to be 31. The select() function is passed a 32-bit integer. Any bit which is turned on in that integer corresponds to a file descriptor which select will then watch. For example, if bits 0, 5 and 23 are set, select() will examine file descriptors 0, 5, and 23 to see if any input has arrived on those descriptors. The last parameter in the select function determines the timeout (i.e. how long select() will wait before giving up). If it is NULL, select() will block indefinitely.

With the addition of the shared memory mailer, a problem now existed. The shared memory information did not appear on a socket, so the select() call could not detect it. However, neither could the select() call be allowed to block indefinitely until data arrived on a socket lest we possibly end in deadlock. The solution was to have the system enter a loop which checked the shared memory mailer and then checked the sockets with an immediate timeout if nothing was found.

```

int mail_system::wait()
{
    // block until something arrives
    bitmask read_fd;
    int result;
#ifdef SHMEM
    // set immediate timeout
    timeval poll_timeout;

    poll_timeout.tv_sec = poll_timeout.tv_usec = 0;
#endif

    // perform a blocking select on the file descriptor of interest
    read_fd = socrw_bit_field;
#ifndef SHMEM
    while ( (result = select(32,read_fd,NULL,NULL,NULL) ) <= 0)
#else
    while ( (fax->msgs_avail() == 0) &&
            (result = select(32,read_fd,NULL,NULL,&poll_timeout)) <= 0)
#endif
    {
        if (result == -1)
        { // a message may have been stuffed during an interrupt
            if (errno != EINTR)
                Abort(0,"sock_mail::wait");
            return(0);
        }
        read_fd = socrw;
    }
    return(1);
}

```

Figure 7-2. mail_system::wait() code

7.1.3 mail_system::fill()

The original purpose of the `fill()` function was to fill the kernel's mailbox by reading messages from the socket. Since this is where messages are received, the code to receive messages from the shared memory channel was placed here.

The new shared memory code first reads all messages out of the shared memory and then reads messages from the socket.

```

mail_system::fill(post_off* mbox; int block_ok = TRUE)
{ // Fill fills upto MAXFIFO messages into mailbox to buffer them and order
  // them by priority. Fill never waits if there are already messages in the
  // mailbox. fill() should empty the socket every chance it gets in case
  // there is a high priority message after a low one.

    generic_msg* msg;
    int soc_size = 0;

    if (this->lock())
    { // already trying to read, don't start another
      return (FALSE);
    }

    if (block_ok && mbox->size() <= 0)
    { // if there is nothing in the mailbox wait for something on the socket
      MEM_ALLOC->extern_coalesce(); // coalesce memory while waiting
      while (this->wait() == 0)
      { // interrupts cause us to get out early. See if a message
        // got stuffed into the queue by mail_local
          if (mbox->size() > 0)
            // got a stuffed message
              return(1);
        }
    }

#ifdef SHMEM
    // get the number of bytes in the socket
    if ((soc_size = this->size()) == 0)
    { // if there is nothing in the socket after a wait we had
      // an error
        perror("socms: fill: bad read on socket, exiting");
        exit(1);
    }
}
}

```

```

else
{ // just check the state of the socket, we don't want to block
  // since we already have valid work to do
  //
  // get the number of bytes in the socket
  soc_size = this->size();
}
#else
}
// pick up ALL messages from the shared memory mail queues for this
// node first.

// while messages exists
while (fax->msgs_avail())
{
  msg = fax->receive(); // get message
  if (msg)
    // the test on gm is overkill
    mbox->new_message(msg);
}
#endif
while (this->size() > 0)
{ // read until the socket is empty or the mailbox is full
  // decrement the byte count on each read
  msg = this->spread();
  if (msg) {
    mbox->new_message(msg);
  }
}
return(1);
}

```

Figure 7-3. mail_system::fill() code

7.1.4 mail_system::mail()

The mail routine is the function which actually 'mails' messages to the appropriate system. The previous method was to first check if the mailer was currently locked. If it was not, then `mail_system::fill()` was called to read in messages. Afterwards, the message was sent on its way by calling `'spwrite()'`.

With the use of the shared memory mailer, additional code was added prior to the standard `'mail()'` code. This code first determined the node destination of the message. If it was the same node as the kernel itself, the system would use the shared message mailer instead of sending the message via a socket.

```
int mail_system::mail(generic_msg* msg, post_off* mbox)
{
#ifdef SHMEM
    // first check the shared memory mail queues
    //
    // get the node destination of the message
    int destination = msg->node_id.get_host_nbr();
    // if it is the same node we are on....
    if (destination == gl_host_id)
    {
        if (!this->lock())
            this->fill(mbox,FALSE);
        // use shared message mailer
        fax->send(msg);
    }
    else
    {
        // handle off node mail via sockets
#endif

        // make sure nothing is waiting on the input
        if (!this->lock())
        {
            // but don't block here
            this->fill(mbox,FALSE);
        }
        // write it out
        return(spwrite(msg));
    }
}
```

```

#ifdef SHMEM
    }    // end else
#endif
}

```

Figure 7-4. mail_system::mail() code

7.2 sock_mail.h

The 'sock_mail.h' file includes the definition of 'fax', which is a pointer to an object of type fast_path. The file sock_mail.h includes 'fast_path.h'.

7.3 fast_path.cc

The 'fast_path.cc' file contains the high-level code which constructs and accesses the shared memory message queues. The header file for these methods can be referenced in Figure 7-5. Also reference Section 7.5, which contains the msg_queue definitions.

```

class fast_path
{
    msg_queue**    mq;    // the pointer to an array of message
                        // queues that are allocated in the shared
                        // memory by the shared memory allocator

public:
    void send(generic_msg*); // send a message to the kernel
                            // on this host whose node_id is
                            // determined by the node field
                            // contained in the message

```

```

generic_msg* receive();    // get a message for me NOTE:
                          // this is a non-blocking call;
                          // it returns NULL if no
                          // messages are available

int msgs_avail();        // tell how many messages are
                          // available for me

fast_path();             // constructor which initializes
                          // the array of message queues

};

```

Figure 7-5. fast path header file

7.3.1 fast_path::fast_path()

The 'fast_path()' function is the constructor for the shared memory message queues. It constructs a message queue for each shared memory node and places a pointer to that queue in the appropriate entry in the mq (message queue) array.

```

fast_path::fast_path()
{
    mq = new msg_queue *[SH_MEM_NODES];
    for (int i = 0; i < SH_MEM_NODES; i++)
        mq[i] = new msg_queue(i);
}

```

Figure 7-6. fast_path::fast_path() code

7.3.2 fast_path::send()

The `send()` function is used to place a message into the appropriate shared message queue for the kernel it is destined for. If the kernel number in the message is higher than the number of shared memory nodes, an error results.

```
void fast_path::send(generic_msg* msg)
{
    int dest_node;
    dest_node = msg->node_id.get_node_nbr(); // get destination node
    if (dest_node > SH_MEM_NODES)
    {
        // probably should do a better job of error reporting here
        exit(3);
        return;
    }
    // put it in the queue. NOTE: queues are indexed 0..SH_MEM_NODES-1
    mq[dest_node - 1]->insert(msg);
}
```

Figure 7-7. `fast_path::send()` code

7.3.3 `fast_path::receive()`

The `receive()` method is used by a kernel to retrieve messages from its shared memory message queue. If there are no messages in the queue, the function returns `NULL`. Otherwise, it removes the first message in the queue and returns a pointer to that message. Note that the function removes the message from its global node id - 1. This is because the kernel nodes are numbered 1 .. `SH_MEM_NODES` while the shared memory message queues are numbered 0 .. `SH_MEM_NODES`.

```

generic_msg* fast_path::receive()
{
    generic_msg* gm;
    static int me = gl_node_id;

    if (msg_avail() == 0)
        return(NULL);    // NOTE: no blocking
    gm = mq[me-1]->remove();
    return(gm);
}

```

Figure 7-8. fast_path::receive() code

7.3.4 fast_path::msgs_avail()

This function returns the number of messages in the shared memory message queue for its kernel.

```

int fast_path::msgs_avail()
{
    int me = gl_node_id;
    return(mq[me-1]->msgs_in_queue());
}

```

Figure 7-9. fast_path::msgs_avail() code

7.4 msg_queue.cc

The msg_queue.cc file contains the low-level code which directly modifies the shared

memory queues. The header file for the message queue routines is contained in Figure 7-10. Also reference Section 7.4 which contains the fast_path definitions.

```
class msg_queue
{
    spin_lock q_lock;           // lock to prevent concurrent access to the
                                // messages stored in the queue
    int msg_count;              // number of messages currently in the queue

    generic_msg* head;          // pointer to head of the linked list of
                                // messages in the queue
    generic_msg* tail;          // and pointer to the tail of the list for
                                // easy insertion of new messages
    node my_id;                 // node id for all messages in this queue
public:

    void insert(generic_msg*); // insert a shared message into
                                // this queue
    generic_msg* remove();     // remove and return a message
                                // from this queue
    int msgs_in_queue();       // tell how many messages in queue
    msg_queue(int);            // construct this queue - the
                                // parameter tell which element
                                // in the array of queues you are
};
```

Figure 7-10. msg_queue header file

7.4.1 msg_queue::msg_queue()

The 'msg_queue()' function instantiates a shared memory message queue for the kernel specified in variable 'me'. The variable 'base' is set to the base address of the shared memory queue. The address for the shared memory queue for the specified kernel is then

set. The node is then marked with its host id and node id.

```
msg_queue::msg_queue(int me)
{
    shared_addr* base = SH_MEM_MGR->get_msg_q_base();
    this = (msg_queue*)((int)base + me * sizeof(msg_queue));
    my_id.set_node(gl_host_id,gl_node_id);
}
```

Figure 7-11. `msg_queue::msg_queue()` code

7.4.2 `msg_queue::insert()`

The `insert()` method is used to insert a message into a shared memory message queue. The first thing that is done is that the node id for the message is set to NULL. This is because the `node_id` parameter is used to point to the next message in the queue and since this message will be added to the end, its next pointer should be NULL.

The next step taken is to acquire a mutual exclusion lock on the shared memory space. This is done through the `acquire()` method in the `spin_lock` class which has been previously described.

If there are no messages in the queue, the head and tail pointers in the queue are set to point to the message being inserted. Otherwise, the address of the message is copied into the `node_id` variable of the last message in the queue. This was done because the compiler would not allow `node_id` to be set to the address of the message. The cause of this problem could not be determined. Since `'bcopy()'` only works on memory space, the compiler had no problem with it.

The 'last' pointer is then set to the new message, the count of messages in the queue is incremented, and the mutual exclusion lock is released.

```

void msg_queue::insert(generic_msg* msg)
{
    msg->node_id.set_node(); // be sure if doesn't point to
                            // anything as it will be at the end
                            // of the list
    q_lock.acquire();      // set mutual exclusion
    if (msg_count == 0)    // if queue is empty, add as only
    {
        head = msg;       // both first..
        tail = msg;       // ...and last message
    }
    else
    {
        // else set tail's next ptr to msg; tail->next = msg
        bcopy((char*)&msg,(char*)&(tail->node_id),4);
        tail = msg;
    }
    msg_count++;          // show one more message in queue
    q_lock.release();     // release mutual exclusion
}

```

Figure 7-12. `msg_queue::insert()` code

7.4.3 `msg_queue::remove()`

The `remove()` method removes a message from the shared memory message queue for the kernel invoking the method. If there are no messages in the queue, the function returns `NULL_MSG`. If there are messages, a lock is then obtained on the shared memory.

The variable 'msg' is then set to pointer to the head pointer for the queue and the head pointer is then set to its next pointer through a `bcopy()`. The need for the `bcopy()` is the same as in `insert()`.

The message count for the queue is then decremented by one and the mutual exclusion lock for the shared memory space is released. The node id for the message is then set to the node id for the receiving kernel and the address of the message is returned.

```
generic_msg* msg_queue::remove()
{
    if (head == NULL_MSG)
        return(NULL_MSG);    // return nothing from an empty queue
    q_lock.acquire();        // else, set mutual exclusion
    generic_msg* msg = head; // get first message
    // head = head->next
    bcopy((char *)&(msg->node_id),(char*)&head,4);
    msg_count--;            // decrease count and
    q_lock.release();      // release mutual exclusion
    msg->node_id = my_id;
    return(msg);
}
```

Figure 7-13. `msg_queue::remove()` code

7.4.4 `msg_queue::msgs_in_queue()`

This function returns the number of messages in the shared memory queue.

```
int msg_queue::msgs_in_queue()
{
    return(msg_count);
}
```

Figure 7-14. `msg_queue::msgs_in_queue()` code

Chapter 8

Conclusion

The success of the port of ESP to the Sequent Symmetry was largely judged by the ability of the code to run test programs written by MCC. All test code ran successfully. K. Stuart Smith of MCC verified that the port had been successfully completed.

8.1 Test Programs

The programs used to test the port of ESP were the standard test programs used by MCC for the ESP system. All the tests compiled without problem. The tests were:

- [1] `tree_top`: this test constructed a tree of height 'x' and width 'y' and then deleted it. The time taken to complete this task was then printed.
- [2] `pod_test`: this program tested the ability of the kernel regarding I/O. Strings, integers, float and double precision variables were printed.
- [3] `block_test`: this program's purpose was to test the method locking code of the kernel. As described in the documentation for the program: "We create some (user specified) number of producers and consumers and scatter them about the configuration. A producer invokes the produce method which results in bumping tokens and unlocking the consume method. When a consumer invokes the consume method, the value of the token count is returned and decremented. If the count goes to zero, then the consume method is blocked, preventing other consumers from

getting in. If a consumer gets a value of zero, then a failure in the method lock is indicated."

- [4] `loop_test`: this test essentially creates another instance of the object with which it will communicate to measure messaging performance.
- [5] `crosscheck`: the purpose of this program is to verify the complete connectivity of all the nodes in the configuration. Each node sends a series of messages (of increasing length) to each node in the system. Replies should occur in under one second or an error is flagged.

8.2 Shared Memory Tests

Tests were also made concerning the shared memory implementation. The test programs were run in both normal and shared memory mode. It should be noted, for purposes of comparing the results, that the communication sockets were created using the `AF_UNIX` flag. Since only the `loop_test` and `tree_top` tests gave timed results, only they could be used in tests of the shared memory implementation. The results of the tests are shown in Table 8-1.

The `loop_test` program only tests response time between two kernels. Ten runs were with the following parameters and the results averaged. As can be seen by the following chart, the shared memory implementation achieves a four to five-fold increase over socket communications.

It should be noted however, that since this test is primarily concerned with communications between kernels, these results are optimal.

Table 8-1. Loop_Test test results

loop_test		
loops	2 kernels/shared memory	2 kernels/sockets
100	183 requests/sec	37 requests/sec
1000	187 requests/sec	37 requests/sec
10000	201 requests/sec	41 requests/sec

The tree_top test constructs and then deconstructs a tree of width 'x' and height 'y'. Ten tests were run and then averaged and the results placed in Table 8-2. As can be seen, although the shared memory implementation increases performance, it is not as significant as the loop_test results. However, there can be no doubt that the shared memory implementation outperforms standard socket communication between ESP kernels.

It should be noted, however, that as more kernels are employed, the effect of the shared memory implementation becomes more pronounced.

Table 8-2. Tree_Top test results

tree_top							
width	depth	2(shar)	2(sock)	3(shar)	3(sock)	4(shar)	4(sock)
2	2	0.6s	0.72s	0.52s	0.83s	0.71s	1.03s
5	2	1.53s	2.13s	1.49s	2.2s	1.31s	1.89s
5	7	5.9s	7.2s	5.12s	6.78s	4.2s	6.02s
9	9	21.83s	25.9s	16.77s	21.13s	12.61s	17.81s

8.3 Future Work

Additional enhancements may still be made to the ESP system. Some possibilities for future work are:

- [1] Add big/little endian support to ESP so that big and little endian machine may communicate with one another.
- [2] Add optional 80387 support to the Sequent ESP context switching code.
- [3] Rewrite the ff1() code to be entirely in assembly language.
- [4] Design a shared text system so that the text (instruction) portion of programs may be shared between multiple kernels on the same node.
- [5] Add #ifdef - #endif code to the #include files which don't have it so they they may be included multiple times without difficulty.

Appendix

app_table.cc

The following function definitions were added:

```
long clock();  
extern char *malloc(unsigned int);  
extern void free(char *);  
int printf(const char *,...);
```

class_input.cc

This module required the inclusion of '`<std.h>`'.

class_table.cc

The `class_table` module needed '`<std.h>`', '`<stdio.h>`' and '"`platform.h`"' to be included.

copy_block.cc

The required '`<std.h>`' to be included and the following function definitions were added:

```
extern char *malloc(unsigned int);
extern void free(char *);
int printf(const char *,...);
```

do_it.cc

The following function definitions were added:

```
extern char *malloc(unsigned int);
extern void free(char *);
int printf(const char *,...);
```

mail_system.h

Several minor changes were enacted in this file to allow it to compile. First, the '#include <netinet/in.h>' line was commented out because it was previously included through an imbedded '#include' chain. Second, the line 'extern int getpid(void)' was commented out because the compiler would not mark it as being external thus causing a compile error. No problems were observed from not declaring it.

kick.cc

Five changes were required to allow this file to compile. The first was to comment out the line including <stdio.h> because of conflicts with a previous include of <stdio.h>. The second change was to add a line including <sys/types.h>. The third modification was to comment out the line defining setrlimit() as 'extern int setrlimit(int,rlimit*);' because of conflict with a previous definition. The line 'rlimit coredumpsize = { 0,0 };' was changed to 'struct rlimit coredumpsize = { 0, 0 };' due to compile problems. Finally,

the code regarding exception handling was commented out since it was not required for the port.

node_kernel.cc

This file needs three changes in order to compile correctly. The first change was to add `<sys/types.h>` to the include list. The second was to remove the line `'extern void abort();'` due to conflicts with a previous definition. Finally, the lines:

```
#ifndef ESKERNEL
#define ESKERNEL
#endif
```

were added to the top of the file so that ESKERNEL-based includes would be permitted.

post_off.cc

This file required the addition of `<sys/types.h>` and `<netinet/in.h>` to the include list.

sys_configuration.cc

The file required `<stream.h>` to be added to the include list and `'extern int rewind(FILE*);'` to be removed because of a conflict with a previous definition.

References

- [1] Smith, K. Stuart and Smith II, Robert J., Experimental Systems Project at the Microelectronics and Computer Technology Corporation, March 1989, Microelectronics and Computer Technology Corporation, 1989.
- [2] Smith II, Robert J., Experimental Systems Kit Hardware, March 1989, Microelectronics and Computer Technology Corporation, 1989.
- [3] Caldwell, Guy S., A Programmable Message Interface for ES-Kit, March 1989, Microelectronics and Computer Technology Corporation, 1989.
- [4] Porter, Claudia, C++ in the ESP Environment, March 1989, Microelectronics and Computer Technology Corporation, 1989.
- [5] Leddy, William J. and Smith, K. Stuart, The Design of the Experimental Systems Kernel, March 1989, Microelectronics and Computer Technology Corporation, 1989.
- [6] Khanna, Arjun, The Use of Public Service Objects to Extend the ES-Kit Kernel, March 1989, Microelectronics and Computer Technology Corporation, 1989.
- [7] Chatterjee, Arunodaya, Performance Issues in the ES-Kit Kernel, March 1989, Microelectronics and Computer Technology Corporation, 1989.

- [8] Hung, Ying T., Porting the ESP Kernel onto the Symult s2010, March 1989, Microelectronics and Computer Technology Corporation, 1989.
- [9] Hahn, Douglas W., Debugging in the Distributed, Object-Oriented ESP Environment, March 1989, Microelectronics and Computer Technology Corporation, 1989.
- [10] Allen, Wayne P. and Saha, Avijit, Parallel Neural-Network Simulation Using Back Propagation for the ESP Environment, March 1989, Microelectronics and Computer Technology Corporation, 1989.
- [11] Stuart, K. Smith, ES-Kit Distributed Kernel: Principles of Operation, Microelectronics and Computer Technology Corporation, 1990.
- [12] Smith, K. Stuart, ES-Kit Software Development Environment: The ISSD and Clients, Microelectronics and Computer Technology Corporation, 1988.
- [13] Surma, Ed, Extensible Software Platform Software Development Environment Programmers Guide, Microelectronics and Computer Technology Corporation, 1990.
- [14] Stevens, Richard W., Unix Network Programming, Prentice-Hall, Englewood Cliffs, NJ, 1990.
- [15] Bershad, Brian N., et al., PRESTO: A System For Object-Oriented Parallel Programming, Department of Computer Science; University of Washington, WA, 1988.
- [16] Bershad, Brian N., et al., An Open Environment for Building Parallel Programming Systems, Department of Computer Science; University of Washington, WA, 1988.

VITA

Joel Patterson was born in Pensacola, Florida USA. He completed his secondary schooling at Great Bridge High School in Chesapeake, Virginia USA in 1985. He gained a Bachelor of Science in Computer Science from Old Dominion University in Norfolk, Virginia. Currently, he is pursuing his Master's Degree in Computer Science at Virginia Polytechnic and State University.