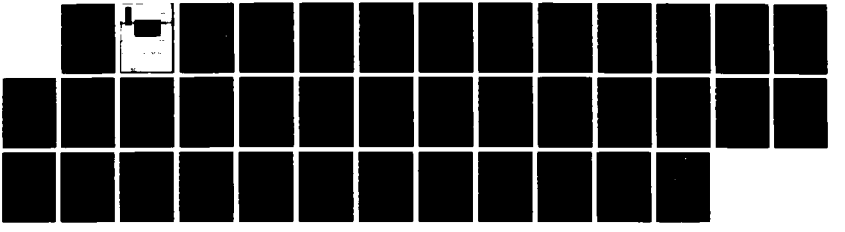
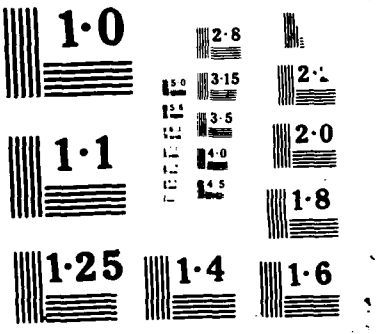
AD-A191 543 ELMWOOD - AN OBJECT-ORIENTED MULTIPROCESSOR OPERATING SYSTEM(U) ROCHESTER UNIV NY DEPT OF COMPUTER SCIENCE J M MELLOR-CRUMMEY ET AL. 70 ___ EP TR-226 1/1

UNCLASSIFIED DACA76-85-C-0001 F/G 12/6 NL




1·0
1·1
1·25
1·4
1·6
2·8
3·15
3·5
4·0
4·5
2·0
1·8

Elmwood – An Object-Oriented Multiprocessor Operating System

J.M. Mellor-Crummey, T.J. LeBlanc,
L.A. Crowl, N.M. Gafter, P.C. Dibble

Computer Science Department
University of Rochester
Rochester, New York 14627

TR 226
September 1987

Elmwood – An Object-Oriented Multiprocessor Operating System

J.M. Mellor-Crummey, T.J. LeBlanc,
L.A. Crowl, N.M. Gafter, P.C. Dibble

Computer Science Department
University of Rochester
Rochester, New York 14627

TR 226
September 1987

Abstract

Elmwood is an object-oriented, multiprocessor operating system designed and implemented as a group project at the University of Rochester. An Elmwood object, consisting of code and data, represents an instance of an abstract data type. Only the code associated with an object may access its data; interaction between objects is via remote procedure call. Access to an object requires that the caller provide an appropriate logical object name, which denotes a kernel-protected pair containing an object reference and a context value. Elmwood provides only basic mechanisms for protection and synchronization; the object itself supplies and interprets the context value, thereby implementing its own policies for protection and synchronization. We describe the Elmwood design, a multiprocessor implementation for the BBN Butterfly Parallel Processor, and our experiences in building a functionally-complete operating system as a group project in four months.

This work was supported in part by the National Science Foundation under grant number DCR-8320136 and DARPA/ETL under grant number [illegible]CA76-85-C-0001. The Xerox Corporation University Grants Program provided equipment used in the preparation of this paper.

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER TR 226	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE *(and Subtitle)* Elmwood - An Object-Oriented Multiprocessor Operating System		5. TYPE OF REPORT & PERIOD COVERED technical report 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) J.M. Mellor-Crummey, T.J. LeBlanc, L.A. Crowl, N.M. Gafter, P.C. Dibble		8. CONTRACT OR GRANT NUMBER(s) DACA76-85-C-0001
9. PERFORMING ORGANIZATION NAME AND ADDRESS Dept. of Computer Science 734 Computer Studies Bldg. University of Rochester, Rochester, NY 14627		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS DARPA / 1400 Wilson Blvd. Arlington, VA. 22209		12. REPORT DATE September 1987 13. NUMBER OF PAGES 33
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Engineering Topographics Laboratories		15. SECURITY CLASS. (of this report) unclassified 15a. DECLASSIFICATION DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

Distribution of this document is unlimited

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

operating systems design and implementation
object-oriented
remote procedure call
probabalistic protection
naming
Butterfly Parallel Processor

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

Elmwood is an object-oriented, multiprocessor operating system designed and implemented as a group project. An Elmwood object, consisting of code and data, represents an instance of an abstract data type. Only the code associated with an object may access its data; interaction between objects is via remote procedure call. Access to an object requires that the caller provide an appropriate logical object name, which denotes a kernel-protected pair containing an object reference and a context value. Elmwood provides only

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

20. ABSTRACT (Continued)

basic mechanisms for protection and synchronization; the object itself supplies and interprets the context value, thereby implementing its own policies for protection and synchronization. We describe the Elmwood design, a multiprocessor implementation for the BBN Butterfly Parallel Processor, and our experiences in building a functionally-complete operating system as a group project in four months.

1. Introduction

In the fall of 1986 a graduate seminar entitled *Operating Systems Construction* was held at the University of Rochester. The goal of the seminar was to gain practical experience in the design and implementation of an operating system. Over a period of four months we designed and implemented an object-oriented operating system called Elmwood. This paper describes the resulting system and summarizes our experiences.

Our primary goals were pedagogical. Since the group was not addressing the needs of a particular user community, the only serious limiting factor was time. Within a short time frame (four months), we hoped to design and implement a small, functionally-complete operating system.

For our hardware base, we needed a machine that was readily accessible and sufficiently well documented for our use. We chose the BBN Butterfly™ Parallel Processor for several reasons. The Butterfly was already the focus of much of our research; we were familiar with it and anxious to learn more about it. A simple operating system prototype would offer a vehicle for exploring implementation alternatives to Chrysalis [14], the standard Butterfly operating system. Documentation for the Butterfly was available and BBN personnel were willing to supply any additional required information. In addition, one of the Department's Butterflys, a small 3-processor system, could be dedicated to the project.

Early in the project we chose several design goals (or constraints). Each goal served to focus the design on specific issues we wished to explore. The goals and their rationales follow.

The design and implementation would be functionally complete. This goal helped to limit our natural tendency towards an ambitious design that we could not implement in the allotted time. To speed development, we agreed to use a readily available programming language, C, and not to attempt compiler modifications. To make our task as simple as possible, we would provide primitive, but powerful, mechanisms with which users could build complex functionality. As a result, *orthogonality of mechanism* became a design goal.

The operating system would support significant protection domains. Protection appears to have been a secondary consideration in the design of Chrysalis. There is little protection between user processes and the procedure call interface to the operating system offers little protection for the system. We wanted to increase protection between processes, particularly processes belonging to different users. Naturally, *the operating system would support multiple users.*

The operating system would be object-oriented. Objects would provide most of the traditional operating system functions. The kernel would implement only basic mechanisms for manipulating objects. Thus, *the operating system would have a small kernel.*

Finally, *the design would be consistent with a multiprocessor implementation*. Although we suspected that time constraints would preclude completion of a multiprocessor implementation during the term, we wanted an operating system design suitable for a shared-memory multiprocessor. In particular, *the operating system would allow processors to share memory, as in the BBN Uniform System* [3].

It is important to note that we chose these goals, somewhat arbitrarily, as interesting constraints on our design. Rather than having goals imposed by some particular architecture or user community, we imposed the goals on ourselves in order to define our areas of interest. Nonetheless, each goal is worthwhile in isolation, and the set of goals appeared to be consistent.

In the best of worlds, a complete specification of the user interface would have preceded any implementation. Given our time constraints, we felt compelled to design and implement in parallel. The group first met on September 4. We discussed project goals over a period of three weeks and reached agreement on September 25. During the same time period we considered several alternatives for the base machine before settling on the Butterfly. In parallel with the operating system design effort, we began implementing a software development environment for the Butterfly. (We assumed the basic functionality of the development environment to be independent of the particulars of the operating system design.) Low levels of the system (*e.g.*, the physical memory manager, the Butterfly address space manager) were designed and implemented while the overall system philosophy and design were debated. The first draft of the object facility was circulated on October 21. The basic development environment was running by November 6. The design of the user interface was completed November 26. On December 16, a single processor implementation of Elmwood running a simple shell successfully loaded and executed several user programs, thus completing the initial implementation phase. During the succeeding term, at a more relaxed pace, we discovered and corrected several bugs, extended the implementation to multiple processors, added a new loader and object registration facility, improved the shell, and did performance studies, which led to several optimizations. This document is an attempt to capture what was learned during those months of spirited debate and cooperative implementation.

In the next section we present the Elmwood user interface, consisting primarily of objects, logical object names, processes, and remote procedure calls. In subsequent sections, we describe the structure of the Elmwood implementation, summarize the performance of Elmwood and compare it to other systems implemented on the Butterfly, and categorize our experiences into a series of lessons, both technical and organizational.

2. Elmwood User Interface

Elmwood provides a number of orthogonal mechanisms to support conventional operating system functions. We chose orthogonality of mechanism as a design goal because we believe it contributes to both the simplicity and flexibility of the user interface. In Elmwood, *objects* encapsulate abstractions within protection domains, *logical object names* provide access to protection domains, *processes* represent asynchrony, *semaphores* and *condition variables* provide synchronization, and *exceptions* report error conditions.

The *object* is the basic unit of protection, encapsulation, and abstraction. An object is a passive entity that consists of an address space containing code and data. All data resides within some object and data local to an object cannot be modified from outside the object. Each object exports a set of operations, called *entry procedures*, that manipulate the object's data.

Invoking an entry procedure requires that the caller provide an appropriate *logical object name* (LON). A LON is similar to a capability for an object; it allows a process to invoke entry procedures defined by some object. The LON refers to a unique object and a user-interpreted *context value*, which provides a mechanism for implementing different protection policies. An object can create and invalidate LONs for itself with any desired context value. Only that object can access and interpret the context value.

A *process* is used to represent asynchronous activities. Unlike most systems, a process does not have its own protected address space or code. Instead, a process executes code associated with objects. Different processes, unless synchronized, execute independently; the underlying mechanism supporting multiprogramming, either time-slicing or multiprocessing, is not visible to the user except as it affects performance. A consequence of the orthogonality between processes and objects is that any number of processes may execute concurrently within an object. Any synchronization required for access to shared variables within an object must be specified in the object's implementation, using *semaphore* and *condition variable* synchronization primitives.

No activity takes place within an object except as the result of a call to an entry procedure. Although interactions between objects are realized using procedure calls, there is no guarantee that the two objects involved are on the same processor. Therefore, we refer to inter-object communication as a *remote procedure call* (RPC). RPC is the only mechanism for communication; processes communicate using a common object as an intermediary.

In the remainder of this section we describe the Elmwood user interface and compare it to related work. Although programmers can use this low-level interface directly, we expect most users to interact with standard libraries that implement higher-level abstractions.

2.1. Object Creation

Object creation and loading are combined in a single system call:

LON result = **load_object** (char *filename, int node)

The string argument is the name of the file containing the executable code for the object. The node number specifies the processor on which the system will load the code and execute the object's initialization routine. If it so chooses, the initialization code may create long-lived processes that continue to perform object-related duties for the duration of the object's existence. By convention, the return value from initialization is the *canonical logical object name* for the newly created object, which can be used later for object deletion.

For simplicity, we chose to combine loading executable files and object creation. This approach avoids the problem of managing executable code that has been loaded but is not ready to be executed (*e.g.*, process templates in Chrysalis).

2.2. Logical Object Names

A logical object name (LON) is an opaque handle for a kernel-protected pair: an *object reference*, used by the kernel to identify a unique object, and a *context value*, which the object itself supplies and interprets. LONs map many-to-one to objects; each LON is associated with a unique object, but many different LONs may refer to the same object. Different LONs for the same object are handles for different context values, enabling the object to distinguish clients.

Each LON is created on behalf of the object to which it refers. An object creates a new LON referring to itself with a new context value using:

LON result = **create_lon** (int context_value)

This is the only mechanism for valid LON creation. Since there is no way to create a LON for some other object, each object has control over the creation of its LONs. A LON may be passed as a parameter or returned as a result of RPC to another object. Once a LON passes from its creator to another object, the creator loses control over the distribution of the LON. However, the object retains ultimate control over the interpretation of context values.

When a LON is no longer needed, it can be invalidated using:

delete_lon (LON old_lon)

Only the object that creates a LON may invalidate it. Any process holding the LON will be unable to use it.

The first parameter to RPC is a LON, which the kernel uses to determine the object being invoked. The kernel only requires that the LON be valid. The object can use the context value to implement any additional protection policy appropriate for that object.

An object can determine the context value of the current client using:

int context_value = **get_current_context** ()

The context value associated with an object's initialization is the context value of the canonical LON, namely 0. The context value associated with a client is implicit in the LON supplied by the client. Since the kernel maintains the association between the LON used to invoke the object and the corresponding context value, there is no mechanism that allows a client to either forge a context value or change the context value associated with its invocation.

Although the user cannot modify kernel space directly, is is possible, with a very small probability of success, to forge a valid LON and thereby gain illegal access to an object. An alternative approach, in which the kernel maintains all LONs, would make LON forgery impossible, but would require considerably more kernel support and expense. Since we are primarily interested in catching errors, rather than malicious users, a probabilistic approach to protection is sufficient.

Each invocation of an object provides an opportunity for object-specific validation. Although the kernel only requires a valid object reference to implement RPC, the object, under programmer control, may refuse to execute any operation it chooses. In particular, the object can treat context values as capabilities; an operation is allowed only if the correct context value is supplied. The primary differences between LONs and capabilities are (a) rights associated with a capability are usually defined and enforced by the kernel; rights associated with a LON are defined and enforced by the associated object, (b) capability distribution can be controlled by the owner through the use of a "copy capability" right; LON distribution is beyond the control of an object once a LON is passed outside the object, and (c) revoking rights granted by a capability requires that the capability be revoked; rights associated with a LON are dynamically interpreted by the object during each call. Using LONs, coarse-grain protection for access to objects is implemented by the kernel; fine-grain protection for access to operations within an object is implemented by each object.

Management of LONs is similar to capability management. A typical scenario follows. Upon initialization, object X registers a LON with a *registration object* or *switchboard*. Processes that wish to invoke object X must first present an appropriate name or password to the registration object. (We assume that every object has a LON for the registration object.) Using the LON returned by the registration object, a process invokes object X. During this first invocation, object X examines the context value of the invocation to determine the identity of the caller. Any caller

presenting the registered LON is recognized by the object as a new client. Object X then creates another LON with a context value that identifies uniquely the new client. This context value is interpreted only by the object itself, therefore it can be a simple integer tag or a pointer to a client-specific data structure. Object X returns this new LON to the caller, which uses it in future calls. If successive calls result in successive levels of validation, each returns a new LON, corresponding to the new level of validation. At any time, the object can refuse service to a process that does not provide a LON with a proper context value.

Levels of validation or service could also be represented by different objects, rather than different contexts within the same object. Using this approach, a file system could create a remote object to manage an open file whose data resides on another processor. The initial request to the file system would return a LON for the remote object, to which all subsequent requests would be addressed. Since LONs are opaque, a client cannot distinguish between a service provided directly by a central file system and a service provided by an auxiliary file server. As a consequence, a service can have a distributed implementation of which the client is unaware.

2.3. Remote Procedure Call

A remote procedure call is a request to an object to execute some operation. Processes calling an object may proceed concurrently, unless explicitly synchronized by the object. Each call indicates the object being invoked, an entry procedure within the object, and the procedure's parameters.

int result = **rpc** (LON object, int entry, *parameters*)

All parameters to **rpc** are passed by value. A procedure terminates execution and returns control to the caller using:

rpc_return (int value)

The specified value is returned to the caller.

To avoid the need for compiler support, Elmwood requires the object programmer to provide a dispatcher for an object's entries, including initialization. This dispatcher must be the first procedure in the load image and should have the following form:

```
#define CANONICAL 0

void main (entry, parameters)
int entry;
parameter type definition;
{
   static int uninitialized = TRUE;
   if (uninitialized) {
        uninitialized = FALSE;
        initialize();
        rpc_return(create_lon(CANONICAL));
   } /* end if */
   switch (entry) {
        case 0  : rpc_return(entry0(parameters));
        case 1  : rpc_return(entry1(parameters));
        default : /* error */
   } /* end switch */
} /* end main */
```

2.4. Exception Handling

Elmwood provides a simple exception handling mechanism. Exceptions are generated using **throw**; they can be caught and handled using **catch**. Since we did not want to modify the compiler, both **catch** and **throw** are procedure calls, used as follows:

```
if (!(throwVal = catch())) {
   /* catch block code for normal execution */
   /* may include the statement throw(i) */
   end_catch();
} else {
   /* exception handler code, uses variable throwVal */
} /* end if */
```

This idiom defines a block of code for an exception handler to execute when a **throw** occurs. If **throw** is called within the catch block, or within any procedure called by the catch block, then flow of control transfers to the exception handler. The variable *throwVal* may be used in the exception handler, and has the value of the throw code passed as a parameter to **throw**. If there are nested catch blocks, then a **throw** will transfer control to the exception handler corresponding to the most recently activated catch block.

System calls in Elmwood report error conditions by generating exceptions. Exceptional conditions can also be caused by asynchronous events, such as the deletion of an object on a process's call chain. To avoid the complexity of handling externally generated exceptions that may occur at arbitrary points in a process's execution, Elmwood provides a system call that checks for a pending asynchronous exception.

throw_check ()

This routine has no effect unless an asynchronous exception is pending, in which case the exception is raised and then caught by the most recently activated catch block.

2.5. Object Deletion

The design of a consistent set of semantics for object deletion must satisfy somewhat contradictory requirements. An object should have control over its own destiny, but it must be possible to delete a rogue object that refuses to exercise responsible control. It should not be possible for one object to delete another arbitrarily, nor to require object deletion at inopportune moments. An object should be allowed to wait to delete itself until no processes are executing within it, or until it reaches some internally consistent state. On the other hand, repeated failed attempts to enlist the cooperation of an object must ultimately lead to forced deletion. In such cases, object deletion may affect processes executing within the object.

These requirements prompted us to provide two distinct object deletion mechanisms: **soft_delete** destroys an object when all activity within the object has ceased; **hard_delete** destroys an object immediately. A **soft_delete** only applies to the current object; **hard_delete** requires the canonical LON of the object to be deleted.

soft_delete ()
hard_delete (LON object_lon)

We expect **soft_delete** to be used in most cases. It marks the current object for deletion, but processes within the object continue to execute. No more processes are allowed to invoke the object's operations; any attempted invocation causes an INVALID_LON exception in the caller. When the last process leaves the object, the object is deleted and its storage is freed. Since **soft_delete** can be used only by the object itself, others wishing to delete the object must use RPC to request that the object delete itself. The delay between the call to **soft_delete** and the return from the RPC gives the object time to release resources and clean up its internal state.

The **hard_delete** routine is used as a last resort to destroy rogue objects that refuse to destroy themselves. A **hard_delete** destroys the object immediately, removing it from the call chain of any processes that may be executing in or through it. A process that is executing the code of the object being deleted will receive an OBJECT_DELETED exception, rather than a return value from the deleted object. A process that has the object on its call chain, but is executing in another object, continues to execute normally. When the process tries to return to the deleted object, an OBJECT_DELETED exception is raised back in the caller of the deleted object. If a process needs

to know that some object on its call chain has been deleted in order to recover allocated resources, the **throw_check** routine can be used to check for an OBJECT_DELETED exception before returning.

We chose not to implement an object hierarchy; therefore, object deletion is not transitive. By convention, we expect object initialization to return a canonical LON to be used by **hard_delete**. We also expect each object to implement a transitive form of **soft_delete** whenever appropriate. Due to the possibility of rogue objects that do not adhere to these conventions, we expect to support the notion of a system administrator and an all-powerful *root* LON. The system administrator can use the root LON to examine and modify kernel state. In particular, the root LON would allow the administrator to discover the canonical LON of a rogue object and to delete it.

2.6. System Objects

To make the Elmwood user interface as simple as possible, we chose to implement the primitives for process management and synchronization as object operations. These objects are implemented by the kernel. They can be invoked like any other object, assuming an appropriate LON is known.

Processes are the basic mechanism for expressing concurrency and asynchrony. A process is the abstraction of a processor, and *may execute concurrently with other processes*. Concurrency is simulated on a single processor by time-slicing. In the Butterfly implementation, true concurrency is possible using multiple processors.

A special form of RPC, called **fork**, creates an asynchronous process to execute an entry procedure in a specified object and then terminate.

LON proc = **fork** (LON object, int entry, *parameters*)

Fork returns a LON for the process. The process LON is valid only as long as the process continues to execute; when the process returns from the entry operation, the LON becomes invalid.

Process management operations are defined as entry procedures associated with the process LON. These include operations to abort the process and to generate exceptions within it. Additional operations to set and interpret process priorities could be added easily.

The only way to forcibly terminate a process is to use process abort.

rpc (LON process, int PROCESS_ABORT)

A **hard_delete** isn't allowed because the LON returned to the user as a result of **fork** is not the canonical LON for the process object. A process abort is analogous to a **soft_delete** on an object.

After a process abort, the next **throw_check** executed by the process generates a PROCESS_ABORTED exception. It is a special case of the more general call:

rpc (LON process, int PROCESS_THROW, int code)

This generates an exception with the given code the next time the process invokes **throw_check**. So long as objects use **throw_check** whenever a consistent state is reached, a process abort will quickly and reliably destroy a process. An object containing code that fails to adhere to this convention may trap a process in an infinite loop, which can be terminated only by deleting the object.

Elmwood also provides system objects that implement semaphores and condition variables [10] for process synchronization. System calls create instances of each synchronization primitive; object operations perform synchronization and delete the instances.

LON sem = **semaphore** (int count)
LON cond = **condition** (LON sem)

The **semaphore** system call creates a new counting semaphore with the given initial value, and returns a LON to represent it. Similarly, **condition** creates a condition variable to be attached to a semaphore.

There are three operations on semaphores. SEM_P and SEM_V are the counting semaphore operations P and V. SEM_DELETE deletes the semaphore, which generates a SEMAPHORE_DELETED exception in any process waiting on the semaphore. All attached condition variables are deleted when the semaphore is deleted.

rpc (LON sem, int SEM_P)
rpc (LON sem, int SEM_V)
rpc (LON sem, int SEM_DELETE)

There are five operations on condition variables. COND_WAIT waits on the condition variable, implicitly performing a SEM_V on the semaphore to which it is attached. COND_AWAITED returns the number of processes waiting on the condition. In general, the result is only a hint, since it might be wrong by the time the calling process inspects the value. COND_NOTIFY causes the highest priority process awaiting the condition to continue, implicitly performing a SEM_P on the semaphore. The process that executes COND_NOTIFY implicitly performs a SEM_V and SEM_P, ensuring the awakened process the opportunity to execute. COND_BROADCAST repeatedly performs a COND_NOTIFY until there are no processes waiting on the condition. Finally, COND_DELETE deletes the condition variable, readies all processes waiting for the condition, and generates a CONDITION_DELETED exception in each

```
rpc (LON cond, int COND_WAIT)
int waiters = rpc (LON cond, int COND_AWAITED)
rpc (LON cond, int COND_NOTIFY)
rpc (LON cond, int COND_BROADCAST)
rpc (LON cond, int COND_DELETE)
```

2.7. Relationship to Previous Work

The primary motivation for multiprocessor systems is to speed up user programs by enabling different parts of an application to execute in parallel. To do this, the application must have a mechanism to share information between parts of an application, and to protect the information from misuse. Protection requires a tradeoff between selectivity and efficiency; more selective protection mechanisms are generally less efficient. Since we feel that the major novelty of Elmwood is its flexible protection scheme, this section concentrates on systems with related approaches to protection.

The Multics system uses *rings* to implement protection [17]. When a process attempts to access a segment, the system checks that the ring containing the executing process is enclosed by the ring associated with the segment. Processes move between rings via controlled access to procedures in other rings. The small number of available protection rings limits support for selective protection policies, either between programs or within a single program.

Hydra uses capabilities to implement protection [7]. Capabilities combine the notion of reference with protection. When presented with a capability to a memory segment, the system need only examine the capability to ensure proper access. In Hydra, each activation of a procedure creates a local name space, which contains all capabilities known to an activation of a procedure. Hydra provides capability promotion to allow a procedure implementing an abstract data type to operate on a memory object to which the caller does not have access. The Hydra protection mechanism is cumbersome because protection of code and data is separate. In addition, the management of a local name space for each procedure introduces substantial overhead.

Eden also uses capabilities to implement protection [1], but the mechanism is considerably simpler. Eden objects contain their code, so there is no need to associate the proper code with a data object. Nonetheless, both Hydra and Eden maintain in the kernel the list of capabilities that an object possesses; this introduces substantial overhead. Programs must explicitly notify the system whenever capabilities are passed between objects. In contrast, the Elmwood kernel is not involved in the distribution of LONs. We rely instead on probabilistic protection for LONs. Therefore, LONs can be passed as arguments or return values without the knowledge of the kernel.

Both Hydra and Eden associate protection with individual object invocations. However, objects often interact using a series of invocations, called a *conversation*. Link-based systems, such as Demos [4], Charlotte [2], and Lynx [18] inherently support conversations because communication uses virtual circuits. Neither Eden nor Hydra explicitly support conversations. Users must pass explicit context values as parameters to each invocation. The inability to represent conversations with capabilities has been noted as a potential problem with Eden [1]. Elmwood supports conversations with LONs. A LON establishes a context defined by the object to which it refers. An object initiates a conversation by creating a new LON with the context for the conversation. Subsequent invocations that are part of the same conversation use the new LON.

3. Implementation Description

Elmwood consists of a series of layers, starting with the Butterfly architecture as a base. The first software layer contains the program development environment, which enables us to enter, execute, and debug programs. The next layer contains mechanisms that support concurrent programming. Subsequent layers implement the Elmwood kernel itself. These layers manage process-like threads, physical memory allocation, address spaces, objects, interprocessor communication, and RPC. All of these layers share kernel space; there is no protection between kernel layers. This section describes the implementation of each of these layers in turn.

3.1. Butterfly Architecture Description

The BBN Butterfly Parallel Processor consists of 8 MHz MC68000 processors connected by an O(n log n) switching network. Each switch node in the switching network is a 4-input, 4-output crossbar switch with a bandwidth of 32 megabits/sec. The Processor Node Controller (PNC), a 2901-based, bit-sliced co-processor, interprets every memory reference issued by the 68000 and communicates with other nodes across the switching network. All the memory in the system resides on individual nodes, but any processor can address any memory through the switch.

The Butterfly implements segmented virtual memory. Each virtual address contains an 8-bit segment number and a 16-bit offset within the segment. A process can have at most 256 segments in its address space, each of which can be up to 64K bytes in size. Each segment is represented by a segment descriptor, which defines the address, extent, and protection of the associated memory. A 4-bit size field in the descriptor offers 16 different segment sizes, ranging from 256 bytes to 64K bytes. To access a segment, the corresponding descriptor must be resident in one of the hardware SARs (Segment Attribute Registers) used during address translation.

A process's virtual address space is represented by an ASAR (Address Space Attribute Register), which defines the address and extent of a group of SARs corresponding to the segments in the

address space. There are 512 32-bit SARs and one (active) 16-bit ASAR per processor. Sixteen SARs, at offsets F8-FF and 1F8-1FF in the SAR bank, are used to access kernel code and data. The address translation scheme provides access to eight of these SARs at a time, regardless of the value of the ASAR. By convention, the sixteen kernel SARs are maintained as two copies of eight SARs.

Two 9600 baud serial lines are used to communicate with the Butterfly. One is a terminal line, the other is a program download line connected to a Vax 11/750 running Unix 4.3BSD. Both lines are connected directly to a single node in the Butterfly, called the *king node*. These are the only devices supported by Elmwood, although the download line can be used to access the Unix file system on the Vax.

3.2. Software Development Environment

Before implementing the operating system, we needed a simple program development environment that would enable us to load a program and execute it, while providing mechanisms to examine the machine state at a level suitable for debugging. Towards this end, the Butterfly contains both a ROM and a RAM implementation of a monitor program called the Ultra Stupid Debugger (USD). USD has an interactive command mode used to reset the processors, load an operating system, and begin execution. It also provides low-level services such as device drivers and exception vectors.

There were two reasons why we chose not to use USD itself. First, USD makes some assumptions about the interface with the operating system that we did not plan to support. In particular, USD generates Chrysalis exceptions in response to hardware exceptions, such as bus errors and division by zero. We wanted the programming environment to provide for explicit management of exception vectors. Second, USD was never intended to provide full debugging support, such as breakpoints and trace mode. Lacking a hardware monitor, we felt additional debugging support would be crucial if we were to make any progress.

BackHo, the Elmwood program development environment, was created by deleting functions from USD that would not be used by Elmwood and adding new debugging facilities. The first version of BackHo was implemented by removing all Chrysalis dependencies from USD. Both BackHo and Elmwood are initially loaded by executing the master portion of a download protocol on the host Vax. The slave portion is executed by ROM USD. The master causes the slave to initialize the system SARs and load the operating system. When the download protocol is finished, control returns to ROM USD, from which both BackHo and Elmwood can be booted.

Like USD, BackHo implements interrupt handlers for both the download line and the terminal line. These handlers communicate with the rest of the system using dual queues, buffered communication primitives implemented with microcode support. BackHo provides Elmwood with a low-level interface for manipulating these queues.

For debugging, BackHo implements instruction stepping, breakpoints, disassembly, memory dumps, and exception handling. Whenever an uncaught exception occurs, BackHo saves the state of the processor in a control block, displays information describing the exception, and enters command mode. If an exception takes place while BackHo is processing an exception, the control block is overwritten and BackHo passes control to ROM USD.

BackHo is written in MC68000 assembler and requires 8K bytes of storage, excluding the disassembler.

3.3. Programming Language Support

The programming language support (PLS) library provides abstractions and services commonly associated with concurrent programming languages. The PLS library provides facilities that require an assembly language implementation, including coroutines, exception handling, and the system call interface to the kernel. With the exception of a few routines hand-coded in assembly language for efficiency, all other Elmwood facilities are implemented entirely in C.

The fundamental abstraction provided by the PLS library is the *coroutine*. A coroutine is a locus of control consisting of a program counter, a stack, a stack pointer, and the general-purpose registers. Coroutine creation requires a stack, a procedure **main**, and an initial argument for **main**. The first transfer to the coroutine calls **main** with the given argument. The stack holds activation records for the coroutine.

Control passes between coroutines either through an explicit transfer or through an interrupt The PLS library provides operations that implement each of these transfers. The explicit transfer operation saves the state of the currently executing coroutine and restores the state of the specified coroutine, which then begins execution. Interrupts cause an implicit transfer of control between the currently executing coroutine and an established interrupt handler. The **iotransfer** routine (inspired by the Modula-2 procedure of the same name [20]) establishes an interrupt handler. It takes two parameters: an interrupt vector and another coroutine. **Iotransfer** establishes the caller as the interrupt handler for the specified interrupt vector. Control then transfers to the specified coroutine. Any future interrupt at the specified vector causes an implicit transfer of control back to the handler, passing a handle for the interrupted coroutine. Using this mechanism, coroutines can be associated with asynchronous devices attached to a processor. In particular, a clock

coroutine handles timer interrupts in Elmwood.

When a coroutine is created, the system provides a default interrupt mask enabling all interrupts. Each coroutine may modify its own interrupt mask using routines supplied by the PLS library. For efficiency, the library includes two simple routines for enabling and disabling clock interrupts.

The PLS library also provides **catch** and **throw** exception handling primitives for use by the kernel and user objects. The **catch** routine records the state of the machine (*e.g.*, registers, program counter, stack pointer) in a *catch record*, pushes it onto a stack of catch records associated with the current coroutine, and returns zero. A **throw** pops a catch record from the stack and restores the state saved in it. Following a **throw**, execution continues at the last call to **catch**, which returns the exception code instead of zero. The system translates hardware exceptions, such as divide-by-zero, into **throw** operations. The **end_catch** routine terminates the scope of the **catch** by popping a catch record off the stack.

Finally, the PLS library implements control transfers between kernel mode and user mode. By default, all coroutines begin execution in kernel mode. Invoking the kernel routine **call_user** causes execution to enter the more restrictive user mode. Control returns to the system when the user executes **rpc_return**, which issues a **trap** instruction. From user mode, **trap** is used to request a kernel service. The PLS trap handler translates traps into synchronous procedure calls to operating system services. During initialization, Elmwood binds the various kernel services to entry point indices corresponding to different **trap** parameters. Generally, higher levels of the kernel implement these services, so invoking a service from the trap handler is actually an *upcall* [6]. By providing a linkage mechanism between users and services in the trap handler, kernel services can be implemented entirely in a high-level language.

3.4. Thread Manager

Coroutines are a low-level mechanism based on the explicit transfer of control between different machine states. *Threads* are a higher-level mechanism, implemented using coroutines, that provide implicit transfer of control between different client states. Coroutine transfers must explicitly identify the next coroutine to execute; thread transfers are handled implicitly by a thread scheduler which chooses the next thread to execute. The state of a coroutine consists of a program counter, stack pointer, stack, and machine registers; the state of a thread consists of the machine state and any other state information needed by the client (*e.g.*, an address space).

An important feature that distinguishes threads from coroutines is that threads may have associated *contexts*. A context consists of a 32-bit value and a pair of procedures that use this value

to save and restore state associated with the thread. The 32-bit value is usually a pointer to some data structure that the save and restore procedures manipulate. Higher layers in the kernel generally implement these procedures, so calls to save and restore thread state are upcalls. Thread contexts provide the mechanism to save and restore floating point registers and address spaces, among other things.

Each time a thread begins a time slice, the corresponding restore procedure is called to restore the thread's context. Similarly, when a time slice ends, the corresponding save routine is called to save the thread's context. In Elmwood, the context mechanism is used to keep track of the object and address space in which the running thread executes.

Most of the operations available for coroutines have analogs for threads. Thread creation creates a coroutine and associates a priority with it. A scheduler coroutine that responds to clock interrupts can use this priority to implement time-slicing between threads. Thread destruction requires the cooperation of the scheduler thread to deallocate the thread's resources, since a thread cannot deallocate its own resources while it is executing.

As with coroutines, a transfer of control between threads may occur implicitly or explicitly. In the usual case, thread transfers occur implicitly as a result of time-slicing provided by a scheduler coroutine. In addition, **yield** and **block** routines provide explicit transfer between threads without scheduler intervention. Both **yield** and **block** suspend execution of the current thread and start execution of the next thread on the ready queue. However, **yield** enters the current thread on the ready queue, whereas **block** does not.

A pair of procedures, **begin_exclusive** and **end_exclusive**, enable atomic operations within threads. The **begin_exclusive** operation sets a flag variable within the thread scheduler to begin an atomic operation. Further execution of that thread is atomic with respect to other threads, until a call to **end_exclusive**. The scheduler never transfers between threads if the flag is set. These procedures are an alternative to the **clock_disable** operation which masks clock interrupts, thereby providing atomicity for the coroutines that implement threads. More general synchronization between threads is based on semaphore and condition variables, implemented using **begin_exclusive** and **end_exclusive**.

3.5. Physical Memory Manager

The physical memory manager provides operations for manager initialization, memory exclusion, memory allocation, and memory deallocation. The memory exclusion operation allows the operating system to exclude regions of bad memory from its available memory. The allocation operation takes a processor node number and the number of bytes to allocate as arguments, and

returns a handle for a segment of physical memory that resides on the specified processor. The deallocation operation takes a physical memory handle as an argument and frees the corresponding physical memory.

The physical memory allocator is an out-of-band version of the standard power-of-two buddy allocator [9]. Physical memory on the Butterfly may not be allocated across 64K byte physical address boundaries. We chose a buddy allocator because it naturally prevents allocation across such boundaries, while other allocation mechanisms must explicitly enforce this restriction.

The standard buddy allocator uses in-band allocation; a linked list of available blocks is embedded in the memory to be allocated. In addition, all blocks, both allocated and unallocated, have a tag field specifying whether or not the block is allocated. We rejected this approach for two reasons. First, in-band allocation assumes that the process doing the allocation can access the data structure embedded in available memory. Although this is possible on the Butterfly, it would require extensive manipulation of SARs to access the data structure. Second, the tag bit embedded within allocated memory causes the system to consume memory at twice the rate of requests, if requests are for a block size that is a power of two. The problem is exacerbated on the Butterfly since linear address spaces must be constructed from 64K byte segments.

To solve these problems, we moved the memory allocation data structure out of available memory and into a kernel data structure. Rather than use a free list for all available memory, and thereby use a substantial amount of kernel memory, we implemented a memory map. Each entry in the memory map has a tag bit, specifying whether or not the corresponding block is allocated, and a size field. The size field specifies the number of successive entries described by the current entry. Large blocks of memory that have been allocated or coalesced together are summarized in one entry.

Each processor node in our Butterfly has 1MB of memory, organized into 256-byte pages. If each physical page could be allocated individually, we would require a memory map with 4K entries. Since we did not expect to support many small objects, our physical memory manager allocates memory in blocks of 1K bytes. Thus, in the worst case, our memory map has 1K entries, requiring 1KB.

Finding a free block within the memory map involves searching the memory map for a block of the appropriate size. Since a free block of size N will appear at indices 0, N, 2N, *etc.*, the search need only examine entries at those indices. In addition, because each entry contains the allocation size, blocks of size larger than N will cause the search to skip entries. If a block of size N is found, it is allocated. Otherwise, we split the first block of size 2N into two buddies, and allocate one of them.

3.6. Address Space Manager

In Elmwood, address spaces are associated with objects. Each Elmwood object exists in its own address space; the address space implements a referencing environment for an object's operations.

An address space on the Butterfly consists of a contiguous block of variable length segments. Each segment can refer to memory on the local processor node, memory on a remote processor node, a special area for memory-mapped control of the PNC, or memory-mapped I/O. In addition to its type and length, segment descriptors contain protection attributes. The memory architecture of the Butterfly requires buddy allocation of address spaces; each address space can consist of 2^i segments, where $3 \leq i \leq 8$.

For an address space to be active on the Butterfly, the block of segment descriptors describing the address space must be resident in the SARs used by the memory management hardware for interpretation of virtual addresses. Also, the ASAR register must be set to indicate which contiguous block of SARs is currently in use for address translation and the number of segments in that block. Any attempt to reference a virtual address in a segment outside the current block will cause a bus error.

Address space management in Elmwood is divided into three layers. In the first layer are routines for manipulating segment descriptors (hereafter referred to as *SAR images*), which are loaded into the segment attribute registers for address translation. The second layer manages contiguous blocks of SAR images representing a referencing environment. The third layer manages Elmwood address spaces.

The routines provided by the first layer construct SAR images from their constituent parts, convert from a physical memory handle provided by the physical memory manager into a SAR image and vice versa, alter the protection attributes of an existing SAR image, validate the constituent parts of an existing SAR image, validate a particular access using a SAR, and translate virtual addresses into physical addresses. The latter capabilities are software simulations of the address translation hardware that Elmwood uses to diagnose bus errors and generate unique system-wide addresses.

The second layer manages contiguous blocks of SAR images, or *SARsets*. This layer creates and destroys SARsets. The creation routine takes the number of segments needed for an address space and returns the smallest SARset permissible by the hardware containing at least that many segments. Additional functions are provided to support reading and writing of SAR images by their relative position in the SARset.

SARs are a critical resource on the Butterfly, and their use must be managed carefully. It is not possible to have the SARsets for all existing objects resident in the SARs at all times. Elmwood uses the SARs as a cache; a consistent shadow copy is maintained in memory at all times. By maintaining a shadow copy, SARsets can be overwritten in the SARs without having to pay the expense of copying them out. Before invoking an operation on an object, the object's SARset must be resident in the SARs. An activation routine in the SARset layer guarantees that the specified SARset is resident in the SARs and is pointed to by the ASAR register. Since it is not possible to maintain all SARsets in the SARs, a policy must exist to choose which SARsets should be overwritten by an incoming SARset. The policy implemented in Elmwood uses least displacement as a criteria. When a SARset is moved into the SARs, it is positioned so that the SARsets it overlaps are of minimum total size. This is an attempt to minimize the number of SAR images that will need to be swapped into the SARs during future activations.

The third layer implements address spaces for Elmwood objects. The Butterfly reserves segment zero in each object to refer to the SARs and PNC microcode functions. Segment one is reserved for the user's stack segment. Successive segments contain code, static data, and dynamic data.

Functions are provided to create and destroy address spaces, expand the dynamic data area in an address space (up to an initial maximum length specified at creation), and activate an address space. Address space activation takes two parameters: an address space and a SAR image for a stack segment. The activation code insures that the SARset corresponding to the address space is active in the SARs and maps the SAR image of the specified stack segment into segment one. If a context switch takes place to a process within the same object, only segment one is swapped. Otherwise, the new address space must be activated, which may involve swapping its SARset into the physical SARs.

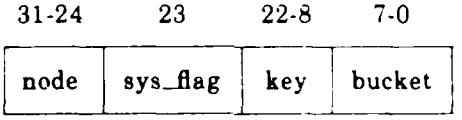
3.7. Object Manager

The object manager implements the basic operations on objects and LONs, including creation and deletion. Currently, the only way to create user objects is through the **load_object** system call; however, the underlying mechanism is more general. Object creation takes arguments that specify the object's size (length of code, static data, and dynamic data segments) and creates an address space in which the object will reside. A procedure supplied to the object creation routine initializes the address space. For now, that procedure is the loader protocol, since object creation is a side-effect of downloading an object. In the future, we could easily add a call to some other procedure in an object template manager, thereby separating object creation from downloading.

In addition to managing the object's address space, the object manager maintains the object's entry point, some flags for managing object deletion, and a reference count of active invocations. The reference count is used by **soft_delete** for object deletion. During RPC, the object manager validates the presented LON, increments the reference count of the object referred to by the LON, and returns the address space of the object, as well as the virtual address of the object entry point. If the validation fails, then the object manager returns an error code to the RPC facility.

When an object invocation terminates, the object's reference count is decremented. If the reference count becomes zero, the exit routine checks the object's flags to see if a **soft_delete** is pending. If so, the object manager deletes the object. All storage occupied by the object and its associated data structures is freed.

In addition to implementing objects, the object manager also manages LONs. LONs are opaque handles that are exported from the object manager. Internally, each LON consists of four fields:



The *node* field indicates the processor node on which the named object resides. The *sys_flag* bit is set if the object is a system object. The *key* and *bucket* fields are used during lookup in a kernel-protected hash table that stores all LONs for objects on a given processor. The *bucket* field is used as a hash code and the *key* value is used to disambiguate collisions in the hash chain.

The kernel protected data structure associated with a LON contains a pointer to the object descriptor (a physical object name or PON) to which the LON belongs and a 32-bit context value. The relationship between LONs and PONs may be many to one. The **create_lon** system call adds new entries to the LON hash table corresponding to new (PON, context value) pairs. **Delete_lon** removes an entry from the LON hash table and frees the kernel-protected pair associated with the LON. In addition, when an object is deleted, the LON hash table is searched and all entries that refer to the deleted object are removed.

The object manager uses the hash table to validate LONs. Elmwood makes no effort to track down LONs that no longer refer to valid entries in the LON hash table; validation fails when a LON is not found in the hash table or it refers to an object already marked for deletion.

3.8. Interprocessor Communication

The Butterfly provides microcode primitives for atomic operations, but most of these primitives support Chrysalis-specific functions. The special-purpose nature of these primitives limits

their usefulness for building general concurrent data structures. In particular, there is no atomic compare-and-swap operation and no atomic operations on longwords (including pointers). As a result, Elmwood data structures were not designed for concurrent access. Kernel operations, such as memory allocation, cannot operate directly on a remote node's data structures. Instead, Elmwood uses message passing to implement kernel operations on remote processor nodes. The implementation of message passing uses the Butterfly's ability to share memory between processors.

To perform a kernel operation on a remote node, a block of storage is allocated to contain a standard message header followed by operation-specific parameters. The local kernel translates the virtual address of the message block into a physical address (which is unique for the entire multiprocessor) and atomically enqueues the physical address of the message on the work queue of the remote node using a microcode operation. The local kernel then issues an interrupt on the remote node using a special PNC function and blocks awaiting completion of the operation by the remote node.

The Elmwood kernel executing on the remote node fields the interrupt and removes an entry from its work queue. Each entry contains the processor number of the requesting node, so the remote kernel can recognize the entry as a remote request. A local thread is created to execute as a proxy on behalf of the caller. The proxy maps in the memory at the physical address specified in the work queue entry and copies the message into its local memory using information in the header fields. Each message header contains an indication of the appropriate procedure stub for carrying out the remote request. The remote stubs retrieve arguments from the message and call the corresponding kernel operation. The stubs are also responsible for putting return values into slots of the message to be propagated back to the caller.

When a proxy completes the operation, the return values are copied into the message block back on the caller's node. An acknowledgement is constructed by concatenating the processor number of the caller's node and the virtual address of the caller's thread. The acknowledgement is enqueued on the caller's processor node and an interrupt is issued to indicate that a reply has arrived. The proxy is destroyed and its resources freed.

Back on the caller's node, the interrupt handler is invoked to retrieve the acknowledgement. The interrupt handler recognizes its own processor number in the value retrieved, and schedules the caller. When the caller is activated by the scheduler, it examines the values returned in its original message block and takes the appropriate action, thereby completing the remote system call.

3.9. Remote Procedure Call

Remote procedure call is the only mechanism in Elmwood for processes to access objects. When a process executes an RPC it crosses a protection boundary, represented by the LON used in the call, even if the call is to an entry in the same object. For this reason, each RPC results in the creation of a new thread. Every such thread contains both a kernel and user stack.

RPC is initiated using the **rpc** system call. All arguments to **rpc** are passed by value. The current RPC implementation accepts four parameters, a LON for the invoked object and three parameters to be passed on to the called procedure. By convention, the first user parameter is an entry point code which the object dispatcher interprets. The other two user parameters are the procedure's arguments. The **rpc** routine generates a trap, which enters kernel mode, pushes the four parameters to **rpc** on the kernel stack, and invokes the kernel's RPC service routine.

Within the kernel, the object manager classifies the called object as a system or user object and determines the processor on which the invoked object resides. If the object is a system object, the entry point routine is executed directly as a procedure call in kernel mode on the corresponding processor node. The major difference between invoking remote objects versus local objects is that remote invocations use the interprocessor communication facility to initiate remote kernel operations.

If the invoked object is not a local system object, the RPC service routine assembles a task descriptor containing the LON, parameters, the identity of the calling thread, and other bookkeeping information. If the invoked object is local, a new thread is created to execute the object invocation. The calling thread enqueues the child thread on the ready queue and blocks awaiting the result from the child. If the invoked object is remote, the task descriptor is inserted in a message and transmitted to the remote node. The current thread blocks awaiting the call's completion. The remote node receives the message and creates a thread to execute the system call indicated in the message.

For both local and remote invocations, the object manager validates the LON and looks up the entry point address of the called object. For system objects, the entry procedure is then called directly. For user objects, the object manager also provides the address space associated with the object. Assuming the LON is valid, a user stack is allocated and the address space for the object is added to the thread's context. Binding the address space to the thread insures that future context switches to this thread automatically activate the address space. Next, the address space is activated, the current state of the task is saved in a catch record (to be restored in case of an exception), the RPC parameters are copied onto the user stack, and control is transferred to the entry point procedure.

When the invoked procedure terminates, either normally or with an exception, control transfers back to the system via a trap. Cleanup includes deallocating any unused catch records established during RPC, detaching the address space from the thread, decrementing the reference count for the object (deleting the object if necessary), and deallocating the user stack. For remote invocations, a reply value is sent back to the caller before completing cleanup, enabling the caller to resume execution in parallel with cleanup on the remote node. The caller is entered onto the ready queue so that it can resume execution. The result of the RPC and any exception code generated are available to the caller. If the procedure terminated with an exception, the same exception is raised in the caller.

4. Performance

The performance results for Elmwood reported here are the result of several man-weeks of careful optimization. We used the C compiler from Green Hills Software Inc., with peephole optimization. We executed the code on a 3-processor Butterfly configuration containing 8 MHz MC68000s, each executing at an effective rate of 0.5 MIPS.

Process creation in Elmwood using the **fork** system call takes 1.73 *ms*. A context switch between processes takes about 200 *us*. In contrast, Chrysalis processes take almost 10 *ms* to create and context switching requires either 219 *us* or 119 *us*, depending on whether or not the context switch occurred as a result of a timer interrupt [16]. Chrysalis process creation is so expensive because, unlike Elmwood processes, each Chrysalis process requires its own address space.

RPC is the fundamental operation for inter-object communication in Elmwood. Elmwood RPC takes 2.30 *ms* in both the local and remote cases, despite differences between the implementations. A breakdown of the time required for the major steps involved in the execution of an RPC between different nodes appears in Table 1. Our measurements provide a detailed accounting for 96% of the time required for RPC. We attribute the remaining 4% (95 *us*) to low-level interrupt processing and instrumentation overhead. Ignoring miscellaneous overhead, the setup time for RPC on the caller's node is 271 *us*; processing time on the callee's node is 1580 *us*; cleanup on the callee's node takes 479 *us*, but is overlapped in time with the return to the original caller, which takes 354 *us*. This overlap explains why remote execution takes no longer than local execution, even though more work is required.

The best evidence of the efficiency of Elmwood RPC is a comparison with the Butterfly implementation of the Lynx programming language [19]. The Lynx implementation is based on Chrysalis and has been carefully optimized. Although Lynx uses message passing rather than RPC, Lynx messages with implicit receipt are very similar to RPC in Elmwood. Implicit receipt in Lynx takes 2.04 *ms* when the processes are on different nodes and 2.76 *ms* when the processes are

Elmwood Remote Procedure Call

Caller:	
Kernel trap; parameter passing	142 *us*
Miscellaneous setup time	40 *us*
Enqueue data block; issue remote interrupt	89 *us*
Callee:	
Field interrupts	28 *us*
Dequeue data; create thread	267 *us*
Return from interrupt; transfer to thread	186 *us*
Thread initialization	26 *us*
Mask interrupts; copy parameter block	160 *us*
Validate LON; attach address space	115 *us*
Allocate stack; activate address space	188 *us*
Miscellaneous setup of user context	159 *us*
Copy params to user stack; call proc; trap	167 *us*
Pass result to caller; issue remote interrupt	284 *us*
Cleanup	479 *us*
Caller:	
Field interrupts	26 *us*
Dequeue data; transfer to caller	290 *us*
Miscellaneous cleanup	38 *us*
Other misc overhead (caller and callee)	95 *us*

Table 1

on the same node. The corresponding figures for Elmwood are surprisingly similar to those of Lynx, even though the semantics of the two operations are slightly different. Unlike Elmwood, Lynx RPC is cheaper if the caller and callee are on different nodes. Since the Lynx implementation performs the same operations in both cases, parallelism due to overlapped execution makes the remote case faster. In Elmwood, the implementation of RPC is quite different depending on the relative locations of the caller and callee. Overlapped execution is not sufficient to overcome the additional overhead associated with RPC to another node. In this light, the performance of Elmwood RPC is very good, particularly since Elmwood RPC involves crossing protection domains enforced by the kernel, whereas Lynx protection is enforced by a compiler.

Chrysalis does not implement an operation similar to Elmwood RPC, so a direct comparison between Elmwood and Chrysalis is not possible. However, we can make some assumptions about the cost of an RPC implementation based on Chrysalis primitives. Table 2 summarizes the cost of selected primitives in Chrysalis [8]. It shows that the time required in Chrysalis to map a memory object into an address space, copy parameters into it, post its handle to another process using a

Execution Time for Selected Chrysalis Operations

Atomic increment remote memory	34 *us*
Block transfer (minimum)	50 *us*
Enqueue or dequeue on a dual queue	80 *us*
Setup catch block	70 *us*
Make an object	1000 *us*
Dynamically allocate memory (average)	370 *us*
Map a memory object into a specific SAR	660 *us*
Map a memory object into any SAR	1350 *us*
Create process (minimum)	9750 *us*

Table 2

dual queue, retrieve the handle on a remote node, map the memory object again, and access the parameters is over 1.5 *ms*. (We assume that memory objects are cached and that a SAR is reserved for parameter transmission.) Any *general* scheme for communication on the Butterfly will have higher costs, although special-purpose implementations can be significantly faster. For example, message passing using statically allocated buffers at known addresses between two executing user processes can be very efficient: 750 *us* per roundtrip message, 530 *us* if modified versions of the Chrysalis dual queues are used [13].

5. Experiences

In this section, we have tried to capture our experiences in designing and building an operating system in a group setting under fairly rigid time constraints. As a group, we discussed the problems that arose during both the design and implementation of Elmwood, and attempted to determine a root cause for each. In many cases, we discovered that problems were caused by our failure to recognize that a particular situation was a subtle violation of a general software engineering principle. In what follows, we have divided our lessons into three categories: (1) distinguishing goals, policies, and mechanisms, (2) problems associated with defining and building abstractions, and (3) performance issues.

5.1. Goals, Policies, and Mechanisms

Measure all design proposals against agreed upon design goals. A major factor in the rapid design of Elmwood was that we agreed on a consistent set of design goals before we began. Having a specific set of goals against which to measure design alternatives is crucial for forward progress. Often we were able to resolve a technical disagreement by demonstrating an inconsistency with the agreed upon goals. Without a clear set of design goals, there may be no way to choose among

different design alternatives.

Separate policy and mechanism. The separation of policy and mechanism [12] has well-known advantages: policies can be implemented outside the kernel and one kernel mechanism can support multiple user policies. Based on our experiences, we can add to this list: the separation of policy and mechanism facilitates speedy implementation of the kernel. Basic kernel mechanisms can be implemented quickly, if specific policies are avoided. User-level policies require system experience and evaluation, and therefore are best implemented over time. The Elmwood kernel implements several simple mechanisms, but few policies. During the design stage, we examined each mechanism with respect to its ability to implement many different policies, although in most cases no specific policies were implemented. Whenever possible, we deferred policy issues to a subsequent user layer. As a result of our design philosophy, we developed very general kernel mechanisms and were able to build the kernel in a short period of time. The few design bottlenecks we did encounter could be attributed to policy discussions disguised as kernel issues, none of which could have been resolved without experience with the system. The consensus of the group is that separating policy and mechanism was a major factor in our ability to complete the kernel quickly.

5.2. Defining and Building Abstractions

Do not sacrifice modularity for expedience. Expedience often leads to generalizations in the implementation, which may be inefficient, but require less code. Later, when the system is working and performance figures are available, optimizations can be used to improve performance. Optimization is complicated, however, if modularity has been sacrificed for expedience. For example, in Elmwood we used calls to the general-purpose memory allocator **malloc** each time storage was needed by the kernel, even though we knew that malloc would be expensive. We chose **malloc** as an expedient solution to the problem of general-purpose memory allocation, a solution that could be tuned later by adding special-purpose allocators (*e.g.*, cached context blocks). Knowing that we would ultimately replace calls to **malloc** with calls to a special-purpose allocator, we should have created abstract interfaces for each different use of the general-purpose allocator. These abstract interfaces would not hide complexity (after all, each interface would be implemented by a single call to **malloc**); rather, they would hide the implementation of the memory allocator. We did not construct these abstract interfaces because we were seduced by the simplicity of the general-purpose interface. Constructing the necessary abstract interfaces would not have affected the solution's expedience, but would have made it easier to add the special-purpose allocators during the performance tuning stage.

Delineate abstraction boundaries with people. In an operating system kernel running within a single address space, there is a natural tendency to blur abstraction boundaries for the sake of efficiency. Absent a compiler and programming language that enforce strict module boundaries, there is no mechanism for limiting this tendency. It is difficult, however, to make critical assumptions about the implementation of an abstraction when someone else is responsible for the code. In Elmwood, each layer of the kernel began with a well-defined interface, discussed by the group as a whole, but implemented by a single person. Since there were more layers than people, each person had to implement more than one layer. Lacking the benefit of our own experience, we found that some abstractions had the same implementor on both sides of the interface. As a result, either the interface was completely ignored, resulting in a single layer, or some subtle assumptions were made about the implementation. In one case, the distinction between a coroutine and a thread was confused because both abstractions were created by the same programmer. Problems later arose with the operating system scheduler, which was technically a coroutine but was being treated as a thread. The confusion went undiscovered for some time, since we had no mechanism, either managerial, language-oriented, or compiler-based, for enforcing interface specifications.

Build a separate machine interface containing all hardware dependencies. Separating machine dependencies from machine-independent functions has obvious advantages when building a portable operating system [5]; however, we recommend the practice even when portability is not an issue. It isn't necessary to completely design the machine interface first; it can be extended as needed during implementation. In Elmwood, we implemented machine-dependent functions as needed. Each layer included as a coherent subset those machine-dependent functions required to implement that layer. Integrating machine-dependent functions within the first layer to require them works well for the fundamental abstractions of the machine; however, some of the machine details may not fall within any of the fundamental abstractions. One of the quirks of the Butterfly memory architecture is the overlap of SARs F8-FF with 1F8-1FF, a convention that must be maintained in software. This particular convention does not represent a significant abstraction, which would merit its own layer in the system, and was not readily identified as part of any machine-dependent abstraction we had implemented. As a result, violations of the convention were difficult to discover. Similarly, management of a volatile SAR, a SAR that can be allocated whenever needed by the kernel, is interspersed throughout the kernel, since no one layer presented an abstraction for this particular function. A separate machine interface, containing all architectural assumptions, would have encapsulated these implementations, thereby avoiding some of the problems we encountered.

Make module transparency explicit. No interface in a hierarchically structured system, especially the machine interface, should hide important capabilities available in lower levels [15].

Such a *loss of transparency* can lead to inefficiency at higher levels. However, there is a tendency in hierarchically structured systems to ignore lower level interfaces and build each level using the abstractions provided by the previous level alone. Therefore, we feel a stronger statement of module transparency is necessary: *each interface should make explicit those abstractions that are available to higher levels due to module transparency.* A loss of transparency then results when the interface exported by a lower layer is not fully contained in the interface of the next higher layer. In Elmwood, the SAR abstraction was implemented in the address space manager, above the physical memory allocator. Although the physical memory allocator did not hide the abstraction of memory which it had not allocated, it did not explicitly export it. This lapse led to the implementation of SAR management routines that assumed SARs described memory allocated by the physical memory allocator, ignoring both unallocated memory and kernel memory. The implicit assumption linking SARs and previously allocated memory went undiscovered until we needed a SAR that referenced kernel memory, in order to implement parameter passing in interprocessor communication. At that point, several SAR management routines had to be rewritten. If we had explicitly exported the notion of unallocated memory from the physical memory layer, we would probably have avoided this problem in the address space manager.

5.3. Performance Issues

Invest in cheap primitives. Expensive operations tend to be avoided, even when performance is a secondary issue. System implementors must believe that the cost of using each low-level primitive is reasonable, or that future optimizations would ultimately lead to a reasonable cost for each primitive. Early on, Elmwood exceptions were believed to be too expensive to use in the kernel, despite the fact that exceptions played an important role in the user interface. We failed to realize that users would be unlikely to accept expensive primitives previously rejected by the system implementors. Although we later discovered an important optimization in the implementation of exceptions, we could not exploit it in the kernel since, by that time, the kernel had been programmed without them. As a result, each procedure in the kernel returns an error code to signal an exceptional condition. Had we considered other implementations of exceptions before deciding not to use them in the kernel, we would probably have realized that our assumption about the inherent costs of exceptions was unfounded. In contrast, we did not avoid procedures in the implementation of abstractions because the C preprocessor provides a mechanism for in-line function substitution, thereby providing procedural abstraction without procedure call overhead

Recognize frequent special cases. General-purpose routines may provide superior functionality, but special-purpose routines are more efficient. When first designing the system, abstract interfaces should include separate operations for frequent special cases. The implementation of

those special cases can use the general-purpose routines during initial development. The optimization phase can then replace calls to general-purpose routines with special-purpose implementations. For example, we decided early on to have general routines for manipulating the interrupt mask and special routines for dealing with clock interrupts. We also implemented a general routine to map segments into SARsets, and a special routine to map stack segments in the *physical* SARs. As a result, we were able to manage stack segments more efficiently, since stack segments are not part of the address space and needn't be maintained in the shadow copy of the SARs. On the other hand, we failed to realize that the coroutine mechanism, upon which processes are built, could exploit a special transfer operation. We eventually discovered that transfers from the idle coroutine or a dying coroutine were saving unnecessary state. Admitting special case operations upfront enables future optimizations to improve performance, while still preserving the individual abstractions.

Redesign for performance. In the first implementation of a system, each layer will contain some extraneous overhead. If this overhead amounts to 50% for each layer in a six-layer hierarchy, the end performance is reduced by an order of magnitude [11]. Therefore, optimizations that remove the extraneous overhead will often yield a factor of 10 performance improvement in any implementation. (Our initial implementation of RPC took 21 *ms*; our current implementation takes 2.3 *ms*.) However, redesign of both the implementation and the user interface may be necessary to gain additional performance. In particular, we can reduce the cost of each operation by amortizing costs over many operations. For example, the first implementation of Elmwood RPC dynamically allocated stack space to serve each RPC. The optimized implementation uses a stack cache. Further improvements in performance are possible if all resources needed to implement RPC are preallocated. We considered providing an **open_object** system call, analogous to opening a file in Unix, which would create an object descriptor for use in each RPC. The object descriptor would represent a binding to an object, a preallocated stack, and any other resources used by RPC. Additional performance could have been gained, but a change to the user interface would have been required. If performance were to become a primary goal, we would expect to redesign the interface based on our experience with the current implementation.

6. Conclusions

We are satisfied with our choice of the Butterfly as our hardware base. Even though the tight coupling between the PNC microcode and the Chrysalis operating system made building an operating system from scratch difficult, we were successful in doing so. The unconventional memory architecture supported by the Butterfly led us to ignore traditional issues in memory management, such as demand-paging, and to concentrate on protection. We also learned a great deal about a

machine on which we base much of our research.

We are also pleased with the basic protection mechanism, namely LONs. Although we have little actual experience writing programs with LONs, we strongly believe that LONs are a general mechanism for implementing flexible user-level policies. The cost of managing LONs compares favorably with capabilities; the flexibility provided by LONs compares favorably with access lists.

We are not entirely satisfied with the Elmwood user interface. We now see the need for an additional system call to extract the context value from any LON associated with an object. This additional functionality would simplify management of LONs, contexts, and multiple clients. Also, we could remove the need for system calls to create processes, semaphores, and condition variables. System objects, which implement these facilities, already use RPC for most of their operations, but special system calls create instances within each object. If we assume the existence of a *name server* object with a well-known LON, then system objects could be registered with the name server, and all system object operations, including creation, could be entry procedures rather than system calls.

Even now, there remains one important, unresolved issue. Throughout the design we made an implicit assumption that the traditional notion of process, including the association between processes and users, could be supported easily by passive protection domains. Each process would be an object; process management functions would be entry procedures associated with each process object. We also assumed that process operations would be controlled; no process could delete another arbitrarily, but each process could be deleted by some other process. These assumptions caused us to overlook the unique interactions between passive protection domains (*i.e.*, objects) and active entities (*i.e.*, processes). For example, the ability to destroy a process at any point in time is not consistent with the need for each object to maintain control over its state; process deletion must eventually depend on the cooperation of some unknown object. The problem is solved in most systems by either (a) associating processes with objects one-to-one, so that a process then becomes a protection domain, or (b) not addressing directly the problem of process management, so that multiple object invocations must be coordinated explicitly to create the abstraction of a coherent process. We rejected both of these choices for Elmwood. We did not anticipate that in so doing the interaction between passive protection domains and active entities would become the most complex and time-consuming aspect of the design. In retrospect, we should have promoted processes to first-class entities much sooner, which would have forced us to address the interaction with objects earlier in the design.

We still believe our original goals to be interesting, worthwhile, and consistent. We also believe we were fairly successful in attaining those goals. The system is both *functionally complete*

and *object-oriented*. *Protection* is clearly an integral part of the system. The Elmwood kernel is *small* and was implemented very quickly. Our design is *consistent with a multiprocessor implementation*; extending the implementation to multiple processors did not change the user interface and was completed in less than one man-month. *Orthogonality of mechanism* has been maintained; in fact, it became our most important goal and the one that most influenced the design. However, our goal to *allow processors to share memory, as in the BBN Uniform System*, was not attained; we set it aside early on for the sake of expedience.

Despite all our attempts to the contrary, Elmwood mechanisms are not entirely orthogonal. In particular, users create processes using **fork** to call an entry procedure. Thus, to create a process representing an asynchronous activity, the user must have access to an object that provides an operation appropriate for an independent process. An alternative, analogous to the Unix fork mechanism, would create another process within the current object with a copy of the creator's stack, thereby decoupling RPC and process creation.

The direct sharing of memory between objects raises semantic questions that we felt could not be addressed adequately in the allotted time. We have since designed consistent semantics for shared memory objects, including operations to map and unmap memory objects in a normal object's address space. While we believe it is possible to add shared memory to Elmwood without compromising other aspects of the design, it will require significant changes in the implementation. Ultimately, that is a step that must be taken, since the performance of applications using a shared-memory multiprocessor may depend on the ability to exploit shared memory.

Acknowledgements

Takahide Ohkami implemented an earlier version of the object manager and assisted in debugging the system. Tim Becker and Steve Whitehead helped implement Elmwood on multiple processor nodes. Sanjay Jain and Rajeev Raman implemented an improved Elmwood shell and a registration object. We are grateful to Walter Milliken of BBN Advanced Computers Inc., who provided important technical assistance.

References

1. G. T. Almes, A. Black, E. D. Lazowska and J. D. Noe, "The Eden System: A Technical Review", *IEEE Transactions on Software Engineering SE-11*, 1 (Jan 1985), 43-58.

2. Y. Artsy, H. Chang and R. Finkel, "Interprocess Communication in Charlotte", *IEEE Software 4*, 1 (Jan 1987), 22-28.

3. BBN Laboratories, "The Uniform System Approach To Programming the Butterfly Parallel Processor", Version 1, Oct 1985.

4. F. Baskett, J. H. Howard and J. T. Montague, "Task Communication in Demos", *Proc. 6th Symp. on Operating Systems Principles*, West Lafayette, Indiana, Nov 1977, 23-31.

5. D. R. Cheriton, M. A. Malcolm, L. S. Melen and G. R. Sager, "Thoth, a Portable Real-Time Operating System", *Comm. ACM 22*, 2 (Feb 1979), 105-115.

6. D. Clark, "The Structuring of Systems using Upcalls", *Proc. 10th Symp. on Operating Systems Principles*, Orcas Island, Washington, Dec 1985, 171-180.

7. E. Cohen and D. Jefferson, "Protection in the Hydra Operating System", *Proc. 5th Symp. on Operating Systems Principles*, Austin, Texas, Nov 1975, 141-160.

8. P. C. Dibble, "Benchmark Results for Chrysalis Functions", Butterfly Project Report 18, Computer Science Department, University of Rochester, Dec 1986.

9. D. E. Knuth, *The Art of Computer Programming:* Volume 1, 2nd Ed. *Fundamental Algorithms*, Addison-Wesley, Reading, Mass., 1973.

10. B. W. Lampson and D. D. Redell, "Experience with Processes and Monitors in Mesa", *Comm. ACM 23*, 2 (Feb 1980), 105-117.

11. B. W. Lampson, "Hints for Computer System Design", *Proc. 9th Symp. on Operating Systems Principles*, Bretton Woods, New Hampshire, Nov 1983, 33-48.

12. R. Levin, E. Cohen, W. Corwin, F. Pollack and W. Wulf, "Policy/Mechanism Separation in Hydra", *Proc. 5th Symp. on Operating Systems Principles*, Austin, Texas, Nov 1975, 132-140.

13. J. R. Low, "Experiments with Remote Procedure Call on the Butterfly", Butterfly Project Report 16, Computer Science Department, University of Rochester, Dec 1986.

14. W. Milliken et al., "Chrysalis Programmer's Manual, Version 2.2", BBN Laboratories, June 1985.

15. D. L. Parnas and D. P. Siewiorek, "Use of the Concept of Transparency in the Design of Hierarchically Structured Systems", *Comm. ACM 18*, 7 (July 1975), 401-408.

16. R. D. Rettberg, "Development of a Voice Funnel System", Quarterly Technical Report No. 18, Bolt Beranek and Newman Inc., Apr 1983.

17. J. H. Saltzer, "Protection and the Control of Information Sharing in Multics", *Comm. ACM 17*, 7 (July 1974), 388-402.

18. M. L. Scott, "Language Support for Loosely-Coupled Distributed Programs", *IEEE Transactions on Software Engineering SE-13*, 1 (Jan 1987), 88-103.

19. M. L. Scott and A. L. Cox, "An Empirical Study of Message-Passing Overhead", *Proceedings of the Seventh International Conference on Distributed Computing Systems*, Berlin, West Germany, Sept 1987 (to appear). Also BPR 17, Computer Science Department, University of Rochester, Dec 1986.

20. N. Wirth, *Programming in Modula-2, 2nd Ed.*, Springer-Verlag, 1982.

END

DATE

FILMED

5-88

DTIC