THE *CHOICES* OBJECT-ORIENTED OPERATING SYSTEM ON THE SPARC ARCHITECTURE

BY

DAVID K. RAILA

B.S., University of Illinois, 1988

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1992

Urbana, Illinois

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

In this thesis I discuss the design and implementation of the *Choices*[2, 3, 10] operating system for the SPARC[11] architecture. *Choices* is an object-oriented operating system written in C++[13] with a small amount of assembler to provide access to machine registers. *Choices* uses object-oriented[12] techniques to provide a modular, customizable, and portable operating system without sacrificing efficiency. It has been designed to run on single processor and shared memory multiprocessor computers.

This thesis presents the results of porting *Choices* to the SPARC architecture. The SPARC architecture is significantly different from the other platforms to which *Choices* has been ported. The SPARC port presented an opportunity to evaluate the structure of *Choices* in the context of a RISC architecture and to reorganize it to be more portable and efficient by providing better abstractions in the subsystem class hierarchies.

Some of the results from this work show: The object-oriented approach results in a portable, modular, and efficient operating system. Encapsulating architecture dependencies behind small, general interfaces enhances portability and efficiency. The SPARC register window system

provides no particular performance advantage or disadvantage with respect to the operating system, and is difficult to program. The SPARCstation virtual address cache requires complex software support that a physical cache does not.

This thesis has two parts. Chapters 2-4 present introductory material consisting of an overview of object-oriented programming, an overview of the relevant subsystems of *Choices*, and an overview of the SPARC architecture. Chapters 5-7 present the implementation details of *Choices* on the SPARC architecture and the experiences gained in implementing *Choices* on the SPARC architecture.

Chapter 2

Object-Oriented Programming Overview

"Object-Oriented" programming [12] has become a common term used to describe software. Different views of what object-oriented means and a lack of common terminology and notation contribute to confusion about object-oriented systems. This chapter provides definitions of the object-oriented model and terminology used with *Choices*.

2.1 Objects and Classes

The object-oriented paradigm uses **objects** to encapsulate a data structure and the operations on that structure into a single unit. All access to the object is through the operations, called **methods**. Programs are constructed by sending **messages**[1] to objects to modify or access the information in their data structure. By hiding the data structure behind methods the implementation of the object is insulated from its users.

[1] A message is the invocation of a method at run-time.

Each method has a name and a **signature** consisting of the types of the arguments it accepts and the value it returns. The entire set of methods and their signatures is the **protocol** of the object.

Sending a message to an object consists of a method lookup and a call to the method with the message arguments. The method lookup matches the signature used in the call to the method that the object provides of the same name and signature. If the signature of the message is not matched by any of the methods the message is an error[2].

The **class** is the basic unit of abstraction in object-oriented programs. Every object is an **instance** of a class. Classes allow similar objects to share structure and code. A class is like a template describing the structure and methods for objects of that class. In some object-oriented languages the classes are objects to which messages can be sent.

2.2 Inheritance and Class Hierarchies

Inheritance allows sharing of structure and protocol between similar classes. A **subclass** inherits the features of its **superclass** or **parent** class. Inherited methods and inherited structure can be used in subclass code, allowing the subclass and superclass to share common features.

Subclasses can extend their protocol by adding structure and methods to their inherited protocol. Subclasses can **override** or **overload** methods inherited from their superclass to implement their own specialized versions of those methods. When a method is dispatched that has been overridden in the subclass, the method matching the signature at the lowest level in the inheritance tree for that particular object is the method used for the message.

[2] In C++ these errors are caught at compile time.

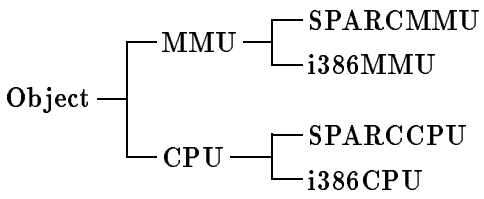

Figure 2.1: A portion of the *Choices* class hierarchy.

Subclassing allows the abstraction of common structures and protocols into **class hierarchies**. In a class hierarchy common methods and structure exist in superclasses and customizations and optimizations are overridden in subclasses.

A portion of a class hierarchy from *Choices* is shown in Figure 2.1 as an example. In this example common methods to all *Choices* objects are contained in the Object class, such as those to set and return the name of the object. The CPU class inherits from Object and implements methods that are common to all processors. Subclasses of CPU will override machine specific methods of CPU to implement them for a particular processor. By inheriting common *Choices* Object methods each machine dependent CPU has access to methods to set and return its name.

2.3 Polymorphism

A method that accepts object arguments and sends messages to the objects to accomplish a task can accept any object that matches the protocol that it uses. This feature, called **polymorphism**, allows re-use of abstract code by invoking the method with different types of objects supporting the proper protocol. Polymorphic methods are usually constructed to operate on objects of superclasses and used with objects of subclasses which override key methods. This

allows generic methods to be written using protocols in superclasses. Implementations re-use the generic code by subclassing and overriding implementation specific methods.

An example of polymorphism can be found in the *Choices* filesystem. The low levels of the filesystem system are written to accept objects of class Disk. An object of any subclass of Disk can be used by the filesystem to do I/O. Examples of Disk subclasses are machine dependent disks that perform machine specific I/O, and network disks which uses the network to perform remote I/O. The subclassed disk object is passed to the I/O subsystem which uses the overridden methods in its polymorphic methods.

2.4 Object-Oriented Design and Frameworks

Classes in the upper levels of hierarchies that are not instantiated directly but exist only to be subclassed are called **abstract classes**. Abstract classes exist to provide interface definitions and partial implementations in polymorphic methods. Subclasses that are instantiated directly are called **concrete classes**. Concrete classes override key methods of their superclasses and are used by polymorphic methods in the abstract classes.

An object-oriented **framework** [6] is a set of interacting abstract class hierarchies implementing a re-usable object-oriented design. The abstract classes within the framework define common protocols and relationships within and external to the framework. Subclasses provide implementation details and customizations. Polymorphic methods bind the abstract partial implementation to objects of the concrete classes at run time.

2.5 Summary

The object-oriented programming paradigm supports abstracting data structures and the access of those structures into classes of objects. Objects are manipulated by sending messages to change or return the state of the object. Hiding implementation details behind methods insulates users of objects from implementation details.

Inheritance within class hierarchies lets subclasses re-use structure and protocol from superclasses. Subclasses extend their protocol by adding methods and customize it by overriding inherited methods.

Polymorphism supports the re-use of methods by passing objects providing the required protocol to polymorphic methods. Abstract classes exist to define an interface specification and a partial polymorphic implementation and are not instantiated. Concrete classes inherit from abstract classes and override key methods to provide implementation details.

Inheritance and polymorphism provide the means for reusing object-oriented designs by constructing frameworks of abstract classes implementing the reusable design. A framework is re-used by providing concrete subclasses of abstract framework classes containing implementation details. By implementing a system as an object-oriented framework a high degree of reuse, modularity, and portability can be achieved.

Chapter 3

Overview Of *Choices* Porting Issues

This chapter is an overview of the major frameworks of **Choices** that are encountered when porting the system to a new architecture. The descriptions are sufficient to provide background information for understanding the SPARCstation implementation described in chapter 5.

3.1 Introduction

Choices[2, 3, 10] is a set of object-oriented frameworks for implementing operating systems. Some goals of *Choices* are to provide parallelism, customization, distribution, portability, and dynamic configuration in an operating system without sacrificing efficiency. To accomplish this each subsystem of the kernel is implemented as a C++ framework.

Within each subsystem implementation details, system policies, machine dependencies, and optimizations are isolated in subclasses. Interfaces between and within frameworks are defined by abstract classes. Subclassing and polymorphism is used within the frameworks to isolate system policies from their mechanisms, to allow dynamic customization, and to provide reusable designs.

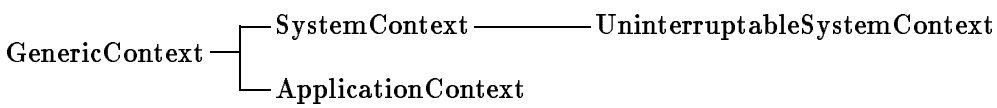

Figure 3.1: ProcessorContext Abstract Class Hierarchy.

3.2 Process System

In *Choices* a thread of control is represented by an instance of the class **Process**. A Process runs in a virtual address space and resource environment called a **Domain** (see section 3.3).

Each Process contains a machine dependent subclass of **ProcessorContext**, which saves and restores the CPU state of the Process. ProcessorContexts overload the methods checkpoint, restore, and restoreFromInitialContext. Checkpoint saves the processor dependent CPU state and returns the value 0 to it's caller. Restore arranges for the first checkpoint to return again, but with the return value equal to the Process argument passed to the call. RestoreFromInitialContext exists to dispatch the ProcessorContext initially.

ProcessorContext constructors are passed arguments that specify the Domain that the Process will run in, the entry point that the Context should start at, and the parent Process of the ProcessorContext. It is the responsibility of the concrete subclasses of ProcessorContext to use this information to dispatch the Process properly.

ProcessorContext is subclassed for each different type of Process such as **ApplicationProcesses**, or **SystemProcesses**. Each subclass of ProcessorContext is optimized to save and restore the minimal amount of machine state for that type of Process. The ProcessorContext hierarchy is shown in Figure 3.1.

3.2.1 Exceptions

The **Exception** class defines objects representing traps and interrupts in *Choices*. Subclasses of Exception override the basicRaise method which is called by CPU specific code when that exception occurs.

Subclasses of CPU override the method installException which takes an Exception and a vector and binds them logically together. When a hardware trap at that vector takes place the processor dependent code calls the basicRaise method on that Exception. On most machines this is implemented using a table of Exceptions stored in the CPU and low level support code written in assembler.

3.3 Memory System

Choices provides an object-oriented framework for virtual memory systems. The framework supports customization and flexibility while taking advantage of hardware support for virtual memory to provide efficient implementations.

3.3.1 Machine Independent Classes

In *Choices* a **Domain** is the virtual memory environment in which Processes run. A Domain consists of a virtual address space into which MemoryObjects are mapped for direct memory addressing. Domains maintain a list of MemoryObjects, their addresses, and permissions. A Domain is a machine independent representation of a virtual memory environment.

The class **MemoryObject** defines logically addressed objects with the methods read and write that transfer data to and from their logical units. Each MemoryObject may be mapped into multiple Domains in multiple places. Processes share data by sharing the same Domain,

	NoAccess	ReadOnly	ReadWrite
Kernel Read Access Allowed	Yes	Yes	Yes
Kernel Write Access Allowed	Yes	Yes	Yes
Application Read Access Allowed	No	Yes	Yes
Application Write Access Allowed	No	No	Yes

Table 3.1: Protection Capabilities in the *Choices* Memory System.

or by sharing a common MemoryObject though their separate Domains. Domains for ApplicationProcesses in *Choices* include the MemoryObjects and mappings of the Kernel Domain, assuring that the Kernel is always available regardless of the active Domain.

Each MemoryObjects has an associated **MemoryObjectCache** that implements paging for that MemoryObject. The MemoryObjectCache maps logical units within the MemoryObject to physical pages in the **Store**, Each MemoryObjectCache has a policy module that determines the paging policy for that MemoryObject and is consulted when a page fault occurs. This allows paging policies to be customized for each MemoryObject to implement working set, copy on write, or other custom paging policies.

The enumerated type **ProtectionLevel** defines support for implementing page protection. It includes three states: NoAccess, ReadOnly, and ReadWrite. Table 3.1 shows the allowable accesses from system and application mode for each of the protection levels. The machine independent code passes these protection levels to the machine dependent code, which is responsible for implementing the levels with the available hardware protection support.

3.3.2 Machine Dependent Classes

Choices defines machine dependent abstract classes in the low levels of the memory system that are subclassed for each memory architecture on which *Choices* runs. The abstract classes define generic protocols that machine dependent subclasses must support. Encapsulating the structure

and algorithms of the machine's memory architecture in subclasses and using a common protocol provides portability, efficiency, and customizability without affecting the machine independent classes.

The hardware Memory Management Unit and Translation Lookaside Buffer cache within the machine are represented by subclasses of the abstract class **MMU**. Subclasses of the MMU class override the methods basicActivate and flushTLB. BasicActivate makes the translation information in the AddressTranslation argument active on the MMU. The flushTLB method is used to flush the MMU's translation cache when mappings are changed to avoid having the MMU use incorrect translation information.

The class **AddressTranslation** encapsulates the translation tables that the MMU uses to translate virtual to physical addresses. Subclasses of **AddressTranslation** override methods addMapping, removeMapping, and changeProtection. AddMapping and changeProtection are passed a virtual address, a list of pages, and a ProtectionLevel to set up the translation table appropriately. The removeMapping method is passed a virtual address and a length to make that region of mappings invalid.

3.3.3 Framework Interactions and Page Faults

Each Domain contains an AddressTranslation that is passed to the MMU::basicActivate method when the Domain is made active on a MMU. Because the kernel mappings are available in any domain and a SystemProcess can run in any domain, a domain change is only necessary when an ApplicationProcess is being dispatched that is not in the same Domain that is active on the MMU currently. The Process switching code changes Domains when necessary by activating the proper Domain's AddressTranslation on the MMU just before the ProcessorContext is restored.

When a page fault occurs the basicRaise message is sent to the Exception object representing page faults in the system. The Exception will determine from the machine the virtual address and the access type that caused the fault[1]. The Exception passes the address and access type to Domain::repairFault, which attempts to repair the fault and update its AddressTranslation. The Domain returns a value of type **FaultErrorCode** to the Exception to communicate whether it was able to repair the fault or not. The Exception returns to re-execute the faulting instruction or kills the current Process depending on the FaultErrorCode.

3.4 System Initialization

The *Choices* model requires the Kernel to exist in low memory and requires disjoint Kernel and application virtual address spaces. When the bootstrap procedure (see section 5.4) passes control to the kernel, *Choices* assumes that it is running in low physical memory with the MMU disabled. It is the bootstrap procedure's responsibility to load the kernel into low memory and start executing it at the entry point.

Choices first sets up a boot allocator. This allocates memory starting at the end of the Kernel and grows upward. Memory allocated by the boot allocator is considered part of the Kernel and is mapped 1-1 when virtual memory is enabled.

To maintain proper C++ semantics, static constructors must be called once the boot allocator is in place. Then the virtual memory system is initialized, the Kernel heap is created, and the first Process is dispatched. The entry point of the first Process is Kernel::main, which finishes machine independent initialization and dispatches applications.

[1]The access type is expressed as a ProtectionLevel.

3.4.1 Machine Dependent Constants

To communicate machine dependent constants to machine independent initialization routines the Kernel::basicInitialize method is overridden by machine specific subclasses of Kernel. This method is responsible for creating memory maps specifying the Kernel VM ranges and the application VM ranges as well as a list of physical memory ranges that the machine independent parts of the Kernel can use to perform initialization.

3.4.2 Virtual Memory Initialization

The AddressTranslation is initialized early while virtual memory is still disabled. Subclasses of AddressTranslation define a static method named init that is called at the beginning of VM initialization. This method is responsible for allocating memory from the boot allocator that will be used for translation tables. This allows the CPU and MMU to use the same addresses when operating on translation tables since the boot memory exists at the same physical and virtual addresses when virtual memory is enabled.

Immediately after the AddressTranslation initialization the MemoryObjects representing the Kernel are created and installed in a new Domain. In general there are low kernel, read-only kernel, high kernel, and kernel heap MemoryObjects. Once installed the MemoryObjects are forced to be addressable to prime the AddressTranslation with the Kernel mappings.

A new SystemProcess is created to run in the Domain with Kernel::main as the entry point. Immediately before the Process is dispatched and after all initialization, the virtual memory based allocator is created and inserted into the Kernel Domain. When the process is dispatched the Domain activation code makes the AddressTranslation for the Kernel Domain active on the MMU and enables the MMU during the Process switch. The Kernel is now running with VM

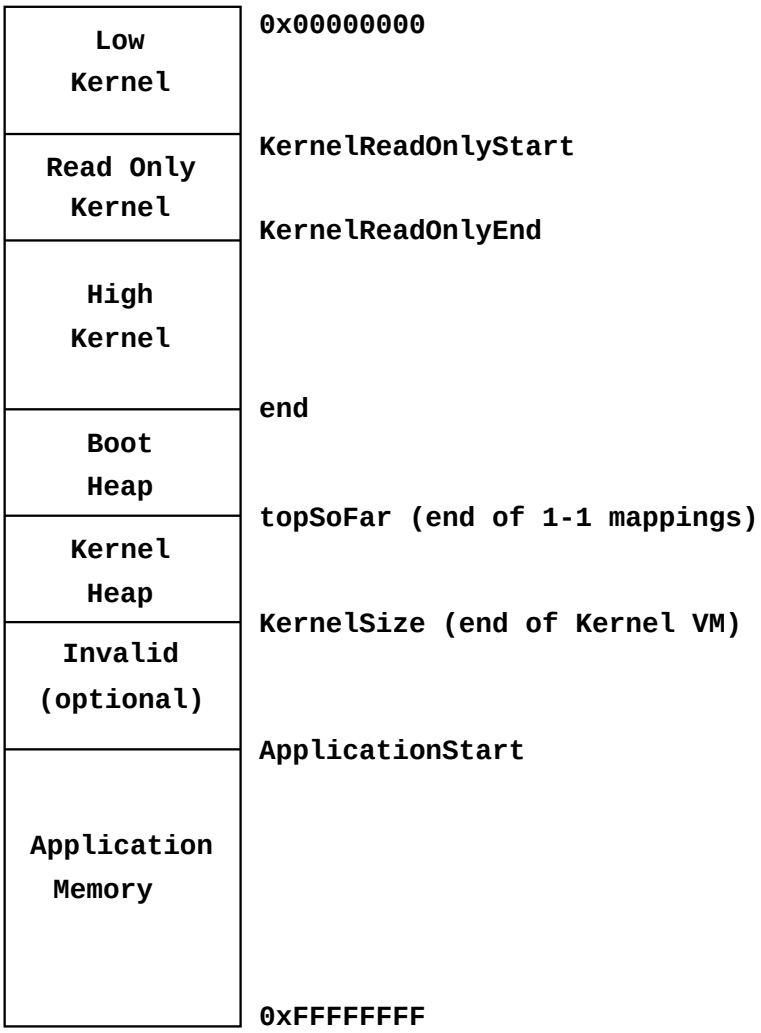

Figure 3.2: The *Choices* Virtual Memory Model.

initialized, the MMU activated, and a Kernel virtual memory allocator available. Figure 3.2 shows the general *Choices* virtual memory layout after initialization.

3.5 Summary

The *Choices* operating system is implemented as a number of interacting subsystem frameworks written in C++. High level structure and interfaces of each subsystem are specified with ab-

Subsystem	Number of Classes	Lines of Interface Code	Lines of Implementation Code
Kernel	58	4374	9659
Common	103	4932	11463
Class Library	32	1927	2812
FileSystem	90	5715	16428
Memory	32	2742	4793
IODevices	19	1475	1325
Instrumentation	10	1205	1500
Schedulers	27	750	2000
UnixCompatibility	42	3940	7560
Total	413	27060	57540

Table 3.2: Source Code Statistics For *Choices* Subsystems.

stract classes and polymorphic methods. Subclassing and polymorphism support customization and isolation of machine dependencies within the lower layers of the system.

The porting effort is isolated in the Process and virtual memory systems and the system initialization model.

Table 3.2 shows statistics for different code portions of *Choices*[2]. Subsystems other than those presented in this chapter are shown for comparison.

[2]The interface code count contains implementation methods that are defined in-line.

Chapter 4

SPARC Architecture Overview

SPARC is an acronym for **S**calable **P**rocessor **ARC**hitecture. It is a processor instruction set architecture, developed by Sun Microsystems. The current version, called SPARC architecture version 8, is published by SPARC International[11]. This chapter gives an overview of RISC, details of the SPARC architecture relevant to writing system software, and an brief overview of the Sun SPARCstation.

4.1 RISC Architectures

RISC is an acronym for **R**educed **I**nstruction **S**et **C**omputer. The RISC philosophy is that complex general purpose instructions complicate CPU design and are not used often enough to justify their implementation. The goal of RISC designs is to keep the instruction set simple and relatively small to avoid complex implementations that could degrade the performance of frequently executed simple instructions[4].

To accomplish the goal of execution rates of one instruction per cycle, RISC CPUs pipeline instructions. Instruction pipelines overlap instruction executions in stages, with each stage

executing in parallel. While one instruction is in the fetch stage, the previous instruction is in the decode stage, and the previous instruction to that is in the execute stage.

The ability to pipeline requires fixed length, fixed execution time instructions operating on register operands to avoid pipeline stalls. A separate set of instructions is used load and store between memory and the register file. This avoids the problem of pipeline stalls due to instructions waiting for operands from memory. The unfortunate side effect of pipelining is that when an instruction causes a trap it may be difficult to find the exact cause since many instructions are executing in parallel.

Another common feature of RISC designs is the large number of uniformly accessed registers. It is necessary to have enough registers to support function calls without having to spill registers to memory and re-use them. To avoid spilling registers to memory the register file should be large. However, large numbers of registers may complicate the chip design and cause excessive overhead in context switching and trap handling where it is necessary to save them.

4.1.1 RISC Architectures and Operating Systems

RISC microprocessors have been designed for high performance application codes based on integer and floating point requirements. Less attention has been paid to operating systems support in these designs since many of the early RISC designs were based on application address trace analysis. At the same time modern operating systems have been evolving towards more modular designs to improve portability, fault tolerance, and extensibility[8][9]. As a result, modern operating system performance does not scale at the same rate as application performance on RISC machines when compared to CISC[1] machines[1].

[1] **C**omplex **I**nstruction **S**et **C**omputer.

Some of the reasons for this scalability problem are the minimal support for trap handling, the large numbers of registers, and the exposed pipelines that operating system code must manage. Special care must be taken in modern operating systems on RISC machines to optimize or reduce the use of primitive operating system functions.

4.2 The SPARC Architecture Components

The SPARC architecture is based on on the Berkeley RISC designs from the University of California, Berkeley. The general organization of a SPARC computer consists of an **I**nteger **U**nit (**IU**), an optional **F**loating **P**oint **U**nit (**FPU**), and an optional **Co**processor (**CP**). The IU executes all integer arithmetic and logical operations as well as supervisor instructions and contains a small set of instructions to control the CP and FPU. The FPU is an IEEE compatible floating point unit that operates on instructions issued by the IU. The CP has no defined operations and is only an interface description that implementations must follow.

4.3 The SPARC CPU

A function on the SPARC has access to 32 general purpose 32 bit registers. These are divided into 4 groups of 8 register. The global registers, addresses %g0-%g7, always reference the same 8 registers, with %g0 is hardwired to zero for convenience. The rest of the register file depends on the value of the current window pointer (see section 4.3.1). The input registers, addressed %i0-%i7, contain the input arguments from the caller. The convention is to use %i6 as the frame pointer, and %i7 for return address storage. The local registers, addressed %l0-%l7, are free for use by the function. The output registers, addressed %o0-%o7, are used for computation and passing arguments to other functions. The convention is to use %o6 as the stack pointer.

A set of load and store instructions move data between memory and the IU registers with a simple addressing scheme. For operations on specialized memories an alternate **A**ddress **S**pace **I**dentifier (**ASI**) may be specified to these instructions providing a convenient memory like interface to alternate memories and devices.

The SPARC has a 32 bit **P**rocessor **S**tatus **R**egister (**PSR**) containing the state of the processor interrupt level, current register window pointer, the mode and previous mode of the processor, and one bit each to signify that the FPU and CP are enabled.

Finally, the **W**indow **I**nvalidation **M**ask (**WIM**) is a register with one bit per register window signifying which register windows are invalid.

4.3.1 Register Windows

The most unique feature of the SPARC architecture is the use of register windows. Each SPARC processor has a number of overlapping register windows, each representing a view of the total number of registers in the CPU. The input, local, and output registers refer to different actual registers depending on the value of the CWP.

The save instruction shifts the window by 16 registers, causing the previous outputs to be the current inputs, and giving the current procedure 16 fresh local and output registers to work with. The restore instruction increments the CWP and returns to the previous window. By using the save and restore instructions in conjunction with procedure call and return, up to 8 arguments can be passed between functions with a single instruction and no memory references[2].

The register windows are arranged in a circular fashion as show in Figure 4.1. The CWP interacts with the WIM to implement re-use of the least recently used window. The WIM is a

[2]With the conventions of using %i6 as the stack pointer and %i7 as the return address only 6 arguments can be passed directly in registers.

bitmap representation of the window set with one bit set to mark each invalid window. When a save instruction attempts to use an invalid window a window overflow trap occurs (window underflow occurs similarly for restore). System software then flushes one or more windows to the stack and manipulates the WIM to allow the save to continue. This activity is invisible to application programs and is handled by system software.

The register window system on the SPARC CPU acts as a cache of the stack of the running process. A function call on a typical CISC CPU pushes arguments on the stack for the callee. On the SPARC the arguments are moved to the output portion of the register window and the callee shifts the window to gain access to them in its input registers. Memory access is deferred until the window must be re-used.

4.3.2 Traps and Register Windows

When the processor takes a trap it disables all further traps, decrements the CWP without checking the WIM and vectors through a table in the operating system. If the IU takes a trap when traps are disabled the machine resets.

Since the CPU shifts the window without checking for overflow at least one window must be invalid. The trap window will either be clean or it will be the invalid window. If it is the invalid window the input and the output registers are busy as outputs of the previous window and inputs to the next window. This means that only the local registers of the current window may be available to the trap routine to fix the overflow condition and do any preparation to call a system trap handler. Since all other traps are disabled this low level trap code cannot make function calls since they may take overflow traps. The processing must be done in assembler using only the local registers.

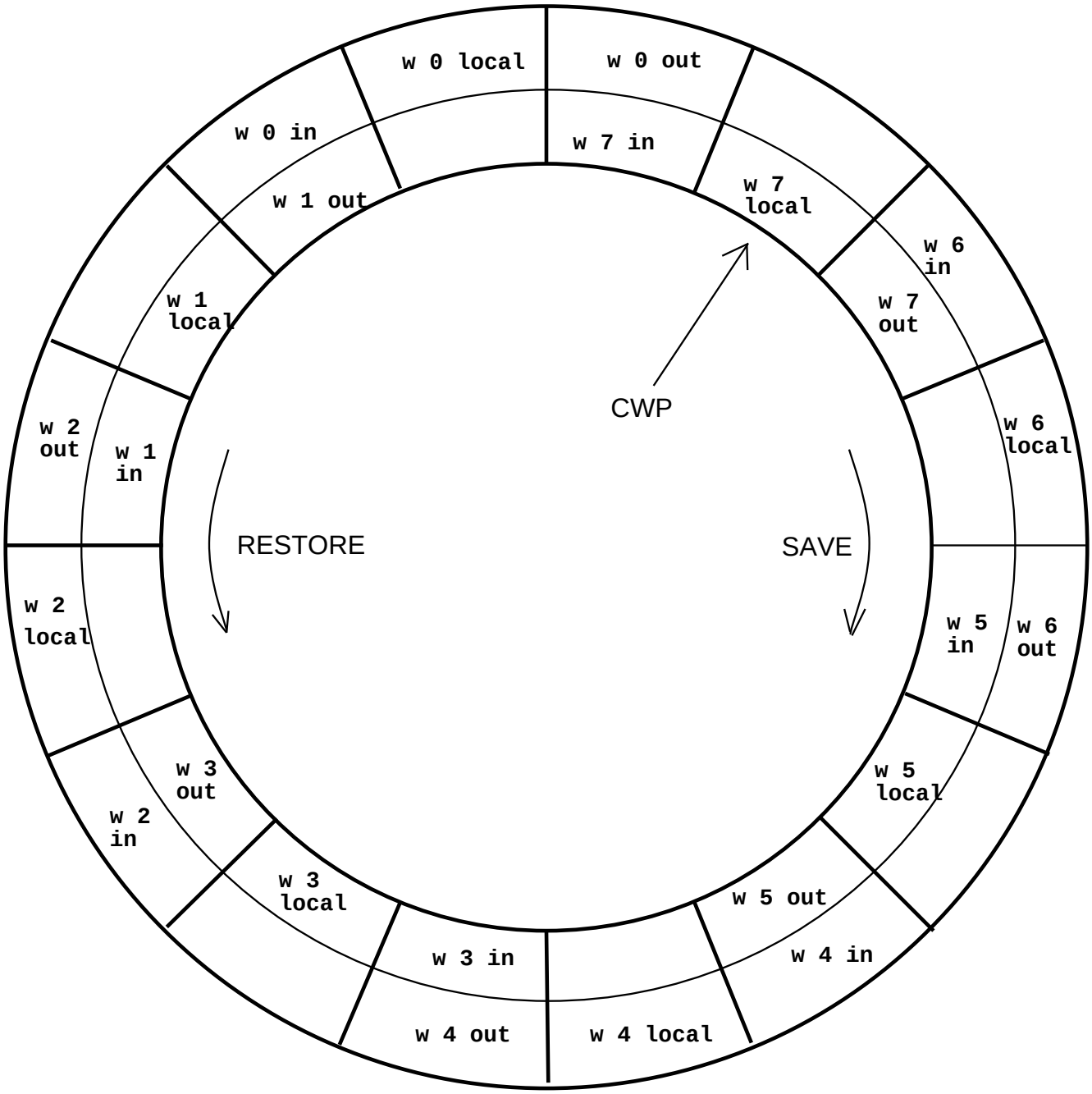

Figure 4.1: The SPARC Register Window Model.

4.4 The SPARC Reference MMU

The SPARC architecture does not require an MMU. It allows designers to use a MMU that is appropriate for their needs, or none at all. However, a SPARC Reference MMU is defined in the SPARC architecture for compatibility between systems. The reference MMU translates 32 bit virtual addresses to physical addresses and supports a 4k fixed page size, page level protections, memory mapped devices, and multiple contexts of address space translation information simultaneously.

4.4.1 Contexts

For efficiency SPARC MMUs can maintain translation information for multiple process address spaces simultaneously. The different translations are referred to as **contexts** and are maintained by the memory management software in the system. By switching the context identifier in the MMU a translation for a different address space is immediately available.

4.4.2 The SPARCstation MMU

The Sun SPARCstation line use a unique MMU design in which the mappings are contained in a memory in the MMU itself instead of in main memory. Mappings are updated by writing to alternate address spaces (see section 4.3) that refer to this memory.

An ASI instruction is used to select one of 8 contexts within the MMU. Each context has a table of 4096 segment pointers that point to **P**age **M**ap **E**ntry **G**roups, or **PMEGs**. A PMEG is a set of page tables for 64 contiguous pages or 256k of memory. All PMEGs in the MMU are shared by the different segment tables. Each of the 64 **P**age **T**able **E**ntries, or **PTE**s, in a PMEG contain a valid, system, writable, cacheable, accessed, and modified bits and a 16 bit

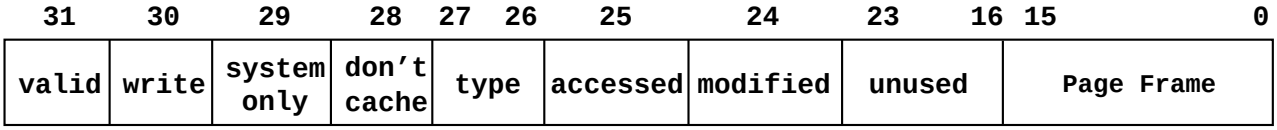

Figure 4.2: The SPARCstation-1 Page Table Entry.

physical page number in addition to a type field specifying a memory address or an I/O address as shown in Figure 4.2.

To translate a virtual address the MMU uses bits 18-29 of the virtual address to select the segment pointer. The offset within the PMEG that the segment pointer points to is in bits 12-17 of the virtual address. The page table descriptor from the PMEG points to the physical page, with bits 0-11 of the address selecting the byte within the page.

Bits 30-31 of the virtual address are required to be identical zeroes or ones, giving two ranges of valid virtual addresses running from 0x00000000 through 0x1fffffff and then from 0xe0000000 through 0xffffffff. Accesses to the virtual address hole are invalid and result in a trap. The SPARCstation 1 MMU is illustrated in Figure 4.3.

4.5 Virtual Address Cache

The SPARC architecture reference allows the use of a virtual address cache in the system. It is specified to be a direct mapped cache, meaning every virtual address maps to one and only one cache line. Because it caches virtual addresses the context and page protection information for the address are stored as part of the cache tag. Since multiple versions of the same virtual address may be cached with different context tags the kernel mappings are required to be identical in all contexts. This allows the cache to ignore the context tag when the reference is from supervisor mode.

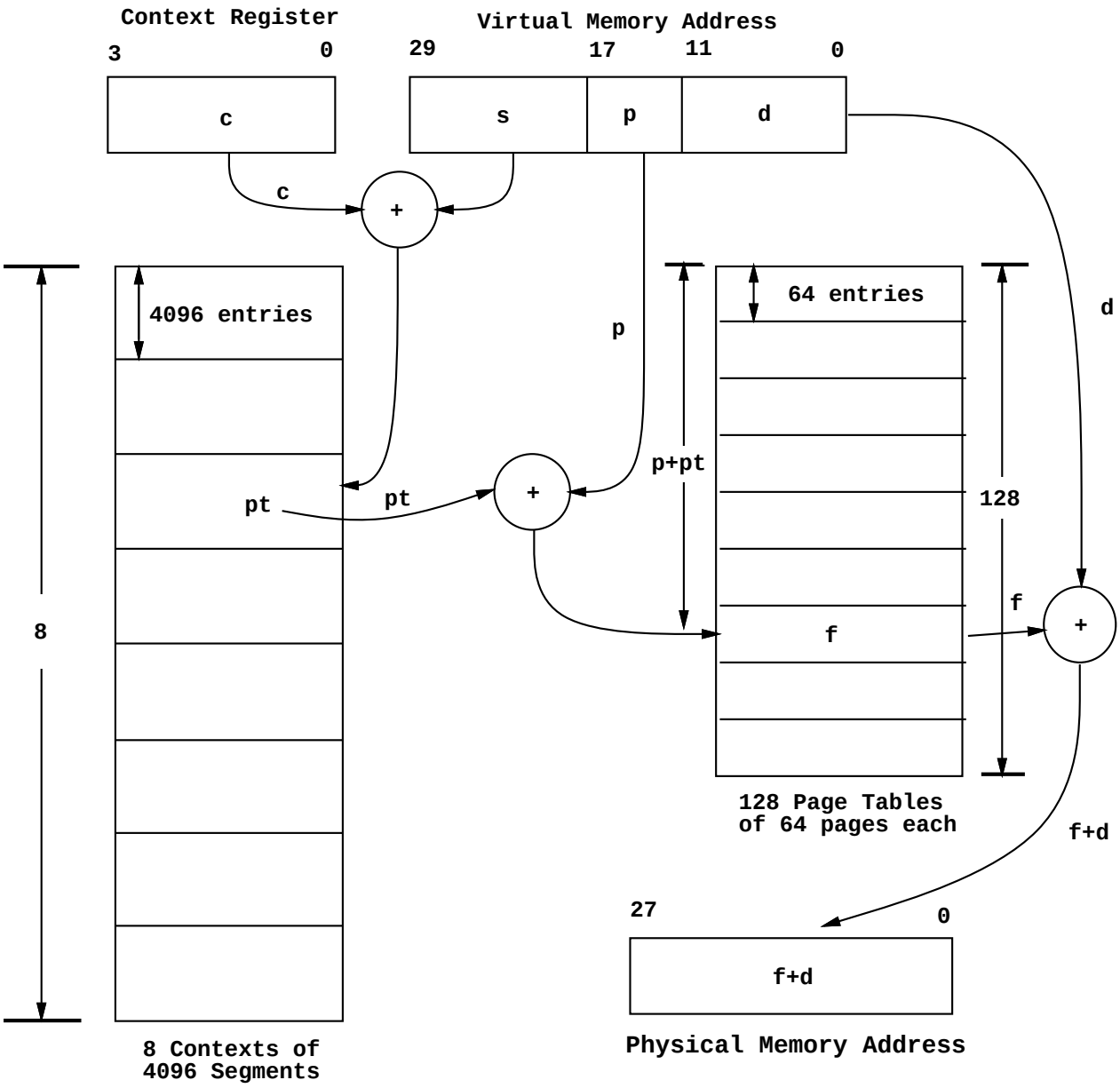

Figure 4.3: The SPARCstation-1 MMU.

Two virtual addresses that reference the same physical address can create an aliasing problem where the two cache lines contain different data for the same memory location. Since the cache is direct mapped this situation can be solved by forcing the addresses to be congruent modulo the cache size, forcing both addresses to map to the same cache line. References from the different addresses then flush, and re-fill the same cache line invisibly. If this method can't be used, system software must mark the page table entries un-cacheable.

4.6 The Sun SPARCstation

The Sun SPARCstation line uses a variety of machine configurations based on SPARC Architecture version 7[3]. The SPARCstation CPU implementations have either 7 or 8 register windows and run at clock speeds ranging from 16 to 40 MHz. The SPARCstation MMUs contain 8 contexts and either 128 or 256 PMEGS. There is an on board boot PROM that initializes the machine, runs basic diagnostics, boots the machine, and hands control to system software. There is a 64k write back cache to improve performance.

4.7 Summary

RISC architecture designs provide simple, fixed length instructions that can be pipelined. Large register files and simple instruction sets in RISC machines favor simple operations found in application code and are not as well suited to modern operating system primitives.

SPARC is a RISC instruction set architecture specification. The version 8 specification is available from SPARC International[11].

[3] SPARC Architecture version 7 is a Sun Microsystems internal standard not available through SPARC International.

The SPARC CPU is based on the register window designs of the Berkeley RISC designs. It has a general register model that uses overlapping register windows as a cache of the stack to pass arguments. The exception mechanism is very simple and works with the register window system and system software for quick exception dispatching to low level support code. The low level support code prepares the machine for higher level exception handling code.

Although the architecture does not require a MMU, a reference MMU is defined for software portability. The SPARCstation MMU is unique in that the translation information is stored in memory within the MMU itself instead of in main memory. The MMU may contain multiple contexts enabling quick process context switching of translation information. The memory system may also provide a virtual address cache that is a direct mapped cache containing context and page protection information as part of the cache tag.

Chapter 5

Choices On The SPARC

This chapter presents the design and implementation of *Choices* on the SPARC architecture and the Sun SPARCstation architecture specifically. Key implementation structures and algorithms that connect machine independent portions of *Choices* to the underlying hardware architecture are presented. Significant class interfaces and implementation details are shown and related to their appropriate *Choices* subsystem frameworks.

5.1 Process System

The *Choices* Process system framework provides machine independent Process management and synchronization code that makes use of concrete subclasses of the abstract class hierarchy ProcessorContext (see section 3.2).

5.1.1 SPARC ProcessorContexts

The ProcessorContext hierarchy for the SPARC architecture parallels the abstract Processor-Context hierarchy as shown in Figure 5.1.

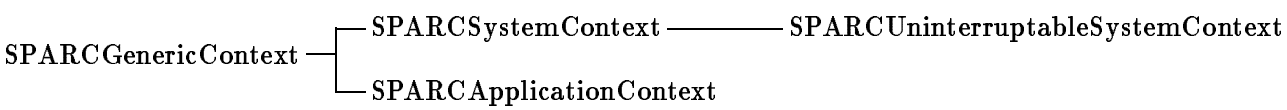

Figure 5.1: The SPARC ProcessorContext Class Hierarchy.

The class **SPARCGenericContext** provides facilities that are generic to all contexts running on the SPARC architecture. This includes storage for a supervisor stack, the processor PSR, global registers, and a single window of registers. It provides methods for construction, and overrides the methods checkpoint, basicRestore, and restoreFromInitialContext.

When the context is constructed the storage areas are used to communicate with the restoreFromInitialContext method by storing the arguments, initial interrupt level, and the entry point of the Process. All SPARCContexts inherit restoreFromInitialContext from SPARCGenericContext. This method prepares an environment resembling a trap from the entry point of the Process at the proper protection level. Returning from the simulated trap starts the Process at the required protection mode of the CPU. Returning from the simulated trap is the only way to set the protection mode of the CPU and the program counter simultaneously.

The class **SPARCSystemContext** is a direct subclass of SPARCGenericContext and provides no additional functionality or storage. It exists as a convenience to the Class system to identify a SystemContext.

The class **SPARCUninterruptableSystemContext** is a subclass of SPARCSystemContext that must have interrupts disabled when it is dispatched. The SPARCUninterruptableSystemContext constructor sets the interrupt level in the storage areas used to communicate with SPARCGenericContext::restoreFromInitialContext and inherits everything else from SPARCSystemContext.

The class **SPARCApplicationContext** adds storage for floating point registers, an application stack for the user Domain, and overrides the checkpoint and basicRestore methods to save the floating point registers if the FPU is enabled. The SPARCApplicationContext checkpoint and basicRestore methods use the SPARCGenericContext checkpoint and basicRestore methods to save all other context other than floating point.

5.1.2 Register Window Management

Choices on the SPARC uses the model that one process at a time makes use of the register windows. Because the register windows cache the stack of the current Process it is necessary to flush them to memory when access to the cached data is needed or to avoid overflowing to a stack in an inactive Domain.

The flush is accomplished by calling n-2 save instructions followed by n-2 restore instructions where n is the number of register windows determined at machine initialization. The flush requires only n-2 saves and restores because the invalid window and the window that context switch is operating in do not need to be flushed.

When processing a window overflow or underflow trap the SPARC CPU resets if a trap occurs (see section 4.3.2). Since the overflow and underflow code flush registers to the stack the system must avoid any memory accesses to the stack that might cause a page fault.

Page faults are controlled by forcing all stacks to be resident and addressable in the SPARCGenericContext and SPARCApplicationContext constructors. The constructors allocate stacks in the proper Domains and use virtual memory primitives to lock down the memory and make it addressable. This greatly simplifies overflow and underflow processing by avoiding the necessary checks and processing of page faults in assembler code.

The price for this simplistic model is physical memory usage by ProcessorContexts even when they are not active. One solution to the problem of memory usage is to allow pageable Application stacks and to guarantee residency and addressability of supervisor stacks when a Process is running. The window management code can then check for and processes a page fault for the user stack using the kernel stack, allowing the save instruction to be re-executed successfully. The details of the solution are presented in section 6.3.1.

5.1.3 Context Switching

When the context switching code is called the CPU is in supervisor mode and running on the supervisor stack of the current Process. The Domain of the switching Process is active and the context switching code does not know if the Domain will change after the switch.

The SPARCGenericContext::checkpoint and SPARCGenericContext::basicRestore methods show the implementation details of context switching on the SPARC. The context switching code is written in C++ with the help of macros that generate assembly language using C statements for data access. These macros are used with the Free Software Foundations Gnu C and C++ compilers [5].

5.1.3.1 Checkpointing The ProcessorContext

The SPARCGenericContext checkpoint method, shown in Figure 5.2, saves the current PSR, disables interrupts, saves the current stack pointer, the current register window's registers, the global registers, and returns zero to the caller. A SPARCApplicationContext checkpoint saves the floating point registers if floating point is enabled in the PSR and calls the SPARCGenericContext::checkpoint to save the rest of the state.

```
Process *
SPARCGenericContext::checkpoint()
{
        /*
         * save psr before we lock everything else out so we restore
         * the previous interrupt level
         */
        unsigned psr;
        asm volatile ("mov %%psr,%0" : "=r" (psr));
        this->psr = psr;

        setInterruptLevel(INTERRUPT_CSWTCH);
        Assert ( interruptLevel() == INTERRUPT_CSWTCH );

        /* save inputs */
        STORE_REG_TO_MEM(%i1, ((unsigned *)this->inputs)[1]);
        STORE_REG_TO_MEM_DOUBLE(%i2, this->inputs[1]);
        STORE_REG_TO_MEM_DOUBLE(%i4, this->inputs[2]);
        /* Gcc needs %fp instead of %i6 */
        STORE_REG_TO_MEM_DOUBLE(%fp, this->inputs[3]);

        /* save globals */
        /* %g0 is hard wired to zero so we use it for %y  */
        asm volatile (" mov %y, %o1");
        STORE_REG_TO_MEM(%o1, ((unsigned *)this->globals)[0]);
        STORE_REG_TO_MEM(%g1, ((unsigned *)this->globals)[1]);
        STORE_REG_TO_MEM_DOUBLE(%g2, this->globals[1]);
        STORE_REG_TO_MEM_DOUBLE(%g4, this->globals[2]);
        STORE_REG_TO_MEM_DOUBLE(%g6, this->globals[3]);

        /* Save stack pointer */
        STORE_REG_TO_MEM(%sp, this->_stackPointer);

        stackRangeCheck(_stackPointer);
        Assert ( ! ((unsigned)_stackPointer & 0x7) );

        return((Process *) 0);
}
```

Figure 5.2: The SPARCGenericContext::checkpoint Method.

5.1.3.2 Restoring The ProcessorContext

The SPARCGenericContext::basicRestore method, shown in Figure 5.3, is the point where the actual context switch occurs. At this point the CPU is running on the supervisor stack of the Process making the switch with interrupts disabled.

The basicRestore method changes to the kernel stack of the new ProcessorContext, restores the PSR, restores the registers for this window, sets the global SPARCGenericContext pointer, and returns the Process passed to it. This causes the last checkpoint of the ProcessorContext to return a non-zero result.

The CPU may be running in a different window than the CWP in the checkpointed PSR. To avoid changing windows the current window is installed in the checkpointed PSR before it is installed as the current PSR.

The global Process pointer is used by the trap handling code to find the kernel stack of the currently executing Process when necessary during low level trap handling (see section 5.2).

5.2 Exception Handling

The *Choices* model for Exception handling (see section 3.2.1) requires the CPU to call raise on the Exception object that has been installed for a particular vector when the CPU takes that exception.

The SPARC CPU model of hardware traps (see section 4.3.2) is very simple compared to CISC CPUs. A CISC CPU implements many low level trap features for operating system support in hardware microcode. Features such as as switching to a system stack and saving the user registers on the system stack are programmed as a low level trap handler on the SPARC. All trap routines except for register window management traps follow the same pattern described

```
void
SPARCGenericContext::basicRestore(Process *p)
{
        setInterruptLevel(INTERRUPT_CSWTCH);
        Assert ( interruptLevel() == INTERRUPT_CSWTCH );

        flushWindows();

        /*  restore old stack so we can take interrupts */
        Assert ( ! ((unsigned)_stackPointer & 0x7) );
        LOAD_REG_FROM_MEM(%sp, this->_stackPointer);

        /* restore psr, we may return to a different window! */
        register unsigned psr;
        asm volatile ("mov %%psr,%0" : "=r" (psr));
        psr &= PSR_CWP; /* psr has our window pointer */
        unsigned newpsr = (this->psr & (~ PSR_CWP)) | psr;
        asm volatile ("mov %0,%%psr" : : "r" (newpsr));
        asm volatile ("nop;nop;nop");

        /* restore globals */
        LOAD_REG_FROM_MEM(%o0, ((unsigned *)this->globals)[0]);
        asm volatile ("mov %o0, %y");
        LOAD_REG_FROM_MEM(%g1, ((unsigned *)this->globals)[1]);
        LOAD_REG_FROM_MEM_DOUBLE(%g2, this->globals[1]);
        LOAD_REG_FROM_MEM_DOUBLE(%g4, this->globals[2]);
        LOAD_REG_FROM_MEM_DOUBLE(%g6, this->globals[3]);

        /* restore inputs, but we don't reload "this" (%i0) until later */
        LOAD_REG_FROM_MEM(%i1, ((unsigned *)this->inputs)[1]);
        LOAD_REG_FROM_MEM_DOUBLE(%i2, this->inputs[1]);
        LOAD_REG_FROM_MEM_DOUBLE(%i4, this->inputs[2]);
        /* Gcc needs %fp instead of %i6 */
        LOAD_REG_FROM_MEM_DOUBLE(%fp, this->inputs[3]);

        /* restore the return value */
        register Process * returnValue = p;
        register unsigned returnSlot asm("%i0");
        asm volatile ("mov %1, %0" : "=r" (returnSlot) : "r" (returnValue));

        SS1ThisProcessorContext = this;

        /* return's from checkpoint() call with Process passed in */
}
```

Figure 5.3: The SPARCGenericContext::basicRestore Method.

here. Window management traps are dispatched directly to the overflow or underflow routine for efficiency.

5.2.1 Low Level Support for Exception Call

When a trap occurs, the SPARC CPU copies the supervisor mode bit to the previous mode bit, sets the supervisor mode bit, disables further traps, and vectors to the trap table where the low level code takes control.

If the trap originated from user mode the low level trap code switches to the supervisor stack of the currently running Process. The CPU state is then saved by storing the PSR, stack pointer, and global registers. Because the hardware does not check for overflow conditions before changing the CWP, the window overflow condition must be repaired if the trap is running in an invalid window.

Now that the kernel stack of the current Process is active and an overflow condition does not exist traps are enabled. With traps enabled the high level handler is called by passing the trap vector and trap stack pointer to SPARCCPU::hardwareInterruptAssist. The hardwareInterruptAssist code looks up the Exception in a table in the CPU and calls basicRaise, passing the trap stack pointer. It is the Exception's responsibility to handle the condition or terminate the Process that took the Exception.

5.2.2 Low Level Support for Exception Return

The Exception return is basically a reversal of the Exception call code, except that it may return to a different register window. When SPARCCPU::hardwareInterruptAssist returns traps are disabled and the old PSR condition codes restored by installing the saved PSR, modified to contain the current CWP in case the trap is returning to a different register window. Because

the trap must return to a valid window the window underflow condition must be repaired if necessary to avoid returning to an invalid window. Finally the saved stack and global registers are restored and the return from trap instruction is executed.

5.3 Memory Management

The *Choices* memory management framework (see section 3.3) defines abstract classes AddressTranslator and AddressTranslation that are subclassed to provide the bottom layers of the virtual memory system implementation. The SPARCstation MMU (see section 4.4.2) has a number of features that are not represented in the *Choices* model of an MMU.

The on chip MMU translation memory requires interaction between the SPARCTranslation and SPARCMMU classes. A translation must exist in the MMU translation memory to be active. Since there are potentially more active AddressTranslations than there are contexts within the MMU the SPARCMMU must implement a replacement policy for AddressTranslations. Since there are a finite number of PMEGS in the MMU they must also be managed with a replacement policy. The MMU is used as a cache of active AddressTranslations, with AddressTranslations aware of the resources they hold in the cache.

5.3.1 The SPARCMMU

The **SPARCMMU** class is a subclass of AddressTranslator and overrides the basicActivate, enabled, and enable methods. Each context in the MMU can cache a single AddressTranslation and each PMEG in the MMU is owned by a single AddressTranslation.

The MMU on the SPARCstation is enabled by the boot PROM and is active when *Choices* gains control of the machine. The SPARCMMU **enable** and **enabled** methods are not used

to enable or disable the chip, but to keep track of the MMU state that *Choices* expects. The *Choices* initialization code (see section 3.4) expects the MMU to be disabled until the first Process is dispatched, when enable is called. This is used as a side effect for taking page faults during system initialization (see section 5.4.2).

5.3.2 Managing Contexts Within The MMU

The **basicActivate** method on the SPARCMMU manages the contexts in the MMU as a cache of SPARCTranslations. It determines if the SPARCTranslation argument is cached, and switches to the proper context if it is. If the translation is not cached, a context must be found to cache the translation information.

The context map in the SPARCMMU has a pointer to the SPARCTranslation for each context that is cached in the MMU. The pointers are initialized to zero to mark unused contexts. When the MMU finds no contexts available to activate a SPARCTranslation the SPARCMMU::findContextVictim method is called which implements the replacement policy to determine which context will be flushed from the cache and re-used. The policy implemented currently is round-robin, although the policy module has been isolated for experimentation with different policies. Alternative replacement policies could keep track of resources used by cached translations and re-use the context with minimal resources, or favor system server process translations over ApplicationProcess translations in a microkernel system with external servers.

The replacement occurs by storing the segment pointers from the context into a table in the SPARCTranslation and noting that it is no longer cached, restoring or initializing the segment pointers for the new translation, and updating the context map in the SPARCMMU.

5.3.3 Managing Page Maps Within The MMU

The PMEG map in the SPARCMMU consists of a pointer to a SPARCTranslation for each PMEG in the MMU that points to the SPARCTranslation that holds the PMEG. A PMEG is held by a single SPARCTranslation, and a translation can hold PMEGs in the MMU when it is not cached. To support invalidating entire segments, a single PMEG in the system is reserved as invalid and all of its page table entries are marked invalid. A segment is invalidated by pointing it at the invalid PMEG.

When a translation that is cached in a context adds a mapping, the segment pointer for the virtual address will point to the invalid PMEG or to a previously allocated PMEG. In the case of a previously allocated PMEG the page table entry for the address is set in the PMEG. If the segment pointer points to the invalid PMEG the PMEG map is consulted to find a free PMEG. If none are available the SPARCMMU::findPMEGVictim method is called which implements the PMEG replacement policy.

PMEG replacement in the MMU is complicated since the translation holding the PMEG may not be cached in a context within the MMU during the replacement. In a cached translation the segment table in the translation's context must be scanned, invalidating references to the PMEG. If the translation is not cached the stored segment pointers referencing the PMEG must be invalidated.

5.3.3.1 Virtual Address Cache

Because the SPARCstation virtual address cache contains translation information as part of the cache tag (see section 4.5) the cache must be flushed when mappings are changed within

Benchmark	Cache Disabled	Cache Enabled
SystemProcess Context Switch	315	32
Reference count (increment+decrement)	136	10
Lock (acquire+release)	63	4
Binary semaphore (P+V)	302	23
Null Proxy Call From Application	405	49

Table 5.1: Virtual Address Cache Effects on Basic *Choices* Benchmarks.

the MMU. *Choices* on the SPARC makes use of the MMU::flushTLB method, which is called when mappings are updated, to flush the virtual address cache.

SPARCMMU::flushTLB contains calls to flush the virtual address cache that can be compiled with or without cache support for testing and performance measurements. Memory references from the cache take 2 processor cycles while cache misses take 14 cycles. Enabling the virtual address cache results in approximately a tenfold performance improvement. Cache effects on some basic system benchmark results are shown in Table 5.1.

Since *Choices* does not support the notion of cacheable MemoryObjects yet (see section 6.3) sharing of cached MemoryObjects between Domains requires specifically placing shared objects at the proper alignments to avoid cache aliasing problems (see section 4.5). With the current *Choices* virtual memory system this placement is performed explicitly by the programmer.

5.3.4 The SPARCTranslation

The *Choices* model of an AddressTranslation is that of an in-memory translation table that the memory management software manages and the MMU reads (see section 3.3.2). Since the SPARCstation MMU contains its own translation memory the SPARCTranslation class must be managed differently on the SPARCstation. A SPARCTranslation can have one of two states, cached in the MMU or not cached. Each call to update, add, or remove translation information proceeds differently depending on this state.

5.3.4.1 Cached SPARCTranslations

When the SPARCTranslation is cached the addMapping, removeMapping, and changeProtection methods operate directly on the MMU by manipulating the context containing the translation. In the case of addMapping a PMEG may need to be allocated to map a new segment. In the case of removeMapping the PMEG may be returned to the pool of free PMEGs in the MMU if it no longer maps any pages. In the case of changeProtection a PMEG may need to be allocated to set the page table entries to the requested protection.

5.3.4.2 Non-Cached SPARCTranslations

When a SPARCTranslation is not cached, calls to addMapping and changeProtection can be ignored since the Domain has complete machine independent information about the mappings for the AddressTranslation. When the SPARCTranslation is made active it will be cached in the MMU and any faults due to ignored calls will result in the Domain updating the translation information in the now cached SPARCTranslation.

If a SPARCTranslation that is not cached receives the removeMapping or changeProtection message it must assure that if it is cached at a later time the addresses for which the call was made are mapped at the proper protection level. This is a problem because the PMEGs that the non-cached SPARCTranslation holds must be updated while the SPARCTranslation is not in the MMU.

One solution is to free the PMEG that the stored segment pointer references and invalidate any segment pointers pointing to the PMEG. When the SPARCTranslation is later cached, accesses to these addresses will fault and the Domain will rebuild the mappings at the proper

protection level. This scheme invalidates entire segments which may be more than necessary, although it is simple and easy to implement.

The solution implemented in the current version of *Choices* on the SPARCstation leaves the segment pointer in place and operates on the PMEG that it points to. The PMEG pointed to is located from the stored segment table. To operate on the PMEG the MMU requires a segment pointer that points to the PMEG to be active in the MMU. The SPARCTranslation class provides this by keeping a spare segment pointer at a well known address that can be borrowed by non-cached SPARCTranslations. Non-cached SPARCTranslations use this spare segment pointer to update PMEGs that they own in the MMU.

5.3.4.3 Protection Model

Choices provides four protection levels of virtual memory (see section 3.3). The SPARCstation implements a system bit and a write bit. The protection levels NoAccess and ReadWrite are implemented directly by the SPARCstation hardware. However, the ReadOnly protection level cannot be directly implemented in the SPARCstation MMU page table entry.

ReadOnly (see Table 3.1) must allow system write capability and user read capability. A page with system write capability must have the system and write bits set in the page table entry. An application read only page must have the system bit and the write bit clear.

To implement the ReadOnly protection level the SPARCTranslation changeProtection method determines from the MMU translation memory if the page table entries already exist. If the mappings exist and are at one of the states corresponding to ReadOnly the page table entry is toggled to the other state. The initial state for ReadOnly mappings is system write, favoring Kernel performance.

5.3.4.4 SPARC Memory System Optimizations

The SPARCstation memory management system can use features of the *Choices* memory model (see Figure 3.2) to optimize the management of the SPARCMMU and SPARCTranslations.

In *Choices* the Kernel mappings are identical in every AddressTranslation and the kernel address range is disjoint from the application range. The management of the SPARCTranslation for the Kernel can be optimized by performing AddressTranslation operations directly on all contexts in the MMU without checking to see if the SPARCTranslation is cached. This frees all contexts in the MMU to be used for Application contexts with the Kernel mappings existing in the non-application portion of the address space. The SPARCTranslation that implements the AddressTranslation for the Kernel Domain is easily identified as the first SPARCTranslation created in the system.

When a SPARCTranslation must be replaced in the MMU the replacement of the segment pointers can be optimized because only the segment pointers for the Application address range need to be saved or restored. The SPARCstation virtual memory layout (see Figure 5.4) allocates less than half of the virtual address space to user processes saving over half of the segment swaps necessary without this optimization.

5.4 Machine Initialization

5.4.1 SPARCstation Bootstrap

The boot PROM in the SPARCstation provides a bootstrap routine as well as basic debugging facilities. The PROM bootstrap can boot an executable file from a local disk, over the network,

or over a serial line. The bootstrap loads the kernel into memory, maps it in the MMU, and begins executing it at the entry point.

5.4.2 SPARCMMU Initialization

Choices expects the machine to be running in physical memory with the MMU disabled. The boot PROM leaves *Choices* running in 1-1 mapped memory with the MMU enabled requiring special initialization. Since the MMU is active, page faults may occur during system initialization due to the boot allocator allocating memory past the end of the kernel (see section 3.4). Although Exceptions are installed in the CPU, the page fault Exception handler will find no Process and no Domain to operate on.

To detect this condition the page fault handler code checks to see if the MMU has been enabled. The MMU will report that is is not enabled until the first process is dispatched, so this is a reliable method to determine if a trap occurs during the boot sequence. If the MMU has not been enabled the Exception dispatch code calls a special boot sequence fault handler which adds the 1-1 mapping into the translation memory in the MMU.

5.4.3 SPARCTranslation Initialization

The SPARCTranslation::init method is used to initialize the maps and translation information within the MMU. The mappings for the kernel are determined by allocating memory from the boot allocator to get the top 1-1 mapped address so far. The top address is increased to the next page boundary and all mappings below that are installed to map virtual addresses to identical physical addresses and mappings for all other addresses are made invalid. This is first done in context 0 and then copied to other contexts in the MMU.

The procedure to duplicate the segment tables in another context relies on the copy code being primed into the cache so that the current context can be changed without mappings existing in the new context. First, the virtual address cache must be completely flushed, invalidated, and enabled. The copy code is cached by copying the running context to itself. Once the cache is primed the context register can be set to the new context and the new context copied by the cached code. This requires the copy code to be in-line assembler to avoid re-filling the cache lines containing the copy code. Once this process completes all contexts are running with identical mappings and the virtual address cache is enabled.

5.4.4 Additional Virtual Memory Initialization

The SPARCstation has a hole in its virtual address space (see section 4.4.2). To assure that Domains don't try to insert MemoryObjects in this hole it is made part of the Kernel VM space and a DummyMemoryObject the size of the hole is inserted in its space. The DummyMemory-Object assures that the space is taken and that an error is reported for accesses in that address range. Since it is in the Kernel Domain it is guaranteed to be shared by all *Choices* Domains.

The SPARCstation has a number of memory-mapped devices and devices that do **D**irect **V**irtual **M**emory **A**ccess, or **DVMA**, to accomplish I/O. Since DVMA operates in context 0 and the kernel can be in any context when an I/O interrupt occurs the dynamic DVMA mappings are duplicated in all contexts in the DVMA address range. At initialization the entire DVMA range is set to invalid. The memory mapped device registers are mapped into context 0 in the assembler boot code before the Kernel initialization begins and are copied to all contexts as part of the machine dependent virtual memory initialization.

The result of the virtual memory initialization on the SPARCstation is shown in Figure 5.4.

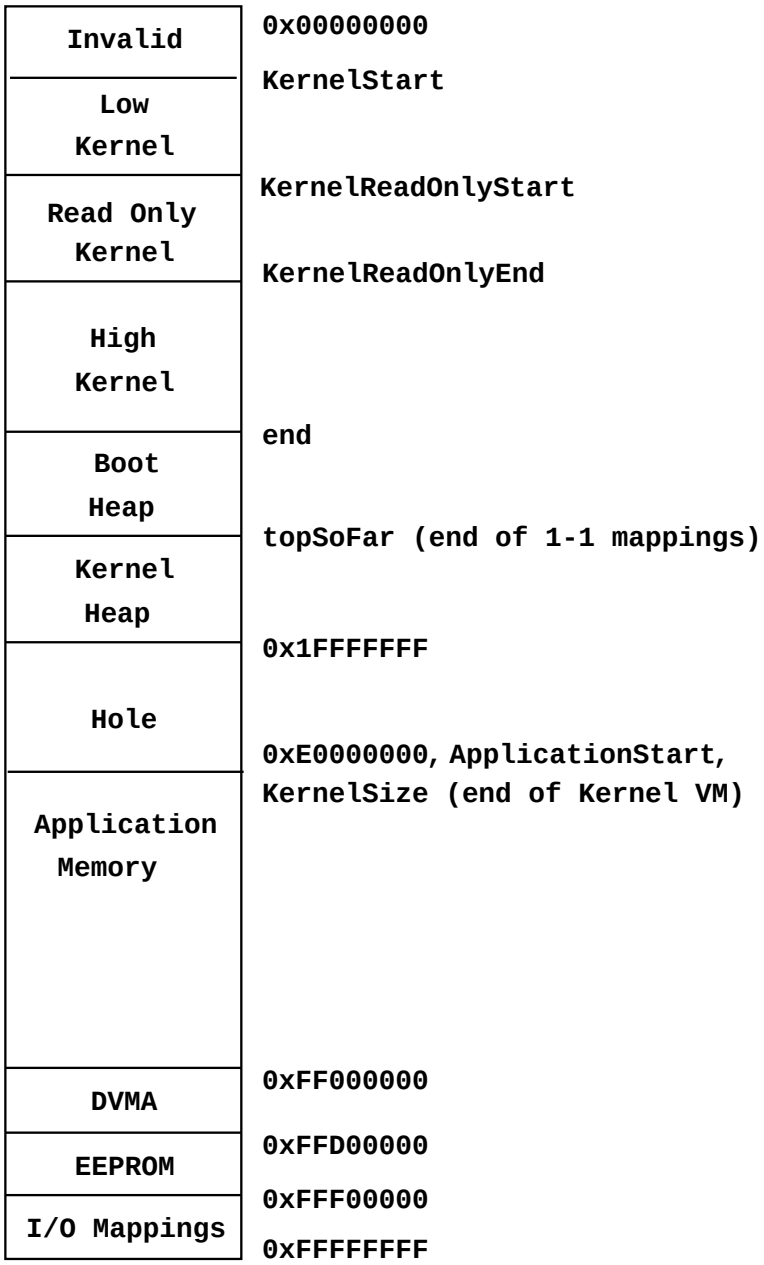

Figure 5.4: The SPARCstation Virtual Memory Layout.

```
void
SS1EthernetConfig()
{
        /*
         * 1. Create the interrupt process and install it in the CPU.
         * 2. Create a new driver le0.  The constructor causes an interrupt.
         */
        Process * p = new InterruptProcess ( Am7990InterruptProcessEntry,
         thisProcess()->domain(), normal, 21);
        Assert (p != 0);
        p->setName("#Am7990InterruptProcess");
        p->ready();
        thisProcess()->relinquishProcessor(); // get it running

        Assert (le0 == 0);
        le0 = new SS1Am7990(LE0_RAP, LE0_RDP);  // Constructor initializes
        Assert (le0 != 0);
}
```

Figure 5.5: SPARC Am7990 Ethernet Configuration Routine.

5.5 Device Drivers

Choices on the SPARC uses SystemProcesses to implement interrupt handlers for device drivers. Drivers written in this way must establish linkage between the Exception and the Process. This is typically done at driver initialization time. The initialization code creates the Process that will service interrupts, starts it running, and completes the initialization. The Process created is an InterruptProcess, which is a form of UninterruptableProcess. This is required so that the context switch does not re-enable interrupts and re-take the interrupt it is servicing. The configuration code for the SS1Am7990 ethernet controller is shown in Figure 5.5.

The interrupt service Process creates an InterruptException object which binds the trap vector to the Exception. The InterruptException contains a special Semaphore that gives the CPU to the driver Process directly when V'd. The driver calls await on the Exception, which

```
void
Am7990InterruptProcessEntry()
{
        AwaitedInterruptException * interrupt =
         new AwaitedInterruptException();
        thisCPU()->setException( 21, interrupt);

        while (1) {
         Debug << thisProcess() << ": awaiting interrupt\n" << eor;
         interrupt->await();
         Debug << thisProcess() << ": interrupt\n" << eor;

         Assert (le0 != 0);
         SPARCTranslation::dmaFlush();
         le0->interruptHandler();
        }
}
```

Figure 5.6: SPARC Am7990 Ethernet Interrupt Process.

P's the semaphore and the InterruptException::basicRaise method V's the Semaphore. The InterruptProcess code for the SS1Am7990 ethernet controller is shown in Figure 5.6.

Interrupts on the SPARCstation persist for the duration of the condition that causes them, imposing some restrictions on interrupt handlers. First, the interrupt handler services the device registers to clear the interrupt, storing away necessary state from the registers. Once the interrupt is cleared the driver is free to perform higher level functions, for example Process switching due to a time slice interrupt. If the interrupt were cleared at the end of the routine a Process switch might re-enable interrupts on the CPU causing the persisting interrupt to be re-taken. The example shown in Figure 5.7 is the TimerManager interrupt handler which manages periodic and timeout timers with a single hardware clock.

```
void
SS1TimerManagerException::interruptHandler()
{
    Debug << "TimerManagerException::basicRaise()\n" << eor;
    Assert( this != 0 );
    Assert ( TimerManagerLock != 0 );

    /*
     * reset the and limit reached bits and clear the interrupt
     */
    limitval = *LIMIT0; // dummy read to clear interrupt.
    FORCE_USE(limitval);
    Assert (! (*COUNTER0 & LIMIT_MASK));

    /*
     * We get some timer interrupts before the Managers are completely
     *   initialized so need to make sure the pointers are non-null
     *   before calling the handlers
     */
    if (TimeoutManager != 0) TimeoutManager->interruptHandler();
    if (PeriodicManager != 0) PeriodicManager->interruptHandler();
    thisCPU()->timesliceTimer()->tick();

    Debug << "TimerManagerException::basicRaise() returning \n" << eor;
}
```

Figure 5.7: SPARC TimerManager Interrupt Handler Routine.

5.5.1 Devices and DVMA

The SPARCstation uses DVMA for I/O which operates in context 0. Since the MMU can be in any context when a device interrupt is taken, the system must map I/O memory into all contexts and avoid cache aliasing problems. The SPARCTranslation::maptoDMA and SPARCTranslation::unMapDMA methods encapsulate locking and converting memory to DVMA compatible addresses in all contexts.

The mapToDMA method locks down the pages and mappings for the source address, flushes the virtual address cache for the source, and copies the segment pointers for the source address into the DVMA range. It returns an address with the same segment offset in the DVMA range that maps the same pages (because it points at the same PMEGS) in all contexts.

The unMapDMA routine frees the mappings for re-use and flushes the virtual address cache at the DVMA address. Device drivers map addresses to DVMA before I/O and unmap them afterwards. The DVMA address range is in high memory on the SPARCstation (see Figure 5.4).

5.5.2 Supported Devices

SPARC *Choices* supports the following on board and Sbus devices:

- On board Zilog Z8530 serial ports.
- Sbus NCR 53C90 SCSI interface.
- On board AMD Am7990 ethernet.
- On board microsecond counter/timers.
- On board L64853 DMA controller.

5.6 Debugging Support

Operating systems running on top of bare machine hardware are notoriously difficult to debug. The size and complexity of the software are not the only problems. Context switching, interrupts, virtual memory, and critical sections sensitive to timing make debugging operating system code much more difficult than typical application debugging.

Choices on the SPARC adds support for the Gnu gdb debugger running on a remote host. To support remote debugging two additions are made to the kernel. First, one of the serial ports is used to communicate with debugging stubs in the kernel that read and write memory in response to commands sent over the serial line by the debugger process running on a UNIX host. Second, the trap handler must recognize debugger traps and pass control to a debugger trap routine which reads the machine state saved at trap time into a buffer used for communication with the remote debugger.

With remote debugger support kernel data structures can be viewed and changed, breakpoints can be set, and stack frames can be traced to find the execution path leading to problematic conditions. Once the scope of the search for a bug is narrowed, only breakpoints need to change and this does not require recompilation. Critical sections can be debugged more easily, as can complicated events such as context switching. Breakpoint code has been added to the Assertion failure code in the system so the debugger can gain control and locate the source of the failure easily.

5.7 Summary

Choices runs on the bare hardware of the SPARCstation and provides the standard *Choices* subsystems. By encapsulating machine and processor dependencies within the subclasses to

hide implementation details of the SPARCstation the subsystem frameworks of *Choices* are re-used with minimal modifications.

The SPARC register window system is used as a cache for the stack of the running process. The register window system requires special maintenance for context switching and trap handling.

The low level assembler support code for trap handling is complex due to the interactions of register windows, traps, and the Process system. Many operations that a CISC CPU would perform in microcode are accomplished in the low level trap assembler on the SPARC.

The unique memory management hardware on the SPARCstation is used as a cache of active AddressTranslations. Policy module methods for the management of the fixed MMU resources are isolated for customization and experimentation. Memory management is optimized using features of the *Choices* memory model. The virtual address cache is supported, which provides a ten fold performance increase over a non-cached system for basic *Choices* benchmarks.

Choices supports the majority of on board devices on the SPARCstation, and includes support for gdb debugging.

Chapter 6

Experiences With *Choices* on the SPARC

This chapter presents the experiences of porting *Choices* to the SPARC architecture. Relevant issues to C++, *Choices*, and the SPARC architecture are discussed and improvements to *Choices* are presented based on experience with the SPARC and other architectures that *Choices* runs on.

6.1 C++

Using C++ to implement a system as large as *Choices* inevitably points out strengths and weaknesses in the language. The main strengths in C++ are the strong typing and efficient dynamic binding. These provide early (compile time) error checking and an efficient run time system.

The main weaknesses are the languages immaturity, the lack of run time information about classes, and the inability to revoke methods in superclasses.

The lack of run time type information is a problem since every additional class must be added to the type system. Although much of the work can be automated the programmer must still provide basic information which can be error prone in a large system such as *Choices*.

The immaturity of C++ has been troublesome because language features can't be used until they are supported by the wide range of compilers that the different *Choices* platforms use. One such feature is is language based exception handling. Using a well structured C++ exception handling mechanism within the Kernel will allow removing many lines of error checking code, improving performance and readability.

The most troublesome feature missing from C++ is method revocation or invalidation. When subsystems undergo design changes, or when prototyping a new subsystem, method signatures in superclasses tend to change. The problem is that old signatures in subclasses are difficult to locate. C++ allows the subclass to overload anything regardless of what is in the superclass. Old signatures become methods local to the subclass that are never called, resulting in unnecessary source code and executable size. Pure virtual functions allow the superclass to require implementation of certain methods. A similar feature to require that subclasses don't implement certain method signatures would be easy to implement using the pure virtual method syntax with a number other than 0.

6.2 The SPARCstation Architecture

The SPARCstation architecture presented numerous challenges in implementing *Choices*. Many of the architecture features are significantly different than features of architectures that *Choices* was designed for, and some of these features interact in subtle ways, complicating the implementation of *Choices* on the SPARC.

6.2.1 Register Windows

The management of register windows (see section 4.3.1) is a complicated issue due to a number of factors. The register windows interact with the stack, which in turn interacts with the virtual memory and Process switching systems. The window management code must be written in assembler and must be properly implemented to allow even simple function calls to work properly. Additionally, parts of the window trap management code must be duplicated in the system trap routine due to the operation of traps on the SPARC CPU (see section 4.3.2).

To process register window traps, the system must guarantee that the memory for the stack cached in the register file is available when the windows are flushed. Thus, the stack must be resident and accessible when the window traps occurs. To make efficient use of memory, stacks should be pageable. The problem with pageable stacks is that a window trap can't take a page fault with traps enabled. C code depends on register windows for function calls precluding calling the page fault handler until the window is saved. However, the window can't be saved until the page fault handler is called.

One simple solution is to lock down all stacks, make them addressable when they are created, and flush the register windows when changing Domains. This simplifies the overflow and underflow code by guaranteeing the availability of stacks at the expense of memory and flushing when not always necessary. This is the method used in current implementation of *Choices* on the SPARC.

A better solution is to have the system guarantee that the supervisor stack for a running Process is resident in a machine independent fashion and to allow user stacks to be pageable. The register window handlers then must check the availability of the stack only if the CPU was in user mode when it took the trap. If the CPU was in user mode and the stack is not

resident the window trap routine switches arguments to simulate a page fault at the stack address and jumps to the system trap dispatcher. The system trap routine switches to the kernel stack, saves the window to the kernel stack due to overflow, and processes the page fault. Upon returning from the trap the save instruction that overflowed and caused the page fault is re-executed and proceeds normally. The change to the window trap code to implement the check and simulate the page fault is simple and has been implemented. The changes to *Choices* machine independent systems to support pageable stacks are presented in section 6.3.

6.2.2 Alternate Register Window Models

The current implementation of *Choices* on the SPARC uses the register window system to cache a single Process at a time. There are other models that can be applied to the register window model, including using only a single window, using a window per Process and sharing the windows between user and supervisor.

The problem with implementing different register window schemes is that the register window model is implemented in pieces in the compiler, the window trap handlers, the system trap handler, and the context switching code. Each scheme requires modification of some or all of these areas to implement. Unfortunately these are the most difficult areas of the system to change, making this kind of experimentation difficult.

6.2.3 Trap Support

RISC processors provide minimal support for trap handling and the SPARC contains this feature[1] as well. The simplicity of the machine puts the responsibility on system software to provide features that make the machine usable, such as user, supervisor, and pageable stacks.

[1]This is a feature from the hardware designer's point of view. It is a problem for operating system designers.

Since this functionality is assumed to be provided at the hardware level by the operating system model, it is implemented at the lowest level possible which is in the low level trap code.

On the SPARC the low level trap code is already complicated by the interaction with the register windows (see section 4.3.2). This code must check the condition of the register windows to determine whether or not there is a condition that must be handled. This is a significant overhead in the SPARC trap handler and contributes to the lack of operating system primitive performance on the SPARC. The addition of the low level Process and Exception model code to the trap handler results in a large amount of complex assembler code that interacts with other parts of the system.

6.2.4 Memory Management

Because the number of contexts is fixed, translations must be swapped in and out of the MMU by the CPU when the number of address spaces exceeds 8. The fixed size context tables maintained within MMU memory require a fixed overhead to swap regardless of the characteristics of the address space. The fixed number of PMEGS requires swapping them in virtual memory intensive systems. The overhead of swapping MMU resources with memory can be play a significant part in context switching when there are more than 8 applications active. In the current *Choices* implementation the context swap requires about 2000 MMU and memory references to perform.

Because the CPU must use ASI instructions to maintain the translation information within the MMU it is not possible to construct a translation table shared by multiple MMUs in a shared memory multiprocessor containing this type of MMU. It is possible to have each CPU perform the operation by sending interrupts to all processors, but this method is inefficient. It

is interesting to note that the MMU in the Sun 600 series multiprocessors is the Cypress 605 which uses main memory translation tables and supports 4096 contexts.

6.2.4.1 Virtual Address Cache

The virtual address cache enhances performance dramatically (see Table 5.1) but requires unique management to use properly. It is not clear that a virtual address cache outperforms a physical memory cache and the maintenance overhead is non-trivial. The special case of supervisor mode ignoring parts of the tag and the cache flush requirements when doing DVMA push cache concerns into areas such as device drivers where they don't belong. The aliasing problems put special requirements on memory management code to align memory properly or disallow cacheing. A physical address cache would likely provide equivalent performance improvements without the maintenance overhead.

The problems with the virtual address cache can be seen in UNIX context switching, where structures at fixed addresses are used to implement a process. When a context switch occurs the structures are reloaded with the new processes data. Since the virtual address cache is active this area must be flushed before reloading. In SunOS this is the most expensive part of the context switch, accounting for about 75% of the time[7]. Since *Choices* does not use this method for context switching its performance in this critical area is not degraded by the need to flush the cache.

6.3 Choices

Choices is indeed a portable system. The frameworks have been directly re-used on the SPARC with minimal change. However, there are a number of areas in which *Choices* can benefit by design modifications based on this experience.

6.3.1 Process and Virtual Memory System Interactions

Choices currently makes no assumptions about the implementation of subclasses of ProcessorContext. However, there are features common to each subclass of ProcessorContext that can be put into the machine independent framework. Each ProcessorContext has a stack for supervisor use. Each ApplicationContext also has a stack for application mode use. The supervisor stack is used for trap processing and must therefore be resident and accessible at all times that the Process is running. Unfortunately this responsibility is duplicated in each subclass.

The Process system should contain these general features and is undergoing modification along with the virtual memory system to support the *Choices* model in processor and machine independent code. The allocation of stacks is being moved into the processor independent ProcessorContext class. This will reduce the amount of processor dependent code and unifies the *Choices* model in the framework rather than in the subclasses.

The Process switching code is being modified to make virtual memory system calls to guarantee that the supervisor stack is locked down and addressable before dispatching the Process. Locking down the supervisor stack only when the Process is running avoids memory and translation use by idle Processes and allows application stacks to be pageable. When the Process is removed from the CPU the supervisor stack memory and mappings can be released.

This support is machine independent and can be implemented efficiently with proper virtual memory support to favor supervisor stacks of active Processes.

6.3.2 Virtual Memory

The *Choices* virtual memory system was designed to support sharing and customizable paging policies. Noticeably absent are basic primitives to support attributes for the stack locking requirements of the new Process system.

There are various static and dynamic attributes that can be placed on a virtual address range. The protection level, cacheability, and locking of physical memory pages and translation tables are examples. A general and extensible attribute system is required to implement the current requirements and support future requirements. With such an attribute system static attributes can be stored in the Domains and dynamic attributes can be passed to the virtual memory framework where necessary.

The immediate requirements are to put restrictions on the implementation of AddressTranslations to implement guaranteed addressability. AddressTranslations, as currently implemented, may discard mappings in their translation tables when they need to re-use the memory for other translation tables. It is necessary in the new Process system and elsewhere[2] to guarantee addressability by having the AddressTranslations lock down ranges of mappings. This can be done by passing a lock mappings attribute to the AddressTranslation on the address range.

Currently the Domain only aligns MemoryObjects to page boundaries. Shared MemoryObjects must be explicitly placed on cache size boundaries or the Domain must have knowledge of alignment for shared MemoryObjects. The SPARCstation port of *Choices* will use a cacheable

[2]Device drivers using DVMA on the SPARC for example.

attribute when mapping a MemoryObject. This will align the MemoryObject in the Domain to an offset modulo the cache size to avoid aliasing (see section 4.5).

6.4 Summary

The *Choices* port to the SPARCstation uncovers opportunities for improvement in C++, the SPARC architecture, and *Choices*. Experience gained from the SPARC port can be used to strengthen several *Choices* frameworks resulting in reduced machine dependent code and the unification of common implementations within the machine independent portions of *Choices* frameworks.

C++, although efficient, lacks maturity and features that would make maintenance easier. Access to class information at run time could be implemented by the compiler to avoid errors. Method revocation would be very useful in rapid prototyping environments with many classes to locate and enforce interfaces changes. Language supported exception handling would allow cleaning up inefficient and difficult to read pessimistic error checking code.

The SPARCstation architecture contains unique interacting features which complicate the *Choices* implementation. The register window system interacts with the operating system process model and the hardware trap model leading to large amounts of complicated assembler code in the implementation and difficulty experimenting with alternate register window models. The MMU model is difficult to manage and is not desirable for multiprocessor applications because of the internal memory. The virtual address cache hardware requires maintenance code in the translation management code and unintuitive places such as device drivers. A physical address cache might provide similar performance without such complications.

Chapter 7

Conclusions

In this thesis I described the *Choices* operating system for the SPARC architecture. The SPARC architecture provided unique opportunities to evaluate *Choices* and the SPARC architecture. There are a number of conclusions that result from this evaluation.

Structuring architecture dependent framework interactions with a small set of general methods is a good way to isolate the architecture from the low levels of the operating system. The encapsulation allows the architecture features to be used without affecting higher level code. For example, the *Choices* memory system on the SPARCstation shows that encapsulating the MMU hardware as well as the translation tables is a good idea. By implementing the framework interactions with a small set of general methods the complexities of the SPARCstation memory management hardware can be managed efficiently without affecting the machine independent parts of the framework.

The implementation of the Process system on the SPARC reveals common features of all *Choices* Process implementations that should exist in the machine independent portions of the framework. The interactions between the process and virtual memory systems to support

pageable application stacks and guaranteed addressable supervisor stacks for running processes should be maintained in machine independent code. This would make the *Choices* Process model machine independent and reduce the amount of machine independent code.

The interactions of the features of the SPARC architecture make low level operating system code complex. The register window management code, which relies on the Process model and the compiler, must be working properly before even simple function calls can operate. Changing the model requires changing some or all of these pieces, making experimentation difficult. Much of this support code must be written in assembler, making the task of modifying it difficult.

The SPARCstation MMU is difficult to manage because of the small number of fixed resources and the minimal page protection implementation. The MMU must be managed as a cache with replacement policies in software, creating overhead in the management code. The SPARCstation page protection bits require additional overhead to implement the *Choices* Read-Only protection state.

The register window implementation on the SPARCstation does not show any particular performance advantage or disadvantage. The extra overhead in trap handling and context switching to deal with the register windows is offset by the benefit of fast procedure calls in code that runs without overflowing the register windows. Evaluating a system with register windows using micro-benchmarks may not produce accurate results for this reason.

The SPARC virtual address cache requires extra overhead to support it's write-back model and the cache aliasing problem. To support memory sharing in *Choices*, care must be taken to align virtual memory regions properly or disable the cache. It is not clear that a physical memory cache could not provide similar performance improvements without the additional overhead.

7.1 Further Work

We[1] are in the process of implementing the framework changes presented in section 6.3 to support Process and virtual memory interactions in machine independent code.

The SPARC port of *Choices* continues to undergo refinement. The low level memory management code is undergoing redesign to eliminate unnecessary dependencies between the MMU and AddressTranslation codes that exist. Instrumentation is being added the SPARC port to allow evaluation of the system. Finally, a port to the Sun SPARC 600 series of multiprocessor machines is underway. This machine is based on SPARC Architecture version 8 and has hardware support for locking. I expect to use most of the SPARCstation code without change.

7.2 Acknowledgements

I would like to thank Sun Microsystems for help and encouragement in working with their machines. I would like to thank the National Science Foundation for funding my salary through the Tapestry project. I would like to thank the Department of Computer Science at the University of Illinois for providing a great environment in which to work. Finally, I would like to thank Roy Campbell for continued support and encouragement.

[1]The *Choices* development group at the University of Illinois.

Bibliography

[1] Thomas Anderson, Henrey Levy, Brian Bershad, and Edward Lsowska. The interaction of architecture and operating system design. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1991.

[2] Roy Campbell, Gary Johnston, and Vincent Russo. Choices (Class Hierarchical Open Interface for Custom Embedded Systems). *ACM Operating Systems Review*, 21(3):9–17, July 1987.

[3] Roy Campbell, Vicent Russo, and Gary Johnston. The design of a multiprocessor operating system. Technical Report UIUCDCS-R-87-1388, University of Illinois at Urbana-Champaign, Department of Computer Science, December 1987.

[4] Robert Dewar and Matthew Smosna. *MicroProcessors, A Programmers View.* McGraw-Hill, 1990.

[5] Free Software Foundation, 675 Mass Ave, Cambridge, MA 02139. *Using and Porting GNU CC*, 1992.

[6] Ralph E. Johnson and Brian Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, pages 22–35, June 1988.

[7] S. R. Kleiman and D. Williams. Sunos on sparc. *Sun Microsystem Technical Report.*

[8] Keith Loepere. *Mach 3 Kernel Principles*, March 1991.

[9] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F Heremann, and C. Kaiser. *Overview of the Chorus Distributed Operating System*, April 1990.

[10] Vincent Frank Russo. *An Object-Oriented Operating System.* PhD thesis, University of Illinois at Urbana-Champaign, 1991.

[11] SPARC International Corporation, P.O. Box 58130, Santa Clara, California. *SPARC Architecture Reference Manual*, 1986.

[12] Bjarne Stroustrup. What is object-oriented programming. In *USENIX '87 C++ Workshop.* USENIX Association, November 1987.

[13] Bjarne Stroustrup. *The C++ Programming Language Second Edition.* Addison-Wesley Publishing Company, Inc., 1991.