The Design of An Operating System for Modern Programming Languages

James Philbin
Research Report YALEU/DCS/RR-997
May 1993

This work is partially supported by Darpa under ONR contract N00014-88-K-0573, NSF DCR-8451415 and NSF CCR-8809919.

The Design of An Operating System for Modern Programming Languages

A Dissertation
Presented to the Faculty of the Graduate School
of
Yale University
in Candidacy for the Degree of
Doctor of Philosophy

by
James Philbin
May, 1993

Abstract

The Design of an Operating System for Modern Programming Languages

by
James F. Philbin
Yale University
1993

This dissertation describes an operating system, called *Sting*, that provides an efficient foundation for the implementation of modern programming languages. Modern languages can be distinguished from traditional ones by their support for concurrency, various synchronization models, anonymous first class procedures, objects, automatic storage management, and topology mapping. Significant efficiencies can be achieved by integrating support for these mechanisms into the operating system.

The fundamental concurrency constructs of *Sting* are virtual machines, virtual processors, and threads. Each is a first class object. *Sting* threads are *extremely* lightweight and the implementation provides significantly more locality of reference than previous systems have. First class virtual processors in combination with threads allow programmers and language designers to build virtual topologies that can be used to enhance the efficiency and portability of parallel algorithms.

Sting separates control mechanisms from policy mechanisms. Policy management is completely customizable. Each virtual machine or virtual processor can have a distinct policy manager, allowing many different scheduling policies to co-exist on the same machine. Several other significant innovations are discussed in the dissertation.

Acknowledgments

I would like to thank the many people who aided me in the completion of this dissertation. Suresh Jagannathan whose many insights and constant encouragement were invaluable. My advisor, Paul Hudak, who excited my initial interest in this work and provided years of guidance, support, and friendship. David Gelernter and Alan Perlis constantly challenged my imagination and expanded the scope of my thinking.

Many of the ideas presented here are the result of my association with people involved in two projects at Yale, the T Project: Norman Adams, Richard Kelsey, David Kranz, and Jonathan Rees and the Tools Project: Nat Mishkin, John Ellis, Bob Nix, and Steve Wood. Interaction with these people provided the most significant and rewarding aspect of my education at Yale. Discussions with Nat Mishkin, Kai Li, and Norman Adams were particularly valuable. Rick Mohr provided me with both stimulating discussion and several of the benchmarks. Henry Cejtin, Monika Rauch and Alvaro Campos have been extremely helpful as early users of *Sting*.

My sincerest thanks to Dennis Philbin, Molly Meyer, and my parents for their support and care over the years. And finally, I wish to thank my wife Tracy to whom this dissertation is dedicated.

This work is partially supported by Darpa under ONR contract N00014-88-K-0573, NSF DCR-8451415 and NSF CCR-8809919.

Contents

Figures

Tables

Chapter 1

Introduction

This thesis describes an operating system, called *Sting*, designed to support the requirements of modern programming languages effectively. *Sting* serves as a coordination substrate that permits the expression of a wide range of concurrency structures within the context of various computation languages. In addition to traditional operating system concerns, *Sting* defines a general-purpose parallel programming model on top of which a broad spectrum of specialized coordination languages, found in modern programming languages, can be efficiently realized.

The design of an operating system should take into account current and probable future trends in computer architecture. One of the most important trends in computer architecture is the growing availability of general-purpose multi-processors. This has led to increased interest in building efficient and expressive software platforms for concurrent programming. Most efforts to incorporate concurrency into modern programming languages involve either the addition of special-purpose primitives (e.g. parallel let operations, futures, events, etc.) or the use of operating system facilities. Both of these approaches have problems.

Special purpose primitives are typically implemented using a dedicated runtime system sensitive to the particular semantics of the primitives. While reasonably efficient, these systems have nonetheless proven difficult to use as a substrate or foundation for a concurrent programming environment. This is because (a) the high-level semantics of the concurrency primitives they support lead to inefficient implementations of other concurrency paradigms; (b) the inaccessibility of the language's runtime structures make it cumbersome for applications to tailor the implementation to their particular requirements; and, (c) the reliance on operating system services for process management and control found in many of these languages incurs high overhead in the pres-

ence of fine-grained, interactive, or realtime concurrency.

There are two general approaches to implementing high-level parallel languages. This thesis will characterize these as the top-down approach (from the language designer's perspective) and the bottom-up approach (from the operating system perspective).

The top-down approach implements a high-level parallel language by building a dedicated (user-level) virtual machine. The machine's main role is as an efficient substrate that implements the specific high-level concurrency primitives found in the coordination sub-language. Given a coordination language L supporting concurrency primitive P, L's virtual machine (L_{vm}) handles all implementation issues related to P; this often requires that the machine manage process scheduling, storage management, synchronization, etc. Because L_{vm} is tailored only towards efficient implementation of P, however, it is often unsuitable for implementing significantly different concurrency primitives. Thus, to build a dialect of L with concurrency primitive P' usually requires either building a new virtual machine or expressing the semantics of P' using P. Both approaches have their obvious drawbacks: the first is costly to implement given the complexity of implementing a new virtual machine; the second is inefficient given the high-level semantics of P and L_{vm}'s restricted functionality.

Rather than building a dedicated virtual machine for implementing concurrency, the bottom-up approach to language implementation uses existing operating system services [Bla90] [TRG+87]. Process creation and scheduling are implemented by creating either heavy- or lightweight OS-managed threads of control; synchronization is handled using low-level OS-managed structures. These implementations are generally more portable and extensible than systems built around a dedicated runtime system, but they necessarily sacrifice efficiency [ABLL91] since every kernel call requires a context switch between the application and the operating system. Moreover, generic OS facilities perform little or no optimization at either compile time or runtime since they are usually insensitive to the semantics of the concurrency operators of interest.

The top-down approach is generally motivated by the goal of efficiency. It relies on the

specialized virtual machine (runtime system) because its abstractions can be implemented more efficiently there than the operating system. The bottom-up approach is motivated by the twin goals of ease of implementation and portability. It trades these for efficiency.

Sting combines the best of both approaches, providing an efficient substrate which is portable and allows easy language implementation, while at the same time introducing new optimizations and new functionality. In contrast to other parallel languages, the basic concurrency objects in *Sting* (threads and virtual processors) are streamlined data structures with no complex synchronization or value transmission requirements. Unlike parallel languages that rely on operating system services for managing concurrency, *Sting* implements all concurrency management in terms of user space objects and procedures, permitting users to optimize the runtime behaviors of their applications without requiring knowledge of underlying kernel services.

1.1 Important Features of Modern Languages

The growing interest in parallel computing has led to the creation of a number of parallel programming languages that define explicit high-level program and data structures for expressing concurrency. These languages typically support (with varying degrees of efficiency) concurrency structures that realize dynamic lightweight process creation [Hal85] [Hor89], high-level synchronization primitives [Rep91] [Sar90], distributed data structures [Car89a], and speculative concurrency [Cla86] [Osb90]. In effect, these high-level parallel languages consist of two sub-languages -- a coordination language responsible for managing and synchronizing the activities of a collection of processes, and a computation language responsible for manipulating data objects local to a given process [CG90].

Modern programming languages can be broadly divided into four classes:

Expression oriented - Common Lisp, Scheme, Multi-Lisp

Object oriented -Actors, SmallTalk, C++, T, Dylan, Self.

Functional - ML, Haskell, Miranda, ID.

Logic programming - Prolog, Concurrent Prolog, Janus.

While *Sting* also supports more traditional programming languages such as Fortran, Cobol, C, Pascal, and Ada, it is expressly designed to efficiently support the operating system requirements of modern programing languages. These form a superset of the requirements of traditional programming languages.

There are several important features that distinguish modern languages from traditional ones. The features most relevant to operating system design are as follows:

Concurrency - At least some of the languages in each of the four classes described above provide a notion of concurrency. These include Ada tasks, Multi-Lisp futures, Linda's tuple spaces, and Concurrent Prolog's terms. Two developments in the recent past, computer networks and parallel computers,[1] have made concurrency one of the fundamental issues in programming language design. Both of these developments have allowed several processors to join simultaneously in the same computation. Providing expressive and efficient mechanisms for coordinating these concurrent computations will continue to be one of the primary goals of programming language designers.

Multiple Synchronization Models - Parallel or asynchronous programming languages use many synchronization protocols. A modern operating environment should as far as possible provide the primitives to support these various protocols.

Anonymous First Class Procedures - Many modern languages support anonymous procedures. These procedures are first class in the sense that they can be passed as arguments to and returned from other procedures. They can also be stored in data structures like any other value in the language. First

1. In fact networks can be regarded as parallel computers.

class procedures provide many novel methods for communicating and coordinating concurrent computations.

Objects - The concept of object found in object oriented languages allows the programmer to build rich hierarchies of customizable types. These can lead to clearer, more portable, and more readily customizable operating system interfaces.

Evaluation Order - Modern languages have different evaluation orders. Some languages such as Haskell and Miranda use lazy evaluation, where the value of an expression is not computed until it is needed. Other modern languages such Multi-Lisp and Actors allow eager evaluation, where the value of an expression is computed simultaneously with the value of other expressions. Some languages, such as Parlog [CG85], use a variant of this called speculative evaluation, that begins evaluating several expressions simultaneously. The first expression to complete evaluation terminates the evaluation of the other expressions. Finally, many modern languages use traditional sequential evaluation.

Automatic Storage Management - Most modern languages support some form of automatic storage management, because automatic storage management allows more expressive programs, while at the same time reducing the programs complexity. Dynamic allocation and reclamation of storage leads to programs with fewer errors and more locality of reference compared to those using explicit storage management.

Topology Mapping - While not yet supported in many programming languages, the ability to control the mapping of processes to processors so as to reduce the communication overhead of a program will become more important as the size of multi-processor computer systems continues to grow and the topologies become more complex.

Exceptions - Many modern languages also support exception handling and the ability to dynamically unwind the side effects of a computation when an exceptional situation occurs. Exceptions are particularly useful for error handling. However, the semantics of exceptions vary widely from language to language. It is important that exceptions which occur during operating system calls can be delivered to a language while honoring the semantics of that language's exception mechanism.

Until now most operating systems have not taken advantage of these developments, and no operating systems has taken advantage of all of them. The principle reason for this is that operating system design and language design have been regarded as independent disciplines. Furthermore, operating systems have been typically designed to support only the requirements of traditional programming languages.

Sting supports these various requirements efficiently. It does so in a new architectural framework that is more general and more efficient than others currently available. It also offers the programmer an increased level of expressiveness and control, and an extraordinary level of customizability. Even though *Sting* is designed to support modern programming languages, it can support traditional programming languages just as efficiently.

1.2 Design Goals and Philosophy

Sting defines a small set of *simple, general, orthogonal,* and *powerful* primitives that can be combined to express a broad range of concurrency paradigms efficiently. These primitives are both semantically simple and combine well. In addition to this general philosophy, there were several important goals that guided the design. These are outlined below:

Generality - The basic concurrency management objects in *Sting* are extremely lightweight threads of control and virtual processors (VPs). Both are first class. Unlike high-level concurrency structures, *Sting* is unencum-

bered by complex synchronization or communication protocols. *Sting*'s threads and VPs are efficient primitives that can be used to express more abstract concurrency paradigms. Virtual processors serve as an abstraction of a physical computing device, and can be used to build abstract machine topologies that support the requirements of the application being implemented. VPs can also be tailored to implement specialized process migration and scheduling protocols. Threads and VPs can be manipulated in the same way as any other value in the language. First-class threads and VPs permit users to experiment with a variety of different execution regimes - from *fully delayed* to *completely eager* evaluation.

Efficiency - Because threads are fully integrated into the base language, and not provided as part of a library package [Coo88], it is straightforward to optimize their implementation and use.

Thread operations are performed by a thread manager. The semantics of threads and the design of the thread manager minimize the cost of thread creation, and put a premium on storage locality. Thus, the execution context for a newly terminated thread (e.g. its stack and heaps) can be recycled and used immediately by other newly-created threads, storage allocation is delayed until the thread can actually execute, and different threads can share the same stack and heap if data dependencies warrant. This last optimization, which we refer to as thread absorption, is a novel design feature. The storage management decisions in *Sting* lead to better cache and page utilization.

Programmability - Beyond simply providing mechanisms for managing concurrency, *Sting* also handles inter-process exceptions, process migration, preemption, non-blocking I/O, per-thread asynchronous storage management, and maintains extensive thread genealogy information. In addition, it provides an infra-structure for implementing multiple address spaces, and long-lived persistent objects. As a result, *Sting* is an expressive operating

system substrate that can be used as a platform upon which an advanced programming environment for parallel computing can be built.

Customizability - *Sting* separates thread policy decisions from thread implementation ones. Although all threads conform to the same basic structure, implementations of different parallel languages built on top of *Sting* can define their own scheduling, migration, and load-balancing policies without requiring modification to the thread manager or to the provided interface. Process migration and scheduling concerns typically handled internally by a runtime library or an operating system in other high-level parallel languages can be customized by applications on a per-VP basis.

Unlike other systems that implement application-dependent scheduling policies, (e.g. Hydra [Wul81]), *Sting* does not incur a performance penalty for this flexibility; scheduling policy decisions are implemented entirely in user space and thus do not require a trap into a low-level system kernel.

Sting was conceived and designed not only as an efficient operating system for MIMD parallel computers, but also as a platform for exploring and comparing different models of parallel programming. Toward this end the various components of the systems were designed to record as much information as possible about the behavior of programs. We believe that this goal of analizability has often been sacrificed for the sake of efficiency, a design choice that is both unfortunate and unnecessary. *Sting* is designed to maintain as much information as the user desires without sacrificing efficiency.

1.3 Problems with Current Operating Systems

Current operating systems present several problems when used as a foundation for a parallel language. These problems relate to the implementation and functionality of threads, the customizability of operating systems policies, the mapping between threads and processors, the interaction between user space and kernel space, and the

memory model supported by the operating system.

The sections that follow discuss the specifics of these problems and give an overview of *Sting*'s solution to them.

1.3.1 Thread Implementation

Current operating systems can be divided into two groups: those that provide heavyweight processes and those that provide lightweight threads. A heavyweight process combines one locus of control with a single address space. Traditional operating systems such as Unix and VMS have used this model of process. These processes are heavyweight because creating one involves creating a new address space map, even if more than one of these processes is sharing all or part of the same virtual address space. The cost of creating a new address space map for each thread is expensive in both time and space, thus the term *heavyweight.*

In the last ten years, two less expensive types of processes have evolved. Both are identified by the term *lightweight* thread. Modern operating systems such as TOPAZ [Bir89], Mach [BGJ+92], and Chorus [RAA+92] have allowed more than one process (usually called a thread) to share the same address space map, thus significantly reducing the cost, in both time and space, of process (thread) creation. For clarity, these processes are referred as *(lightweight) kernel threads.* Kernel threads, while less expensive than heavyweight processes, are still at least an order of magnitude more expensive than a procedure call.

Languages like ML and Modula2+, as well as runtime libraries such as C Threads and Posix threads [Soc90], have introduced a second type of lightweight thread, called *a user space thread,* or simple *user thread.* User threads are less expensive to create and destroy then kernel threads because they have no address space map associated with them and the creation/destruction of a thread does not require crossing the protection boundary into the kernel, an expensive operation. Unlike kernel threads, user threads do not require kernel data structures (e.g. control blocks and stacks).

User threads, however, have several problems. They are implemented on top of kernel threads, but the kernel of the operating system has no knowledge of them. So, for example, when a user space thread blocks in the kernel (e.g. while making a system call or page fault) all the user threads which share the kernel thread also block, even though many of them might be able to do useful work. Another problem is that when no user threads are available to run in some address space, there is generally no way to communicate this information to the kernel, even though some other address space might have threads which are ready to run.

The cost of thread creation and destruction is important to both language designers and application programmers because it determines the granularity of parallelism available to a program. The cheaper the cost of thread creation, the finer the granularity of parallelism that the language or system can support. *Sting* improves on the thread designs mentioned above by providing *extremely lightweight threads* that are known to the kernel.

Sting threads are created in user space, but they are lighter weight than those of other thread systems, because the creation of the thread is decoupled from the creation of the execution context of the thread. This decoupling has several beneficial effects: First, it reduces the cost of thread creation (see Section 2.7 on page 33). Second, it reduces the storage requirements of the system while at the same time increasing locality of reference (see Section 3.3.2 on page 57). Finally, it allows *Sting* to perform a new dynamic optimization called *thread absorption,* that has the dual benefits of reducing the cost of thread evaluation while both reducing storage requirements and increasing locality even more (see Section 3.3.3 on page 59).

Another problem with existing thread systems is that they usually provide a limited number of synchronization constructs. These generally include mutexes and condition variables, and sometimes monitors [Hoa74]. However, the synchronization model is usually built into the thread system and thus it is often difficult, and almost always inefficient, to implement synchronization models not supported by the thread system. *Sting* solves this problem by making synchronization completely orthogonal to thread

creation and scheduling. Thus, *Sting*'s threads can be coordinated using any synchronization model the language implementer (or user) desires. This coordination can be achieved efficiently since there is no extra overhead incurred by implementing the synchronization model in terms of the model supported by the thread system. Details of thread synchronization are discussed in Section 3.4 on page 68.

The combination of inexpensive *Sting* threads and the fact that various synchronization models can be built efficiently using them, provides the language designer or the programmer with enormous flexibility in designing and implementing various models of parallel computation. Furthermore, because *Sting*'s threads are so lightweight, the user or language designer can easily encapsulate them in other objects to implement additional functionality.

Existing operating systems do not provide adequate facilities for debugging and performance evaluation. In contrast, *Sting* provides extensive facilities for both. These are described in Section 2.11 on page 41. These facilities make *Sting* an excellent vehicle for testing new ideas in either language or system design.

1.3.2 Operating System Customization

The various thread systems mentioned above do not allow the customization of scheduling or load balancing policies. Unless scheduling policies can be customized, an operating system must be targeted to one kind of operating environment, e.g. real time, interactive, or batch. An operating system which can be customized for various environments offers many advantages. The two most important are: (1) the programmer doesn't need to learn another operating system interface every time he builds programs for a different operating environment, and (2) programs become portable across different types of operating environments. For example, a program designed for a real time environment could be implemented and debugged in an interactive environment and then painlessly moved to the real time system.

The ability to customize thread migration and load balancing strategy, on an application by application basis, allows the implementer to use a strategy which is appropriate

for both the application being developed and the architecture on which the application is running. Unlike other systems, *Sting* allows the complete customization of scheduling, load balancing, and thread migration decisions. Customization is explained in detail in Section 4.7 on page 97 and Section 5.8 on page 131.

Current thread systems do not typically allow the user to run a thread on a particular processor. This functionality is particularly important when communication costs inside a parallel machine are non-uniform. This is the case in systems with topologies such as hypercubes or meshes, as well as systems like the BBN Butterfly. *Sting* introduces the novel concept of a first class virtual processor and allows a thread to be mapped onto any virtual processor. As a result, the programmer can map threads onto processors to optimize inter-thread communication costs.

First class processors provide the *Sting* programmer with another important feature not found in other operating systems. They allow users to build virtual topologies and map them onto physical topologies. This gives programmers two advantages. (1) It allows them to express an algorithm in terms of the topology of the problem domain and yet map it automatically and efficiently onto a particular physical topology. (2) It also allows the user to port an application built on one physical topology onto another physical topology by simply implementing a new virtual topology, without changing any other part of the program. Section 4.8 on page 109 describes virtual topologies.

1.3.3 Interaction between User Space and Kernel Space

All operating systems implement some sort of kernel threads or processes. These kernel threads require large data structures in the kernel, principally a control block and stack for each thread. Whenever a kernel call is made explicitly, because of a system call, or implicitly because of an exception such as a page fault several expensive operations must occur. The protection mode on the processor must be changed, which might involve switching page tables. The thread switches to a kernel stack and begins running on that stack. This entails a loss of locality because the kernel stack is unlikely to be loaded into the caches.

Sting avoids these problems and their associated costs. *Sting* has no kernel threads and thus, non of the kernel data structures associated with them. *Sting* handles all system calls and exceptions by using the user space execution context (stack, etc.) of the thread that makes the system call or generates the exception. This saves storage and increases locality.

Sting maps the entire kernel into each virtual address space at the same location. Part of the kernel is mapped with read only access and part is mapped with no access in user mode. This means that many traditional system calls can now be made by simple procedure calls or, with compiler integration, with as little as a single instruction overhead. *Sting* also reduces the cost of system calls and exceptions because the protection boundary can be crossed with only one instruction to change the protection on the kernel pages with no cache flush being necessary. Further, since the *Sting* compiler saves only live registers before making a kernel call, it is unnecessary to save a thread's context on entering the kernel. Finally, since the kernel runs using the user space stack of the *Sting* thread no switch to a kernel stack is necessary. As a result, crossing the kernel protection boundary is no more expensive than making a procedure call. Details of this functionality can be found in Section 5.7.1 on page 130.

In most operating systems, when a thread blocks in the kernel no other threads in that address space can run because the kernel is not aware of the user space thread and the kernel cannot communicate this information to the user space scheduler. The reverse problem also arises when the user space scheduler has no work to do, i.e. no threads are ready to run, the scheduler cannot inform the kernel. This means that the kernel thread spins until its quantum expires.

Because *Sting*'s threads are known to the kernel, the kernel can easily communicate with the address space and vice versa. Thus, when a thread blocks in the kernel, the kernel informs the user space scheduler of this event and another thread can be run. Likewise, when the address space has no thread available to run, it can communicate that fact to the kernel. User space/kernel space communication is described in Section 5.7.1 on page 130.

1.3.4 Memory Management

Multiple instruction multiple data (MIMD) parallel machines and operating systems fall into two broad classes: those which support shared memory and those which support message passing. The shared memory model has significant benefits for the programmer. The principal benefit is that shared memory is a simpler cognitive model. Shared memory systems are also easier to debug. *Sting* is based on a memory model called *Distributed Shared Virtual Memory*. This allows *Sting* to run on MIMD machines with disjoint memories. *Sting* threads are designed to run efficiently on distributed shared memory. Section 5.5.1 on page 124 discusses distributed shared virtual memory.

Another important trend in computer architecture, especially in micro-processor design, is the reliance on locality of reference to improve performance. This reliance is apparent in two trends in the design of the memory hierarchy. First, the number of levels in the hierarchy have increased. Microprocessors routinely have on chip primary instruction and data caches with provisions for off chip secondary caches. On some machines main memory is divided into local and global memory. This division is likely to be standard on future machines, with a further subdivision into a hierarchy where some of the memory is completely local to a particular processor, some of the memory is only accessible to a locally close group of processors, and some memory fully global. Finally, the backing store stage of the memory hierarchy which previously included only disks and tapes, now includes disk caches and even more complex RAID[1] subsystem. Each of these stages exploits the data locality that exists in almost all programs.

The second trend in memory design is the increase in size of every level of the memory hierarchy, from the number of registers in the processor, to the size of the caches, main memory, and disk subsystems. This when combined with the growing complexity of memory hierarchies, magnifies the importance of locality of reference for achieving high performance even further.

1. Redundant Array of Inexpensive Disks.

As locality of reference becomes increasingly important, it is crucial that operating systems be designed not only to exploit, but also to increase the locality of reference in programs. This observation is true for single processor systems, and has even greater significance for multi-processor systems.

The *Sting* design increases locality in several important ways:

- delaying thread execution context allocation (see Section 3.3.2 on page 57),
- providing each thread with it's own local stack and heaps (see Section 3.5.3 on page 75 and Section 3.5.4 on page 77),
- allocating not only frames on the stack, but any objects whose lifetimes do not exceed the dynamic extent of a procedure call (see Section 3.5.3 on page 75),
- allocating shared heaps on the basis of thread groups and environments (see Section 3.5.5 on page 78).

Unlike other operating systems, *Sting* is designed to support automatic storage allocation and reclamation (garbage collection). Most garbage collection algorithms can benefit from efficient access to page tabes. *Sting*'s page tables are mapped into user space and can therefore be accessed extremely quickly.

Each evaluating thread has an execution context associated with it. The execution context is composed of a thread control block, a stack, a private heap, and a shared heap. *Sting* allocates all data that is private to the thread, i.e. not shared with any other thread, in either the stack or the private heap. This has several advantages. Stacks and private heaps can be located in local non-coherent memory, thus reducing the memory contention bottleneck associated with shared memory machines (see Section 3.5.3 on page 75 and Section 3.5.4 on page 77). In addition, the thread's private heap can be garbage collected independently of any other thread. This significant innovation is described in Section 3.5.4 on page 77.

1.3.5 Compiler Integration

In general, conventional operating systems do not provide integration between the compiler and the operating system. As a result, many significant compile time optimizations are lost. In contrast *Sting* is designed to take advantage of compiler integration. At present, several of these have been implemented, including minimal register saves on synchronous context switches, and minimal cost saving of the current continuation during context switch (see Section 4.6 on page 91).

Sting's stack are designed so that a compiler can allocate objects other then activation frames in them. This has the significant advantage of improving locality and decreasing the cost of garbage collection. Section 3.5.3 on page 75 discusses these and other optimizations that improve locality of reference.

1.4 Overview of Sting

The *Sting* operating system architecture is composed of several layers of abstraction, see Figure 1-a on page 17. The lowest layer is the *abstract physical machine* (Section 5.4 on page 122), which is composed of *abstract physical processors* (Section 5.7 on page 129). It corresponds to the micro-kernel in newer operating systems. Each abstract physical processor is composed of a *virtual processor controller* (Section 5.8.1 on page 131) and a *virtual processor policy manager* (Section 5.8.2 on page 132). The virtual processor controller handles all interactions with the virtual machine, while the virtual processor policy manager makes policy decisions for the virtual processor controller. The virtual processor policy manager is completely customizable.

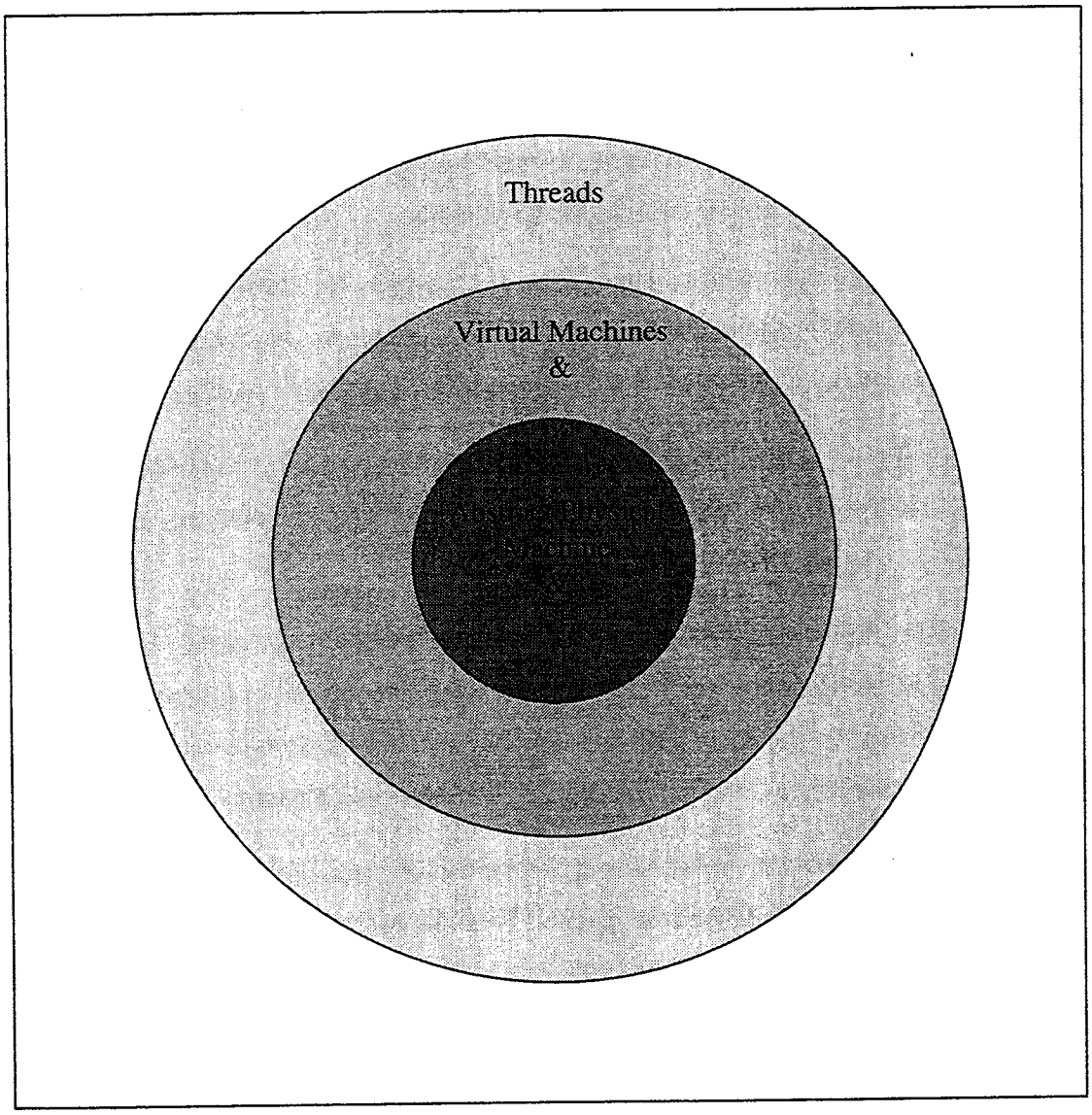

Figure 1-a : The Sting Architecture

The second layer consists of *virtual machines* and *virtual processors* (VPs). A virtual machine is composed of virtual processors and a virtual address space. Virtual machines are mapped onto physical machines, and each virtual processor is mapped onto a physical processor. Virtual machines and virtual processors are first class objects in *Sting* and they provide the user with an unusual level of expressivity. Each virtual processor is composed of a *thread controller* (Section 4.5 on page 91) and a

thread policy manager (Section 4.7 on page 97). These are analogous to the virtual processor controller and policy manager. The thread controller handles all interactions with the virtual processor, including thread state transitions and exceptions. The thread policy manager makes all policy decisions for the thread controller. It is completely user customizable, allowing applications to make scheduling and load balancing decisions which are appropriate for a particular application.

The highest layer of abstraction defines *threads* (Chapter 2). Threads are extremely lightweight processes which are run on virtual processors. As mentioned above threads are first class, allowing them to express almost any kind of parallel construct.

For clarity, it should be mentioned that each of the above layers is a software system. Thus, when we speak of a "abstract physical machine," we are referring to the software layer that controls the actual physical hardware.

The rest of this thesis will describe these layers of abstraction in detail; however, it does not discuss the details of the file system, network interaction, or other I/O devices. Instead, it focuses on the most innovative aspects of the *Sting* architecture: control flow, storage management, and policy making in the operating system.

The various components of *Sting* have significant interaction. Each component relies on the functionality of the other components. This makes it somewhat difficult to discuss one aspect of the system without referring to other aspects not yet explained. To minimize this problem, this thesis introduces the most familiar concepts (threads) first, and then proceeds to less familiar ones. The rest of the thesis is structured as follows: Chapter 2 discusses threads and their implementation. Chapter 3 describes execution contexts and their relation to threads. Chapter 4 explains virtual machines and virtual processors and in Chapter 5 abstract physical machines and abstract physical processors are discussed. Chapter 6 gives performance details for the various benchmarks we have run. Finally, our conclusions and future research are described in Chapter 7.

Chapter 2

Threads

Threads are the locus of control in *Sting* and all code is executed in the context of some thread. *Sting*'s threads are *extremely lightweight*. They can be created with very little overhead, and more than one thread can be created in the same virtual address space. *Sting*'s threads are first class, i.e. they can be passed to and returned from procedures and stored in data structures. *Sting*'s threads also have values. The value of a thread is the value of the expression it evaluates. Because *Sting*'s threads are first class and have values they can be used in ways that threads in other thread systems, such as Mesa or C Threads, cannot. In particular, they can be used to build dynamic data structures. This functionality is important for languages such as Linda and Multi-Lisp. Furthermore, since *Sting*'s threads evaluate lazily, normally, or eagerly, these dynamic data structures can be used for both speculative and infinite computation, a requirement for both functional and logic languages. Decisions about the evaluation strategy are made by the programmer using *Sting*.[1]

Sting threads are *extremely* lightweight not only because they reside in user space, but also because they are much lighter in weight than those in most other lightweight thread systems. Because the thread itself is a very small data structure, on the order of ten words of memory, which contains among other things:

1. We will use the term programmer throughout this thesis to denote either someone building an application directly on top of *Sting* or a language designer using *Sting* as the substrate for some parallel language.

- the closure of the expression which the thread is to evaluate,
- the dynamic state in which the thread was created, i.e. the dynamic bindings, exception bindings, and dynamic windings which are extant when the thread is created. These are discussed in detail below.
- the priority and quantum of the thread, and
- the genealogy of the thread.

The thread data structure can be small because the execution context[1] of the thread is not allocated until the thread begins evaluating. Threads can be preemptible or non-preemptible as the programmer desires. They can also be used as co-routines.

2.1 Comparison with Other Thread Systems

System designers have designed and implemented many different lightweight thread systems over the years. One of the earliest thread systems was incorporated into the language Mesa [LR80], developed at Xerox PARC. Mesa incorporated multiple threads of execution, called processes, in the same address space using a *fork/join* style of parallelism. The Mesa design was innovative and its ideas have influenced the design of most contemporary thread systems.

Mesa's processes were lightweight. Creating and destroying threads was inexpensive compared with other multiprocessing systems of the time. The low cost of process management resulted from the fact that all the processes reside in the same address space with no protection mechanism, and from the low storage overhead associated with processes. The implementation used cactus stacks [BW73] [Ste77]. Unfortunately this memory model suffers from poor locality of reference and is not suitable for today's cached memory architectures.

Mesa threads are first class values in the language, meaning that they can be passed to or returned from procedures and stored in data structures. The value computed by a

1. See Chapter 3.

Mesa thread can be accessed using the **JOIN** statement, but the thread can only be joined once. With the exception of the **JOIN** statement, Mesa separates parallelism from synchronization.

While the Mesa thread system was well integrated into the language, the Mesa memory model did not support threads as well as it might have. For example, the lack of garbage collection engendered the need for a **DETACH** statement, which informed the system that the value of the thread would never be used, and that the resources devoted to the thread could be recycled as soon as the thread finished execution. Lack of garbage collection also raised the possibility of dangling references. Since threads are recycled, a reference to a thread which has terminated could cause the entire system to crash.

Despite these shortcomings, Mesa made several important contributions to the design of thread systems. These include the integration of both compiler support and exception handling into the thread system. Finally, Mesa was designed as a single user, single address space system, thus there is no protection mechanism associated with processes. This was a significant departure from the "heavyweight" kernel processes of other operating systems. Mesa also obscured the difference between the operating system and the programming language, thus allowing the Mesa programmer to rely on the semantics of the language across different operating environments.

Mesa was followed by systems such as Mach [TRG+87]], TOPAZ [BGHL87], and Synthesis [MP89] which implemented lightweight threads in the kernel of the operating system. The threads of these systems are "lightweight" in so far as they shared the same (user) address space, but the thread handling mechanisms are located in the kernel in order to provide protection from stray writes to memory. These kernel threads are more expensive than Mesa threads. Their thread data structure storage requirements are much higher, partly because they have both user space and kernel space data structures associated with them, but also because neither Mach or Topaz devote sufficient attention to implementing structures such as the control stack inexpensively. In addition, in these and other kernel thread systems, most thread operations cross the

protection boundary between user and kernel space. One final weakness in kernel thread systems is that threads are not first class values. Instead, they are represented by *non-unique* identifiers such as an integer.

Modula-2+ threads [RLW85], C Threads [CD88]], and POSIX threads [Soc90], are other important thread systems. Since they are similar we will discuss only C Threads here. C Threads has three different implementations: one implementation uses co-routines. One uses heavyweight Unix kernel processes that communicate through overlapping address spaces. And finally, one implementation relies on Mach threads. Here we will focus on the implementation that uses Mach threads since it is the most efficient of the three.

C Threads relies on a conception of threads that is very similar to that of Mesa. The model supported is fork/join. Both systems share similar problems, i.e. threads can't be joined multiple times and thus can not be used to denote values in data structures which might be accessed more than once, and threads which are never joined must be detached. Furthermore, C Threads recycle Mach kernel threads, because of the expense of creating kernel threads. The use of kernel threads makes thread creation and destruction extremely costly.

Despite these shortcomings, C Threads did introduce the ability to associate data with a thread and to access that data in a thread relative manner. Although significant, this functionality was not integral to the design or function of the system. Instead, it appears that Cooper and Draves added this feature to overcome the problem of non-unique thread identifiers. *Sting*'s first class threads allow the user to form any desired associations between threads and other data using the normal facilities of the language.

Psyche [MSLM91] is the parallel operating system most closely related to *Sting*. Designed and implemented at approximately the same time, Psyche and *Sting*, share a similar goal, namely to support multi-model parallel programming efficiently. Psyche differs from our work in that it builds a kernel interface designed to support many different thread models, but does not define a fundamental thread model of its own, rather

it specifies a mechanism (software interrupts) for communicating with the kernel. The Psyche research is more closely related to our work on virtual machines and physical machines which is discussed in subsequent chapters.

In contrast to Psyche, our approach has been to define the fundamental constructs needed to build the various models of parallelism, to implement them efficiently, and then to allow the language designer to build different models out of these constructs. *Sting*'s threads are one of these fundamental constructs. We believe that this is a more efficient method of supporting multiple models of parallelism. However, *Sting* does not prohibit a language designer from implementing any desired thread abstraction using virtual machines.

2.2 Sting Innovations

Sting is a language independent substrate for implementing parallel systems and languages. *Sting*'s threads are innovative in several ways. Perhaps the most important innovation is that the thread data structure is disjoint from it's execution context. This allows threads to be extremely small and inexpensive to create. The separation of the thread data structure from the execution context data structure facilitates a new type of optimization on threads called *Thread Absorption.* This separation also allows *Sting* to support both lazy and eager evaluation strategies efficiently, as explained below. Another innovation is that threads have values and those values can be used multiple times, thus allowing threads to be used as distributed data structures.

Unlike any of the above mentioned operating systems, threads are garbage collected and unlike any of the various parallel languages, discussed above, each thread can be garbage collected independently.

The *Sting* thread system supports dynamic winding, dynamic binding, and exception bindings. The primitives for dynamic winding and exception binding can be used to build various models of exception handling, but *Sting* also provides a default exception handling model for programmers.

Another innovative aspect of *Sting* is its support for thread groups which provide the user with a new method of organizing parallel computations. Thread groups also support the gathering of information about a thread's genealogy, its performance, and other information used for debugging. These various features will be discussed in more detail throughout the rest of this chapter and the next.

2.3 Thread Operations

Figure 2-a on page 25 lists the various operations that can be performed on threads. Designed as building blocks with very simple semantics, these operations allow language designers to design thread systems with more complicated semantics, although *Sting* threads have proven quite useful in their own right.

Figure 2-a separates the *Sting* operations according to functionality. The semantics of these primitives are reasonably obvious. Their functionality will be defined briefly below, but not in detail. The interested reader should see [Phi93] for details.

Thread operations include those that create threads, execute them, and determine their value(s). Other operations block, suspend, terminate, or synchronize threads. Finally, there are operations for referencing the current thread and the current virtual processor.

Significantly, *Sting* is designed so that control flow is completely orthogonal to synchronization.[1] This is important for two reasons: first it allows threads to be used with many different synchronization constructs. Second, it affords designers broad latitude in choosing synchronization constraints on threads that they deem appropriate.

1. Figure 2-a includes, under the heading of thread synchronization, several procedures provided by *Sting* that support different models of synchronization. These procedures are not integral to the design but are provided for programmer convenience.

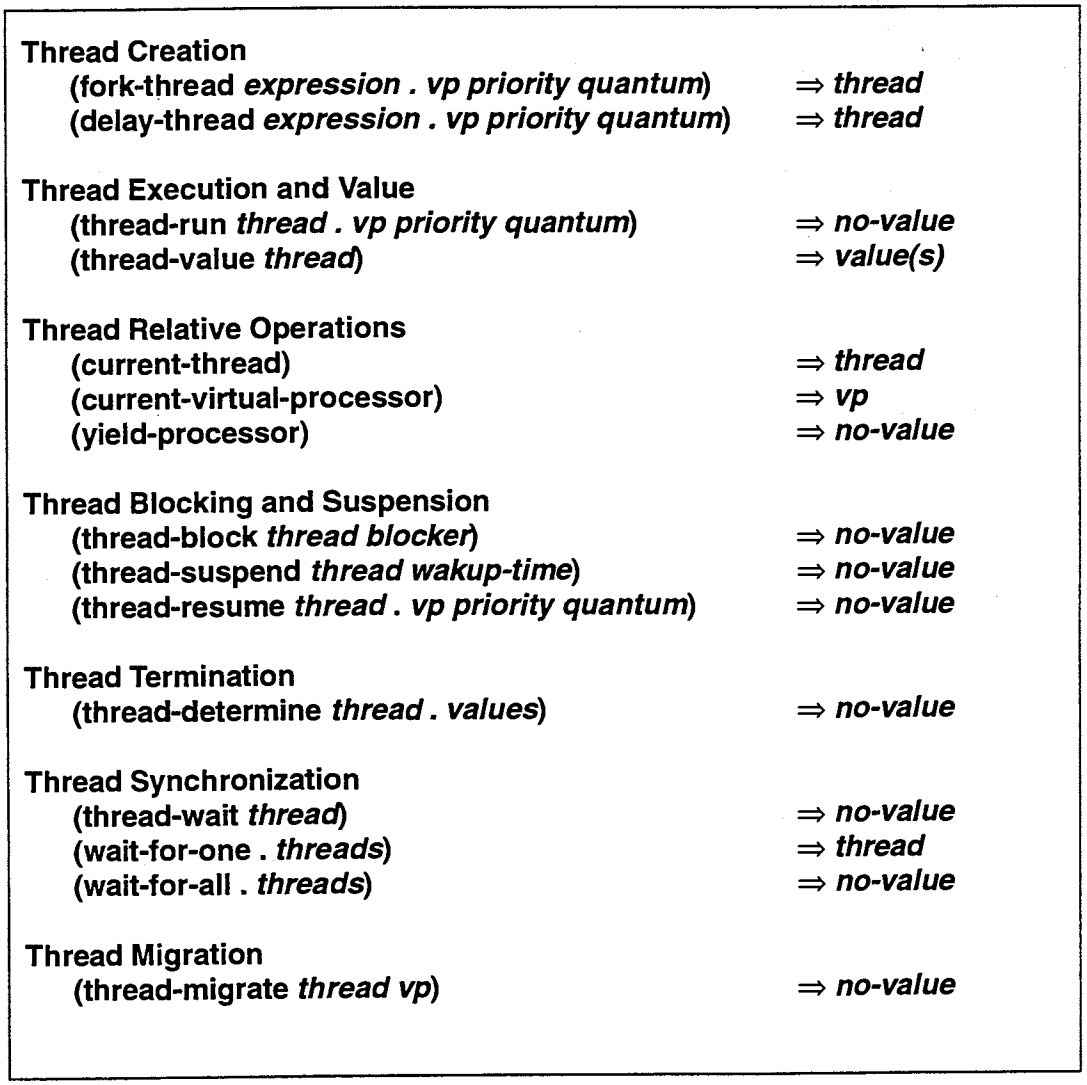

Figure 2-a : Thread Operations

Thread Creation - The thread creation special forms create a thread to evaluate *expression*. *vp* designates the virtual processor on which to evaluate the thread; *priority* specifies the priority assigned to the thread; and *quantum* specifies the time quantum allotted to the thread. However, the thread policy manager may regard the *vp* argument as a hint. It is free to schedule the thread on some other virtual processor. fork-thread creates a thread and immediately schedules it to run on some virtual processor. delay-thread

creates a delayed thread, which will not be evaluated until either its value (thread-value) or effect (thread-wait) is demanded.

Thread Execution and Value - thread-run is a procedure which schedules a non-running (delayed, blocked, or suspended) thread on some processor. *vp, priority,* and *quantum* are hints for the thread policy manager and are handled in a manner similar to thread creation.

thread-value returns the value(s) of the expression evaluated by *thread.* If the thread had not finished evaluating, i.e. is undetermined, then the caller is blocked until the thread completes and its value is available.

Thread Relative Operations - There are three thread relative operations. **current-thread** returns the current thread, i.e. the thread that is its caller. **current-virtual-processor** returns the virtual processor on which the calling thread is running. **current-virtual-processor** can be used for virtual processor relative addressing in virtual topologies (see Chapter 4). **yield-processor** causes the current thread to relinquish the virtual processor to another thread that is ready to run. The current thread goes to the ready state and will be rerun when the thread policy manager decides that it is the thread with the highest priority.

Thread Blocking and Suspension - **thread-block** and **thread-suspend** move *thread* from the running or ready state to the blocked or suspended state respectively. **thread-resume** schedules a blocked or suspended thread to run on some virtual processor. As with fork-thread vp, priority, and quantum are hints to the thread policy manager.

Thread Termination - **thread-determine** causes *thread* to become determined with *values* as its value. If *thread* is in the delayed or scheduled state then its value is set to *values* and its state is set to determined without ever evaluating the thunk associated with the thread. If *thread* has begun evaluating then it is signalled to call its exit handler with *values* as its arguments. The

exit handler unwinds *thread* (see Section 2.9.3) and then sets it value to *values* and it state to determined.

Thread Synchronization - There are three thread synchronization procedures. **thread-wait** causes its caller to block until its argument, *thread*, is determined. **wait-for-one** and **wait-for-all** cause the caller to block until one or all, respectively, of *threads* become determined.

Thread Migration - Normally, the thread policy manager decides when to migrate a thread from one virtual processor, based on load balancing or other criteria. **thread-migrate** allows the explicit migration of *thread* from its current virtual processor to another virtual processor.

2.4 A Simple Example

To illustrate how a user might program with threads, consider the program shown in Figure 2-b that defines a Sieve of Eratosthenes prime finder implementation.

```
(define (filter op n input)
 (let loop ((x     (hd input))
            (output (make-stream))
            (last?  true))
  (cond ((zero? (mod x n))
         (loop (rest input) output last?))          (last?
         (op (lambda ()
              (filter op x output)))
         (loop (rest input) (attach x output) false))
        (else
         (loop (rest input) (attach x output) last?)))))


(define (sieve op n)
 (let ((input (make-integer-stream n)))
  (op (lambda ()
       (filter op 2 input)))))
```

Figure 2-b : Concurrent Sieve of Eratosthenes

This implementation relies on a user-defined synchronizing stream abstraction that provides a blocking operation on stream access (**hd**) and an atomic operation for appending to the end of a stream (**attach**). Note that the definition makes no reference to any particular concurrency paradigm; such issues are abstracted by its **op** argument.

We can define various implementations of a prime number finder that exhibit different degrees of asynchronous behavior. For example,

```
(let ((filter-list (list)))
```

```
(sieve (lambda (thunk)
         (set filter-list
           (cons (fork-thread (thunk))
                 filter-list)))
         n))
```

defines an implementation in which filters are generated lazily; once demanded, a filter repeatedly removes elements off its input stream, and generates potential primes onto its output stream. To initiate a new filter scheduled on a virtual processor (VP) using a round-robin thread placement discipline, we might write:

```
(thread-run (car filter-list) (next-vp))
```

(next-vp) returns the next VP, in the round robin order, on which the expression is evaluated. **next-vp** is a virtual topology expression (see Chapter 4). A virtual machine's public state includes a vector containing its virtual processors. It is easy to build virtual topology expressions using this vector.

By slightly rewriting the above call to sieve, we can express a more lazy implementation:

```
(let ((filter-list (list)))
  (sieve (lambda (thunk)
           (let ((thread (fork-thread
                          (begin (map thread-run filter-list)
                                 (thunk)))))
             (set filter-list (cons filter-list thread)))
           (map thread-block filter-list))
           n))
```

In this definition, a filter that encounters a potential prime p, creates a lazy thread L and requests all other filters in the chain to block. When L's value is demanded, it unblocks all the elements in the chain, and proceeds to filter all multiples of p on its input stream. This implementation throttles the extension of the sieve and the consumption of input based on demand.

We can also define an eager version of the sieve as follows:

```
(sieve (lambda (thunk) (fork-thread (thunk))) n)
```

Evaluating this application schedules a new thread responsible for filtering all multiples of a prime.

This simple exercise highlights some interesting points about the system. First, *Sting* treats thread operations as ordinary procedures, and manipulates the objects referenced by them just as any other Scheme object; if two filters attached via a common stream are terminated, the storage occupied by the stream may be reclaimed. *Sting* imposes no *a priori* synchronization protocol on thread access - application programmers or language designers are expected to build abstractions that regulate the coordination of threads.

The threads created by **filter** may be determined (i.e. terminated) in one of two ways. The top-level call to **sieve** may be structured so that it has an explicit handle on these threads; the **filter-list** data structure used to create a lazy sieve is such an example. One can then evaluate:

```
(map thread-determine filter-list)
```

to terminate all threads found in the sieve. *Sting* also provides *thread groups* as a means of gaining control over a related collection of threads. A thread group is closed over debugging and thread operations that may be applied *en masse* to all of its members. Every thread is a member of some thread group. Thread groups provide opera-

tions analogous to ordinary thread operations (e.g. termination, suspension, etc.) as well as operations for debugging and monitoring (e.g. resetting, listing all threads in a given group, listing all groups, profiling genealogy information, etc.) Thus, when the thread *T*, the caller of **sieve,** is terminated, a user can request that all of *T*'s children (which are defined to be part of *T*'s group) be terminated thus:

```
(terminate-thread-group (thread-group T))
```

Second, lazy or delayed threads are distinguished from scheduled ones. A lazy thread defines a thread object closed over a thunk and dynamic state (but which is not scheduled on any virtual processor). A scheduled thread is represented by the same data structure, but is scheduled to run on some VP and will eventually be assigned an execution context (see Chapter 2). Applications can choose the degree of laziness (or eagerness) desired. Only the thread controller (see Chapter 4) can initiate a thread transition to the *evaluating* state - the interface does not permit applications to insist that any specific thread immediately run on some virtual processor. Thread policy managers (see Chapter 4) may be fair or unfair. *Sting* imposes no constraints on thread policy managers in this regard.

2.5 Thread Data Structure

Sting threads are implemented by a small data structure, which has several features that distinguish it from other thread systems. This section gives a brief overview of the thread data structure and discusses the reasons for the various fields in it.

The thread data structure consists of the following fields:

Mutex - Since the thread data structure can be accessed by other threads, a mutex controls all access to it.

States - During the course of its lifetime, a thread can be in any one of the following states: delayed, scheduled, evaluating, absorbed, determined. (For a further discussion of these states see Section 2.8.)

Flags - Each thread has several flags that control different aspects of its behavior. For example, the flags determine whether a thread can absorb other threads, and whether the thread maintains information about genealogy, debugging, or performance.

Thunk - Each thread evaluates a nullary procedure. This procedure is associated with the thread when it is created.

Execution Context - When a thread begins evaluating, *Sting* allocates an execution context for it. Execution contexts include a thread control block, a stack, a private heap, and a shared heap.

Values - When a thread completes evaluation it has a value (or values). The value(s) of the thread is the value(s) of the thunk that the thread evaluates.

Waiters - When a thread is evaluating, other threads may be blocked waiting for its value. Each thread can be associated with a set of the threads that are waiting for its completion.

Group - Every thread is associated with some thread group.

Dynamics - Each thread has associated with it a set of dynamic windings, dynamic bindings, and exception bindings.

Virtual Processor - A thread is always associated with a virtual processor.

Priority and Quantum - Each thread has both a priority and a quantum. These fields are used by the thread policy manager to determine the thread's relative scheduling order.

Meta Information - Various kinds of meta information may be associated with each thread. This information may relate to several different aspects of the thread's evaluation including: data that helps in debugging errors or in analyzing performance. Information about the threads genealogy may

also be recorded. Flags control the nature and scope of meta information recorded in each thread.

These fields are packed into a data structure consisting of ten memory words. In contrast, other thread systems do not separate the thread data structure from its execution context. This separation provides the foundation for several important innovations. *Sting*'s thread data structure contains all the necessary information to evaluate the thread, but no more. The small size of a thread allows the user to create a potentially infinite number of threads in a virtual machine. The number of possible threads is only limited by the capacity of the virtual address space, and its backing store.

2.6 Threads have Values

In most thread systems, a thread can only communicate the value of its computation through side effects. This requires synchronization, a significant disadvantage. Since *Sting*'s threads can be used to represent values, it is easy to use them to build distributed data structures[1] which require no synchronization.

A thread may have more than one value and thus can be readily integrated into languages such as Common Lisp, Scheme, ML, and Haskell which support multiple values. Further, because threads have values and they are small data structures they can easily implement any evaluation order from fully lazy to fully eager. Lazy evaluation delays the evaluation of an expression E until the value of E is demanded, i.e. referenced, by some other expression. Eager evaluation is the opposite of lazy evaluation. With eager evaluation the value of an expression is computed before it is needed. In fact, it may never be needed.

2.7 Thread Creation

Sting creates threads by using the fork-thread special form. For example,

`(fork-thread expression virtual-processor)` ⇒ ***thread***

1. Examples of these include Tuple Spaces, Paralations, and Futures.

creates and schedules a thread which will evaluate *expression* on *virtual-processor*. The thread is returned as the value of the **fork-thread** expression. **fork-thread** is actually syntax for the create-thread form. The above expression is syntactically transformed into:

```
(create-thread (lambda () expression) vp) ⇒ thread
```

Sting stores the nullary-procedure created by `(lambda () expression)` in the thunk field of the thread data structure. The thread group for a new thread is the same as that of its parent (i.e. the thread that created it) unless the thread is created as a result of creating a new thread group.

When a thread is created *Sting* captures and stores its dynamic context in the thread data structure. The dynamic context is composed of the dynamic windings, dynamic bindings and the exception bindings in effect in the parent thread at the time of creation. Thus, even though a thread may be evaluated long after it has been created it will be evaluated in the proper dynamic context, i.e. the one in which it was created. We believe that *Sting* is the only thread system which offers this functionality.

Finally, every thread is created with an explicit or default notion of its current virtual processor, priority, and quantum. The thread policy manager uses these three quantities to control thread scheduling and migration strategies, as discussed in more detail in Chapter 4.

2.8 Thread States

In the course of its lifetime, a thread can enter several states:

Delayed - When first created, a thread is in the delayed state. A delayed thread will not evaluate its thunk until its value is demanded either explicitly, by scheduling the thread to execute, or implicitly, by dereferencing the value of the delayed thread.

Scheduled - A scheduled thread is one which has been scheduled to run, on some virtual processor, but which has not yet begun evaluating.

Evaluating - An evaluating thread has started executing the thunk associated with it, but has not yet determined its value.

Absorbed - An absorbed thread is one that is being evaluated in the execution context of another thread. *Thread absorption* is an optimization introduced by *Sting*. It is discussed in detail in Chapter 3.

Determined - A determined thread is one that has completed the evaluation of its thunk, and has stored the resulting values in the thread data structure.

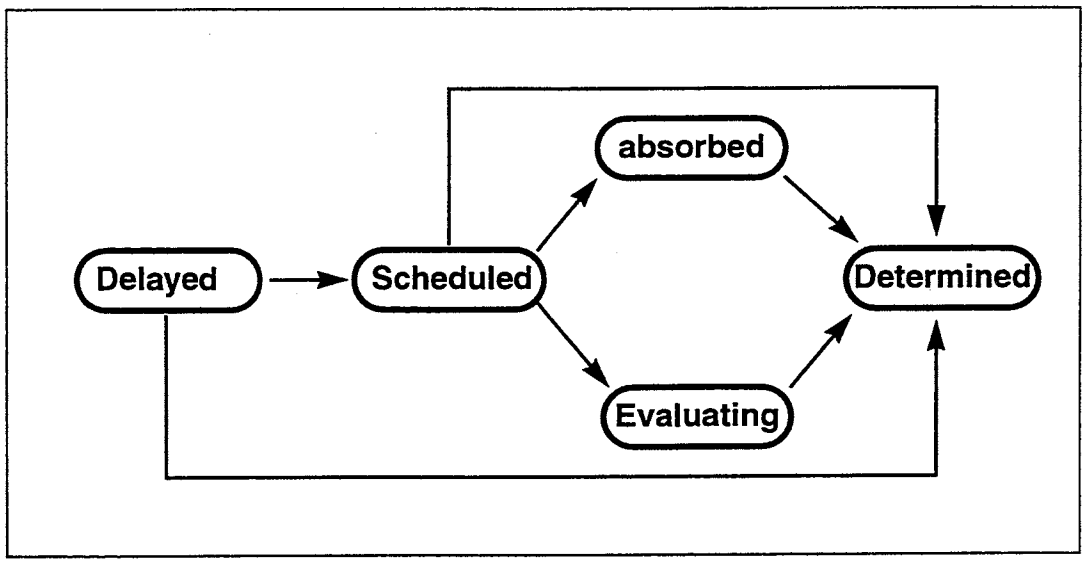

Figure 2-c : Thread States

Figure 2-c is a state transition diagram for threads. When created all threads are in the delayed state. Thus, the most primitive thread creation operation is

(delay-thread *expression* . *virtual-processor*) ⇒ thread

delay-thread takes an expression and an optional virtual processor argument. It creates a thread in the delayed state and returns that thread as its value. A thread in the delayed

state can be forcibly determined, e.g.

```
(thread-determine thread value)
```

This transition, from delayed to determined, occurs without ever executing the thunk associated with the thread. The value of a thread determined in this manner is the value(s) passed to the **thread-determine** procedure.

2.9 Thread Dynamics

Unlike threads in most other thread systems, each *Sting* thread has associated with it a dynamic binding environment, a dynamic exception environment, and a set of dynamically nested windings [HL]. As a result *Sting* threads can be used in such languages as Common Lisp and Scheme which support dynamic (or fluid) bindings. Languages such as modula-2 and -3, Common Lisp, and ML which provide exception handling facilities can also be implemented using *Sting*.

2.9.1 Dynamic Binding Environment

Every thread has a dynamic binding environment associated with it. When a thread is created a new dynamic binding environment is created which is inferior to the dynamic binding environment of its parent at the moment of thread creation. This dynamic environment exists for the life of the thread.

The dynamic binding environment is implemented in *Sting* using a technique known as deep binding. The dynamic binding environment is implemented by a tree that corresponds to the tree of threads on a virtual machine. Whenever a thread is created a new branch of the tree is created for that thread. Within a thread the dynamic binding appears as a stack[1] of bindings, i.e. identifier/value pairs. Each binding has a dynamic scope. It is removed from the environment when the evaluation exits its scope. The value of a dynamic binding is found by traversing the dynamic environment from the current leaf back toward the root until a binding for the appropriate identifier is found.

1. This stack is currently implemented as a list, but more efficient implementations are possible.

2.9.2 Exception Binding Environment

Exception handlers are bound to identifiers in the exception binding environment. The exception environment is a dynamic environment similar to the dynamic binding environment and it is also implemented using a deep binding strategy. The exception environment is a dynamic environment because we wish to allow users to dynamically bind the handlers for specific exceptions. When a thread is created, *Sting* creates a new exception environment inferior to the current exception environment and associates the thread with the new environment.

For example, if procedure P might signal exception E when called, then P's caller might wish to define a handler for E in its (the caller's) dynamic environment. This is so because if A and B both call P, but A wants to handle the exception E, if it occurs, using procedure H_1 and B wants to handle the exception E using H_2 then the two different handlers must be associated with the exception in a dynamic fashion. Dynamic binding is the obvious way to do this.

Exceptions are procedures. A exception is signalled by simply calling the exception with the appropriate arguments.

```
(exception arg1 arg2...)
```

When invoked, the exception searches for the current handler for the exception in the exception binding environment and then invokes the handler with the same arguments passed to the exception. Thus, all handlers for an exception must have the same signature.

Many different exception handling mechanisms can be used with *Sting*. *Sting* does not implement any particular exception handling discipline, rather it provides the building blocks for whatever style exception handling mechanism a particular language requires.

2.9.3 Dynamic Windings

In any program where it is possible to use side effects, it is desirable to restore objects to a consistent state when control leaves the current dynamic context for whatever reason (including a throw to a catch point). In particular, this is important when communicating with objects external to a program which have state, such as file systems, or input/output devices. For example, an exception handler for an end-of-file exception may decide to throw out of the current dynamic context to some previous dynamic context. If this occurs then it might be important to ensure that the file is closed.

Sting provides a dynamic-wind primitive which allows the user to define dynamic winders and unwinders for returning (throwing) into and out of a continuation.

```
(dynamic-wind before during after)
```

The **dynamic-wind** form takes three arguments which are thunks. It evaluates the *before, during,* and *after* thunks in that order. *Sting* guarantees that whenever control flows into *during* the *before* expression will be executed. It also guarantees that whenever control flows out of *during* the *after* expression will be executed. This is true even if control flows into or out of *during* more than once or because of a throw.

2.10 Thread Groups

Sting allows the user to aggregate threads that cooperate on a particular computation or sub-computation. These aggregates, called *thread groups*[1] improve performance and enhance debugging. When a thread group is created by a call to **fork-thread-group** (see Figure 2-e) the root thread of the group is created at the same time. *Sting* associates any threads that are descendents of the root thread of a group with the same group. When created a thread is thus placed in the same group as its parent, unless the thread is created as the root thread of a new group. As with virtual processors and threads, thread groups are first class, with the concomitant benefits.

1. Thread groups were used in Mul-T but did not contain the functionality that they do in *Sting*.

Sting organizes thread groups in a genealogy tree just as it does threads. The root of the thread genealogy tree is the *root thread group* of the virtual machine on which it is running. The root thread of the root thread group is the root thread for its virtual machine. When *Sting* creates a virtual machine, it also creates both the root thread group and the root thread of that virtual machine. Figure 2-d shows an example thread group structure. The circles represent threads and the rectangles denote thread groups. The circles (threads) contained in a rectangle (thread-groups) are the threads in the group represented by that rectangle. The edges between the circles show the genealogy of the threads.

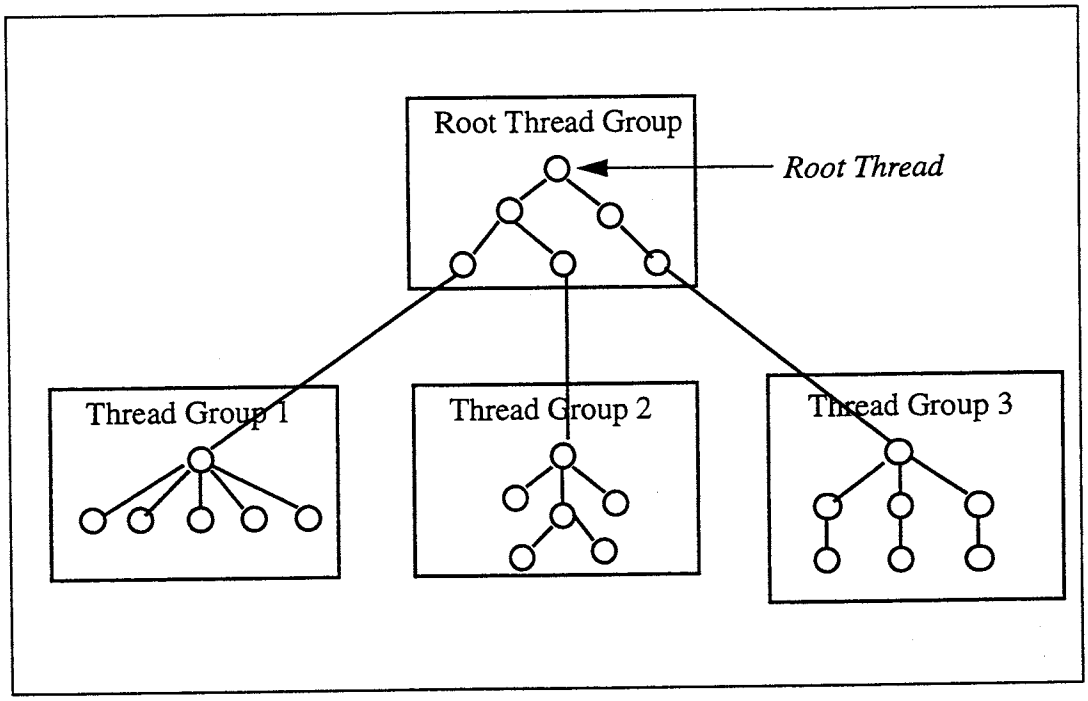

Figure 2-d : Thread Group Hierarchy

Sting allocates a new shared heap[1] to each thread group that is created. Every thread in the group uses the shared heap to allocate data that will be shared by other threads in the group. When a thread group terminates, *Sting* garbage collects all the live threads

1. Shared heaps and garbage collection are discussed in Chapter 3.

in the group as well as the shared heap.

Thread groups can be blocked, suspended, or terminated in a manner similar to threads. When any of these operations are performed on a thread group all the threads in the group make the requested transition if possible. Threads not in a group G can wait on the completion of G. Figure 2-e shows the interface to thread groups. The reader interested in the details of the interface should see [Phi93].

The operations on thread groups are not quite as straightforward as those on threads. When a request is made to block or suspend a thread group all threads in that group are blocked or suspended, but the threads maintain their own state, so that when the thread group is resumed, only those threads that are *ready* or *running* will actually be resumed.

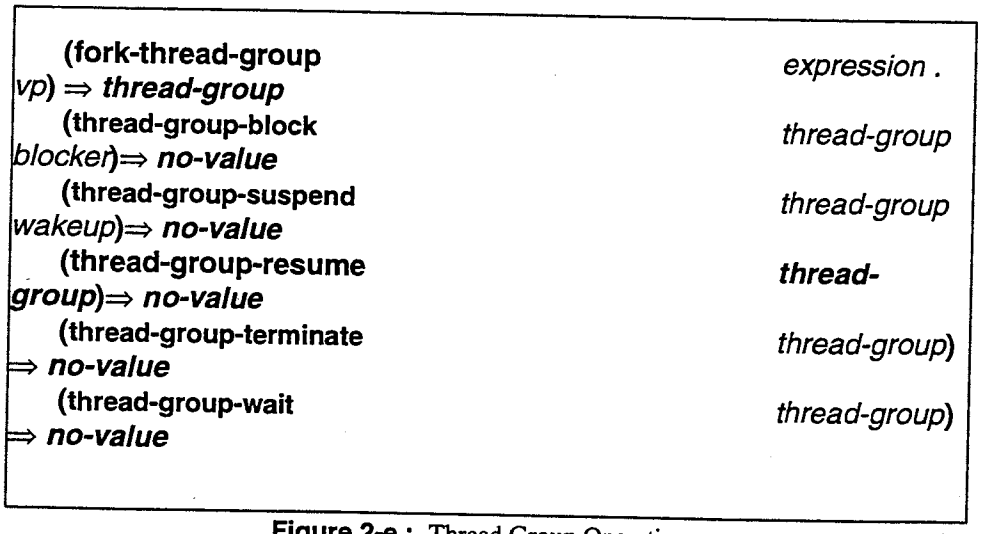

Figure 2-e : Thread Group Operations

Thread groups are an important tool for controlling memory sharing in a hierarchical memory architecture. Since objects shared by a thread group are contained in the group's shared heap, they are grouped "close"[1] to each other in memory and thus have better locality. When using a machine with a hierarchical memory structure, it is advantageous to map threads in the same group onto processors that are "close" in the

1. By "close" we mean that take a similar amount of time to access the same address in memory.

hierarchy. For example, if a parallel machine is created out of shared memory multi-processors, then performance will probably be much better if the threads in a group are scheduled on the same shared memory multi-processor, assuming that the number of processors is similar to the number of threads.

Data parallel programming constructs can also benefit from using thread groups because of the locality of the shared heap associated with the group. First class tuple spaces [Jag91] are an example of such a data parallel construct. If the active tuples in a tuple space as well as the threads which share access to the tuple space are organized into a thread group, then the tuples will be localized in the thread group's shared heap. As a result, both access to tuples and garbage collection of the tuple space will be much faster.

Thread groups can also act as a locus of scheduling. For example, the thread policy manager[1] could implement a strategy where no thread in a group is scheduled to run unless all threads in the group are scheduled to run.

Thread groups are also useful for debugging. When a thread in a group encounters an error, the error handler can suspend all threads in that group and invoke the debugger. The debugger can then be used to inspect the state of the thread that encountered the error, or any other thread in the group. *Sting*'s default error handler exhibits this behavior. With *Sting* the user can also signal a thread group to suspend execution and then inspect any or all threads in the group.

It should be noted that the use of thread groups in the *Sting* system is an optional organizing tool that has proven useful in practice, but is not mandatory.

2.11 Thread Meta Information

Each thread has a set of flags associated with it that the system uses to determine whether certain information should be gathered in the course of evaluating the thread.

---

1. See chapter 4.

Currently the system gathers three kinds of *meta* information: genealogy, performance, and debugging.

The ability to gather this sort of information must be an integral part of the design of a system supporting parallelism, not an afterthought. Many modern programming language environments have been built without sufficient attention to tools for understanding both the behavior and performance of the various constructs in the language. This is particularly true for parallel programming environments. Meta information allows the various tools for debugging and analysis to be constructed.

2.11.1 Thread Genealogy

Each thread has its own genealogy, i.e. each thread can know it's ancestors, and its descendents and their relative ages. The genealogy tree is constructed so that the oldest or initial thread is the root of the tree and each succeeding generation is ordered from oldest to youngest in that tree. Figure 2-f shows a thread genealogy tree. Each thread T in the tree is labeled with a generation number from 0 to i, with zero being the oldest generation, and a relative age in that generation from 0 to j, with 0 being the oldest member of that generation.

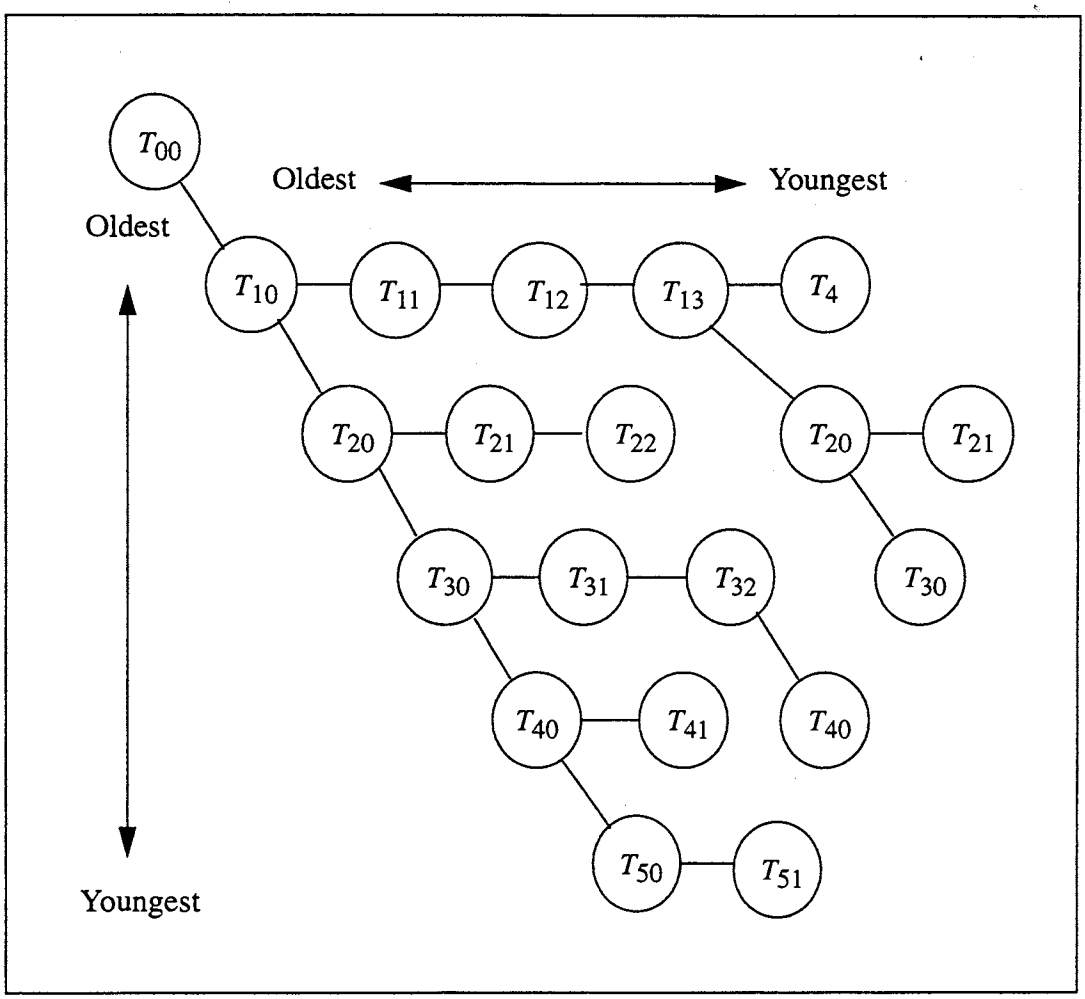

Figure 2-f : Thread Genealogy

Thread genealogy is useful for both debugging and algorithm analysis. It allows a thread encountering an error to examine not only it's parent and siblings, but also, the organizational structure of the entire computation at the approximate point the error occurred. A serious discussion of debugging parallel applications is beyond the scope of this thesis, but we believe the need for genealogy information is a fundamental prerequisite for debugging parallel applications.

The second important use of genealogy information is in algorithm analysis. Genealogy can be used for this in two ways. First, it lets the designer examine the asynchro-

nous and communication structures of an algorithm. This can be done on the fly as the computation is occurring or as a post-mortem analysis after the computation has completed by using a visualization tool which displays the dynamics of the genealogy structure as the computation evolves.

Genealogy also aids algorithm analysis buy allowing the user to evaluate the performance of an algorithm on a thread by thread basis. This is done by using performance information (see below) in conjunction with genealogy information. Since threads are small data structures it is possible to retain the entire thread tree for post-mortem analysis.

Sting allows the user to keep thread genealogy in one of two ways; either the genealogy contains all the threads both living or dead, called complete genealogy, or the genealogy contains only live threads, called living genealogy. The type of genealogy maintained, if any, depends on the setting of genealogy flags in the thread data structure.

2.11.2 Performance Information

The performance information gathered by a thread includes traditional measurements such as creation time, completion time, time spent in user mode and system mode, spent time waiting in user mode and kernel mode, number of page faults, and number of I/O blocks. In addition, *Sting* allows the user to gather other information not normally recorded, including: maximum stack size, amount of private heap and shared heap storage allocation, the number of private garbage collections that a thread has incurred, and the number of shared heap garbage collections that a group has incurred. Performance data can also be consolidated at the thread group, virtual processor, and virtual machine level.

2.11.3 Debugging Information

Most of the above information is sufficient in combination with a typical debugger to debug multi-threaded applications. There are, however, three other useful pieces of information that can be recorded by a thread.

First, whenever a thread blocks, it can record the object on which it is blocked. This may be another thread, a mutex, a condition variable, or some user defined synchronization object. This information helps not only to debug synchronization problems, but can also help locate deadlocks when they exist.

Second, it is possible to log the history of a thread's state changes along with time stamps. This information is useful for building an animated graphical representation of a computation, either on the fly as the computation takes place, or post mortem.

2.12 Thread Migration

Sting allows threads to migrate from processor to processor, but the system is designed to help minimize and amortize the cost of migration. There are two features of *Sting* that reduce this cost.

The first results from the small size of the thread data structure. It is much cheaper to migrate threads which are either in the delayed or scheduled state, because they do not yet have execution contexts and only the minimal amount of data must be moved to another machine.

The second aspect of the design that promotes economical thread migration relies on the shared virtual memory[1] to lazily copy pages or cache lines from one processor to another. This enables the rest of the system to migrate a thread from one processor to another simply by removing a reference to it from the ready queue on one processor and inserting it into the ready queue on another processor. When the thread is resumed on the new processor the data required by the thread, e.g. it's TCB, stack, etc., is

1. See Section 3.5, Section 4.3, and Section 5.5.1.

faulted over to the new processor as needed. In most applications the data associated with a thread is private to it and migrating the data from one processor to another will not increase memory contention.

2.13 Thread Termination

Threads terminate or complete either because the thunk which the thread evaluates has returned, or because the user invokes the thread's exit handler, a dynamically enclosing continuation. Figure 2-g shows a simplified version of the procedure in which each thread's thunk is wrapped. All threads start evaluation by invoking this procedure.

```
1: (define (start-thread thread)
2:   (let ((z (no-value)))
3:     (catch exit
4:       (dynamic-wind
5:         (lambda ())
6:         (lambda ()
7:           (set-exit-handler! thread exit)
8:           (set! z ((thread.thunk thread))))
           ;; Thread termination
9:         (lambda ()
10:          (set-thread-value! thread z)
11:          (wake-up-waiters thread)
12:          (thread-state->dead))))))
```

Figure 2-g : Thread Startup and Termination

The first thing the **start-thread** procedure does is ensure that the termination code is executed, even if the exit handler for the thread is invoked. This is done by creating a temporary variable **z** to hold the value of the thread (line 2), and then wrapping the evaluation of the thread's thunk in an **dynamic-wind** form (line 3). The *before* argument (line 4) to the dynamic wind is a procedure that returns no value, because the

thread has no state associated with it which needs to be rewound if it is entered again.[1] In the *during* argument (lines 5 - 8) a downward only (dynamic) continuation is created using **catch** and stored as the exit handler of the thread (line 7). In line 8, the real work of the thread is performed by evaluating the thunk associated with the thread.

The *after* argument (line 9 - 12) ensures that the thread competes properly. It does this by first storing the value computed by the thunk in the thread and then setting the thread state to *determined* (line 10). *Sting* then resumes all threads waiting for the value of this thread (line 11), recycles the execution context,[2] and then calls the thread controller to run a new thread (line 12).

This wrapper procedure around the thread's thunk controls the evaluation environment of the thread's thunk. Language designers can use wrappers to augment the semantics of thread start-up and completion without necessitating any change in the thread system.

2.14 Thread Performance

Table 2-a, "Thread Performance," on page 48 shows performance numbers for threads in our implementation of *Sting*. The performance measurements were made on a Silicon Graphics Personal Iris with one 25 megahertz MIPS R3000 processor.

1. See the discussion of dynamic winding in Section 2.9.3 on page 38.
2. See Chapter 3.

Test	Time in μseconds
1: Thread Creation	8.9
2: Scheduling a Thread	18.9
3: Synchronous Context Switch	3.7
4: Thread Fork and Value	44.9
5: Thread Absorption	7.7
6: Thread Block and Resume	27.9
7: Speculative Fork (2 threads)	68.9
8: Barrier Synchronization (2 threads)	144.8

Table 2-a : Thread Performance

Sting is currently written in the T dialect of Scheme and is compiled using the Orbit Compiler [KKR+86]. Orbit was designed for CISC architecture machines and does not generate particularly good code for RISC machines. In addition, Orbit does not perform many optimizations, such as inter-procedural analysis, which would enhance the speed of all the test programs.

The first performance number is the time to create a thread and capture it's dynamic context.[1] The second benchmark above is the time it takes a user to schedule a delayed thread to run on a simple lifo scheduler. The third test shows the cost of doing a context switch, i.e. `(yield-processor)`. The fourth benchmark shows the time needed to execute the form

```
(thread-value (fork-thread 0))
```

This time represents all the overhead associated with the use of threads. It includes the cost of creating, scheduling, evaluating (including context switch in and out), and cleaning up the resources associated with a new thread.

The fifth test is exactly the same as the fourth test except that the thread is evaluated

1. While this time is competitive with other thread systems, it currently takes more than twice the number of instruction necessary to create the thread. This is because T performs storage allocation by using a subroutine with approximately 20 instructions. The next version of the system will use a storage allocator which takes 2 or 3 instructions to allocate a piece of storage.

using thread absorption,[1] i.e. is evaluated in the execution context of the thread demanding its value. It shows that thread absorption is significantly faster than actually allocating the execution context for the thread. Test six shows the cost of blocking and resuming a running thread.

Finally, tests seven and eight show the costs of two types of synchronization. In test seven, two threads are forked but when one thread completes the other thread is terminated and the value of the completing thread is returned. In test eight, two null threads are forked and the caller waits until both threads have completed.

While the above numbers are not directly comparable with those published for other thread systems which do not run on SGI hardware, they are however very competitive and we believe that when compiled by a more sophisticated compiler than Orbit, most of the above timings would be improved by 100%.

1. Thread absorption is discussed in detail in Chapter 3.

Chapter 3

Thread Evaluation and Execution Context

3.1 Introduction

A thread, when either delayed or scheduled, is a small data structure, as explained earlier, but when a thread begins evaluation *Sting* allocates an execution context for it. A thread's execution context is composed of a thread control block (TCB), a stack, a local heap, and a shared heap, which is inherited from its thread group. Delaying the allocation of the execution context until a thread begins executing provides *Sting* with several advantages in terms of storage utilization and data locality. These in turn lead to significant improvements in efficiency. While threads are first class, execution contexts are not. Execution contexts are internal to *Sting* and are invisible to the user. This allows them to be reused when a thread completes, and also leads to other opportunities for optimization. The various components of the thread execution context are discussed throughout the rest of this chapter.

The thread control block is a record which contains information relevant to the current state of the evaluating thread. In many respects, it is analogous to a process control block in traditional operating systems, It contains information about the state of an evaluating thread and space to save the thread's VP state on a context switch. It also contains references to the stack, the private heap, and the shared heap.

A thread uses its stack in the traditional manner, for allocating objects, including pro-

cedure call frames. Stack allocated objects are only accessible to the associated thread and their lifetime is known not to exceed the lifetime of the procedure which created them. The stack is different from those in traditional languages such as C, Fortran, or Pascal, in that it contains not only parameters and local variables, but can also contain data created dynamically that only exist for the dynamic extent of the procedure call. The compiler determines whether or not an object can be allocated on the stack.

Objects that are only accessible to the thread that created them and whose lifetimes *may exceed*[1] the lifetime of the procedure that created them are allocated in the private heap. Objects in the private heap can never be referenced by any thread other than the one that created them. The fact that both the stack and private heap cannot contain shared data allows them to be allocated in physical memory that is local to the physical processor. Accessing such data is fast, because there is no contention for it and it is trivially coherent. This is particularly important on disjoint, or partially disjoint memory systems.

The shared heap is associated with all the threads in a thread group. Shared objects, i.e. those known to be accessible to threads other than the one which created them, are allocated in the shared heap (see Section 3.5.5). Thus the memory coherence problem which occurs on all shared memory multi-processors only applies to shared heaps. Private heaps and stacks can be implemented in memory that is not coherent with respect to the rest of the processors because no other processor will access it. Both the private and shared heaps are actually a series of heaps organized and garbage collected in a generational manner. Since a thread's private heap is inaccessible to other threads, i.e. cannot contain objects which are referenced from other threads, they can be garbage collected independently without stopping the evaluation of other threads in the system. The *Sting* garbage collector is quite novel. It is discussed in Sections 3.5.4 and 3.5.5 below.

1. That is, the compiler cannot determine that the object's lifetime does not exceed the lifetime of the creating procedure.

3.2 Comparison with Other Systems

There have been many different approaches to handling the memory requirements for implementing control state in various parallel languages and thread systems. These include: the stack and data segments of languages like C and Pascal, the cactus stacks of Mesa, spaghetti stacks in Interlisp, the similar macaroni stacks of Steele, and finally heap allocated control state in implementations such as ML/NJ [App90] and Mul-T [KHM89]. There are various restrictions and costs associated with these techniques for implementing control state. The *Sting* approach reduces the cost of implementing control state while at the same time removing some of the restrictions associated with the above mentioned strategies.

Similarly, there have been many different approaches to handling the memory requirements for data, other than that used for implementing control state, in a program. These include: the static allocation of local and common areas in Fortran, data segments in languages like C and Pascal where allocation is static (compile time) or dynamic (malloc and free), and heaps in languages which support automatic storage management including garbage collection.

Reducing the absolute number of execution contexts created during the evaluation of a parallel program is a related problem to reducing the cost of a program's memory requirements. This is often referred to as constraining the amount concurrency found in a program. Several different methods of constraining concurrency have been proposed. These include a master/slave strategy such as used in WorkCrews [VR88], and load based inlining and lazy task creation as implemented in Mul-T [MKH90]. *Sting* introduces a new method of constraining concurrency called *thread absorption.* We compare thread absorption with these other approaches in the appropriate sections below.

3.3 Thread Control Blocks (TCBs)

When a thread begins evaluation a TCB is allocated for it either from the TCB pool

associated with each virtual processor, or if that pool is empty from the global pool of TCB's. The TCB data structure has a size on the order of 100 words of storage. In order to improve locality the TCB is allocated at the base of the stack. Thus the base of the stack and the TCB are allocated on the same page in memory.

The TCB data structure contains the following fields:

State - During the course of evaluating the thread the TCB can enter several different states. We refer to these as thread evaluation sub-states, or simply TCB states.

Exit Handler - Each evaluating thread has an exit handler which is used to unwind the stack and deallocate resources associated with the thread. The exit handler is discussed in Section 2.13 on page 46.

Next Waiter - If this thread is waiting for some other thread to terminate, this field is used to record that fact. The set of threads waiting for a thread to complete evaluation is implemented as a list anchored in the **thread-waiters** slot of the thread being waited on and threaded through the TCB **next-waiter** field.

Blocker- Whenever a thread blocks (or suspends) the object on which the thread blocked is recorded in this field. This information is never used by the thread system, but it is supplied for debugging purposes.

Wakeup Time - This field is used to record the time at which a suspended thread should be resumed.

Saved VP State Block - This is space in the TCB for storing the entire state of the VP when the thread is releasing control of the processor for whatever reason. Context switching and the use of the saved VP state block is discussed in Chapter 4.

Local Stack - An area of memory used for storing objects created during the execution of the thread that are private to the thread, i.e. they are never ref-

erenced by any other thread, and whose lifetimes do not exceed the dynamic extent of the procedure which created them.

Local Heap - An area of memory used for storing objects that are private to the thread, and whose lifetimes may exceed the dynamic extent of the procedure which created them.

Shared Heap - An area of memory used for storing objects created by the thread which are shared with other threads.

The use of these fields will be discusses in more detail throughout the rest of this chapter.

3.3.1 TCB States

Once a thread begins evaluating, it may enter several possible sub-states. These states are associated with the thread control block. An evaluating thread's TCB can be in any of the following states:

- *initialized* - The TCB has been initialized but is not currently associated with any thread, rather it is contained in the pool of initialized TCB's associated with each virtual processor or in the global TCB pool.
- *ready* - The thread is ready to be run, but is not currently running on any virtual processor.
- *running* - The thread is currently executing on some virtual processor.
- *blocked* - The thread is blocked, i.e. not able to run, waiting for some unspecified event to occur.
- *suspended* - The thread has been suspended either indefinitely or for some specified time.

- *terminating* - The thread has begun terminating it's execution, either because it has finished it's computation or because it has been requested to terminate it's computation prematurely.

Figure 3-a shows a state transition diagram for both threads and TCB's. The TCB states can be regarded as thread evaluation sub-states.

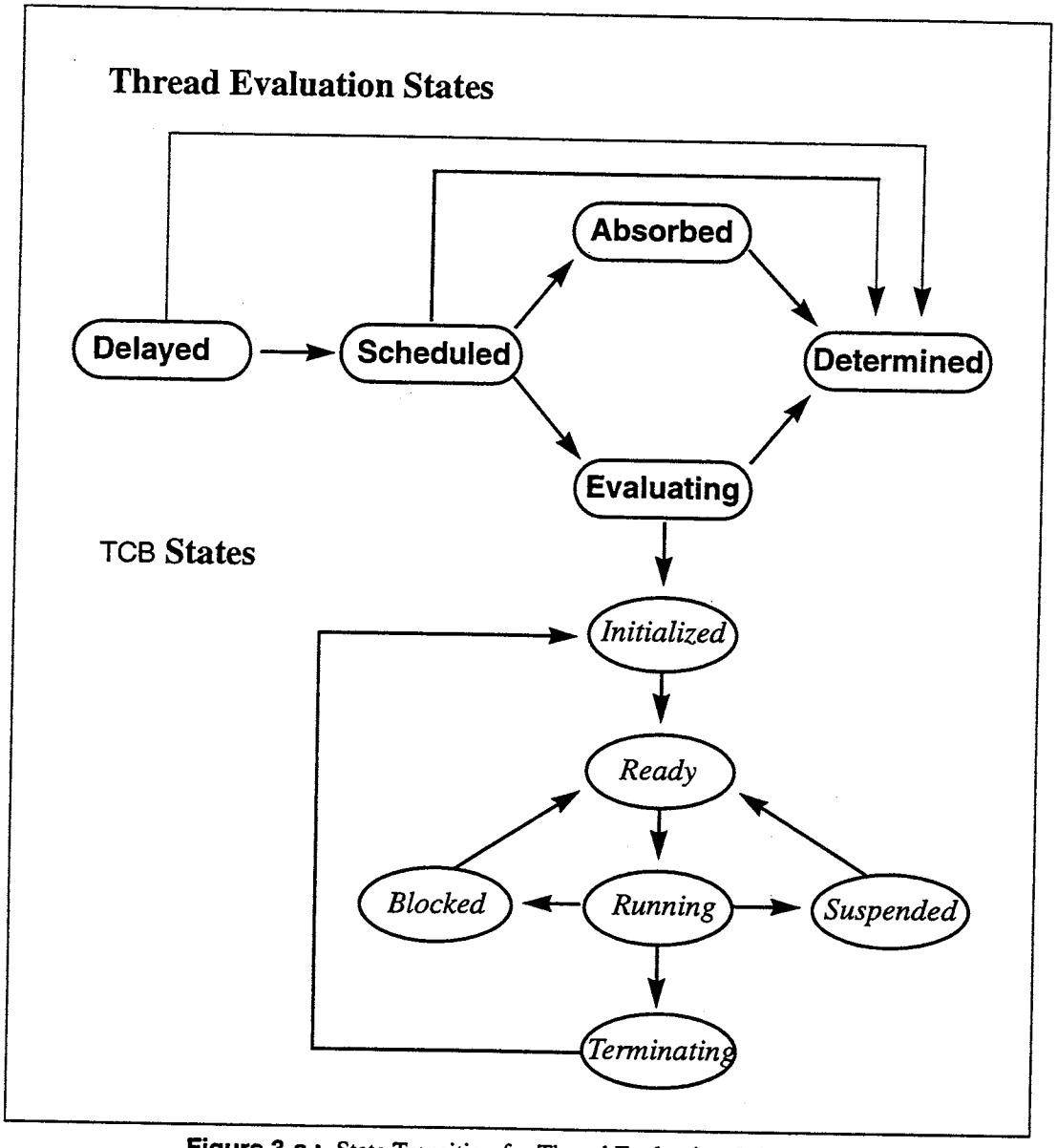

Figure 3-a : State Transition for Thread Evaluation Sub-States.

When a thread has finished evaluating, i.e. has become determined, its TCB is re-initialized and its entire execution context is returned to the pool of execution contexts associated with the thread's virtual processor. This pool is arranged in a LIFO order so that when a new context is allocated it is the one most recently used, and thus is likely to be already loaded into the caches and main memory of the system.

Evaluation sub-state transitions[1] are accomplished more quickly than in other operating systems, because there is no need to lock either the thread or the TCB on a sub-state transition. This is possible because only the thread itself changes its TCB state. Or, said another way, it is the thread itself which calls the thread controller when it wants to change its state and the thread controller's procedures run in the execution context of the thread which calls them.

3.3.2 Advantages of Delaying Execution Context Allocation

Sting does not assign an execution context to a thread until it begins evaluation. Delaying the allocation of the execution context until thread evaluation time results in several advantages in storage conservation and locality of reference.

Since execution contexts are internal to the *Sting* system, i.e. a user program can never gain access to them. Execution contexts can be recycled for use by other, as yet unevaluated, threads when a thread completes executing. Execution contexts that have been created but which are not currently associated with an evaluating thread are pooled, in a lifo manner, on the various virtual processors.

Sting allocates an execution context to a scheduled thread when it begins evaluating on a virtual processor. The allocation strategy is designed to improve data locality. The execution context is allocated in one of four ways:

- If the thread executing on the virtual processor just prior to beginning the evaluation of a new thread has completed execution, its execution

1. Evaluation sub-state transitions are similar to thread state transitions in other thread system, because these systems assume that a thread is always in an "evaluating" state.

context is available for immediate re-allocation. Further, its execution context is the best candidate for allocation because it has the most locality relative to the virtual processor, i.e. the physical memory and physical caches associated with the virtual processor are most likely to contain the execution context that has been most recently used.

- If the thread executing prior to starting a new thread has not finished its execution, *Sting* allocates an execution context from a lifo pool of execution contexts associated with the virtual processor it is about to run on. The execution context allocated is again the one with the most locality, since it is the most recently used of the execution contexts available.

- If the pool of execution contexts associated with the virtual processor is empty, then the execution context is allocated out of the global pool of execution contexts which, like the VP local pools, is organized as a lifo queue, giving it the best chance for memory residence.

- Finally, if the global pool of execution contexts is empty *Sting* creates a new execution context and allocates it to the thread. Since this execution context has never been used before, it has no locality. It should be pointed out, however, that this is the case which occurs least often.

The local pools of execution contexts on each VP interact with the global pool of execution contexts in the same manner as those of Anderson, et. al. [ALL89]. There is a parameter P_{max} associated with each virtual machine that is the maximum number of execution contexts in a VP pool. If a VP pool overflows, i.e. has more than P_{max} execution contexts returned to it, it returns half of the execution contexts in the local pool to the global pool. If a VP pool underflows, i.e. has no execution contexts in it when one is allocated, then the VP acquires one half of P_{max} execution contexts from the global pool. This strategy keeps the number of execution contexts in each VP's local pool relatively balanced while at the same time avoiding contention on the global pool.

Delaying execution context allocation also minimizes the number of execution con-

texts in use at any one time, since the number in use corresponds to the number of threads that are currently being evaluated, not to the number of threads that have been created.

Delaying execution context allocation has the additional advantage of reducing the cost of thread migration from one virtual processor to another. It is much cheaper to migrate a thread that has not yet begun evaluating, and therefore has no execution context, because only the thread data structure, which is small, is moved. In order to migrate an evaluating thread, all of the objects in the execution context must also be migrated. This can be done eagerly, by copying the context from one VP to another, or lazily, by "faulting" the cache lines and pages associated with the execution context to the new VP as they are referenced.[1]

The final advantage of delaying execution context allocation is that it allows *Sting* to perform an optimization called *thread absorption.*

3.3.3 Thread Absorption

Thread absorption[2] is a novel optimization introduced by *Sting* that has the effect of improving locality of reference and reducing the storage required for the evaluation of threads. Thread absorption can occur when one thread must wait for the completion of another thread which is not yet evaluating. Waiting for completion occurs most often because one thread demands the value of another thread. This optimization is especially important in fine grained and data parallel programming models.

For example, thread absorption can occur when a thread, T_1, requests the value of another thread, T_2. If T_2 is not determined then T_1 blocks until the value of T_2 becomes available. However, in the case where T_2 has not yet begun evaluating, it is possible for T_1 to evaluate T_2 using its own execution context. When thread T_2 is eval-

1. Lazy migration is only possible because *Sting* is base on a shared virtual memory.
2. Thread absorption was called thread stealing in earlier papers on *Sting*. Thread Absorption is a more descriptive term and it is not easily confused with the term stealing as used in lazy task creation.

uated using the execution context of T_1 we say that T_1 has *absorbed* T_2. Thread absorption does not slow down the evaluation of T_1 since by definition T_1 has to block until T_2 is finished evaluating. In fact, it speeds up the evaluation of T_1 because T_1's execution context is not switched out with the attendant loss of locality.

Thread absorption results in reduced storage requirements since it avoids the allocation of an execution context by using the same one to evaluate two different threads. More importantly, it results in improved locality of reference for two reasons. First, the execution context of T_1 is already loaded in the physical memory and caches of the processor, so evaluating T_2 in T_1's context will cause fewer cache and page faults. Second, the value computed by T_2 will already be in the cache when T_1 resumes evaluating.

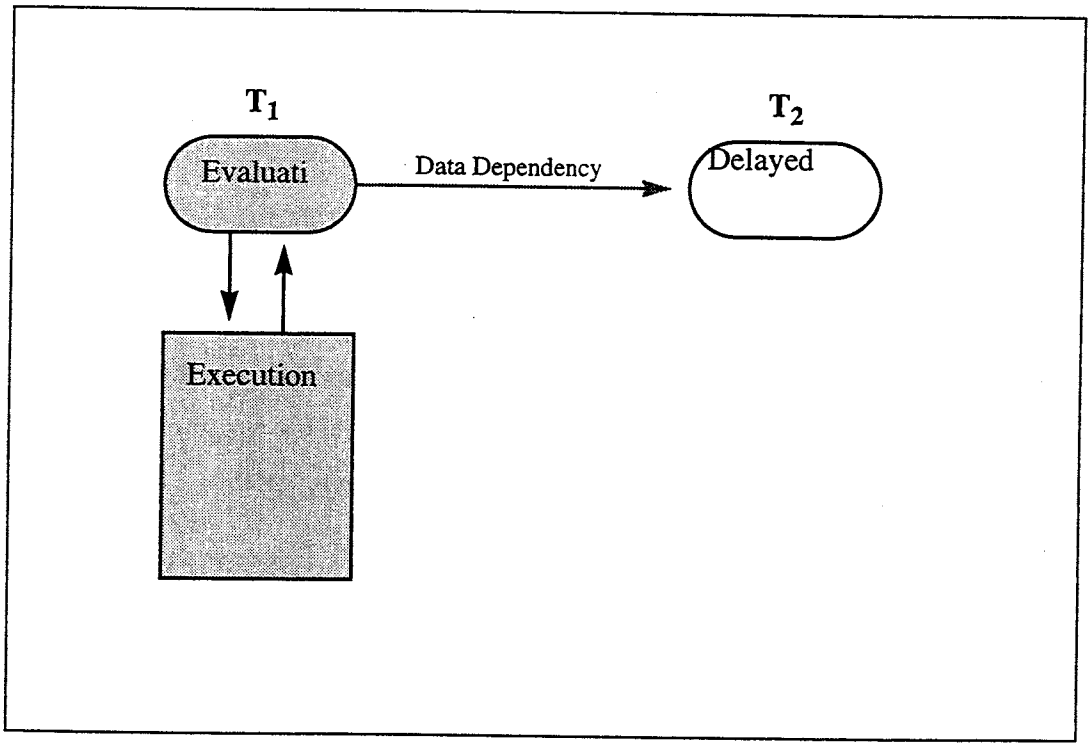

Figure 3-b : Before Thread Absorption

It is easiest to explain thread absorption by using an example. Figure 3-b shows a thread, T_1, which is evaluating and is about to wait for the completion of another

thread, T_2, either because it has demanded it's value, **`(thread-value T2)`**, or is merely synchronizing with it, **`(thread-wait T2)`**. T_2 has not yet begun evaluating, i.e. it is either in the delayed or scheduled state. This allows us to evaluate T_2 using T_1's execution context. Thread absorption relies on two conditions:

- T_1 cannot proceed until T_2 completes and thus its execution context will be inactive until that time, and
- T_2 does not yet have an execution context because it has not yet begun evaluating.

Figure 3-c is similar to Figure 3-b except the T_1 has dynamically absorbed T_2 and T_2 is now running using T_1's execution context.

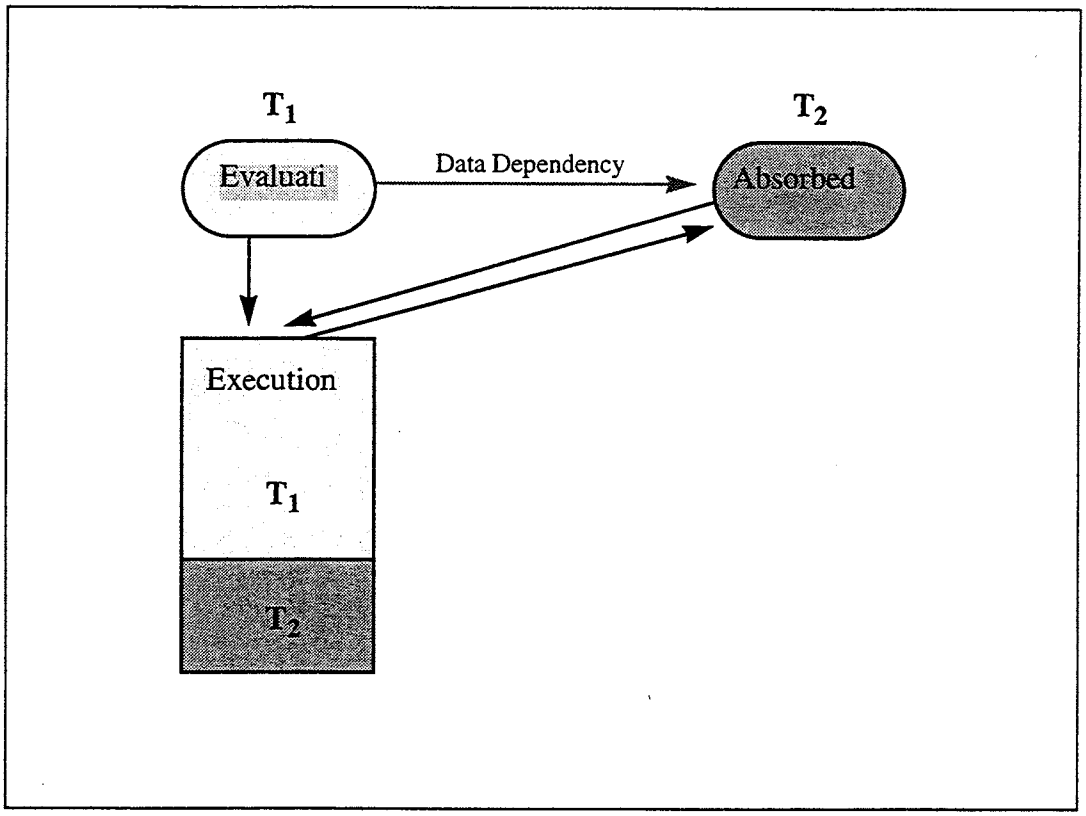

Figure 3-c : Thread T_1 has Absorbed T_2

The thread absorption has been accomplished by making T_1's execution context refer to T^2 and then starting T_2 by simply making a procedure call. Figure 3-d shows the states of T_1 and T_2 after T_2 has finished evaluating and T_1 has resumed.

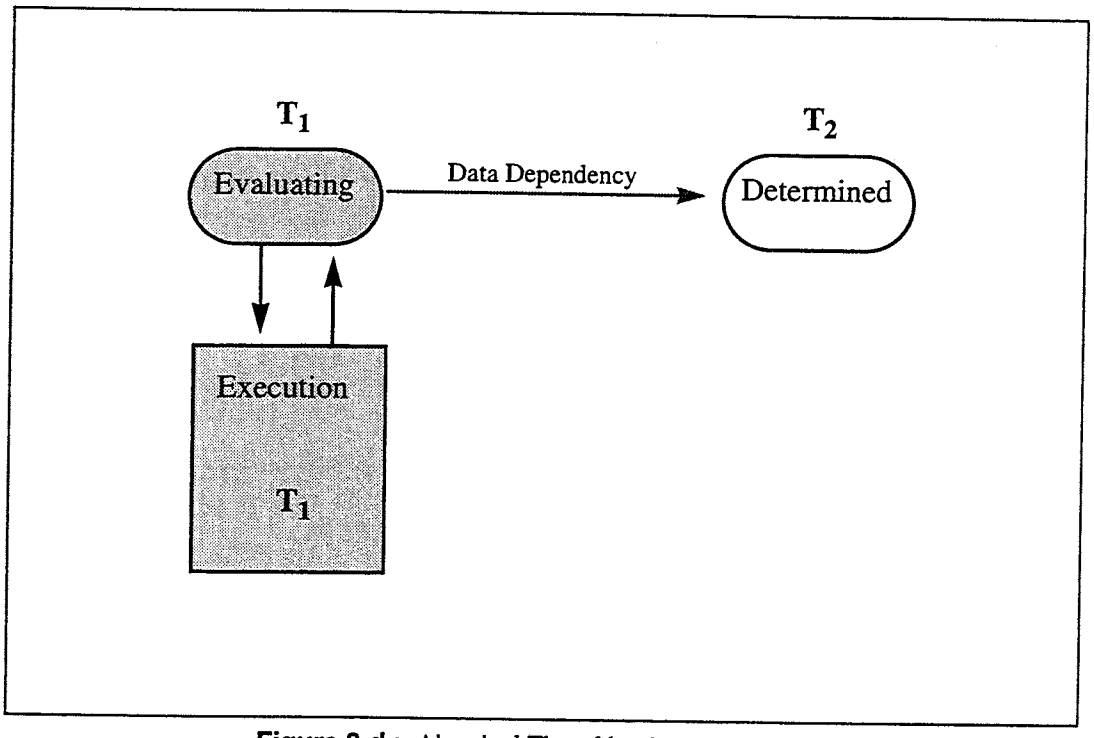

Figure 3-d : Absorbed Thread has been Determined

In order to look at the mechanics and simplicity of thread absorption, we show example Scheme code for it in Figure 3-e .

```
1:  (define (thread-absorb thread)
2:    (let ((absorber (current-thread)))
        ;; setup absorbed thread
3:      (thread-state->absorbed thread)
4:      (set-thread.vp!  thread (current-vp))
5:      (set-thread.tcb! thread (current-tcb))
6:      (set-tcb.thread! (current-tcb) thread)
        ;; evaluate the absorbed thread
7:      (let ((val (thread-start thread)))
          ;; finished evaluating thread
8:        (set-thread.vp! thread (current-vp))
9:        (set-tcb.thread (current-tcb) absorber)
10:       val)))
```

Figure 3-e : Code for Thread Absorption

The **thread-absorb** procedure is called by the absorber, i.e. the thread, in our example T_1, that is about to block on an unevaluated thread, in our example T_2. In line 2 the current thread is saved so that we can resume it when the absorbed thread completes. In lines 3 to 6 the thread and TCB data structures are changed to make the execution context belong to the absorbed thread. This is done by changing the absorbed thread's state to *absorbed* (3), associating the current virtual processor with the absorbed thread (4), and giving the TCB and thus the execution context to the absorbed thread (5 and 6).

The absorbed thread is evaluated by simply calling the **start-thread** (see Figure 2-f) procedure with the absorbed thread as its argument. The last thing that start-thread does before it returns is a garbage collection of the private heap. This is usually a opportune time to do a garbage collection because all the local objects created by the absorbed thread (in its stack and private heap) will be dead (unreachable) except for the value of the absorbed thread.

When the absorbed thread completes it simply returns its value to the **thread-absorb** procedure. We should note that the absorbed thread executes in the dynamic context

that was extent when the thread was created, and thus the dynamic windings and bindings are those appropriate to the absorbed thread. This occurs because the dynamic context of the absorbed thread was captured when the thread was created, and in was stored in the thread's data structure. Thus we can change the dynamic context at no cost when we absorb a thread.

After the absorbed thread completes and control returns to the **thread-absorb** procedure the current virtual processor is stored in the absorber's thread data structure (8), because the execution context might have migrated to a different processor while it was evaluating the absorbed thread. Then the execution context is returned to the absorber (9). This takes only one instruction since the absorber's **thread.tcb** field still refers to the execution context. And finally the value of the absorbed thread is returned to the absorber which resumes execution automatically.

Thread absorption is particularly useful for languages which support lazy evaluation. This is because delayed threads can be used to implement lazy values. When the lazy value is accessed using **thread-value** the delayed thread will automatically be absorbed.

Furthermore, the overhead of allocating a dynamic context and evaluating a thread in that context is as least 4 to 5 times more expensive then the overhead of thread absorption (see Table 2-a).

3.3.4 Limiting Concurrency: Related Work

There are two other optimizations which are similar to thread absorption, load based inlining[1] and lazy task creation. These strategies have only been used in the context of Multi-Lisp, but they can be applied to other parallel languages. We should note that it would not be possible to apply these optimizations to other thread systems such as Mach, Topaz, or C Threads because they allocate the execution context at the time of thread creation.

1. The use of the term *inlining* is an unfortunate one, because it confuses the notion of textual substitution, as when a compiler *inlines* a procedure, and the notion of avoiding the creation of a thread.

3.3.4.1 Load Based Inlining versus Thread Absorption

Load based inlining is a technique that was tried in both Multi-lisp and Mul-T. The idea is that when the load on a processor reaches some threshold the calls to `(future exp)`, which are equivalent to `(fork-thread exp)` will be inlined by simply calling the future's thunk as a procedure, rather than creating a thread.

However, load based inlining has several drawbacks as noted by [MKH90]. It requires programmer involvement. First to decide where load based inlining should be applied and secondly the programmer must decide at what load threshold inlining should occur.

Load based inlining also suffers from the fact that when an inlining decision is made it is irrevocable, and thus there is no way to change the decision and "un-inline" a thread if some processor becomes available. This might result in starvation or deadlock. The cause of this is that once the load passes the threshold every succeeding thread creation is inlined until one of the inlined threads blocks. Many of these threads might be able to run if they weren't inlined. Consequently, one or more processors may have no work to do because there are no executable threads on the system, i.e. threads which are not blocked and which have not been inlined, but there maybe many threads that could be executed if they weren't inaccessible as a result of having been inlined.

Thread absorption does not suffer from this problem. While thread absorption decisions cannot be revoked, they do not cause starvation. This is because no thread is absorbed unless the absorber is about to block and thus there will never be any absorbed thread which could run if it hadn't been absorbed.

As [Moh91] points out

> "Perhaps the most serious problem with load-based inlining is that, for some programs, *irrevocable inlining is not a correct optimization.* Irrevocable inlining can lead to deadlock because it imposes a specific sequential evaluation order on tasks whose data dependencies might require a different evaluation order." (italics theirs)

Again, thread absorption does not suffer from this problem. The only time a thread is absorbed is when the absorbing thread can not proceed. Thus no deadlock can occur because of incorrect order of evaluation.

There are also conditions under which load based inlining creates too many task [PW89] [Wee89]. This is because load based inlining with distributed task queues is unable to achieve oldest-first scheduling. Thread absorption does not suffer from this problem because the absorbing thread is never runnable.

3.3.4.2Lazy Task Creation versus Thread Absorption

Lazy task creation [MKH91] solves many of the problems associated with load based inlining. Lazy task creation works by always inlining the evaluation of every thread, but doing so in such a way that the decision to inline the thread can be revoked if some processor becomes idle. This is done by having each processor maintain a fifo queue of inlined threads evaluating on that processor. If a processor becomes idle it removes the oldest thread from the inlined thread queue of some other processor and begins executing it. Threads are never created until there is a reason to create them, i.e. a processor becomes available to evaluate them, hence the term *lazy* task creation.

Because lazy task creation allows revoking the decision to inline a thread it removes many of the problems associated with load based inlining: no programmer intervention is necessary, processor starvation does not occur, avoidable deadlocks do not occur since no sequential order of thread evaluation is imposed, and too many tasks are not created because tasks are only created when they are actually needed to improve the parallelism of the system.

Lazy task creation works very well with programs, such as divide and conquer ones, that have bushy call trees, but it does not work well for other kinds of programs, such as data parallel programs which are not organized in a bushy tree fashion., e.g. programs using tuple spaces [CG89][CG90][Jag91], or programs which iterate over a linear data structure. Load Based inlining does not work well on these kinds of programs either. *Sting*'s threads, however, are useful for both types of programs.

The primary advantage of lazy task creation is that the average cost of thread creation is extremely low for bushy call tree types of programs. But there are several disadvantages associated with it.

- The scheduling order for the tasks created lazily is determined by the technique, and it is not possible for the programmer to use some scheduling discipline more suitable to the program.

- The average cost of thread creation is very low, but Multi-lisp threads do not support dynamic contexts, or the gathering of meta information on a per thread basis. Properly supporting dynamic contexts as *Sting* does would more than double the cost of lazy task creation.

- Several of the capabilities available in *Sting* can not be implemented if lazy task creation is used. It is not possible to map a thread to particular processor, nor is it possible to schedule threads according to priority, quantum, or some other criterion.

- Lazy task creation, as implemented in Mul-T, relies on having one global heap per processor. The are two problems with this. First it results in less locality when a task is stolen then if the thread had its own heap, and second, since thread stacks and heaps can not be private to the thread, they can not take advantage of machines which have both local and shared memory. *Sting*'s threads do not have these problems.

- In order to garbage collect a system using lazy task creation, as implemented in Mul-T, all threads on the system must be stopped. The garbage collector can be parallel, but no threads can execute while garbage collection is taking place.

- Lazy task creation may not be as effective as *Sting* in load balancing. This is because when a thread is migrated it is the oldest thread on a processor, and migrating it may involve significant data movement. Certainly, it costs more than to migrate an unevaluated *Sting* thread. Further, lazy task creation dictates the load balancing policy, which

may not be the most appropriate for a given program. *Sting* on the other hand allows the programmer to customize the load balancing strategy based on the needs of the program.

It should be pointed out that load based inlining, lazy task creation, and thread absorption all fail in the presence of speculative computation. This is because while there may be a terminating solution all processors may be running non-terminating threads. For both load based inlining and thread absorption one non terminating thread is enough to cause the program to not terminate, whereas with lazy task creation it take N non-terminating threads, where N is the number of processors on the system, and each processor is running a non-terminating thread. It is important to note that thread absorption can be selectively turned off in a *Sting* system and when it is turned off *Sting* handles speculative computation correctly. In particular, wait-for-N discussed in the next section handles speculative concurrency correctly even in the face of non-terminating threads, assuming of course, that the thread policy manager is fair.

3.4 Thread Waiting and Thread Barriers

Sting supports any of the various synchronization constructs: mutexes, condition variables, monitors, etc. These constructs are all independent of the implementation of threads. Given a some form of atomic lock, the simplest being a binary mutex, any synchronization construct desired can be created using simple data structures and the following three procedures which *Sting* provides:

(current-thread),

(thread-block-thread thread blocker),

(thread-resume thread).

But one type of synchronization, *barrier synchronization*, provided by *Sting* is built into the thread system because it relies on the implementation of threads. Two types of barrier synchronization are supported. The simpler of the two corresponds to join points in other thread systems.

```
(thread-wait thread)
```

waits for thread to terminate. The more interesting type of barrier synchronization involves synchronization of multiple threads at one barrier. The construct

```
(wait-for-N count thread1 thread2 ...)
```

blocks the current thread until at least `count` threads in the list have completed. Using the **wait-for-N** construct it is easy to implement and-parallel and or-parallel constructs. *Sting* provides two other barrier synchronization forms which are implemented in terms of **wait-for-N**. These are:

```
(wait-for-one thread1 thread2 ...)

(wait-for-all thread1 thread2 ...)
```

As mentioned above all of these forms work for speculative computation even in the face of non-terminating threads.

3.5 Memory and Object Management in *Sting*

As explained in Chapter 1 *Sting* is designed to support object oriented, functional, logic programming, and fully polymorphic languages, as well as automatic storage allocation and reclamation in the context of these languages. In order to explain how *Sting* manages memory we must first explain how objects are represented and how the virtual address space associated with each virtual machine is organized.

3.5.1 Object Representation in Memory

The fundamental memory structure in *Sting* is called an *extent*, see Figure 3-f . A memory extent is a contiguous set of memory locations that begin with a descriptor. The rest of the locations associated with the extent contain the data for the object that it represents.

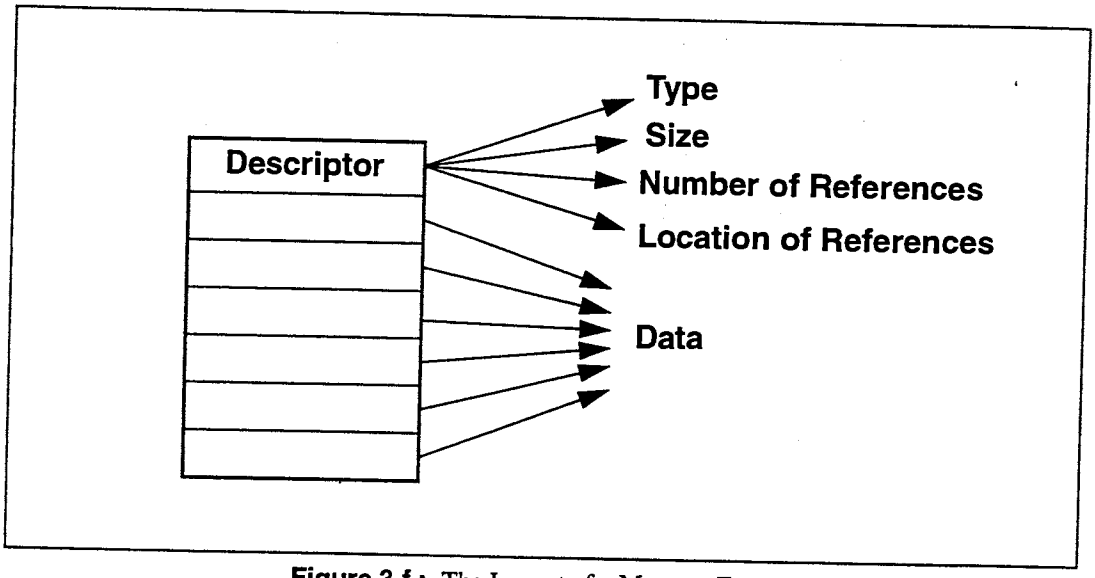

Figure 3-f : The Layout of a Memory Extent

The descriptor contains information about the type of the object, its size, and the number and location of references (pointers) contained in the object. This information is encoded in one or two words of memory depending on the type of the object. The descriptor serves two purposes. It is used for dynamic (runtime) type checking when type information cannot be determined at compile time, e.g. for polymorphic languages, and it supplies the garbage collector with the information needed to copy the extent and trace through its references. *Sting*'s memory extents, and their descriptors, are very similar to those found in the various Scheme, Smalltalk, Common Lisp, ML, and Haskell implementations.

3.5.2 Areas and Extent Allocation

Sting's virtual machines each define a separate virtual address space (see Section 4.3). The virtual address space is divided into *areas*. An area is a contiguous region of memory which is used to store extents. Every memory extent is contained within some area. They do not span areas. *Sting*'s areas are similar to those of [Bis77], but the mechanisms used to manage them are significantly different. Figure 3-g shows vari-

ous areas mapped onto the virtual address space as well as the internal structure of an area.

Areas are used for organizing data which exhibit strong locality, i.e. objects which tend to be used at approximately the same time during the course of a computation. Said another way, *Sting* uses areas to organize objects which exhibit temporal locality so that they also exhibit spacial locality. This organization diminishes the cost of memory accesses in a hierarchical memory system, since objects which are used at a similar time have a greater chance of being brought into the memory hierarchy at the same time, because they are on the same cache line or page. *Sting*'s areas are used to build stacks, private heaps, shared heaps, pools, and possibly other objects. Each area type has different constraints on object allocation and reclamation. We discuss the constraints as they relate to each type of area in the sections that follow.

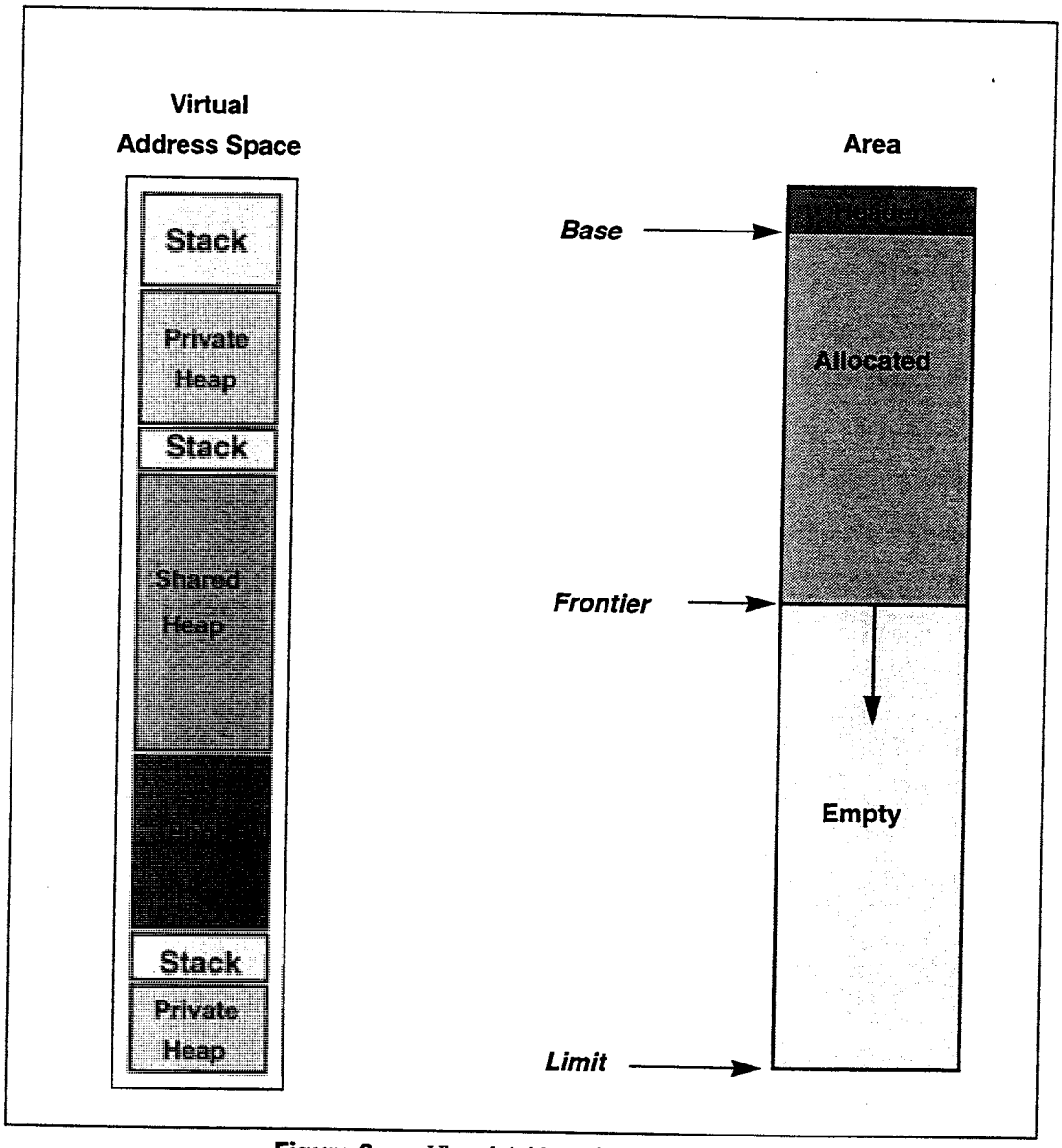

Figure 3-g : Virtual Address Space and Areas

Each area has a header which describes the storage allocation and reclamation strategy for it, and includes fields for the various kinds of information needed to implement the strategy. An area header contains the following fields:

Mutex - Each area has a mutex associated with it. The mutex is used to synchronize access to shared areas. Private areas such as stacks and private

heaps are inaccessible to other threads and need no locking during allocation or reclamation.

Type and Flags - There are several different types of areas including: stack, stack block, private heap, shared heap, pool, and root area. The flags give information such as whether an area is garbage collectable, or whether the area's frontier is cached in a register.

Base - The address of the first location in the area after the area's header.

Frontier -The address of the boundary between the last extent allocated and the empty part of the area. The area frontier is the boundary between that part of an area where objects have been allocated and the part that is empty.

Limit - The address of the last location in the area that can be allocated.

Older and Younger - These two fields are used to store generation information in the case of heaps, or stack chains in the case of stacks.

Root - The root object in the area. This is the root of garbage collection for the area.

Pool - Heaps can contain pools of objects which are allocated and reused. This field refers to such a pool if it exists.

Incoming Reference Set - The set of locations outside this area which contain references into it.

Outgoing Reference Set - The set of locations in this area which contain references to locations outside it.

The uses of these fields are discusses in detail throughout the rest of this chapter. It is important to note, however, that the actual memory cost of each area is only the amount of memory that has been allocated in it. Thus the part of an area that is empty (see Figure 3-g) has no physical memory associated with it[1], either in the caches, including the page frame cache, or on backing store. This is because no reference to a

memory location can exist until an object is created in it. The important consequence of this is that the memory requirements of a program or thread are independent of the number of areas that it uses.

All extents are allocated in the same manner independently of the type of area in which they are allocated. The overhead for allocating an extent in a thread's stack or private heap is two instructions. The overhead for allocating an object in a shared heap is five instructions. Both of these overheads are significantly less then the cost of explicit storage allocation, e.g. an **ALLOC** and **FREE** in Unix. This of course does not count the cost of garbage collection which we have attempted to minimize as discussed below.

Many languages which support garbage collection use a technique called polling to handle interrupts. This technique checks a flag at regular intervals to see if an interrupt has occurred. It only checks the flag when storage is in a consistent state. For example, the compiler for such a language might generate the instructions which check for interrupts at every procedure return point and at the back edges of all loops. This technique has two drawbacks. First, it increases both the amount of code generated and executed and thus increases the time to execute a program. Second, an interrupt has to wait an arbitrary, though bounded, amount of time before it is handled. This may be acceptable in interactive systems, but is certainly not suitable for real time systems. *Sting* is designed for both real time and interactive use, and thus, polling for interrupts is not an acceptable solution.

Storage allocation in *Sting* is faster because threads do not have to worry about being interrupted during storage allocation. Separate areas allow *Sting* interrupt handlers to have there own stack and private heap. This avoids the problem of interrupts during allocation altogether, and thus, simplifies object allocation.

---

1. This is not strictly correct. The amount of memory allocated in an area could exceed the actual amount of data created that area by at most one word less than the page size of the system.

3.5.3 Thread Stacks

Stacks have been used in traditional languages such as Fortran, C, Pascal, or Mesa, to efficiently record the control state of a program by pushing and poping procedure activation frames. However, many modern programming languages such as Scheme, ML, and Haskell support the use of first class continuations. One of the many reasons for this is that first class continuations can be used to implement thread systems [Wan80] [HFM84] [CM90]. Several implementations [Mul-T, ML, Scheme 48] which support first class continuations have decided to implement the control state using a linked list of activation frames which are allocated along with all other objects in the heap, foregoing the use of a stack. This is because it leads to a simple and elegant implementation of first class continuations [App90].

The problem with allocating activation frames in the heap is that it leads to exceedingly poor cache performance on hierarchical memory machines [TA92], because stacks exhibit significantly more locality then heaps. Another problem associated with this technique is that it results in the allocation of more storage than that required by a stack, and thereby incurs additional storage allocation and garbage collection costs.

Further, allocating all objects, including activation records, in the heap also increases the cost of garbage collection because it does not take advantage of the zero cost of collecting stack allocated objects. This cost is zero because the dead objects in the stack are collected (deallocated) as a result of poping the activation frame. And incur no additional overhead from that required to pop the activation frame if it is consed in the heap.

It is also true that many algorithms, particularly recursive ones, can allocate the majority of their objects on the stack, because their lifetime does not exceed the dynamic extent of their creator. Allocating objects on a stack can thus significantly reducing the cost of garbage collection for these algorithms.

For all of these reasons *Sting* threads record control state using stacks. Further, as many objects as possible, in addition to activation frames, are allocated on the stack.

Thread stacks are implemented as areas.

In order to allocation an object on a thread's stack the compiler must ensure that the lifetime of that objects does not exceed the dynamic extent of the procedure which creates the object. This is because when the procedure returns all the objects created by it are reclaimed by simply decrementing the frame pointer. Thus, the overhead for garbage collecting a stack allocated object is at most two instructions, but since many objects may be claimed at once, the cost of collecting a stack object is on average less than one instruction.

Another constraint on stack allocated objects is that any references they contain can only refer to other objects:

- that are in the same dynamic extent,
- that are in a previous dynamic extent, or
- that are in some heap.

Objects which are in the same dynamic extent can refer to each other because they all die at the same time,[1] on exiting the dynamic extent, and therefore all references between them will be deallocated simultaneously. A stack allocated object can also refer to any object which is in a previous dynamic extent. This is because stack references to objects in previous dynamic extents are guaranteed to be reclaimed *before* objects to which they refer. Finally stack allocated objects can refer to objects in heaps because the thread associated with the stack is suspended while the heap is garbage collected and the stacks that contain these references are traced by the garbage collector.

The minimum size of a stack area in *Sting* is one kilobyte. This means that many fine grained threads can be evaluating concurrently. However, the default stack size is 64

1. A compiler can convert references between objects in the same dynamic extent to offsets using an optimization similar to environment collapsing. This optimization decreases storage overhead while increasing data locality.

kilobytes. Users can modify the default stack size as well as control the initial stack size of any thread. Allowing initial stack sizes to be small conserves the address space, not the amount of memory actually used, since as we have pointed out above backing store is only allocated when the memory is actually used. The amount of backing store associated with a stack corresponds to the maximum depth the stack reached during the evaluation of its thread.

Stack chains are another aspect of *Sting* which improves locality and reduces storage requirements. A stack may be composed of several stack blocks chained together. Each stack block is a contiguous set of locations. Stack chains are made efficient by making the continuation at the top of a stack block be a procedure which simply sets the stack pointer to point at the base of the next stack block. The continuation at the base of each new stack block is a procedure which when called simply returns into the previous stack block.

3.5.4 Thread Private Heaps

Thread private heaps are used to allocate objects whose lifetimes might exceed the lifetime of the procedure that created them. We say *might exceed* because it is not always possible for the compiler to determine the lifetime of an object in higher order, or fully polymorphic programming languages such as Scheme or ML. Furthermore, it may not be possible to determine the lifetimes of some objects in languages which allow calls to unknown procedures.

References contained in a private heap can refer to other objects in the same private heap, or objects in shared heaps, but they cannot refer to objects in the stack. References in the stack may refer to objects in the private heap, but references in the shared heap may not. No other thread can access objects which are contained in a thread's stack or private heap. Thus, both thread stacks and private heaps can be implemented in local memory on the processor without any concern for synchronization or memory coherency.

Thread private heaps are actually a series of heaps organized in a generational manner.

Storage allocation is always done in the youngest generation in a manner similar to other generational collectors. As objects age they are moved to older generations. Readers interested in generational collection should see [Ung84] and [App90].

All garbage collection of the private heap is done by the thread itself. In most thread systems that support garbage collection all threads in the system must be suspended during a garbage collection. In contrast, *Sting*'s threads garbage collect their private heaps independently and asynchronously with respect to other threads. Thus, other threads can continue their computation while any particular thread collects its private heap this can lead to better load balancing and higher throughput. A second advantage of this garbage collection strategy is that the cost of garbage collection is charged to the thread that allocates the storage, rather that to all threads in the systems, which is the case with the more traditional collectors mentioned above.

3.5.5 Thread Group Shared Heaps

Each thread group has a shared heap associated with it. The shared heap is allocated when the thread group is created. The shared heap like the private heap is actually a series of heaps organized in a generational manner.

References in shared heaps may only refer to other objects in shared heaps. This is because any object which is referenced from a shared object is also a shared object and, therefore must reside in a shared heap. This constraint on shared heaps is enforced by ensuring that whenever a reference is stored in a shared heap either the object referred to is in a shared heap, and by induction any objects it refers to are in a shared heap, or if the object referred to is in a private heap it is garbage collected into the shared heap. That is, the graph of objects reachable from the referenced object is copied into the shared heap and any references to the object from the stack or private heap refer to the new location of the shared object in the standard way.

Thus the reference discipline observed between the three areas associated with a thread are as follows:

- The stack can contain references to objects in its thread's previous dynamic extents, its private heap, or in a shared heap.
- The private heap can contain references to objects in itself or in some shared heap, but not to objects in the stack. And,
- The shared heap can only contain references to objects in itself or other shared heaps.

Like private heaps shared heaps are organized in a generational manner, but garbage collection of shared heaps is more complicated than that for private heaps because many different threads can be accessing objects contained in the shared heap. In order to garbage collect a shared heap all threads in the thread group associated with the heap are suspended. In addition, all threads in groups inferior to the group associated with the shared heap are suspended. This is because any of these threads can access data in the shared heap. However, other threads in the system, i.e. those not inferior to the group associated with the heap being collected, continue execution independent of the garbage collection.

Each shared heap has a set of incoming references associated with it. These sets are maintained by checking for stores of references that cross area boundaries. After the threads associated with the shared heap have been suspended the garbage collector uses the set of incoming references as the roots for the garbage collection. Any objects reachable from the incoming reference set are copied to the new heap. When the garbage collection is complete the threads associated with the shared heap are resumed.

There are several other attributes of memory areas that are beyond the scope of this dissertation. For example, an area may act as a pool for the explicit allocation and deallocation of objects in the traditional manner of C, C++, Pascal, or Modula 2. As another example, area pools might be extremely useful in exploiting the locality inherent in large data bases. Finally, in *Sting* there is also an area, called the *root area* which can only be garbage collected when the entire virtual address space is garbage collected. The importance of the root area is that references into it can be assumed to

be valid when garbage collecting either thread private heaps or shared heaps.

Chapter 4

Virtual Machines and Virtual Processors

A virtual machine is an abstraction that is mapped onto all or part of a physical machine. A virtual machine is composed of:

- a virtual address space;
- one or more virtual processors;
- a virtual topology;
- a root environment; and
- a root thread.

Virtual machines create and destroy virtual processors. Each virtual machine has at least one virtual processor, called the *root virtual processor*, and one thread called the *root thread*, but it may have any number of virtual processors and threads. The virtual machine defines both the mapping of virtual processors to physical processors, and the virtual topology, i.e. the inter-connection graph of the virtual processors, which may or may not correspond to the physical topology.

The virtual machine manages its virtual address space, which is shared by all virtual processors and threads in the virtual machine. The virtual machine's address space contains the root environment, which is the root of the graph of live objects contained

in the address space. Throughout this dissertation the term address space is used synonymously with virtual address space. The term physical address space will always be used when referring to the hardware address space in a physical machine.

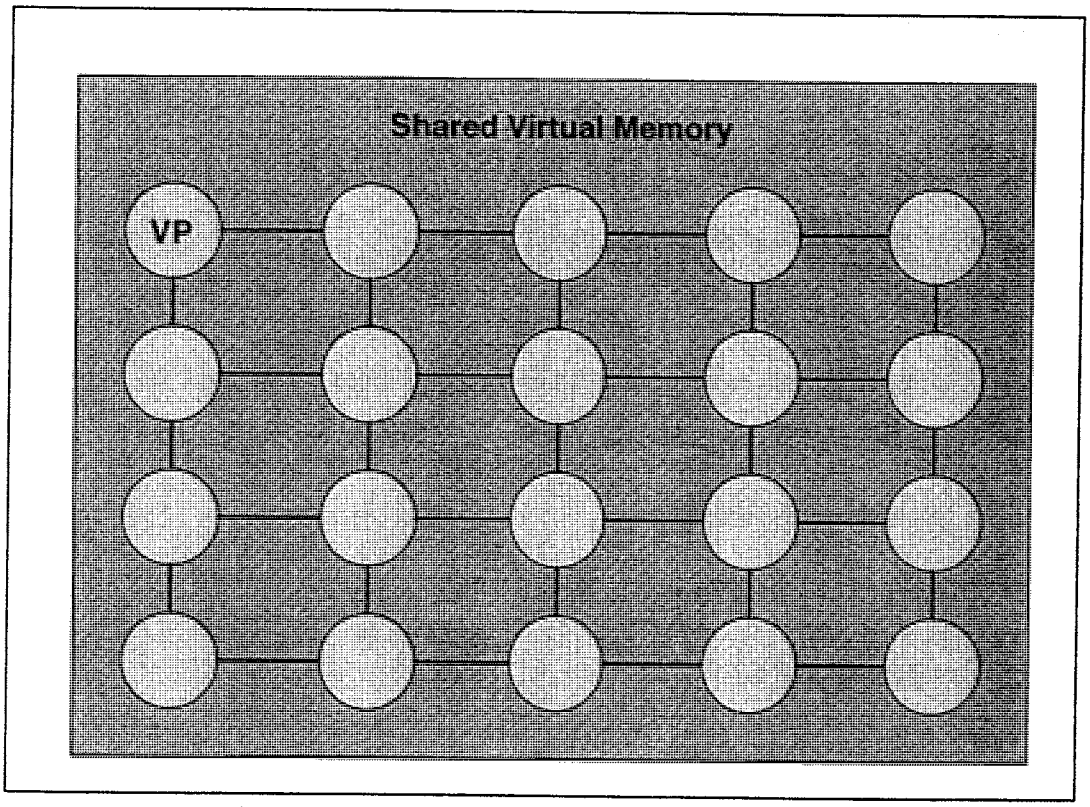

Figure 4-a : A Virtual Machine with a 2D Mesh Topology

As with other operating system kernels, the physical memory is hidden from all virtual machines. Virtual machines only have access to their virtual memory. This allows physical memory to be configured in many different ways. For example, the physical machine could have a physically shared or physically disjoint memory without apparent difference in the virtual memory. In any case, the virtual memory abstraction is always that of a shared virtual memory [LH86][Li88], i.e. every address in a virtual address space is accessible from any virtual processor in the machine.

Finally, the virtual machine is persistent. It may be suspended and then resumed at

some later time. When resumed, the virtual machine will be in exactly the same state as when suspended.

4.1 Comparison with Other Systems

A virtual machine loosely corresponds to a Unix kernel process, a Mach task, or a Topaz address space [BGHL87]. Each of these entities define an address space and a thread, but in these other systems the thread is a kernel thread, and therefore, heavy-weight. Furthermore, these systems have no concept of customizable virtual processor, and no virtual topologies.

Schedular activations [ABLL91] address one of the problems that *Sting*'s virtual machines are intended to solve, i.e. user space threads blocking in the kernel and informing the kernel when no thread is available to run in a kernel process or task, but again they do not have customizable virtual processors or virtual topologies.

Psyche [MSLM91] is the operating system whose goals are most closely aligned with ours. Psyche endeavors to provide an efficient foundation for any thread paradigm or any parallel language. Marsh, et al explain the reason for this:

> "Users want, and different runtime environments define, threads of various kinds, many of which may be incompatible with the kernels notion of process. Some environments expect one thread to run at a time, as in the co-routine like scheduling of Distributed Processes [Han78] and Lynx [Sco91]. Some want to build cactus stacks, with dynamic allocation of activation records as in Mesa [LR80]. Some want to use customized scheduling policies, such as priority-based scheduling.... Some want to have processes interact using communication or synchronization mechanisms that are difficult to implement with kernel-provided operations. Some simply want to create a very large number of threads, more than the kernel can support."

These are the same reasons that drove the *Sting* design. However, Psyche and *Sting* are very different approaches to solving this problem.

"Psyche kernel processes are used to implement the virtual processors[1] that execute user level threads." [MSLM91] In many respects Psyche's virtual processors are similar to kernel threads in other multi-processor operating systems such as Mach and Topaz (Taos), but they are different from these in that Psyche provides user customizability by defining data structures which the kernel and user code can share. These data structures contain software interrupt handlers that are user provided procedures that are called when situations occur in the kernel which might effect the operation of the virtual processor. For example, when a user thread blocks in the kernel, the kernel calls the software interrupt handler for this condition. The handler may decide to block the virtual processor, run another thread, or do whatever else is appropriate for the particular thread system being implemented. Software interrupts allow the kernel to notify the virtual processor whenever an event which might be of interest to it occurs in the kernel.

Given Psyche's virtual processors it is possible to implement many different thread semantics and many different scheduling policies, and thus, Psyche is fully customizable. However, *Psyche does not separate control and policy mechanisms in the virtual processor* and thus each thread package must implement both of these mechanisms. *Sting* provides efficient control mechanisms for threads, while allowing the virtual machine to be customized by implementing only a policy manager with a small well defined interface. In Psyche, user threads are distinct from kernel threads. In *Sting*, lightweight threads are integrated into the kernel design so that no kernel threads are necessary.

We believe the *Sting* approach provides a significant increase in programmer efficiency, while at the same time providing an increase in program efficiency compared to that of more traditional operating systems.

1. While the *Sting* and Psyche designers arrived at this term independently, the term denotes a similar and very useful concept.

4.2 Operations on Virtual Machines

Sting provides the traditional operations on virtual machines that are normally associated with kernel processes. Virtual Machines can be created, destroyed, blocked, suspended, and resumed. In addition, any thread can know on which virtual machine it is running.

When a virtual machine is created, its root virtual processor, its root thread, and the root thread group are also created. The root thread evaluates a thunk which is an argument to the **create-virtual machine** operation.

Figure 4-b shows the operations which can be performed on virtual machines. The semantics of these operations are sufficiently obvious that we will not discuss them further here.

(create-virtual-machine *thunk priority quantum***)**	-> *vm*
(block-virtual-machine *vm blocker***)**	-> *no-value*
(suspend-virtual-machine *vm wakeup-time***)**	-> *no-value*
(resume-virtual-machine *vm***)**	-> *no-value*
(destroy-virtual-machine *vm status***)**	-> *no-value*
(current-virtual-machine)	-> *vm*

Figure 4-b : Operations on Virtual Machines

4.3 Virtual Address Space

The virtual address space associated with the virtual machine is similar to traditional ones except that is shared or distributed across all the processors in the system. There are an increasing number of parallel processors which implement shared virtual mem-

ory, e.g. Dash [Hen91], Kendall Square One [Bel92][Rot92], and the work of Kai Li on the Intel Hypercube [Li89].

The shared virtual memory may be implemented on top of a physically shared memory machine, such as Sequent Symetry or the SGI PowerSeries, or it may be implemented on a disjoint memory machine such as the Intel Hypercube or the Ncube, or on a hybrid machine, where part of the memory is shared and part of it is local, such as the IBM RP3 or the RS6000 Multi-processor.

One virtual address space may be fully or partially mapped into another address space. The unit of address mapping is the area (see Section 3.5.2). Thus an area in one virtual machine can be mapped into the address space of another virtual machine. In the limit, two virtual machines can shared the same address space. Shared virtual memory is discussed further in Section 5.5.1.

4.4 Virtual Processors

A Virtual Processor (VP) is an abstraction of a hardware processor. As such, it is responsible for the creation, destruction, scheduling, and migration of lightweight threads. It also handles interrupts (hardware and software) and virtual processor controller up calls, i.e. software interrupts generated by the abstract physical machine, for example, when a thread blocks in the abstract physical machine.

Each VP is associated with both a virtual machine and an abstract physical processor (see Section 5.7). A physical processor may run VPs associated with many different virtual machines. More than one VP from the same virtual machine can also run on the same physical processor.

Sting VP's are first class objects. This means they can be passed to and returned from procedure calls, and can be stored in data structures. Being first class, VP's provides *Sting* with several capabilities that other operating systems lack:

- the user can explicitly map a thread to a particular virtual processor;

- the user can build abstract topologies using VP self-relative addressing; and,
- the policies of any VP can be easily customized.

Control and policy are separated in the virtual processor, just as they are in the abstract physical processor (see Section 5.7). Each virtual processor is composed of two software components: the *thread controller* and the *thread policy manager*. Figure 4-c shows the relationship between the virtual machine, the virtual processor, and user code. The thread policy manager is completely contained within the virtual processor. User code and threads interact with the thread controller and the thread controller calls the thread policy manager to make policy decisions for it.

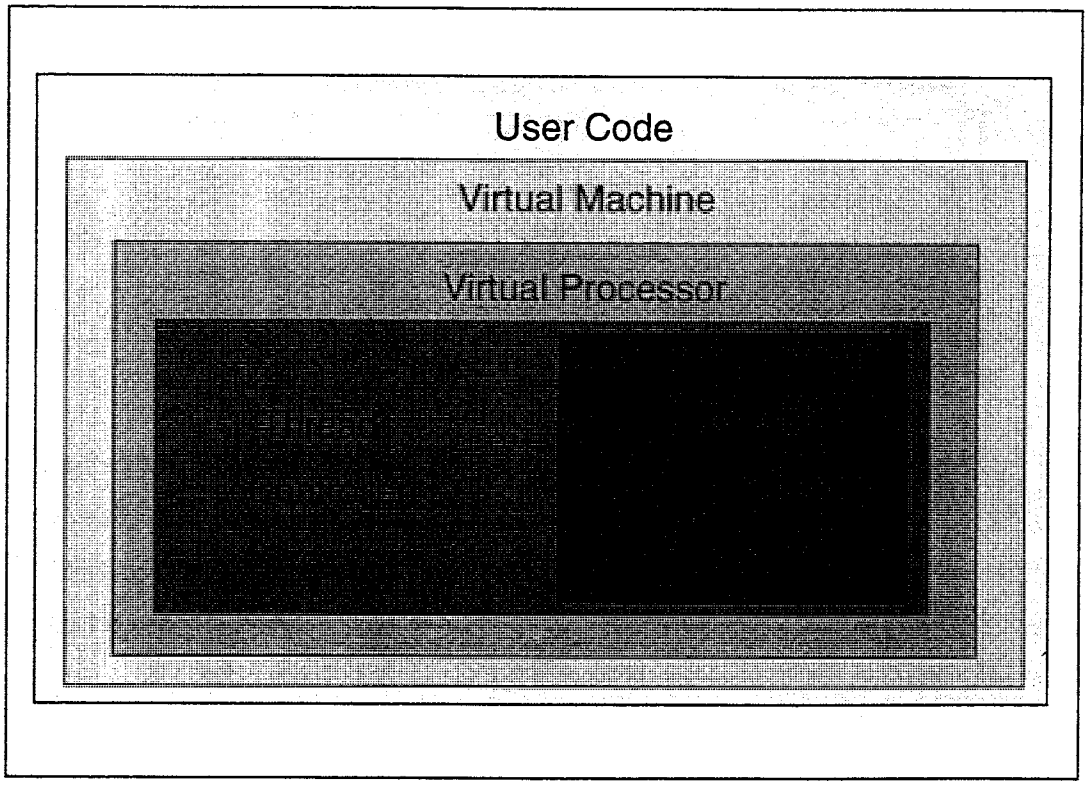

Figure 4-c : Separation of Control and Policy in the Virtual Processor

4.4.1 Virtual Machine and Virtual Processor States

Both virtual machines and virtual processors can change their state in a manner similar to that of evaluating threads. Figure 4-d show the various states that virtual machines and virtual processors can be in. It also show the valid transitions from one state to another.

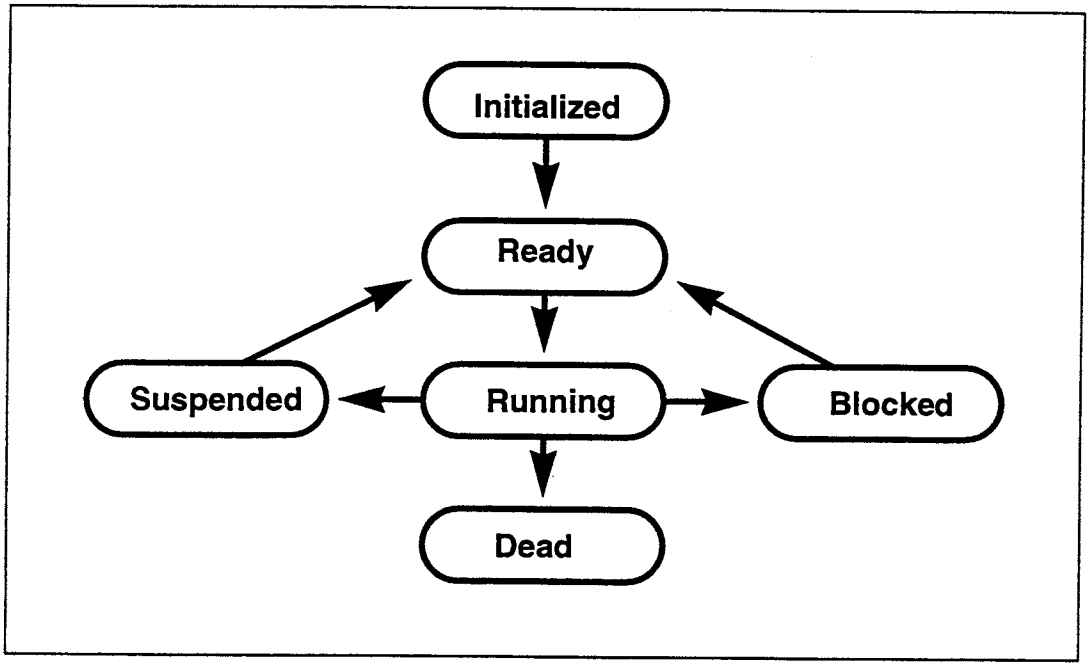

Figure 4-d : Virtual Machine and Virtual Processor State Transitions

When a virtual machine enters a state all the virtual processors enter the same state. Thus, if a virtual machine is suspended all threads and processors in that virtual machine are suspended. Likewise when a virtual processor is suspended or blocked all threads ready or running on that processor are suspended or blocked. Virtual machine and virtual processor states are described in more detail in Section 5.6.2.

4.4.2 Operations on Virtual Processors

The virtual machine is responsible for the creation, destruction, and control of virtual processors. Virtual processors can be created, destroyed, blocked, suspended, and resumed. In addition, any thread can know on which virtual processor it is running.

When a virtual processor is blocked or suspended all its threads are blocked or suspended. When a virtual processor is resumed its threads that are ready to run are resumed. Finally, when a virtual processor is destroyed its threads are either migrated to other virtual processors or they are terminated in an orderly manner by signalling their termination exception handlers.

(create-virtual-processor *pp***)**	-> *vp*
(block-virtual-processor *vp blocker***)**	-> *no-value*
(suspend-virtual-processor *vp* **.** *wakeup-time***)**	-> *no-value*
(resume-virtual-processor *vp***)**	-> *no-value*
(destroy-virtual-processor-and-threads *vp***)**	-> *no-value*
(exit-virtual-processor-and-migrate-threads *vp***)**	-> *no-value*
(current-virtual-processor)	-> *vp*

Figure 4-e : Operations on Virtual Processors

Figure 4-e shows the operations which can be performed on virtual processors. The reader will note the similarity to operations on virtual machines. This is because a virtual machine can be regarded as a collection of virtual processors. The semantics of these operations are sufficiently obvious that they will not be discussed further here.

4.4.3 Virtual Processor Meta Information

Each virtual processor can record performance and debugging information. This information includes:

- the number of threads created,
- the number of threads created in the delayed state and not initially scheduled,
- the number of threads scheduled,
- the number of threads absorbed by other threads,
- the number of thread blocks, suspends, and resumes,
- the number of threads terminated,
- the number of threads determined,
- the number of threads migrated from and to a VP,
- the number of **thread-wait**'s and resumptions that have occurred on a VP,
- the number of blocks, resumptions, and terminations at thread barriers,
- the number of garbage collections that have occurred on the VP,
- the number of TCB's created, allocated, and that were dead and reused,
- the number of stack overflows and underflows that occurred as well as the number of stack blocks created, allocated, and released,
- the number of mutexes created, acquired, and released, and
- the idle time, user time, system time, page faults, etc. that are usually recorded in a kernel thread.

Statistics gathering can be turned on and off under user control.

4.5 Thread Controller

The *thread controller* handles the virtual processor's interaction with other system components such as physical processors and threads. While several types of interaction between the physical processor and the virtual processor occur, two are particularly important for threads and virtual processors to execute efficiently. The first occurs when a thread makes a system call which causes it to block in the kernel of the physical processor. In this case, the physical processor notifies the virtual processor that the current thread has blocked and that another thread should be scheduled.

The second type of interaction occurs when the virtual processor has no work to do, i.e. no thread to execute. In this case, rather than spin waiting for more work to arrive, the virtual processor notifies the physical processor that it has no work, thus allowing the physical processor to run another virtual processor.

The lack of support for these two types of interactions in traditional operating systems introduces significant inefficiencies in other lightweight thread systems that have been implemented. There have been two main approaches to solving this problem, that of Anderson, et al [ABLL91] and that of Marsh, et al [MSLM91]. *Sting*'s approach is closer to that of [MSLM91], but with significant differences, which are discussed below.

4.6 Context Switching with Continuations

Sting threads can suspend execution for any number of reasons, but in general these reasons fall into three categories:

- the thread has blocked waiting for some event to occur;
- the thread has been interrupted by either a software or hardware generated exception; or

- the thread has explicitly yielded its virtual processor to another thread.

In each of these cases, when a thread suspends itself, it saves its current continuation in the thread control block associated with it. A continuation can be thought of as the entire context of a computation at a particular program point. This context includes the stack and private heap as well as any references to objects in shared heaps. However, saving the current continuation involves saving only the currently active registers in the virtual machine. This is because the thread controller knows that *the continuation will only be invoked once*, and thus no copying of the stack or private heap is necessary.

When a continuation is invoked it continues the computation from the point at which the continuation was saved. A suspended thread is resumed simply by invoking its saved continuation. The idea of using continuations for context switching stems from the work of Wand [Wan80].

Whenever a running thread needs to make a state transition it calls the thread-state-transition procedure. A simplified version of this procedure is shown in Figure 4-f . A running thread can make transitions to the following states: ready, blocked, suspended, and terminating. Thus all context switching is done using this procedure.

The implementation of the *Sting* thread controller highlights a number of interesting issues. The state transition procedure is shown in Figure 4-f . Note that operations on TCBs found in this procedure are not available to user applications. The procedure takes one argument - the desired next state for the current thread (i.e. the thread which has entered the thread controller).

Since the thread controller is written in *Sting*, all synchronous calls to thread controller procedures are treated as ordinary procedure calls; thus, live registers used by the procedure running in the current thread are saved automatically in the thread's TCB. The procedure first attempts to acquire a new thread to execute from the ready queue of this VP. The thread policy manager procedure **tpm-get-next-thread** is used for this purpose. Note that **tpm-get-next-thread** enqueues the current thread's TCB if the TCB

is in a ready state; this permits the thread to be rerun at some future point.

```
(define (state-transition next-state)
  (let* ((vp (current-vp))
         (next (let ((next (tpm-get-next-thread vp)))
                 (if (not next)
                     (vp.root-tcb vp)
                     next))))
    (cond ((eq? next (current-tcb)))
          ((tcb? next)
           (set-tcb.state next tcb-state/running)
           (if (eq? next-state tcb-state/dead)
               (return-tcb-to-vp-pool vp))
           (set-vp.current-tcb vp next)
           (set-thread.vp (tcb.thread next) vp)
           (save-current-tcb-registers)
           (restore-tcb-and-registers))
          ((thread? next)
           (cond ((thread-stolen? next)
                  (state-transition next-state))
                 (else
                  (setup-new-thread next vp)))
                  (save-current-tcb-registers)
                  (start-new-thread next))))))
```

Figure 4-f : Thread State Transition Procedure

If the queue is empty, the root thread of the current VP is invoked (via the call to **(vp.root-tcb vp)**). This procedure may (a) perform housekeeping operations, (b) simply re-invoke the state transition procedure, or (c) request that the abstract physical processor switch to a new VP. If the queue is not empty, a new thread (or TCB) is returned. A new thread is returned only if the thread is *not* evaluating. The expression

(fork-thread *expression vp*)

creates a thread (call it *T*) and schedules it to run on *vp* by enqueing *T* on *vp*'s ready queue. It becomes eligible for evaluation when the thread controller removes it from the queue.

Once a thread begins evaluation, it is never directly stored in any queue maintained by a VP. Its TCB is stored instead.[1] Thus, a TCB returned by **pm-get-next-thread** is always associated with an *evaluating* thread.

The outermost conditional in **state-transition** performs the actual context switch. If the current TCB happens to be the TCB returned by **pm-get-next-thread**, it is simply rerun - no extra register saves or restores need be performed (other than those needed to execute the call/return sequence to/from this procedure).

If the object returned by **pm-get-next-thread** is another TCB (distinct from the current TCB), all live TCB registers of the current TCB are saved, and the continuation, encapsulated in the **next** TCB is restored. If the current TCB is dead (because it has terminated), it is returned to the current VP's pool of TCBs. Because **restore-tcb-and-registers** is a primop, the compiler treats **save-current-tcb-registers** as if it were in tail call position; thus, when the saved thread resumes execution (i.e. assuming its TCB has not terminated), it simply returns from **state-transition**.

If the object returned is a thread, a TCB is allocated for it via the call **setup-new-thread** *provided* the thread has not been absorbed (see Section 3.3.3). The current thread state is saved (via the call **save-current-tcb-registers**, and the new thread begins execution via the call to the primop **start-new-tcb**. This primop sets up a new continuation encapsulating the evaluation of **start-new-thread** and commences its evaluation using **tcb** as its dynamic context.

Here again, the compiler treats the register save operation as if it were in a tail call position; thus, when the saved thread resumes execution it simply falls through the conditional and returns to the caller. The code for **start-new-thread** is shown in Figure 4-g .[2] This procedure takes a thread as its argument and evaluates the thunk associated with the thread in such a way so as to ensure proper termination and cleanup of the thread.

---

1. The thread associated with a non-terminated TCB is accessible via the **thread** slot found in that TCB.
2. Figure 4-g is similar to Figure 2-g , but it provides more details of thread termination.

```
1: (define (start-new-thread thread)
2:   (let ((z (no-value)))
3:     (catch exit
4:       (dynamic-wind
5:         (lambda ())
6:         (lambda ()
7:           (set-exit-handler! thread exit)
8:           (set! z ((thread.thunk thread))))
           ;; Thread termination
9:         (lambda ()
10:          (set-thread-value! thread z)
11:          (thread-gc thread)
12:          (wake-up-waiters thread)
13:          (re-initialize-tcb (current-tcb))
14:          (tcb-state->dead dead))))))
```

Figure 4-g : Thread Startup

start-new-thread first creates a temporary to hold the value of the thread (line 2). Next it creates a continuation using the **catch** expression (line 3). This continuation is store in the exit handler slot of the thread's TCB. In line 4, a dynamic wind expression ensures that if a throw out of the evaluation of the thread's thunk (line 8) occurs the thread's stack will be unwound properly, thereby permitting resources such as locks held by the thread to be properly released. In addition, it ensures that the cleanup code for the thread (lines 9 -13) will be executed.

The termination code stores the value of the thread's thunk as part of the thread state, garbage collects the thread stack and private heap (line 10), wakes up all threads waiting for this thread's value (line 11), reinitializes the TCB state (line 12), and finally makes a tail recursive call to the thread controller transition procedure to choose a new thread to run. Because the evaluation of the thread's thunk is wrapped in a dynamic wind form, it is guaranteed that the termination code will be executed even if a thread terminates abnormally.

Garbage collection must take place before the thread's waiters are awakened because

objects that outlive the thread (including the object(s) returned by the thread's thunk) found on its private heap must be migrated to a shared heap. Failure to do so would allow other threads to obtain references to the newly terminated thread's storage.

Preemption is disabled within the thread controller. Disabling preemption guarantees that the thread controller will not be interrupted as a result of timer expiration. Other interrupts need not be disabled since data structures maintained by the thread controller are not manipulated by VP interrupt handlers. Each VP has separate areas (implemented as small stacks and heaps) used by interrupts and the garbage collector for servicing any storage requirements they may have.

It is possible for the compiler to minimize the number of registers saved when a thread current continuation is saved. In the simplest case, when an explicit, i.e. synchronous, call to the state transition procedure is made, the compiler knows exactly which registers are live and saves only those registers. Thus, only the minimal number of registers is saved when saving the current continuation explicitly. When an implicit, i.e. asynchronous, call to the state transition procedure is made, because of a preemption timeout, things are more complicated. There are two different optimizations that can be performed. The first and simpler one is that the compiler can record in the TCB data structure whether a thread has ever used a floating point register [MP89]. The state transition procedure checks this flag to see whether the floating point registers need to be saved, and if not it avoids the cost of saving them. The second approach has the compiler associate with every procedure closure a word which indicates the number of live integer and floating point registers used by that procedure. The state transition procedure then uses this word to save only the registers which are used by the current procedure when its continuation is saved.

Sting keeps the current continuation in the **saved-vp-state-block** of the TCB. This decision was made so as to avoid stack overflow during a state transition. However, the current continuation could be saved on the top of the stack. This later approach has two advantages: it reduces the size of the TCB and it improves locality because the top of the stack is more likely to be in the primary cache than the base of the stack. We

have come to believe that this latter approach is the better one.

4.6.1 Kernel Calls and Continuations

In Sting there are no kernel processes, only lightweight user space threads. When a thread makes a call into the abstract physical machine, i.e. one that corresponds to a kernel call in a traditional OS, no kernel process executes the call; rather, the kernel call executes using the dynamic context (stack, private heap, etc.) associated with the thread. Kernel processes and kernel stacks are unnecessary.

If the execution of a thread is blocked in the kernel of the physical machine while waiting for some event to occur, the thread simply saves its current continuation and then calls its virtual processor to execute some other thread. When the event on which the thread is waiting occurs, the thread is resumed by simply invoking its saved continuation. The interface to the thread controller is made available to the abstract physical machine kernel so that the thread can notify the virtual processor that it has blocked.

Mach 3.0 also uses "continuations" to avoid creating an excessive number of kernel stacks [DBRD91]. Mach's "continuations" however are really thunks that rely on the fact that the thunk has been "hand crafted" to captured enough of the state to continue the computation. *Sting*, as mentioned above, has no kernel threads so no kernel stacks are necessary. Further, since *Sting*'s continuations capture the entire state of the evaluating thread no hand crafting is necessary.

4.7 Thread Policy Manager

Each virtual processor also contains a *thread policy manager*. The thread policy manager, which is analogous to the virtual processor policy manager (see Section 5.8.2 on page 132), makes all policy decisions relating to the scheduling and migration of threads on virtual processors. The thread controller is a client of the thread policy manager and it is inaccessible to user code. The thread controller calls the thread policy manager whenever it needs to make a decision concerning:

- the initial mapping of a thread to a virtual processor;
- which thread a virtual processor should run next when the current thread releases the virtual processor for some reason; or
- when and which threads to migrate from or to a virtual processor.

The thread controller is conceptually the same on each virtual processor. This is not the case with policy managers. The policy manager of each virtual processor may be different. This ability is particularly important for real time applications where each processor may be controlling a different subsystem with different scheduling requirements.

The thread policy manager presents a well-defined interface to the thread controller. The data structures that the thread policy managers use to make their decisions are completely private to them. They may be local to a particular thread policy manager or shared among the various instances of the thread policy manager, or some combination thereof, but they are never available to any other part of the system. The thread policy manager can thus be customized to provide different behaviors for different virtual machines. This allows the user to customize policy decisions depending on the type of program being run. For example, a computationally intensive program such as a fluid dynamics simulator, might use a non-preemptible lifo scheduling policy, because each thread should run as long as possible and because the lifo scheduling order is optimal for the particular structure of the algorithm being used, while a window manager or user shell might use a priority based fifo policy for the obvious reasons.

It is also worth noting that each thread has an associated priority and quantum. These fields are only for the use of the thread policy manager. They allow the implementation of the full gamut of scheduling strategies from quantum based real time scheduling to priority based interactive scheduling.

4.7.1 Thread Policy Manager Interface

The thread policy manager communicates with the thread controller through a small, simple, and easily implemented interface. Whenever the thread controller needs to make a policy decision it calls on the thread policy manager to make that decision.A thread policy manager can be implemented by any module which conforms to the policy manager interface.

Figure 4-h shows the interface to the thread policy manager. The interface can be divided into five components: policy manager initialization, initial placement policy, scheduling policy, guards used by the thread controller to ensure that priority and quantum data are of the correct type, and migration policy.

VP Initialization	
(tpm-initialize-vp *vp***)**	-> *no-value*
Initial Placement Policy	
(tpm-allocate-vp *vp thread***)**	-> *vp*
Thread Scheduling Policy	
(tpm-dequeue-ready-thread *vp***)**	-> *thread* / *#F*
(tpm-enqueue-ready-thread *vp thread***)**	-> *no-value*
(tpm-wakeup-suspended-threads *vp***)**	-> *no-value*
Scheduling Data Integrity	
(tpm-ensure-priority *priority***)**	-> *no-value*
(tpm-ensure-quantum *quantum***)**	-> *no-value*
Migration Policy	
(tpm-vp-idle *vp***)**	-> *no-value*

Figure 4-h : Thread Policy Manager Interface

Below we briefly describe the functionality of each of the interface procedures.

tpm-initialize-vp - This procedure is called when a VP is created. It is responsible for initializing any data structures associated with the thread policy manager on the VP that is its argument.

tpm-allocate-vp - This procedure is called whenever a thread is to be scheduled on a VP. While the user may request that a thread be scheduled on a

particular VP, the thread policy manager has the final say regarding the VP on which a thread is scheduled.

tpm-dequeue-ready-thread - This procedure returns the next thread which is ready to run or *false* if there is no ready thread. This procedure is called from the state transition procedure when a thread has yielded the processor for whatever reason.

tpm-enqueue-ready-thread - This procedure is called when a thread becomes ready to run. It is responsible for enqueuing the thread according to the priority and quantum associated with the thread.

tpm-wakeup-suspended-threads - This procedure is responsible for moving any suspended threads into the appropriate ready queue if the real clock time has passed the requested wakeup time.

tpm-ensure-priority - This procedure ensures that it argument conforms to the thread policy manager's notion of a valid priority.

tpm-ensure-quantum - This procedure ensures that it argument conforms to the thread policy manager's notion of a valid quantum.

tpm-vp-idle - This procedure is called if there is no thread runnable. It can do several things:

- It can call the abstract physical processor to inform it that the VP is idle.
- It can migrate threads from either a global queue or a local queue associated with some VP.
- It can decide to do housekeeping chores for the thread policy manager.

We should point out that the classification of the thread policy manager's interface procedures is not complete. For example, both **tpm-enqueue-ready-thread** and **tpm-dequeue-ready-thread** could be use to handle load balancing and thread migration on

the virtual processor. **tpm-enqueue-ready-thread** could also be used to handle initial mapping of a new thread to a virtual processor. Thus **tpm-allocate-vp** and **tpm-vp-idle** are redundant procedures, but that are useful because they simplify the implementation of both the thread controller and the thread policy manager.

4.7.2 Policy Dimensions

As mentioned above, thread policy managers make three types of policy decisions: (1) which processor to schedule a thread on; (2) which thread to run next on a given processor; and (3) when to migrate a thread to another processor and onto which processor to migrate it. There are many different strategies for making these decisions, depending on both the application environment and the implementation of a particular application. Below we discuss various strategies for these three policy dimensions.

4.7.2.1 Initial Placement Decisions

There are many possible ways to decide on which virtual processor to run a thread. The two principle criteria that can be used to make this decision are load balancing and "nearness" in the communications topology, to other threads with which data is shared. The reasons for load balancing are obvious, although load balancing strategies are not necessarily so. The reason the nearness matters is that it can significantly reduce the cost of communication overhead. A third, though less important reason for mapping a thread to a particular processor is that the processor has a hardware device connected to it which the thread needs to access efficiently.

Below we describe several different load balancing strategies that we have implemented and several which we have not implemented but which may be appropriate under certain circumstances.

Parent's VP - This strategy involves mapping a new thread to the same VP that its parent was running on when it was created. This strategy does not attempt to balance the load on the machine initially, rather it relies on some migration strategy, discussed below, to balance the load on the processor. It

does take advantage of the locality normally associated with a parent and its children threads.

Local with Threshold Overflow to Global - This strategy is similar to the Parent's VP strategy. The difference is that when the number of ready threads on a VP reaches a threshold some number, typically half, of the threads are moved to a global queue. When the local ready queue is empty the VP requests threads out of the global queue. This strategy exploits locality just as the Parent's VP strategy does. It has the further advantage that the load on an individual processor never goes above a particular threshold. The disadvantage of this strategy is possible contention on the global queue.

Topology Mapping - This strategy relies on the programmer to specify the thread/VP mapping. It has the significant advantage that the programmer can take advantage of the data sharing and data distribution attributes of a particular program to improve its efficiency. It has the disadvantage that it requires more work on the part of the programmer. Any of the other placement strategies can be implemented in such a way so that they honor programmer specified mappings, but make appropriate decisions when the programmer doesn't specify the mapping. We discuss topology mapping in more detail below.

Random Placement - This strategy involves randomly mapping a new thread to a VP. It relies on the randomness to create an even distribution of threads on a processor. This strategy has the disadvantage that it does not take any advantage of the memory or communications architecture of the machine.

Round Robin - This strategy involves relying on a global counter to distribute the threads evenly across all VPs. It has the advantage of being simple and guaranteeing a good distribution. It has the disadvantage that the global counter must have a lock and thus may become a bottleneck, and like ran-

dom placement, it does not take any advantage of the memory or communications architecture of the machine.

Idle VP - While there are many possible load based strategies the idle processor strategy is perhaps the simplest. Whenever a VP is idle it places itself on the idle VP queue and whenever a new thread is created it is mapped onto the VP at the head of the queue. If no VP is idle some other strategy is used map the thread to a VP. The advantage of this strategy is that idle processors quickly receive new work that is created. The disadvantage is that the number of threads per VP may be very unbalanced, and it does not take any advantage of the memory or communications architecture of the machine.

Load Based - This strategy relies on each VP maintaining some notion of its load and inserting itself in global queue ordered from the least loaded to the greatest loaded VP. When a thread is mapped to a VP it is always mapped to the least loaded VP. This strategy has obvious advantages, but its disadvantages are that the global queue may be a significant bottleneck is the threads are very lightweight and it does not take any advantage of the memory or communications architecture of the machine.

There are many more possible initial placement strategies. Some can be devised by combining elements of those mentioned above. The performance of a program or algorithm can depend crucially on the initial placement strategy used. It is also clear that for programs where computation costs completely dominate communications costs, i.e. embarrassingly parallel programs, initial placement decisions are less important, because they do not need to take advantage of the communications topology; however, initial placement decisions will still effect load balancing. For fine grained parallel programs, especially if run on a machine with a complex topology or distributed hierarchical memory, initial placement decisions are fundamental to good performance.

4.7.2.2Scheduling

While there are many different strategies for initially mapping a thread to a virtual processor, there are many more possible strategies for scheduling those threads once they are mapped. Rather than discuss these strategies in detail we will discuss various dimensions along which decisions about the scheduler design must be made.

Local or Global Queues - The first decision the scheduler designer must make is whether the queue(s) will be global, i.e. associated with the virtual machine, or local, i.e. associated with each virtual processor, or some combination of the two. Global queues have the advantage of being more fair, but the disadvantage of being a potential source of contention, and therefore a bottleneck.

Number and Class of Queues - Ready threads can divided into three classes for the purposes of scheduling:

- Those that have been scheduled but never been run and thus have no execution context.
- Those that have been run before and were last in the *running* state before going to the *ready* state.
- And those that have been run before and were last in the *blocked* state before going to the *ready* state.

Ready threads of all classes can be scheduled in one queue, but it may be advantageous to separate threads in different classes into different queues. For example, consider an implementation in which the scheduler maintains one queue for threads that have been scheduled but never run and another queue for threads which have been run. The policy manager can quickly migrate threads that have never been run to other VPs without having to migrate the thread's execution context (because it doesn't have one). The scheduler designer may also wish to discriminate between threads that were *running* and threads that were *blocked*, since the former, having been run more recently,

are likely to have more cache locality than the latter.

Ordering Within a Queue - There are various strategies for ordering threads within a particular queue. The two simplest are LIFO and FIFO, but since threads can have both a priority and a quantum associated with them much more complicated orderings can be created.

Locking Discipline - Another decision the scheduler designer needs to make is locking discipline required for the queues. If the queues are global and have multiple readers and writers, they must be locked on every access. If the queues are local, and have only one reader, i.e. the local VP, and many writers, they only need to be locked for writing. Finally, it is possible that a queue could be completely private to a processor and not need a lock. This is possible if the queue contains only threads that were previously running on that processor and those threads can never be migrated to another processor.

Quantum Discipline - The final decision the scheduler designer must make concerns the amount of time each thread will run before yielding the processor, i.e. its quantum. There are several possibilities. The quantum for each thread may be determined statically by the programmer. This is generally done when the schedule for the threads is predetermined prior to the execution of a program. Other possibilities are that the quantum is the same for all threads; it is the same for all threads at a given priority, but different for threads at different priorities; or the quantum can vary dynamically under programmer control or under policy manager control.

These various scheduler dimensions allow the thread policy manager to be designed for various real time, computationally intensive, and interactive environments. Another interesting aspect of the thread policy manager is that it is possible to implement a completely static scheduler for a particular program if an optimal or near optimal schedule is known for the threads.

4.7.2.3Example Schedulers

To date, several different thread policy managers have been implemented and tested.

Global LIFO - This scheduler has one global queue in a LIFO order.

Global FIFO - This scheduler has one global queue in a FIFO order.

Local LIFO 1 Queue - This scheduler has one local queue per VP that is ordered in a LIFO manner.

Local FIFO 1 Queue - This scheduler has one local queue per VP that is ordered in a FIFO manner.

Local LIFO 2 Queue - This scheduler has two local queues per VP. One contains threads that have never been run and the other contains threads that have been. Both queues are ordered in a LIFO manner.

Local FIFO 2 Queue - This scheduler has three local queues per VP. One contains threads that have never been run, one contains threads that have been and were running, and the third contains threads that were blocked. All three queues are ordered in a FIFO manner.

Local LIFO 3 Queue - This scheduler has three local queues per VP. One contains threads that have never been run, one contains threads that have been and were running, and the third contains threads that were blocked. All three queues are ordered in a FIFO manner.

Each of these schedulers were implemented with less than 230 lines of code. 160 of those lines were in a library shared by all the schedulers. Given this library each schedular was implemented with less than 70 lines of code. We think this demonstrates how easily a thread policy manager can be customized.

These thread policy managers are provided as part of a library of standard thread policy managers that are delivered with the system. Thus, most *Sting* users will not find it necessary to implement a policy manager, rather they will simply load the appropriate

policy manager for their application from the library. The standard policy mangers implement not only scheduling decisions, but also initial placement and migrations decisions.

4.7.2.4Migration Decisions

The final issue in thread policy manager design concerns strategies for thread migration. Thread can be migrated for various reasons. The most common is to balance the load on a virtual machine across the various virtual processors. But there are two other important reasons for migrating a thread from one VP to another. The first is performance. It may be much more efficient to move a thread closer to the resources it is using, whenever possible. The second reason is reliability. If a processor fails a checkpointed thread can be migrated to another processor and resumed there.

As with the other policy decisions there are many possible strategies for migrating threads. We discuss a few that relate to load balancing below.

Migration for load balancing usually occurs when a VP is idle. This is because there is little or no overhead[1] in having an idle VP search for work to do, since it would not be doing work otherwise. The strategy does have to be careful to ensure that a VP does not continue to search for work if new work has arrived in the VP's ready queue(s).

Random Search - This strategy entails picking a VP at random and grabbing some number of threads, usually half, from it's queue(s). This strategy can be tuned in several way. The thread policy manager may try to migrate threads that have not yet started evaluation first. In order to reduce the cost of migration. If there are no threads that have not started evaluation then the policy manager may choose to migrate threads that are evaluating, or to look for another processor with threads which have not yet started to evaluate. The advantage of this strategy is that the amortized cost of finding work is good. The disadvantage is that it may decrease the locality of the threads in the virtual machine.

1. There is the potential to create more contention for shared resources however.

Ordered Search - Another strategy is for the thread policy manager to conduct an ordered search from VP that are close to it own to those which are farther away. This approach has the advantage that it has a better chance of maintaining the locality of threads.

Load Based - If a queue of processors ordered by their load is being kept to improve the initial mapping of threads to VPs, then it can be used by an idle processor to find the most heavily loaded processor and take some, again usually half, of its threads. Because there are many possible ways of measuring the "load" on a virtual processor, the thread policy manager is responsible for calculating it in a manner suitable to the policy being implemented.

There are many other possible strategies for using migration to balance the work on a machine and improve performance. The appropriate strategy will depend not only on a particular program, but also on the memory hierarchy and the communications topology of the physical machine.

4.8 Virtual Topologies

Virtual topologies are another novel aspect of *Sting*. The advantage of virtual topologies is that they allow the building of customized topology abstractions which model data dependencies or the communications structure of a program and map them onto a particular physical topology. These abstractions allow the programmer to map threads to the appropriate virtual processors which, in turn, are mapped to the appropriate physical processors. Furthermore, they allow the programmer to express the mapping in terms of the communications structure of the program, using self relative processor addressing, while completely ignoring the topology of the physical machine. The programmer can use either global addressing on the virtual machine or VP relative addressing. In either case expressivity is enhanced.

There are three main advantages to virtual topologies:

- The programmer can specify the mapping of threads to virtual processors in terms of the topology of the program or algorithm being implemented.
- The virtual topology can be specified in terms of the physical topology so that it maps virtual processors onto physical processors in an optimal or near optimal manner for supporting either memory efficiency or communications efficiency.
- Programs are portable across different physical topologies without modification and while maintaining their efficiency. Assuming the virtual topology is similarly efficient on different physical topologies.

Allowing the programmer to specify an algorithm in terms of its topology leads to an increase in expressivity and cognitive efficiency. For example, a divide and conquer algorithm can be expressed in terms of a binary tree topology, with expressions (parent-vp), (left-child-vp), and (right-child-vp) used to schedule threads on the appropriate virtual processor. The virtual topology can map a tree of arbitrary depth onto some physical topology so that the programmer need not be concerned with boundary conditions on the binary tree. It is much easier to write (and read) program where the thread/virtual processor mapping is expressed in terms of the structure of the algorithm rather than the structure of the physical machine on which it is running.

Each physical machine exports a set of procedures for accessing physical processors based on the physical topology of the machine. These procedures allow the virtual topology designer to map a virtual processor onto a particular physical processor. The topology designer is responsible for doing this in an efficient manner given the underlying physical topology.

The implementation of a virtual topology is independent of the program that is using that topology. This makes programs portable across different topologies. The same program can run, without modification, on different physical topologies by simply customizing the virtual topology to the new physical topology. Thus, virtual topologies

allow parallel algorithm to be portable and optimized for different architectures.

Virtual topologies are easy to implement in *Sting* because virtual processors are first class. Each virtual machine has an associated virtual topology. For example, a virtual machine might have a mesh topology while the physical machine on which it is running may have a hypercube topology. In such a case, the virtual topology would define the mapping of a virtual processor in the mesh to a physical processor in the cube.

The virtual topology is user customizable. Customization is made easy for two reasons: first the programmer can specify the topology using the

```
(create-virtual-processor pp) -> vp
```

operation to build data structures (and procedures to access them) containing virtual processors, because they are first class. Second, the mapping between the physical and virtual processor is specified when creating the virtual processor. The data structure which contains the virtual processor can be defined in such a way that an appropriate global addressing and/or the appropriate self-relative addressing is defined. Self-relative addressing expressions are defined in terms of the

```
(current-virtual-processor)
```

operation.

An example might be useful for demonstrating some of the ideas we have been discussing. Figure 4-i shows a procedure for mapping a virtual topology representing a 3D mesh onto a physical topology which is a 2D mesh.

```
(define (create-3D-mesh depth)
  (let* ((pm-width  (get-pm-width))
         (pm-height (get-pm-height))
         (3D-mesh   (make-array w h depth)))
    (let -*- ((i 0))
      (if (< i pm-width)
          (let -**- ((j 0))
            (if (< j pm-height)
                (let -***- ((k 0))
                  (if (< k depth)
                      (let ((vp (create-vp (get-pp i j))))
                        (set-vp-address vp (vector i j k))
                        (set-aref! 3D-mesh vp)
                        (-***- (+ k 1)))
                      (-**- (+ j 1))))
                (-*- (+ i 1))))
          3D-mesh))))
```

Figure 4-i : 3D to 2D Mapping

The procedure **create-3D-mesh** creates a three dimensional array of virtual processors. The array is the height and width of the physical machine with a depth equal to the **depth** argument to the procedure. The mapping collapses the three dimensional array onto the two dimensional array, so that each virtual processor in the depth dimension is mapped onto the same physical processor. The procedures **pm-height**, **pm-width**, and **get-pp** are provided by the physical machine kernel. Absolute addressing of virtual processors is simply an array reference into the array return by create-3D-mesh, i.e. **`(aref 3D-mesh i j k)`**.

The virtual policy designer creates a data structure representing the address of the VP and stores that address in the **vp-address** slot of the VP using the **set-vp-address** procedure. In our example this is a vector containing the coordinates of the point in the mesh that the virtual processor occupies.

The reader will notice that **create-3D-mesh** creates **depth** times as many virtual processors as there are physical processors, which means that **depth** virtual processors

will be multiplexed on each physical processor. A more efficient mapping is given in Figure 4-j .

```
(define (create-3D-mesh depth)
  (let* ((pm-width  (get-pm-width))
         (pm-height (get-pm-height))
         (3D-mesh   (make-array w h depth)))
    (let -*- ((i 0))
      (if (< i pm-width)
          (let -**- ((j 0))
            (if (< j pm-height)
                (let ((vp (create-vp (get-pp i j))))
                  (set-vp-address vp (vector i j))
                  (let -***- ((k 0))
                    (if (< k depth)
                        (block (set-aref! 3D-mesh vp)
                               (-***- (+ k 1)))
                        (-**- (+ j 1)))))
                (-*- (+ i 1))))))))
          3D-mesh))))
```

Figure 4-j : Improved 3D to 2D Mapping

In this mapping each element in the depth dimension of the array **3D-mesh** contains the same virtual processor. Thus the number of virtual processors created corresponds to the number of physical processors in the physical machine. Any threads mapped onto a processor in the third dimension will be mapped onto the same virtual processor. This is more efficient and retains the same locality properties as the first version, while eliminating the overhead of multiplexing **depth** number of virtual processors on the same physical processor.

Having created an absolute addressing procedure we can now create relative addressing procedures. We could, for example, define relative addressing procedures called **up-vp**, **down-vp**, **left-vp**, **right-vp**, **front-vp**, **back-vp**. These are nullary procedures which use **`(current-virtual-processor)`** to determined the values of these procedures. Figure 4-k shows the procedure **up-vp**.

```
(define (up-vp)
  (let ((address (vp-address (current-virtual-processor))))
    (aref 3D-mesh (vref address 0)
                  (vref address 1)
                  (+ (vref address 2) 1))))
```

Figure 4-k : Virtual Processor Relative Addressing in the UP Direction

up-vp uses the address associated with the virtual processor, i.e. **vp-address**, when it was created to access the virtual processor in the 3D mesh which is in the *up* relationship to itself. The procedure as presented in Figure 4-k does not handle boundary conditions, but this is trivially added. The definitions of the other procedures are similar.

The virtual topology we have presented in the above example is trivial; but it is sufficient to explain the fundamentals of constructing a virtual topology. Mapping a mesh onto a hypercube, or a binary tree onto an omega net, or any other mapping is done in the same way. That is, the steps for creating a virtual topology are:

- First, create a set of virtual processors which are mapped onto the appropriate physical processor.
- Then associate an address in the virtual topology with each virtual processor.
- Next store the virtual processor in a data structure which is used for absolute addressing in the virtual topology and define the appropriate access procedures to the data structure.
- Finally, define procedures for addressing relative to the current virtual processor.

The idea of mapping computations to specific processors was first explored by Hudak [Hud86], but his processors where neither first class (he used processor ids) nor virtual. User defined topologies were more difficult to implement because there were no

procedures to access the physical topology provided by the language.

A significant body of work on mapping one topology onto another optimally or near optimally exists. The work of Bhatt and Greenberg [Bha85] [Gre91] is particularly relevant to building efficient virtual topologies. Their algorithms and those of others[1] can be used in conjunction with *Sting* to build all kinds of virtual topologies which can be mapped optimally or near optimally onto different physical topologies.

Until now we have specified that each virtual machine has a virtual topology, but in fact a virtual machine may have more that one virtual topology mapped onto its physical processors. Thus complex algorithms that exhibit more than one structural topology can use the appropriate virtual topology for different parts of the algorithm.

Finally, we should point out that the user is not required to create the virtual topology. We expect that an implementation of *Sting* on a particular physical machine would provide standard libraries for creating virtual machines with particular topologies. For example, a procedure such as

(create-virtual-machine-as-3D-mesh *main height width depth*)

would create a virtual machine with the requested size and topology and execute the nullary procedure *main* in it.

1. [Aie90], [Aie91], [Bet88], [Cha88], [Ho87], [Lei85], [Ros81]

Chapter 5

The Abstract Physical Machine and Abstract Physical Processors

5.1 Introduction

As with other recent operating systems,[1] *Sting* is based on a micro-kernel, called the *Abstract Physical Machine*, but while it is similar in concept to other micro-kernel based operating systems, it introduces several important innovations. These are discussed below.

The abstract physical machine forms the lowest or kernel level of *Sting*'s software hierarchy, and plays three important roles in the *Sting* architecture. First, it provides a *safe, secure,* and *efficient* foundation which supports the virtual machine model. Second, it isolates the rest of the *Sting* architecture from hardware dependencies. And third, it controls and coordinates access to the physical hardware of the system.

Unlike most modern operating systems, *Sting* is designed to work on *both* single processor and MIMD multi-processor hardware platforms. MIMD systems can be categorized by their architecture and their processor inter-connection topology. *Sting*'s abstract physical machine implements idealized models of these components, improving both portability and customizability. However; these two components are not nec-

1. For example Chorus [11], Mach [12], or Psyche [9].

essarily independent of each other. Inter-processor communication may be through memory (e.g. in a physically shared memory machine), or an inter-processor communication network may be used to implement the shared virtual memory model (e.g. in machines with disjoint memory), or separate networks may be used for memory coherency and inter-processor communication. *Sting* supports three distinct classes of physical memory architecture found in MIMD machines, i.e. shared, disjoint, and partially disjoint (see Section 5.5.1). *Sting* also supports the various interconnection topologies used in MIMD machines, as discussed in Section 5.6.

The abstract physical machine resembles other micro-kernel operating systems. It has four components: processor control, virtual memory, inter-thread communication, and a device driver interface. Traditional operating system facilities, such as file systems, name servers, and network management are implemented as virtual machines. User programs communicate with these subsystems using the abstract physical machine's inter-thread communication facility.

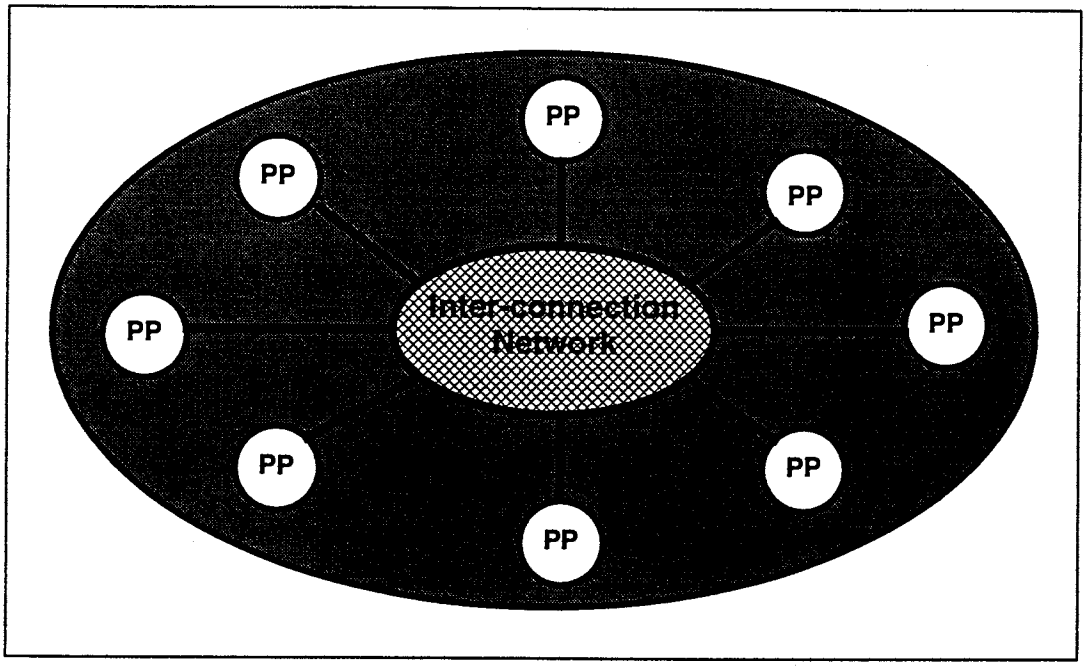

Figure 5-a : The Abstract Virtual Machine

Figure 5-a is a graphical depiction of the abstract virtual machine. The dark shaded part represents the physical memory. The abstract physical processors are represented as circles labeled PP. The physical processors are connected by the inter-connection network, which represents any possible topology.

The abstract physical machine is implemented by two distinct components: the *machine independent kernel* defines operations not dependent on physical hardware, and the *physical device kernel* defines machine dependent operations. The machine independent kernel is responsible for the interface to the other layers of the *Sting* architecture, as well as the interface to the physical device kernel. The physical device kernel is responsible for implementing portable abstract physical devices.

5.2 Problems with Current Micro-Kernels

Current micro-kernel designs exhibit several weaknesses, which *Sting* improves upon. In current micro-kernels, program code in the micro-kernel is significantly different, from that found in user programs. This occurs because many of the facilities available to user programs are not available at the kernel level. *Sting* addresses this problem by implementing the abstract physical machine in the *root virtual machine*. The root virtual machine has all the facilities that are available to any other program (or subsystem) running in a virtual machine including, a virtual address space, virtual processors, and threads. In addition, it has abstract physical processors, device drivers, and a virtual memory manager. This uniform program model provides the micro-kernel programmer with a significant increase in efficiency, both in programming time and execution time.

The fact that the abstract physical machine is implemented in a virtual machine has several other important implications:

- In *Sting*, unlike other operating systems, there are no heavyweight threads. All threads are lightweight.

- There are no kernel threads or stacks for implementing system calls. All system calls use the execution context of the thread making the system call. This is possible because portions of the abstract physical machine is mapped into every virtual machine in the system.

- Asynchronous programming constructs in the abstract physical machine are implemented using threads as in any other virtual machine. Threads in the abstract physical machine can be created, terminated, and controlled in the same manner that threads in any virtual machine can.

- When a thread blocks in the kernel it can inform its virtual processor that it has blocked. The virtual processor can than choose to execute some other thread. This is true for inter-thread communication as well as for I/O (e.g. page faults).

- Exceptions are handled in a uniform manner using standard threads, thus increasing programming efficiency. Furthermore, device driver implementers can use a broader range of programming techniques than would otherwise be available.

5.3 *Sting* Micro-Kernel Innovations

In addition to improving on the weaknesses found in other micro-kernels, *Sting* introduces several significant innovations. The most important is its ability to customize policies for virtual machine management. Each abstract physical processor is composed of two components: the virtual processor controller, which is responsible for controlling the state transitions that a virtual processor can make, and the virtual processor policy manager, which is responsible for making policy decisions for the controller regarding the scheduling and mapping of virtual processors to abstract physical processors. Because the policy manager is customizable, *Sting* can be targeted at different operating environments. The virtual processor controller and policy manager are discussed in detail below.

Another *Sting* innovation is the abstract physical machine's memory model. This model is called *Persistent Shared Virtual Memory* (PSVM). It is similar to the standard shared (distributed) virtual memory model, with the significant difference that the virtual address space persists across interruptions in system and/or virtual machine shutdowns.

There are two advantages to this model. First, it is very straightforward to implement persistent object systems using the PSVM model. Second, the entire *Sting* system or any virtual machine can be powered down and resumed without any loss of data. *Sting* is also intended to be small enough to be used on personal computers where suspending and resuming are required functionality.

Another innovative aspect of *Sting* is that the abstract physical processor is guaranteed to be always running a thread.[1] Thus, any instruction which runs on a physical processor is associated with some thread. The immediate consequence of this is that every exception is handled in the context of some thread. Each thread is in turn associated with a virtual processor, a virtual machine, an abstract physical processor and an abstract physical machine. Because of these associations, exceptions can be delivered to the appropriate handler at any layer of the *Sting* architecture. Figure 5-b shows the abstract physical machines position in the hierarchy of the architecture.

1. For example, the APM boots itself in the root thread of the root virtual machine.

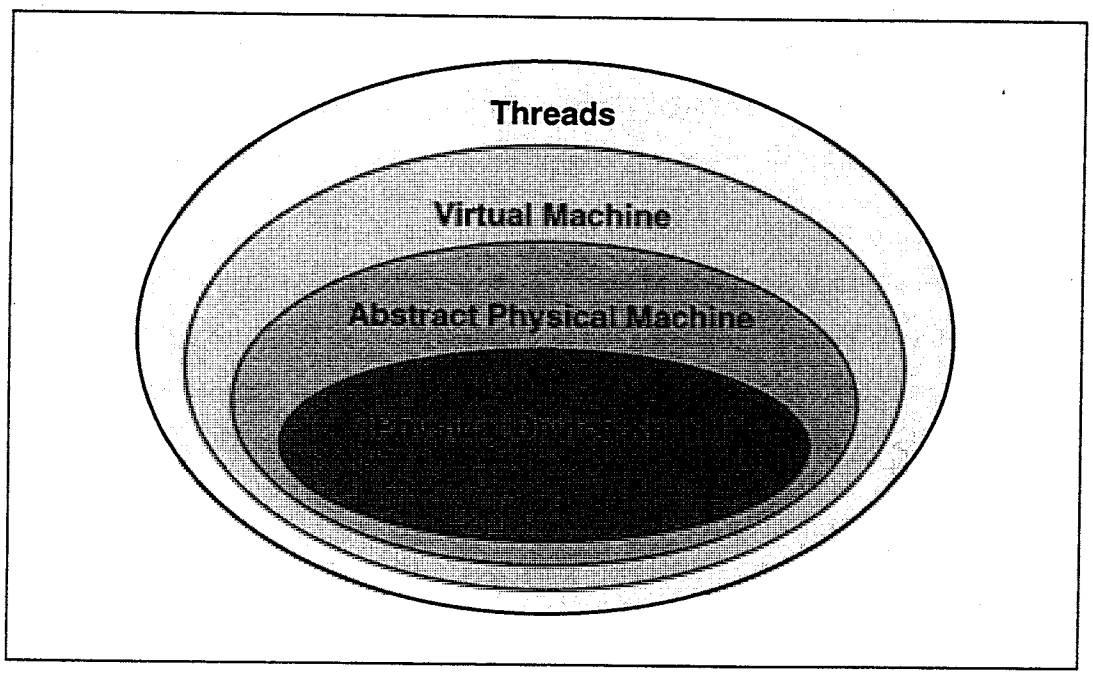

Figure 5-b : Levels of the Sting architecture

The final innovation also results from the uniformity gained by implementing the abstract physical machine in the root virtual machine. That is, exception handlers may use automatic storage allocation and reclamation (garbage collection), increasing both programmer efficiency and expressivity.

5.4 The Abstract Physical Machine

The abstract physical machine is composed of the following components:

- some number of abstract physical processors connected in a physical topology,
- a physical memory architecture implementing a PSVM, and
- other hardware devices such as clocks, and I/O controllers.

The abstract physical machine controls and coordinates the interaction of all hardware elements, including handling hardware device interrupts, and managing the physical

memory hierarchy. The abstract physical machine is also responsible for providing safe and secure access to the various hardware components for other parts of the *Sting* architecture.

In *Sting*, many virtual machines may run on the abstract physical machine, but there is only one physical machine.

5.5 Virtual Machine Creation and Destruction

The abstract physical machine is responsible for creating and destroying virtual machines. A new virtual machine is created by calling the ***create_virtual_machine*** procedure. In order to create a virtual machine the abstract physical machine does the following:

- it creates a virtual address space, and
- then maps the abstract physical machine into the virtual address space.
- It then creates a root virtual processor in the address space, and (possibly) creates other virtual processors.
- Next, it allocates the necessary abstract physical processors and maps the virtual processors onto them,
- then it maps the program to execute into the virtual address space, and finally,
- it schedules the root virtual processor to run on it's abstract physical processor.

Destroying a virtual machine happens as a result of calling the ***exit-virtual-machine*** procedure. When this procedure is called the following actions are taken:

- The ***thread-terminate-exception*** is sent to any non-root threads executing on any virtual processors in the virtual machine.

- When all the non-root threads have terminated the root thread on each virtual processor, except the root virtual processor, is sent the ***virtual-processor-terminate-exception.*** The virtual processor then requests that it's abstract physical processor terminate it.
- Any devices which the virtual machine has opened and not closed are closed.
- Any persistent areas in the virtual address space are unmapped, possibly being garbage collected before being unmapped.
- Finally, the root thread of the root virtual processor calls it's abstract physical processor, which de-allocates the virtual address space associated with the virtual machine.

5.5.1 Persistent Shared Virtual Memory

The *Sting* memory model is based on the concept of shared virtual memory. *Sting*'s memory model differs from that of other operating systems in so far as its shared virtual memory is *persistent.* This is a fundamental innovation of Sting.

Sting is based on shared virtual memory because it provides a parallel programming model which is simpler and more regular than the message passing model common on many current MIMD parallel processors. The simplicity and regularity of this model improves the programmer's cognitive efficiency. It is easier to think about algorithms when one can use references freely, without worrying about which processor's memory contains a particular object. Furthermore, message passing models can be implemented efficiently on a shared memory mode. *Sting* provides innovative message passing facilities that are discussed below.

The shared virtual memory model is *weakly coherent.* Although the memory is shared, it cannot be used for synchronization, and thus specialized constructs must be provided to synchronize access to shared mutable data. Weak memory coherency is preferable to strong memory coherency because it eliminates the need to synchronize

memory at every memory access. Instead memory access is only synchronized when explicitly requested, thus reducing the overall cost of synchronization. We will not go into the details of shared virtual memory or coherency here; rather, we note that there has been a significant amount of research demonstrating the viability of the shared virtual memory model, and a strong research effort continues in the area.

Because the virtual address space is *shared* it may be mapped onto more than one processor. Thus, although all reads and writes are guaranteed to be to the same memory location, the ordering on accesses to a location is non-deterministic. In order to ensure a particular ordering on a location or set of locations, a separate synchronization construct is necessary. There are many different ways to provide this synchronization. For example, the machine may provide a separate synchronization bus or network, or the processor may provide *load linked* and *conditional store* instructions.

It is the physical device kernel's responsibility to ensure that the address space is weakly coherent if the hardware doesn't. The shared virtual memory abstraction can be built on top of either a physically shared or a physically disjoint memory or a partially disjoint memory.

As implementations of shared virtual memory improve, it is likely that machines will be composed of processors which have both local and global memory. Examples of these types of architecture include the BBN Butterfly, Kendall Square KS1, IBM's Multi-processor RS6000 machine and several new commercial systems currently under development. *Sting* is designed to take advantage of this distinction between private and shared physical memory. It does so in two ways:

- It define the thread data structure so that it can exploit this distinction. (see Section 3.5)
- It provides a mechanism that assumes all dynamically created data are initially private and are dynamically moved to the shared memory when necessary (see Sections 3.5.4 and 3.5.5).

5.6 Abstract Physical Machine Topology

The topology of the abstract physical machine is the same as the physical hardware. If hardware processor 0 is a direct neighbor of hardware processor 1, then abstract physical processor 0 is a direct neighbor of abstract physical processor 1. Any virtual processor *N* running on abstract physical processor 0 is a direct neighbor of virtual processor *M* running on abstract physical processor 1. This means that any thread or virtual processor running on a physical processor can know how many hops it will take to communicate a message to any other thread or virtual processor.

5.6.1 Inter-Thread Communication

In order to build protected subsystems, e.g. a network name server, it is necessary that two threads on different virtual machines be able to communicate with each other. To support this requirement *Sting* provides an inter-thread communication (ITC) facility. This facility is similar to inter-process communication facilities found in other microkernels. Threads communicate with each other through ports.

While it is obvious that all threads in the same virtual machine can communicate with each other using shared virtual memory, it is less obvious that two threads on different virtual machines can also communicate using shared virtual memory. This can be accomplished by having the virtual address spaces of the two virtual machines partially[1] overlap and then using objects in the shared part of the address space for communication. This technique, however, may be inefficient on certain types of physical architectures where the communication network has either high latency and/or low bandwidth. For example, on a network of workstations connected together by an ethernet.

Ports allow threads, whether on the same or different virtual machines, to communicate with each other efficiently. There are two advantages of using ports to communicate between threads on different virtual machines: (1) It is simpler than creating an

1. It is also possible for the address spaces of two virtual machines to fully overlap.

overlapping address space and building the data structures necessary to communicate. (2) It is also guaranteed to be at least as efficient as using shared virtual memory, and possibly more efficient.

These same two advantages are true for inter thread communication in the same virtual machine, but if the threads are in the same virtual machine, there is a third advantage that is equally important, namely that threads can send messages containing valid references between themselves. This is not possible using a micro-kernel which are not based on shared virtual memory. The ramifications of this are important. It means that threads on the same virtual machine can use ports to efficiently communicate full fledged objects (i.e. full semantic and type information) between themselves. Typical message passing machines can not do this.

The ability to communicate full fledged objects, not only provides the programmer with increased expressivity, but it also avoids many of the costs found in typical message passing systems, e.g. the marshalling and unmarshalling of data, and the cost of copying the objects from memory into a message buffer.

The conventional argument against the use of shared virtual memory is that it is too inefficient. *Sting*'s ports allow the user to have all the benefits of shared virtual memory (in particular sharing object and type representations), while avoiding the potential costs associated with implementing communication in terms of memory.

5.6.2 Virtual Machine States and State Transitions

Each abstract physical machine manages the virtual machines that are mapped onto it. Figure 5-c [1] shows a state transition diagram for virtual machines.

1. Figure 5-c is a copy of Figure 4-d on page 88. It is reproduced here for the readers convenience.

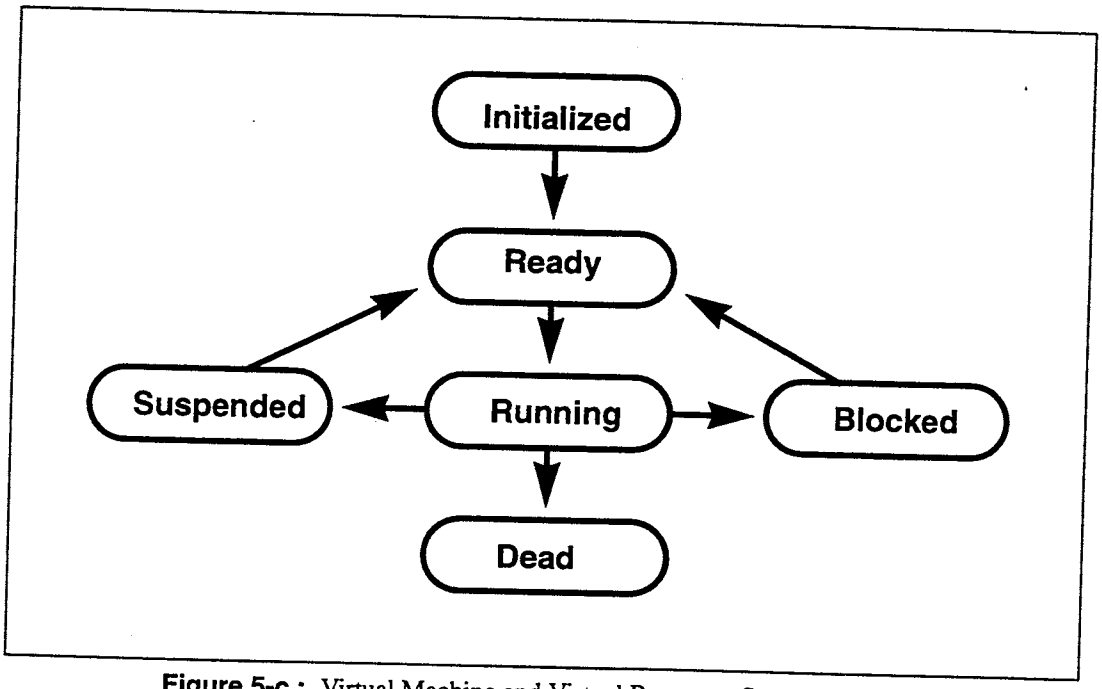

Figure 5-c : Virtual Machine and Virtual Processor State Transitions

Virtual machines can enter any of the following states:

Initialized - The virtual machine has been created, but it has never been run.

ready - The virtual machine is ready to run, but none of its virtual processors are currently running on any physical processor.

running - The virtual machine is running, which means that at least some of its virtual processors are running on physical processors.

blocked - The virtual machine is blocked waiting for some event. All virtual processors and the threads associated with them are also blocked.

terminating - The virtual machine is in the process of terminating itself. This includes disposing of all threads and virtual processors associated with it.

dead - The virtual machine has been terminated and it can no longer be run.

A running virtual machine can be blocked waiting for some event, in which case all of

its virtual processors and threads are also blocked. When the virtual machine is resumed, each virtual processor is scheduled to run on the physical processor associated with it. Virtual machines can also be suspended. If, when the virtual machine is suspended, a time quantum τ is specified the virtual machine is resumed automatically after τ time[1] has passed.

5.7 Abstract Physical Processors

Abstract physical processors multiplex virtual processors mapped onto them. Just as *Sting* uses abstract physical machines to hide architectural details of a hardware machine architecture, it uses *abstract physical processors* to hide the architectural details of a hardware processor architecture. In both cases, the abstractions provide portability.

The number of abstract physical processors in the abstract physical machine has a one to one correspondence with the actual number of hardware processors in the machine. Each abstract physical processor handles:

- the local portion of the physical memory architecture,
- exceptions, both synchronous and asynchronous, and
- its interconnection with the other processors

in the abstract physical machine. These aspects of the abstract physical processor are covered in the following three sections.

An important aspect of *Sting* is that each hardware processor is always associated with a current abstract physical processor, a current virtual processor, and a current thread. Thus, all instructions executed on a hardware processor are executed in the context of an abstract physical processor, a virtual processor, and a thread.

1. The time quantum can be specified to be real time or virtual time.

5.7.1 Access from/to Virtual Processors

Virtual processors are multiplexed on physical processors in the same manner that processes are multiplexed on a processor in more traditional operating systems. The abstract physical processor divides the instruction cycles of the hardware processor among the virtual processors mapped onto it. Abstract physical processors context switch virtual processors either because of preemption or because of an explicit request to do so by the virtual processor it is currently running.

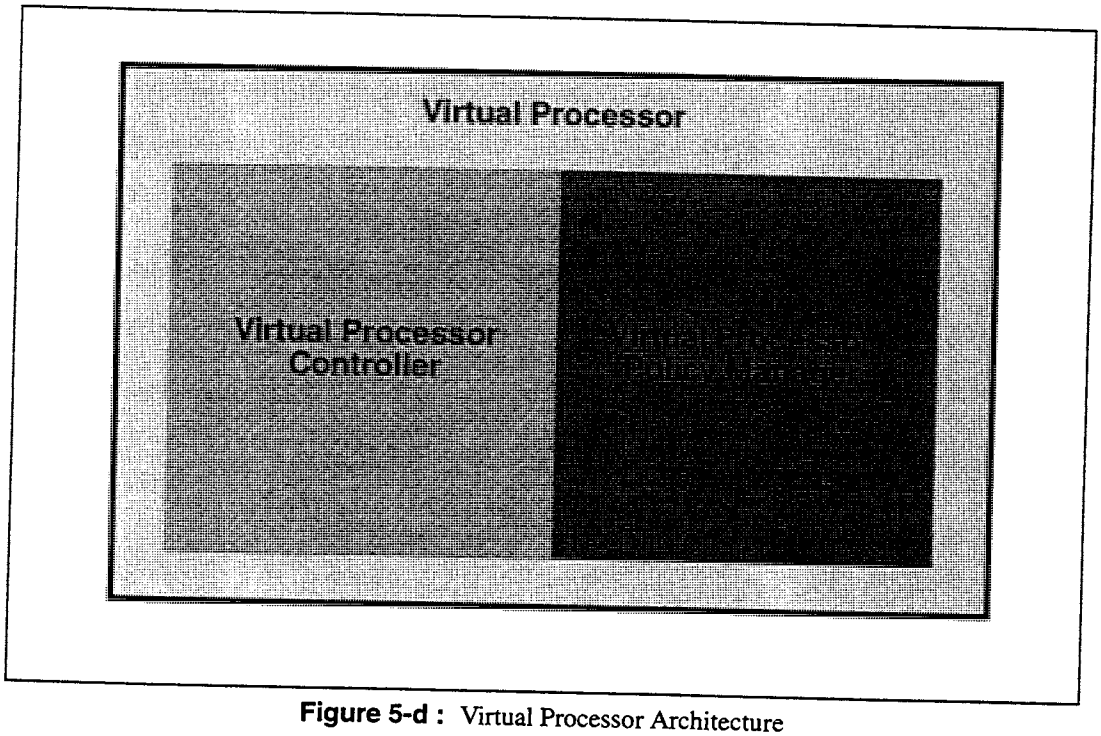

Figure 5-d : Virtual Processor Architecture

The physical processor abstraction is divided into two separate components, the *virtual processor controller* and the *virtual processor policy manager* (see Figure 5-d). Each physical processor has its own VP controller and VP policy manager. The VP controller defines the various mechanisms associated with the VP abstraction, while the VP policy manager makes any policy decisions required by the VP controller. This separation of control from policy enables the customization of physical machine policies without changing the code that implements the abstract architecture. The separa-

tion of control from policy in abstract physical processors is similar to the separation of control from policy in virtual processors. There are two principle differences between them. First, the control and policy management of virtual processors have different requirements than the control and management of thread. Second, the virtual processor policy manager can only be customized by a particular implementation of *Sting*, whereas the thread policy manager can be customized by any user.

5.8 Virtual Processor Management

When a virtual machine wishes to schedule a virtual processor on a physical processor it calls the virtual processor controller on that physical processor. Likewise, when a virtual machine wishes to remove a virtual processor from an abstract physical processor it calls the virtual processor controller on that physical processor.

5.8.1 Virtual Processor Controller

Each abstract physical processor's VP controller manages the virtual processors which are mapped onto it. The VP controller manages all virtual processor state changes. Virtual Processor states are similar to virtual machines states (see Figure 5-c). The virtual processor states are as follows:

- *ready* - The VP is not currently running on any physical processor, but is ready to run.
- *running* - The VP is currently running on a physical processor.
- *blocked* - The VP is blocked waiting for some event.
- *terminating* - The VP is in the process of terminating itself. This includes disposing of all threads associated with the VP, by either migrating them or terminating them.
- *dead* - The VP has been terminated and can no longer be executed on any physical processor.

The VP Controller performs context switches between VPs. When a VP context switch occurs, the current VP state and the state of the thread running on the VP are saved and another VP is run.

5.8.2 Virtual Processor Policy Manager

Each physical processor also contains a VP policy manager that makes all policy decisions relating to the scheduling and migration of virtual processors on physical processors. The VP controller is a client of the VP policy manager, i.e. the VP controller calls the VP policy manager to make decisions about the following:

- which abstract physical processor to associate with a VP when the VP is created;
- which VP to run next when the current VP releases the physical processor for some reason; or
- when and which VP to migrate from/to another abstract physical processor.

While the VP controller is conceptually the same on each physical processor, each VP policy manager may be different.

The VP policy manager presents a well-defined interface to the VP controller. The data structures which the VP policy manager uses to make its decisions are completely private to it. These data structures may be local to a particular VP policy manager or shared among the various instances of the VP policy manager, or some combination thereof, but no other component of the system has access to them. The VP policy manager can be customized to provide different behaviors to different instances of *Sting*. This functionality allows it to be customized for different operating system environments as diverse as real time, interactive, or computationally intensive systems. As mentioned above, this customization cannot be done dynamically, rather it must be done by linking in a new virtual processor policy manager when a particular version of *Sting* is built.

5.9 Exceptions and Continuations

One of the most important functions of the abstract physical machine is handling any exceptional conditions that might occur at the hardware level. Hardware exceptions are extraordinary or non-normal conditions which occur in the course of executing a program. They are classified as either *synchronous* or *asynchronous*. Synchronous exceptions are those that result from the attempt to execute an instruction (e.g. TLB miss, invalid instruction, or memory access violation). They are synchronous with respect to the instruction stream the processor is executing. Asynchronous exceptions, also known as interrupts, are those that are caused by some condition that is not related to the instruction stream. In general, asynchronous exceptions are caused by device interrupts (e.g. count down timer expiration, or a dynamic memory transfer completion notification), but they are also occasionally caused by some internal error in the processor (e.g. a parity error on an internal bus).

All exceptions have an associated type. Each type has a *handler* that performs some action to deal with the exceptional condition raised. Exception handlers are procedures that execute within a thread; they take as arguments the data associated with a particular instance of the exception type. Particular exception types are not discussed here, since the exceptional circumstances *Sting* encounters are similar to those found on other operating system. Rather, the discussion will concentrate on the general mechanisms *Sting* uses to deal with exceptions.

As mentioned above, every physical processor has a current thread, and every instruction executed on a processor is executed in the context of that thread. Similarly, every exception, whether synchronous or asynchronous, also occurs in the context of some instruction, i.e. the instruction being executed at the time the exception occurs. Thus, an exception raised on processor P is handled using the execution context of P's current thread. Unlike other operating systems which use special exception stacks in the micro-kernel to handle exceptions, *Sting* uses the execution context of the current thread to dispatch the exception to the appropriate handler.

When an exception occurs, an exception dispatcher is invoked *implicitly* in the context of the current thread. Before this is done the current thread is interrupted, and its current continuation is saved. The exception dispatcher is then invoked using the execution context of the current thread. The dispatcher in turn directs the exception to the intended thread. Since the interrupted thread has access, directly or indirectly, to every other component of the *Sting* architecture, the exception dispatcher can direct the exception to any thread it desires, including the current thread.

Exceptions are always dispatched to threads. Dispatching the exception involves:

- finding the thread responsible for handling the exception, i.e. the target of the exception;
- then if the target is *running*, interrupting it and saving its current continuation, and finally,
- pushing a continuation composed of the exception handler and its arguments onto the target thread's stack.

After dispatching the exception, the exception-dispatcher has several choices about what to do next.

- It can resume the current thread, i.e. the one the dispatcher is running in, by simply returning into it.
- It can resume the thread receiving the exception.
- It can call the current thread controller to run another thread.

The dispatcher makes these choices based on the priority of the exception and the priority of the thread receiving the exception.

When the thread receiving the exception is resumed it will handle the exception by executing the pushed exception continuation. Using continuations to handle exceptions is one of the novel aspects of *Sting*. An exception continuation can be thought of as a procedure of no arguments, e.g.

```
(lambda () (handler arg1 arg2 ...),
```

that is a closure containing the handler for the exception type and the data, i.e. arguments, associated with the particular instance of the exception. This representation of exceptions is elegant in several ways:

- Handling an exception simply involves calling it, since it is a thunk.
- The exception is handled in the execution context of the thread receiving it.
- The exception once dispatched becomes the current continuation of the receiving thread and it is executed automatically when the thread is resumed.
- The exception is not handled until the receiving thread resumes execution, Determining when this happens depends on both the exception's priority and the receiving thread's priority.

When the exception handler finishes, the thread handling the exception continues executing where it was interrupted (if the exception handler simply returns), i.e. it returns into the continuation that was saved prior to pushing the exception continuation. However, the exception handler need not invoke the saved continuation. It may decide to continue the execution of the thread in some dynamically enclosing continuation.

```
 1: (define (exception-dispatcher type . args)
 2:   (save-current-continuation)
 3:   (let ((target handler (get-target&handler type args)))
 4:     (cond ((eq? target (current-thread))
 5:            (apply handler args))
 6:           (else
 7:            (signal target handler args)
 8:            (case ((exception-priority type))
 9:              ((continue)   (return))
10:              ((immediate)  (switch-to-thread resume target))
11:              ((reschedule) (thread->ready resume)))))))
```

Figure 5-e : Pseudo Code for the *Sting* Exception Dispatcher

Figure 5-e shows pseudo-code for the *Sting* exception dispatcher. In line 2, the current continuation is saved on the stack of the current thread. The continuation can be saved on the stack because it cannot escape and it will only be called once. On line 3[1] the dispatcher finds the thread for which the exception is intended and the handler for the exception type. Line 4 checks to see if the target of the exception is the current thread and if so (line 5) does not push the exception continuation. Rather, the dispatcher simply applies the handler to its arguments. This is valid since the dispatcher is already running in the context of the exception target, i.e. the current thread.

If the target of the exception is not the current thread, the dispatcher sends the exception to the target thread (line 7). Sending a thread a signal is equivalent to interrupting the thread and pushing a continuation containing the signal handler and its arguments onto the thread's stack, and resuming the thread which causes the signal handler to be executed.

After signaling the target thread, the handler decides which thread to run next on the processor (line 8). It may be one of: itself (line 9), the target thread (line 10), or the thread with the highest priority (line 11). If the target thread is not running on the current virtual processor, and it is the thread being resumed on the physical processor, then a virtual processor context switch must also occur. This is done in **switch-to-target** on line 10. For a more detailed discussion of thread execution contexts, and continuations, see Chapter 3.

5.9.1 Synchronous Exceptions

Synchronous exceptions are generated when a physical processor recognizes some exceptional circumstance caused by the execution of an instruction. Some of these are error conditions such as *invalid memory reference* or *invalid instruction*, some are potential error conditions, such as *integer* or *floating point overflow*, others are internal control conditions, such as *cache miss* or *page fault*, and finally, some are program requested exceptions such as *system call* or *break* instructions.

---

1. A multiple valued let.

All synchronous exceptions are caused and handled by the current thread running on the processor. This is because the instruction causing the exception is part of the current thread, and it is targeted to that thread, i.e. all synchronous exceptions are always targeted to the current thread. Synchronous exceptions appear to the rest of the system as a nullary procedure invoked by the current thread. Furthermore, it will appear as though the exception handler has been invoked by the instruction which caused the exception. The code in line 4 of Figure 5-e could be modified to

```
((or (synchronous? type) (eq? target (current-thread)))
```

Thus, taking advantage of the fact that all synchronous exceptions are handled by the current thread.

The handler for a particular synchronous exception has four options:

- Signal an error.
- Perform some auxiliary operation which resolves the exception and then resumes the thread by calling its current continuation. The thread is resumed implicitly by simply returning from the exception handler and continuing the instruction which caused the exception.
- Performing some auxiliary operation which resolves the exception and then calling some continuation other than the current continuation, this is equivalent to returning from the handler into a different part of the instruction stream which caused the exception.
- Passing the exception to a dynamically enclosing exception handler which in turn has the same four options.

The exception handler can use any of the *Sting* system facilities while handling the exception. All asynchronous exceptions are interruptible, unless interrupts have been explicitly disabled by the handler.

As with software exceptions, any synchronous exception handlers may be dynamically bound to a new handler. In order to find the appropriate handler for an asynchronous

exception the dispatcher searches the dynamic exception environment for the nearest dynamically enclosing handler.

5.9.1.1 Asynchronous Exceptions

An asynchronous exception or "interrupt" occurs when some hardware device in the system signals a hardware processor that an external event has occurred that needs its attention. For example, an I/O controller might signal a processor that there is data available for it. Asynchronous exceptions are "asynchronous" with respect to the current instruction stream. They are significantly different from synchronous exceptions in that they can relate to any thread in the system not just the current thread. Some examples of asynchronous exceptions are *data ready* on some device, *DMA transfer complete*, or *countdown timer expiration.*

Each hardware device connected to the physical processor has a set of asynchronous exception types which it can generate. Each asynchronous exception type can have a thread associated with it; exceptions of that type, when raised, are directed to that thread. Each abstract physical processor and each virtual processor in *Sting* has associated with it an exception thread. Exceptions can be targeted not only to threads, but also to physical processors and virtual processors. Exceptions that are directed to an abstract physical processor or a virtual processor are handled by the exception thread associated with that processor. When an interrupt occurs, all interrupts at a lower priority are disabled until the exception handler either completes or enables them explicitly.

Exception handlers have a large number of options available to them for processing the exception. Some of these options are particularly important for building device drivers which handle interrupts. Examples of these options are:

- The handler can process the exception immediately, deferring all other interrupts at the same or lower priority.
- Multiple exceptions of the same type can be handled by the same thread. The thread can be interrupted with new exceptions during the

course of handling a previous one. This can be done by having the handler put the data associated with the exception on a queue. The queue can be ordered using any appropriate priority scheme. If a new exception arrives for the thread while it is handling a previous one the new exception is queued and the thread continues handling the previous exception. When it completes handling the exception, the thread checks the queue for any other exceptions waiting to be handled. The waiting exceptions are handled in the order of their priority. Thus, exceptions of the same type can be prioritized according to any ordering the device driver implementer desires.

- Another technique is for the exception handler to create a new thread to process each exception. This technique is particularly important if the handler for a particular exception type blocks while processing. Creating a new thread to process the exception allows the exception handler thread to continue handling other exceptions.

There is another important distinction between *Sting*'s exception handling facilities and those found in other operating systems. Since threads that handle exceptions are no different from other threads in the system (i.e. they have there own local and shared heaps) and since exception handlers are ordinary procedures, *Sting* exception handlers can allocate storage dynamically. Exception generated data will be automatically recovered by the garbage collector in the same way that any other storage in the system is recovered. This uniformity between the exception handling mechanism and the rest of *Sting* allows the device driver implementer programming expressivity and efficiency not available in other systems, while at the same time opening up new implementation possibilities.

5.10 Physical Device Kernel

The physical device kernel allows *Sting* to be easily portable across different hardware platforms. The idea of isolating hardware dependencies in an operating system is not new, and modern operating systems generally employ this idea. But *Sting* is inno-

vative in the way it implements various aspects of this interface, (1) most notably its use of continuations, (2) in the uniform use of threads in both kernel and user space, and (3) allowing device drivers to be directly callable in user space.

The physical device kernel implements an abstraction of three different components of the hardware:

- the physical memory architecture,
- the physical topology, and
- any other physical devices which are attached to the system.

The physical device kernel supports the various hardware architectures mentioned above by defining abstractions of the physical hardware. The interface from the abstract physical machine to the physical device kernel is the same on any architecture implementing *Sting*. Thus, the physical device kernel abstraction maps the same operating system architecture onto different hardware architectures. On multi-processors, it also hides the distinction between physically shared and physically disjoint memory.

5.10.1 Physical Memory Architectures

The physical device kernel is responsible for managing its portion of the physical memory. Shared memory machines can be divided into two classes, those with uniform memory access (UMA) and those with non-uniform memory access (NUMA). In UMA systems all memory locations can be accessed in approximately the same amount of time. In NUMA machines some portion of memory is physically closer to a given processor then the rest of memory and can therefore be accessed more quickly than non-local memory. Examples of UMA machines include the Encore Multimax, the Sequent Symetry, and the SGI PowerSeries, as well as many other commercial systems. Examples of NUMA machines include the BBN Butterfly, the Stanford DASH, and the IBM RP^3, These shared memory systems ensure memory consistency using a variety of protocols.

Examples of *disjoint memory machines* include the Intel Hypercube, the nCube, and Masspar computers. In these systems, each processor has its own local memory. Communication between processors is done using explicit message passing. The local memory is always consistent with respect to its processor, and thus there is no memory consistency problem with respect to the physical memory architecture. Some work has been done on implementing shared virtual memory on these systems using software and compiler technology, but to date it requires explicit declaration in the programs to support shared variables.

The third class of memory architecture is called *partially disjoint.* In these systems each processor is connected to a local as well as to a shared physical memory. The advantage of this type of architecture is that only the shared memory needs consistency support. Local memory does not require it. The real advantage of these systems is that all data which is either local to a particular thread, including but not limited to stacks, or which is immutable (i.e read only data) can be stored in the processor's local memory. Only data which is both shared and mutable needs to be stored in the physically shared memory. The premise underlying this type of architecture relies on the well established fact that the amount of shared mutable data is a small portion of almost all programs and therefore the bandwidth requirements on the network that interconnects the shared memory are an order of magnitude less that on a fully shared memory machine. Partially disjoint memory is likely to become the memory architecture of choice in the future, and *Sting* is designed to take advantage of this type of architecture.

5.10.2 Parallel Computers Composed of Networks of Workstations

Another goal of *Sting* is aimed at supporting an important type of parallel computer which is not mentioned in the above discussion, namely parallel computers built out of networks of workstations (and other parallel machines). These systems are referred to as network parallel processors (NPP) or Hypercomputers. These architectures currently have disjoint physical memories, but in the future we expect these networks to

have partially disjoint physical memories, or even shared physical memories.

An important difference between parallel machines built out of networks and the standard MIMD machines mentioned above is that NPP systems can add and remove physical processors to and from the system dynamically. Additionally, decisions concerning when a processor is donated or withdrawn to/from the abstract physical machine is made by the hardware machine which is adding/removing the processor(s) to/from the abstract physical machine. The abstract physical machine can dynamically add a physical processor to the system using the following two strategies:

- If any of the existing abstract physical processors is running more than one virtual processor, one or more of them along with their associated threads are migrated to the new abstract physical processor. The virtual processors migrated are determined by the virtual machine policy manager discussed in Section 5.8.2.

- If no existing abstract physical processor has more than one virtual processor then the existing virtual machines are notified of the existence of a new abstract physical processor by the abstract physical machine and any virtual machine can at its discretion request that the abstract physical machine add a new virtual processor to it.

Likewise, when an abstract physical processor is dynamically removed from its abstract physical machine, any of the virtual machines that have a virtual processor mapped onto that abstract physical processor, are notified that it is being removed from the system. The virtual machine then has at least three options:

- it can migrate the virtual processor, with its associated threads, to another abstract physical processor,

- it can migrate the threads associated with the virtual processor to other virtual processors and then remove that virtual processor from the virtual machine, or

- it can terminate all the threads on the virtual processor and then remove the virtual machine.

Just as Network Parallel Machines can have any of the physical memory architectures described above, they can also have any of the physical inter-connection topologies mentioned above. When a new physical processor is added to (or removed from) the abstract physical machine the appropriate information is added to or removed from the abstract physical machine. The topology mechanisms, which are described below, work on both traditional MIMD and Network Parallel Machines. Finally, any network of inter-connected parallel machines can have more than one abstract physical machine running on it. This capability permits effective partitioning of the network.

5.11 Abstract Physical Machine Topology

Sting is designed to support any possible physical interconnection topologies. There have been a large number of different topologies used in various parallel processors. These include buses, hierarchical buses, 2D and 3D meshes, hypercubes, omega nets, and fat trees. An important aspect of *Sting* is that each abstract physical machine knows all processor locations in its physical topology. This information is made available to virtual machines, allowing them to implement virtual topologies that are transparently mapped onto the actual physical topology of the hardware. For a further discussion of virtual topologies in *Sting* see Section 4.8.

Chapter 6

Results

Sting has been implemented on an eight processor Silicon Graphics PowerSeries 480. The machine is composed of eight 75mhz MIPS R3000 processors communicating through a shared memory system with snoopy caches. Synchronization is supported at the hardware level using a synchronization bus which is mapped into a region of the physical memory. Synchronization is achieved by loading (locking) and storing (unlocking) into addresses in that region of memory.

Sting has been implemented on top of Unix. While most of the data structures of the abstract physical machine are implemented in the current system, we have relied on Unix to provide I/O handling, including paging.

The *Sting* system has been implemented using the language T [RA82], a dialect of Scheme [CR91] and an enhanced version of the ORBIT compiler [KKR+86]. In this chapter we present statistics gathered using various benchmark programs. Some are from the suite used by Mohr [Moh92] to benchmark lazy tasks in Mul-T. Others were programmed by various *Sting* users.

6.1 Statistical Categories

Each virtual processor in the *Sting* system maintains statistics about threads, execution contexts, and mutexes. Each of the tables presented later in this chapter will contain entries for each of the statistics.

The following statistics about *threads* are recorded by each VP:

Created - The **number of thread created by the program.**

Delayed - The number of threads created in the delayed state.

Scheduled - The number of threads scheduled to evaluate.

Absorbed - The number of threads absorbed by other threads.

Blocked - The number of times a thread blocked for any reason.

Resumed - The number of times a blocked thread resumed execution.

Determined - The number of threads that determined (i.e. completed) a value.

Wait - The number of times a thread blocked by making a call to thread-wait. This number is included in the Blocks count.

Resumed from Wait - The number of times a thread blocked in thread-wait resumed execution.

Blocked on Group - The number of times a thread blocked on a group of other threads. This occurs as a result of calls to wait-for-n-threads.

Resumed from Block on Group - The number of times a thread, which was blocked on a group of threads, resumed execution.

Suspended - The number of times a thread was suspended.

Terminated - The number of times a thread was forcibly terminated.

Idle - The number of times the virtual processor had no work to do. When a virtual processor has no work it runs its root thread which increments this counter and then spins until there is more work to do.

Migrated - The number of thread explicitly migrated from one virtual processor to another.

The following statistics about *thread execution contexts* are maintained by the system:

Created - The number of thread control blocked created. This occurs when the virtual processors pool of thread control blocks is empty and it must create a new thread control block.

Allocated - The number of thread control blocks allocated from the virtual processor's pool of thread control blocks.

Re-used - The number of thread control blocks that were reused because an unevaluating thread was proceeded by a thread which had just terminated.

The following statistics about *mutexes* are maintained by the system:

Created - The total number of mutexes created by the system.

Acquired - The number of times a mutex was acquired.

Released - The number of times a mutex was released.

6.2 Benchmarks

Sting's performance has been tested using the following benchmarks:

Abisort - a program that does an adaptive bitonic sort,

Allpairs - a program that find the all pairs shortest path problem,

Matrix Multiply - a program that multiplies two matrices, and

Thread Queens - a program that solves the N queens problem using threads.

Tuple Space Queens - a program that solves the N queens problem using first class tuple spaces.

N Body - a program that solves the N Body program using the Barns-Hut algorithm. N Body is implemented using tuple spaces.

Thread Policy Managers - The final benchmark shows the results of running Abisort on eight processors using six different thread policy managers. The program was run on each different policy manager without modification:

The results of each of these benchmark are placed in four tables. The first three show statistics for each processor in the computation. The first table shows statistics for the program when run on one and two processors. The second table shows statistics for four processors and the third table shows statistics for eight processors. These three tables show the "balance" between the processors. The fourth table consolidates the totals for all processors.

Each table also includes statistics for a virtual processor (VP_0) that was running a thread containing the read-eval-print loop that was used to execute the program. Aside from creating the initial thread VP_0 did no work.

The results of the thread policy manager benchmark are placed into 7 tables. The first six show the statistics for each of the eight processors under the six different policy managers. The seventh consolidates the first six in order to compare their performance.

Several of these benchmarks are extremely fined grained (i.e. each thread does almost no work), particularly those from Mohr. Mohr originally ran these benchmarks on the Encore MultiMax, whereas the processors on the SGI machine on which we ran these benchmarks are an order of magnitude faster then those on the MultiMax. This increase in speed makes the granularity much finer.

6.3 Abisort

Abisort performs an "adaptive" bitonic sort [BN89] of n = 16,384 numbers. The adaptive algorithm achieves optimal complexity (O(n log n) rather than the O(n $\log^2$ n) of the standard bitonic sort algorithm) by storing bitonic sequences in a special tree data structure. Still, adaptive bitonic sort performs about twice as many comparisons as a merge sort, and has somewhat greater bookkeeping costs. However, its parallel divide-and-conquer merge operation allows virtually linear speedup when $n >> p$. Such speedup is not possible with straightforward implementations (on MIMD machines) of other divide-and-conquer sorts such as merge sort and quicksort which contain significant sequential phases.[1]

Abisort creates a tree containing 106497 threads. As with most tree based sorting algorithms each node has data dependencies with its children. This can been seen in when abisort is run on a single processor (see Table 6-a). Every thread in the tree can be *absorbed* by its parent, so only one thread has an execution context and every other thread is evaluated in that execution context. Even with eight processors almost all threads are absorbed by other threads.

Abisort was run using a thread policy manager that implemented a global lifo scheduler. This type of scheduler should balance the load on the processors, but incur increasing contention on the global queue as more processors are added. Table 6-a, Table 6-b, and Table 6-c show that this does in fact happen and the work load is extremely well balanced across the processors. One of the reasons for this is that there is almost no blocking; however, as we increase the number of processors starvation starts to occur (even with of 100,000 threads!) because of the program's extremely fine granularity. This can be seen by looking at the idle statistic in Table 6-d. As more processors are added to the computation the number of times a processor goes idle, for lack of work, increases super-linearly. Thus, starvation accounts for the sub-linear speedup. On four processors the efficiency is 97%, but on eight processors it has fallen

1. Abisort is one of Mohr's benchmarks and the description of the algorithm is largely taken from [Moh91].

to 80%.

Sting's ability to conserve storage and increase locality can be seen in the thread execution context statistics. When abisort is executed using one processor only one execution context is created. Thus, one context is used to evaluate 106497 threads! Even with eight processors only 50 contexts are created. Notice that as the number of execution contexts increases, there are fewer opportunities for thread absorption, but the number of re-used contexts increases. A re-used context is a "hot" context since it is already loaded into the cache.

Threads	VP_0	VP_1	Total	VP_0	VP_1	VP_2	Total
Created	1	106496	1064967	1	54262	52234	106497
Delayed	0	0	0	0	0	0	0
Scheduled	1	106496	106497	1	54262	52234	106497
Absorbed	0	106496	106496	0	54258	52227	106485
Blocked	0	0	0	0	4	5	9
Resumed	0	0	0	0	4	5	9
Determined	0	106496	106497	0	54263	52234	106497
Wait	0	0	0	0	4	5	9
Resume Wait	0	0	0	0	4	5	9
Block on Group	0	0	0	0	0	0	0
Resume Group	0	0	0	0	0	0	0
Suspended	0	0	0	0	0	0	0
Sleep	0	0	0	0	0	0	0
Terminated	0	0	0	0	0	0	0
Idle	0	0	0	0	3	1	4
Migrated	0	0	0	0	0	0	0
Execution Contexts							
Created	1	1	2	0	3	4	7
Allocated	0	0	0	0	6	6	12
Reused	0	0	0	0	2	0	2
Mutexes							
Created	1	106500	106501	1	54274	52250	106533
Acquired	2	532488	532490	2	271375	261235	532621
Released	2	532488	532490	2	271375	261235	532621

Table 6-a : Abisort with 1 and 2 Processors

Threads	VP_0	VP_1	VP_2	VP_3	VP_4	Total
Created	1	26540	26654	26672	26630	106497
Delayed	0	0	0	0	0	0
Scheduled	1	26540	26654	26672	26630	106497
Absorbed	0	26527	26640	26664	26623	106454
Blocked	0	12	10	8	8	38
Resumed	0	12	10	8	8	38
Determined	0	26533	26660	26670	26634	106497
Wait	0	12	10	8	8	38
Resume Wait	0	7	10	11	10	38
Block on Group	0	0	0	0	0	0
Resume Group	0	0	0	0	0	0
Suspended	0	0	0	0	0	0
Sleep	0	0	0	0	0	0
Terminated	0	0	0	0	0	0
Idle	0	13	4	11	7	35
Migrated	0	0	0	0	0	0
Execution Contexts						
Created	0	3	6	5	4	18
Allocated	0	7	9	8	9	33
Reused	0	4	1	4	5	14
Mutexes						
Created	1	26552	26678	26692	26646	106585
Acquired	2	132805	133393	133488	133273	532978
Released	2	132805	133393	133488	133273	532978

Table 6-b : Abisort with 4 Processors

Threads	**VP_0**	**VP_1**	**VP_2**	**VP_3**	**VP_4**	**VP_5**	**VP_6**	**VP_7**	**VP_8**	**Total**
Created	1	13582	12960	13334	13314	13450	13263	13356	13237	106497
Delayed	0	0	0	0	0	0	0	0	0	0
Scheduled	1	13582	12960	13334	13314	13450	13263	13356	13237	106497
Absorbed	0	13550	12921	13291	13280	13409	13242	13333	13202	106228
Blocked	0	25	34	34	32	30	27	22	34	238
Resumed	0	25	34	34	32	30	27	22	34	238
Determined	0	13573	12951	13332	13308	13428	13294	13363	13248	106497
Wait	0	25	34	34	32	30	27	22	34	238
Resume Wait	0	23	27	39	26	18	40	31	34	238
Block on Group	0	0	0	0	0	0	0	0	0	0
Resume Group	0	0	0	0	0	0	0	0	0	0
Suspended	0	0	0	0	0	0	0	0	0	0
Sleep	0	0	0	0	0	0	0	0	0	0
Terminated	0	0	0	0	0	0	0	0	0	0
Idle	0	12	10	14	12	16	15	12	11	102
Migrated	0	0	0	0	0	0	0	0	0	0
Execution Contexts										
Created	0	7	8	5	6	10	4	3	9	52
Allocated	0	16	22	18	17	22	17	13	29	154
Reused	0	16	14	19	16	8	21	14	14	123
Mutexes										
Created	1	13610	12992	13354	13338	13490	13279	13368	13273	106737
Acquired	2	68148	68148	60728	66864	67491	66709	67079	66571	535028
Released	2	68148	68148	60728	66864	67491	66709	67079	66571	535028

Table 6-c : Abisort with 8 Processors

Threads	**1**	**2**	**4**	**8**
Created	1064967	106497	106497	106497
Delayed	0	0	0	0
Scheduled	106497	106497	106497	106497
Absorbed	106496	106485	106454	106228
Blocked	0	9	38	238
Resumed	0	9	38	238
Determined	106497	106497	106497	106497
Wait	0	9	38	238
Resume Wait	0	9	38	238
Block on Group	0	0	0	0
Resume Group	0	0	0	0
Suspended	0	0	0	0
Sleep	0	0	0	0
Terminated	0	0	0	0
Idle	0	4	35	102
Migrated	0	0	0	0
Execution Contexts				
Created				50
Allocated	0	12	33	154
Reused	0	2	14	123
Mutexes				
Created	106501	106533	106585	106737
Acquired	532490	532621	532978	535028
Released	532490	532621	532978	535028
Timing (secs)				
	65.2	29.7	16.8	9.1

Table 6-d : Abisort with 1, 2, 4, and 8 Processors

6.4 Allpairs

Allpairs solves the all-pairs shortest path problem [Aho, *et al*] on a directed linear graph of n = 117 nodes (footnote) using a parallel version of Floyd's algorithm. Starting with an n x n connectivity matrix C, where $C_{i,j}$ gives the length of the edge connecting nodes i and j (or 0 if i and j are not adjacent), execution continues until $C_{i,j}$ contains the length of the shortest path from i to j for all i and j. The algorithm iterates sequentially through all vertices K. During step K, all pairs of vertices are checked to see if going through vertex K produces a shorter path; that is Ci,j is updated if $C_{i,k} + C_{k,j} < C_{i,j}$. These operations may all proceed in parallel. To see why, note that in step k no element of row k or column k will change; this is so because $C_{k,k} = 0$. Since all computations in step k will only reference values from row k and column k, all vertex pairs can be safely handled in parallel during a given step.[1]

Handling all vertex pair tests in parallel would produce a rather fine-grained program, but this is not necessarily a problem on a MIMD machine because a coarse-grained parallelism is easily obtained by having each (potential) thread handle the n vertex pair tests in a single matrix row. This is the strategy adopted for **allpairs**. Thus the parallel version of **allpairs** has n sequential steps separated by barrier synchronization; in each step there are n potentially parallel tasks. The tasks are created by a divide-and-conquer traversal of the index set of the matrix; the additional overhead of such a traversal compared to an interactive traversal is negligible because of the coarse grain of each task.

Allpairs creates 13573 threads. Allpairs is another tree algorithm in which the value each thread is dependent on the value of its children. Thus in the single processor case all but one thread is absorbed and even in the eight processor case 11204 threads or 83% are absorbed.Allpairs is even finer grained (even though it creates fewer threads) than abisort and consequently more starvation occurs. The number of times a processor goes idle increases much more dramatically than in abisort. This explains why the

1. Allpairs is another of Mohr's benchmarks and again the description of the algorithm is largely taken from [Moh91].

efficiency goes from 84% on two processors, to 55% on four processors, and down to 21% on eight processors.

Allpairs also uses more execution contexts than abisort. This is because more blocking occurs. Even so, only 174 contexts were created for 13573 threads. And again as more processors are added to the computation, more execution contexts are created, fewer threads are absorbed, and the number of contexts allocated and re-used both increase.

Threads	**VP_0**	**VP_1**	**Total**		**VP_0**	**VP_1**	**VP_2**	**Total**
Created	1	13572	13573		1	6777	6795	13573
Delayed	0	0	0		0	0	0	0
Scheduled	1	13572	13573		1	6777	6795	13573
Absorbed	0	13572	13572		0	6705	6667	13372
Blocked	0	0	0		0	61	89	150
Resumed	0	0	0		0	61	89	150
Determined	0	13573	13573		0	6832	6741	13573
Wait	0	0	0		0	61	89	150
Resume Wait	0	0	0		0	89	61	150
Block on Group	0	0	0		0	0	0	0
Resume Group	0	0	0		0	0	0	0
Suspended	0	0	0		0	0	0	0
Sleep	0	0	0		0	0	0	0
Terminated	0	0	0		0	0	0	0
Idle	0	0	0		0	3	1	4
Migrated	0	0	0		0	0	0	0
Execution Contexts								
Created	0	1	1		0	2	3	5
Allocated	0	0	0		0	124	73	197
Reused	0	0	0		0	5	1	6
Mutexes								
Created	1	13576	13577		1	6785	6807	13601
Acquired	2	67868	67869		2	35074	34790	69875
Released	2	67868	67869		2	35074	34790	69875

Table 6-e : Allpairs with 1 and 2 Processors

Threads	VP_0	VP_1	VP_2	VP_3	VP_4	Total
Created	1	3412	3347	3395	3418	13573
Delayed	0	0	0	0	0	0
Scheduled	1	3412	3347	3395	3418	13573
Absorbed	0	3140	3095	3117	3121	12473
Blocked	0	250	231	256	264	1001
Resumed	0	250	231	256	264	1001
Determined	0	3427	3360	3396	3390	13573
Wait	0	250	231	256	264	1001
Resume Wait	0	258	259	249	235	1001
Block on Group	0	0	0	0	0	0
Resume Group	0	0	0	0	0	0
Suspended	0	0	0	0	0	0
Sleep	0	0	0	0	0	0
Terminated	0	0	0	0	0	0
Idle	0	263	297	328	294	1182
Migrated	0	0	0	0	0	0

Execution Contexts

Created	0	2	22	17	16	57
Allocated	0	218	228	227	230	903
Reused	0	50	45	59	47	201

Mutexes

Created	1	3420	3435	3471	3482	13825
Acquired	2	20193	19885	20111	20153	80361
Released	2	20193	19885	20111	20153	80361

Table 6-f : Allpairs with 4 Processors

Threads	**VP_0**	**VP_1**	**VP_2**	**VP_3**	**VP_4**	**VP_5**	**VP_6**	**VP_7**	**VP_8**	**Total**
Created	1	1601	1644	1646	1674	1706	1794	1744	1763	13573
Delayed	0	0	0	0	0	0	0	0	0	0
Scheduled	1	1601	1644	1646	1674	1706	1794	1744	1753	13573
Absorbed	0	1301	1389	1359	1360	1371	1494	1461	1469	11204
Blocked	0	244	217	230	244	261	229	246	240	1911
Resumed	0	243	217	229	244	260	229	246	240	1911
Determined	0	1594	1643	1601	1654	1646	1868	1777	1790	13573
Wait	0	244	217	230	244	261	229	246	240	1911
Resume Wait	0	224	211	204	235	221	292	260	264	1911
Block on Group	0	0	0	0	0	0	0	0	0	0
Resume Group	0	0	0	0	0	0	0	0	0	0
Suspended	0	0	0	0	0	0	0	0	0	0
Sleep	0	0	0	0	0	0	0	0	0	0
Terminated	0	0	0	0	0	0	0	0	0	0
Idle	0	381	480	470	390	400	337	407	378	3243
Migrated	0	0	0	0	0	0	0	0	0	0
Execution Contexts										
Created	0	21	42	55	12	26	12	3	3	174
Allocated	0	226	232	232	240	237	241	249	238	1895
Reused	0	50	67	65	65	54	49	64	68	482
Mutexes										
Created	1	1685	1812	1866	1722	1810	1842	1756	1775	14301
Acquired	2	11148	11328	11255	11580	11643	12481	12144	12156	93770
Released	2	11148	11328	11255	11580	11643	12481	12144	12156	93770

Table 6-g : Allpairs with 8 Processors

Threads	1	2	4	8
Created	13573	13573	13573	13573
Delayed	0	0	0	0
Scheduled	13573	13573	13573	13573
Absorbed	13572	13372	12473	11204
Blocked	0	150	1001	1911
Resumed	0	150	1001	1911
Determined	13573	13573	13573	13573
Wait	0	150	1001	1911
Resume Wait	0	150	1001	1911
Block on Group	0	0	0	0
Resume Group	0	0	0	0
Suspended	0	0	0	0
Sleep	0	0	0	0
Terminated	0	0	0	0
Idle	0	4	1182	3243
Migrated	0	0	0	0
Execution Contexts				
Created	1	5	57	174
Allocated	0	197	903	1895
Reused	0	6	201	482
Mutexes				
Created	13577	13601	13825	14301
Acquired	67869	69875	80361	93770
Released	67869	69875	80361	93770
Timing (secs)				
	9.95	5.93	4.55	5.82

Table 6-h : Allpairs on 1, 2, 4, and 8 Processors

6.5 Matrix Multiply

The matrix multiply benchmark does a simple matrix multiply of two 500 x 500 matrices. Each row of the result matrix is computed by a separate thread. There are no data dependencies between the threads, and thus, this benchmark is completely parallel. This benchmark was written as a master/worker program, with one master and 500 workers.

Matrix multiply creates 501 threads. Because there are no data dependencies there is no blocking or waiting done by any thread. The work load is well balanced across the various processors. The granularity of this program is reasonable and there is little contention on the global fifo queue. This can be see by the small number of times the virtual processors go idle. We suspect that the idle loops occur during the startup of the program when the master has generated little work.

Because there are no data dependencies and no blocking only one execution context is created on each virtual processor and all the threads on that processor use that context over and over again. So even though stealing is not possible the same execution context is used over and over.

The speedup is linear up to four processors. With eight processors the efficiency has fallen up to 86%. We believe efficiency has fallen off because the master cannot generate work fast enough for the workers.

Threads	VP_0	VP_1	Total	VP_0	VP_1	VP_2	Total
Created	1	500	501	1	500	0	501
Delayed	0	0	0	0	0	0	0
Scheduled	1	0	501	1	500	0	501
Absorbed	0	0	0	0	0	0	0
Blocked	0	0	0	0	0	0	0
Resumed	0	0	0	0	0	0	0
Determined	0	501	501	0	250	251	501
Wait	0	0	0	0	0	0	0
Resume Wait	0	0	0	0	0	0	0
Block on Group	0	0	0	0	0	0	0
Resume Group	0	0	0	0	0	0	0
Suspended	0	0	0	0	0	0	0
Sleep	0	0	0	0	0	0	0
Terminated	0	0	0	0	0	0	0
Idle	0	0	0	0	1	2	3
Migrated	0	0	0	0	0	0	0
Execution Contexts							
Created	0	1	1	0	1	1	2
Allocated	0	2	2	0	2	1	3
Reused	0	499	499	0	249	249	498
Mutexes							
Created	1	500	501	1	500	0	501
Acquired	0	1002	1002	0	501	501	1002
Released	0	1002	1002	0	501	501	1002

Table 6-i : Matrix Multiply with 1 and 2 Workers

Threads	**VP_0**	**VP_1**	**VP_2**	**VP_3**	**VP_4**	**Total**
Created	1	500	0	0	0	501
Delayed	0	0	0	0	0	0
Scheduled	1	500	0	0	0	501
Absorbed	0	0	0	0	0	0
Blocked	0	0	0	0	0	0
Resumed	0	0	0	0	0	0
Determined	0	126	125	125	125	501
Wait	0	0	0	0	0	0
Resume Wait	0	0	0	0	0	0
Block on Group	0	0	0	0	0	0
Resume Group	0	0	0	0	0	0
Suspended	0	0	0	0	0	0
Sleep	0	0	0	0	0	0
Terminated	0	0	0	0	0	0
Idle	0	1	2	2	2	7
Migrated	0	0	0	0	0	0
Execution Contexts						
Created	0	1	1	1	1	4
Allocated	0	2	1	1	1	5
Reused	0	124	124	124	124	496
Mutexes						
Created	1	500	0	0	0	501
Acquired	0	252	250	250	250	1002
Released	0	252	250	250	250	1002

Table 6-j : Matrix Multiply with 4 Workers

Threads	VP_0	VP_1	VP_2	VP_3	VP_4	VP_5	VP_6	VP_7	VP_8	Total
Created	1	0	0	0	500	0	0	0	0	501
Delayed	0	0	0	0	0	0	0	0	0	0
Scheduled	1	0	0	0	500	0	0	0	0	501
Absorbed	0	0	0	0	0	0	0	0	0	0
Blocked	0	0	0	0	0	0	0	0	0	0
Resumed	0	0	0	0	0	0	0	0	0	0
Determined	0	63	63	63	62	62	62	63	63	501
Wait	0	0	0	0	0	0	0	0	0	0
Resume Wait	0	0	0	0	0	0	0	0	0	0
Block on Group	0	0	0	0	0	0	0	0	0	0
Resume Group	0	0	0	0	0	0	0	0	0	0
Suspended	0	0	0	0	0	0	0	0	0	0
Sleep	0	0	0	0	0	0	0	0	0	0
Terminated	0	0	0	0	0	0	0	0	0	0
Idle		2	2	2	1	2	2	2	2	15
Migrated	0	0	0	0	0	0	0	0	0	0
Execution Contexts										
Created	0	1	1	1	1	1	1	1	1	8
Allocated	0	1	1	1	2	1	1	1	1	9
Reused	0	62	62	62	61	61	61	62	61	492
Mutexes										
Created	1	0	0	0	500	0	0	0	0	501
Acquired	0	126	126	126	124	124	124	126	126	1002
Released	0	126	126	126	124	124	124	126	126	1002

Table 6-k : Matrix Multiply with 8 Processors

Threads	**1**	**2**	**4**	**8**
Created	501	501	501	501
Delayed	0	0	0	0
Scheduled	501	501	501	501
Absorbed	0	0	0	0
Blocked	0	0	0	0
Resumed	0	0	0	0
Determined	501	501	501	501
Wait	0	0	0	0
Resume Wait	0	0	0	0
Block on Group	0	0	0	0
Resume Group	0	0	0	0
Suspended	0	0	0	0
Sleep	0	0	0	0
Terminated	0	0	0	0
Idle	0	3	7	15
Migrated	0	0	0	0
Execution Contexts				
Created	1	2	4	8
Allocated	2	3	5	9
Reused	499	498	496	492
Mutexes				
Created	501	501	501	501
Acquired	1002	1002	1002	1002
Released	1002	1002	1002	1002
Timing (secs)				
	247.0	123.5	62.0	35.7

Table 6-1 : Matrix Multiply with 1, 2, 4, and 8 Workers

6.6 N Queens with Threads

Queens finds all solutions to the n queens problem, where n queens are placed on an $n \times n$ chessboard in such a way that no queen may capture another. For these experiments a chessboard with $n = 14$ was chosen. In this formulation, a queen is placed on one row of the chessboard at a time. Each time a queen is legally placed, a thread is created to evaluate a recursive call to queens that finds all solutions stemming from the current configuration. Threads are not created after a specified cutoff depth in the tree is exceeded.

Bit vectors are used to build a compact representation, leading to fine thread granularity. Since there exist manifest data dependencies among threads in this example (a queen on one row needs to know the positions of queens on all other rows), many scheduled threads can be absorbed, limiting the overall storage requirements needed by this program.

The thread version of Queens was run on one, two, four, and eight processors with cutoff depths of one, two, three, and four respectively. As the cutoff depth increases more threads are spawned. Thus, in this benchmark as more processors are added more threads are created. The number of threads grows exponentially as the number of processors grows linearly.

Like abisort and allpairs, the value of every thread in the tree depends on that of its children, and thread absorption is possible. On one processor all threads except the root thread of the tree are absorbed and only one execution context is necessary, but on eight processors only 7510 out of 11167 threads are absorbed. This is because the amount of blocking increases as the number of processors increases. Even so, the number of execution contexts grows logarithmically in terms of the number of threads created, a very good property for memory conservation.

Using two and four processors the efficiency is 99% and with eight processors the efficiency is 90%. This fall off in efficiency is due to creating two much parallelism. If we use a depth cutoff of three on eight processors is takes 13.31 seconds to compute

queens of 14, an efficiency of 98%.

Queens shows an important aspect of *Sting* programs, namely that since threads are so cheap to create a reasonable amount of excessive parallelism has little or no effect on efficiency. This in turn means that a programmer has more flexibility in using *Sting* threads than those of other systems.

Threads	VP_0	VP_1	Total		VP_0	VP_1	VP_2	Total
Created	1	14	15		1	78	92	171
Delayed	0	0	0		0	0	0	0
Scheduled	1	14	15		1	78	92	171
Absorbed	0	14	14		0	18	20	38
Blocked	0	0	0		0	65	65	130
Resumed	0	0	0		0	64	66	130
Determined	0	15	15		0	83	88	171
Wait	0	0	0		0	65	65	130
Resume Wait	0	0	0		0	64	66	130
Block on Group	0	0	0		0	0	0	0
Resume Group	0	0	0		0	0	0	0
Suspended	0	0	0		0	0	0	0
Sleep	0	0	0		0	0	0	0
Terminated	0	0	0		0	0	0	0
Idle	0	1	1		0	3	1	4
Migrated	0	0	0		0	0	0	0
Execution Contexts								
Created			1					6
Allocated	0	1	1		0	65	66	131
Reused	0	0	0		0	1	1	2
Mutexes								
Created	1	14	15		9	78	92	171
Acquired	0	30	30		11	426	436	862
Released	0	30	30		11	426	436	862

Table 6-m : Thread Queens with 1 and 2 Processors

Threads	VP_0	VP_1	VP_2	VP_3	VP_4	**Total**
Created	1	286	384	406	458	1535
Delayed	0	0	0	0	0	0
Scheduled	1	286	384	406	458	1535
Absorbed	0	120	165	166	147	598
Blocked	0	182	186	192	205	765
Resumed	0	187	199	197	182	765
Determined	0	356	410	404	365	1535
Wait	0	182	186	192	205	765
Resume Wait	0	187	199	197	182	765
Block on Group	0	0	0	0	0	0
Resume Group	0	0	0	0	0	0
Suspended	0	0	0	0	0	0
Sleep	0	0	0	0	0	0
Terminated	0	0	0	0	0	0
Idle	0	6	3	1	6	16
Migrated	0	0	0	0	0	0
Execution Contexts						
Created						14
Allocated	0	182	187	193	205	767
Reused	0	48	45	42	35	170
Mutexes						
Created	1	286	384	406	458	1535
Acquired	0	1441	1564	1580	1550	6135
Released	0	1441	1564	1580	1550	6135

Table 6-n : Thread Queens with 4 Processors

Threads	VP_0	VP_1	VP_2	VP_3	VP_4	VP_5	VP_6	VP_7	VP_8	Total
Created	1	1255	1503	1192	1217	1384	1700	1325	1590	11167
Delayed	0	0	0	0	0	0	0	0	0	0
Scheduled	1	1255	1503	1192	1217	1384	1700	1325	1590	11167
Absorbed	0	809	1021	809	823	894	1169	816	1169	7510
Blocked	0	267	286	256	272	255	292	301	264	2193
Resumed	0	275	281	257	270	260	293	298	259	2193
Determined	0	1278	1467	1274	1261	1333	1636	1301	1617	11167
Wait	0	267	286	256	272	255	292	301	264	2193
Resume Wait	0	275	281	257	270	260	293	298	259	2193
Block on Group	0	0	0	0	0	0	0	0	0	0
Resume Group	0	0	0	0	0	0	0	0	0	0
Suspended	0	0	0	0	0	0	0	0	0	0
Sleep	0	0	0	0	0	0	0	0	0	0
Terminated	0	0	0	0	0	0	0	0	0	0
Idle	0	8	2	5	5	9	4	8	8	49
Migrated	0	0	0	0	0	0	0	0	0	0
Execution Contexts										
Created										26
Allocated	0	260	282	254	269	252	286	294	255	2152
Reused	0	191	165	214	177	190	183	194	191	1505
Mutexes										
Created	1	1255	1503	1192	4404	1384	1700	1325	1590	11167
Acquired	0	3623	4084	3583	21990	3701	4462	3825	4294	31194
Released	0	3623	4084	3583	21990	3701	4462	3825	4294	31194

Table 6-o : Thread Queens with 8 Processors

Threads	**1**	**2**	**4**	**8**
Created	15	171	1535	11167
Delayed	0	0	0	0
Scheduled	15	171	1535	11167
Absorbed	14	38	598	7510
Blocked	0	130	765	2193
Resumed	0	130	765	2193
Determined	15	171	1535	11167
Wait	0	130	765	2193
Resume Wait	0	130	765	2193
Block on Group	0	0	0	0
Resume Group	0	0	0	0
Suspended	0	0	0	0
Sleep	0	0	0	0
Terminated	0	0	0	0
Idle	1	4	16	49
Migrated	0	0	0	0
Execution Contexts				
Created	1	6	14	26
Allocated	1	131	767	2152
Reused	0	2	170	1505
Mutexes				
Created	15	171	1535	11167
Acquired	30	862	6135	31194
Released	30	862	6135	31194
Timing (secs)				
	104.74	52.43	26.42	14.54

Table 6-p : Thread Queens with 1, 2, 4, and 8 Processors

6.7 N Queens with Tuple Spaces

The tuple space version of **Queens** is similar to the thread version except that it uses first class tuple spaces for parallelism instead of threads. The first class tuple spaces are, however, implemented using *Sting*'s threads.

The tuple space version uses the same size chessboard as the thread version ($n = 14$). In this formulation, a queen is placed on one row of the chessboard at a time, similarly to the threads version. But each time a queen is legally placed, a new tuple-space is created to hold the value of an active tuple that is "spawned" to find all solutions stemming from the current configuration. Other threads access this value by "rd"-ing the tuple-space, and either absorb the thread if it has not started evaluating or block if it is evaluating but the value has not yet been computed.

Like the thread version, the tuple space version uses a depth cutoff to limit the number of threads created. The depth cutoffs used were the same as those used for the threads version. Both versions create the same number of threads. The tuple space version is 98% efficient on four processors and 88% efficient on eight processors. Thus, both versions show approximately the same efficiency.

This result is surprising since tuple spaces are implemented using threads and therefore should be slower, but a close look at the data explains it. The tuple space version absorbs between 105% and 425% more threads than the thread version. Because of this, it also blocks much less then the thread version. Consequently, the tuple space version creates many fewer execution contexts then the thread version. Finally, there is much less starvation in the tuple space version, where on eight processors a processor was idle only 21 times, than in the thread version, where on eight processors a processor was idle 49 times.

The fact, that tuple spaces are somewhat slower is the reason for its competitive performance. The tuple space version has threads with larger granularity and thus there are many more opportunities for absorbing other threads than in the thread version. The difference between these two benchmarks emphasizes how much faster thread

absorption is than blocking. It also shows that the granularity of threads is probably less important on *Sting* than on other systems.

Finally, we should point out that in spite of all the problems caused by the fine granularity of the thread version, it is just as fast as the tuple space version.

Threads	VP_0	VP_1	Total		VP_0	VP_1	VP_2	Total
Created	1	14	15		1	78	92	171
Delayed	0	0	0		0	0	0	0
Scheduled	1	14	15		1	78	92	171
Absorbed	0	14	14		0	77	85	162
Blocked	0	0	0		0	1	1	2
Resumed	0	0	0		0	0	2	2
Determined	0	15	15					
Wait	0	0	0		0	1	1	2
Resume Wait	0	0	0		0	0	2	2
Block on Group	0	0	0		0	0	0	0
Resume Group	0	0	0		0	0	0	0
Suspended	0	0	0		0	0	0	0
Sleep	0	0	0		0	0	0	0
Terminated	0	0	0		0	0	0	0
Idle	0	0	0		0	3	1	4
Migrated	0	0	0		0	0	0	0
Execution Contexts								
Created			1					3
Allocated	0	1	1		0	1	2	3
Reused	0	0	0		0	6	0	6
Mutexes								
Created	1	14	15		0	78	92	171
Acquired	0	30	30		0	171	179	350
Released	0	30	30		0	171	179	350

Table 6-q : Tuple Space Queens with 1 and 2 VPs

Threads	VP_0	VP_1	VP_2	VP_3	VP_4	**Total**
Created	1	435	356	346	397	1535
Delayed	0	0	0	0	0	0
Scheduled	1	435	356	346	397	1535
Absorbed	0	420	356	342	389	1507
Blocked	0	3	1	1	1	6
Resumed	0	2	1	0	3	6
Determined	0	425	366	348	396	1535
Wait	0	3	1	1	1	6
Resume Wait	0	2	1	0	3	6
Block on Group	0	0	0	0	0	0
Resume Group	0	0	0	0	0	0
Suspended	0	0	0	0	0	0
Sleep	0	0	0	0	0	0
Terminated	0	0	0	0	0	0
Idle	0	4	7	8	2	21
Migrated	0	0	0	0	0	0
Execution Contexts						
Created						6
Allocated	0	3	1	1	2	7
Reused	0	3	8	6	4	21
Mutexes						
Created	1	435	356	346	397	1535
Acquired	0	862	735	700	797	3094
Released	0	862	735	700	797	3094

Table 6-r : Tuple Space Queens with 4 Processors

Threads	VP_0	VP_1	VP_2	VP_3	VP_4	VP_5	VP_6	VP_7	VP_8	Total
Created	1	1442	1353	1236	1623	1588	1308	1302	1314	11167
Delayed	0	0	0	0	0	0	0	0	0	0
Scheduled	1	1442	1353	1236	1623	1588	1308	1302	1314	11167
Absorbed	0	1434	1348	1233	1603	1582	1308	1298	1308	11108
Blocked	0	3	1	1	2	2	2	2	1	14
Resumed	0	0	2	2	1	2	2	5	0	14
Determined	0	1441	1352	1243	1609	1592	1311	1310	1309	11167
Wait	0	3	1	1	2	2	2	2	1	14
Resume Wait	0	0	2	2	1	2	2	5	0	14
Block on Group	0	0	0	0	0	0	0	0	0	0
Resume Group	0	0	0	0	0	0	0	0	0	0
Suspended	0	0	0	0	0	0	0	0	0	0
Sleep	0	0	0	0	0	0	0	0	0	0
Terminated	0	0	0	0	0	0	0	0	0	0
Idle	0	8	7	6	4	6	7	9	11	21
Migrated	0	0	0	0	0	0	0	0	0	0
Execution Contexts										
Created										11
Allocated	0	2	2	2	2	3	3	3	1	18
Reused	0	5	3	9	4	8	2	9	1	41
Mutexes										
Created	1	1442	1353	1236	1623	1588	1308	1302	1314	11167
Acquired	0	2891	2710	2492	3225	3193	2626	2631	2622	22390
Released	0	2891	2710	2492	3225	3193	2626	2631	2622	22390

Table 6-s : Tuple Space Queens with 8 Processors

Threads	**1**	**2**	**4**	**8**
Created	15	171	1535	11167
Delayed	0	0	0	0
Scheduled	15	171	1535	11167
Absorbed	14	162	1507	11108
Blocked	0	2	6	14
Resumed	0	2	6	14
Determined	15	171	1535	11167
Wait	0	2	6	14
Resume Wait	0	2	6	14
Block on Group	0	0	0	0
Resume Group	0	0	0	0
Suspended	0	0	0	0
Sleep	0	0	0	0
Terminated	0	0	0	0
Idle	0	4	21	21
Migrated	0	0	0	0
Execution Contexts				
Created	1	3	6	11
Allocated	0	3	7	18
Reused	0	6	21	41
Mutexes				
Created	15	171	1535	11167
Acquired	30	350	3094	22390
Released	30	350	3094	22390
Timing (secs)				
	103.8	52.6	26.4	14.7

Table 6-t : Tuple Space Queens with 1, 2, 4, and 8 Processors

6.8 N Body Problem

The *N*-body problem is the problem of simulating the evolution of a system of bodies under the influence of gravitational forces. Each body is modeled as a point mass and exerts forces on all other bodies in the system. The simulation proceeds over time-steps, each step computing the net force on every body and thereby updating that body's position and other attributes. If all pair wise forces are computed directly, the algorithm has a time complexity of $O(n^2)$.

The Barnes-Hut algorithm for the N-body problem has an expected running time of O(n log n) if the bodies are uniformly distributed in space. It exploits the idea that the effect of a cluster of particles at a distant point can be approximated by one body whose location is the center of mass of the cluster and whose mass is the sum of the masses in the cluster. It thus assumes that all bodies are contained in a fixed sized cube (in the 3-dimensional case) or in a square (in the 2-dimensional case).

We implemented the two-dimensional version of the problem. To partition the plane the Barnes-Hut algorithm recursively splits the original square into four equally sized quadrants until each quadrant contains only 1 body. This partition can be represented by a tree, called a BH-tree. Each node represents a square **S** in the partition and it is the parent of the squares that are created by splitting **S**.Thus, each node has at most four children and the expected height of the BH-tree is O(log n) if the bodies are uniformly distributed in the space. At each node, we store the center of mass and the total mass of the nodes in its square.

To compute the force that is exerted on a body **B**, we traverse the BH-tree in the following way: We start at the root of the tree. If the length of the square of the visited node **x** is larger than the distance of **B** to the center of mass of **x**, we compute the force that the mass stored at **x** exerts on **B**. Otherwise we recur on all children of **x**. Thus whether a subtree of the BH-tree is visited depends on the location of **B**. In each iteration, a new BH-tree is built and the new location of all bodies are computed.

This program generates a number of relatively coarse-grained threads independent of

the actual input. Each thread has access to a distributed data structure (i.e. a tuple space) that holds the most recent BH-tree. Each thread is expected to access the distributed data structure O(log n) times for each body and iteration. Barrier synchronization is used to ensure that all forces are computed before a new iteration is initiated. To optimize storage usage, a new set of threads is created on every iteration; storage generated during one iteration becomes available to threads created in subsequent ones; the overall aggregate storage requirements are thus minimized.

Table 6-u show the results of running the program using 3500 bodies, for six iterations with a velocity of one using on one, two, four, and eight processors. The efficiency is 100%, 75%, and 66% on two, four, and eight processors respectively. The reason for the decrease in efficiency as more processors are added is that the amount of parallelism in the problem is limited. This can be seen in the both the amount of blocking being done combined with the number of times processors are idle.

Threads	1	2	4	8
Created	7	13	25	49
Delayed	0	0	0	0
Scheduled	7	13	25	49
Absorbed	0	0	0	0
Blocked	6	3822	11395	25062
Resumed	6	3822	11395	25062
Determined	7	13	25	49
Wait	0	0	0	0
Resume Wait	0	0	0	0
Block on Group	0	0	0	0
Resume Group	0	0	0	0
Suspended	0	0	0	0
Sleep	0	0	0	0
Terminated	0	0	0	0
Idle	1	3830	12714	49119
Migrated	0	0	0	0
Execution Contexts				
Created				
Allocated	7	13	25	49
Reused	0	0	0	0
Mutexes				
Created	105	3929	10406	12727
Acquired	1556091	1579672	1630048	1730415
Released	1556091	1579672	1630048	1730415
Timing (secs)				
	751	363	251	142

Table 6-u : *N* Body with 1, 2, 4, and 8 Processors

6.9 Thread Policy Management

The final benchmark compares six different thread policy managers. We do this by running the abisort benchmark (see Section 6.3) using the different policy managers. The program run is exactly the same for each policy manager, i.e. the source is not changed and it is not recompiled. The following thread policy managers were tested:

GLIFO - A policy manager that uses one global queue for scheduling all the threads on virtual processors. The queue is organized in a last in first out manner.

GFIFO - A policy manager that uses one global queue for scheduling all the threads on virtual processors. The queue is organized in a first in first out manner.

L1 LIFO Random - A policy manager that uses one local ready queue on each virtual processor. The queue is organized in a last in first out manner. The initial mapping of thread to virtual processor is done by picking a processor at random.

L1 LIFO Round Robin- A policy manager that uses one local ready queue on each virtual processor. The queue is organized in a last in first out manner. The initial mapping of thread to virtual processor in a round robin manner using a global counter access to which is synchronized.

L1 FIFO Random - A policy manager that uses one local ready queue on each virtual processor. The queue is organized in a first in first out manner. The initial mapping of thread to virtual processor is done by picking a processor at random.

L1 LIFO Round Robin - A policy manager that uses one local ready queue on each virtual processor. The queue is organized in a last in first out manner. The initial mapping of thread to virtual processor in a round robin manner using a global counter access to which is synchronized.

Table 6-v through Table 6-aa show the results for each of the eight processors running abisort under the various policy managers. An examination of the tables show the each of the policy managers does a good job of scheduling threads evenly across the processors with the variance in load under 5% for all of them. The round robin schedulers distribute the threads with essentially no variance. This even distribution comes at the cost of acquiring an extra lock for each thread scheduled, however.

All of the policy managers show good balance in other respects. The the number of threads absorbed by each processor is well balanced as is the amount of blocking and the number of times a processor goes idle.

Table 6-ab show a comparison of the results for the six systems tested. This table shows that for abisort a fifo strategy works better than the lifo strategy and that the global queue works much better than the local queues. The thread policy manager that works best, **GFIFO,** as we might expect, has less idle time, less blocking, and creates fewer execution contexts than any of the others. The one that performs worst, **L1 LIFO Random**, not surprisingly has much more blocking, more idle time, and creates more execution contexts than the others.

The machine we ran these benchmarks on is a physically shared memory machine and the results of these policy managers show less variance than they might on a physically disjoint memory machine. Abisort is only one benchmark; to understand the behavior of various thread policy managers many more benchmarks need to be studied. Finally, there are many other thread policy managers that we would like to test in the future.

Threads	**VP_0**	**VP_1**	**VP_2**	**VP_3**	**VP_4**	**VP_5**	**VP_6**	**VP_7**	**VP_8**	**Total**
Created	1	13274	13449	12997	13268	13385	13525	13251	13347	106497
Delayed	0	0	0	0	0	0	0	0	0	0
Scheduled	1	13274	13449	12997	13268	13385	13525	13251	13347	106497
Absorbed	0	10889	11102	10600	10837	10895	11076	10938	10959	87296
Blocked	0	1472	1526	1547	1506	1514	1576	1491	1546	12178
Resumed	0	1500	1570	1568	1533	1491	1495	1520	1501	12178
Determined	0	13191	13527	13123	13209	13372	13443	13357	13275	106497
Wait	0	1472	1526	1547	1506	1514	1576	1491	1546	12178
Resume Wait	0	1500	1570	1568	1533	1491	1495	1520	1501	12178
Block on Group	0	0	0	0	0	0	0	0	0	0
Resume Group	0	0	0	0	0	0	0	0	0	0
Suspended	0	0	0	0	0	0	0	0	0	0
Sleep	0	0	0	0	0	0	0	0	0	0
Terminated	0	0	0	0	0	0	0	0	0	0
Idle	0	30	33	33	22	35	27	21	28	229
Migrated	0	0	0	0	0	0	0	0	0	0
Execution Contexts										
Created										50
Allocated	0	1164	1206	1226	1207	1177	1286	1187	1201	9654
Reused	0	1181	1191	1180	1214	1148	1191	1204	1238	9547
Mutexes										
Created	1	13274	13449	12997	13268	13385	13525	13251	13347	106497
Acquired	0	33351	34073	33363	33524	33540	34156	33611	33729	269347
Released	0	33351	34073	33363	33524	33540	34156	33611	33729	269347

Table 6-v : Abisort with Global LIFO Policy

Threads	VP_0	VP_1	VP_2	VP_3	VP_4	VP_5	VP_6	VP_7	VP_8	Total
Created	1	13541	13370	13167	13510	13220	13155	13198	13335	106497
Delayed	0	0	0	0	0	0	0	0	0	0
Scheduled	1	13541	13370	13167	13510	13220	13155	13198	13335	106497
Absorbed	0	13512	13335	13128	13468	13189	13127	13171	13288	106218
Blocked	0	29	30	27	40	34	30	25	36	251
Resumed	0	50	26	21	25	39	35	27	28	251
Determined	0	13572	13364	13148	13493	13244	13161	13200	13315	106497
Wait	0	29	30	27	40	34	30	25	36	251
Resume Wait	0	50	26	21	25	39	35	27	28	251
Block on Group	0	0	0	0	0	0	0	0	0	0
Resume Group	0	0	0	0	0	0	0	0	0	0
Suspended	0	0	0	0	0	0	0	0	0	0
Sleep	0	0	0	0	0	0	0	0	0	0
Terminated	0	0	0	0	0	0	0	0	0	0
Idle	0	21	25	25	22	19	18	27	25	182
Migrated	0	0	0	0	0	0	0	0	0	0
Execution Contexts										
Created										38
Allocated	0	18	21	21	22	20	17	19	21	159
Reused	0	20	12	13	14	13	18	15	15	120
Mutexes										
Created	1	13541	13370	13167	13510	13220	13155	13198	13335	106497
Acquired	0	27259	26849	26412	27143	26607	26448	26507	26775	214000
Released	0	27259	26849	26412	27143	26607	26448	26507	26775	214000

Table 6-w : Abisort with Global FIFO Policy

Threads	**VP_0**	**VP_1**	**VP_2**	**VP_3**	**VP_4**	**VP_5**	**VP_6**	**VP_7**	**VP_8**	**Total**
Created	1	13332	13322	13258	13191	13309	13332	13440	13312	106497
Delayed	0	0	0	0	0	0	0	0	0	0
Scheduled	1	13318	13400	13233	13096	13297	13504	13023	13254	106497
Absorbed	0	12835	12800	12709	12727	12867	12845	13023	12745	102551
Blocked	0	397	405	401	384	411	433	365	415	3211
Resumed	0	371	359	420	417	385	432	402	425	3211
Determined	0	13277	13234	13287	13264	13351	13381	13479	13224	106497
Wait	0	397	405	401	384	411	433	365	415	3211
Resume Wait	0	371	359	420	417	385	432	402	425	3211
Block on Group	0	0	0	0	0	0	0	0	0	0
Resume Group	0	0	0	0	0	0	0	0	0	0
Suspended	0	0	0	0	0	0	0	0	0	0
Sleep	0	0	0	0	0	0	0	0	0	0
Terminated	0	0	0	0	0	0	0	0	0	0
Idle	0	170	115	164	191	175	120	144	122	1201
Migrated	0	0	0	0	0	0	0	0	0	0
Execution Contexts										
Created										52
Allocated	0	419	400	411	408	432	445	378	410	3303
Reused	0	80	95	89	82	82	76	64	75	643
Mutexes										
Created	1	13332	13322	13258	13191	13309	13332	13440	13312	106497
Acquired	0	28265	28233	28238	28167	28450	28563	28538	28204	226658
Released	0	28265	28233	28238	28167	28450	28563	28538	28204	226658

Table 6-x : Abisort with Local LIFO and Random

Threads	VP_0	VP_1	VP_2	VP_3	VP_4	VP_5	VP_6	VP_7	VP_8	Total
Created	1	13227	13259	13396	13240	13499	13332	13347	13196	106497
Delayed	0	0	0	0	0	0	0	0	0	0
Scheduled	1	13313	13312	13312	13312	13312	13312	13312	13312	106497
Absorbed	0	10889	11102	10600	10837	10895	11076	10938	10959	87296
Blocked	0	251	254	258	237	256	243	237	235	1971
Resumed	0	244	247	267	226	272	244	238	233	1971
Determined	0	13163	13339	13410	13184	13546	13413	13326	13116	106497
Wait	0	251	254	258	237	256	243	237	235	1971
Resume Wait	0	244	247	267	226	272	244	238	233	1971
Block on Group	0	0	0	0	0	0	0	0	0	0
Resume Group	0	0	0	0	0	0	0	0	0	0
Suspended	0	0	0	0	0	0	0	0	0	0
Sleep	0	0	0	0	0	0	0	0	0	0
Terminated	0	0	0	0	0	0	0	0	0	0
Idle	0	80	69	75	42	59	85	57	51	518
Migrated	0	0	0	0	0	0	0	0	0	0
Execution Contexts										
Created										36
Allocated	0	257	254	264	237	265	250	246	237	2010
Reused	0	55	51	52	44	46	68	45	37	398
Mutexes										
Created	1	13227	13259	13396	13240	13499	13332	13347	13196	106497
Acquired	0	27386	27696	27906	27350	28088	27810	27647	27228	221111
Released	0	27386	27696	27906	27350	28088	27810	27647	27228	221111

Table 6-y : Abisort with Local LIFO and Round Robin

Threads	VP_0	VP_1	VP_2	VP_3	VP_4	VP_5	VP_6	VP_7	VP_8	**Total**
Created	1	13326	13398	13370	13342	13233	13344	13371	13311	106497
Delayed	0	0	0	0	0	0	0	0	0	0
Scheduled	1	13302	13388	13229	13106	13305	13512	13413	13112	106497
Absorbed	0	13260	13272	13275	13257	13117	13253	13292	13017	105743
Blocked	0	81	80	78	72	82	77	75	63	608
Resumed	0	78	91	97	66	78	56	91	51	608
Determined	0	13366	13385	13392	13314	13217	13322	13425	13077	106497
Wait	0	81	80	78	72	82	77	75	63	608
Resume Wait	0	78	91	97	66	78	56	91	51	608
Block on Group	0	0	0	0	0	0	0	0	0	0
Resume Group	0	0	0	0	0	0	0	0	0	0
Suspended	0	0	0	0	0	0	0	0	0	0
Sleep	0	0	0	0	0	0	0	0	0	0
Terminated	0	0	0	0	0	0	0	0	0	0
Idle	0	43	27	16	35	19	24	27	15	206
Migrated	0	0	0	0	0	0	0	0	0	0
Execution Contexts										
Created										40
Allocated	0	61	57	54	51	54	58	44	43	422
Reused	0	45	40	40	41	48	32	54	33	333
Mutexes										
Created	1	13326	13398	13370	13342	13233	13344	13371	13311	106497
Acquired	0	27059	27094	27092	26953	26764	26959	27133	26412	215466
Released	0	27059	27094	27092	26953	26764	26959	27133	26412	215466

Table 6-z : Abisort with Local FIFO and Random

Threads	VP_0	VP_1	VP_2	VP_3	VP_4	VP_5	VP_6	VP_7	VP_8	Total
Created	1	13340	13444	13244	13238	13269	13304	13375	13282	106497
Delayed	0	0	0	0	0	0	0	0	0	0
Scheduled	1	13313	13312	13312	13312	13312	13312	13312	13312	106497
Absorbed	0	13264	13391	13164	13140	13193	13262	13298	13236	105948
Blocked	0	48	61	56	48	42	57	52	56	420
Resumed	0	28	59	47	49	64	64	50	59	420
Determined	0	13296	13458	13211	13198	13260	13366	13381	13327	106497
Wait	0	48	61	56	48	42	57	52	56	420
Resume Wait	0	28	59	47	49	64	64	50	59	420
Block on Group	0	0	0	0	0	0	0	0	0	0
Resume Group	0	0	0	0	0	0	0	0	0	0
Suspended	0	0	0	0	0	0	0	0	0	0
Sleep	0	0	0	0	0	0	0	0	0	0
Terminated	0	0	0	0	0	0	0	0	0	0
Idle	0	22	11	16	23	21	13	22	21	149
Migrated	0	0	0	0	0	0	0	0	0	0
Execution Contexts										
Created										38
Allocated	0	40	42	39	35	29	43	43	41	306
Reused	0	22	31	28	26	30	30	30	34	243
Mutexes										
Created	1	13340	13444	13244	13238	13269	13304	13375	13282	106497
Acquired	0	26798	27166	26659	26594	26707	26942	26962	26867	214695
Released	0	26798	27166	26659	26594	26707	26942	26962	26867	214695

Table 6-aa : Abisort with Local FIFO and Round Robin

Threads	**G LIFO**	**G FIFO**	**L LIFO Random**	**L LIFO RR**	**L FIFO Random**	**L FIFO RR**
Created	106497	106497	106497	106497	106497	106497
Delayed	0	0	0	0	0	0
Scheduled	106497	106497	106497	106497	106497	106497
Absorbed	87296	106218	102551	87296	105743	105948
Blocked	12178	251	3211	1971	608	420
Resumed	12178	251	3211	1971	608	420
Determined	106497	106497	106497	106497	106497	106497
Wait	12178	251	3211	1971	608	420
Resume Wait	12178	251	3211	1971	608	420
Block on Group	0	0	0	0	0	0
Resume Group	0	0	0	0	0	0
Suspended	0	0	0	0	0	0
Sleep	0	0	0	0	0	0
Terminated	0	0	0	0	0	0
Idle	229	182	1201	518	206	149
Migrated	0	0	0	0	0	0
Execution Contexts						
Created	50	38	52	36	40	38
Allocated	9654	159	3303	2010	422	306
Reused	9547	120	643	398	333	243
Mutexes						
Created	106497	106497	106497	106497	106497	106497
Acquired	269347	214000	226658	221111	215466	214695
Released	269347	214000	226658	221111	215466	214695
Timing (secs)						
	14.77	11.40	16.94	14.76	14.14	14.68

Table 6-ab : Abisort with Various TPMs

Chapter 7

Concluding Remarks

The goal of the *Sting* design has been to provide a foundation for modern programming languages, particularly parallel languages, that is powerful while at the same time being both simple and efficient. We believe that these goals have largely been met. The architecture is composed of a small set of simple concepts that compose well. The implementation is also small comprising less than 7,000 lines of scheme code.

Sting has proven to be a useful foundation for designing and implementing parallel programming constructs. In addition to threads as a programming paradigm, futures, first class tuple spaces, engines[HF84], and speculative constructs have been implemented.

Continuations have proven to be a powerful and elegant way of handling complex control issues in the operating systems. They have simplified the implementation of context switching, interrupts, and exceptions. It was the use of continuations that first made us realize it would be possible to eliminate kernel threads altogether.

The goal of separating control from policy lead to the discovery of small and simple interfaces to the virtual processor policy manager and the thread policy manager. This in turn has made policy customization not only straightforward, but also easy. *Sting*'s flexibility of policy management makes it unique among current operating systems.

The implementation of first class tuple spaces by Suresh Jagannathan of NEC Research Institute [Jag91] has demonstrated the usefulness of *Sting*. His system is

interesting in several respects, but perhaps the most interesting consequence of using *Sting* is that tuples can contain not only immediate data, but also reference data such as threads, procedures, and objects. In fact, any data in the system can be placed in a tuple. This not only leads to enhanced expressiveness in his system but also to increased efficiency, since data can be communicated by reference rather than by copying. The tuple space implementation is between 1000 and 1500 lines of code.

Finally, *Sting* has demonstrated that with sufficiently lightweight threads parallel programs can have a significant measure of excess parallelism and still perform efficiently. This allows the parallel programmer or language designer greater freedom and flexibility.

7.1 Future Research

We believe that *Sting* provides a rich foundation for future research. Below is a list some of the projects we envision for the future.

Sting is currently implemented on top of Unix. The next step is to move it to bare hardware. Much of the abstract physical machine has been implemented, but it is impossible to completely test the effectiveness of the design without removing the intervening layer of Unix.

Sting was designed as a vehicle for implementing and testing new paradigms of parallelism. We have implemented several of the well known paradigms, but there are many others to explore. *Sting* provides a "flat" playing field on which to compare the relative efficiencies of competing paradigms and implementations. There is much useful work to be done in this area.

The *Sting* memory management system is quite novel and there remains a large number of experiments that should be performed to determine the effectiveness of the design. For example: What are the number and cost of inter-area references? What are the number and percentage of intra-stack references, intra-private-heap references, intra-shared-heap references? What is the mean reference distance, i.e. how much

locality is exploited? What percentage of objects that are typically allocated in the heap can be allocated in the stack? What are the number of loads and stores on shared objects as compared to private objects? There are many other questions of a similar nature that would be useful to answer. There are also other possible design choices in the memory management system, e.g. eliminating private heaps or having shared heaps on a per virtual processor basis rather than on a thread group basis. These design modifications should be directed by the answers to the questions above.

The memory management system was also designed with a eye toward using it to implement a persistent object store bases on persistent shared virtual memory. In order to do this we envision adding first class environments to *Sting*. These environments would fill the same function as directories in file systems. With the advent of 64 bit addresses it is possible to map a large persistent object space into many different virtual machines. The object space would reside at a fixed location in each virtual memory and thus object IDs would simply be references with no extra level of indirection or swizzling necessary. The generational nature of *Sting*'s garbage collector combined with the fact that different areas can be collected independently make it ideally suited for a persistent object store.

Sting's shared virtual memory model allows the implementation of "polymorphic ports." These are ports to which any object can be sent or received without defining a type for the port. This is possible because all references are consistent in the shared virtual memory. This is not the case in disjoint address space architectures such as the hypercube. Polymorphic ports allow the transmission of "active messages." Active messages are represented as thunks. Transmitting an active message involves simply sending the closure of the thunk to a port. The receiver of the message "decodes" the message simply by invoking the thunk. Active messages can be thought of as polymorphic chunks of work that the sender requests the receiver to perform. We believe this paradigm might be very useful for structuring parallel algorithms and should be explored further.

There are several enhancements to the system we would like to make. Two important

ones are adding first class continuations and implementing lazy task creation. These should be straightforward to implement given the current design. It would be extremely interesting to compare the performance of thread absorption versus lazy task creation across a broad spectrum of programs.

Finally, the *Sting* programming environment is extremely primitive; many useful enhancements in terms of profiling/metering, debugging, and program visualization could be added.

References

[ABHKK90] A. Agarwal, L. Beng-Hong, D. Kranz, and J Kubiatowicz. APRIL: A processor architecture for multiprocessing. In *Proc. of the 17th Annual Symposium on Computer Architecture*, pages 104–114, New York NY (USA), May 1990. IEEE.

[ABLL91] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler activations: effective kernel support for the user-level management of parallelism. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 95–109. Association for Computing Machinery SIGOPS, October 1991.

[Agh86a] Gul Agha. *ACTORS A Model of Concurrent Computation in Distributed Systems.* MIT Press, 1986.

[Agh86b] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, Mass., 1986.

[ALL89] Thomas Anderson, Edward Lazowska, and Henry Levy. The Performance Implications of Thread Management Alternatives for Shared Memory MultiProcessors. *IEEE Transactions on Computers*, 38(12):1631–1644, December 1989.

[ANP89] Arvind, Rishiyur Nikhil, and Keshav Pingali. I-Structures: Data Structures for Parallel Computing. *Transactions on Programming Languages and Systems*, 11(4):598–632, October 1989.

[App90] Andrew Appel. A Runtime System. *Journal of Lisp and Symbolic Computation*, 3(4):343–380, November 1990.

[AS85] Harold Abelson and Gerald Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, Cambridge, Mass., 1985.

[BCZ89a] John K. Bennett, John B. Carter, and Willy Zwaenepoel. Munin: Shared memory for distributed memory multiproces-

sors. Technical Report Rice COMP TR89-91, Rice University, April 1989.

[BCZ89b] John K. Bennett, John B. Carter, and Willy Zwaenepoel. Shared memory access characteristics. Technical Report COMP TR89-99, Rice University, September 1989. Submitted to ISCA 90.

[BGHL87] A.D. Birrell, J.V. Guttag, J.J. Horning, and R. Levi. Synchronization Primitives for a Multiprocessor: A Formal Specification. In *Proceedings of the 11th Symposium on Operating Systems Principles*, pages 94–102, November 1987.

[BGJ+92] David L. Black, David B. Golub, Daniel P. Julin, Richard Rashid, Richard P. Draves, Randall W. Dean, Alessandro Forin, Joseph Barrera, Hideyuki Tokuda, Gerald Malan, and David Bohman. Microkernel Operating System Architecture and Mach. In *Workshop Proceedings Micro-Kenels and other Kernel Architectures*, pages 11–30, April 1992.

[BHK+91] Mary G. Baker, John H. Hartman, Michael D. Kupfer, Ken W. Shirriff, and John K. Ousterhout. Measurements of a distributed file system. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 198–212. Association for Computing Machinery SIGOPS, October 1991.

[Bir89] Andrew D. Birrell. An Introduction to Programming with Threads. Technical Report 35, DEC Systems Research Center, 1989.

[Bis77] Peter Bishop. *Computer Systems with a Very Large Address Space and Garbage Collection.* PhD thesis, MIT Laboratory for Computer Science, 1977.

[Bla90] David Black. Scheduling Support for Concurrency and Parallelism in the Mach Operating System. *IEEE Computer*, 23(5):35–43, May 1990.

[BN84] A. Birrel and B. Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems*, 2(1):39 –59, February 1984.

[BR90] R. Bisiani and M. Ravishankar. PLUS: A distributed shared-memory system. In *Proc. of the 17th Annual Symposium on Computer Architecture*, pages 115–125, New York, May 1990. IEEE.

[BT88] Henri E. Bal and Andrew S. Tanenbaum. Distributed programming with shared data. Technical Report IR-149, Vrije Universiteit, March 1988.

[BW73] D. G. Bobrow and Ben Wegbreit. A Model and Stack Implementation of Multiple Environments. *Communications of the ACM*, 16(10):591–603, October 1973.

[CBZ91] John B. Carter, John K. Bennett, and Willy Zwaenepoel. Implementation and performance of Munin. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 152–64. Association for Computing Machinery SIGOPS, October 1991.

[CD88] Eric Cooper and Richard Draves. C Threads. Technical Report CMU-CS-88-154, Carnegie-Mellon University, June 1988.

[CG86] K.L Clark and S. Gregory. PARLOG: Parallel Programming in Logic. *ACM Transactions on Programming Languages and Systems*, 8(1):1–49, 1986.

[CG89] Nick Carriero and David Gelernter. Linda in Context. *Communications of the ACM*, 32(4):444 – 458, April 1989.

[CG90] Nick Carriero and David Gelernter. *How to Write Parallel Programs: A Guide to the Perplexed.* MIT Press, 1990.

[CM90] Eric Cooper and J.Gregory Morrisett. Adding Threads to Standard ML. Technical Report CMU-CS-90-186, Carnegie-Mellon University, 1990.

[CM91] John Zahorjan Cathy McCann, Raj Vaswani. A dynamic processor allocation policy for multiprogrammed shared memory multiprocessors. Technical report, University of Washington, Seattle, WA, February 1991.

[DBRD91] Richard P. Draves, Brian N. Bershad, Richard F. Rashid, and Randall W. Dean. Using continuations to implement thread management and communication in operating systems. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 122–36. Association for Computing Machinery SIGOPS, October 1991.

[DH88] R. Kent Dybvig and Robert Hieb. Engines from Continuations. Technical Report Computer Science Technical Report No. 254, Indiana University, 1988.

[DH90] R. Kent Dybvig and Robert Hieb. Representing Control in the Presence of First-Class Continuations. In *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, pages 66–77, 1990.

[DMKJ91] M.D. Durand, T. Montaut, L. Kervella, and W. Jalby. Modeling the impact of memory contention on dynamic scheduling. Technical report, IRISA, 1991.

[DSB88] J. Dongarra, D. Sorenson, and P. Brewer. Tools and Methodology for Programming Parallel Processors. In *Aspects of Computation on Asynchronous Processors*, pages 125–138. North-Holland, 1988.

[ea85] William Clinger et. al. The Revised Revised Revised Report on Scheme or An UnCommon Lisp. Technical Report AI-TM 848, MIT Artificial Intelligence Laboratory, 1985.

[FM90] Mark Feeley and James Miller. A Parallel Virtual Machine for Efficient Scheme Compilation. In *Proceedings of the 1990 Conf. on Lisp and Functional Programming*, pages 119–131, 1990.

[FP89] Brett D. Fleisch and Gerald J. Popek. Mirage: A coherent distributed shared memory design. *Proceedings of the 12th Symposium on Operating System Principles*, pages 211–222, December 1989.

[FR90] C. Fricker and P. Robert. On the representation of memory references generated by a program with application to the analysis of cache memories. Technical Report 1158, INRIA, Domaine de Voluceau, Rocquencourt B.P. 105 78153 Le Chesnay Cédex France, February 1990.

[GA91] Ramesh Govindan and David P. Anderson. Scheduling and IPC mechanisms for continuous media. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 68–80. Association for Computing Machinery SIGOPS, October 1991.

[GJSJ91] David K. Gifford, Pierre Jouvelot, Mark A. Sheldon, and James W. O'Toole Jr. Semantic file systems. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 16–25. Association for Computing Machinery SIGOPS, October 1991.

[GM84a] R. Gabriel and J. McCarthy. Queue-Based Multi-Processing Lisp. In *Proceedings of the 1984 Conf. on Lisp and Functional Programming*, pages 25–44, August 1984.

[GM84b] R. Gabriel and J. McCarthy. Queue-Based Multi-Processing Lisp. In *Proceedings of the 1984 Conf. on Lisp and Functional Programming*, pages 25–44, August 1984.

[Gre91] David Saks Greenberg. Full utilization of communication resources. Technical report, Yale University, June 1991.

[Hal85] Robert Halstead. Multilisp: A Language for Concurrent Symbolic Computation. *Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[HCD89] Waldemar Horwat, Andrew Chien, and William Dally. Experience with CST: Programming and Implementation. In *ACM*

SIGPLAN '89 Conference on Programming Language Design and Implementation, pages 101–109, June 1989.

[HD90] Robert Hieb and R. Kent Dybvig. Continuations and Concurrency. In *Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 128–137, March 1990.

[HF84] Christopher T. Haynes and Danial P. Friedman. Engines Build Process Abstractions. In *Proceedings of the 1984 ACM Lisp and Functional Programming Conference*, pages 18–24, 1984.

[HFW84] Christopher T. Haynes, Danial P. Friedman, and Mitchell Wand. Continuations and Coroutines. In *Proceedings of the 1984 ACM Lisp and Functional Programming Conference*, pages 293–298, 1984.

[HL] Chris Hanson and John Lamping. Dynamic Binding in Scheme. Unpublished Paper.

[Hoa74] C. A. R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, pages 549–557, October 1974.

[Hud86] Paul Hudak. Para-functional programming. *Computer*, 19(8):60–70, August 1986.

[IH89] Takayasu Ito and Robert Halstead, Jr., editors. *Parallel Lisp: Languages and Systems.* Springer-Verlag, 1989. LNCS number 41.

[Jag91] Suresh Jagannathan. Customization of First-Class Tuple-Spaces in a Higher-Order Language. In *Conference on Parallel Languages and Architectures Europe*, pages 254–276, June 1991.

[JP92a] Suresh Jagannathan and James Philbin. A Customizable Substrate for Concurrent Languages. In *ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, June 1992.

[JP92b] Suresh Jagannathan and James Philbin. A Foundation for an Efficient Multi-Threaded Scheme System. In *Proceedings of the 1992 Conf. on Lisp and Functional Programming*, June 1992.

[JP92c] Suresh Jagannathan and Jim Philbin. STING: A Customizable Substrate for Concurrent Symbolic Computing. Technical Report 91-003-3-0050-1, NEC Research Institute, 1992.

[KHM89] David Kranz, Robert Halstead, and Eric Mohr. Mul-T: A High Performance Parallel Lisp. In *Proceedings of the ACM Symposium on Programming Language Design and Implementation*, pages 81–91, June 1989.

[KKR+86] David Kranz, R. Kelsey, Jonathan Rees, Paul Hudak, J. Philbin, and N. Adams. ORBIT: An Optimizing Compiler for Scheme. *ACM SIGPLAN Notices*, 21(7):219–233, July 1986.

[KLMO91] Anna R. Karlin, Kai Li, Mark S. Manasse, and Susan Owicki. Empirical studies of competitive spinning for a shared-memory multiprocessor. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 41–55. Association for Computing Machinery SIGOPS, October 1991.

[KR90] R.M. Karp and V. Ramachandran. A survey of parallel algorithms for shared-memory machines. Technical report, University of California, Berkeley, 1990.

[KS91a] Hongyi Zhou Karsten Schwan, Ahmed Gheith. Building families of object-based multiprocessor kernels. Technical report, Georgia Institute of Technology, Atlanta, GA, February 1991.

[KS91b] James J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 213–25. Association for Computing Machinery SIGOPS, October 1991.

[LABW91] Butler Lampson, Martín Abadi, Michael Burrows, and Edward Wobber. Authentication in distributed systems: theory and practice. In *Proceedings of 13th ACM Symposium on Operating*

Systems Principles, pages 165–82. Association for Computing Machinery SIGOPS, October 1991.

[LGG+91] Barbara Liskov, Sanjay Ghemawat, Robert Gruber, Paul Johnson, Liuba Shrira, and Michael Williams. Replication in the Harp file system. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 226–38. Association for Computing Machinery SIGOPS, October 1991.

[LH73] Henry Lieberman and Carl Hewitt. A Real-Time Garbage Collector Based on the Lifetime of Objects. *Communications of the ACM*, 26(6):419–429, June 1973.

[LH86] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *Proceedings of the 1986 5th Annual ACM Symposium on Principles of Distributed Computing*, pages 229–239, 1986.

[Li88] Kai Li. Ivy: A shared virtual memory system for parallel computing. *Proceedings of the 1988 International Conference on Parallel Processing*, 2:94–101, August 1988.

[LP91] Z. Lahjomri and T. Priol. KOAN: A shared virtual memory for the iPSC/2. Technical Report 1504, INRIA, 1991.

[LR80] Butler Lampson and D. Redell. Experiences with Processes and Monitors in Mesa. *Communications of the ACM*, 23(2):104–117, February 1980.

[LS88] Barbara Liskov and Liuba Shrira. Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems. In *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation*, pages 260–268, 1988.

[LS89] Kai Li and Richard Schaefer. A Hypercube Shared Virtual Memory System. *Proceedings of the 1989 International Conference on Parallel Processing*, 1:125–131, 1989.

[MH92] Nir Shavit Maurice Herlihy, Beng-Hong Lim. Low contention load balancing on large-scale multiprocessors. Technical report, DEC Cambridge Research Labratory, Cambridge, MA, May 1992.

[Mic90] Sun Microsystems. *Lightweight Processes*, 1990. In SunOS Programming Utilities and Libraries.

[MKH90] Rick Mohr, David Kranz, and Robert Halstead. Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, June 1990.

[MKH91] Rick Mohr, David Kranz, and Robert Halstead. Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs. *Communications of the ACM*, 2(3):264–280, June 1991.

[Moh91] Eric Mohr. Dynamic Partitioning of Parallel Lisp Programs. Technical Report YALEU/DCS/RR-869, Yale University, October 1991.

[MP89] Henry Massalin and Calton Pu. Threads and Input/Output in the Synthesis Kernel. *Proceedings of the 12th Symposium on Operating System Principles*, pages 159–166, December 1989.

[MP91] Keith Muller and Joseph Pasquale. A high performance multi-structured file system design. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 56–67. Association for Computing Machinery SIGOPS, October 1991.

[MSLM91] Brian D. Marsh, Michael L. Scott, Thomas J. LeBlanc, and Evangelos P. Markatos. First-class user-level threads. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 110–21. Association for Computing Machinery SIGOPS, October 1991.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, 1990.

[oOS90] Technical Committee on Operating Systems. Thread Extensions for Protable Operating Systems. Technical report, IEEE Computer Society, December 1990.

[Osb90] Randy Osborne. Speculative Computation in MultiLisp. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 198–208, 1990.

[PC90] Greg Papadopolus and David Culler. Monsoon: An Explicit Token-Store Architecture. In *Proceedings of the 1990 Conference on Computer Architecture*, pages 82–92, 1990.

[Phi93] James Philbin. The Sting Reference Manual. Technical report, NEC Research Institute, January 1993.

[PW89] J. Pehoushek and J. Weening. Low Cost Process Creation and Dynamic Partitioning in Qlisp. In *Proceedings of the U.S./Japan Workshop on Parallel Lisp*, pages 182–199, June 1989.

[RA82] Jonathan A. Rees and Norman I. Adams. T: A Dialect of Lisp or, LAMBDA: The Ultimate Software Tool. In *Proceedings of the ACM Symposium on Lisp and Functional Programming*, pages 114–122, 1982.

[RAA+92] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrman, C. Kaiser, S. Langlois, P. Leonard, and W. Newhauser. Overview of the Chorus Distributed Operating System. In *Workshop Proceedings Micro-Kenels and other Kernel Architectures*, pages 39–69, April 1992.

[Raf76] A. Rafii. *Empirical and Analytical Studies of Program reference Behavior.* PhD thesis, Stanford University, Stanford, California 94305, July 1976.

[Rau75] B. R. Rau. The stack working set: a characterization of spatial locality. Technical Report 95, Stanford U., Stanford CA (USA), July 1975.

[RC86] Jonathan Rees and William Clinger, editors. The Revised[3] Report on the Algorithmic Language Scheme. *ACM Sigplan Notices*, 21(12), 1986.

[Rep91] John Reppy. CML: A Higher-Order Concurrent Language. In *Proceedings of the SIGPLAN'91 Conference on Programming Language Design and Implementation*, pages 293–306, June 1991.

[RLW85] Paul Rovner, Roy Levin, and John Wick. On Extending Modula-2 For Building Large, Integrated Systems. Technical Report 3, DEC Systems Research Center, January 1985.

[RO91] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 1–15. Association for Computing Machinery SIGOPS, October 1991.

[RPLEK91] Jr Richard P. LaRowe, Carla Schlatter Ellis, and Laurence S. Kaplan. The robustness of NUMA memory management. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 137–51. Association for Computing Machinery SIGOPS, October 1991.

[RS91] Thomas L. Rodeheffer and Michael D. Schroeder. Automatic reconfiguration in Autonet. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 183–97. Association for Computing Machinery SIGOPS, October 1991.

[RSaU] Larry Rudolf, Miriam Slivkin-allalouf, and Eli Upfal. A Simple Load Balancing Scheme for Task Allocation in Parallel Machines.

[RV91] P. Venkat Rangan and Harrick M. Vin. Designing file systems for digital audio and video. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 81–94. Association for Computing Machinery SIGOPS, October 1991.

[SD89] C. Scheurich and M. Dubois. Dynamic page migration in multiprocessors with distributed global memory. *IEEE Transactions on Computers*, 38(8):1154–1163, August 1989.

[SR90] Vijay Saraswat and Martin Rinard. Concurrent Constraint Programming. In *Proceedings of the 17th ACM Symposium on Principles of Programming Languages*, pages 232–246, 1990.

[Ste77] Guy Lewis Steele. Macaroni is Better than Spaghetti. In *Proceedings of AI and Programming Languages Conference*, pages 60–66, August 1977.

[Ste78] Guy Steele Jr. Rabbit: A Compiler for Scheme. Master's thesis, Massachusetts Institute of Technology, 1978.

[SW91] Frank Schmuck and Jim Wyllie. Experience with transactions in QuickSilver. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 239–53. Association for Computing Machinery SIGOPS, October 1991.

[TG89] A. Tucker and A. Gupta. Process Control and Scheduling Issues for Multiprogrammed Shared-Memory Machines. In *Proceedings of the 12th Annual ACM Symposium on Operating Systems Principles*, pages 114–122, 1989.

[TRG+87] A. Tevanian, R. Rashid, D. Golub, D. Black, E. Cooper, and M. Young. Mach Treads and the UNIX Kernel: The Battle for Control. In *1987 USENIX Summer Conference*, pages 185–197, 1987.

[TSS88] Charles Thacker, Lawerence Stewart, and Edward Satterthwaite, Jr. Firefly: A Multiprocessor Workstation. *IEEE Transactions on Computers*, 37(8):909–920, August 1988.

[Ung84] David Ungar. Generation Scavenging: A Non-Disruptive High Performance Storage Reclamation Algorithm. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 157–167, 1984.

[USCH92] David Ungar, Randall B. Smith, Craig Chambers, and Urs Holzle. Object, Message, and Performance: How they Coexist in Self. *Computer*, 25(10):53–64, October 1992.

[VR88] M. Vandevoorde and E. Roberts. WorkCrews: An Abstraction for Controlling Parallelism. *International Journal of Parallel Programming*, 17(4):347–366, August 1988.

[VZ91] Raj Vaswani and John Zahorjan. The implications of cache affinity on processor scheduling for multiprogrammed, shared memory multiprocessors. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 26–40. Association for Computing Machinery SIGOPS, October 1991.

[Wan80] Mitch Wand. Continuation-Based MultiProcessing. In *Proceedings of the 1980 ACM Lisp and Functional Programming Conference*, pages 19–28, 1980.

[Wee89] J. Weening. *Parallel Execution of Lisp Programs.* PhD thesis, Stanford University, June 1989.

[WLH81] William Wulf, Roy Levin, and Samuel Harbison. *HYDRA/C.mmp: An Experimental Computer System.* McGraw-Hill, 1981.

[YTR+87] Michael Young, Avadis Tevanian, Richard Rashid, David Golub, Jeffrey Eppinger, Jonathan Chew, William Bolosky, David Black, and Robert Baron. The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System. Technical Report CMU-CS-87-155, Carnegie-Mellon University, 1987.