An Overview of the STING Operating System

James Philbin

***NEC* Research Institute**
4 Independence Way, Princton, NJ 08540
philbin@research.nj.nec.com
Tel (609) 951-2749 Fax (609) 951-2482

Abstract

Sting is an operating system designed to serve as a highly efficient substrate for modern programming languages. Sting includes support for: various models of parallelism and synchronization; lazy, normal, and eager evaluation; automatic storage management; and topology mapping.

Sting is different from other operating systems in several respects. The basic concurrency objects in Sting, virtual processors and lightweight threads, are first class. They provide the user with a level of expressiveness and control unavailable in other operating systems. Thread scheduling and migration are completely customizable. This allows Sting to be used in real-time, interactive, and batch oriented operating environments. Sting also provides a novel system of storage management, including a new kind of parallel garbage collection. Finally, Sting introduces several new optimizations which significantly improve locality of reference and reduce storage requirements, thereby increasing efficiency.

1.0 Introduction

Sting is an operating system designed to support modern programming languages such as Scheme, SmallTalk, ML, Modula3, or Haskell. It provides a foundation of low level, orthogonal constructs, which allows the language designer or implementer to build the various constructs required by these languages easily and efficiently.

Modern programming languages have more extensive requirements than traditional programming languages such as Cobol, Fortran, C, or Pascal. The list below identifies some of the requirements that distinguish modern from traditional languages.

- *Parallelism* - The growing availability of general purpose multi-processors has lead to increased interest in building efficient and expressive platforms for concurrent programming. Most efforts to incorporate concurrency into high-level programming languages involve the addition of special purpose primitives to the language.
- *Multiple Synchronization Models* - There are many synchronization protocols used in parallel or asynchronous programming. A modern operating environment should as far as possible provide the primitives to support the various protocols.
- *Lazy and Eager Evaluation* - Many modern languages support either lazy evaluation or eager evaluation or both. It is important for the operating system to provide the full range of evaluation strategies from lazy to eager.
- *Automatic Storage Management* - This has become a fundamental feature of many modern languages, because automatic storage management allows more expressive programs, while at the same time reducing both the number of errors in and the complexity of programs.
- *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.

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

Sting was conceived and designed not only as an efficient operating system for parallel computers, but also as an experimental platform for exploring and comparing different models of parallel programming. To date we have

Published in Proceedings of the 4th NEC Software Conference, pp. 371-379, Dallas, Texas, USA, Oct. 19.92

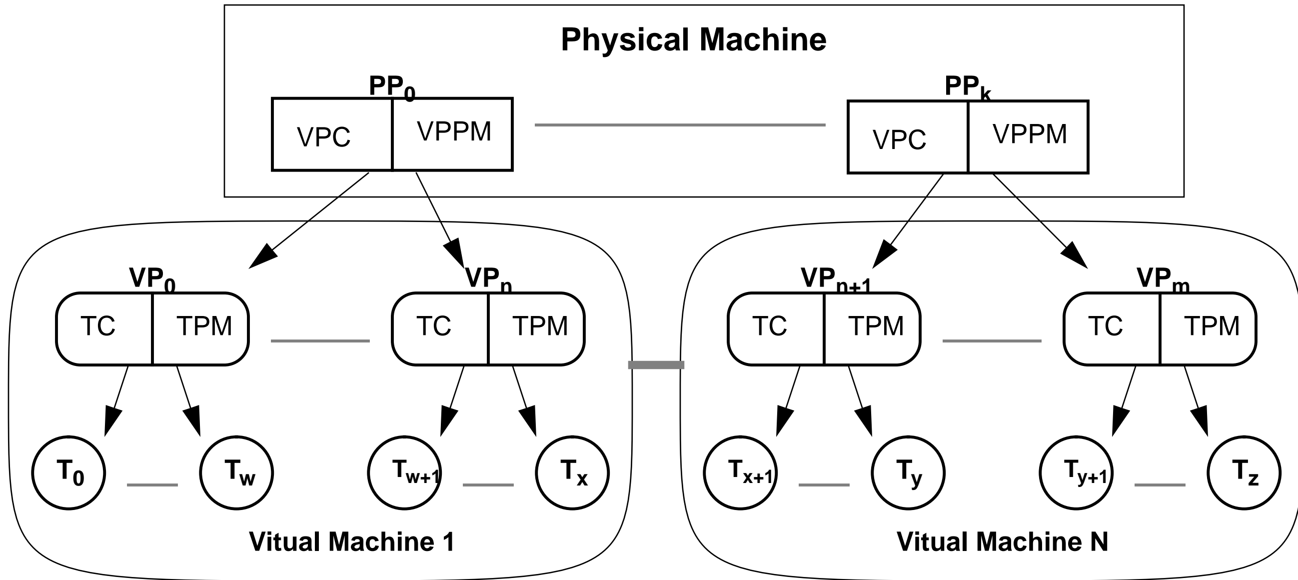

Figure: 1 The Sting Abstract Architecture

implemented several different algorithms corresponding to different paradigms of parallelism including: result parallelism, master/slave parallelism, and speculative parallelism.

We have also implemented several different parallel programming models including: futures [3], first class tuple spaces [6], and engines[4] and compared them using various parallel algorithms that have been implemented in Sting. A report analyzing the relative performance and merits of the various programming models using several different algorithmic paradigms will be forthcoming.

The Sting operating system architecture is composed of several layers of abstraction, see Figure 1. The first layer is the *physical machine*, which is composed of *physical processors*. This layer corresponds to the micro-kernel in newer operating systems. 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, with each virtual processor mapped onto a physical processor. The third layer of abstraction defines *threads*. Threads are lightweight processes which are run on virtual processors.

For clarity, it should be mentioned that each of the above layers is a software system. Thus, when we speak of a “physical machine”, we are referring to the software layer that controls the actual physical hardware. The same holds true for “physical processor”. Throughout this paper, unless otherwise noted, the phrases *physical machine* and *physical processor* will denote the software systems rather than the actual hardware.

The rest of this paper 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.

2.0 Physical Machines and Physical Processors

The lowest level software system in the Sting architecture is the physical machine layer. This layer coordinates the physical processors and creates and manages virtual machines.The physical machine abstraction is equivalent to the micro-kernel in other modern operating systems such as Chorus [11], Mach [12], or Psyche [9].

The physical machine kernel is an abstraction of the physical hardware. The physical hardware is composed of some number of processors, physical memory, and devices connected in a physical topology. The physical machine controls and coordinates the interaction of the hardware elements, including the handling of hardware device interrupts, and the management of the physical memory hierarchy.

The physical machine is also responsible for providing safe and well-controlled access to the various hardware components for other parts of the Sting architecture. In Sting, many virtual machines may run on a single com-

puter system, but there is only one physical machine. Finally, the physical machine has its own private virtual address space, as well as complete access to any other virtual address space in the system.

The physical machine layer is an abstraction which presents the same interface on different types of physical hardware. The abstraction maps the same operating system architecture onto different hardware architectures, including both single and multi-processors. On multi-processors, it also hides the distinction between physically shared and physically disjoint memory.

2.1 Physical Processors

The number of physical processors in the physical machine corresponds to the actual number of hardware processors. Each physical processor handles hardware interrupts and its portion of the physical memory architecture. In addition, physical processors multiplex virtual processors that are mapped onto them.

In Sting, virtual processors are multiplexed on physical processors in the same manner that processes are multiplexed on a processor in more traditional operating systems. That is, the physical processor divides the instruction cycles of the hardware processor between itself and all of the virtual processors mapped onto it. Physical processors context switch virtual processors either because of preemption or because the virtual processor explicitly requests it.

The physical processor abstraction is divided into two separate components, the *virtual processor controller* and the *virtual processor policy manager*. 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 which implements the abstract architecture..

2.2 Virtual Processor Controller

Each VP controller manages the virtual processors which are mapped on to its associated physical processor. The VP controller manages all virtual processor state changes.

A virtual processor can be in any of the following states:

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 state of the current VP and the current thread running on that VP are saved and another VP with its current thread is run.

2.3 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 any of the following:

- the initial mapping of a virtual processor to a physical processor;
- which VP to run next when the current VP releases the physical processor for some reason; or
- when and which VP to migrate from or to the 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. They 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, thus allowing it to be customized for different operating system environments as diverse as real time, interactive, or computationally intensive systems.

3.0 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:

- one or more virtual processors;
- a virtual topology;

- a virtual address space;
- 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 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 local address space, which is shared by all virtual processors 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 virtual machine's address space.

As with other operating system kernels, the physical memory is hidden from all virtual machines. Virtual machines only have access to virtual memory. This allows physical memory to be configured 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 virtually shared memory, i.e. every address in a virtual address space is, at least conceptually, accessible from any physical or virtual processor in the machine. The physical machine abstraction ensures that the virtual memory is coherent.

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.

3.1 Virtual Processors

A Virtual Processor is an abstraction of a hardware CPU. 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, e.g. when a thread blocks in the physical machine kernel.

Each VP is associated with a virtual machine and with a physical processor. 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. The first classness of 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.

Just as control and policy are separated in the physical processor, so they are in the virtual processor. Each virtual processor is composed of two software components: the *thread controller* and the *thread policy manage*r.

3.2 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 while 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 the lightweight thread systems that have been run on them. There have been two main approaches to solving this problem, that of Anderson, et. al.[1] and that of Marsh, et. al.[9]. Our solution is closer to that of Marsh, but it is distinct because it relies on the use of continuations to simplify the interaction as explained below.

3.2.1 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 (see below) associated with it. A continuation can be thought of as the entire context of a computation at a particular program point. However, saving the current continuation involves saving only the currently active registers in the virtual machine. 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 [13].

3.2.2 Kernel Calls and Continuations

While Sting's virtual machines serve some of the same purposes as kernel processes in operating systems such as Unix and Mach, they are in fact quite different.

In Sting there are no kernel processes, only lightweight user space threads. When a thread makes a call into the 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 and local heap) associated with the thread. Kernel processes and 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. In order for the thread to notify the virtual processor that it has blocked in the physical machine kernel, the interface to the thread controller must be available from the physical machine kernel, which it is.

3.3 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, 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, i.e. 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.

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

Just as the VP policy manager presents a well-defined interface to the VP controller, so the thread policy manager presents a well-defined interface to the thread controller. The data structures which 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 to different virtual machines. This allows the user to customize policy decision depending on the type of program being run. For example, a computationally intensive program such as a fluid dynamics simulator, might use a lifo scheduling policy, while a window manager or user shell might use a priority based fifo policy.

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.

To date, we have implemented and tested several different thread policy managers. A report on their design and performance will be forth coming.

3.4 Virtual Topologies

Virtual topologies are another novel aspect of Sting. Virtual topologies are easy to implement in Sting thanks to the first classness of virtual processors. 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 because each virtual processor and thread can perform self-relative operations. Self-relative operations are done using the procedures current-virtual-processor and current-thread.

The idea of mapping computations to specific processors was first explored by Hudak [5], but since his processors where neither first class nor virtual, user defined topologies were more difficult to implement.

The advantage of virtual topologies is that the user can build topology abstractions which model the communications structure of their program. For example, a user programming an algorithm in which the communication pattern between threads looks like a binary tree might build abstractions called

- left-child,
- right-child, and
- parent.

These allow the programmer to map the threads to the appropriate virtual processors which in turn are mapped to the appropriate physical processors. Virtual topologies enable a programmer to run the same program, without modification, on different physical topologies by simply customizing the virtual topology to the new physical topology. Virtual topologies allow parallel algorithm to be portable and optimized across different architectures.

A significant body of work on mapping one topology onto another optimally or near optimally exists. It is principally due to Bhatt and Greenberg [2]. Their algorithms 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.

4.0 Threads

Threads are the means of execution in Sting. Any code that is executing is being executed by some thread. Sting's threads are first class, i.e. they can be passed to and returned from procedures and stored in data structures. Sting threads also have values. The value of a thread is the value of the expression it evaluates. Being first class and having values allows Sting's threads to be used in ways that threads in other thread systems are not, namely as dynamic data structures. Since Sting's threads can be evaluated lazily, normally, or eagerly, these dynamic data structures can be speculative or infinite.

All threads in Sting are lightweight in the sense that they reside in user space. Sting threads are in some sense lighter weight than those in most other lightweight thread systems. This is because the thread itself is a very small data structure, on the order of ten words of memory, which contains among other things:

- the expression which the thread is to evaluate,
- the dynamic state in which the thread was created,
- the priority and quantum of the thread, and
- the genealogy of the thread.

The thread data structure can be small because the dynamic context (see below) of the thread is not allocated until the thread begins evaluating. Threads can be either preemptible or non-preemptible and can also be used as co-routines.

4.1 Thread States

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

delayed - When a thread is first created it is in the delayed state. A delayed thread will not evaluate its value until that value is demanded either explicitly or implicitly.

scheduled - A scheduled thread is one which has been scheduled to run, but which has not yet begun evaluating.

evaluating - An evaluating thread has started executing the expression associated with it and has not yet determined the value of that expression.

stolen - A stolen thread is one that is being evaluated in the dynamic context of another thread. Stealing is an optimization introduced by Sting. It is discussed below.

determined - A determined thread is one which has been assigned a value.

4.2 Thread Genealogy

Each thread has its own genealogy, i.e. each thread knows it's ancestors, and its descendents and their relative ages. Thread genealogy is useful for debugging and algorithm analysis. It aids algorithm analysis in two ways. First, it lets the designer examine the asynchronous and communication structures of an algorithm. Second, it allows the user to evaluate the performance of an algorithm on a thread by thread basis.

Sting allows the user to keep thread genealogy in one of two ways; either the genealogy contains all the threads whether living or dead, or the genealogy contains only live threads. The thread policy manager makes this decision concerning genealogy.

4.3 Thread Groups

Threads are organized in groups called *thread groups*. The root thread of the group is created at the same time as the thread group. Each new thread belongs to the same thread group as its parent, unless it is the root thread of a group.

As with virtual processors and threads, thread groups are first class, with the concomitant benefits. Sting uses thread

groups to aggregate threads that are cooperating on a particular computation or sub-computation.

When a thread group is created, Sting allocates a new shared heap to it. Every thread in the group uses this heap to allocate data which will be shared by other threads in the group. When a thread group terminates,, Sting garbage collects all the threads in the group and removes the subtree of the genealogy tree represented by that thread group.

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*, the root thread of which 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.

Thread groups are useful not only for organizing the threads in a computation, but also for debugging. When a thread in a group encounters an error, it suspends all threads in that group. It then invokes the debugger which can be used to inspect the state of the thread which encountered the error, or that of any other thread in the group. Likewise, the user can interrupt the execution of an entire thread group at any point and then inspect any or all threads in the group.

5.0 Thread Dynamic Contexts

When a thread is created it is a small data structure, as explained above, but when a thread begins evaluation Sting allocates a dynamic context for it. Delaying the allocation of the dynamic 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.

A thread's dynamic 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.

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 the following information about the thread: its current state, a requested next state, the set of threads currently blocked waiting for it to complete, and the thread's geneology. It also contains references to the stack, the local heap, and the shared heap.

A thread uses its stack in the traditional manner, for allocating objects, including procedure call frames. Stack allocated objects are only accessible to the associated thread and their lifetime is known not 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.

Each stack is composed of a series of stack blocks linked in a chain. The dynamic context is initially allocated with one small stack block.[1] When a block overflows, a new block, twice the size of the previous block, is chained onto it. A thread does not de-allocate stack blocks until the thread completes execution. Stack blocks are yet another method of reducing the storage requirements of threads in Sting.

Objects which are only accessible to the thread which created them and whose lifetimes *may exceed* the lifetime of the procedure which created them are allocated in the local heap. Objects in the local heap can never be referenced by any thread other than the one that created them. The fact that both the stack and local heap cannot contain shared data allows them to be allocated in physical memory that is local to the physical processor, and therefore fast. 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. Thus the memory coherence problem which occurs on all physically or virtually shared memory multi-processors only applies to shared heaps.

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 heaps are 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, but a detailed discussion of it is beyond the scope of this paper. Interested readers should see [10].

5.1 TCB States

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

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.

1. This is a tunable parameter, but it is usually on the order of 16Kb, but it can be as small as 1Kb.

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 for some specified time or indefinitely.

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.

When a thread has finished evaluating, i.e. has become terminated, its TCB is re-initialized and its entire dynamic context is returned to the pool of dynamic contexts associated with the thread's virtual processor.

TCB state transitions are accomplished more quickly then in other operating systems, because there is no need to lock the TCB on a 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 dynamic context of the thread which calls them.

5.2 Delaying TCB Allocation and Locality

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

Since dynamic contexts are internal to the Sting system, i.e. a user program can never gain access to them, when a thread completes executing, its dynamic context can be recycled for use by other, as yet un-evaluated, threads. Dynamic 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 a dynamic context to a scheduled thread when it begins evaluating on a virtual processor. The dynamic context is allocated in one of three ways:

- If the thread executing on the virtual processor just prior to beginning the evaluation of a new thread has completed execution, then its dynamic context is available for re-allocation. Further, its dynamic 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 dynamic context that has just finished executing.
- If the thread executing prior to starting a new thread has not finished its execution, then Sting allocates a dynamic context from the pool of dynamic contexts associated with the virtual processor. Since the pool is organized in a lifo manner, the dynamic context allocated is again the one with the most locality; since, of the dynamic contexts available, it was used most recently.
- Finally, if the thread executing prior to starting a new thread has not completed its execution and the pool of dynamic contexts associated with the virtual processor is empty, Sting creates a new dynamic context and allocates it to the thread. Since this dynamic 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.

Delaying dynamic context allocation minimizes the number of dynamic contexts in use at any one time, since the number of dynamic contexts in use corresponds to the number of threads that are currently being evaluated, not to the number of threads that have been created. Delaying dynamic 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 dynamic context, than it is to migrate an evaluating one, because threads which have not begun evaluating have no dynamic context associated with them. Thus, only the thread data structure, which is small, is migrated. The final advantage of delaying dynamic context allocation is that it allows Sting to perform an optimization called *thread stealing*.

5.2.1 Thread Stealing

Thread stealing is a novel optimization introduced by Sting. It occurs when one thread demands the value of another thread. It has the effect of improving locality of reference and reducing the storage required for the evaluation of threads.

In Sting, all threads have a value, i.e. the value of the expression which the thread evaluates. When a thread has completed the evaluation of its expression, the value of that expression is stored in the thread data structure and the thread state becomes *determined*.

When a thread T^1 requests the value of another thread T^2 it blocks until the value of T^1 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 dynamic context. When thread T^2 is evaluated using the dynamic context of T^1 we say that T^1 has *stolen* T^2. Thread stealing

does not slow down the evaluation of T^1 since by definition T^1 has to block until T^2 is finished evaluating.

Thread stealing results in reduced storage requirements since the same dynamic context is used to evaluate two threads. It results in improved thread locality for two reasons. First, the dynamic context of T^1 is already loaded in the physical memory and caches of the processor, so evaluating T^2 in its 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.

6.0 Conclusion

Sting is a platform for building efficient[7][8] asynchronous programming primitives and experimenting with new parallel programming paradigms. In addition, the design also allows different concurrency models to be evaluated competitively. Sting has proven to be an efficient and expressive environment for implementing both parallel programming languages and parallel algorithms.

We have programmed several interesting parallel algorithms using Sting. In the future, we would like to expand this collection of algorithms to include some large problems in physics which are under study at NECI.

Sting is currently a prototype implemented on an eight processor Silicon Graphics PowerSeries 480 running Unix. The next step is to integrate Sting into the microkernel of an operating system such as Mach or Chorus.

Acknowledgement

The design of Sting is joint work with Suresh Jagannathan of NEC Research Institute.

References

[1] 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.

[2] David Saks Greenberg. *Full Utilization of Communication Resources*. PhD thesis, Dept. of Computer Science, Yale University, 1991.

[3] Robert Halstead. Multilisp: A Language for Concurrent Symbolic Computation. *Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[4] C.T. Haynes and D.P. Friedman. Engines Build Process Abstractions. In *slfp84*, pages 18–24, Computer Science Department, Indiana University, 1984.

[5] Paul Hudak. Para-Functional Programming. *IEEE Computer*, 19(8):60–70, August 1986.

[6] Suresh Jagannathan. A Generalized Framework for First-Class Tuple-Space Systems. Technical Report 92-048-3-0050-4, NEC Research Institute, June 1992.

[7] Suresh Jagannathan and James Philbin. A Customizable Substrate for Concurrent Languages. In *ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, June 1992.

[8] 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.

[9] 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.

[10] James Philbin. XXXX. Technical Report CMU-CS-88-154, NEC Research Institute, October 1992.

[11] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrman, C. Kaiser, S. Langlois, P. Leonard, and W. Neuhauser. Overview of the chorus distributed operating system. In *Usenix Workshop: Micro-Kernels and Other Kernel Architectures*, pages 39–69. Association for Computing Machinery SIGOPS, April 1992.

[12] 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.

[13] Mitch Wand. Continuation-Based MultiProcessing. In *Proceedings of the 1980 ACM Lisp and Functional Programming Conference*, pages 19–28, 1980.