A Portable Run-Time System for the SR Concurrent Programming Language

Gregory D. Benson and Ronald A. Olsson
Department of Computer Science
University of California, Davis, CA 95616, U.S.A
{benson,olsson}@cs.ucdavis.edu
tel:(916)752-6476 fax:(916)752-4767

Abstract

A run-time system is the glue between the compiler-generated code and the underlying platform. In addition to having a significant impact on performance, run-time systems are also largely responsible for the degree of portability of a language implementation. Concurrent programming languages are particularly dependent on threads and communication, both of which can vary greatly from platform to platform.

This paper presents our current work in developing a portable run-time system for the SR programming language. We argue that portability must be addressed at multiple layers in a run-time system, and not just in a single virtual machine layer. We also present detailed design issues of our run-time system, especially with respect to the thread interface. Performance results are presented for three Linux-based thread packages. Although our work focused on the SR implementation, our overall approach and general observations should also be applicable to implementations of other concurrent programming languages.

1 Introduction

The implementation of a concurrent programming language (parallel or distributed) often consists of a *run-time system* (RTS)[1] in addition to a compiler. Run-time systems are used to simplify code generation, to implement the dynamic aspects of complex language elements, and to interface with the underlying operating system and/or architecture. Often the gap between the semantics of the language and the underlying platform is too large to accommodate direct translation from source code to object code. As such, a run-time system provides interfaces for manipulating the internal representation of language elements. The compiler's code generator is then responsible for producing code that properly invokes the run-time system interfaces. Indeed, many sequential languages require run-time systems for all of these reasons, but concurrent languages typically require extensive support for threads and communication.

Run-time systems have a large impact on both the performance and portability of programs written in concurrent programming languages. While performance is often a primary concern for language implementations, portability is becoming increasingly more important as the target platforms for parallel and distributed computing grows more diverse. Although current trends in hardware indicate convergence to networks of workstations and shared-memory multiprocessors, the range of software support (i.e., thread packages, communication interfaces, and operating system kernels) remains in flux. In addition, portability is often a reason to use a concurrent programming language, but program portability is limited to the actual number of platforms supported by the language implementation.

A common approach to achieve portability is to define a set of interfaces to low-level functionality such as threads and communication. The interfaces define a *virtual machine*, which is accessed by higher-level modules of a run-time system. Platform dependencies are restricted to the implementation of the virtual machine. Both Panda [9] and Nexus [14] are based on this approach. These systems have demonstrated good portability among the platforms they support. However, each system has been developed for a finite number of platforms. An implementor that wants to port a language to a platform for which a virtual machine layer does not support must now port the virtual machine. For some languages that do not utilize

[1] We use RTS to denote both "run-time system" and "run-time support."

all of a virtual machine's functionality, this approach may be more effort than implementing the language directly on top of the new platform. This observation in conjunction with desire to directly utilize different operating system interfaces has led us to tackle portability by isolating system dependencies to specific components in the run-time system. This approach is more of a "philosophy" rather than a methodology and is complementary to the virtual machine approach.

We are using the SR concurrent programming language [2] as a vehicle to explore various run-time support issues. In particular, we have been investigating the design and implementation of SR's run-time support on top of microkernel operating systems [6, 8]. This work combined with a project in which SR was implemented using Panda [23] has prompted us to develop a more flexible run-time system that is intended to run on a wide range of platforms.

This paper presents our approach to portability in the SR run-time system and some general observations on portable run-time systems based on our experience. In addition to presenting specific design and implementation details of our run-time system, we also give performance results for three Linux-based thread packages: a custom user-level package, MIT PThreads [22], and LinuxThreads [20]. These results reveal general performance implications for these thread packages and the performance differences based on thread-package functionality. Although our work focused on the SR implementation, our overall approach and general observations should also be applicable to implementations of other concurrent programming languages.

The rest of this paper is organized as follows. Section 2 discusses various approaches for dealing with portability in run-time systems. Section 3 briefly describes the SR concurrent programming language. Section 4 presents several specific design issues we encountered with our portable version of the SR run-time system. Section 5 reports on the performance of SR implemented with various thread packages on top of the Linux operating system. Finally, Section 6 offers some concluding remarks and directions for future work.

2 Dealing with Portability

Portability is a complicated software engineering problem that is handled in different ways by different types of programs, from applications to compilers to operating systems. For example, the GNU C compiler uses a register transfer language to simplify retargeting the back-end to different architectures.

In many ways a run-time system is similar to an operating system kernel. A run-time system provides a set of interfaces to basic services. These interfaces are an abstraction of the underlying operating system and architecture. The "users" of the RTS interfaces are compiler-generated programs or perhaps a language interpreter.

Many programs can achieve reasonable portability by adhering to ANSI C and POSIX library functions. However, concurrent programming languages make extensive use of threads and communication interfaces. Unfortunately, the interfaces for threads and communication can vary greatly from platform to platform, thus making it difficult to implement a run-time system for multiple targets. This is true despite standardization efforts such as MPI [24] and PThreads [17]. It is also not clear if the standard interfaces are always desirable. For example, it has been observed that many current thread interfaces, including PThreads, are not very flexible [15].

Portability can be viewed from two perspectives: from *portability clients* (programs for which portability is desired or required) and from *portability providers* (programs, libraries, or interfaces designed to provide portability). It is infeasible for portability clients to completely rely on portability providers, as a particular provider usually only targets a subset of possible platforms. To this end, clients must take measures to ensure that multiple providers can be utilized. For example, the GNU Autoconf program [21] is a tool to aid portability clients in assessing the capabilities of a target platform so appropriate, automatic adjustments can be made to the source code. Such a tool is necessary to effectively target a large number of different platforms, otherwise manual techniques result in an unwieldy number of conditional compilation macros, making code unmanageable. We generally take the point of view of the language implementation, a portability client, and are concerned with techniques that allow a language to be implemented as quickly as possible with good performance for a large number of target platforms.

Tools can help manage portability to some extent, but, especially in the domain of run-time systems, proper structuring can also result in good portability. This section identifies two structuring approaches for dealing with portability in run-time systems: the virtual machine approach and the component approach.

2.1 The Virtual Machine Approach

Some efforts aim to provide portable run-time support either by utilizing standardized interfaces or by providing a portable *virtual machine* (see Figure 1). The virtual machine is simply a layer between the run-time system and the kernel primitives or the hardware. This layer provides an abstraction of the underlying operating system and architecture.

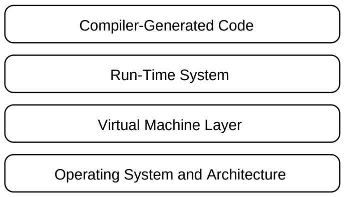

Figure 1: The Virtual Machine Approach

The Panda system [9] is a portable platform for supporting parallel programming languages. The platform consists of two layers: the system layer and the Panda layer. The system layer interfaces with operating system kernel primitives; this level is kernel dependent. The Panda layer provides a virtual machine for supporting run-time systems. The virtual machine supports threads, message passing, remote procedure call, totally-ordered group communication, and collective communication. Panda targets run-time systems that provide distributed shared data or distributed shared objects.

Nexus [14] is a distributed thread library that provides a virtual machine layer similar to Panda. Nexus was designed to be used as a back-end for concurrent programming languages, e.g., CC++ and Fortran-M [13]. It provides support for creating threads on a processor, for specifying synchronization among the threads of a node, and for communication among threads through the active-message paradigm.

2.2 The Component Approach

The component approach, illustrated in Figure 2, attempts to separate run-time system functionality and system-dependent interfaces into individual modules[2]. Various forms of the component approach have been proposed in [7] and [18]. The basic idea is to isolate system dependencies into components so that they can be specialized on a per-platform basis. Note that this approach can be complementary to the virtual machine approach, in that system-dependent components may be implemented in terms of virtual machine interfaces for a particular platform. However, from the perspective of the language implementation there is no "bottom" layer unless a particular implementation chooses to use such a layer.

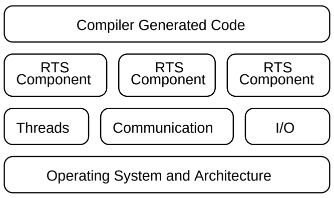

Figure 2: The Component Approach

A component is a very general entity; it encapsulates some type of functionality. We do not place any constraints on components; for example, a component can be implemented as an object-oriented class. However, due to performance requirements it is often necessary to avoid traditional black-box techniques [19]. As a consequence, some components may have exposed state that can be manipulated by other components.

As a specific example, consider a run-time system that supports processes and semaphores (see Figure 3). For some implementations, processes and semaphores may be implemented in terms of the low-level interfaces provided by a thread package. In particular, processes may be implemented as threads and semaphores may be implemented with condition variables. These basic thread primitives exist in many thread packages. However, a more advanced thread package or operating system kernel may provide efficient semaphores. The right-hand side of Figure 3 illustrates this specialization. Although this example

[2] Figure 2 depicts only two levels of components, but it is feasible to have arbitrary hierarchies of components.

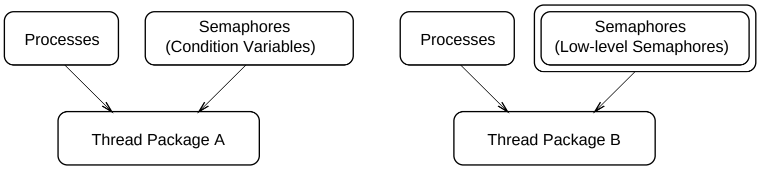

Figure 3: Component Specialization

is simple enough to be handled well by either a virtual machine layer or by conditional compilation, it also illustrates the kind of flexibility that the component approach can provide.

Although the component approach facilitates specialization of components, it is desirable to minimize the number of specialized components. That is, more specialized components means less portable run-time systems. The idea is that performance optimizations through specialization will incur a cost in portability. In this view, portability can be achieved locally, that is, within individual components.

We are still refining our vision of the component approach. Many issues still need to be resolved before such an approach can be generally useful. An important concern is the determination and/or specification of dependencies among components. A very straightforward approach is to clearly document dependencies and any state that is shared among components, therefore when a component is specialized, all the information needed for replacement is provided. This approach is being used successfully in the Flux Operating System Toolkit [11], a toolkit for building kernels.

Another crucial concern is the mitigation of specialization costs. We foresee a technique and corresponding tool that helps track components and their compositions. A good model for such a technique is the *x*-kernel [16], which is an architecture for specifying and composing micro-protocols. While the domain of the *x*-kernel is constrained to communication protocols, it demonstrates how different functional components can be combined to produce high-performance systems.

2.3 A Comparison of the Virtual Machine and Component Approaches

It is important to understand the differences between the virtual machine approach and the component approach, and their relative merits. The virtual machine approach presents a fixed interface to a run-time system. Such an interface results in less complexity at the system independent RTS layer and is the cornerstone for portability in the virtual machine approach. In addition, virtual machines can be designed to be extremely modular "underneath" the fixed interface. Such modularity results in virtual machines that are easier to port (e.g., both Panda [9] and Nexus [14] have modular designs). However, the fixed interface can hide system dependent details that may be important to the implementation of a run-time system. If a language implementor "codes around" the fixed interface, the advantages of the virtual machine are lost.

The component approach offers greater flexibility in specializing performance critical portions of a run-time system. In addition, depending on the particular virtual machine, the component approach may lead to quicker development time for new platforms because only the low-level functionality that is required by the run-time system is implemented. This is true when compared to virtual machines, such as Nexus [14], that support several abstractions, many of which may not be needed by a particular run-time system. However, the component approach, at least initially, may result in more complex run-time systems from the flexibility of specialization.

3 The SR Programming Language

SR is a language for writing concurrent programs, both parallel and distributed [2, 3]. Many of SR's abstractions correspond to modern hardware, including shared memory multiprocessors and networks of workstations. While SR does not provide expensive abstractions such as distributed shared memory or distributed shared objects [5], it does provide simple and consistent mechanisms for expressing several forms of explicit communication. SR's syntax and semantics allow the programmer to concentrate on the communication and synchronization structure of a program without having to be concerned with low-level interfaces or interface definition languages. Higher-level abstractions can be built on top of SR's communication mechanisms.

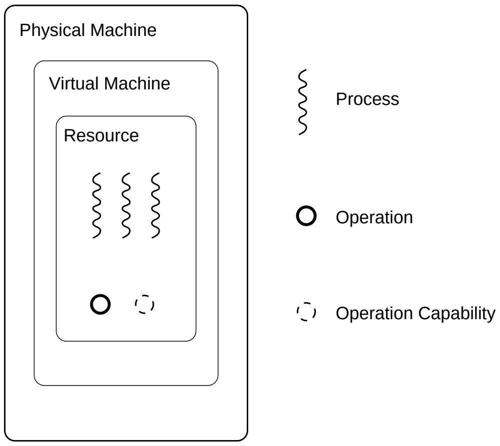

Figure 4: The SR Programming Model

The main language constructs are resources and operations. Resources encapsulate processes and variables they share; operations provide the primary mechanism for process interaction. SR also provides the concept of an address space, called a virtual machine or simply VM; processes that execute within the same address space can share variables and operations. Resources and virtual machines can be dynamically created and destroyed. Figure 4 illustrates the main SR language constructs.

An operation can be considered a generalization of a procedure: it has a name, and can take parameters and return a result. An operation is serviced in one of two different ways—by a **proc** or by **in** statements—and can also be invoked in two ways—by **call** and **send**. A **proc** resembles a conventional procedure in having parameters and a body, but invoking a **proc** causes a new process to be created to service that invocation. The **in** statement allows an existing process to accept operation invocations from other processes. Operation capabilities serve as pointers to operations; they can be passed as variables, thus allowing the creation of arbitrary communication topologies.

4 Design and Implementation Issues

The impetus for developing a portable run-time system grew out of two projects: MkSR and SR on Panda. In MkSR [6, 8] we are investigating how to directly support the SR run-time system on top of the Mach microkernel [1]. Likewise, SR on Panda [23] was a project in which the SR run-time system was modified to run on the Panda [9] portability layer.

In both projects, we started with the original SR implementation, which we refer to as UnixSR. The run-time system for this implementation is monolithic and tightly coupled to a custom, user-level thread interface, to Unix I/O, and to Unix sockets. It became evident that a different RTS structure was required to facilitate implementations for MkSR and SR on Panda as well as future platforms. We sought an approach to portability that would allows us to adapt the SR RTS to a wide range of environments (including virtual machines such as Panda [9], microkernels, and direct hardware). Our observations of the virtual machine approach led to the general principles of the component approach presented in Section 2.2. This section describes several design and implementation issues we encountered while developing our portable run-time system for SR. We are taking an incremental approach to realizing the component approach, and have focused primarily on the thread component of the run-time system.

4.1 The New RTS Structure

In restructuring the UnixSR RTS, we identified the system dependent components. The RTS is now divided into system independent and system dependent components (see Figure 5 [3]). This structure isolates operating system dependent interfaces to threads, network communication, and I/O. This structure does not fully realize the component approach described in Section 2.2 as we still have a separation of system dependent and independent components.

[3] The virtual machines component in the system independent level represents the SR virtual machine language construct, not virtual machines in the sense of Section 2.1

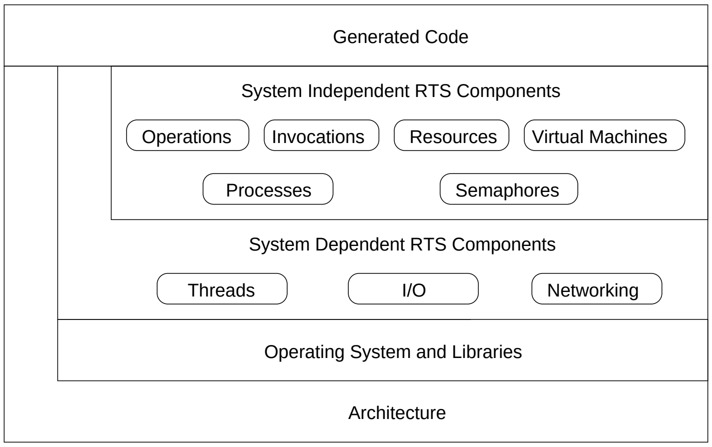

Figure 5: The RTS Structure

4.2 Threads

An SR program may have several processes executing concurrently. In the RTS, an SR process is implemented as a thread. However, threads can have different semantics depending on the thread package or operating system. A portable run-time system must be aware of the differences in thread systems. By examining these differences we have formed an abstract thread interface that can be specialized for specific thread systems. This technique results in a thread component that can be specialized without affecting dependent components. This section describes how we handle different thread characteristics and briefly discusses the issues involved in network and I/O components.

The interface to low-level thread calls is implemented as a set of macros, which are used to avoid procedure-call overhead but do not affect the basic structure of the thread component. For example, the `PROC_CREATE(pr)` macro takes a pointer to an internal SR process structure and creates a thread based on the information in the structure (i.e., function address, arguments, stack[4] , etc.). In many cases the `PROC_CREATE` macro can be simply implemented as a thread creation function call supplied by the thread system. For example, with MIT PThreads `PROC_CREATE` is defined as follows:

```
#define PROC_CREATE(pr) do { \
  pthread_t tmp_pt; \
  pthread_create(&tmp_pt, &pt_attr_default, \
                (start_routine) sr_intermediate_pr, (void *) pr); \
} while(0)
```

For other thread interfaces, `PROC_CREATE` may have to be more complicated, or perhaps implemented by a real function. We have also defined macro interfaces for mutex variables, condition variables, thread-specific data, thread priorities, and other thread manipulation functions.

4.2.1 Thread Scheduling

An important aspect of a thread system is whether or not it schedules threads preemptively. The presence of preemption or true multiprocessing requires the RTS to be reentrant. That is, internal RTS state must be protected by critical sections so that two or more threads executing in the RTS do not interfere. In contrast, using a non-preemptive thread system requires that the run-time system selectively yield thread execution so that other threads can be scheduled.

The UnixSR RTS used a custom user-level thread package with non-preemptive scheduling. However, a variant of UnixSR, called MultiSR, supports true multiprocessing on certain platforms. This version

[4] While some thread packages allocate a stack automatically at thread creation time, some do not. To handle the case in which the RTS must allocate and deallocate the thread stack explicitly we include a stack address field in the process structure.

of SR has locks to protect RTS data structures. Locks are defined as macros. So, if a non-preemptive thread system is used, the lock macros can be defined as empty.

Our portable thread component takes advantage of the MultiSR lock macros. However, we have replaced all blocking and unblocking operations previously based on the custom user-level thread package with mutex variables and condition variables. We modified the internal synchronization interfaces in this manner in order to target the most common synchronization mechanisms found in thread packages and operating systems.

4.2.2 Thread Creation

The way in which a thread is created varies among different thread systems. Usually, the thread creation routine takes a pointer to the function to be executed when the new thread is created and a single void pointer argument, which is passed to this function. Arbitrary data can be passed to new functions using a pointer as the argument to be passed to the function. Some thread creation routines take additional parameters including a stack pointer, the stack size, and thread attributes. Using `PROC_CREATE` (Section 4.2) hides the details of how the thread creation routine is called.

Process creation in the SR run-time system occurs in two phases: a *setup phase* and an *activation phase*. In the setup phase an internal data structure, called a `proc`, is allocated and filled in with appropriate values (i.e., the function to start executing, arguments to be passed, a pointer to the stack, a pointer to the owning resource, and other information). After a `proc` has been setup, another routine is called to activate the proc. The activation routine calls `PROC_CREATE`, but instead of passing the actual function to create as a new thread, an *intermediate function* is passed. Therefore the intermediate function is executed before the creation of every new process. The intermediate function is created with a pointer to the associated `proc` data structure as its argument. The purpose of the this function is to hide thread creation details from the rest of the run-time system and aids in implementing SR's semantics by establishing a common place where internal administration that is required after low-level thread creation can execute.

The intermediate function performs two basic tasks. First, it associates the *proc* structure, passed as an argument, with the new thread. For many thread packages this is accomplished using thread-specific data. The implementation of this association operation is hidden using the `CUR_PROC_SET` macro. Second, the intermediate function calls the actual function to be called designated in the `proc` structure with four arguments, also extracted from the `proc` structure. This calling convention conforms to the code generated by the SR compiler.

There are two reasons for using an intermediate function, rather than calling the function to be created directly. First, it allows the function to be called to have an arbitrary number of arguments. Many thread packages only allow one argument to be passed. Second, it performs the initial association of the `proc` structure to the thread. This association cannot take place inside the newly created function, because these functions are sometimes called directly, and not necessarily as the result of process creation.

It is feasible to eliminate the intermediate function, by having the SR compiler generate functions that only take a single argument, and from this argument extract other arguments as needed. Using this approach would also require the function to be called with a bit specifying whether the function was called as a result of process creation, or if the function was called directly. It is unclear if these modifications would result in more efficient code, because in the case of direct calls, each function would have an additional dereference phase to arguments and an additional conditional statement to determine if the function was called directly.

4.2.3 Thread Identification

Thread identification is needed in the SR run-time system to associate a `proc` structure with each thread. This association is needed to allow threads to determine in which resource they are executing so that threads can access resource variables; and to support certain SR constructs and debugging. In many thread packages, the association can be accomplished with thread-specific data.

4.3 Communication and I/O

We have only begun to develop communication and I/O components for the SR run-time system. While porting SR to Mach and Panda, we encountered different types of communication mechanisms that cannot be hidden as easily from the rest of run-time system as thread interfaces. For example, UnixSR uses a network idle thread to poll for incoming packets. However, the Panda implementation uses an upcall to receive packets and the Mach implementation uses a blocking thread to wait for packets. Each of these networking paradigms requires specialization of the network component. In addition, Panda provides an optimized remote procedure call, which can be used to specialize the higher-level system-independent remote operation invocation code.

Our approach to I/O is to remove any notion of Unix I/O from all RTS components. This enables us to more easily target non-Unix platforms, such as Windows 95, Windows NT[5], Mach [1], and Fluke [12]. A more difficult issue in dealing with portable I/O is whether or not the underlying thread package or operating system supports blocking I/O threads. If the underlying interfaces allow threads to block on I/O operations, then the RTS is greatly simplified. Otherwise, the RTS must incorporate extra bookkeeping to manage threads blocked on I/O. Such management requires involvement from the thread scheduler or possibly an idle thread that is similar to the network idle thread.

5 Implementation Performance

To assess the portability and performance of our new run-time system, we have built implementations using various thread packages on the Linux operating system. These thread packages include SRThreads (a custom user-level thread package based on the thread package used in UnixSR), MIT PThreads (version 1_60 beta6), and LinuxThreads (version 0.4). Each of the these packages support different thread features. Table 1 lists the prominent features of each of the tested thread packages. In the table, UnixSR refers to the custom thread package used by the original SR implementation. The I/O support row indicates whether or not the thread system allows multiple threads to invoke I/O operations without blocking the entire program.

Feature	UnixSR	SRThreads	MIT PThreads	LinuxThreads
thread-level	user	user	user	kernel
scheduling	non-preemptive	non-preemptive	preemptive	preemptive
I/O support	yes	no	yes	yes

Table 1: Thread Package Features

By comparing these thread packages we can determine the overhead of our new RTS structure, as well as show the costs associated with using different Linux thread packages.

We also have implementations using CThreads [10] and Panda Threads [9], but we do not provide performance numbers here because the results are not directly comparable to the results for the Linux-based thread packages. These implementations currently run only on architectures that differ from our Linux test machine. In addition, we have ported our run-time system to other platforms such as a DEC Alpha running Linux and a DECstation 5000/240 running Ultrix.

Microbenchmark	UnixSR	SRThreads	MIT PThreads	LinuxThreads
local call, optimized	0.30	0.32	2.38	0.45
inter-resource call, no new process	2.35	3.01	16.51	8.68
inter-resource call, new process	10.04	12.98	115.17	532.00
process create/destroy	9.42	9.66	115.56	541.10
semaphore pair	0.48	0.50	3.37	2.41
semaphore requiring context switch	1.70	1.76	14.72	32.77

Table 2: Process and Synchronization Performance (in microseconds)

Table 2 presents the benchmark results for some uses of key language-level mechanisms. The test machine is a Pentium 166Mhz with 64 MBs of RAM running Linux 2.0.24. The microbenchmarks presented in this section come from a set of benchmarks designed to evaluate the performance of SR's

[5] Windows 95 and Windows NT are registered trademarks of Microsoft Corporation

concurrency mechanisms [4]. The benchmarks were run on a lightly loaded system and the results are the medians of several runs, where each run consisted of 10^6 to 10^8 iterations depending on the benchmark. The benchmarks do not include program startup time or RTS initialization.

By comparing the performance results for UnixSR and SRThreads we see the cost of our new portability structure. Recall that SRThreads is based on the same low-level context switching code used in UnixSR. The portable thread interface adds relatively little overhead, except in process creation. This extra overhead is primarily due to the intermediate function used to start SR processes discussed in Section 4.2.2.

Both MIT PThreads and LinuxThreads perform considerably worse than UnixSR and SRThreads. The performance difference is partially due to the added cost of thread identification and locking. The added cost of thread identification and locking can be seen in the local call benchmark. For UnixSR and SRThreads, the local call requires no locking and thread identification only requires a pointer dereference. However, for MIT PThreads and LinuxThreads, the local call includes the time required to access thread-specific data. In MIT PThreads, a global pointer (thread identifier) is used to access thread-specific data, but access to this data is protected by internal mutex variables. In LinuxThreads, the current thread's stack is used as the thread identifier, but, like MIT PThreads, internal mutex locks are used to protect access to thread specific data.

The combined, added cost of thread identification and the locking of RTS data structures (see Section 4.2.1) can be seen in the inter-resource call, no new process benchmark. Thread identification is needed associate the current thread with its associated process structure on entry to a function because this information is not passed as an argument. In order to make the inter-resource call, the generated code allocates a new invocation block. The allocation pool is protected by a lock. Locking is only required in the preemptive thread packages: MIT PThreads and LinuxThreads. We have not yet determined the individual costs of thread identification and locking for the different thread packages.

Comparing MIT PThreads to LinuxThreads is interesting because both offer preemptive scheduling, but MIT PThreads is a user-level implementation, whereas LinuxThreads takes advantage of thread-support in the Linux 2.0 kernels. Locking and thread identification perform better in LinuxThreads because it uses stack-based identification and has less complicated implementations for thread-specific data and mutex locks compared to MIT PThreads. LinuxThreads performs worse than MIT PThreads for process creation and context switching because the kernel boundary must be crossed for these operations. It is important to note that the added process creation time and context switch time may be worth the ability to utilize true multiprocessors, something that MIT PThreads lacks.

6 Conclusions

In this paper we have presented a new, portable run-time system for the SR concurrent programming language. In particular, we have shown how we have designed a portable interface for utilizing different thread systems. Our interface incurs only a slight performance penalty. We believe such an approach is necessary to facilitate a high-degree of portability across diverse platforms, including different virtual machine layers. Our approach to handling thread interfaces should be applicable to other concurrent programming languages.

We have also reported on the performance of our new run-time system implemented with various Linux-based thread packages. Our results indicate the relative merits and costs associated with a custom user-level thread package, MIT PThreads, and LinuxThreads.

Our future work will include development of communication and I/O components. In addition to using our microbenchmarks for performance analysis, we will also use macrobenchmarks (i.e., applications). We also intend to further pursue the component approach to structuring run-time systems.

Acknowledgments

We thank the anonymous referees for their helpful suggestions. This work is partially supported by the United States National Security Agency University Research Program.

References

[1] M. Accetta et al. A new kernel foundation for UNIX development. In *Proceedings of the Summer 1986 USENIX Conference*, pages 93–112, June 1986.

[2] G. R. Andrews and R. A. Olsson. *The SR Programming Language: Concurrency in Practice.* The Benjamin/Cummings Publishing Co., Redwood City, California, 1993.

[3] G. R. Andrews, R. A. Olsson, M. Coffin, I. Elshoff, K. Nilsen, T. Purdin, and G. Townsend. An overview of the SR language and implementation. *ACM Transactions on Programming Languages and Systems*, 10(1):51–86, January 1988.

[4] M. S. Atkins and R. A. Olsson. Performance of multi-tasking and synchronization mechanisms in the programming language SR. *Software – Practice and Experience*, 18(9):879–895, September 1988.

[5] H. E. Bal, M. F. Kaashoek, and A. S. Tanenbaum. Orca: A language for parallel programming of distributed systems. *IEEE Transactions on Software Engineering*, 18(3):190–205, March 1992.

[6] G. D. Benson and R. A. Olsson. Towards microkernel support for the SR concurrent programmiing language. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 1513–1524, Sunnyvale, CA, August 1996.

[7] G. D. Benson, R. A. Olsson, and R. Pandey. On the decomposition of run-time support for concurrent programming languages. In *Proceedings of the Workshop on High-Level Programming Models and Supportive Environments*, pages 86–94, Honolulu, Hawaii, April 1996. held in conjuction with the 10th International Parallel Processing Symposium (IPPS'96).

[8] G.D. Benson and R.A. Olsson. *The Design of Microkernel Support for the SR Concurrent Programming Language*, chapter 17, pages 227–240. Languages, Compilers, and Run-Time Systems for Scalable Computers. Kluwer Academic Publishing, Boston, MA, 1996. B. K. Szymanski and B. Sinharoy (editors).

[9] R. Bhoedjang, T. Rühl, R. Hofman, K. Langendoen, H. Bal, and F. Kaashoek. Panda: A portable platform to support parallel programming languages. In *Symposium on Experience with Distributed and Multiprocessor Systems IV*, pages 213–226, San Diego, California, September 1993. USENIX.

[10] E. Cooper and R. Draves. C threads. Technical Report CMU-CS-88-154, Carnegie Mellon University, Department of Computer Science, 1988.

[11] B. Ford and Flux Project Members. *The Flux Operating System Toolkit.* Department of Computer Science, Universiy of Utah, 1996. `http://www.cs.utah.edu/projects/flux/oskit/html`.

[12] B. Ford, M. Hibler, J. Lepreau, P. Tullmann, G. Back, and S. Clawson. Microkernels meet recursive virtual machines. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, pages 137–151, Seattle, WA, October 1996. USENIX Association.

[13] I. Foster and K. M. Chandy. Fotran M: A Language for Modular Parallel Programming. *Journal of Parallel and Distributed Computing*, 25(1), 1995.

[14] I. Foster, C. Kesselman, and S. Tuecke. The Nexus approach to integrating multithreading and communication. *Journal of Parallel and Distributed Computing*, 37(1):70–82, August 1996.

[15] M. Haines. On designing lightweight threads for substrate software. In *The Proceeding of the Annual Technical Conference on UNIX and Advanced Computing Systems*, Anaheim, California, January 1997. USENIX.

[16] N. C. Hutchinson and L. L. Peterson. The *x*-Kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, January 1991.

[17] Institute for Electrical and Electronic Engineers. *POSIX P1003.1c, Threads Extension for Portable Operating Systems*, 1995.

[18] L. V. Kale, Milind Bhandarkar, Narain Jagathesan, Sanjeev Krishnan, and Joshua Yelon. Converse: An Interoperable Framework for Parallel Programming. In *Proceedings of the 10th International Parallel Processing Symposium*, pages 212–217, April 1996.

[19] G. Kiczales. Foil for the workshop on open implementation. Technical report, Xerox PARC, 1994. `http://www.parc.xerox.com/PARC/spl/eca/io/workshop-94/foil/main.html`.

[20] X. Leroy. *LinuxThreads*, 1996. `http://sunsite.unc.edu/pub/Linux/docs/faqs/Threads-FAQ/html/`.

[21] D. MacKenzie. *Autoconf.* Free Software Foundation, Inc., 2.8 edition, 1996.

[22] C. Provenzano. *MIT PThreads*, 1996. `http://www.mit.edu:8001/people/proven/pthreads.html`.

[23] T. Rühl, H. Bal, R. Bhoedjang, K. G. Langendoen, and G. D. Benson. Experience with a portability layer for implementing parallel programming systems. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 1477–1488, Sunnyvale, CA, August 1996.

[24] The MPI Forum. MPI: A message-passing interface standard. *The International Journal of Supercomputing Applications*, 8(3/4), 1994.