Compiler to Interpreter: Experiences with a Distributed Programming Language

Robert M. Gebala and Carole M. McNamee
Department of Computer Science
California State University, Sacramento
Sacramento, CA 95819-6021 U.S.A.
{gebalar,mcnamee}@dromio.ecs.csus.edu

Ronald A. Olsson
Department of Computer Science
University of California, Davis
Davis, CA 95616-8562 U.S.A.
olsson@cs.ucdavis.edu

Please send editorial correspondence regarding this paper to Olsson

February 6, 2001

Abstract

One interpretive approach for handling concurrency is to provide an interpreter instance for each executing language-level process. Such an approach has mainly been applied to concurrent implementations of logic and functional languages. This paper describes the use of this approach in constructing an interpreter for an imperative, distributed programming language from an existing compiler and run-time system (RTS). Primary design goals were to exploit the existing compiler to the extent possible as well as to have minimal impact on the run-time system used to support concurrency. We have been successful in meeting these goals. Additionally, performance results show our interpreter's execution times compare favorably to the times required for compilation, linkage, and execution of small programs or programs with a significant number of calls to the RTS; on such programs, our interpreter's performance also compares favorably to that of the standard Java implementation. However, for larger programs and programs with fewer calls to the underlying RTS, the conventional compiler-based implementation outperforms the interpreter implementation. For many distributed programs in which network costs dominate, the performance of the two implementations differ little.

Keywords: Interpreters, concurrent programming languages, concurrent programming language implementation, process communication and synchronization

1 Introduction

Having an interpreter-based, rather than compiler-based, language implementation often simplifies the task of building language tools such as debuggers. For example, a debugger that resides within the same address space as the language translator can share parts of the symbol table. We endeavored to develop a debugger for the SR concurrent programming language [1, 2] and desired to do so within an interpreted language implementation. Unfortunately, the standard SR language implementation (version 2.3) is compiler-based; that implementation is called *stsr* in this paper. Our problem, then, was to develop an interpreter for SR, within the constraints of limited resources. Rather than begin a new implementation effort, we built an interpreter that builds on components of *stsr*.

Our interpreter, called *sri*, implements a significant subset of the SR language. *sri* supports all of SR's language model and mechanisms for concurrency and program distribution, but it does not support all of SR's data types (e.g., multi-dimensional arrays). (This limitation is due only to our constrained resources for building *sri*.) *sri* uses a naive approach to interpretation: it executes a given SR program directly from the parse tree for the program.

The development of and experimentation with *sri* has produced the following notable results.

Code reuse: *sri* reuses much of the existing standard SR implementation; such code reuse reduced greatly the effort to develop the interpreter. *sri* reuses significant parts of the SR compiler, e.g., for parsing and type checking. It does not use the code generation component; instead it includes new code for executing directly from the parse tree. *sri* reuses the entire SR run-time support system (RTS) without change. The RTS provides primitives for process creation, communication and synchronization between processes, input/output, etc.

Approach to Interpretive Concurrency: *sri* provides concurrency by executing each SR process via its own interpreter. These interpreter instances coordinate their activities via a common run-time support system — namely, the same RTS used in *stsr*. Although this general approach of having multiple interpreter instances has been used in concurrent implementations of logic and functional languages [7], it has not been previously used in the implementation of imperative concurrent languages, where interpreter instances tend to interact often and in more complex ways. *sri* supports concurrent execution within a single UNIX process as well as across multiple UNIX processes, possibly distributed across multiple physical machines.

Performance: Despite *sri*'s naive approach to program execution, it obtains good performance — in many cases better than that of *stsr* or that of the standard Java implementation — on small programs and programs where RTS or network costs dominate. Smaller, short-running programs (e.g., student's programs) are the target for *sri*. For larger programs and programs with fewer calls to the underlying RTS, the *stsr* and the standard Java implementation outperform *sri*.

These results and the techniques on which they are based are applicable to other concurrent language implementations, and should be of interest to those developing such implementations from scratch or by extending an existing compiler-based implementation. Besides the above specific results, this paper also touches upon some of the general tradeoffs between compiler-based and interpreter-based implementations of concurrent programming languages.

The rest of this paper is organized as follows. Section 2 provides relevant background on the SR language and its current compiler-based implementation (*stsr*). Section 3 gives a high-level overview of *sri*, focusing on its approach to dealing with concurrency. Section 4 presents the performance of *sri* and compares it with that of *stsr* and that of the standard Java implementation. Section 5 discusses some of the key issues in the design of *sri* and their impact on performance, the effort involved in implementing *sri*, and some limitations of *sri*. Section 6 summarizes related efforts. Finally, Section 7 contains some concluding remarks.

2 Background: SR and its compiler-based implementation

This section presents overviews of the model of computation for the SR concurrent programming and SR's standard, compiler-based implementation. Reference [2] describes SR language mechanisms and their implementation in detail.

2.1 SR Model of Computation

An SR program can execute within multiple address spaces, which may be located on multiple physical machines connected by a network. The SR model of computation allows a program to be split into one or more address spaces called *virtual machines*, or simply VMs. Each VM defines an address space on one physical machine. A VM is created dynamically, and contains instances of *resources*, SR's modular component. Resource instances are created dynamically, and multiple instances may co-exist in a particular virtual machine.

SR provides dynamic process creation, message passing, semaphores, remote procedure calls, and rendezvous through a mechanism called an *operation*. An operation defines a service that must be provided within the declaring resource.

The code that services an operation is located in the body of a resource. The body is split into units called `processes` or `procs`. Processes are created implicitly when the enclosing resource is created. Procs are instantiated when they are explicitly invoked.

Figure 1 summarizes SR's model of computation for an example program executing on two different physical machines. Arrows represent communication paths between processes, which may be executing within the same resource, within different resources in the same shared address space, within resources that do not share the same address space, or even within resources located on different physical machines connected by a network.

2.2 stsr (the standard SR implementation)

The two key components of *stsr* directly available to the user are the compiler (*sr*) and the linker (*srl*). The *sr* compiler translates an SR source program into an annotated parse tree, which the code generator later traverses to produce C code modules. These modules are compiled using a C compiler, and the resulting object code files are linked by *srl* with the SR run-time support system (RTS) and C run-time libraries to form an executable program. The RTS provides mechanisms for managing job servers, run-time memory, resources, and processes. An SR program begins with the implicit creation of the main virtual machine. An instance of the main resource is then created on the main VM. Each VM is implemented using a single UNIX process. Thus, initially, the SR program executes within a single UNIX process. Within that process, multiple SR processes may execute, with the RTS providing support for their pseudo-concurrent execution.

Distributed SR programs (i.e., those that make use of multiple virtual machines) cooperate with a stand-alone execution-time manager, called *srx*. While not really a part of the RTS, *srx* is spawned as a separate UNIX process by the RTS when the program explicitly creates its first VM. Once *srx* has started, other VMs may be created, thus creating new UNIX processes, which may execute on remote physical machines.

3 Overview of *sri*

An interpreter performs traditional compiler functions, such as parsing and type checking. In addition, it executes the source code. We refer to the portion of the interpreter that actually executes the source code as an *executor*.

Our *sri* interpreter is based on the SR (version 2.3) implementation. It combines functionalities of this implementation including the SR compiler, linker, and run-time support (RTS).

sri consists of two major phases:

- building the parse tree. This phase reuses the SR compiler's code for lexical analysis, parsing, symbol-table management, and type-checking.
- executing the parse tree. This execution phase uses newly-written code to walk through and interpret the nodes in the parse tree. It calls routines in the existing RTS code as needed, for example, for interprocess communication and synchronization and for creating new processes. Each process runs its own instance of an executor.

sri first builds the parse tree for the entire program; it then passes control to the executor. Key aspects of these two major phases of *sri* are discussed below.

Of special significance is that, internally, the RTS is unaware of whether it is executing as part of *sri* or as part of an executable linked with code compiled by *sr*. It simply services requests made via its normal interfaces [11].

3.1 Parsing and Symbol Table

Two key *sri* front-end activities are building a parse tree and building symbol tables. These use existing *sr* code, mostly without change.

To illustrate *sri*'s parsing phase, consider the SR expression `read(a, b)`. *sri*'s parsing phase builds the parse subtree shown in Figure 2. The left tree of the `O_LIBCALL` node is an `O_SYM` node that gives the name of the predefined function being called by the program. The right subtree is an `O_LIST` node, which describes the arguments to the function.

One key change made to the SR compiler code involves the deallocation of each resource's parse tree and symbol table. *stsr* compiles the source code one resource at a time, freeing the parse tree and symbol table after generating the target code for the resource. *sri*, on the other hand, interprets the entire source program, thus it needs to retain each resource's parse tree and symbol table.

3.2 Program Execution

Figure 3 provides an overview of how *sri* executes SR programs. An instance of the *sri* interpreter runs for each virtual machine in the program; two such instances are indicated in Figure 3. Each instance runs as a separate UNIX process; the two processes run concurrently. The instances of the *sri* interpreter communicate via messages exchanged by their run-time support systems (RTSs).

As indicated in Figure 3, each instance of the *sri* interpreter supports multiple SR processes through its RTS, thus providing concurrency within the VM. In *sri*, the code within each instance of an SR process is executed by its own instance of the executor. The executor traverses the program's parse tree, taking appropriate actions for each node it encounters. Executor procedures call RTS routines to, for example, invoke operations and create virtual machines. Thus, the stack for each SR process contains frames for both executor procedures and RTS procedures (described further in Section 3.2.1). The RTS provides context switching between processes, and interprocess communication and synchronization.

Below, we discuss in more detail the *sri*'s executor and how it interacts with the RTS.

3.2.1 Executor

The executor's general approach is very simple: It traverses the parse tree, using node type and the results of node evaluation to determine the next node to execute. The executor is organized as a collection of procedures, each one implementing all or part of some language feature(s). Almost every parse tree node is handled by a set of procedures. As an example, an invocation of a typical predefined SR function will involve at least three procedure calls: a call to an executor routine to handle the library call node; a call to an executor routine to collect the parameters expected by the predefined function (which in turn may involve more calls to evaluate each actual parameter); and a call to the RTS routine that performs the actual work.

The top-level executor procedure, `h_node`, takes as its only argument a pointer to the node in the parse tree at which it should begin executing. It then invokes the executor procedure that handles the particular node type of its argument.

To illustrate *sri*'s executor phase, consider again the parse subtree shown in Figure 2. After parsing, *sri* transfers control to its executor phase by invoking `h_node`. When the executor encounters the `O_LIBCALL` node, it invokes an executor routine that uses a table to map the string in the `O_SYM` node (i.e., "read") to the appropriate *sri* routine, `h_sr_read`. The executor then invokes `h_sr_read`. The `h_sr_read` routine collects arguments — i.e., the names of the variables whose values will be read — and finally invokes the RTS `sr_read` function to actually read the values from the standard input.

Because executor procedures invoke RTS routines, stack frames for both the executor and RTS routines will be stored on the stack of the currently executing process (*CEP*). The stack frame for each executor routine includes the arguments passed to the routine. These arguments include pointers to: the parse tree node where the *CEP* is currently executing; the resource parameters and resource-level variables for the resource instance containing the *CEP*; and the process parameters and local variables for the *CEP*'s process instance. The "pointer to the next node to execute" is kept implicitly within this stack by the node pointers that are passed to the executor procedures and local variables within those procedures. Because of the close connection between the parse tree and the stack, the stack can become deep. For example, SR's grammar uses right recursion in its rule for a sequence (block) of statements. In the resultant parse subtree, the list of statements is represented through the nodes' right pointers. During execution of the last statement in the sequence, the stack will contain stack frames for the executor routine `h_seq` for each statement in the sequence.

Variables are organized in variable tables. When the executor encounters a new block of code, it allocates a new list of variables that appear in that block. These lists are linked together to form a stack of lists, called the *variable table*, representing variables local to the process. The executor searches this table according to normal scoping rules to locate variables as needed. Note that the resource variables, operation structures, and process variable tables are created dynamically, reflecting the dynamic nature of resources and processes.

3.2.2 Run-Time Support System (RTS)

As noted in Section 3, the same RTS is used in *sri* as in *stsr*. The *sri* executor code invokes routines in the RTS. The RTS supports processes, context switching between processes, and interprocess communication and synchronization. The RTS provides a context for each process, i.e., a stack consisting of stack frames for procedures from the executor and RTS, as described in Section 3.2.1.

The rest of this section describes key interactions between the executor and the RTS: how processes are created, how virtual machines are created, and how program execution is begun.

The executor initiates the creation of a new process by calling the RTS routine `sr_invoke`, passing it an invocation block containing the parameters for the process as well as a pointer to the node that is the root of the parse subtree for that process. The RTS creates a new SR process, giving it a context and passing it a pointer to the invocation block. When scheduled by the RTS to execute, this new process will begin execution in the executor routine `interp_new_process`, which will extract the node pointer for the invoked proc from the invocation block. After allocating a variable table for the process, `interp_new_process` begins executing the actual code in the proc by passing control to the `h_node` executor procedure. Thus, as noted earlier, each SR process is run by its own executor.

The executor initiates the creation of a new virtual machine by calling an RTS routine. The RTS in turn performs the same actions as described in Section 2.2: It creates a new UNIX process on the specified machine. This new UNIX process runs its own copy of *sri* (see Figure 3) and waits for a request for resource creation. Upon receiving such a request, the executor will execute the initial code of the newly created resource.

Actual execution of the user's program begins when *sri* calls the RTS's main procedure. At this point, program startup is driven by the RTS, as in *stsr*. For *sri*, a special SR resource is predefined. After the RTS's main procedure initializes the RTS data structures, it creates an SR process to execute the special SR resource, the executor's generic resource creation procedure. This all-purpose resource initial code builds — based on the symbol table information for the resource — a structure for the resource variables and operations for an instance of the main resource. It then passes control to the `h_node` executor procedure to begin execution of the main resource's initial code.

The initial code of the main resource typically creates additional processes. These processes are created as described above.

4 Performance

This section discusses the execution time performance of *sri*. Section 4.1 compares *sri*'s execution time performance with that of *sri* on the same SR programs. Section 4.2 then compares *sri*'s execution time performance on those SR programs with that of a standard Java implementation on similar Java programs.

4.1 *sri* vs. *stsr*

We compare the execution time of *sri* with the sum of the execution times for *stsr*, including the compiler (`sr`), the linker (`srl`), and the executable (`a.out`). In many educational or program development environments, a user's perspective of run-time performance includes the time spent compiling, linking, loading, and executing a program. SR was designed to be used in such an environment and as such this is a reasonable assumption. However, this assumption is clearly not reasonable for all environments, particularly production environments.

We use the UNIX `time` command to determine the user and system times for *sri* and the user and system times required for *stsr* to compile, link, and execute to run several different SR programs. All programs contain less than 100 lines of code. Our tests were performed on a Sun SparcStation running SunOS, an HP 9000 running HP/UX, and a PC running Linux. Results were consistent across all platforms. All timing results reported in this paper are taken from a 100 MHz HP 9000/715 and include both user and system time (e.g., time spent in the operating system allocating new memory to a process). We give results for two programs, whose execution behaviors represent two extreme cases: a compute-intensive program and and an RTS-intensive program. These results, however, are sufficient to explain the execution behaviors of other programs run using *sri*, since other programs combine compute-intensive and RTS-intensive activities. In addition to the timing results, we also comment on the memory requirements of *sri*.

To illustrate *sri*'s run-time performance, consider the compute-intensive code fragment shown in Figure 4, which uses a bubble sort to order an array `A` of size `N`. Figure 5 shows a comparison of the amount of time spent in user and system mode by *sri* and `sr+srl+a.out` for increasing values of `N`.[1] For short executions, i.e., for small values of `N`, the costs of compiling and linking are much greater than the cost of actual program execution; for longer executions, the overhead associated with *sri*'s implementation eventually dominates. For this program run on the HP 9000/715 system, the crossover point between which implementation performs better occurs for a value of `N` between 200 and 300. In general, the location of the crossover point depends on the particular program and particular system on which it is run.[2] The crossover point on the Sun occurs at about 500 and on the PC occurs at about 200.

In general, *sri* outperforms the *stsr* for short-running programs because *sri* runs as a single UNIX process whereas *stsr* runs as several UNIX processes: one for each part of `sr+srl+a.out`, and others for compiling the generated C code and for link editing to form the executable program. Each of those processes needs to be created, requires memory for it to be allocated, and performs I/O; e.g., the C compiler reads the C source file and writes an assembly source file. In contrast, *sri* is a single

[1]The system times for *sri* are relatively small for this program and so are barely visible in this figure.

[2]For some programs run on some systems, it is possible that no crossover point exists (e.g., if one implementation always outperforms the other) or that multiple crossover points exist (e.g., if program execution times do not increase as smoothly as they do in this example).

process and, for example, does little I/O (it only reads the SR source file and performs any I/O specified in the user's program). These differences are reflected in the system times seen in Figure 5. The compiling and linking activities of `sr+srl+a.out` also require additional computation; e.g., the C compiler parses its input, builds a symbol table, etc. The cost of these activities is reflected in the user times seen in Figure 5. After some point (N just before 200), though, the cost of *sri*'s interpretation overtakes these costs.

Within *sri*, the amount of time spent on parsing is negligible. *sri* 's executor spends most of its time on symbol table lookup, table creation for variables, and node walking. Expression evaluation is also relatively costly in *sri*. To get the r-value of an integer variable, for instance, at least three executor calls are involved: one to handle expression evaluation, a table lookup on the variable, and a call to fetch the value of the variable. (Adding support for other SR data types would no doubt cause *sri* to run slower. The executor would have to perform more tests to distinguish instances of polymorphic functions, such as addition, and it would have to distinguish the data type of each operand, performing type conversions when necessary.)

On the other hand, *sri* performs closer to *stsr* on RTS-intensive programs because these programs spend most of their time in the RTS rather than in node walking or symbol table lookup. That is, both implementations are executing more frequently in the RTS code, which is the exact same code; what differs is how that code is invoked: from generated code or from within the interpreter.

As an example, consider the RTS-intensive program implementing barrier synchronization as shown in Figure 6. A coordinator process synchronizes `NW` worker processes `N` times. The corresponding timing comparison, in Figure 7, shows that *sri*'s execution time compares more favorably with *stsr*'s than for the compute-intensive program sort of Figure 4. The crossover point for the RTS-intensive program is between 1,500 and 2,000, considerably higher than that for the example compute-intensive program seen earlier.

Although our primary focus in performance was the overall time for compiling, linking, and executing, we also measured separately the execution times of our benchmark programs. Table 1 shows the breakdown for the sort program and Table 2 shows the breakdown for the barrier program; for comparison, both tables include the *sri*'s total translation and execution costs. The costs for compiling and linking for *stsr*, which are constant per program, are the dominant costs in the execution of these programs over the range of the specified N.

For many distributed programs that use a fair amount of communication, such as client/server programs, timing tests reveal no significant difference between the times for programs executed via *sri* and programs executed via the *stsr* implementation. Not surprisingly, network costs dominate these executions.

sri executes the sort and barrier programs described above using SR's standard memory allocation and stack size. However, in general, *sri* requires more memory than *stsr* since the executor needs both compile-time structures, parse trees and symbol tables, and run-time variable tables and other data structures. SR processes, within *sri*, also require larger stack sizes to accommodate stack frames for RTS and interpreter routines. Recall from Section 3.2.1 that when executing the last of a sequence of program statements, a process stack will hold, at least, one executor procedure stack frame for every sequential statement in the sequence.

To illustrate these higher memory needs and larger stack requirements of *sri*, consider the program fragment in Figure 8. On both Sun SPARCstation-5/SunOS and HP/UX 9000, increasing the stack size by 4K allows the number of invocations of `sum` to be increased by 16 for *stsr*, but only by 2 for *sri*.

4.2 *sri* vs. Java implementation

We wrote Java programs similar to the SR programs in Section 4.1. This section compares the performances of *sri* and the standard Java implementation.

We used the same testing methodology and systems described in Section 4.1. For the Java implementation, though, we measure the sum of the translation time (`javac`) and execution time (`java`).

Figure 9 compares *sri* on the SR sort program (Figure 4) with the standard Java implementation on a similar sort program written in Java. As the figure indicates, *sri* outperforms the Java implementation for values of `N` up to about 500. The reason is that the Java implementation spends most of its time (about 8.38 seconds) on translation (`javac`). Its execution time is relatively low, both as a part of the overall time and compared to *sri*'s time.

Figure 10 compares *sri* on the SR barrier program (Figure 6) with the standard Java implementation on a similar barrier program written in Java. The Java barrier program uses native Java synchronization (a synchronized procedure, a condition variable, and `wait` and `notifyAll`). As the figure indicates, *sri* outperforms the Java implementation for values of `N` up to about 11,500. Again, the Java implementation spends considerable time (about 9.41 seconds) on translation (`javac`). The crossover point is higher here than for the sort programs because *sri* spends most of its time executing in RTS code rather than interpreting parse tree nodes; the Java implementation spends much of its time executing similar actions and not as much time interpreting byte code (where it has the advantage over *sri*).

The data in Figures 9 and 10 for the Java implementation are without the JIT (just-in-time) optimization. As one might expect, the Java implementation performs worse with JIT than with no-JIT for short-running programs. For example, for the barrier program with `N` less than about 2,000, no-JIT is better because the additional overhead for actually performing the JIT optimization is greater than any improvement it obtains.

5 Discussion

Our approach to handling concurrency (Section 3.2.2) provides an interpreter instance for each executing language-level process. These interpreter instances coordinate their activities via a common RTS (run-time system). A significant advantage of this approach is that it allows us to reuse the same RTS used in *stsr*. An alternative approach would be to integrate the concurrency within the interpreter itself. (This alternative approach is used in some functional/logic interpreters; see Section 6.) The interpreter would coordinate the executions of multiple processes, i.e., it would provide context switching and interprocess communication. In this approach, the preexisting RTS could not be used directly. It would need to be combined with the other parts of the interpreter; we believe doing so would have required significantly more development effort.

The development of *sri* took one graduate student about six months, with about two months of work from others. The additions to the interpreter consisted of approximately 5,000 lines of code. To put that number in perspective, it represents about 30% of the size of the RTS and 25% of the *stsr* compiler.

Recall *sri*'s naive approach to execution, which traverses the parse tree invoking interpreter procedures at each node. An alternative approach is to generate some form of intermediate code and then execute that code. Execution would be, in many cases, dramatically faster. For example, the costs involved in walking the parse tree (such as procedure invocation) would disappear. Also, the variable tables could be organized as hash tables instead of linked lists. In addition, the parse trees and symbol tables would no longer be needed during execution, thereby freeing a significant amount of memory. Using intermediate code could also facilitate separate translation, which would be useful for larger programs. Given that we intended *sri* to be used mainly on small programs, this possible advantage was not compelling.

Given our limited resources, we chose the naive approach for its simplicity of implementation. Rather than have to develop an intermediate code format and modify the compiler to generate intermediate code, we focused our efforts on the more central issues, e.g., how to reuse the RTS.

As noted in Section 1, *sri* does not currently fully implement all SR mechanisms, because we focused our effort on the features for interprocess communication and synchronization, and on the mechanisms for distributed programming. The most significant limitation of *sri* is that it supports a limited collection of data types: integers, semaphores, virtual machine capabilities, resource capabilities, and operation capabilities. (A capability is essentially a pointer.) It also supports 1-dimensional arrays. *sri* does not support multidimensional arrays, records, strings, characters, or reals. (As noted in Section 4.1, adding these features would affect *sri*'s performance.) In addition, parameters for resource creation and operation invocation must have integer or operation capability type and must be passed by value. Only a subset of SR's predefined functions are currently implemented.

The way in which *sri* executes causes each VM to reparse the source file, and to build the parse tree and symbol table. This reparsing is relatively fast, but it may generate multiple warnings for errors in the source code. This reparsing is also susceptible to a race condition with multiple *sri*'s (which can be active at the same time) overwriting intermediate files. This only happens for multiple VMs executing on the same physical machine or on different physical machines using

NFS. This phenomenon has not been observed in practice, including many runs of a small *sri* validation suite. One solution would be to pass the parse tree to other VM's. However, doing so is not straightforward in a non-NFS environment. Other possible solutions would be to include virtual machine identifiers with their intermediate file names or to execute from intermediate code representations (as mentioned above).

As mentioned in Section 3, *sri* uses without change much of the SR compiler's front-end. A few code sections use `#ifdef`'s to specialize the code for *sri* to handle arguments typically passed to the linker and the executable, and to invoke interpreter routines for resource registration and initialization. *sri*'s executor, however, is new code. The only notable change to the RTS is small: *sri* must pass command line arguments to the RTS main routine to support the execution of distributed programs.

The use of an interpreter necessitated the following change in the use of command line arguments. *sri* separates command line arguments into three categories: those that involve the input source file names, those that involve linking (e.g., process stack size) as marked by the `-l` option, and those that are passed as arguments, via the `-x` option, to the executing program. One notable link-time option that *sri* does not support is the use of libraries. For example, standard SR allows libraries for SRWin, an interface to the X-window system, and other C code to be linked into the executable program the linker creates. That is not possible, unless *sri* were to employ dynamic linking [6]. Combining the parsing and execution phases of SR programs into one phase within an interpreter means that error messages from two phases (such as warning messages in parsing and error messages in execution) will be combined. To let the user better handle this situation, *sri* provides a new command line option that redirects the parser's error messages to a file.

6 Related Work

Widely used interpreted languages such as Lisp [10], Icon [5], and Pascal [14] — list and string processing, and a typical imperative language, respectively — have implementations that differ substantially from *sri*'s. While both Lisp and Icon interpreters have large underlying run-time support systems, they differ from *sri* in that the latter is designed for a distributed programming language where process interaction is the primary concern of the RTS. Sequential Pascal interpreters have a relatively simple design and requires no need for an extensive RTS. More recently, Java [3] has emerged as an interpreted language with features facilitating distributed computing. Like SR's implementation, Java's implementation requires an extensive RTS; however, unlike SR's implementation, Java's primary implementation is interpreter-based [4]. Standard Java implementations translate Java source code to Java byte code, which is then interpreted by the Java virtual machine. Section 4.2 discussed some performance tradeoffs between this approach and *sri*'s naive approach.

CCAL [8] is an interpreted distributed programming language as an experiment with high-level programming language concurrent control abstractions. CCAL provides no control regime to the user, and is primarily used for prototyping application-specific control forms.

Reference [9] presents an overview of several interpretation techniques and their performance characteristics. *sri* is an example of a Type-2 system: source code is compiled into a high-level intermediate code — a parse tree in *sri*'s case — that is then interpreted. Type-2 systems provide the efficiency of a compiler for the source-language translation and the flexibility of an interpreter for execution.

Reference [12] describes a technique for converting interpreters into compilers. Not surprisingly, these compilers consistently outperform the interpreters from which they are created. However, for languages that are heavily dependent upon run-time support systems, this improvement is less dramatic; it is an artifact of the ratio of time spent interpreting with time spent in the run-time system. This paper describes the inverse process (i.e., converting a compiler into an interpreter) and shows performance results that are consistent with the above.

Reference [13] also observes that interpreters for languages that rely upon run-time systems compete more favorably with their compiler counterparts. This observation is based upon experiences with an interpreter for a process-oriented simulation language with a large run-time support system.

7 Conclusion

sri successfully implements a significant subset of the SR concurrent programming language, including concurrency, interprocess communication and synchronization, and mechanisms for distributed programming. To implement concurrency within *sri*, each SR process is executed via its own instance of an interpreter. Each interpreter executes directly from a parse tree representation of the code. Recall that one of the design objectives was to reuse as much of the existing compiler and RTS code as possible. Much of the interpreter code for handling parse tree nodes was derived from the code generator module. It would be interesting to see if we could 'reverse engineer' the code generator to produce an interpreter. If we were to start from scratch to build both a compiler-based language implementation and an interpreter-based language implementation, we would try to build the compiler's code generator and the interpreter's executor from a common base. We would also have the implementations share a common RTS, as we have done with *stsr* and *sri*.

As shown in Section 4, for small problem sizes, *sri* outperforms *stsr* and *sri* outperforms the standard Java implementation. Due to its naive approach to interpretation, *sri* performs better on RTS-intensive programs than it does on compute-intensive programs. For many distributed programs where network costs dominate, the costs associated with the network dominate and the times associated with execution via *sri* and *stsr* are indistinguishable. *sri* is also appropriate in cases where the source program needs to be modified and executed frequently, such as in the initial stages of the program development, or as in typical student programs.

Acknowledgments We thank Robert Keller for helpful comments on this work, Janine Taylor for her efforts in adapting the standard SR validation tool to work with *sri*, and Greg Benson for discussions and the Java barrier code. We also thank the anonymous referees for their many thoughtful comments, which helped us to improve the content and presentation of this paper.

References

[1] 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.

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

[3] G. Cornell and C. S. Horstmann. *Core Java*. Sun Microsystems, Inc., Mountain View, CA, 1996.

[4] J. Gosling. Java intermediate bytecodes. *ACM Sigplan Notices*, 30(3):111–118, 1995.

[5] R. E. Griswold and M. T. Griswold. *The Icon Programming Language (Third Edition)*. Peer-to-Peer Communications, San Jose, CA, 1997.

[6] W. Ho and R. A. Olsson. An approach to genuine dynamic linking. *Software—Practice and Experience*, 21(4):375–390, April 1991.

[7] P. Kacsuk. *Execution Models of Prolog for Parallel Computers*. The MIT Press, Cambridge, MA, 1990.

[8] P. Kearns, C. Cipriani, and M. Freeman. CCAL: An interpreted language for experimentation in concurrent control. In *Proceedings of the 1987 Conference on Interpreters and Interpretive Techniques*, pages 283–291, St. Paul, Minnesota, June 1987.

[9] P. Klint. Interpretation techniques. *Software — Practice and Experience*, 11:963–973, November 1981.

[10] J. McCarthy. LISP: A programming system for symbolic manipulations. Report, ACM Annual Meeting, Extended Abstract, Cambridge, Mass., September 1959.

[11] A. Morgenstern and V. Thomas. The SR run-time system interface. Technical report, The University of Arizona, Department of Computer Science, August 1992.

[12] F. Pagan. Converting interpreters into compilers. *Software — Practice and Experience*, 18(6):509–527, June 1988.

[13] R. Rozin and S. Treu. A hybrid implementation of a process-oriented programming language for system simulation. *Software — Practice and Experience*, 21(6):557–579, June 1991.

[14] N. Wirth. The programming language Pascal. *Acta Informatica*, 1(6):35–63, 1971.

Table 1: Breakdown of *stsr* costs for the sort program

	$N = 50$	$N = 1,000$
stsr execution	0.03	1.17
stsr compilation and linking	2.16	2.16
stsr total	2.19	3.33
sri total	0.16	38.74

Table 2: Breakdown of *stsr* costs for the barrier program ($NW = 10$)

	$N = 100$	$N = 12,000$
stsr execution	0.05	2.25
stsr compilation and linking	2.74	2.74
stsr total	2.79	4.99
sri total	0.29	24.16

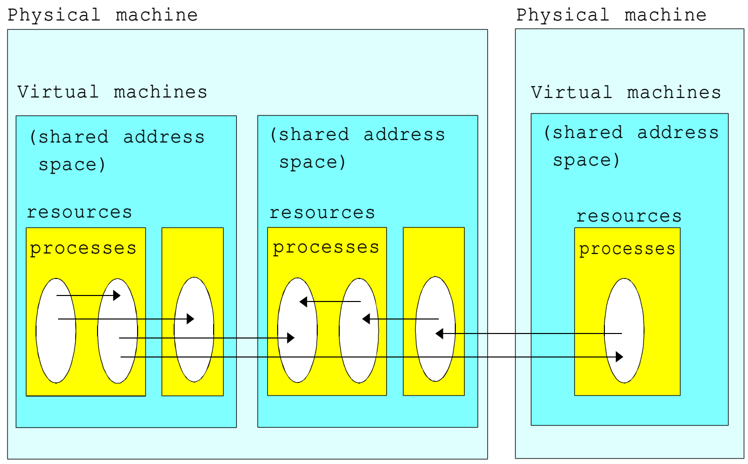

Figure 1: SR model of computation.

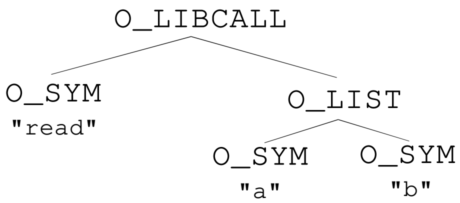

Figure 2: Parse subtree for the statement `read(a, b)`.

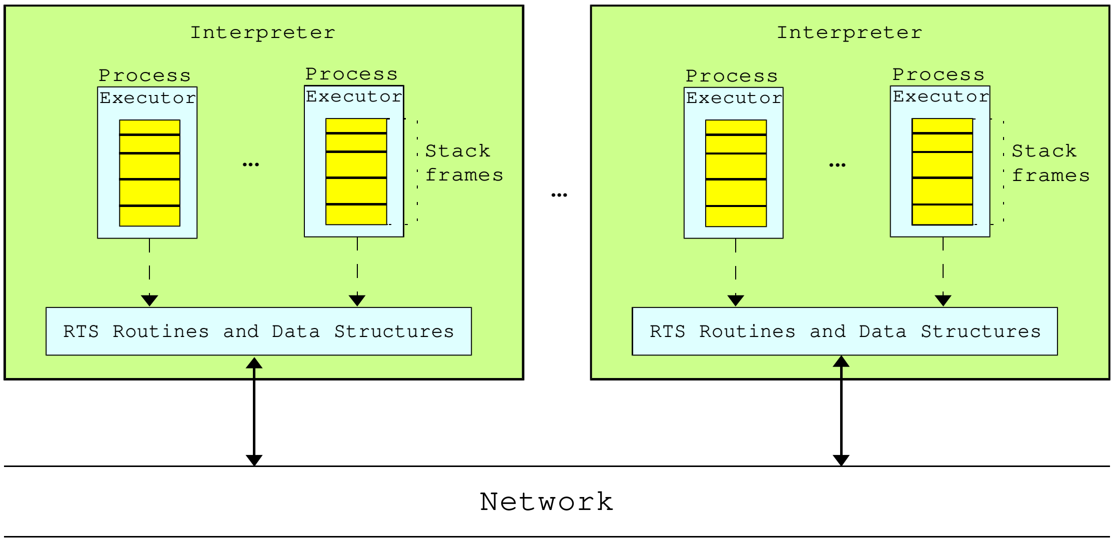

Figure 3: Overview of *sri* execution model.

```
        # A has already been initialized
fa i := 1 to N-1 ->
   fa j := 1 to N-1 ->
      if A[j] < A[j+1] -> A[j+1] :=: A[j] fi
   af
af
```

Figure 4: A compute-intensive sorting fragment

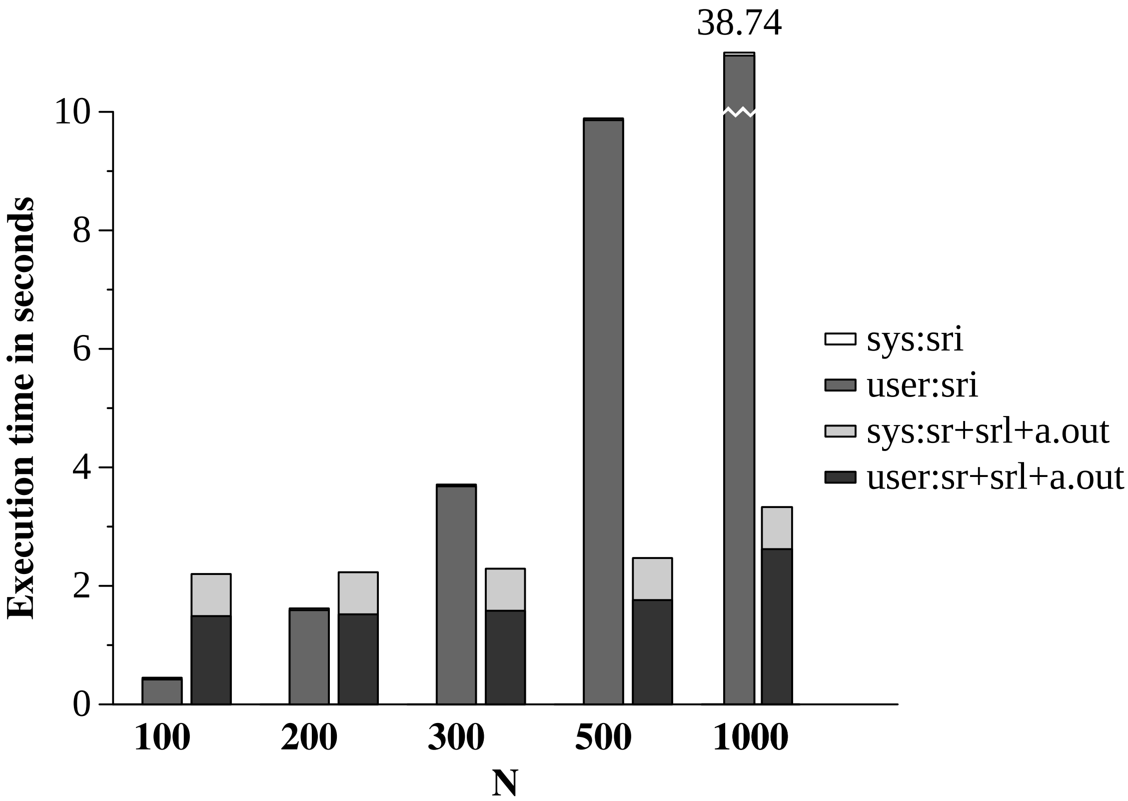

Figure 5: Timing comparison of *sri* vs. *stsr* for the sort program (Figure 4).

```
process worker(w := 1 to NW) #NW worker processes
   fa i := 1 to N ->
      # do work
      V(done)
      P(continue)
      nap(0)    #forces context switch for this example
   af
end

process coordinator
   fa i := 1 to N ->
      fa w := 1 to NW -> P(done) af
      fa w := 1 to NW -> V(continue) af
   af
end
```

Figure 6: An RTS-intensive barrier synchronization code fragment

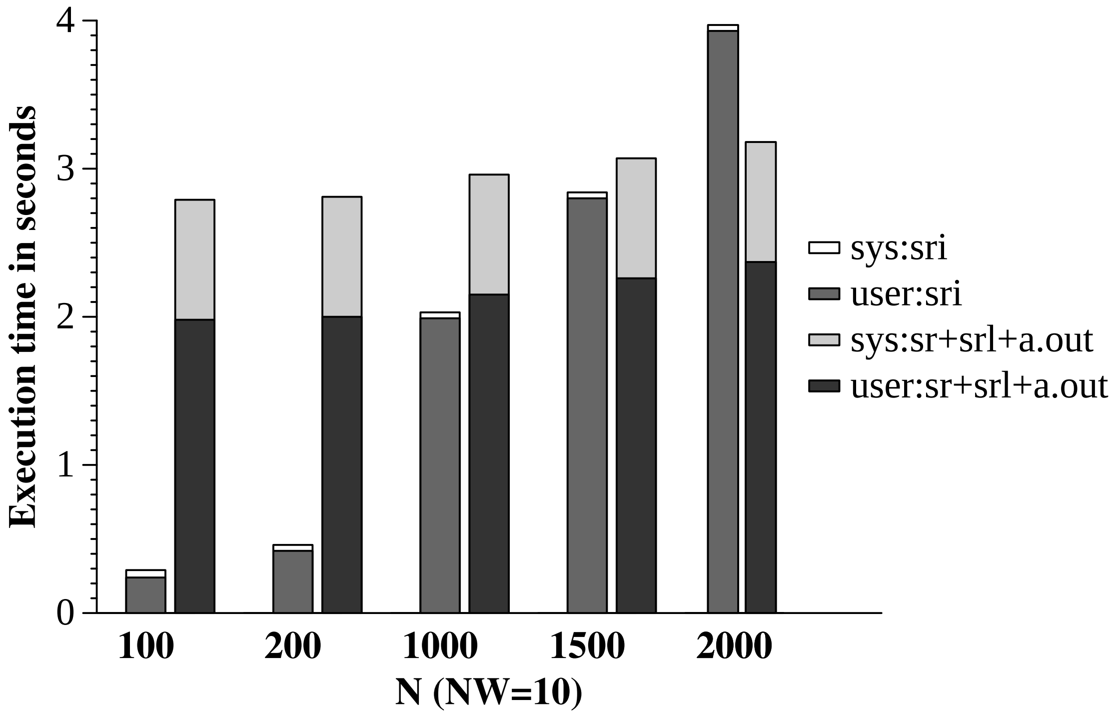

Figure 7: Timing comparison of *sri* vs. *stsr* for the barrier program (Figure 6).

```
proc sum(v) returns r
  if v > 1 -> r := v+sum(v-1)
  [] v = 1 -> r := 1
  [] else  -> r := 0
  fi
end
```

Figure 8: A recursive SR program fragment used to compare *sri* vs. *stsr* memory requirements.

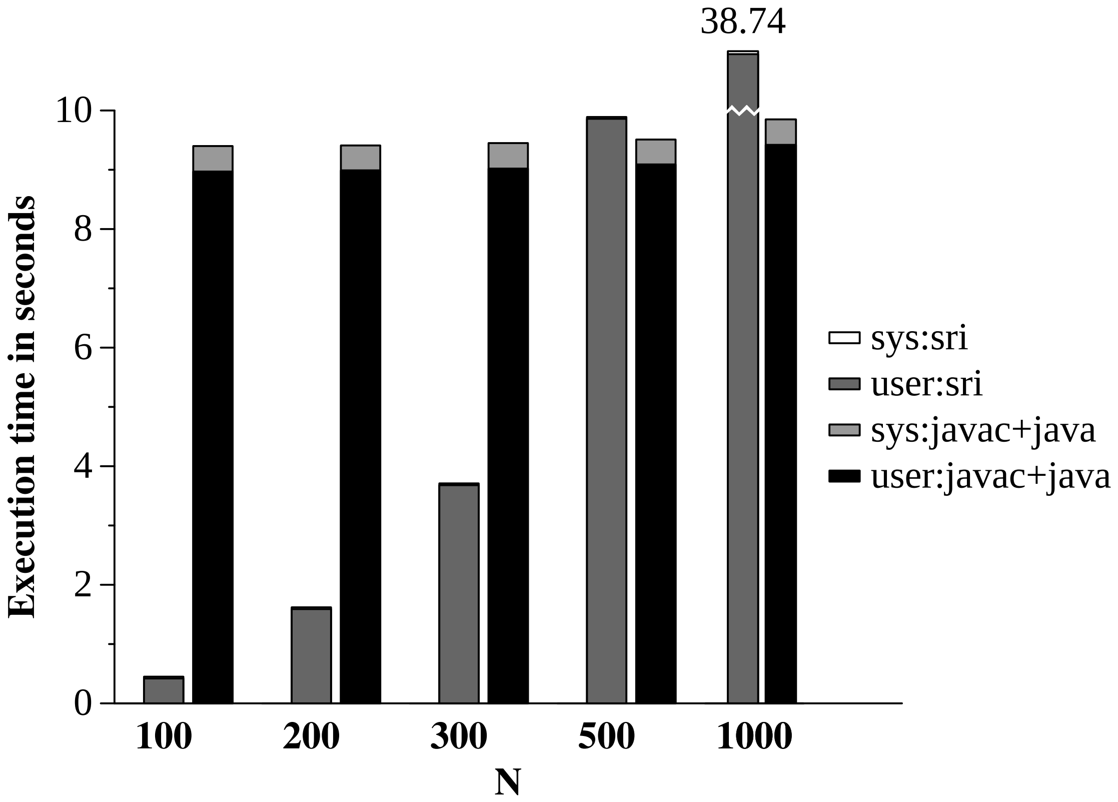

Figure 9: Timing comparison of *sri* vs. Java implementation on similar sort programs.

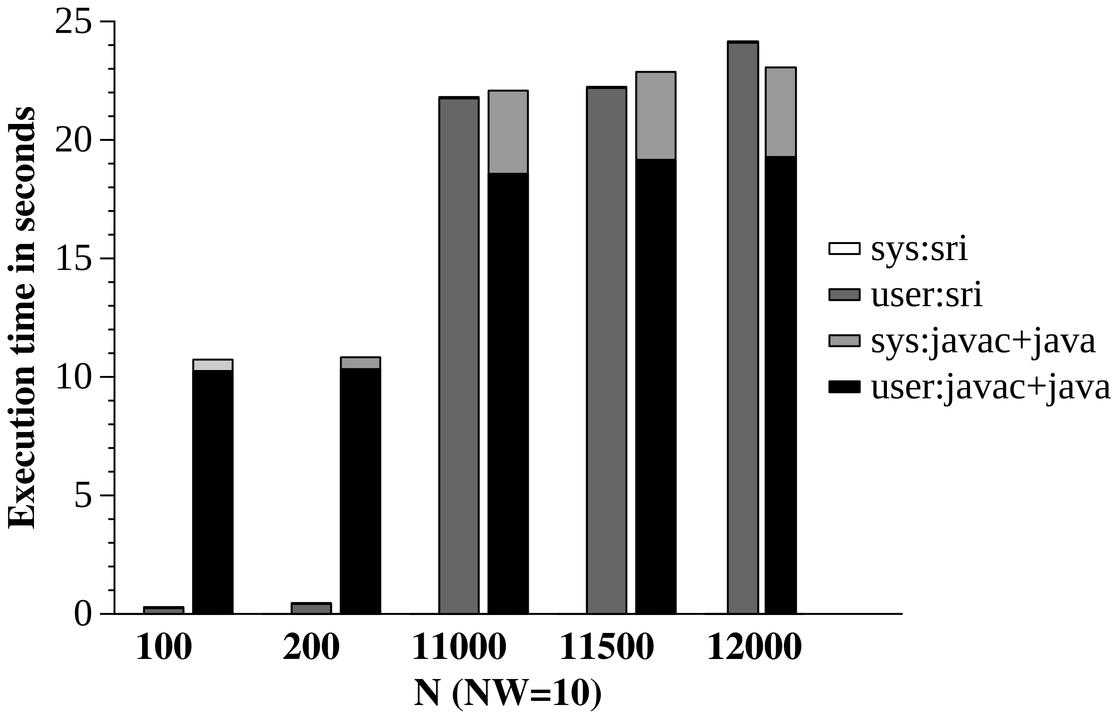

Figure 10: Timing comparison of *sri* vs. Java implementation on similar barrier programs.