Arachne: A Portable Threads System Supporting Migrant Threads on Heterogeneous Network Farms

Bozhidar Dimitrov, *Student Member*, *IEEE*, and Vernon Rego, *Member*, *IEEE*

Abstract—We present the design and implementation of Arachne, a threads system that can be interfaced with a communications library for multithreaded distributed computations. In particular, Arachne supports thread migration between heterogeneous platforms, dynamic stack size management, and recursive thread functions. Arachne is efficient, flexible, and portable—it is based entirely on C and C++.

To facilitate heterogeneous thread operations, we have added three keywords to the C++ language. The Arachne preprocessor takes as input code written in that language and outputs C++ code suitable for compilation with a conventional C++ compiler. The Arachne runtime system manages all threads during program execution. We present some performance measurements on the costs of basic thread operations and thread migration in Arachne and compare these to costs in other threads systems.

Index Terms—Heterogeneous thread migration, user-level threads, compile-time code transformations, C++.

1 Introduction

Threads enable cheap and highly effective forms of concurrency in software executions. They may be viewed as miniprocesses—each equipped with only a program counter and a stack—that are not directly associated with page tables, file descriptors, signals, and timers. All control information required for a thread's management (i.e., id, arguments, program counter, stack pointer) is kept in a *thread control area* (TCA) within the thread. The cost of creating a user-space thread is small: the initialization of space, control information, and the invocation of an associated *thread function*. Many threads may run within a single process and share its CPU time-slice.

In systems that support threads, concurrency can be had through the creation of two or more communicating threads. Despite programming complexities induced by concurrency, it is our experience that the potential benefits of threads-based concurrency amply justify the extra care required in its use. Programming difficulties stem mainly from data sharing, race conditions, and the need for data protection. Our motivation for designing and building threads systems lies in their ability to deliver requisite functionality and performance in different applications. In particular, we have significant experience with the use of threads in distributed simulations [13], [10], [16] and network protocols [6], [5], [7]. Within the class of discrete-event simulation paradigms, for example, it is generally accepted that process-oriented simulations (e.g., CSIM [23], Si [20]) offer the simplest programming model [1], and are surprisingly easy to construct using threads [22]. Indeed, users need never know that powerful application-level constructs, provided in a domain-layer above a simulator kernel, are actually implemented in terms of threads.

On large distributed-memory systems, multithreading offers untold promise. Some inkling of this potential can be found in [14], where four diverse applications are presented in the context of the Ariadne threads system [15]. While Ariadne supports threads on shared-memory, and—with the help of communication libraries—also on distributed-memory machines (e.g., Intel Paragon, workstation clusters), its threads can only migrate between homogeneous platforms. The examples provided in [14] include adaptive quadrature for numerical integration, a parallel quicksort on shared-memory, a linear solver based on SOR (successive over-relaxation), and a particle physics application on distributed-memory. The examples in [14] show how threads support, and in particular thread migration, enables a simpler distributed-programming model than the standard single-threaded model with send and receive primitives.

In the general case, there are many questions yet to be dealt with, particularly in regard to the semantics of threads-based messaging [4], [7] and the structure of computations with distributed-communicating threads. Regardless of how these questions are eventually addressed, there are two factors that significantly impact upon the runtime performance of large systems of distributed threads: *load-imbalance* and *nonlocal data access*. If processors operate on unequal loads, efficiency falls; if threads make frequent and repeated accesses to nonlocal data, messaging overhead rises. Both lead to an execution-time increase and poor performance. The question then is, how should we develop a framework that supports *functionality* for balancing loads and curtailing remote-data accesses?

• *The authors are with the Department of Computer Sciences, Purdue University, 1398 Computer Science Building, West Lafayette, IN 47907-1398. E-mail: {bdd, rego}@cs.purdue.edu.*

Manuscript received 3 Aug. 1996.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number 100265.

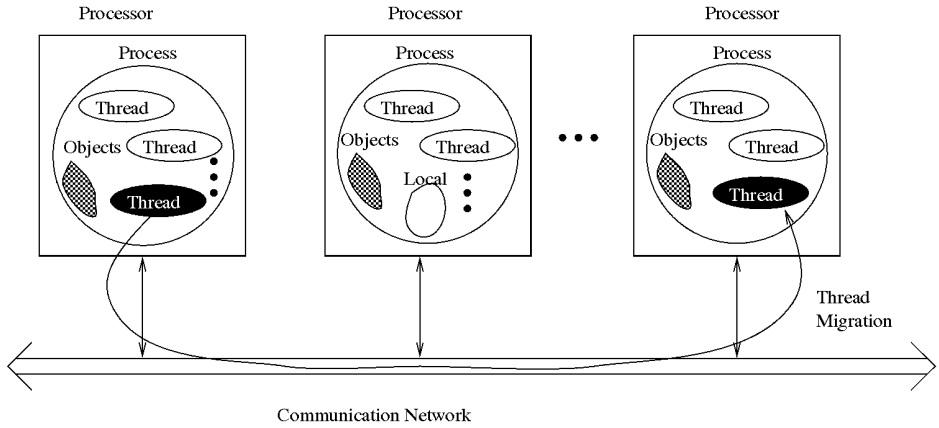


Fig. 1. Distributed threads and objects.

A simple solution to the above problem is to exploit the fact that a distributed system *can* be implemented with threads that are free to migrate (i.e., move their data and computation state) from one processor to another. As shown in Fig. 1, a distributed computation can be viewed in terms of threads that operate on objects. Threads from highly loaded processors may migrate to processors with less load. Though, in general, this is best done under the supervision of an appropriate load-balancing algorithm, there are situations where such algorithms may not be necessary. For example, on shared-memory multiprocessors, Ariadne [15], [14] sees balanced loads as a direct result of placing runnable threads in a shared queue.

To appreciate the utility of thread migration, consider a distributed (discrete-event) simulation application, for example, a large queuing system. Such a system may be used as a modeling abstraction of a telecommunication system. If the queuing system's `Facility` objects (a `Facility` contains a "server" and physical space for holding queued "customers") are distributed across processors, then distinct customers (threads) may access distinct facilities concurrently. When a customer thread, hosted by a given process, invokes a `reserve(F)` primitive—to request service from a server—on a facility *F* that happens to be located within the same process, the thread accesses a *local* object. If *F* turns out to be a *proxy* for a facility object that is hosted by another process,[1] then the thread must access a remote host. Such an access is effected by *transparently* migrating the requesting thread to the remote host. At the receiving end, the remote host places the migrant thread in a simulation calendar and uses an appropriate distributed simulation protocol to decide when the thread should run. The thread may or may not return to its sending host, depending on simulation logic. Observe that the migration activity takes place within a domain-level `reserve()` primitive, obviating the need for user-specified send and receive calls. Thus, thread migration gives the user a powerful abstraction that enhances distributed application development.

In general, if threads on one processor require frequent access to objects on remote processors, these threads may migrate to processors hosting their respective objects, so that subsequent data accesses become local. By invoking a migration primitive on host *A*, a thread suspends its own execution on *A* and transfers to host *B*. Once there, it resumes execution starting with the instruction immediately following the call to migrate. With appropriate layering and support for locating objects [15], [13], thread migration can be made transparent to end-users. We have successfully adopted this approach in the ParaSol parallel simulation system [13], [10], [16], which exploits optimistic and adaptive simulation protocols.

In this effort, we focus on the design of a threads system that supports thread migration on heterogeneous networked machines. On networks of homogeneous machines [15], [11], [2], thread contexts and stacks are well-defined across process boundaries and, thus, in such environments, efficient migration is hinged upon conventional context and stack manipulation. On networks of heterogeneous machines, however, contexts and stacks are meaningless across architectures. Because of this, the formidable problems of context and stack translation make thread migration on heterogeneous systems a real challenge. To accommodate heterogeneity, we take a transformation-oriented approach that both builds upon and removes a limitation of our earlier work, namely the Ythreads [21] system. In Ythreads, each thread is allowed to perform thread operations (creation, changing priority, yielding, etc.) only within the base thread function associated with the thread. Our current design in Arachne removes this limitation and, in addition, extends the scope of support by providing an interface to different communication libraries.

The outline of the rest of the paper is as follows. In Section 2, we present some background on work in this area and the origins of our design. Section 3 contains some basics on Arachne's functionality, and Section 4 presents details on the code transformations performed by our preprocessor. Details on the runtime system are given in Section 5, and Section 6 addresses difficulties that a designer faces in providing heterogeneous migration. In this section, we also present design solutions that we chose for Arachne, based on the discussion in Sections 4 and 5. We present some performance results in Section 7, and conclude in Section 8.

2 Related Work

There are two basic types of threads systems: user-space threads and kernel-space threads. While kernel-space threads

1. The other process may be running on the same processor or on a remote machine. In either case, we call the access a remote access, since it involves crossing a process boundary.

are fully managed by the operating system (OS) kernel, user-space threads are managed (i.e., created, scheduled, destroyed) without OS intervention. Therefore, the management of user-space threads is considerably cheaper than the management of kernel-space threads, both in terms of time as well as system resources. Because of the lack of direct OS support, however, user-space threads exhibit some limitations. For example, user-space threads cannot be scheduled by the OS scheduler, though they may be scheduled (i.e., time-sliced, preempted) by a user-level threads scheduler [15], [14] that is not recognized by the OS. If a user-space thread blocks on a system call, the process that hosts this thread must also block. At the time a user-space thread is created, a limit must be provided for its stack size. In general, user-space threads systems are run-time-only and implement context-switch actions by using either assembly code [19] or C-library `setjmp()` and `longjmp()` primitives [15], [8]. Very few user-space systems support thread migration. Those that do (e.g., [2], [15], [19]) invariably impose some restrictions and, in particular, offer threads that can migrate only between machines of the same type. In general, the Ariadne system [15], [14] is the least restrictive system in this class, and even provides support for time-slicing and user-defined scheduling.

Systems that support thread migration do so by using context-switching functionality: A running thread suspends execution, its stack is copied to a thread-template on a remote machine, the thread is recreated on the remote machine, and the newly created thread resumes execution when allowed to proceed. On the receiving machine, the new thread resumes computation at the statement following its request to migrate. A typical restriction seen in systems belonging to this class is a hard limit—which must be specified at the time of a thread's creation—on a thread's stack size. This restriction either severely curtails the number of nested function calls that a thread can invoke or leads to inefficient memory usage.

2.1 Ythreads

On networks of heterogeneous machines, differences in hardware, software, and operating systems severely complicate the migration approach taken by runtime-only systems. In one machine's view, the stack of a thread operating on a different type of machine is meaningless. Help is required in interpreting such a stack—at the very least, information on the type of the various data contained in the stack. Because transforming stacks at runtime is prohibitively complicated, our prior work, *viz.* the Ythreads system, advocated a solution based on the use of a preprocessor for an existing language [21]. With such a setup, the user writes programs in that language, preprocesses the code, and compiles with a conventional compiler. Though Ythreads provide for heterogeneous thread migration, it is limited in that thread primitives can only be invoked in top-level thread functions. They cannot be invoked in nested functions.

2.2 Emerald

A somewhat similar effort has been reported in [24], where a group of researchers rewrote an existing Emerald compiler in a way that enables heterogeneous thread migration.[2] This compiler examines the source code and inserts appropriate instructions immediately before migration requests. These instructions encode and pack a function's state. Thus, threads and objects can freely migrate between the four platforms supported by Emerald: Sun 3, Sun SPARC, VAX, and HP. While this approach is powerful, it has its disadvantages. First, there is the need for application development in the Emerald language and continued compiler development for the support of different machine architectures; the different compilers will need maintenance and support. Adding a fifth architecture to the list of supported platforms will require porting the entire compiler. Also, there are practical problems related to integrating Emerald threads and objects with software that already relies on some threads library. Finally, there is the important issue of performance. It was found that the compiler developed in the project [24] took 60 percent longer to migrate and subsequently invoke a function than the Emerald compiler that supports only homogeneous thread migration [9].

2.3 SimCal

The idea behind our design, especially the use of a code preprocessor, has its origins in work done by Malloy and Soffa [12]. By augmenting Pascal with certain Simula control structures, the authors defined a language called SimCal. They were not interested in thread systems or thread migration, but wanted to add retentive control capabilities to Pascal. In their system, as well as in ours, a preprocessor reads all user files and produces output conforming strictly to an existing programming language. The primary task of the preprocessor is to preserve the state of thread functions (the values of their local variables and the point reached in execution) in such a way that these thread functions can be restarted at a later time by an appropriate scheduling mechanism.

3 The Arachne System

Arachne provides complete support for heterogeneous thread migration, building upon the Ythreads work [21]. But there is one restriction: All points where a thread *may* be suspended must be known at the time of compilation. This is because our preprocessor must insert special code at these points for preserving thread state prior to suspension so that the thread is able to resume execution at some later point in time. Other than this, Arachne threads may be created with any number of arguments;[3] there is complete support for recursion, and threads that get suspended, activated, or migrated may have arbitrarily deep stacks. Arachne does not limit a thread's stack size at the time of its creation, but instead manages space dynamically. Requests for creation, suspension, migration, etc., can be made in any function that is invoked by a thread through a chain of arbitrary length. Such chaining is impossible in Ythreads and SimCal.

As indicated in the previous section, Arachne was designed to replace Ariadne in supporting the ParaSol distributed simulator. This substitution enables ParaSol to also run on heterogeneous networked machines. We designed Arachne to support standard functions as opposed to class methods (as in

2. Emerald [9] is a language in which objects may migrate between homogeneous machine architectures.

3. At present, the types are limited to ones that can be handled by the variable argument facility in C/C++: `int`, `long`, and `double`. In the future, we will accept all possible types.

Java). This is because the ParaSol API provides users with functions (e.g., `reserve()`, `release()`, `hold()`) that are invoked by application-level threads (e.g., transactions, customers). Thread migration is encapsulated within application-level constructs (e.g., `reserve()`). This enables a user to develop code without resorting to send and receive calls or keeping track of processor ids. Objects are assigned to different processors at the start of program execution, and threads that need to access remote objects consult a local-object locator before migrating to a remote host. Since our threads represent functions with state, Arachne has to save and restore the state of these functions. The idea can be extended to include state management of class methods, though this not required by ParaSol. We plan to implement this extension in the next version.

3.1 Functionality

Arachne's heterogeneous migration facility is made possible by augmenting the C and C++ programming languages with three new keywords: `thread`, `strand`, and `call`. Our system consists of a code preprocessor, `app`, which transforms application code from this slightly extended C++ language into "standard" C++.[4] The resulting code is then compiled with a conventional C++ compiler and linked with the Arachne runtime library, `libarachne.a`. Compiling and linking on each distinct architecture is required. This yields a set of platform-dependent executables for all machines that will participate in a distributed program's execution.

The Arachne preprocessor, its output, and the runtime library all conform to C++. Thus, if a C++ compiler is available on a given architecture, Arachne will run on that architecture. If a communications library supports two distinct platforms that each support Arachne, then Arachne threads can migrate between those platforms.

3.2 The Arachne Primitives

Thread operations in Arachne are initiated by invoking any one of the following nine functions, the so-called *Arachne primitives*.

- `a_create` (`int *id`, `void` (`*func`)(`void`), `int prio`, ...). Creates a new thread with priority `prio` to run the preprocessed function `func()`. Despite the signature, as will be discussed later, `func()` can have any number of arguments. The thread ID is returned in `id`.
- `a_destroy` (`int id`). Destroys thread `id`.
- `a_migrate` (`int proc`). Migrates the current thread to the process `proc`. Typically, `proc` will reside on a different machine.
- `a_resume` (`int id`). Resumes thread `id`.
- `a_set_prio` (`int id`, `int prio`). Sets the priority of thread `id` to `prio`.
- `a_suicide` (`void`). Destroys the current thread.
- `a_suspend` (`void`). Suspends the current thread.
- `a_yield` (`void`). Gives control of the CPU to some other ready thread. If none is available, the current thread continues execution. Arachne maintains the invariant that, at any time, the executing thread has priority higher than or equal to all nonsuspended threads in the current process.
- `a_yield_to` (`int next_id`). Gives CPU control to thread `next_id`, if it is ready to run, as follows from the Arachne scheduling invariant.

All of these operations represent *potential* suspension points, namely, places where the currently executing thread *may* give up the CPU so that another ready thread may execute. But reaching a suspension point does not necessarily mean that a context-switch will occur. The primitive `a_set_prio()` is a suspension point, but a context-switch will not occur if the thread lowers some other thread's priority.

3.3 Thread and Strand Functions

At any point in a thread's execution, its stack is defined to consist of the activation records of all functions that it has invoked, but not yet returned from. Preserving a thread's stack requires preserving the activation records of all such functions.

Arachne makes a distinction between three kinds of functions: thread functions, strand functions, and "regular" C/C++ functions. A *thread function* is a function given as the second argument to the `a_create()` primitive—it is the function with which a thread begins its execution. All thread functions are invoked directly by the Arachne runtime scheduler. Thread function definitions must be preceded by the Arachne keyword `thread`. This is the first of three extensions that Arachne introduces to C++.

A function that invokes an Arachne primitive is called a *strand function*. In addition, a function that invokes itself or another strand function is also called a *strand function*. Thus, each thread function is a special kind of strand functions—because it can invoke strand functions and Arachne primitives. However, thread functions are different from strand functions in that thread functions are recognized by and are invoked by the Arachne scheduler, while strand functions are invoked by threads. Strand function definitions must be preceded by the keyword `strand`. This is the second extension of C++.

The above definitions lead to a few important conclusions. First, an invocation chain must contain exactly one thread function, which is the one that runs immediately after the scheduler. Second, a thread function can invoke both strand and regular functions. Third, strand functions can invoke only strand and regular functions. Finally, regular functions can invoke only regular functions. As a direct result of these conclusions, it follows that *a thread may be suspended only if its stack contains one thread and some number (possibly zero) of strand functions*. This is true, because only thread and strand functions may invoke Arachne primitives, and invoking an Arachne primitive is the only way to cause a thread suspension. This idea is illustrated in Fig. 2.

The final extension to C++ is the keyword `call`. Due to the limited look-ahead capabilities of our preprocessor, this keyword must be placed in front of every strand function invocation. We realize that this requirement is somewhat cumbersome and plan to eliminate it in our next release.

4 Code Transformations

4.1 The Code Preprocessor

The Arachne preprocessor, called `app`, consists of a scanner and a parser, and performs syntax-directed parsing. It does not

4. We refrain from using the term ANSI C++, because that standard has not yet been finalized.

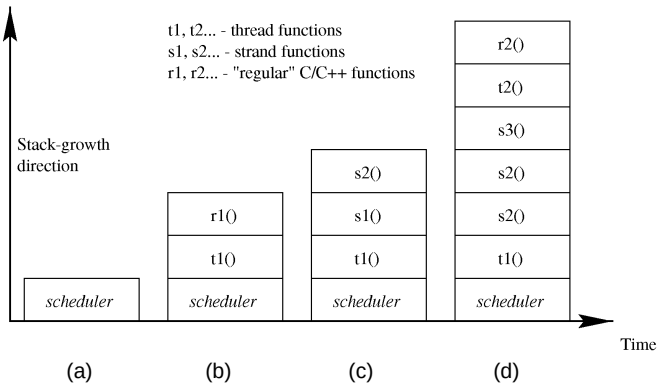


Fig. 2. Four snapshots of the runtime stack: (a) no thread is running; (b) a thread is executing thread function `t1()`, which then invokes a regular function `r1()`—a context-switch is not possible within `r1()`; (c) after returning from `r1()`, thread `t1()` invokes strand function `s1()`, which in turn invokes strand function `s2()`; (d) a longer chain, which shows that strand functions can be recursive; once again, no context-switch is possible within regular function `r2()`.

have a front- or a back-end, and it does not build an intermediate representation of the user code. Instead, as soon as a token is matched, the scanner writes it in the preprocessed stream and returns its value to the parser. If there is a need to output manipulated tokens or to insert code in the preprocessed stream, the parser turns the scanner output off, writes the necessary tokens, and then turns scanner output back on.

To develop **app**, we extended the public C and C++ grammars written by Roskind [17], [18]. Our parser has 115 terminals, 186 nonterminals, 681 grammar rules, and 1,290 states.

4.2 Context-Switching Between Threads

While an Arachne program executes, the scheduler executes the following loop: It examines the contents of the global variable **`next_thread`** in order to obtain the id of the thread to run next. It then invokes the thread function corresponding to that thread. The thread function may invoke strand and regular functions, and may execute for an arbitrarily long time. Eventually, the thread will either run to completion or will reach a suspension point. In either case, the runtime system determines the next thread to run, and loads this thread's id into the variable **`next_thread`**. If a thread suspends its execution, all functions on the thread's stack are forced to return control to the function that invoked them. These returns are enforced all the way, until the scheduler is the only function left on the machine runtime stack. If a thread runs to completion, the scheduler automatically obtains control. In either case, the result is that the scheduler runs to the bottom of its iteration loop to check if Arachne's termination conditions have been met; if so, the system terminates and, if not, the scheduler goes through another iteration.

There is one important detail left unaddressed in the sequence of actions described above. In normal program execution, when a C or C++ function returns from its invocation, the values of its local variables are lost. Thus, if a thread needs to resume its execution after suspending and giving up control of the CPU, it must preserve the states of all functions on its stack—one thread and some number of strand functions. It must do this just *before* it forces all its functions to return in preparation for the suspension. In Arachne, we define a function's state to consist of the values of all its local variables and the point reached in its execution.

If, at times of context switching, functions were not forced to return, but instead invoked the next thread's thread function, there would be no need of state preservation—the function states would be preserved on the machine stack. But this would cause a machine stack overflow, and will crash the program execution.

4.3 Preservation of Local Variables

The **`app`** preprocessor is responsible for preserving the states of all thread and strand functions. We explain how this is done in terms of a thread function **`t1()`**. While preprocessing **`t1()`**, **`app`** creates **`struct t1_ar`** in the global scope—a data-type, tailored especially to **`t1()`** that contains fields for all of **`t1()`**'s local variables and parameters. The code of **`t1()`** is then manipulated to access local variables through a pointer to a variable of type **`struct t1_ar`**. At runtime, when a thread is created to run **`t1()`**, a variable of type **`struct t1_ar`** is created (on the heap) and is passed to **`t1()`**. This pointer is preserved within the thread table in the entry corresponding to the thread in question. In this way, since the memory holding the local variables is now in the heap, instead of on the stack, **`t1()`**'s local variables are preserved between thread activations. When a thread is destroyed, the memory for the heap structure is returned to the free-store.

In a similar manner, data-types are tailored for all strand functions also. When a strand function **`s1()`** is invoked by a thread, memory is allocated for a struct of the type **`struct s1_ar`**, and strand **`s1()`** is made to access its local variables through a pointer to that structure. The structure's memory is returned to the heap when the strand function completes its execution.

4.4 Preserving of Suspension Points

To enable the saving and subsequent restoration of a thread's suspension point, **`app`** inserts an integer value called **`state`** in each data type that is tailored to a thread or a strand function. Just before each point of suspension in the program text, **`app`** inserts a statement to increment the value of **`state`**; just after each point of suspension, **`app`** inserts a jump label. At the start of each function's code, **`app`** inserts a **`switch`** statement which directs execution control to a jump label based on the value of **`state`**. Given a particular value of **`state`**, a jump (via **`goto`**) is performed to the appropriate label, thus enabling a thread to resume after suspension.

The invocation of each thread and each strand function is immediately followed by this **`switch`** statement. On the very first invocation of a thread or a strand, **`state`** is equal to zero, and no jump occurs. Execution proceeds until the first suspension point. At this point, the value of **`state`** is incremented by one. If a context-switch does occur, the function returns and the scheduler invokes another thread. Later on, when the first thread is allowed to resume execution, the same function is invoked once again. But now, the **`switch`** statement examines **`state`** and forces a **`goto`** jump to the label lying immediately after the first suspension point.

4.5 An Example

Let us assume that **app** preprocesses a file called **test.C**. It will then produce the files **test_a.C**, **test_a.h**, **test_pack.C**, and **test_unpack.C**. To illustrate the manipulations performed by **app**, let us assume that **test.C** contains the thread function **t1**, defined as shown in Fig. 3. The function **t1()** is identified as a thread function by the keyword **thread**; the function **a_migrate()** is an Arachne primitive, and **s1()** is a strand identified by the keyword **call**. All strand invocations must be preceded by this keyword or else an error message is generated.

The global structure corresponding to **t1()** is shown in Fig. 4. This structure, and all others (for other thread and strand functions), are written into the file **test_a.h**.

Observe that **struct t1_ar** contains fields for the two local variables, **ch** and **l**, one field for the parameter **i**, and the ever-present field **state**. If any preprocessed function happens to contain a local variable named **state**, **app** generates an error message.

4.6 Thread- and Strand-Function Manipulations

The preprocessed version of **t1()** can be seen in Fig. 5. This function and all other preprocessed thread and strand functions are placed in the file **test_a.C**.

Although the code manipulations may appear overwhelming at first glance, they can be grouped into only a few categories:

- *Removal of the keywords* **thread** *or* **strand**. After having served their purpose, which is to trigger preprocessor activation, the keywords **thread** and **strand** are deleted from the code before C++ compilation. This also brings the code closer to C/C++ syntax.
- *Removal of all function arguments.* To preserve the semantics of parameter passing, thread function parameters must be evaluated only once, which is at the time of thread creation. Thread parameters are added as fields of the thread's activation record, and their values are defined when a thread is created. The code for thread creation assigns these values directly to the corresponding fields, leaving the preprocessed thread function free of parameters.
- *Relocation of all local variable declarations from the function body to a global (heap-based) data-structure.*
- *Preservation of local variable initializations.* If a local variable declaration also contains an initialization (e.g., the variable **ch** in **t1()**), the initialization cannot be part of the **struct** definition, and so is left in the body of the preprocessed function as a stand-alone statement.
- *Modification of access to local variables.* In a preprocessed thread or strand function, all accesses to "local" variables become accesses to the corresponding field of **t1_ar**.
- *Declaration and setting of the value of a pointer to a thread's or strand's context.* A preprocessed thread or strand function has only one local variable—a pointer to the structure holding its (original) local variables. This pointer, **ar**, is defined and initialized using the code

  struct t1_ar *ar = (struct t1_ar *) (cur_ar->ar).

  The value of **cur_ar** is set by the runtime system at the same time as setting **next_thread**.

```
thread void t1 (int i)
{
   char ch = '+';
   unsigned long l;

   a_migrate (procs[0]);
   call s1 (i, l);
}
```

Fig. 3. An example thread function definition.

```
struct t1_ar {
   short state;
   int i;
   char ch;
   unsigned long l;
};
```

Fig. 4. Definition of **t1()**'s global structure, **t1_ar**.

```
void t1 (void)
{
   struct t1_ar *ar = (struct t1_ar *) (cur_ar->ar);

   _did_yield = 0;
   switch (ar->state)
    {
     case 0: goto t1_state_0;
     case 1: goto t1_state_1;
     case 2: goto t1_state_2;
     case 3: goto t1_state_3;
    }

t1_state_0:
   ar->ch = '+';

   a_migrate (procs[0]);
   (ar->state)++;
   if (_did_yield)
    return;

t1_state_1:
   pushAR ();

t1_state_2:
   cur_ar = cur_ar->next;
   s1 ((ar->i), (ar->l));
   cur_ar = cur_ar->prev;
   (ar->state)++;
   if (_did_yield)
    return;
   else
    popAR ();

t1_state_3:
   a_suicide ();
}
```

Fig. 5. Code for the preprocessed version of thread **t1()**.

- *Insertion of a series of jump labels.*
- *Insertion of a series of statements just before and just after calls to primitives and calls to strands.* To prevent the machine stack from overflowing, threads and strands should return control to the scheduler during a context-switch. **App** provides for this by inserting code just after a call to a primitive, forcing a **return**. Calls to strands are handled in a similar manner, except that there is need for a label just *before* the strand call (so that the computation can be restarted after the thread is reactivated), and just *after* the strand call (so that a **return** is possible in case the strand effects a

context-switch); the strand's context must be set just before its invocation, and reset just after its return, by manipulating `cur_ar`. The routine `pushAR()` allocates space for a strand context and pushes it onto a stack of activation records that are contained in the *runtime* thread's context; `popAR()` pops the top context and frees all memory held by this context. Both `pushAR()` and `popAR()` are contained in `libarachne.a`.

- *Insertion of a* `switch` *statement to determine entry points.*
- *Insertion of a call to the primitive* `a_suicide()`. When a thread runs to completion, its context should be freed and the runtime environment should be updated accordingly. This is only needed for thread functions.

5 The Runtime Support Subsystem

Arachne provides support for multithreaded applications on distributed systems of heterogeneous processors. For this, the application must run as a system of distributed (Unix) processes. The Arachne library gives each process the ability to create, destroy, migrate, and manipulate threads in any way that suits the application. Arachne relies on a communication library to provide facilities for messaging and distributed process management. This is accomplished through a flexible communications interface; the current implementation is interfaced with the PVM library.

In a distributed application that is based on distributed threads, the OS treats Arachne processes as standard user-level processes. This is possible only because a process's thread contexts, heaps, and stacks are maintained within its address space. An application and, hence, a process, schedules its own user-space threads.

In an Arachne application, a process may host one or more active threads. At any given time, each active thread is in one of three possible states: *current*, *ready*, and *suspended*. A thread is in the *current* state if it is currently executing. On a uniprocessor, only one thread may be in this state at any given time; on a multiprocessor with n processors, up to n threads may be in this state at any given time, with each hosted by a process on a distinct processor. Threads that are ready to run if scheduled, but currently await the scheduler's attention, are said to be in a *ready* state. Threads that are in the *suspended* state must be subjected to a "resume" operation before they can be placed in the *ready* state and eventually run.

5.1 The Thread-Context Area

During a thread's execution and, in particular, when a thread is in the *suspended* state, its context is located in a special area known as the *thread context area* (TCA). Therefore, migrating a thread from one host to another is equivalent to transferring its TCA to a receiving process and restarting the thread's computation using this TCA. A graphical representation of a TCA is shown in Fig. 6.

The TCA consists of a two structures: a *thread record* and a *thread activation stack*. The thread record contains all information that is useful in scheduling a thread—its id, state, priority, and a pointer to its thread function. The size of a thread record is fixed; only the contents of the fields `prio` and `state` are modifiable during its execution. Thread activation stacks are abstractions of machine stacks, and are

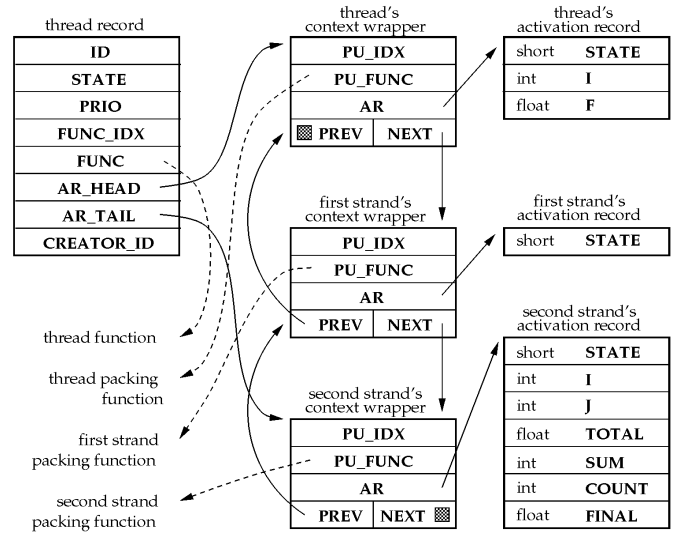


Fig. 6. A thread context area (TCA) containing two strands. A TCA consists of a fixed-size thread record and an expandable thread activation stack—a doubly linked list of context wrappers.

discussed below. The *thread table* is the collection of all thread records contained in a single Arachne process. To enhance performance, the thread table is statically allocated during program initialization in the current version of Arachne. As a result, its size imposes a hard limit on the number of threads simultaneously active within a single process.

5.2 Thread Migration

We now turn our attention to what is perhaps the most important runtime feature of Arachne, namely, support for efficient thread migration between different machine architectures. As shown in Fig. 6, an active thread may be viewed in terms of its activation stack: a doubly linked list of *context wrappers*. Each context wrapper contains a pointer to a thread's or strand's activation record, and also a pointer to a function that can pack or unpack this record. The TCA maintains the dynamic thread activation stack for the purpose of context-switching and thread migration. This activation stack is simply Arachne's abstraction of the machine stack. A strand context is added to the end of the doubly linked list by `pushAR()` every time a strand is invoked. The strand context is removed by `popAR()` when the strand runs to completion.

Migrating a thread from one process to another involves four distinct steps: converting a suspended thread's state into a machine-independent representation, sending the representation to a remote machine, receiving and reconstructing the thread's TCA from the machine-independent representation and, finally, performing a context-switch to the immigrant thread so that it may resume execution. Each of these steps is explained below.

5.2.1 Machine-Independent TCA Conversion

The context wrappers within a TCA are similar to C++ objects in that they contain data, as well as functions to operate on this data. The primitive `a_migrate()` invokes these functions in the correct order. When invoked, `a_migrate()` performs the following actions in sequence: It initializes the network buffer, packs the thread record (whose fields are well-defined and

```
int t1_pack (ar_p act_rec)
{
   struct t1_ar *ar = (struct t1_ar *)(act_rec->ar);

   (ar->state)++;
   if (pvm_pkshort (& (ar->state), 1, 1) < 0)
      return FALSE;
   if (pvm_pkint (& (ar->i), 1, 1) < 0)
      return FALSE;
   if (pvm_pkbyte (& (ar->ch), 1, 1) < 0)
      return FALSE;
   if (pvm_pku long (& (ar->l), 1, 1) < 0)
      return FALSE;

   return TRUE;
}
```

Fig. 7. Code for packing the contents of the activation record of function `t1()`.

fixed for all TCAs), traverses the stack of context wrappers and invokes their corresponding packing functions and, finally, flags the end of the message by inserting the integer `-1`.

The packing functions are responsible for the conversion of all values into platform-independent formats. Going back to our example in Fig. 3, the preprocessor generates not only the `struct t1_ar`, but also the two functions `t1_pack()` and `t1_unpack()`. These two functions are responsible for packing and unpacking `t1()`'s activation record before and after migration. These functions make calls to the underlying communications substrate. The packing routines are written into file `test_pack.C`, and the unpacking routines are written into file `test_unpack.C`.

In Fig. 7, the code of `t1_pack()`is shown. The global pointer to the current activation record, `cur_ar->ar`, is defined to be of type `void *` so that it is allowed to point to any activation record. An appropriate type-cast must be applied (the very fist line of the function's body) before any data can be correctly retrieved.

We have chosen to convert TCAs from machine-dependent format into machine-independent format and back, instead of directly from one machine format to another. This makes our solution very modular and readily portable to any architecture that supports the independent format External Data Representation (XDR). XDR is readily available on virtually all hardware platforms, and is supported by most popular heterogeneous data-communication libraries (including PVM).

5.2.2 Sending Threads in Messages

After a TCA is converted into a machine-independent form and copied into a buffer, the communications substrate constructs a message containing the TCA and sends the message to a remote machine. For example, with PVM, a single call to `pvm_send()` ships the TCA data to a remote process whose id is specified in an argument to `a_migrate()`. Observe that once a thread leaves one process to migrate to another, its TCA should be removed from the sending process, and its corresponding thread record should be freed. Indeed, these are the last actions performed by `a_migrate()`.

5.2.3 Reconstructing Threads from Messages

In each Arachne process, the communications substrate receives incoming messages from the network and places them in a buffer. How frequently the runtime system retrieves messages from this buffer depends on facilities provided by the communications substrate and the needs of the application. In the prototype implementation, the runtime system checks for incoming thread messages every time a context switch is performed; a newly-arrived message is transformed into a thread and immediately placed on the ready queue. When an incoming thread message is detected, the scheduler invokes a generic "unpacking driver" to process the message. For each such message read from the network buffer, the driver uses the unpacking functions specified in the TCA to reconstruct a thread record and activation stack. Because raw function pointers are meaningless across heterogeneous processor boundaries, a special corrective procedure must be used on these. This mechanism is described in Section 6.

A TCA always contains a thread record and at least one context wrapper, i.e., the wrapper belonging to the thread function. The first field in a context wrapper, i.e., `pu_idx`, is valid if it contains any nonnegative value, and identifies the appropriate unpacking function to be used in unpacking this context. A value of `-1` indicates that there are no more contexts present in this TCA (see Fig. 6).

Because all Arachne programs are linked with the library `libarachne.a` and an architecture-specific version of the communications substrate, packing and unpacking yield all the required functionality. With PVM, for example, when a thread migrates from a machine with architecture *A* to a machine with architecture *B*, the packing routines do a conversion from an *A*-specific to a machine-independent representation; the unpacking routines reverse the operation, converting the machine-independent representation to a *B*-specific representation.

5.2.4 Resuming a New Immigrant Thread

When the current thread invokes an Arachne primitive that requires it to relinquish control of the CPU, the scheduler obtains control. In the absence of asynchronous interrupts, which can enable the system to respond to arriving threads whenever they arrive, the scheduler attempts to processes incoming threads each time it obtains control. If it finds one or more incoming threads, the necessary thread reconstruction work is performed, and the "new" threads are added to the ready queue. Only after this does the scheduler locate the highest priority thread to give it CPU control. Thus, if a "new" thread has a higher priority than the other threads, it will be the next thread to run.

5.3 Distributed Termination

In general, detecting termination conditions is difficult, primarily because processes may appear idle even though messages may be in transit. Such messages may potentially reactivate an idle process. Some termination algorithms (see, for example, [3]) assume specific network topologies. In Arachne, we use an algorithm that is capable of handling any topology—an algorithm that is also used by the Ariadne [15] threads system. This algorithm has proven to be simple and effective, and exhibits low demands on the network.

In Arachne, a process is either *active* or *inactive*. Each thread created by a process is an *offspring* of that process. A process is said to be *active* if at least one of its offspring is

active (i.e., current, ready, or suspended), or if it currently hosts at least one offspring of another process. If both conditions are false, a process is *inactive*.

To determine its status, a process maintains an integer `num_live`, which is a count of all its live offspring, and an integer array `proc_destr`, which indicates how many offspring belonging to other processes it has destroyed while it was active. Each time a process changes its status from active to inactive, it sends an "annihilation" message—the value of the corresponding element of `proc_destr`—to all other Arachne processes. It also sends a message to a special "controller" process each time it makes a transition from active to inactive, or vice-versa. The controller maintains a count of the number of active processes in the system. When this number becomes zero, the controller sends termination messages to all processes. Upon receipt of such a message, each process performs all necessary clean-up operations and exits.

The above algorithm attempts to limit its use of network bandwidth by sending very small messages, and only at times when a state transition occurs. However, if there are frequent state transitions, even this algorithm can be quite burdensome. We chose to use it because of its simplicity, and because it assumes nothing of the underlying network topology. These are significant advantages.

6 The Challenge of Thread Migration

6.1 Architecture Dependencies

The biggest challenge to thread migration lies in the transformation of a thread's computational state from a machine of one type to a machine of a completely different type. At the very least, this entails devising solutions to the following problems: The runtime stack is machine-dependent; the size of machine words may differ, the representation of the base types may use a different number of words, the byte ordering may either be big endian or little endian, and some architectures require that data be aligned on even addresses or some other word-multiple. Conventions for calling functions and handling return values may differ across architectures and operating systems. Besides instruction sets, the number and sizes of CPU registers may also differ across hardware platforms. And, finally, compilers on different machines may produce different code or be capable of yielding different levels of optimization.

Arachne solves stack-related problems by preprocessing all thread and strand functions in user code (as described in Section 4). With the help of any one of a number of suitable communication libraries,[5] Arachne produces a machine-independent "logical" runtime stack. Just before the system enables a thread to migrate, Arachne encodes its own "local" stack and forces all active thread and strand functions to `return`. When a migrant thread's context arrives at a remote machine, the Arachne system executing on that machine recreates the thread's stack and invokes the appropriate thread and strand functions. This effectively solves the problems related to function invocation and parameter passing. All code that Arachne emits is C++, and each machine's compiler is responsible for dealing with the different instruction sets, register allocation, and optimizations.

5. Arachne provides an interface through which it can exploit different communication libraries. The current implementation uses PVM.

6.2 Address Space

In homogeneous networked environments, because processes are typically created from the same binaries, function pointers can be assumed to be valid across machine boundaries. But this is not true in heterogeneous network environments and, thus, an additional mechanism is required for correcting the values of all function pointers. This is done between the time a thread leaves one machine and the time it resumes execution on a destination machine. The preprocessor `app` and runtime library `libarachne.a` work together to solve this problem. `App` outputs an array of pointers to all thread functions, an array of pointers to all packing functions, and another array of pointers to unpacking functions. While compiling and linking on machines with different architectures, the correct addresses of all functions are placed in the corresponding fields of these three arrays. The thread record, and thread and strand context-wrappers, contain indices of the thread function, and the thread and strand packing and unpacking functions. These integer values are precise descriptions of the indices of the necessary functions in the three arrays of pointers. When a thread is initialized, the function index and the function pointers are properly set. When a thread migrates, the runtime environment reassigns the function pointer using the function index. The index is assigned by the preprocessor, and is correct at all times.

7 Performance Results

We report Arachne's performance as measured on three machines: a Sun Sparc 5 (with a 70 MHz CPU), a Sun Sparc 20 (with four 50 MHz CPUs), and a Silicon Graphics Indigo (with a 132 MHz CPU). We also compare Arachne's performance to the performance of the Pthreads system (version 1.6), the SunOS 5.3 multithreads system (Sun-MT, a kernel-based system), and the Ariadne threads systems, all operating on the Sun Sparc 20.

Each test was designed to repeat thread or strand operations sufficiently often to make the total runtime of the test exceed 60 CPU seconds. For each run, the average time required to execute an operation is obtained by dividing the runtime by the total number of operation invocations. To account for variability, the time required to perform an operation is obtained as an average over 10 independent runs. The standard error was found to be insignificant. We used Unix's `clock(3C)` facility to measure runtimes. This command has a granularity of 1 millisecond and returns the total time that a process spends with the CPU, ignoring context-switches and the execution of other processes.

In Table 1, we show the timings for the following three basic thread operations:

1) *Thread creation*. This refers only to the creation, not the execution of a thread.
2) *Null thread execution.* This refers to the creation, execution, and subsequent destruction of a thread whose function contains no user code.
3) *Context-switching*. This refers to the transfer of control from the current thread to a ready thread. Because the

TABLE 1
RUNTIMES (IN μSEC) FOR PERFORMING THREAD OPERATIONS

Operation	Sparc 5	Sun Sparc 20				SGI
	Arachne	Arachne	Ariadne	Pthreads	Sun-MT	Arachne
Thread Create	18	14	35	49	1,000	13
Null Thread	35	24	40	69	1,130	19
Context-Switch(0)	22	13	15	31	12.5	11
Context-Switch(1)	27	16	15	–	12.5	14
Context-Switch(2)	30	18	15	–	12.5	16

Note that Arachne exhibits competitive runtimes, even though context-switching cost grows as a function of stack depth.

amount of time required for completing a context-switch depends on the depth of the activation chain (i.e., the number of strands invoked by the yielding and target threads), we performed three different measurements using zero, one, and two active strands in the yielding thread.

We also report execution times for strand operations. The results are presented in Table 2. Strands possess a mixture of thread and ANSI C++ function properties. In the early stages of system design, we expected strand operations to be inexpensive, and our experiments support the conjecture. The most expensive strand operation involves its very first invocation, since this invocation requires the creation and initialization of a strand's context. This operation is roughly equivalent to the *null thread* operation and the timings for both are close. Because all strand operations involve an extra `if`-statement, they are a little more expensive than the null thread operation. The least expensive operation by far, and also one with potential for the most frequent use, is the *strand invoke* operation. The *strand destroy* operation refers to the deallocation of a strand's context, and each strand is allowed only one invocation of this operation.

Finally, we also report timings for thread migration. As previously discussed, the time required for a thread to migrate depends on the time required for each of several operations: returning control to the scheduler, packing a TCA, sending and receiving the message containing the TCA, unpacking the TCA, and reconstructing and finally running the thread. Migration times for threads holding activation records of sizes 12 bytes and 128 bytes can be seen in Table 3.

As seen in the table, migration times are much larger than the sum of two context-switches and a null thread operation. This is due to the large amount of time required for sending and receiving the message containing the thread's TCA. While it may appear that such overheads offset any advantages due to thread migration, we observe that if a thread requests data located on a different processor, equivalent networking overhead will be paid every time such a request is fulfilled. That is, if threads are not used, the same amount of information must be sent in a message and the cost will be almost the same. Observe, however, that when a thread migrates to its data, the messaging cost for repeated data access is paid only once.

To get a rough idea of Arachne-induced preprocessor overheads on user code, we measured the time taken by two different programs to compute the sum of all positive integers smaller than 10^7. One program computed without thread or strand functions and the other program performed the summation within a strand function. The result was an overhead of 61 percent, due mainly to indirect addressing of local variables in the TCA. We observed similar results for code compiled and linked with level zero, level one, and level two optimizations.

8 CONCLUSIONS AND FUTURE WORK

In designing and implementing Arachne, our main goal was a threads system supporting efficient thread migration on heterogeneous networks of machines. To achieve this in a reasonable amount of time, we traded complex requirements for flexibility and utility. Because of this, the initial implementation lacks certain features. For example, because Arachne does not currently offer built-in synchronization mechanisms, the user must resort to available OS mechanisms. But these may be overly general and not uniform across hardware platforms. Arachne's design does provide for the incorporation of such features, which we plan to implement in the second phase of the effort. We also intend to provide two mechanisms for obtaining multiple current threads of control on shared-memory multiprocessors. One mechanism is based on a kernel-space threads interface. The other is based on Ariadne's multiprocess design [15], meant for architectures that do not support kernel-space threads.

Besides providing for a thread migration functionality, we also aimed for a robust and practical implementation. When problems we encountered gave rise to multiple solutions, our application-oriented motivation and background led us to favor simpler but reasonably complete solutions over more complex but possibly also more elegant ones. We plan on some immediate performance enhancements, such as the inclusion of Ariadne's constant-cost context-switching mechanism, to circumvent Arachne's overheads in deeply nested situations. This will enable Arachne to provide low-context switching overhead while leaving its powerful heterogeneous-migration framework intact.

We also plan on using our current framework to support object-migration by building global `structs` that are specific to every class definition. In this way, we can have all user threads inheriting from a `Thread` superclass. The thread function will then be the `run()` method of the class and will take no parameters. All parameters will be members of the class. This will not only result in a cleaner interface, but it will also remove the restriction of only `int`, `float`, and `double` thread function arguments (these are currently due to the use of the variable-argument facility in C/C++).

Adding to the above, if me make `app` build intermediate code representation, the need for changing C/C++ (through

TABLE 2
RUNTIMES (IN μSEC) FOR PERFORMING STRAND OPERATIONS

Operation	Sun Sparc 5	Sun Sparc 20	SGI
Strand Create and Invoke	29	22	13
Strand Invoke	3	3	2
Strand destroy	17	9	4

TABLE 3
RUNTIMES (IN μSEC) FOR PERFORMING MIGRATION

Operation	Same Subnet	Across Subnets
Migrate (12 byte ar)	4,304	4,412
Migrate (128 byte ar)	4,896	4,900

adding the three Arachne keywords) will be eliminated. Thread functions can be identified easily, since they are the `run()` methods of classes that inherit from `class Thread`. Strand function definitions and invocations (the keywords `strand` and `call`) can be deduced from looking at function bodies.

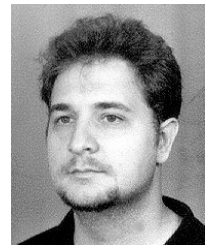
We have not attempted to provide support for templates or exceptions, since these two features have not been finalized in the C++ language definition, and different compilers handle them in different ways or impose different restrictions. Templates and exceptions were also not part of the original C++ grammar that we based our work on. We do believe, however, that `app`'s code modifications do not affect the correctness of code containing templates and exceptions.

ACKNOWLEDGMENTS

This work was supported in part by grants ONR-9310233, ARO-93G0045, and BMDO-34798-MA.

REFERENCES

[1] J. Banks and J. Carson, "Process Interaction Simulation Languages," *Simulation*, vol. 44, no. 5, pp. 225–235, May 1985.
[2] J.S. Chase, F.G. Amador, E.D. Lazowska, H.M. Levy, and R.J. Littlefield, "The Amber System: Parallel Programming on a Network of Multiprocessors," *Proc. Symp. Operating System Principles*, pp. 147–158, 1989.
[3] E. Dijikstra and B.P. Scholten, "Termination Detection for Diffusing Computations," *Information Processing Letters*, vol. 11, no. 1, pp. 1–4, 1980.
[4] I. Foster, C. Kesselman, and S. Tuecke, "The Nexus Approach to Integrating Multithreading and Communication," technical report, Argonne Nat'l Laboratory, 1995.
[5] J. Gomez, E. Mascarenhas, and V. Rego, "The CLAM Approach to Multithreaded Communication on Shared-Memory Multiprocessors: Design and Experiments," *IEEE Trans. Parallel and Distributed Systems*, vol. 9, no. 1, pp. 36-49, Jan. 1998.
[6] J. Gomez, V. Rego, and V. Sunderam, "Scheduling Communication in Multithreaded Programs: Experimental Results," Technical Report TR-97, Purdue Univ., 1997.
[7] J.C. Gomez, V. Rego, and V.S. Sunderam, "Efficient Multithreaded User–Space Transport for Network Computing: Design and Test of the TRAP Protocol," *J. Parallel and Distributed Computing*, vol. 40, no. 1, pp. 103–117, Jan. 1997.
[8] Std. 1003.1c-1995, *Information Technology—Portable Operating System Interface* (*POSIX*)—*Part 1: System Application Program Interface* (*API*) [*C Language*]. IEEE, 1995.
[9] E. Jul, H. Levy, N. Hutchinson, and A. Black, "Fine-Grained Mobility in the Emerald System," *ACM Trans. Computer Systems*, vol. 6, no. 1, pp. 109–133, 1988.
[10] F. Knop, "Software Architectures for Fault-Tolerant Replications and Multithreaded Decompositions: Experiments with Practical Parallel Simulation," PhD thesis, Purdue Univ., Aug. 1996.
[11] K. Li, "IVY: A Shared Virtual Memory System for Parallel Computing," *Proc. Int'l Conf. Parallel Processing*, pp. 94–101, 1988.
[12] B. Malloy and M. Soffa, "Conversion of Simulation Processes to Pascal Constructs," *Software: Practice and Experience*, vol. 202, no. 2, pp. 191–207, Feb. 1990.
[13] E. Mascarenhas, F. Knop, and V. Rego, "ParaSol: A Multi-threaded System for Parallel Simulation Based on Mobile Threads," *Proc. Winter Simulation Conf.*, pp. 690–697, Dec. 1995.
[14] E. Mascarenhas and V. Rego, "Migrant Threads on Process Farms: Parallel Programming with Ariadne," Technical Report TR 95-081, Dept. of Computer Sciences, Purdue Univ., Dec. 1995.
[15] E. Mascarenhas and V. Rego, "Ariadne: Architecture of a Portable Threads System Supporting Thread Migration," *Software Practice and Experience*, vol. 26, no. 3, pp. 327–357, Mar. 1996.
[16] E. Masceranhas, "A System for Multithreaded Parallel Simulation and Computation with Migrant Threads and Objects," PhD thesis, Purdue Univ., Aug. 1996.
[17] J.A. Roskind, "`cpp5.y`: Yacc-Compatible Input File Defining a C++ Grammar," ftp: ics.uci.edu:/pub/gnu/c++grammar2.0.tar.Z.
[18] J.A. Roskind, "`cpp5.l`: Flex Input File Defining a C++ Lexical Analyzer," ftp: ics.uci.edu:/pub/gnu/c++grammar2.0.tar.Z, 1989-1990.
[19] J. Sang, F. Knop, V. Rego, J. Lee, and C. King, "The Xthreads Library: Design, Implementation and Applications," *Proc. 17th Ann. Computer Software and Applications Conf. (COMPSAC '93)*, Nov. 1993.
[20] J. Sang, E. Mascarenhas, and V. Rego, "Mobile-Process Based Parallel Simulation," *J. Parallel and Distributed Computing*, vol. 33, no. 1, pp. 12–23, Feb. 1996.
[21] J. Sang, G. Peters, and V. Rego, "Thread Migration on Heterogeneous Systems via Compile-Time Transformations," *Proc. Int'l Conf. Parallel and Distributed Systems (ICPADS)*, pp. 634–639, 1994.
[22] J. Sang and V. Rego, "A Simulation Testbed Based on Lightweight Processes," *Software, Practice and Experience*, vol. 24, no. 5, pp. 485–505, May 1994.
[23] H.D. Schwetman, "Using CSIM to Model Complex Systems," *Proc. Winter Simulation Conf.*, pp. 246–253, 1988.
[24] B. Steensgaard and E. Jul, "Object and Native Code Thread Mobility Among Heterogeneous Computers," *Proc. ACM Symp. Operating Systems Principles*, pp. 68–78, 1995.



Bozhidar Dimitrov is a PhD candidate and a research fellow in the Computer Science Department of Purdue University. He received his MS from Purdue University in 1996 and his BS in computer science/mathematics from Furman University in 1994. His research interests lie in desktop video conferencing, networking, and threads systems. He is a student member of the IEEE.

Vernon Rego is a professor of computer sciences at Purdue University, West Lafayette. He graduated with an MS and PhD in computer science from Michigan State University (East Lansing) in 1983 and 1985, respectively. His research interests include experimental software systems for parallel and distributed computing and simulation on heterogeneous networks, threads systems, network protocols, and probability and software engineering. He was awarded an IEEE Gordon Bell Prize for his contributions to parallel processing and a 1988 German Research Council award for networking research. He is an editor for *IEEE Transactions on Computers*, an advisory board member of the advanced distributed simulation consortium, and a member of IEEE and ACM.