Achieving Efficiency *and* Portability in Systems Software: A Case Study on POSIX-Compliant Multithreaded Programs

Yasushi Shinjo, *Member*, *IEEE*, and Calton Pu, *Senior Member*, *IEEE*

Abstract—Portable (standards-compliant) systems software is usually associated with unavoidable overhead from the standards-prescribed interface. For example, consider the POSIX Threads standard facility for using thread-specific data (TSD) to implement multithreaded code. The first TSD reference must be preceded by pthread_getspecific(), typically implemented as a function or macro with 40-50 instructions. This paper proposes a method that uses the runtime specialization facility of the Tempo program specializer to convert such unavoidable source code into simple memory references of one or two instructions for execution. Consequently, the source code remains standard compliant and the executed code's performance is similar to direct global variable access. Measurements show significant performance gains over a range of code sizes. A random number generator (10 lines of C) shows a speedup of 4.8 times on a SPARC and 2.2 times on a Pentium. A time converter (2,800 lines) was sped up by 14 and 22 percent, respectively, and a parallel genetic algorithm system (14,000 lines) was sped up by 13 and 5 percent.

Index Terms—Performance, portability, threads, software libraries, concurrent programming, runtime specialization, thread-specific data.

1 Introduction

Portability of systems software means the adoption of a standard application program interface (API), e.g., POSIX [27] for different flavors of Unix. Accordingly, every implementation of such standard packages must implement the most general interpretation of each procedure and function. Otherwise, a so-called standard API would not work the same way across different platforms. Consequently, each procedure invocation of a standard API carries the full overhead of the required general implementation, even though only a subset of functionalities is used in most cases. This has been accepted as an "inevitable" cost trade-off of portability versus performance.

The very nature of the standardization process favors a minimalist API with general facilities since, otherwise, the proliferation of special cases would make a standard too big for full compliance in practice. The need for standard APIs to be small and general makes it challenging to achieve *both* performance and portability in systems software. Since portability dictates the use of standard library functions in an API, one way to meet the challenge is through the management of multiple implementations of similar library functions. Sometimes, these alternative implementations are called "lightweight" since they carry a much smaller overhead for common simple cases. For example, SunOS includes two implementations that convert a string representation into an integer value. One is `strtol()`, which is a general implementation that can deal with nondecimal representations and returns the address of the first nonnumerical character. The other is `atoi()`, which is a lightweight implementation that can deal with only decimal representations and cannot return the address of the first nonnumerical character. Providing multiple implementations makes library maintenance more difficult and decreases portability. Another example is the GNU MP (Multiple Precision) library, which contains assembly and C versions of code for arbitrary precision arithmetic operation. The assembly version is fast and nonportable, while the C version is slower and portable.

In addition to the trade-off between portability and performance, there is a similar trade-off between simplicity and performance in systems software. Sometimes performance requirements make the programmer decide to sacrifice simplicity by choosing faster library functions within a standard API. For example, to write a multithreaded program, a programmer often chooses locking instead of using Thread Specific Data (TSD) because TSD has been considered expensive for a long time [18], [24]. Using TSD can simplify the program and can make it less likely to deadlock [25]. However, the POSIX Threads package specifies the invocation of the function `pthread_getspecific()` before a TSD variable becomes accessible. Typically, this call adds 40-50 instructions, while locking can be implemented with fewer instructions.

Instead of providing multiple implementations, we propose a method to recover the performance of generic library functions in a standard API through *specialization*. Our method also recovers the performance of simple but

- *Y. Shinjo is with the Department of Computer Science, University of Tsukuba, 1-1-1 Tennoudai, Tsukuba, Ibaraki 305-8573, Japan. E-mail: yas@cs.tsukuba.ac.jp.*
- *C. Pu is with the College of Computing, Georgia Institute of Technology, 801 Atlantic Drive, Atlanta, GA 30332-0280. E-mail: calton@cc.gatech.edu.*

Manuscript received 27 July 2004; revised 19 Feb. 2005; accepted 25 Apr. 2005; published online 15 Sept. 2005.
Recommended for acceptance by R. Schlichting.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-0155-0704.

slow library functions in the standard API. Specialization is a well-established technique to improve the performance of systems software by eliminating unnecessary code automatically [23], [28]. Tool-based specialization [5] eliminates code made unnecessary by *invariants*. Specialization has two main advantages. First, the maintainers of standard library functions do not have to provide a number of fixed alternate implementations since specialization generates the most appropriate implementation for a given set of invariants. Second, programmers do not have to choose fast but complex library functions since specialization creates automatically correct code from the simple source code plus invariant specifications.

The main contribution of this paper is an experimental demonstration of the specialization method to eliminate the apparently unavoidable standard API overhead from production POSIX-compliant multithreaded programs without thread-aware compiler or memory mapping support. (See Section 6 for comparisons with previous work with such support.) Our method *adds* invariants to enable automated runtime specialization using Tempo, a program specializer for the C language [5]. In this paper, we show invariants in POSIX-compliant multithreaded programs with TSD. The POSIX Threads package specifies the invocation of the function `pthread_getspecific()` before a TSD variable becomes accessible. Typically (see Section 2.3 for a more detailed discussion), this call adds 40-50 instructions to a simple global reference due to the dynamic binding of TSD. In many applications, the thread does not require a dynamic binding and fixed TSD suffices. Although the TSD address is unknown at compile-time, runtime specialization is able to use the fixed TSD invariant to transform each invocation of `pthread_getspecific()` into a simple constant reference.

Our method follows the specialization-based method developed by Marlet et al. [19], [20]. This method consists of two steps: first, develop a correct multithreaded program based-on a single-threaded program by adding TSD access and, second, improve performance through specialization. We apply the method to two system library routines: a small module, a random number generator (10 lines of C code, as described in Section 3), a medium module, and a time converter (2,800 lines of C code in 14 files, as described in Section 5). In addition, we use a parallel genetic algorithm program (GALOPPS, Section 4) to demonstrate the performance gains in a large program (14,000 lines of C code in 40 files).

Experimental results confirm significant performance improvements while preserving source code POSIX compliance. The random number generator with TSD is sped up by a factor of 4.8 on a SPARC, and factor of 2.2 on a Pentium. The parallel genetic algorithm program is sped up by 13 percent on the SPARC and 5.1 percent on the Pentium. The time converter is sped up by 14 percent on the SPARC and 22 percent on the Pentium. Consequently, the specialized TSD code has an efficiency level comparable to locking-based code and much better scalability for multiprocessors (see Sections 3.4.3, 4.3.3, and 5.4.3). Our results strongly suggest that TSD with specialization is a good technique for the development of multithread software that needs to be efficient, portable (standards-compliant), and scalable.

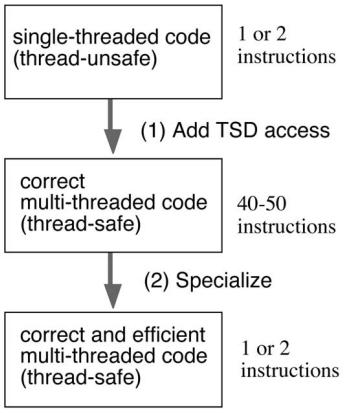


Fig. 1. Developing efficient multithreaded code with TSD and specialization.

The rest of the paper is organized as follows: Section 2 outlines the constraints introduced by the POSIX Threads standard and the "unavoidable" overhead associated with TSD. Sections 3, 4, and 5 describe the application of our method to a random number generator, a parallel genetic algorithm that uses the random number generator, and a time converter, respectively. For each, we summarize the problem, apply specialization using Tempo, and show execution performance gains as well as good multiprocessor scalability. Section 6 summarizes related work and Section 7 concludes the paper.

2 Systematic Method to Specialize POSIX-Compliant Multithreaded Programs

We first outline a specialization-based method for developing multithreaded programs with TSD. Second, we show the performance problem of general implementations of POSIX library functions. Third, we illustrate the performance problem of POSIX TSD through a simple microbenchmark. Fourth, we summarize the principles of operation and the usage of the partial evaluator Tempo. Finally, we explain the experimental environments used in the following sections.

2.1 Method for Developing Correct and Efficient Multithreaded Programs with TSD and Specialization

In a previous paper [23], we applied the specialization-based method developed by Marlet et al. [19], [20] to the RPC stack. We eliminated the overhead of copying and layering by using compile-time specialization. In this paper, we will apply this method to multithreaded parallel programs by using runtime specialization and, in so doing, eliminate the overhead of TSD access.

Fig. 1 shows our method for creating a correct multithreaded program based on a single-threaded program. The first step of our method uses TSD to contain thread-specific variables instead of lock-based mutual exclusion because using TSD is much easier and simpler than using locking. Furthermore, programs with TSD are potentially scalable in

```
int atoi( char *nptr ) {
  return strtol (nptr, NULL, 10);
}

int strtol( char *nptr,
  char **endptr, int base ) {
  return __strtol_internal( nptr, endptr,
    base, 0 );
}

int __strtol_internal(char *nptr,
  char **endptr, int base, int group ) {
  if(base < 0 || base == 1 || base > 36){
    errno = EINVAL ;
    return 0;
  }
  s = nptr ;
  ...
  for( ; c != '\0' ; c = *++s ){
    if( c >=  '0' && c <=  '9' )
      c -=  '0'  ;
    else
    ...
    i *= (unsigned long  int) base;
    i += c;
  }
  ...
  if( endptr != 0 )
    *endptr = (char  *) s;
  return negative ? -i : i;
}
```

Fig. 2. The library functions `atoi()` and `strtol()` in GNU Libc 2.2.

symmetric multiprocessors (SMPs). However, using TSD adds 40-50 instructions in a straightforward implementation, while locking can be implemented with fewer instructions (for details see Section 2.3).

The second step of our method improves multithreaded code performance through specialization. By carefully exploiting the runtime invariants of TSD code, we are able to reduce the overhead of using TSD and get almost the same performance as the unsafe version (see Section 2.4 and the performance evaluation experiments in Sections 3, 4, and 5). Our method is fairly general and independent of particular software tools. For example, we use the Tempo specializer in our experiments, but other program specializers could be used (see related work in Section 6).

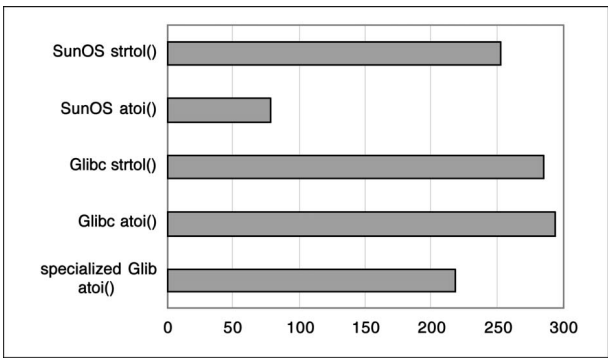


Fig. 3. The execution times of simple library functions that convert a string to an integer in nanoseconds.

```
pthread_key_t tsdkey1 ;
iret_tsd() {
  int *p ;
  p = pthread_getspecific( tsdkey1 );
  return( *p );
}
```

Fig. 4. A function returning a value of a TSD variable.

2.2 The Performance Problem of General Library Functions

We illustrate the performance penalty introduced by the POSIX standard with two simple library functions `atoi()` and `strtol()`. Both functions convert a string representation into an integer value, but `strtol()` is more general than `atoi()`. The function `strtol()` takes two additional arguments: the pointer to the last nonnumeric character in the string and the base for conversion. Fig. 2 is an actual code fragment from GNU Libc (Glibc). The code of `atoi()` passes two constants to `strtol()`: `NULL` for `endptr` and `10` for `base`. Therefore, the implementation of function `strtol()` includes unnecessary checking of parameters when it is called from `atoi()`. This is an example of *invariants* that remain unchanged during program execution. Specialization can take advantage of these invariants to eliminate redundant code. In Section 2.4, we show how invariants are used by Tempo in compile time specialization to eliminate this kind of overhead while preserving the generality of the source code.

Fig. 3 compares the execution times of `atoi()` and `strtol()` to show the performance penalty in Glibc and the gains from specialization (to be explained in Section 2.4). All the execution times were measured on an UltraSPARC II running SunOS 4.7 for the string argument "1." Since the general implementation of `atoi()` of GNU Libc calls `strtol()` internally, `atoi()` is slower than `strtol()`.

2.3 The Performance Problem of POSIX TSD

We explain the TSD performance penalty introduced by the POSIX Threads standard with the help of a simple microbenchmark program `iret_tsd()` shown in Fig. 4. The microbenchmark is compared to a non-TSD program `iret_extern()` shown in Fig. 5 since both simply return an integer value. The `iret_tsd()` follows the POSIX-prescribed way to access TSD variables. At program initialization time, `pthread_key_create()` is called once to create a key for the TSD variable. At thread initialization time, memory for the TSD variable is allocated and its address is bound to the key by `pthread_setspecific()`. Thereafter, each time the function using TSD is called,

```
int var1 ;
iret_extern() {
  return( var1 );
}
```

Fig. 5. A function returning a value of an external variable.

TABLE 1
Execution Times of a Simple Function that Returns the Value of an External Integer Variable or a TSD Variable

	Execution time in nanoseconds		Slow down	Number of instructions	
Environment	extern	TSD		extern	tsd
SunOS 5.7/SPARC64 IV 450MHz	6.7	111	17	2	48
SunOS 5.8/UltraSPARC II 480MHz	10.5	161	15	2	48
Linux Glibc / Pentium 550 MHz	15	80	5.3	1	49

`pthread_getspecific()` is invoked to initialize the base pointer to the TSD variable specified by the key. From that moment, the TSD variables become accessible from the program. In `iret_tsd()`, the prescribed `pthread_getspecific()` call introduces significant overhead since it contains 40-50 instructions while the simple global reference in `iret_extern()` compiles into one or two instructions.

Table 1 compares the overhead of `pthread_getspecific()` to a simple external variable access. The multithreaded program with TSD is slower by a factor of 17 on a SPARC64 running SunOS, 15 on a UltraSPARC running SunOS, and 5.3 on a Pentium running Linux (for repeated access to a warm cache with a fixed size stack). This repeated cost due to `pthread_getspecific()` prescribed by POSIX is really unnecessary since the value of `tsdkey1` is the same for every invocation of `pthread_getspecific()`, and the result it returns is also the same for each thread. This is an example of *an invariant* that remains unchanged during program execution. Specialization can take advantage of this invariant to eliminate redundant code. In Section 2.4, we show how invariants are used by Tempo in runtime specialization to eliminate this kind of overhead while preserving standard compliance at the source code level.

There are some system-specific ways to reduce this kind of overhead. First, the POSIX standard allows the function `pthread_getspecific()` to be implemented as a macro, typically in a system-specific manner. A macro implementation eliminates function invocation instructions, but it is still more expensive than a simple global variable access. In practice, `pthread_getspecific()` is a real function in many systems, such as SunOS, Linux, FreeBSD, and Irix. Second, it is feasible to include special compiler support for such functions. For example, in Section 6, we briefly discuss the Thread Local Storage facility in Microsoft's Visual C++. Third, it is possible to use special kernel support for memory mapping to implement user-level TSD, as is also discussed in Section 6. In contrast, our method does not rely on special compiler or kernel memory mapping support.

2.4 Compile Time and Runtime Specialization with Tempo

Tempo [5] is a well-known C (and Java) program specializer based on partial evaluation (in this paper, we use the terms partial evaluator and program specializer interchangeably). Tempo supports both compile time and runtime program specialization through invariants and hints specified in C and ML. Fig. 6 illustrates the runtime specialization process in Tempo. Tempo takes the source code plus *hints* that specify invariants in the source code. Using a program analysis technique called Binding Time Analysis [12], Tempo discovers the program variables whose values only depend on information contained in the invariants.

If these static/invariant values are known at compile-time, we perform compile time specialization to generate specialized C code. On the other hand, if these static/invariant values are computed at runtime, we perform runtime specialization to generate C code called a *template* that has *holes* to be filled in by a *runtime specializer* in C (also generated by Tempo) (Fig. 6). The runtime specializer can be seen as a linker/loader specialized for this template. It runs at program load/initialization time, evaluates the static/invariant part of the program, and fills the holes with the computed invariant values (Fig. 6).

We use the function `atoi()` in Fig. 2 to illustrate compile-time specialization in the Tempo partial evaluator.

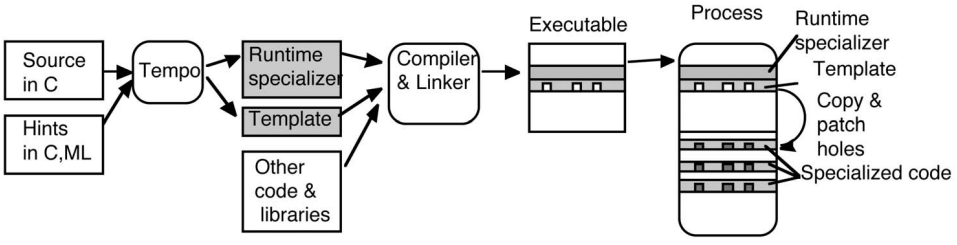


Fig. 6. Runtime specialization in Tempo.

```
entry_point := "atoi(D)";
post_inlining := true ;
post_do_inline := ["__strtol_internal",
  "strtol", "atoi" ];
post_inlining_mode := FLAT ;
```

Fig. 7. Hints in ML for specializing `atoi()`.

Fig. 7 shows hints (written in ML) for specializing the function `atoi()` in Fig. 2. The keyword `entry_point` specifies the function to be specialized. The character "D" in the parentheses means that those arguments of the function can vary (dynamic). The keywords `post_inlining`, `post_do_inline`, and `post_inlining_mode` are declared to force inlining on all the functions being specialized. No special hints in C are required for this specialization, and two empty declarations are included in Fig. 8 for completeness.

Tempo takes the source program in Fig. 2 plus the hints in Fig. 7 and Fig. 8, and produces the specialized code shown in Fig. 9. Compared with the original code in Fig. 2, the specialized code omits some if-branches and function invocations. Therefore, the specialized code is faster than the original general code in Fig. 2. Fig. 3 shows the performance improvement by this compile-time specialization. The specialized `atoi()` is faster than the original `atoi()` by 25 percent.

We again use the simple function in Fig. 4 to illustrate runtime specialization in the Tempo partial evaluator. Unlike explicit constants in `atoi()`, the return value of `pthread_getspecific()` is unknown at compile-time, so only runtime specialization is able to use the invariant for each thread. Fig. 10 and Fig. 11 show hints in ML and C for specializing the simple function in Fig. 4 for each thread. The keyword `entry_point` specifies the function to be specialized, which is "`iret_tsd`." The keyword `static_locations` specifies invariant variables and invariant fields in structures. The TSD key `tsdkey1` is declared an invariant. The keyword `external_functions` specifies a candidate for evaluation at specialization time, which is "`pthread_getspecific`." This means that "`pthread_getspecific`" can be invoked once at specialization time if all of its arguments are invariant, and the invariant result can be reused at every invocation time. Finally, the keyword `lift_all` tells the runtime specializer to evaluate pointer values including the return value of `pthread_getspecific()`. Without this parameter, Tempo considers fetching the pointer values lightweight and leaves them in the specialized code. With this parameter, Tempo moves them to the runtime specializer if they are static.

Tempo takes the source program in Fig. 4 plus the hints in Fig. 10 and Fig. 11, and produces a template and a runtime specializer in C. Fig. 12 shows the decompiled code that is generated by the runtime specializer based on the template. The most significant difference between the original code and the specialized code is that the specialized code includes a constant reference (the address of an external variable `H0`) instead of the function invocation. `H0` is a hole in the template. The runtime specializer fills the hole `&H0` with the value returned by `pthread_getspecific()`. Consequently, the specialized program runs with one or two instructions that load an integer value from a known absolute address. In other words, the specialized code will execute as fast as the function in Fig. 5.

In the above specialization example, we use the address of TSD (the value returned by `pthread_getspecific`) as the invariant. By specifying this invariant in hints, Tempo specializes the code automatically. In the rare case where a program changes the address of TSD by invoking `pthread_setspecific()` after initialization, the program can always be translated into an equivalent one that does not change the address of

```
void set_analysis_context(const char **str)
{
}
void set_specialization_context(){
}
```

Fig. 8: Hints in C for specializing `atoi()`.

```
int _Gatoi_1_0_0(const char *nptr) {
  base = 10;
  s = nptr ;
  ...
  for( ; c != '\0' ; c = *++s ){
    if( c >=  '0' && c <=  '9' )
      c -=  '0'  ;
    else
    ...
    i *= (unsigned long  int) base;
    i += c;
  }
  ...
  return negative ? -i : i;
}
```

Fig. 9: The specialized code of `atoi()`.

```
entry_point := "iret_tsd()";
static_locations := ["tsdkey1"];
external_functions :=
  EVALUATE["pthread_getspecific"];
lift_all := true ;
```

Fig. 10. Hints in ML for specializing `iret_tsd()`.

```
void set_analysis_context(){
}
int var1 ;
void *pthread_getspecific(unsigned key) {
  return( &var1 );
}
```

Fig. 11. Hints in C for specializing `iret_tsd()`.

```
int H0;
int tmp_iret_tsd_1(){
  return *((int *)(&H0));
}
```

Fig. 12. The specialized code of `iret_tsd()`.

TSD, manually or semiautomatically. The key idea is that we allocate a TSD variable that holds the address of the original TSD variable. This translation corresponds to the following translation for a normal variable.

Before translation:

```
int *p ; /* corresponds a TSD variable */
p = malloc(sizeof(int));
      /* pthread_setspecific(),
        initialization */
...
p = malloc(sizeof(int));
      /* pthread_setspecific(), changing */
```

After translation:

```
#define p (*pp)
  int **pp ; /* corresponds a TSD variable */
  pp = malloc(sizeof(int *));
        /* pthread_setspecific()
           initialization */
  p = malloc(sizeof(int));
        /* the content of pp is invariant */
  ...
  p = malloc(sizeof(int));
        /* the content of pp is invariant */
```

Although this translation introduces an indirection overhead, this overhead will be eliminated by specialization.

Our method introduces some additional source code processing time. In the parallel GA system GALOPPS (see Section 4), for example, running Tempo required 7.5 seconds on a Sun Enterprise 450. This time includes analyzing 230 lines of C code, generating 450 and 700 lines of the template and runtime specializer, and compiling these generated programs. Building the entire application program that consists of 14,000 lines of C files required 40 seconds. Therefore, using Tempo added 20 percent more time for building the application.

In our method, we envision a program specializer such as Tempo being used by programmers or maintainers who write or modify source code. A programmer finds the appropriate specialization invariants and directs Tempo to generate specialized code. Several techniques have been introduced to facilitate the task of program specialization, including Specialization Classes [7] and Specialization Scenarios [6], [15], [16], [17]. Furthermore, if a specialization facility is integrated into gcc and a dynamic linker, we can generate templates and runtime specializers with gcc, and automatically use specialized code through the dynamic linker.

Although our method requires accessing the source code, it is applicable to non-open-source products. For open-source software, anyone can use our method. For non-open-source software, vendors with access to source code can apply this method. Compiled object code of the templates and runtime specializers are shipped to customers. The customers first call the runtime specializers, generate the specialized code, and then use the specialized code.

2.5 Experimental Environments

In the following sections, we measured the performance on two SPARC SMPs (symmetric multiprocessors) and a Pentium SMP. These represent two distinct and dominant architectures. One SPARC SMP has eight SPARC64 IV 450 MHz processors [32], each with an 8MB external cache, and a total of 6,114 MB of shared main memory. Its operating system is Solaris 7 (SunOS 5.7). The other SPARC SMP (Sun Enterprise 450) has four UltraSPARC II 480 MHz processors [33], each with an 8 MB external cache, and a total of 2,048 MB of shared main memory. Its operating system is Solaris 8 (SunOS 5.8). The Pentium SMP has eight Pentium III 550 MHz processors, each with a 2 MB external cache and a total of 3,890 MB of shared main memory. Its operating system is Red Hat Linux 6.2 with kernel 2.2.16 SMP and GNU Libc Version 2.1.3 (Linux Thread 0.7).

We use Tempo Version 1.194 (1999/04/27) on the SPARCs, 1.202 (2000/07/10) on the Pentium, and gcc version 2.95.2 (1999/10/24 release). All measured programs including the time converter in Section 5 are compiled with the optimization flag "`-O2 -malign-functions=32`" and "`-mcpu=v8`" on the SPARCs and "`-O2 -malign-functions=5`" on the Pentium.

In the experiments, we measured the peak performance for speedup and the mean performance for throughput. The peak performance ignores the effects of cache refill misses, but is representative of how these functions would be used (e.g., in tight inner loops) [26]. During each execution, the function being measured is invoked repeatedly from 1,000 to 1,000,000 times. The number of trials is high enough to reach several tens of milliseconds to several hundreds of milliseconds. The resulting times were obtained using the Unix `gettimeofday()` system call in multiuser mode, and were divided by the number of iterations. The same benchmark programs were executed 100 times. We used a single thread to measure the peak performance for speedup and multiple threads to measure the mean performance for throughput.

In addition to speedup and throughput, we measured the break-even point. The break-even point is the number of times the specialized code must be executed before the cost of runtime specialization (at initialization) is recovered. This number is calculated from the gain of specialization and the time to generate the specialized code.

We also tuned the data and instruction layout carefully to minimize the influence of instruction layout in our experiments. UltraSPARC fetches up to four instructions per clock from an aligned group of eight instructions [33]. When the fetch address mod 32 is equal to 20, 24, or 28, three, two, or one instruction(s), respectively, will be added to the instruction buffer. This means that the start address of a function affects the execution times. For some functions, we tried all the combinations of the start addresses. We found that the alignment at mod 32 usually minimizes the fluctuations of execution time due to instruction layout. A full discussion of the execution code layout is beyond the scope of this paper. We performed this tuning for both the original code and the specialized code.

```
struct random *random_gettsd();

float randomperc(){
  struct random *sr; int cur ;
  sr = random_gettsd();
       /* calls pthread_getspecific() */
  cur = sr->sr_jrand + 1 ;
  if (cur >= 55) {
    cur = 1 ;
    advance_random(sr);
  }
  sr->sr_jrand = cur ;
  return((float)sr->sr_buf[cur]);
}
```

Fig. 13. A random number generator `randomperc()` with a TSD variable.

3 Specializing a Random Number Generator with TSD

In this section, we specialize the random number generator with TSD in the GALOPPS parallel GA system [14], and show the speedup due to the specialization. The sequential and parallel GA each consume a massive amount of random numbers as do simulation applications, such as Monte-Carlo methods. This section deals with the effect of specialization for a single function. Section 4 shows the impact of specialization on the entire parallel GA system.

3.1 Implementing a Random Number Generator with TSD

We extended the function `randomperc()` in GALOPPS Version 3.2.2, as shown in Fig. 13. This function uses a subtractive method in [13], and returns a single random number between 0.0 and 1.0. The original version used two file-scope variables to save the internal state, and using these file-scope variables is unsafe for multithreaded programs. We packed the file-scope variables into a TSD structure obtained by `random_gettsd()`. Internally, the function `random_gettsd()` invokes `pthread_getspecific()` and returns the result of this call. In addition, `random_gettsd()` handles the allocation of the memory for the structure `random`, the initialization of the structure, and the address registration through `pthread_setspecific()` when called the first time in the thread. It also creates a TSD key when it is called the first time in the process through `pthread_once()`. Then, the function `randomperc()` tries to get a random number in the buffer `sr->sr_buf[]`. If `cur(sr->sr_jrand + 1)` is greater than or equal to 55, the buffer is empty. In this case, the function calls an external function `advance_random()` to refill the buffer. Finally, it fetches and returns a single random number.

3.2 Hints to the Partial Evaluator Tempo

Fig. 14 shows selected hints (written in ML) for specializing the random number generator. The other hints are the same as those in Fig. 10. The keyword `entry_point` specifies the function to be specialized ("`randomperc`"). The function `random_gettsd()` is marked to be evaluated at specialization time by the runtime specializer. Similarly,

```
entry_point := "randomperc()";
static_locations := [];
external_functions :=
  EVALUATE[ "random_gettsd" ];
```

Fig. 14. Hints in ML for `randomperc()`.

```
struct random random_data;
set_analysis_context(){
}
struct random *random_gettsd(){
    return( &random_data );
}
```

Fig. 15. Hints in C for `randomperc()`.

Fig. 15 shows hints in C for specializing the random number generator.

3.3 The Specialized Program

Tempo takes the source code outlined in Section 3.1 and the hints described in Section 3.2, to produce a template and a runtime specializer in C. Fig. 16 shows what the specialized program might look like if generated in C. Similar to Fig. 12, the specialized code includes a constant reference (the address of an external variable `H0`) instead of the invocation of the function `random_gettsd()`. The runtime specializer fills the hole `H0` with the result of invoking `random_gettsd()`, and the holes `H1` and `H2` with the value of `sr` and the address of `sr->sr_buf`. The size of the specialized code was 96 bytes on the SPARCs and 72 bytes on the Pentium.

3.4 Experimental Results

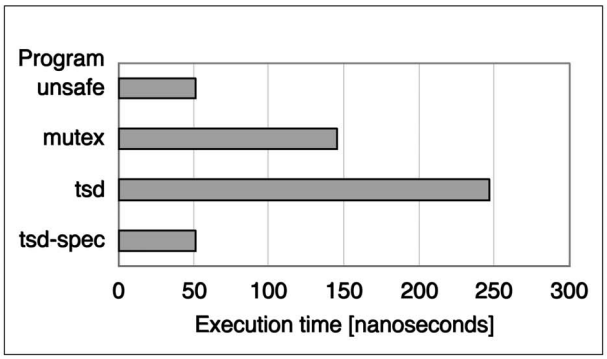
3.4.1 Speedup

First, we measured the peak execution times of the random number generator to assess the speedup due to specialization. We used the following versions for comparison:

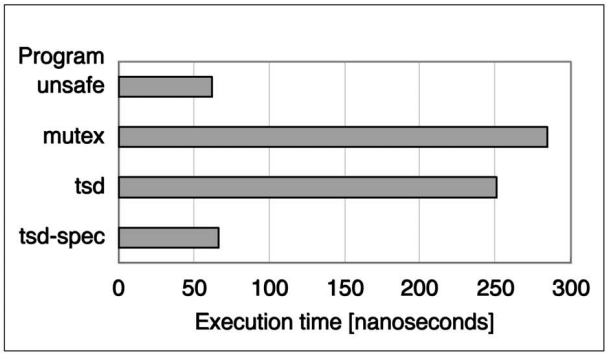
- **unsafe**. The original function in GALOPPS.
- **mutex**. The intermediate state protected with a mutex.
- **tsd**. The random number generator with TSD (Section 3.1).
- **tsd-spec**. The specialized version of **[tsd]** (Section 3.3).

```
int H0, H1, H2;
float randomperc() {
  struct random *sr;
  int cur;
  sr = (struct random *)(&H0);
  cur = sr->sr_jrand + 1;
  if (55 <= cur) {
    cur = 1;
    advance_random((struct random *)(&H1));
  }
  sr->sr_jrand = cur;
  return (float)((double *)(&H2))[cur];
}
```

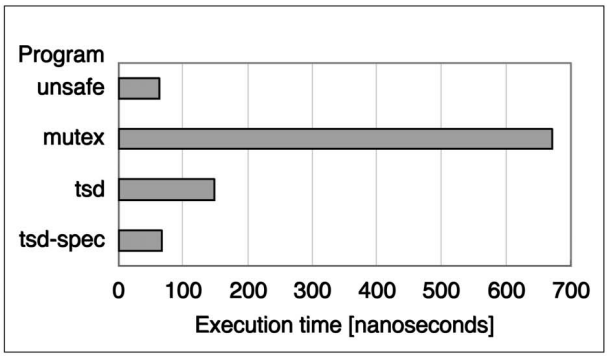
Fig. 16. The specialized random number generator.



(a)



(b)



(c)

Fig. 17. Execution time to generate a single random number. (a) SunOS/SPARC64 IV 450MHz. (b) SunOS/UltraSPARC II 480 MHz. (c) Linux/Pentium III 550 MHz.

Fig. 17 summarizes the execution times to generate a single random number including the outer loop. The TSD version is slower than the unsafe version by a factor of 4.8 on the SPARC64 IV running SunOS, 4.0 on the UltraSPARC II running SunOS, and 2.3 on the Pentium running Linux. Runtime specialization improves the performance of the TSD version by a factor of 4.8 on the SPARC64 IV, 3.8 on the UltraSPARC II, and 2.2 on the Pentium. This means that the specialized TSD code achieves nearly the maximum improvement possible (almost the same performance as the unsafe version).

3.4.2 The Breakeven Point

Table 2 shows the breakeven points of specializing the random number generator. On the SPARC64, we see the gain is 0.196 microseconds by comparing the TSD and specialized versions of the random number generator, and it takes 4.3 microseconds for runtime specialization. Therefore, the breakeven point is 22. Similarly, the breakeven points are 21 and 16 on the UltraSPARC and the Pentium, respectively. We conclude that runtime specialization will produce significant performance savings in moderately frequently executed small-size modules.

3.4.3 Scalability

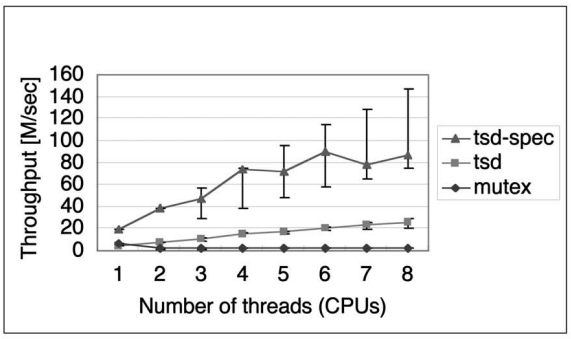
The results of parallel processing of the benchmark program are summarized in Fig. 18. The x-axis is the number of worker threads. The y-axis is throughput in millions of random numbers generated per second. For each point, the benchmark program is executed 100 times. This figure shows the average throughput, with the error bar representing 90 percent of samples. The mutex version shows declining throughput even for two processors. In contrast, the TSD versions are scalable (approximately linearly) for up to eight CPUs (four with UltraSPARC II). Although it is a microbenchmark, the test program shows a potential bottleneck due to locking in mutex. Another potential source of overhead is the shared state variables among threads and physical CPUs, which may increase the number of cache misses as the number of CPU increases.

4 Application: A Parallel GA System

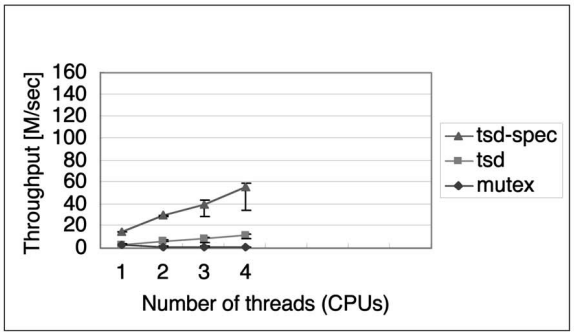
In Section 3, we described the specialization of a simple function and its microbenchmark, with the improvement by a factor of 4.8 on the SPARC64, 3.8 on the UltraSPARC, and 2.2 on the Pentium. In this section, we use a parallel GA system to show the impact of such specialization on the performance of a large application program.

TABLE 2
Breakeven Points of Specializing the Random Number Generator

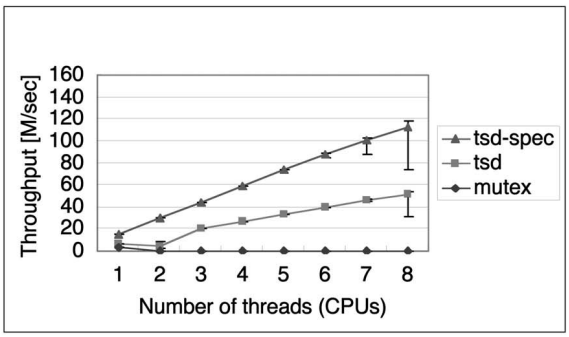
Processor	Gain (in microseconds)	Runtime Specialization (in microseconds)	Breakeven Point
SPARC64	0.196	4.3	22
UltraSPARC	0.185	3.8	21
Pentium	0.081	1.3	16



(a)



(b)



(c)

Fig. 18. The throughput of random number generators (average and 90 percent range as error bars). (a) SunOS/SPARC64 IV 450MHz. (b) SunOS/UltraSPARC II 480 MHz. (c) Linux/Pentium III 550 MHz.

4.1 Parallel GALOPPS System

Genetic algorithms mimic the process of evolution in a population. Evolution happens as individuals in the population reproduce, and random mutations change an offspring. When individuals in a population reproduce, two parents are selected, and portions of the parents' chromosomes are combined to generate the chromosomes of their offsprings (crossover). Genetic algorithms use random number generators extensively when combining chromosomes as well as selecting parents and simulating mutations. For our large program benchmark, we chose the GALOPPS system [14], a parallel genetic algorithm system based on the isolated subpopulation model. In this model, each processor operates independently on an isolated subpopulation (Fig. 19). After several generations, the processors share their best individuals with the other processors (migration). In GALOPPS, each processor computes several populations and exchanges the best individuals through checkpoint files or PVM messages.

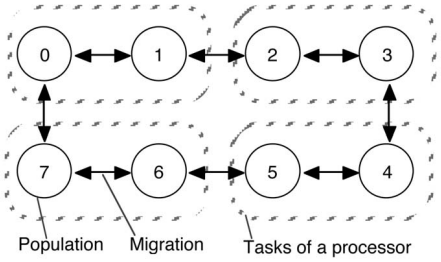


Fig. 19. A parallel genetic algorithm based on the isolated population model.

We parallelized the checkpointing version of GALOPPS using POSIX Threads. First, we created a worker thread for each "processor" in GALOPPS. Each thread operates several populations. Second, we allocated shared memory to all the populations, so each thread can exchange the best individuals through the shared memory instead of checkpoints.

4.2 Specializing the Crossover Function in GALOPPS

We chose the `crossover()` function shown in Fig. 20 for specialization since it consumes more than half of the random numbers. The `crossover()` function makes 60 percent of invocations to the `rnd()` function, which in turn makes 90 percent of the invocations to `randomperc()`, already specialized in the previous section (shown in Fig. 16). Fig. 21 shows selected hints written in ML for specializing the function `crossover()`. The other hints in ML and C are analogous to those in Fig. 14 and Fig. 15, respectively.

```
void crossover(...){
  if(...){
    ...
    for (i = 0; i < numfields; i++)
      randomlst[i] = rnd( 0, 1 );
    ...
  }
   ...
}
int rnd(int low, int high){
  if (low >= high)
    i = low;
  else {
    if (high <= 65535L)
      i = (randomperc()*(high-low+1))+low;
    else
      i = (double_randomperc()*
            (high-low+1))+low;
    if (i > high)
      i = high;
  }
  return (i);
}
```

Fig. 20. The overview of `crossover()` and `rnd()`.

```
entry_point := "crossover(D,D,D,D,D,D)";
```

Fig. 21. Hints in ML for specializing `crossover()`.

Tempo takes the source code of the functions `crossover()`, `rnd()`, and `randomperc()`, plus the hints in ML and C (Fig. 21, Fig. 14, and Fig. 15), and produces a template and a runtime specializer. An overview of the specialized code is shown in Fig. 22. The code is specialized with respect to the function `crossover()`. The specialized `crossover()` function calls the specialized `rnd()` function, which in turn calls the specialized `randomperc()` function (Fig. 16). From the specialized `rnd()` function, two "`if`" branches, two integer operations, and two integer-to-float conversions are eliminated. The size of the specialized code was 1,004 bytes on the SPARCs and 777 bytes on the Pentium.

4.3 Experimental Results

We measured the performance of the parallel GALOPPS in the experimental environments and using the measurement method described in Section 2.5. We ran the parallel GALOPPS to solve the traveling salesman problem example included in the GALOPPS 3.2.2 distribution and used the same parameters and settings as in the example. The mutations were permutations among 20 cities to find the optimal path. Good mutations reduced the total distance traveled between the 20 cities. The number of subpopulations was set to eight and each population had 100 individuals. We created 1, 2, 4, or 8 worker threads, so each thread computed 8, 4, 2, or 1 subpopulation. Each thread exchanged migrants every four generations, and repeated this cycle 10 times. The probability of crossover and mutation was set to 0.9 and 0.2, respectively.

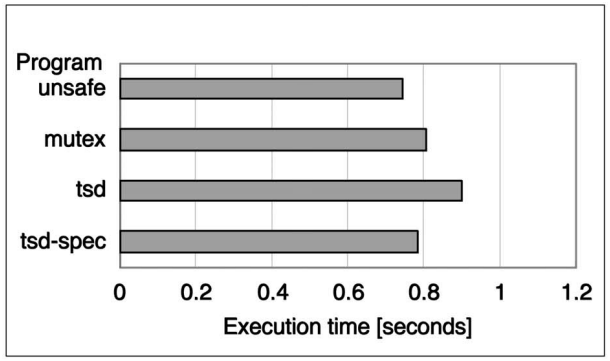
4.3.1 Speedup

First, we measured the peak execution times to obtain the speedup by specialization using a single thread. We used the same random number generators for comparison as in Section 3.4.1. Fig. 23 summarizes the execution times of these versions. On the SPARC64 (Fig. 23a) and the UltraSPARC (Fig. 23b), specialization improves performance by 13 percent and 8 percent, respectively, and the specialized TSD version becomes faster than the mutex version and comparable to the unsafe version. On the Pentium (Fig. 23c), the TSD version is already faster than the mutex version since locking appears to be more expensive. Specialization improves the TSD version performance by another 5.1 percent. We observe that on the Pentium, the specialized TSD version is even faster than the original unsafe version. This is because the specialization eliminates parameter passing and two "`if`" branches, as described in Section 4.2.

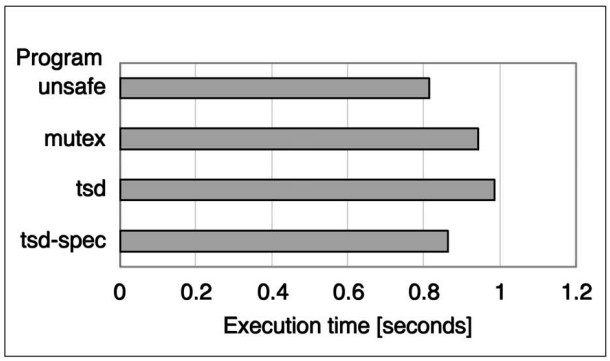
```
void tmp_crossover(...){
  if(...) {
    ...
    for (i = 0; i < numfields; i++)
      randomlst[i] = tmp_rnd();
    ...
  }
   ...
}

int tmp_rnd() {
  i = (tmp_randomperc() * 2.0) + 0.0;
  if (i > (int)(&H10) )
    i = (int)(&H11);
  return( i );
}
```

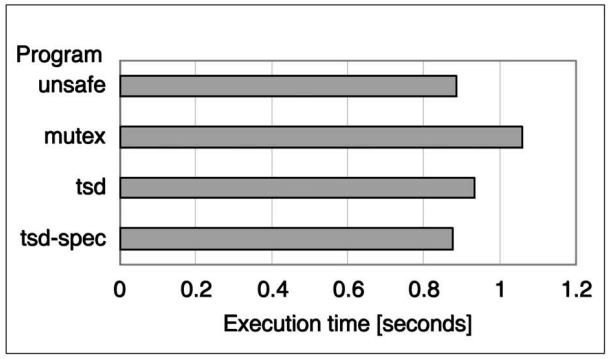
Fig. 22. The specialized code of `crossover()` and `rnd()`.



(a)



(b)



(c)

Fig. 23. Execution time of a parallel genetic algorithm. (a) SunOS/SPARC64 IV 450MHz. (b) SunOS/UltraSPARC II 480 MHz. (c) Linux/Pentium III 550 MHz.

TABLE 3
The Breakeven Points of Specializing the Function `crossover()`

Processor	Gain (in microseconds)	Runtime Specialization (in microseconds)	Breakeven Point
SPARC64	5.21	23.3	5
UltraSPARC	8.12	35.2	5
Pentium	3.75	3.88	2

4.3.2 The Breakeven Point

Table 3 shows the breakeven points of specializing the function crossover(). They were only five on the SPARCs and two on the Pentium. We conclude that runtime specialization may produce nontrivial performance gain for moderately frequently executed large size code.

4.3.3 Scalability

Fig. 24 shows the throughput of the parallel genetic algorithm in parallel processing. The x-axis is the number of worker threads. This number is equal to the number of CPUs that are obtained from the operating system (limited to eight or four in our experiments due to hardware limitations). The y-axis is throughput in generations per second.

For such a large application program, it is interesting to observe the same trend as in the microbenchmark in Section 3. The mutex version shows immediately decreased throughput. Both the nonspecialized and the specialized TSD versions show the same scalability as the number of threads and CPUs grows.

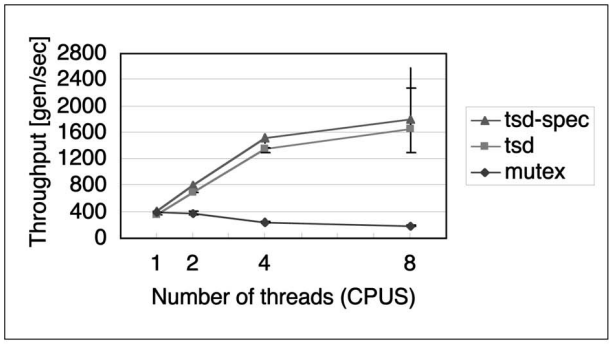
5 Specializing a Time Converter with TSD

We applied our method to a time converter library function, `localtime_r()`, which is longer than the random number generator and shorter than GALOPPS crossover. Time converter functions are invoked extensively and frequently, e.g., a network packet monitor such as tcpdump invokes this function every time it dumps a matched packet and an HTTP server calls this function every time it receives a request for logging.

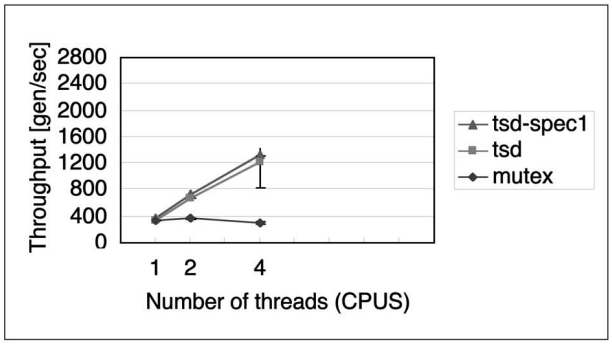
5.1 Implementing the Function `localtime_r()` with TSD

The function `localtime_r()` is a reentrant version of `localtime()`. Both functions convert an integer time value (`time_t`) into a structure `struct tm` to get the month, day, hours, minutes, seconds, etc. Although `localtime_r()` has a reentrant interface, its original implementation locks time zone data that are to be shared with the function `tzset()`. The function `tzset()` changes the time zone data according to the environment variable `TZ`.

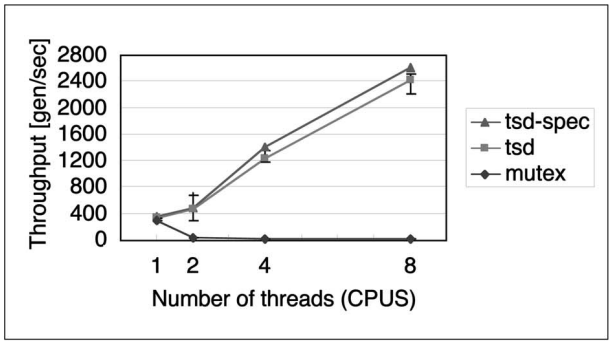
Since time zone changes happen very rarely, we wrote a new function `localtime_r()` that has a local copy of the time zone data for each thread. While the shared time zone



(a)



(b)



(c)

Fig. 24. Throughput of a Parallel Genetic Algorithm (average and 90 percent range as error bars). (a) SunOS/SPARC64 IV 450MHz. (b) SunOS/UltraSPARC II 480 MHz. (c) Linux/Pentium III 550 MHz.

```
extern  int      tz_shared_version ;

struct tm *
localtime_r( time_t *t, struct tm *tp ) {
  struct tz_tsd *tsd ;
  tsd = tz_gettsd();
        /* calls pthread_getspecific() */
  if( tsd == NULL ) {
    return( NULL );
  }
  if( tz_shared_version > tsd->version ) {
    tz_tsd_refresh( tsd );
  }
  return( tz_convert(tsd,t,1,tp) );
}
```

Fig. 25. An implementation of `localtime_r()` with TSD.

is unchanged, each thread can convert time values using its local copy without locking. If the shared time zone is changed by `tzset()`, each thread refreshes the local copy by locking the shared time zone variable.

Fig. 25 shows an overview of our new function `localtime_r()`. First, this function calls `tz_gettsd()` to dynamically link the TSD structure `tz_tsd`. Internally, the function `tz_gettsd()` invokes `pthread_getspecific()` and returns its value. In addition, `tz_gettsd()` handles the allocation of memory for the structure `tz_tsd`, the initialization of the structure, and address registration through `pthread_setspecific()` when it is called the first time in the thread. Second, the function `localtime_r()` compares the version numbers of the shared time zone data and the cached copy in TSD. If the cache is stale, it is refreshed by calling `tz_tsd_refresh()`. The version number of the shared variables is incremented when the function `tzset()` is invoked to change the time zone. Finally, the function `localtime_r()` calls `tz_convert()` which performs the time conversion using the TSD structure without locking.

5.2 Hints to the Partial Evaluator Tempo

Fig. 26 shows selected hints for the Tempo partial evaluator in ML to specialize the function `localtime_r()` described in Section 5.1. The other hints in ML and C are analogous to those in Fig. 14 and Fig. 15, respectively. The function `tz_gettsd()` is marked to be evaluated at specialization time by the runtime specializer.

5.3 The Specialized Program

Tempo takes the source code outlined in Section 5.1 and hints summarized in Section 5.2 to generate a template and a runtime specializer in C. Fig. 27 shows how the specialized program might look if generated in C. The runtime specializer calls `tz_gettsd()`, executes the initialization code, which includes `pthread_setspecific()`, and

```
entry_point := "localtime_r(D,D)";
external_functions := EVALUATE["tz_gettsd"];
```

Fig. 26. Hints in ML for specializing `localtime_r()`.

```
int H0, H2, H3 ;
struct tm *
tmp_localtime_r(time_t *t, struct tm *tp) {
  struct tz_tsd *tsd;
  tsd = (struct tz_tsd *)(&H0);
  if( tz_shared_version > tsd->version ) {
    tz_tsd_refresh((struct tz_tsd *)(&H2));
  }
  return tz_convert((struct tz_tsd *)(&H3),
                t, 1, tp);
}
```

Fig. 27. The specialized code of `localtime_r()`.

replaces the subsequent invocations of `tz_gettsd()` with the constant address of TSD. The constant nature of the TSD address results in the replacement of the `tz_gettsd()` invocation with the TSD address at the hole `&H0`. The other holes `&H2` and `&H3` are filled with the value returned by `tz_gettsd()`. In addition to eliminating one function invocation, an "`if`" branch is eliminated. The size of the specialized code was 96 bytes on the SPARCs and 54 bytes on the Pentium.

5.4 Experimental Results

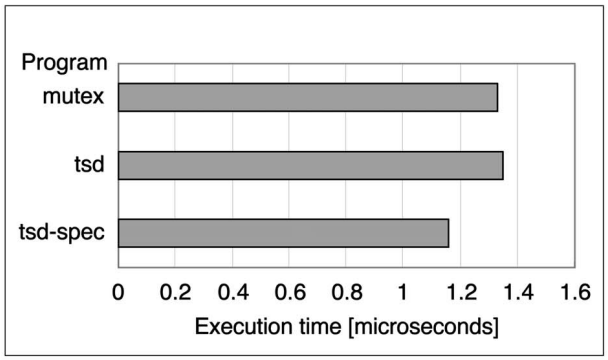
We measured the performance of the specialized time converter. The performance of the time converter depends on the time zone. In the experiments, the time zone was initialized to GMT, and its definition was read from a 56-byte file included in the Linux distribution.

5.4.1 Speedup

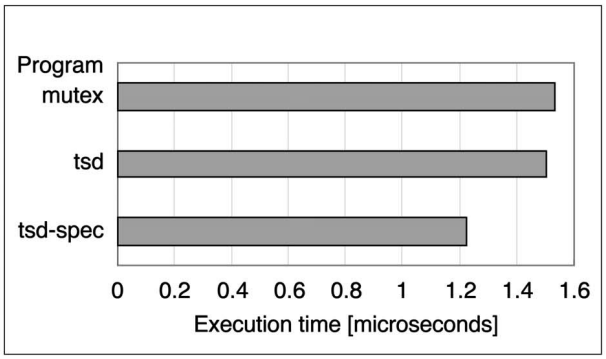
First, we measured the peak execution times to obtain the speedup due to specialization. We used the following versions for comparison:

- **mutex**. This version protects shared variables through `pthread_mutex_lock()` and `pthread_mutex_unlock()`, as implemented in GNU libc version 2.
- **tsd**. The version described in Section 5.1, executed without specialization.
- **tsd-spec**. This is the specialized version of [**tsd**] (described in Section 5.3).

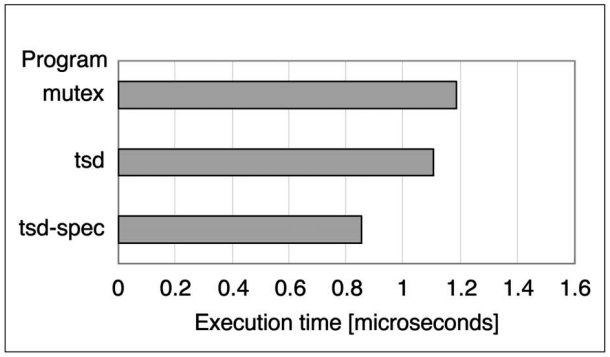
Fig. 28 summarizes the execution times of these versions including the outer loop that continually invokes one of the versions. In both SunOS and Linux, we extracted the source code of the time converter from GNU libc version 2, compiled it with the same options as others (see Section 2.5), and linked it statically. Therefore, the execution times do not include the overhead of dynamic linking within the time converter module. All the execution times include the overhead of invoking external dynamically linked functions outside the time converter modules, such as `strcmp()` and `strlen()`. We also tuned the data and instruction layout for the outer loop and each `localtime_r()` of these versions including nonspecialized versions. On the SPARC64 (Fig. 28a), despite moving the locking primitives out of the critical path, the new TSD program is slower than the mutex one. However, specialization improves performance by 14 percent, and the specialized TSD version



(a)



(b)



(c)

Fig. 28. Execution times of `localtime_r()`. (a) SunOS/SPARC64 IV 450MHz. (b) SunOS/UltraSPARC II 480 MHz. (c) Linux/Pentium III 550 MHz.

becomes faster than the mutex version. On the UltraSPARC (Fig. 28b) and the Pentium (Fig. 28c), the TSD version is already faster than the mutex version since locking appears to be more expensive. Specialization improves the TSD version performance by another 18 percent and 22 percent, respectively.

5.4.2 The Breakeven Point

Table 4 shows the breakeven points of specializing the function `localtime_r()`. Since they were small enough, we conclude that runtime specialization will produce significant performance savings in frequently executed medium-sized systems code.

5.4.3 Scalability

Fig. 29 shows the throughput of `localtime_r()` in parallel processing. The x-axis is the number of worker threads. This number is equal to the number of CPUs that are obtained from the operating system (limited to eight or four in our experiments due to hardware limitations). The y-axis is throughput in million executions per second of each version when invoked repeatedly. The results are consistent with the experiments in Section 3 and Section 4. The mutex version shows immediately declining throughput. In contrast, the TSD versions are scalable (approximately linearly) up to eight CPUs.

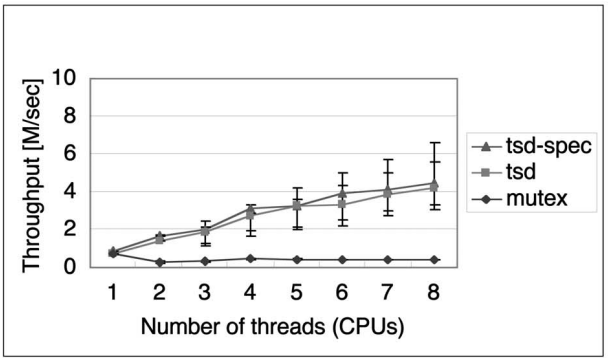
6 Related Work

Much of systems research has focused on efficiency gains. Other systems properties such as maintainability and portability have been considered "software engineering" problems. The main contribution of this paper is a method to achieve efficiency through specialization and portability through standards compliance. We demonstrated our method's usefulness in three experiments on the application of TSD for multithreaded programs. To the best of our knowledge, there is little previous research on achieving efficiency and portability simultaneously. Therefore, we briefly compare our research with previous work on multithreading, TSD, standards, and specialization.

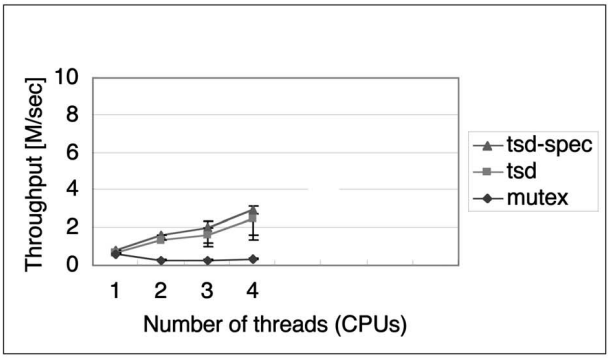
Research on multithreading primarily has focused on efficient (fixed) implementations, including the cooperation between user-level and kernel-level threads, and resource allocation including scheduling and register allocation [1], [8], [21], [31], [34]. In this paper, we have shown that TSD with specialization can reduce the

TABLE 4
Breakeven Points of Specializing the Function `localtime_r()`

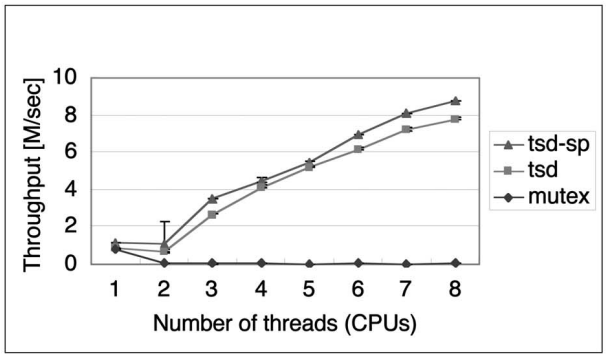
Processor	Gain (in microseconds)	Runtime Specialization (in microseconds)	Breakeven Point
SPARC64	0.190	4.45	24
UltraSPARC	0.278	3.90	15
Pentium	0.248	1.19	5



(a)



(b)



(c)

Fig. 29. The throughput of `localtime_r()` (average and 90 percent range as error bars). (a) SunOS/SPARC64 IV 450MHz. (b) SunOS/UltraSPARC II 480 MHz. (c) Linux/Pentium III 550 MHz.

synchronization overhead and provide much better scalability. Eraser is a dynamic verifier for multithreaded programs [29]. Eraser checks that all shared memory accesses follow a consistent locking discipline dynamically. In this paper, we have shown a systematic method to create a correct and efficient multithreaded program with a base single-threaded program, TSD, and specialization.

Compiler support in Visual C++ [4] provides Thread Local Storage (TLS) at a much lower overhead than the POSIX Threads package, which is equivalent to dynamic TLS. Static TLS is similar to our specialized case, although it still requires four load instructions instead of the single load instruction for global variable access. In contrast, our method only requires a general C compiler and a C program specializer, neither of which needs to recognize threads explicitly.

Kernel support for memory mapping can be used to implement efficient user-level TSD. For example, kernel-level TSD can be used to store the pointer to the structure for the currently running user-level thread. In an early system, kernel-level TSD is realized as a per-process memory region while a group of processes share most of the memory [2]. If we can extend the per-thread structure, user-level TSD can be implemented with a single indirection overhead. However, the POSIX Threads Standard does not expose the per-thread structure. In our method, no kernel support for memory mapping is needed, and there is no indirection overhead.

POSIX has chosen the internal locking approach for functions that maintain a state between invocations [11], [27]. For functions such as `localtime()`, POSIX has chosen to provide alternate reentrant versions such as `localtime_r()` that do not use TSD. The regular POSIX TSD based on a key-value scheme has been considered expensive for a long time and is used infrequently [18], [24]. Our TSD-based systematic development method largely eliminates the performance penalty of POSIX compliance.

The previous work on specialization targets file systems and network protocol stacks including RPC [3], [9], [22], [23]. To the best of our knowledge, this paper is the first one on systematic specialization of multithreaded programs, particularly by using the runtime specialization facilities of Tempo. Tempo can specialize Java programs with Harissa, a Java-to-C translator, and Assirah, a C-to-Java translator [30]. Currently, Tempo for Java supports only compile-time specialization. We used Tempo as a runtime specializer for the C language. Other runtime specializers for the C language, such as 'C [26] and DyC [10] could be used in place of Tempo, with appropriate specifications for those specializers.

7 Conclusion

In this paper, we addressed the problem of achieving efficiency *and* portability in systems software through a methodical approach. Typically, standard-compliant code carries apparently unavoidable overhead due to the standard-prescribed interface, which may add 40-50 instructions to simple system functions. Using specialization tools such as Tempo, we can eliminate the performance penalty during execution to achieve efficiency while maintaining the source code standard compliance to achieve portability. Our method preserves standard compliance by *giving* declarations of invariants with the source code instead of modifying it. We eliminate the execution overhead by using the partial evaluator Tempo's runtime specialization facility.

We applied our method to the creation of efficient and portable (POSIX-compliant) code using thread-specific data (TSD) for multithreading systems and application software. In this case, the main source of TSD overhead was the POSIX-prescribed interface to TSD variable binding such as

`pthread_getspecific()`. Using our method, Tempo generates templates and runtime specializers to convert the standard-prescribed function calls in the source code into simple global references at runtime, when the TSD memory location is invariant for each thread.

We demonstrated the effectiveness of our method in three representative experiments on widely used production software: a short system function (random number generator), a large application (parallel genetic algorithm program), and an intermediate library routine (time converter). For each case, we showed that runtime specialization gave significant performance improvements, a low breakeven point (a few executions of the specialized code to recover the runtime specialization overhead), and better scalability of TSD on SMPs compared with locking.

In most cases, specialized TSD code has performance comparable to that of direct access to global variables (i.e., code written for single threaded programs). Concretely, the execution time of the TSD-based random number generator is reduced by a factor of 4.8 on a SPARC running SunOS and by a factor of 2.2 on a Pentium running Linux. The execution time of the TSD-based parallel genetic algorithm is reduced by 13 percent on the SPARC and 5.1 percent on the Linux. The time converter's time is reduced by 14 percent on the SPARC and 22 percent on the Pentium. Our results suggest strongly that TSD with specialization is a good technique for the development of multithreaded systems software that is efficient, standard-compliant, and scalable.

ACKNOWLEDGMENTS

This work was done while Yasushi Shinjo was visiting Oregon Graduate Institute and Georgia Institute of Technology. Yasushi Shinjo was partially supported by a sabbatical grant from Ministry of Education, Culture, Sports, Science, and Technology (MEXT), Japan. This research was started when Calton Pu was with the Oregon Graduate Institute and was completed at the Georgia Institute of Technology. Calton Pu was partially supported by Defense Advanced Research Projects Agency (DARPA) grants from the Quorum, Ubiquitous Computing, and PCES programs, a US National Science Foundation grant from the CCR division, and a grant from Intel.

REFERENCES

[1] T. Anderson, B. Bershad, E. Lazowska, and H. Levy, "Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism," *Proc. 13th ACM Symp. Operating Systems Principles (SOSP-13),* pp. 95-109, Dec. 1991.

[2] "Balance 8000 Parallel Programming," Sequent Computer Systems, Inc., 1985.

[3] B. Bershad, S. Savage, P. Pardyak, E.G. Sirer, M.E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers, "Extensibility Safety and Performance in the SPIN Operating System," *Proc. 15th ACM Symp. Operating Systems Principles (SOSP-15),* pp. 267-283, Dec. 1995.

[4] A. Cohen and M. Woodring, *Win32 Multithreaded Programming.* O'Reilly, 1998.

[5] C. Consel, L. Hornof, F. Noël, J. Noyé, and E.N. Volanschi, "A Uniform Approach for Compile-Time and Runtime Specialization," *Proc. Int'l Seminar on Partial Evaluation,* pp. 54-72, Feb. 1996.

[6] C. Consel, J.L. Lawall, and A.-F. Le Meur, "A Tour of Tempo: A Program Specializer for the C Language," Research Report 1299-03, LaBRI, Apr. 2003.

[7] C. Cowan, A. Black, C. Krasic, C. Pu, J. Walpole, C. Consel, and E.N. Volanschi, "Specialization Classes: An Object Framework for Specialization," *Proc. Fifth IEEE Int'l Workshop Object-Orientation in Operating Systems,* pp. 72-78, Oct. 1996.

[8] R.P. Draves, B.N. Bershad, R.F. Rashid, and R.W. Dean, "Using Continuations to Implement Thread Management and Communication in Operating Systems," *Proc. 13th ACM Symp. Operating Systems Principles (SOSP-13),* pp. 122-136, 1991.

[9] D.R. Engler, M.F. Kaashoek, and J. O'Toole, "Exokernel: An Operating System Architecture for Application-Level Resource Management," *Proc. 15th ACM Symp. Operating Systems Principles (SOSP-15),* pp. 251-266, Dec. 1995.

[10] B. Grant, M. Mock, M. Philipose, C. Chambers, and S.J. Eggers, "The Benefits and Costs of DyC's Runtime Optimizations," *ACM Trans. Programming Languages and Systems (TOPLAS),* vol. 22, no. 5, pp. 932-972, Sept. 2000.

[11] M.B. Jones, "Bringing the C Libraries With Us into a Multi-Threaded Future," *Proc. Winter 1991 USENIX Conf.,* pp. 81-91, Jan. 1991.

[12] N.D. Jones, C.K. Gomard, and P. Sestoft, *Partial Evaluation and Automatic Program Generation.* Prentice-Hall, 1993.

[13] D.E. Knuth, *The Art of Computer Programming, Volume 2: Seminumerical Algorithms.* Addison-Wesley, 1969.

[14] E.D. Goodman, "An Introduction to GALOPPS—The Genetic Algorithm Optimized for Portability and Parallelism System," Technical Report #96-71-01, Michigan State Univ., 1996.

[15] A.-F. Le Meur, C. Consel, and B. Escrig, "An Environment for Building Customizable Software Components," *Proc. IFIP/ACM Conf. Component Deployment,* pp. 1-14, June 2002.

[16] A.-F. Le Meur, J.L. Lawall, and C. Consel, "Specialization Scenarios: A Pragmatic Approach to Declaring Program Specialization," *Higher-Order and Symbolic Computation,* vol. 17, no. 1, pp. 47-92, 2004.

[17] A.-F. Le Meur, J.L. Lawall, and C. Consel, "Towards Bridging the Gap Between Programming Languages and Partial Evaluation," *Proc. ACM SIGPLAN Workshop Partial Evaluation and Semantics-Based Program Manipulation,* pp. 9-18, Jan. 2002.

[18] B. Lewis and D.J. Bere, *Multithreaded Programming with Pthreads.* Sun Microsystems Press/Prentice Hall, 1998.

[19] R. Marlet, S. Thibault, and C. Consel, "Mapping Software Architectures to Efficient Implementations via Partial Evaluation," *Proc. IEEE Conf. Automated Software Eng. (ASE '97),* pp. 183-192, 1997.

[20] R. Marlet, S. Thibault, and C. Consel, "Efficient Implementations of Software Architectures via Partial Evaluation," *J. Automated Software Eng.,* vol. 5, no. 4, pp. 411-440, Oct. 1999.

[21] B. Marsh and M. Scott, "First-Class User-Level Threads," *Proc. 13th ACM Symp. Operating Systems Principles (SOSP-13),* pp. 110-121, 1991.

[22] D. McNamee, J. Walpole, C. Pu, C. Cowan, C. Krasic, A. Goel, P. Wagle, C. Consel, G. Muller, and R. Marlet, "Specialization Tools and Techniques for Systematic Optimization of System Software," *ACM Trans. Computer Systems,* vol. 19, no. 2, pp. 217-251, May 2001.

[23] G. Muller, R. Marlet, E.N. Volanschi, C. Consel, C. Pu, and A. Goel, "Fast, Optimized Sun RPC Using Automatic Program Specialization," *Proc. 19th IEEE Int'l Conf. Distributed Computing Systems (ICDCS '98),* pp. 249-258, May 1998.

[24] S.J. Norton and M.D. Dipasquale, *Thread Time: the Multithreaded Programming Guide.* Hewlett-Packard/Prentice Hall, 1997.

[25] J.K. Ousterhout, "Why Threads Are a Bad Idea, for Most Purposes," invited talk at the 1996 USENIX Technical Conf., Jan. 1996, http://home.pacbell.net/ouster/threads.pdf.

[26] M. Poletto, Wi.C. Hsieh, D.R. Engler, and M.F. Kaashoek, "'C and tcc: a Language and Compiler for Dynamic Code Generation," *ACM Trans. Programming Languages and Systems (TOPLAS),* vol. 21, no. 2, pp. 324-336, 1999.

[27] "Portable Operating System Interface (POSIX), Part 1: System Application Program Interface (API)," IEEE, 1996.

[28] C. Pu, T. Autrey, A. Black, C. Consel, C. Cowan, J. Inouye, L. Kethana, J. Walpole, and K. Zhang, "Optimistic Incremental Specialization: Streamlining a Commercial Operating System," *Proc. 15th ACM Symp. Operating Systems Principles (SOSP-15),* pp. 314-324, Dec. 1995.

[29] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, T. Anderson, "Eraser: A Dynamic Data Race Detector for Multithreaded Programs," *Proc. 16th ACM Symp. Operating Systems Principles (SOSP-16),* pp. 27-37, Dec. 1997.
[30] U.P. Schultz, J.L. Lawall, and C. Consel, "Automatic Program Specialization for Java," *ACM Trans. Programming Languages and Systems (TOPLAS),* vol. 25, no. 4, pp. 452-499, July 2003.
[31] Y. Shinjo and Y. Kiyoki, "A Lightweight Process Facility Supporting Meta-Level Programming," *Parallel Computing,* vol. 22, no. 11, pp. 1429-1454, 1997.
[32] "The SPARC64 Processor," HAL Computer Systems, Inc., 1998.
[33] *UltraSPARC-II User's Manual.* Sun Microsystems, Inc., 1997.
[34] A. Waldspurger and W.E. Weihl, "Register Relocation: Flexible Contexts for Multithreading," *Proc. 20th Ann. Int'l Symp. Computer Architecture,* pp. 120-130, 1993.

Yasushi Shinjo received the PhD degree from University of Tsukuba in 1993 and he is an associate professor in the Department of Computer Science at University of Tsukuba. He is interested in and has been published in the areas of operating systems, parallel and distributed computing, security, and virtual systems. He is a steward of the Special Interest Group on Operating Systems and Systems Software (SIGOS), Information Processing Society of Japan, (IPSJ). He is an editor of *IPSJ Journal* and an editor of *IPSJ Transactions on Advanced Computing Systems* (ACS). His research has been supported by the Ministry of Education, Culture, Sports, Science and Technology (MEXT), Japan, and the Japan Science and Technology Agency (JST). He is a member of the ACM, IEEE, IPSJ, and Japan Society for Software Science and Technology (JSSST).

Calton Pu received the PhD degree from University of Washington in 1986 and he is a professor and John P. Imlay, Jr. Chair of Software in the College of Computing at Georgia Institute of Technology. He is building software tools to support information flow-driven applications such as digital libraries and electronic commerce. He is interested in and has been published in the areas of operating systems, transaction processing, and Internet data management. He has published more than 50 journal papers and book chapters, 140 conference and refereed workshop papers, and served on more than 90 program committees. He served as PC cochair for SRDS, ICDE, and general chair for CIKM and ICDE. His research has been supported by the US National Science Foundation (NSF), Defense Advanced Research Projects Agency (DARPA), Office of Naval Research (ONR), and industry grants from AT&T, IBM, HP, and Intel. He is a member of the ACM, a senior member of the IEEE, and a fellow of AAAS.