StackThreads/MP User's Manual

Jan 1999
This manual corresponds to StackThreads/MP version 0.76

Kenjiro Taura

University of Tokyo
7-3-1 Hongo, Bunkyo-Ku, Tokyo, Japan

Copyright © 1998 Kenjiro Taura

1 Introduction

StackThreads/MP provides fine-grain multithreading in GNU C/C++, *without* a sophisticated preprocessor or a native code generator specifically implemented for multithreading. Instead, it works with unmodified GNU C compiler and preprocessor.

With StackThreads/MP, programmers can parallelize existing C code. Unlike traditional thread libraries such as Pthreads, StackThreads/MP supports *fine-grain* threads, threads whose granularities are much smaller than typical threads in traditional thread libraries. StackThreads/MP creates and finishes a thread with a very small overhead (about 10 RISC instructions). Thus it enables a programming style in which a thread is dynamically created when an independent task is identified. This allows a natural description of many parallel programs, such as parallel divide-and-conquer.

Other goals of StackThreads include (1) help parallel language developers implement parallel primitives, (2) provide common runtime framework in which multiple parallel languages can interoperate, and (3) propose minimal changes to the current calling standard so that any standard-conforming languages support a common multithreading model in future.

1.1 Reporting Bugs

Please report bugs and comments on the StackThreads/MP system and this document to the author.

`sthreads-author@yl.is.s.u-tokyo.ac.jp`.

Please join StackThreas/MP mailing list for getting helps and listening to announcements (visit `http://www.yl.is.s.u-tokyo.ac.jp/sthreads`).

1.2 Platforms

Currently, StackThreads/MP is running under the following platforms:

1. Solaris on SPARC (v8+ or v9) processors (currently, 32 bits only)
2. Solaris on x86 processors

3. Linux on x86 processors
4. Linux on Alpha processors
5. IRIX on MIPS processors (currently, 32 bits only)
6. Digital UNIX ver 4.0 on Alpha processors
7. Windows NT on x86 processors.

The following software must be installed on your platform.

1. GNU C compiler (version 2 required)
2. GNU awk
3. GNU make

GNU C compiler and GNU awk are mandatory. We do not recommend you to think about how to dispense with them. GNU make is somewhat optional but we use 'ifeq' notations to write our Makefiles. Be sure these commands are invoked by typing `gcc`, `gawk`, and `gmake` (or `make`), respectively. Also be sure that `nm` (either native or GNU) works correctly.

1.3 Installation

Installation consists of the following steps.

1. Uncompress and untar the distribution in the directory to which you want to install StackThreads. For example, if you want to install it under /usr/local/src, you put sthreads.tar.gz in /usr/local/src and type

   ```
   % gzip -dc sthreads.tar.gz | tar xf -
   ```

   You will find directory /usr/local/src/sthreads created. We hereafter refer to this directory as the toplevel directory.
2. In your .cshrc file (or in a file appropriate to your shell), set the following environment variables.
   a. `ST_DIR` to the toplevel directory to which you unzipped this software.
   b. `ST_OS_TYPE` to the operating system on which you use this software. Possible values are:
      1. `solaris` (for Solaris)
      2. `irix` (for IRIX)
      3. `osf1` (for Digital UNIX)
      4. `linux` (for Linux)

5. `winnt` (for Windows NT)

c. `ST_CPU_TYPE` to the CPU on which you use this software. Possible values are:

1. `sparc` (for SPARC)
2. `mips` (for MIPS)
3. `alpha` (for Alpha)
4. `i386` (for Intel)

d. `ST_THREAD_PKG` to the underlying thread package. This specifies the thread package that StackThreads/MP uses. Possible values are:

1. `st_solaris_thread` (for Solaris threads. possible only on Solaris. recommended on Solaris).
2. `st_pthread` (for Pthreads. possible except on Windows NT. The only choice on Linux and Digital UNIX).
3. `st_sfork_thread` (for sproc. possible only on IRIX. recommended on IRIX).
4. `st_nt_thread` (for NT thread. possible only on NT. The only choice on NT).
5. `st_no_thread` (for uniprocessors. available on every platform).

Also add `$(ST_DIR)/bin` to your path. These variables are needed not only when you build StackThreads/MP, but also when you compile applications that use StackThreads. Here is an example startup file for `csh`, to use StackThreads/MP on SPARC Solaris.

```
setenv ST_DIR /usr/local/src/sthreads
setenv ST_THREAD_PKG st_solaris_thread
setenv ST_OS_TYPE solaris
setenv ST_CPU_TYPE sparc
```

Here is an example startup for `sh`, to use StackThreads/MP on SPARC Solaris.

```
ST_DIR=/usr/local/src/sthreads
export ST_DIR
ST_THREAD_PKG=st_solaris_thread
export ST_THREAD_PKG
ST_OS_TYPE=solaris
export ST_OS_TYPE
ST_CPU_TYPE=sparc
export ST_CPU_TYPE
```

3. Compile the core library (`libst.a`) by:

```
% cd src
% make
```

In success, this generates library file `libst.a` as `$(ST_DIR)/lib/libst.a`

4. Compile some utilities (`libstext.a`) by:

```
% cd ../ext_src
% make
```

In success, this generates library file `libstext.a` as `$(ST_DIR)/lib/libstext.a`

5. Perform a simple test by:

```
% cd ../ex/small
% make
```

It builds many small test programs. When everything is compiled successfully, run any. Most programs do not take any command line arguments and those that do display appropriate messages when invoked with no arguments. Some programs are verbose and others are not, but all programs say `OK` in the last line when nothing went wrong.

On i386, program `nestcall` does not run successfully. This is a limitation of the current implementation with unmodified GCC. See Section 10.1 [Floating SP Problem], page 73 for the description of the problem. We have a patch to GCC that fixes this problem. See Chapter 8 [Applying Safety Patches to GCC], page 55 for how to apply patches.

On SPARC, `strpass` does not run successfully. This is another limitation of the current implementation with unmodified GCC. See Section 10.3 [Structure passing on SPARC], page 76 for the description of the problem. We have a patch to GCC that fixes this problem. See Chapter 8 [Applying Safety Patches to GCC], page 55 for how to apply patches.

Try program `bin` with various number of processors. The number of processors can be specified with `-nw` P, where P is the number of processors. Specify a value between 1 and the number of processors you have. Generally, performance is maximized when P is the number of processors on small systems (like four processor systems) and a value slightly smaller than it on medium- or large-scale systems (like 10 or more processor systems). Giving a value higher than the number of processors is allowed, but is generally a bad idea. For example, if you have 16 processors, try:

```
./bin -nw 1
./bin -nw 5
./bin -nw 10
./bin -nw 14
./bin -nw 15
./bin -nw 16
```

and see how speed is improved.

6. Compile documents by:

```
% cd ../doc
% make
```

This generates `stman.dvi` and `stman.info`. The `stman.info` should be copied into the directory where info files are installed in your system.

7. If you want to install StackThreads/MP for more than one platforms, *and* you want to share the source for them, you can copy directories `lib`, `bin`, and `include` to whatever directory you like. If you do that, run ranlib for archives and change your `ST_DIR` environment variable to the directory you copied them into. For example, if your StackThreads/MP source is in `/usr/local/src/sthreads` and you want to copy `lib`, `bin`, and `include` to `/usr/local/lib/sthreads`,

   ```
   % cd /usr/local/src/sthreads
   % cp -r lib bin include /usr/local/lib/sthreads
   % ranlib /usr/local/lib/sthreads/lib/libst.a
   % ranlib /usr/local/lib/sthreads/lib/libstext.a
   ```

   and change your `ST_DIR` settings to:

   ```
   setenv ST_DIR /usr/local/lib/sthreads
   ```

   in `csh`, or to:

   ```
   ST_DIR=/usr/local/lib/sthreads

   export ST_DIR
   ```

   in `sh`.

   When you have done that, you may delete the source tree, if you ever wish to do that.

   If you are not going to use StackThreads/MP on multiple platforms, or you do not mind having a separate source tree on each platform, do not bother to copy the above directories. You can just keep them in place.

8. At this point, you may also apply patches to gcc. This is not mandatory (except when you use gcc 2.8.1 on SPARC, which has a simple bug in handling `-mflat` option), so you can normally skip it and consider applying pathces later. These patches guarantee the safety of StackThreads/MP programs compiled by gcc. The distribution includes diffs for gcc 2.7.2.3 and gcc 2.8.1, in `gccpatch/gcc-2.7.2.3` and `gccpatch/gcc-2.8.1`, respectively. See Chapter 8 [Applying Safety Patches to GCC], page 55 for more detail.

   Patch gcc 2.8.1 on SPARC! gcc 2.8.1 on SPARC processors has a simple bug in handling `-mflat` option that StackThreads/MP heavily relies upon. When you are going to use StackThreads/MP on SPARC with gcc 2.8.1, apply patches.

2 The Basics

2.1 An Example

The following program illustrates the use of StackThreads/MP using the "Fibonacci" function.

```
 1
 2  #include <st.h>
 3
 4
 5  int fib(int n)
 6  {
 7    if (n < 2) {
 8
 9      return 1;
10    } else {
11      int a, b;
12
13
14
15
16      a = fib(n - 1);
17      b = fib(n - 2);
18
19      return a + b;
20
21
22    }
23  }
24
25  void pfib(int n, int * r, st_join_counter_t *c)
26  {
27    if (n < 2) {
28      *r = 1;                     /* write the result */
29      st_join_counter_finish(c, 1);/* say I have done */
30    } else {
31      int a, b;
32      st_join_counter_t cc[1];
33      ST_POLLING();               /* check if somebody is asking work
34                                     and free stack */
35      st_join_counter_init(cc, 2);/* initialize join counter */
36      ST_THREAD_CREATE(pfib(n - 1, &a, cc)); /* fork fib(n-1, ...) */
37      pfib(n - 2, &b, cc);        /* fork fib(n-2, ...) */
38      st_join_counter_wait(cc);   /* wait for child's completion */
```

```
39      *r = a + b;                 /* write the result */
40      st_join_counter_finish(c);  /* say I have done */
41      ST_POLLING();               /* check if somebody is asking work
42                                     and free stack */
43    }
44  }
45
46  int st_main()
47  {
48
49    int n = 33;
50    long pfib_time, sfib_time;
51
52    {
53      long t0, t1, t2, t3;
54      int pr, sr;
55      st_join_counter_t c[1];
56
57      /* do parallel fib */
58      t0 = st_current_time_ms();
59      st_join_counter_init(c, 1);
60      pfib(n, &pr, c);
61      st_join_counter_wait(c);
62      t1 = st_current_time_ms();
63      pfib_time = t1 - t0;
64
65      /* do sequential fib */
66      t0 = st_current_time_ms();
67      sr = fib(n);
68      t1 = st_current_time_ms();
69      sfib_time = t1 - t0;
70
71      if (sr != pr) {
72        fprintf(st_errout, "wrong\n");
73        st_stack_trace_and_die();
74      }
75    }
76
77    printf("pfib: %d ms on %d processors, sfib: %d ms\n",
78           pfib_time, st_n_toplevel_workers(), sfib_time);
79    return 0;
80  }
81
```

`fib` is a sequential function that computes Nth Fibonacci number. `pfib` is a parallel version. The algorithm is just to create a thread for the first recursive call (line 36). Before going into details of the program, we summarize some practical points.

1. Any file that uses StackThreads/MP primitives should include `<st.h>` (line 2).
2. A simple way to build a StackThreads/MP program is to define `st_main` instead of `main`. The `main` function defined in the library processes some command-line arguments that are common to all StackThreads/MP programs (such as `-nw`), performs necessary setup, and invokes `st_main`, passing unrecognized command-line arguments in `argc`, `argv`, and `envp`. The return value of `st_main` becomes the exit status of the program. There is a way to use StackThreads/MP from your own `main` (see Section 6.2 [Calling StackThreads/MP Procedure from Sequential Procedure], page 46).
3. If you want to maintain a single source which is sometimes compiled with StackThreads/MP and sometimes without it, use `STHREADS` predefined macro. It is defined iff the program is compiled for StackThreads/MP. (see Section 7.1 [STGCC/STG++: A wrapper for GCC], page 50 for more details).

2.2 Compiling

Compile the sample program in the previous section (call it `fib.c`) by:

```
% stgcc fib.c -o fib
```

We use a simple frontend driver for `gcc`, called `stgcc` (`stg++` is provided for C++ programs). It is NOT a preprocessor, but a simple script that invokes `gcc` with appropriate options (such as include search paths and library search paths). Also, it automatically links the StackThreads/MP library and utilities.

It is intended to accept ALL gcc options (unless it fundamentally conflicts with StackThreads/MP), so that any C program that can be compiled with gcc can at least be compiled with stgcc.

StackThreads/MP provides a small utility that generates a simple Makefile for new applications. Try:

```
stmkmf "app_name" > Makefile
```

For example, if you want to create an application called "fib", you type:

```
stmkmf fib > Makefile
```

The generated Makefile has some necessary settings such as `CC=stgcc` and some useful targets (such as `clean`). It assumes that you write a single source file (`fib.c` or `fib.cc`). For programs with multiple sources, look for `BASES` variable in the generated Makefile. Add *base names* of other object files there. For example, if the executable `fib` is going to be generated from `fib.o`, `myutil.o`, and `main.o`, you add `util main` to the `BASES` variables, like this:

```
BASES = $(APP) util main
```

If you already have a Makefile for your program, do not bother to generate it and merge it into the old one. The generated Makefile does nothing particularly important. Simply set your `CC` variable to `stgcc` in your original Makefile.

2.3 Running

If you successfully compiled the sample program, you should have obtained executable `fib` (or `fib.exe` on Windows NT). Run the program normally by typing the program name from the command line, like this:

```
harp:366% ./fib
```

You will see the following message on the terminal:

```
harp:366% ./fib
pfib: 10714 ms on 1 processors, sfib: 2114 ms
```

By default, a StackThreads/MP program runs using a single processor. You can use multiple processors by adding `-nw` option at the command line. For example, if you want to use 10 processors, try this:

```
harp:367% ./fib -nw 10
pfib: 1091 ms on 10 processors, sfib: 2062 ms
```

`st_main` does not see the command line arguments `-nw 10` in its `argv`; they are removed by the runtime system before entering `st_main`. (see Section 7.3 [Command Line Options Common for All StackThreads/MP Programs], page 53 for more options).

Assuming you create a large number of threads, performance is generally maximized when you give a number equal to the number of available processors (on small-scale systems like 4 processor systems) or a number slightly smaller than it (on medium- or large-scale systems like 16 or 64 processor systems). It generally does not improve performance to give a number larger than the number of processors. Also do not try to use many processors when the machine is heavily loaded.

You may notice that the parallel version runs much (5 times) slower than the sequential version. There are simple hacks to improve this up to a point like 3 times, but for very small functions like `fib`, the overhead like this is the current state of the art. In practice, the granularity of a thread is typically much larger. StackThreads/MP well tolerates a thread whose granularity is something like 200 instructions. (see Section 9.2 [A recommended programming style for performance], page 64 for several advices to achieve good performance on StackThreads/MP).

NOTE: The number of processors specified by `-nw` is irrelevant to the number of threads you can create in the program. By giving `-nw 10`, the program spawns 10 *OS-level threads* during its initialization. An OS-level thread is a thread directly supported by OS, such as LWP on Solaris. We hereafter call an OS-level thread a *worker*, to avoid confusion between threads supported by StackThreads/MP (created via `ST_THREAD_CREATE`) and OS-level threads. StackThreads/MP runtime system dispatches dynamically created threads to workers. Also note that although we say `-nw` specifies the number of *processors*, it actually specifies the number of workers. In particular, a worker is not permanently bounded to a particular processor. Confusing workers with processors does not normally cause any problem.

2.4 Basic Primitives

Let's now move our focus to the parallel Fibonacci function.

```
1  void pfib(int n, int * r, st_join_counter_t *c)
2  {
3    if (n < 2) {
4      *r = 1;                      /* write the result */
5      st_join_counter_finish(c);   /* say I have done */
6    } else {
7      int a, b;
8      st_join_counter_t cc[1];
9      ST_POLLING();                /* check if somebody is asking work
```

```
10                                    and free stack */
11      st_join_counter_init(cc, 2); /* initialize join counter */
12      ST_THREAD_CREATE(pfib(n - 1, &a, cc)); /* fork fib(n-1, ...) */
13      pfib(n - 2, &b, cc);         /* fork fib(n-2, ...) */
14      st_join_counter_wait(cc);    /* wait for child's completion */
15      *r = a + b;                  /* write the result */
16      st_join_counter_finish(c);   /* say I have done */
17      ST_POLLING();                /* check if somebody is asking work
18                                    and free stack */
19    }
20  }
```

The algorithm is a simple parallel recursion in which the first recursive call is done in parallel with the caller. This is done by

```
ST_THREAD_CREATE(fib(n - 1, &a, cc));
```

at line 12. Unlike sequential function, the parallel version takes two additional parameters. The second parameter, r, is the pointer to a storage to which the result value will be stored. The third parameter, c, is a pointer to a storage through which the function signals the end of its computation to the caller.

The third parameter is necessary because, once a procedure call is done in parallel with the caller, the caller at some point needs to make sure that the procedure is finished. This synchronization is done via what we call "synchronization counter." The type of the counter is `st_join_counter_t`. A storage for a counter is declared at line 8:

```
st_join_counter_t cc[1];
```

Line 11 initializes the counter to 2.

```
st_join_counter_init_(cc, 2);            /* initialize join counter */
```

This declares that two procedures will be associated to the counter cc. The two procedures are called at line 12 and 13.

```
ST_THREAD_CREATE(pfib(n - 1, &a, cc)); /* fork fib(n-1, ...) */
pfib(n - 2, &b, cc);       /* fork fib(n-2, ...) */
```

`cc` is given to both procedure calls.

When `pfib` is finished, it decrements the counter received through the third argument at line 5 and 16.

```
st_join_counter_finish(c);
```

When two procedures are finished, the value of the counter will become zero, because it is initialized to two and decremented twice. At line 14, the parent waits for the value of the counter to become zero.

```
st_join_counter_wait(cc);
```

Note that the procedure first writes the return value and then decrements the counter. Conversely, the parent first waits on the counter and then reads the return value. This order is important; this guarantees that when the parent reads the value, the called procedure has written a value. If, for example, the procedure first decrements the counter and then writes the result value, it is possible for the parent to observe that a counter becomes zero before the callee has written the result value.

If you feel that this program is very awkward just for writing small things like fib, you are right. This example is meant to describe primitives and mention general caution in parallel programming. You can usually make the program syntactically nicer by having a data structure that embeds the synchronization counter and wrapper functions that encapsulate synchronization. In the above pfib function, for example, you might define a data structure "int_box" as follows:

```
typedef struct int_box
{
   int val;
   st_join_counter_t c[1];
} * int_box_t;
```

and two wrapper functions void int_box_write(int_box_t b, int x) and int int_box_read(int_box_t b), as follows:

```
void int_box_write(int_box_t b, int x)
{
  b->val = x;
  st_join_counter_finish(b->c);
```

```
}

int int_box_read(int_box_t b, int x)
{
  st_join_counter_wait(b->c);
  return b->val;
}
```

It is also not difficult to add a new synchronization primitive for different constraints. See Section 9.4 [Implementing Synchronization Primitives], page 69 for the bottom-level machinery for implementing new synchronization primitives.

Up to this point, we have explained that a thread is created by `ST_THREAD_CREATE` and threads can be synchronized by a synchronization counter. Remaining things are calls to `ST_POLLING` at line 9 and 17. Well, it does not have any visible effect. One thing that happens in `ST_POLLING` is that the current processor checks if an idle processor is requesting a thread to the current processor, and gives a thread if any (and the current processor has enough threads). You do not have to understand the details about how threads are migrated or when a thread belongs to a particular processor, but you have to understand that threads never get executed in parallel **unless you occasionally call** `ST_POLLING`, which distributes work if there are idle processors. See Section 9.3 [Where should you insert ST'POLLING?], page 66 for hints as to where to insert `ST_POLLING`.

In summary, converting a sequential program into a parallel one involves the following steps:

1. Extract Parallelism: Insert `ST_CREATE_THREAD` where parallelism is necessary. Unlike traditional thread libraries, creating threads much more than the number of processors is no problem in StackThreads/MP.
2. Introduce Synchronization: Insert synchronization to make sure whatever condition you want to guarantee is satisfied at an appropriate point. For example, wait for another thread to write a result or wait for another thread to release some resources.
3. Distribute Work: Insert `ST_POLLING` to periodically distribute work among processors.

3 APIs

3.1 Creating A Thread

Syntax:

```
ST_THREAD_CREATE(e);
```

where *e* is a procedure call expression in C/C++ **that does not have nested procedure calls** in argument positions. On SPARC, special care must be taken when you pass structures as arguments (see Section 10.3 [Structure passing on SPARC], page 76). Unlike a similar thread creating functions in traditional thread libraries such as Pthreads, the procedure can take any number of parameters and, more importantly, the cost of a thread creation is very small (typically something like 10 instructions in addition to a procedure call overhead). Hence, if your computation can tolerate the overhead of a procedure call, it is likely to tolerate the overhead of creating a thread as well.

The actual constraint for argument positions is slightly more relaxed; we actually permit procedures that never call StackThreads/MP primitives, most notably procedures in standard libraries (such as printf, atoi, etc.). But you can remember the above rule as a simple safety net. Here are some examples:

```
ST_THREAD_CREATE(f(1, 2, 3, 4, 5));
```

is valid,

```
ST_THREAD_CREATE(f(atoi(argv[1])));
```

is valid, because we know atoi never call StackThreads primitives. Finally,

```
ST_THREAD_CREATE(f(g(x)));
```

is invalid if g calls any StackThreads primitive.

3.2 Polling

Syntax:

```
ST_POLLING();
```

Although it does not have any instantly visible effect, you must make sure that this macro is periodically called. It does two tasks. One is to check if a thread migration request from an idle processor is coming and respond to it if any. The other is to free some stack frames. See Section 9.3 [Where should you insert ST'POLLING?], page 66 for where to insert this in your program.

Do not worry so much about the overhead of `ST_POLLING`. If all the workers are already busy and you do not have to free any stack frame, the overhead is something like 10 – 20 instructions.

3.3 Synchronization

StackThreads/MP provides standard synchronization primitives including mutex, semaphore, and condition variables. Mutex and condition variables behave like their Pthread counter part, and semaphore (not supported by Pthread) behaves like its Solaris thread counter part. StackThreads/MP also supports a very commonly used producer-consume synchronization primitive, called join counter.

All these waiting primitives (mutex lock, semaphore wait, condition wait, and join counter wait) block the currently running thread (the thread created by `ST_THREAD_CREATE`) and schedule another thread, when the waiting condition is not immediately satisfied.

NOTE: In the following description, we say "xxx signals an error, when ... ". The meaning of this statement depends on how the library was compiled. These procedures are defined in file `src/st_sync.c` in StackThreads/MP distribution and the file contains a macro `#define ST_DIE_ON_ERROR 1` at the head of it. When this flag is set to 1 (the default), "signals an error" means the application will die when an error occurs (It has nothing to do with UNIX signals). This is useful when debugging (and also when your application is stable enough not to perform such an error). When this flag is zero, it simply means the procedure returns an error value (`ST_FAULT`).

3.3.1 Join Counter

Syntax:

```
int st_join_counter_init(j, v);
int st_join_counter_wait(j);
int st_join_counter_finish_1(j, v);
int st_join_counter_finish(j)
int st_join_counter_spawn_1(j, v);
int st_join_counter_spawn(j)
int st_join_counter_destroy(j);
       st_join_counter_t * j, unsigned long v;
st_join_counter_t x = ST_JOIN_COUNTER_INITIALIZER;
```

Join counter provides a simple producer-consume style synchronization. It internally maintains a non-negative integer (counter). `st_join_counter_init(`*j*`, `*v*`)` initializes the counter of *j* to *v*. It must be called when no other threads access to *j* concurrently (e.g., immediately after creation). `st_join_counter_finish_1(`*j*`, `*v*`)` and `st_join_counter_spawn_1(`*j*`, `*v*`)` atomically decrements and increments the counter of *j* by *v*, respectively. `st_join_counter_finish(`*j*`)` and `st_join_counter_spawn(`*j*`)` are synonyms for `st_join_counter_finish_1(`*j*`, 1)` and `st_join_counter_spawn_1(`*j*`, 1)`, respectively. `st_join_counter_wait(`*j*`)` waits for the counter of *j* to become zero. `st_join_counter_destroy(`*j*`)` destroys *j*. It must be called when no other threads access to *j* concurrently. It also checks if the counter is zero. If this is not the case, it signals an error. `st_join_counter_t` *x* `= ST_JOIN_COUNTER_INITIALIZER(c)` defines a global or static variable that is initialized to *c*.

It is allowed to recycle a storage for a counter many times, as long as the counter is correctly initialized before each use and destroyed after each use.

Typically, you use a synchronization counter to wait for the completion (or an arrival to a certain point) of child threads. Here is a simple example.

```
{
  st_join_counter_t j[1];

  st_join_counter_init(j, n);

  for (i = 0; i < n; i++) {

    ST_THREAD_CREATE(f(..., j));
```

```
    }

    st_join_counter_wait(j);
  }
```

Assume that `f(..., j)` calls `st_join_counter_finish(`*j*`)` when it is finished, `st_join_counter_wait(`*j*`)` waits for the completion of all threads created in the loop.

`st_join_counter_spawn` and `st_join_counter_spawn_1` are provided in cases where you do not know how many threads are going to be associated with the counter. Each time you create a thread, you increment the counter by `st_join_counter_spawn` and then create a thread.

All these functions return zero upon success and signals an error if an error occurred.

3.3.2 Semaphore

Syntax:

```
int st_sema_init_1(s, c, lim);
st_sema_init(s, c);
int st_sema_wait(s);
int st_sema_trywait(s);
int st_sema_post(s);
int st_sema_destroy(s);
       st_sema_t * s; int c, int lim;
st_sema_t x = ST_SEMA_INITIALIZER_1(c, p);
```

Semaphore manages limited resources shared by threads. A semaphore internally maintains a counter. `st_sema_init_1(`*s*`, `*c*`, `*lim*`)` initializes the counter of *s* to *c*. When *lim* is positive, it specifies the upper limit on the counter. `st_sema_init(`*s*`)` is synonym for `st_sema_init_1(`*s*`, 0)` (no upper limits). `st_sema_wait(`*s*`)` waits the counter of *s* to become positive and decrements the counter by one. `st_sema_trywait(`*s*`)` behaves like `st_sema_wait(`*s*`)`, except it immediately returns value `ST_BUSY`, when the counter is not positive. `st_sema_post(`*s*`)` increments the counter of *s* by one. If the upper limit was imposed by `st_sema_init(`*s*`)` and the counter would exceed that value after the increment, an error is signaled. `st_sema_destroy(`*s*`)` destroys *s*. It signals an error if the counter of *s* is not the same as the value given at initialization (i.e., `st_sema_post` were not called as many times as `st_sema_wait`.). `st_sema_t` *x* `= ST_SEMA_INITIALIZER_1(`*c*`, `*p*`)` defines a global or static variable *x* whose counter is *c* and the upper limit is *p*.

All these functions return zero upon success and signals an error if an error occurred. Read standard thread books on introduction to semaphore.

3.3.3 Mutex

Syntax:

```
int st_mutex_init(m);
int st_mutex_trylock(m);
int st_mutex_lock(m);
int st_mutex_unlock(m);
int st_mutex_destroy(m);
       st_mutex_t * m;
st_mutex_t x = ST_MUTEX_INITIALIZER;
```

Mutex is an essentially a binary semaphore with upper limit 1 (a semaphore that can only have 0 or 1 as its value). The mutex is either in locked or unlocked state. `st_mutex_init(`*m*`)` initializes it to an unlocked state. `st_mutex_lock(`*m*`)` waits for *m* to become unlocked and lock it. `st_mutex_trylock(`*m*`)` behaves like `st_mutex_lock(`*m*`)`, except that it immediately returns value `ST_BUSY` when *m* is locked. `st_mute_unlock(`*m*`)` unlocks *m*. It signals an error if *m* is not locked. `st_mutex_destroy(`*m*`)` destroys *m*. It signals an error if *m* is locked. `st_mutex_t` *x* `= ST_MUTEX_INITIALIZER` defined a global or static mutex.

3.3.4 Condition Variable

Syntax:

```
int st_cond_init(c);
int st_cond_wait(c, m);
int st_cond_signal(c);
int st_cond_broadcast(c);
int st_cond_destroy(c);
       st_cond_t * c; st_mutex_t * m;
```

Condition wait is a general mechanism upon which various synchronization can be built. Read standard thread books on how to use it.

3.4 Spin-Locks

StackThreads/MP provides a simple set of spin-lock primitives. A location can be atomically read and locked by *read-and-lock* primitives. They wait for the location to become unlocked and then atomically lock the location. They return the value previously written in the location. A locked location can be unlocked with *write-and-unlock* primitives. They take a value and a locked location as the arguments, write the value to the location, and unlock the location. The location must be locked or the behavior is undefined. These primitives are useful for performing atomic read-modify-write on a single location without a separate "lock" word. In addition a value can be read even when the location is currently locked.

CAUTION: All locking primitives described below (`st_read_and_lock_xxx`) fail after certain number of locking attempts. When they fail, the program terminates after printing stack trace information. This is typically a good thing, as it makes debugging easier (a simple error that forgets to unlock a location is detected at the next lock attempt). On the other hand, it is in theory possible to kill a program that is otherwise correct. However, in most cases, when you fail to obtain a lock, it indicates that you lock a location too long or you spin-lock a highly-contended location, both of which are considered wrong. In most cases, you should change the locking strategy for performance anyway. See Section 3.4.1 [Initialization, Lock, and Unlock], page 19 for more about this.

3.4.1 Initialization, Lock, and Unlock

Syntax:

```
typedef struct st_int_loc st_int_loc_t;
typedef struct st_long_loc st_long_loc_t;
typedef struct st_ptr_loc st_ptr_loc_t;

int st_read_and_lock_int(il);
long st_read_and_lock_long(ll);
void * st_read_and_lock_ptr(pl);

void st_write_and_unlock_int(il, i);
void st_write_and_unlock_long(ll, l);
void st_write_and_unlock_ptr(pl, p);

int ST_INT_LOC_INIT(il, i);
long ST_LONG_LOC_INIT(ll, l);
void * ST_PTR_LOC_INIT(pl, p);
```

```
    st_int_loc_t * il;   int i;
    st_long_loc_t * ll;  long l;
    st_ptr_loc_t * pl;   void * p;

st_int_loc_t v = ST_INT_LOC_INITIALIZER(x);
st_long_loc_t v = ST_LONG_LOC_INITIALIZER(x);
st_ptr_loc_t v = ST_PTR_LOC_INITIALIZER(x);
```

Type `st_int_loc_t` is `int` type that is 'tagged' by 'lock bits.' Similarly, `st_long_loc_t` and `st_ptr_loc_t` are tagged `long` and `void *`, respectively.

`st_read_and_lock_int(`*il*`)` waits for the location pointed to by *il* to become unlocked. When it is unlocked, it atomically lock the location and returns the value previously written at the location pointed to by *il*. Lock bits are stripped. `st_read_and_lock_long(`*ll*`)` and `st_read_and_lock_ptr(`*pl*`)` work similarly, but on locations that hold values of type `st_loc_long_t` and `st_loc_ptr_t`, respectively.

`st_write_and_unlock_int(`*il*`, `*i*`)` writes value *i* to the location pointed to by *il* and unlocks the location. The location must be locked upon calling the procedure. `st_write_and_unlock_long(`*ll*`, `*l*`)` and `st_write_and_unlock_ptr(`*pl*`, `*p*`)` are similar procedures for `st_long_loc_t *` and `st_ptr_loc_t *`, respectively.

`ST_INT_LOC_INT(`*il*`, `*i*`)` initializes the location pointed to by *il* by *i*. The location is initialized to 'unlocked' state. `ST_LONG_LOC_INT(`*ll*`, `*l*`)` and `ST_PTR_LOC_INT(`*pl*`, `*p*`)` are similar procedures for `st_long_loc_t *` and `st_ptr_loc_t *`, respectively.

`st_int_loc_t` *v* `= ST_INT_LOC_INITIALIZER(`*x*`);` defines a global or static variable of `st_int_loc_t` that is initialized to *x*. `ST_LONG_LOC_INITIALIZER(`*x*`)` and `ST_PTR_LOC_INITIALIZER(`*x*`)` are similar initializers for `st_long_loc_t` and `st_ptr_loc_t`, respectively.

`st_read_and_lock` and `st_write_and_unlock` can be used to perform an atomic read-modify-write sequence on a single location. Here is a typical example.

```
st_int_loc_t il = ST_INT_LOC_INITIALIZER(1);
 void read_modify_write()
{
    int x = st_read_and_lock_int(&il);
    st_write_and_unlock_int(&il, 3 * x);
}
```

This code atomically multiplies the value at the location *il* by 3.

Remark 1: Primitives `st_read_and_lock_xxx` never block the calling thread; they just wait for the location to become unlocked by spinning. Therefore, **do every effort to make the critical section as short as possible**. How short should they be? Well, as a rough approximation, avoid anything more than a couple of arithmetics and a couple of loads and stores. Avoid dynamic memory allocations such as malloc when you are holding a lock. Also, **never** try to lock a location when you are locking another location. It is a common source of deadlocks.

If you cannot avoid having a long critical section, you should not use spin-lock primitives in the first place, but use blocking synchronization primitives. As already shown, StackThreads/MP provides synchronization primitives that block the calling thread and schedule another thread when necessary. `st_mutex_t` will be most useful for such purposes.

Nonetheless, spin-locks are useful for short-critical sections. You cannot unconditionally adopt blocking primitives, as critical sections can be made short in many cases and in such cases spin-locks are both time- and space- efficient.

To avoid deadlocks and severe performance degradation due to a wrong use of spin-locks, `st_read_and_lock` primitives will timeout after a certain number of attempts. More specifically, when they fail to obtain a lock 1,000 times in a row, they sleep the calling worker 1 millisecond and retry. They repeat this process 10 times. When the lock is not still granted, it displays an error message and the application dies. It may kill an application that would otherwise be correct. However, it can detect a common programming error in which the programmer forgets to write a necessary unlock operation. Even if the application works, holding a lock 10 milliseconds typically degrades performance of the application, so you should take the error message as a strong suggestion that your locking strategy is wrong.

Remark 2: These primitives assume that two bits are embedded for tags within a single word. For example, on machines where `int` is 32 bits, `st_int_loc_t` uses the least significant two bits for tags. The value is left-shifted by two bits and stored in the most significant 30 bits. Consequently the most significant two bits are discarded when a value is stored. Conversely, when a value is read, the most significant two bits are sign-extended. Therefore, values representable by `st_int_loc_t` are restricted to -2^30 ... 2^30 - 1. For pointers, we assume the value of a pointer is 4 bytes aligned. That is, the least significant two bits of a pointer are zeroes and we use these two bits for tags. When a value that is not properly aligned is given to `st_write_and_unlock_ptr`, the least significant two bits are just discarded.

3.4.2 Reading and Checking Locations

Syntax:

```
int st_read_int(il);
long st_read_long(ll);
void * st_read_ptr(pl);

int ST_INT_LOC_CHECK(il, i);
int ST_LONG_LOC_CHECK(ll, l);
int ST_PTR_LOC_CHECK(pl, p);

int ST_INT_LOC_LOCKED(il);
int ST_LONG_LOC_LOCKED(ll);
int ST_PTR_LOC_LOCKED(pl);
```

`st_read_int(`*il*`)` reads the value at the location pointed to by *il*, but does not lock the location. It can immediately read the value even if the location is locked. `st_read_long(`*ll*`)` and `st_read_ptr(`*pl*`)` are similar procedures for locations of type `st_long_loc_t *` and `st_long_ptr_t *`.

`ST_INT_LOC_CHECK(`*il*`, `*i*`)` is 1 when *il* is currently unlocked and its value equals to *i*. This is semantically equivalent to `(st_read_int(`*il*`) == `*i*`)`, but on some CPUs, the former is considerably faster. `ST_LONG_LOC_CHECK(`*ll*`, `*l*`)` and `ST_PTR_LOC_CHECK(`*pl*`, `*p*`)` are similar primitives for `st_long_loc_t *` and `st_ptr_loc_t *`, respectively.

`ST_INT_LOC_LOCKED(`*il*`)` is 1 if the location *il* is currently locked. `ST_LONG_LOC_LOCKED(`*ll*`)` and `ST_PTR_LOC_LOCKED(`*pl*`)` are similar primitives for `st_long_loc_t *` and `st_ptr_loc_t *`, respectively.

3.4.3 Try Lock and Lock Any

Syntax:

```
int st_read_and_lock_any_int(il, els, n, ires);
int st_read_and_lock_any_long(ll, els, n, lres);
int st_read_and_lock_any_ptr(pl, els, n, pres);
 int st_try_read_and_lock_int(il, ires);
int st_try_read_and_lock_long(ll, lres);
int st_try_read_and_lock_ptr(pl, pres);
         int els; int n;
```

```
st_int_loc_t * il;  int * ires;
st_long_loc_t * ll; long * lres;
st_ptr_loc_t * pl;  void ** pres;
```

Suppose *il* points to address *a*, `st_read_and_lock_any_int(`*il*`,` *els*`,` *n*`,` *ires*`)` locks any of locations `{` *a*, *a* + *els*, *a* + 2 *els*, ... , *a* + (*n* - 1) *els* `}`. It returns the index of the locked location. That is, if address *a* + *k els*, it returns *k*. It also writes the value previously written at *a* + *k els* to **ires*. The locked location can then be freed by applying `st_write_and_unlock_int` on the address *a* + *k els*. `st_read_and_lock_any_long` and `st_read_and_lock_any_ptr` are similar functions for `st_long_loc_t *` and `st_ptr_loc_t *`, respectively.

These primitives are useful when multiple resources are available and you want to lock whatever resource is available.

`st_try_read_and_lock_int(`*il*`,` *ires*`)` tries to lock the location *il*. If it successfully obtains the lock, it writes the value previously written at *il* to **ires* and returns zero. It otherwise returns -1. `st_try_read_and_lock_long` and `st_try_read_and_lock_ptr` are similar functions for `st_long_loc_t *` and `st_ptr_loc_t *`, respectively.

3.4.4 Fetch and Add

Syntax:

```
int st_fetch_and_add_int(il, ix);
long st_fetch_and_add_long(ll, lx);
        st_int_loc_t * il; int ix;
        st_long_loc_t * ll; long lx;
```

`st_fetch_and_add_int` atomically increments the value written at *il* by *ix* and returns the value previously written at *il*. `st_fetch_and_add_long` is a similar function for `st_long_loc_t *`.

3.5 Yield

Syntax:

```
void st_yield();
```

temporarily deschedules the current thread. The thread will be scheduled later. For most parallel programs, there should be no reasons why you have to deschedule a thread that is runnable. We discourage using this procedure to wait for a particular event to occur. In such circumstances, you have to suspend the thread and let another thread resume it when a condition is satisfied.

3.6 Stacks and Contexts

There are several procedures that inform you of the use of stack of the running worker. All these procedures are normally of no interest for you. They are provided primarily for debugging and understanding stack usage and for precisely controlling when stack frames are reused.

3.6.1 Examine Current Stack Size

Syntax:

```
long st_stack_used_bytes()
```

returns the stack size of the calling worker in bytes. It returns the distance between the current stack top and the stack bottom. It specifically does not return the stack limit, but return the size of the stack currently occupied. The stack size increases when a procedure is called and may decrease when a procedure is returned or `ST_POLLING` or `ST_FREE_STACK` is called.

3.6.2 Show Stack Trace

Syntax:

```
void st_stack_trace()
void st_show_context(c)
        st_context_t c;
void st_show_exported_frames()
int st_n_live_exported_frames()
```

`st_stack_trace()` prints stack trace of the calling worker, from the current frame down to the stack bottom. It stops printing when it reaches the stack bottom of the calling worker, 50 frames

have been printed, or when it encounters any procedure that is not postprocessed. It stops printing after printing 50 frames to safely print corrupted stacks. If you want to print more frames, find definition of `MAX_STACK_TRACE_DEPTH` in `st.c`, change it as you like and recompile the library).

`st_show_context(`*c*`)`, where *c* is a pointer to a context filled by `st_suspend_thread`, displays frames represented by *c*.

`st_show_exported_frames()` displays suspended contexts that are allocated in the calling worker's stack. `st_n_exported_frames()` returns the number of such contexts.

3.6.3 Invoke Stack Management

Syntax:

```
void ST_FREE_STACK()
```

invokes a stack management. This examines frames that are no longer used and reset the stack pointer of the calling worker to an appropriate point (the point above which there are no frames still used). This is implicitly called when you perform `ST_POLLING()`, so this should be usually of no interest for you. Use this API only when you understand how StackThreads/MP works and when you want to free stack frames without causing thread migration.

3.6.4 Stack Pointers and Frame Pointers

Syntax:

```
void * asm_get_fp()
void * asm_get_sp()
void * asm_get_gp()
```

`asm_get_fp()` returns the frame pointer, `asm_get_sp()` the stack pointer. On machines that have a global register, `asm_get_gp()` returns the value of the global register. On machines that do not have a global register, `asm_get_gp()` is not available. Every procedure returns the value visible to the *calling procedure*. For example, when a procedure `f` calls `asm_get_fp()`, it returns the value of the frame pointer register *when* `f` *is running*.

When a procedure calls `asm_get_fp()` and `asm_get_sp()`, you may be surprised at their values; in StackThreads/MP execution, the frame pointer register always points to the running frame, whereas the stack pointer register always points to the stack top of the worker. These two values do not significantly differ in sequential execution, because both point to the frame allocated at the stack top. In StackThreads/MP, however, a running frame may not be on the stack top and may be far away from the stack top. It may even be another worker's stack.

Also, when a procedure calls `asm_get_sp()` several times, it may return different values each time. This is also because other frames may be above the running frame. The value of `asm_get_fp()`, on the other hand, returns the same value, as long as they are called from the same procedure. The returned value thus serves as the identifier of the procedure call.

3.7 Profiler

Syntax:

```
void st_config_profile_resolution(resolution);
       int resolution;
void st_config_profile_max_intervals(buffer_size);
       int buffer_size;
void st_config_profile_filename_prefix(filename);
       char * filename;
void st_begin_profile();
void st_end_profile();
```

See Chapter 5 [Performance Profiler], page 43 for more information.

3.8 Timing

Syntax:

```
long st_current_time_ms()
long st_current_time_us()
```

return the current (real) time in milliseconds and microseconds, respectively. The absolute value does not make sense; the only difference between two calls do. The resolution depends on the

operating system. Specifically, on Windows NT, the resolution is 10ms, so `st_current_time_us` is no more useful than `st_current_time_ms`.

3.9 Workers and Worker Groups

A worker refers to an underlying OS-thread such as a thread of Solaris Thread library or Pthreads library. Normally, StackThreads/MP program creates a fixed number of workers at program initialization and they share threads created by `st_main`, directly or indirectly. In this simple use of StackThreads/MP, you do not have to understand the following procedures and do not even have to know the notion of workers or worker groups.

StackThreads/MP allows other ways to access the provided multithreading facility. Specifically, you can dynamically create another group of workers and add workers to an existing group. Creating a group of workers dynamically is useful in several circumstances. First, when you want to have multiple groups of workers and to keep them isolated; threads are shared only among workers within a single group and threads are never migrated from one group to another. Second, creating a group of workers at runtime allows you to increase or decrease the number of OS-threads (the number of threads visible to OS) according to the application's need. In particular, an application can spend most of the time with a single thread and create a group of workers to perform a compute-intensive task in parallel. When the task is finished, all workers will die and the application goes back to its single-threaded state. Actually, such an application could be built without dynamically creating worker groups, if you do not care about the fact that many idle threads may consume CPU time when the application has no parallelism. Third, unlike other StackThreads/MP procedures, the procedure that creates a worker group (`stf_create_worker_group`) can be called from programs that is not compiled by `stgcc`. Thus, it can be used as a callback from library whose source is not available.

3.9.1 Create A Group

Syntax:

```
#include <st_foreign.h>
void* stf_create_sync_worker_group(wgc, n, f, a0, a1, a2, a3)
void  stf_create_async_worker_group(wg, wgc, n, f, a0, a1, a2, a3)
void* stf_wait_for_wg_to_exit(wg)
int   stf_did_wg_exit(wg)
        struct worker_group * wg;
```

```
struct worker_group_conf * wgc;
int n;
void * (*f)(void *, void *, void *, void*);
void * a0, * a1, * a2, * a3;
```

Both `stf_create_sync_worker_group` and `stf_create_async_worker_group` create a group of workers that initially consists of *n* workers. *f* specifies the entry function and *a0 ... a3* its arguments; the group as a whole performs *f(a0, a1, a2, a3)*, sharing threads created directly or indirectly from *f*. *wg* is a pointer to an uninitialized storage of type `struct worker_group`.

Created workers exit when *f* returns, no matter whether there are running threads that belong to the group. `stf_create_sync_worker_group` returns when workers exit. The return value is the return value of *f*.

On the other hand, `stf_create_async_worker_group` returns immediately. `stf_wait_for_wg_to_exit(`*wg*`)` waits for workers that belong to *wg* to exit (by blocking the underlying worker) and returns the return value of the entry function of *wg*. `stf_did_wg_exit(`*wg*`)` returns 1 if workers that belong to *wg* have already exited and 0 otherwise.

3.9.2 Add a Slave Worker

Syntax:

```
void st_add_slave_worker()
```

adds a worker to the worker group that the current thread belongs to. Currently there are no ways to remove a worker from a group.

3.9.3 Check Messages between Workers

Syntax:

```
void ST_STEAL_REQ_CHECK();
```

checks if an idle worker in the same group as the calling worker is requesting a thread migration from the calling worker. If it is, it gives an appropriate thread to the requesting worker. This is

implicitly called by `ST_POLLING()`, so this should be usually of no interest for you. Use this API only when you understand how StackThreads/MP works and when you want to migrate threads without freeing stack frames.

3.9.4 Deschedule Worker

Syntax:

```
void st_yield_os_thread()
void st_sleep_os_thread_ms(ms)
        int ms;
```

`st_yield_os_thread()` instructs the operating system to temporarily deschedule the calling *worker* (not thread). Recall that in general many threads are running on top of a worker. Descheduling a worker effectively deschedules all threads that are running on top of it.

The exact effect depends on the operating system and the underlying execution machinery of OS-level threads. Specifically, it is likely to have no effects when the number of workers are less than the number of available CPUs.

`st_sleep_os_thread_ms(`*ms*`)` instructs the operating system to deschedule the calling *worker* (not thread) for *ms* microseconds.

Use these APIs only when you understand how StackThreads/MP works and you are sure they are really necessary. Specifically, it is usually a bad idea to call these primitives to achieve a synchronization. You should normally suspend the current thread (not the entire worker) and explicitly resume it when it can make a further progress.

3.10 Callback

Syntax:

```
ST_CALLBACK_BEGIN();
ST_CALLBACK_END();
```

These two procedures together declare that the surrounding procedure may be called from a sequential procedure (a procedure that is compiled by plain `gcc`). When a procedure *f* is called from a sequential procedure, `ST_CALLBACK_BEGIN()` must be placed in the top of *f*'s variable declaration (`ST_CALLBACK_BEGIN()` is a macro that is expanded into a variable declaration). `ST_CALLBACK_END()` must be placed immediately before returning from *f*. More precisely, *f* must execute `ST_CALLBACK_BEGIN()` before any sub-procedure call in *f* and `ST_CALLBACK_END()` after any sub-procedure call in *f*. `ST_CALLBACK_END()` may syntactically appear twice or more (unless only one of them is executed at runtime). See Chapter 6 [Cooperating with Sequential Modules], page 45 for more about this topic.

3.11 Graceful Exits

Syntax:

```
void st_wg_exit(wv);
void st_wg_die(wv);
void st_app_exit(av);
void st_app_die(av);

        void * wv; int av;
```

`st_wg_exit(`*wv*`)` exits the worker group the calling thread belongs to and returns *wv* as the return status of the worker group. When the worker group is the toplevel group, the group that executes `st_main`, it exits the entire application. This is the preferred way of terminating an application.

`st_wg_die(`*lv*`)` behaves like `st_wg_exit(`*av*`)`, except that it prints stack trace of the calling worker before exiting. It is advisable to call this procedure whenever a fatal error occurs, so that the application prints some useful pieces of information for debugging.

`st_app_exit(`*av*`)` simply kills the entire application by `exit(`*av*`)` on UNIX. `st_app_die(`*av*`))` behaves like `st_app_exit(`*av*`)`, except it prints the stack trace of the calling worker before dying.

3.12 Thread Manipulation

3.12.1 Suspending A Thread

Syntax:

```
void st_suspend_thread(st_context_t c);
```

where `st_context_t` is defined as follows:

```
typedef struct st_context * st_context_t;
```

This procedure and the next (resume_context) are the bottom-level primitives on top of which you can build synchronization. We do not expect (and do not want) the programmers to write a program at this level. We still describe them because they are sometimes useful to implement a new synchronization primitive that cannot be easily synthesized from existing ones.

`st_suspend_thread` suspends (or blocks) a current thread and fills the structure pointed to by C with pieces of information, which we call the 'context' of the thread, so that another thread can later resume it. Upon calling this procedure, a field 'valid' must be cleared (filled by zero). Here is an example that illustrates the correct use of this procedure:

```
struct st_context c[1];
c->valid = 0;
st_suspend_thread(c);
   ... (xxx) ...
```

The control does not immediately return to `... (xxx) ...`. It will be executed only when somebody explicitly resumes it (by `st_resume_context` described next).

When `C->valid == 0`, it indicates that the structure is going to be filled by a context of a thread, but it has not been completely filled. `st_suspend_thread` sets 'valid' to 1 when it is ready to be scheduled. Before anything else, `st_suspend_thread` checks if `C->valid` is cleared and signals a runtime error if it is not.

In most cases, you want to store the pointer c to somewhere else before calling `st_suspend_thread`, so that another thread can find it. Typically, c is stored in some sort of "waiting queue," in which threads are waiting for a condition to be satisfied. To do this, you must do the following steps in the right order.

1. clear c,
2. store c somewhere, and
3. call `st_suspend_thread`.

The order between 1 and 2 is important; otherwise, another thread may find c and try to resume it, while c has not been completely filled by `st_suspend_thread`.

3.12.2 Resuming A Thread

Syntax:

```
void st_resume_context(st_context_t c);
```

`st_resume_context` is the bottom level primitive that resumes a thread that has been suspended by `st_suspend_thread`. It takes a pointer to a structure `struct st_context` that has been passed to `st_suspend_thread`, and resumes the execution of the thread that called `st_suspend_thread`.

3.12.3 Procedure Information

Syntax:

```
st_proc_info_t st_get_proc_info(pc, zero_if_not_found, pi)
        void * pc; int zero_if_not_found; st_proc_info_t pi;
int st_is_fork_point(pc)
        void * pc;
void st_add_procedure_information_table(table)
        st_proc_info_t * table;
```

Not documented yet.

3.13 Thread and Worker ID

Syntax:

```
extern long tls(worker_id);
long st_n_current_workers();
extern long tls(thread_id);
```

`tls(worker_id)` is an identifier of the calling worker within the worker group it belongs to. It holds a value from zero to *n*-1, where *n* is the number of workers withing the worker group the calling worker belongs to. When a worker group is created by `stf_create_async_worker_group` or `stf_create_sync_worker_group`, the entry function starts execution on the worker zero.

`tls(thread_id)` is a global identifier of the calling worker. It holds from zero to *n*-1, where *n* is the number of workers which have ever been created from the beginning of the program execution. The same identifier is (currently) not reused.

`st_n_current_workers()` returns the number of workers that currently belongs to the worker group that the calling worker belongs to. The worker group is initially created with the number of workers specified with the argument to `stf_create_async_worker_group` or `stf_create_sync_worker_group` (or the argument to `-nw` option in the case of the toplevel worker group) and may later be extended by `st_add_worker()`. The value returned by `st_n_current_workers()` at any moment may be smaller than the number of workers you have requested to create. For example, if you create a worker group that initially has 4 workers and performed `st_add_worker()` 3 times, `st_n_current_workers()` at a given moment may return any value from 1 to 7.

`tls(worker_id)` and `tls(thread_id)` can be used to guarantee that accesses to a single location do not occur concurrently. For example, the following code increments an element of a counter `counters_array` by one. The summation of all the elements in `counters_array` hold the total number of updates performed. Assuming `counters_array` is acceseed only from this worker group, there are no race conditions in `counters_array` because each access that may occur in parallel at runtime is given a unique value for `tls(worker_id)`.

Syntax:

```
x = compute_increment();
counters_array[tls(worker_id)] += x;
```

Note that the value of `tls(worker_id)` or `tls(thread_id)` may change even within a single function, as a result of thread suspension or migration. Therefore, in practice, it is always dangerous to assume that they are persistent across a procedure call (including `ST_POLLING()` and `ST_STEAL_`

`REQ_CHECK()`). For example, you cannot manually optimize away `tls(worker_id)` by caching it to a local variable; the following code is probably unsafe.

Syntax:

```
w = tls(worker_id);
x = compute_increment();
counters_array[w] += x;
```

Here, there are no guarantee that `w` at the two increments corresponds to the current worker id, because `compute_increment` may cause the rest of the computation to block or migrate.

The difference between `tls(worker_id)` and `tls(thread_id)` is important only when you create multiple worker groups within a program. However, it is recommended that you use `tls(worker_id)` where possible, because such programs are likely to be reusable.

Having said all, using `tls(worker_id)` or `tls(thread_id)` to gurantee freedom from races are not recommended at all. It is advisable to use a general synchronization method or a spin lock method, such as `semaphore` or `st_read_and_lock_any`. For example, the following code achieves the same effect without using `tls(worker_id)`, given that `counters_array` is an array of `st_int_loc_t`.

Syntax:

```
int v;
int x = compute_increment();
int idx = st_read_and_lock_any_int(counters_array,
                                   sizeof(st_int_loc_t), n, &v);
st_write_and_unlock_int(counters_array + idx, v + x);
```

It dynamically finds a location that is not locked. The size of `counters_array` is now arbitrary; it does not have to match the number of workers just to avoid out of range indices.

4 Memory Management

It may sound strange to talk about memory management in this manual. Yes, you can call any memory allocation function, including the standard malloc, from StackThreads/MP programs. However, once you start using StackThreads/MP in any serious parallel programs, you will soon notice that the system's default "malloc" function becomes a severe bottleneck. More specifically, typical malloc functions (including ones in Solaris libc) simply serialize all malloc calls using a mutex. If many threads contend, they cause many thread context switches of the underlying workers. In my experience, calling mallocs within a parallel section almost always makes a significant (negative) effect on performance. There are many xxx-malloc libraries, but most of them similarly serialize all malloc calls or are simply MT-unsafe (sure?).

We therefore provide some recommended alternatives to malloc. We also provide a mechanism with which you can easily replace the underlying memory allocator without changing application sources. With this mechanism, you can switch between several memory allocators (including the system's malloc). In this way, you will know how important are they for performance of parallel programs and what I am saying is not just a threat :-)

There are two (orthogonal) alternatives to the default malloc. One is to replace malloc with SGC library, and the other is a region-based allocator built on top of an underlying malloc. They are orthogonal. You can use either of them or combine both. We describe them in the following sections and then describe a mechanism with which you can easily switch between different memory-management alternatives.

4.1 Parallel Conservative Garbage Collector (SGC)

SGC is a conservative garbage collection (GC) library for shared-memory platforms. Its interface is the same as malloc, but calls to malloc are serialized only occasionally. In addition, it provides GC. A GC is also performed in parallel. If your particular program suffers from GC, you can control when to invoke a GC for better performance. If you are a creature who irrationally thinks GC is a slow thing, you can turn off GC and explicitly free objects (in this case you are effectively using SGC as a concurrent malloc/free library). SGC is provided as a separate library. You can obtain it from `http://www.yl.is.s.u-tokyo.ac.jp`. To use it from StackThreads/MP programs, you must define `WITH_STHREADS` in `config.mk` when you compile it, and you must define `WITH_SGC` when you compile StackThreads/MP library. See `README.parallel` in SGC for details. It is currently available on SPARC (Solaris), Pentium (Solaris), and R10000 (IRIX). Port to Pentium on Linux will come soon. SGC was derived from Bohem's conservative GC library (which is mainly for sequential platforms) and can be used with or without StackThreads/MP.

4.1.1 GC_MALLOC and GC_FREE

The basic API is almost identical to malloc/free:

Syntax:

```
#include <sgc.h>
void * GC_MALLOC(sz);
        unsigned int sz;
void GC_FREE(ptr)
        void * ptr;
```

Header file `sgc.h` comes with SGC, so you have to give an appropriate include search path to `stgcc/stg++` (with `-I$SGC_DIR`, where `$SGC_DIR` is the directory when SGC is in).

`GC_MALLOC(`*sz*`)` allocates *sz* bytes of memory. `GC_FREE(`*ptr*`)`, where *ptr* is a value previously returned by a call to `GC_MALLOC`, returns the allocated memory to the allocator. Functions and variables control behavior of SGC:

4.1.2 Controlling GC behavior

Syntax:

```
void GC_gcollect(void);
int GC_dont_gc = 0;
int GC_free_space_divisor = 4;
```

`GC_gcollect()` explicitly triggers a GC at a point you desire. When `GC_dont_gc` is zero (this is the default), `GC_MALLOC` automatically triggers a GC when necessary. In this mode, you neither have to call `GC_FREE` nor `GC_gcollect`. When `GC_dont_gc` is non-zero, `GC_MALLOC` does not trigger GC (it always expands heap when heap overflows). You reclaim objects either by calling `GC_gcollect` at a point you desire or by calling `GC_FREE` on individual objects.

`GC_free_space_divisor` controls heap expansion policy. It only makes sense when `GC_dont_gc` is zero. The larger that value is, the more frequently GCs occur and the less aggressively the system expands heap. When heap overflows, SGC either triggers a GC or expands heap. Roughly, when this value is D, it tries to keep one Dth of heap available for allocation. More specifically, suppose

a simple application that repeats allocating objects forever and keeps M bytes of them live all the time, the system tries to keep heap size (roughly) H to M + $M/(D\text{-}1)$, so that H/D bytes are available for allocation. The long-term behavior in this case is that every H/D bytes of allocations trigger a GC, which reclaims H/D bytes from somewhere in heap. The default value is 4, meaning that it tries to keep 25% of the heap free for allocation. If you feel the system spends too much time on GC, set it to 3 or 2. Other values are hardly useful (note that `D = 1` effectively prohibits GC).

There are other functions in SGC. See `sgc.h` and `README.parallel` in SGC.

4.2 Region-Based Memory Management

Memory management based on explicit regions is available on top of any malloc/free implementation. In this memory management model, an object is allocated on a specified region. You can dynamically create a region and then allocate an arbitrary number of objects in that region. Unlike malloc and free, you do not free individual objects; instead, you *reset* a region. Resetting a region reclaims *all* objects allocated in the region. The model is borrowed from *Aiken and Gay, Memory-Management with Explicit Regions, PLDI 98*, but is different from theirs in that (1) we do not provide any safety guarantee, (2) we do not require a dedicated compiler that understands region, and (3) we provide a reset operation, which make blocks associated with a region immediately reusable for subsequent allocations from the region.

4.2.1 Region Basics

Syntax:

```
#include <ralloc.h>
st_region_t st_make_region(void);
void * st_region_malloc(sz, rg);
void st_reset_region(rg);
void st_abandon_region(rg);
        st_region_t rg;
        unsigned sz;
```

`ralloc.h` is located at a place `stgcc` automatically searches. `st_make_region()` creates a new region. `st_region_malloc(`*sz*`,` *rg*`)` allocates *sz* bytes of memory from region *rg*. `st_reset_region(`*rg*`)` resets region *rg*. That is, it makes all blocks allocated in region *rg* reusable for subse-

quent requests with `st_region_malloc(..., rg)`. `st_abandon_region(rg)` is similar to `st_reset_region(rg)`, but it abandons all blocks associated with *rg*; it gives them back to the underlying memory allocator.

A region abstracts objects' death point. The basic observation is that objects allocated during a course of computation often die together. Such objects can be allocated from a single region and then freed all at once. This saves time for memory management and avoids fragmentation without a general and sophisticated mechanism.

A region internally keeps as many free lists as the number of workers. Each worker has its own private free list. Each free list is a list of fixed-sized chunks. A single chunk is, by default, 7KB. Associated with a free list are the current chunk pointer (C), which points to the chunk from which allocations are served, the current chunk limit pointer (L), which points to the end of the current chunk, and the allocation pointer (A), which points to a point in the current chunk from which the next allocation will be served. An allocation simply checks if A + the requested size exceeds L, and if it does not, it returns A and advances it by the requested size. Otherwise, the allocation goes on to the next chunk, changing C, L, and A appropriately. When a worker runs out of its private free list, it calls the underlying memory allocator to obtain a new chunk. No attempts are made to obtain one from another worker's free list.

`st_reset_region(rg)` simply resets its current chunk pointer to the first chunk of the free list and sets other pointers accordingly. It does not give chunks to the underlying allocator. It is `st_abandon_region(rg)` that does this.

4.2.2 Creating A Region

When a region is created by `st_make_region()`, free lists are empty. Therefore, allocations from a region gradually grow its free lists, occasionally calling the underlying allocator. As is mentioned in the beginning of this section, it may still suffer performance of parallel sections, if the underlying allocator serializes all allocation requests. One idea is to use SGC as the underlying allocator. If you want to (or must) deal with this problem with the default malloc, you may pre-allocate some chunks upon region creation. Here are the full APIs for creating regions. `st_make_region()` is repeated for completeness.

Syntax:

```
#include <ralloc.h>
st_region_t
```

```
st_make_region_aux(chunk_size, init_size,
                   pre_alloc_option, alloc, free);
st_region_t
st_make_region_highly_concurrent_1(init_size, alloc, free);
st_region_t
st_make_region_highly_concurrent(size);
st_region_t
st_make_region();
```

With `st_make_region_aux`, you can control all the parameters. *chunk_size* is the size of a single chunk. It is the unit of allocation via the underlying allocator. *alloc* and *free* specify the underlying allocator; *alloc* is a function that behaves similarly to the `malloc` in libc, whereas `free` is a function that behaves similarly to `free` in libc.

init_size specifies the initial size of the region in bytes. Each worker allocates $init_size/N$ bytes from the underlying allocator when it makes the first allocation from the region, where N is the number of workers in the worker group the calling worker belongs to. The net effect is that, the region soon becomes approximately *init_size* bytes in total as long as all workers will soon attempt to allocate at least one byte from this region. If you can approximate the total size of allocations that are going to be made from the region, you can give this value to avoid occasionally calling underlying allocators in parallel sections.

pre_alloc_option specifies whether this first allocation is made at the region's creation time. When *pre_alloc_option* == `st_region_pre_allocated_option_none`, no pre-allocation is performed upon creation. Each worker allocates the specified bytes upon first allocation. When `pre_alloc_option == st_region_pre_allocated_option_self`, the calling worker allocates the specified bytes only for the calling worker. When `pre_alloc_option == st_region_pre_allocated_option_all`, the calling worker allocates the specified bytes for all the workers.

A commonly used combination is provided by `st_make_region_highly_concurrent_1`. It intends to be useful for regions from which many allocations are going to be made in a parallel section. It allocates `init_size` bytes upon creation and evenly distributes them to all the workers. `st_make_region_highly_concurrent` is a simple wrapper to `st_make_region_highly_concurrent_1`. It gives the value of a macro `RG_ALLOC_FUNC` and `RG_FREE_FUNC` to `st_make_region_highly_concurrent_1` as the underlying alloc and free functions, respectively. Their default values are the system malloc and free, respectively, but you can override them before including `ralloc.h`. For example,

```
#define RG_ALLOC my_malloc
#define RG_FREE my_free
```

```
#include <ralloc.h>
```

replaces `malloc/free` with `my_malloc/my_free`.

4.3 How to Switch between Allocators

On top of two memory allocators described, StackThreads/MP provides a simple layer with which you can easily switch both the underlying allocator and whether the region-based allocator is used. The idea is that once you write a program in such a way that allocates and frees memory using the following primitives, you can experience several memory management mechanisms without changing your program.

4.3.1 Changing the Underlying Allocator

Syntax:

```
#include <st_mm.h>
void * MM_ALLOC(sz);
void MM_FREE(ptr);
void MM_ZERO_CLEAR(ptr, sz);
void MM_GC_POINT();
void MM_SETUP(fd);
```

They are macros whose behavior change according to the following parameter.

```
#define UNDERLYING_MM ...
```

It chooses the basic memory allocator and can be selected from the following values:

```
UNDERLYING_MM_SGC_AUTO_GC
UNDERLYING_MM_SGC_EXP_GC
UNDERLYING_MM_SGC_EXP_FREE
UNDERLYING_MM_MALLOC
```

`UNDERLYING_MM_SGC_AUTO_GC` uses SGC and relies on its automatic memory management. `UNDERLYING_MM_SGC_EXP_GC` uses SGC, but triggers GC only when explicitly requested by `MM_`

GC_POINT(). UNDERLYING_MM_SGC_EXP_FREE uses SGC, but never frees objects by GC. Finally, UNDERLYING_MM_MALLOC uses the system's default malloc.

MM_ALLOC calls the appropriate allocator. MM_FREE(*ptr*) frees object pointed to by *ptr* when UNDERLYING_MM_SGC_EXP_FREE or UNDERLYING_MM_MALLOC is given, and otherwise has no effects. MM_ZERO_CLEAR(*ptr*, *sz*) clears a block of *sz* bytes from *ptr*, only when MM_ALLOC does not zero-fill allocated memory. Whenever you need to zero-clear allocated memory, use MM_ZERO_CLEAR instead of bzero, to avoid duplication of work in case MM_ALLOC already does it. MM_GC_POINT() invokes a GC when UNDERLYING_MM_SGC_EXP_GC is given, and has no effects otherwise. MM_SETUP(*fd*) sets GC_dont_gc parameter appropriately when a UNDERLYING_MM_SGC_xxx is given. It also sets GC_free_space_divisor to *fd*.

4.3.2 Changing Region-based Allocator

Interfaces to region-based memory manager is also wrapped.

Syntax:

```
#include <st_mm.h>
st_region_t MM_MAKE_REGION();
st_region_t MM_MAKE_REGION_AUX(chunk_size, init_size,
                               pre_alloc_option, alloc, free);
st_region_t MM_MAKE_REGION_HIGHLY_CONCURRENT_1(init_size,
                               alloc, free)
void * MM_REGION_MALLOC(sz, rg);
void MM_REGION_FREE(ptr);
void MM_RESET_REGION(rg);
void MM_ABANDON_REGION(rg);
```

They are macros whose behavior change according to the following parameter.

```
#define RALLOC ...
```

RALLOC is either 0 or 1. When RALLOC is 1, all MM_REGION_xxx, except for MM_REGION_FREE(*ptr*), are equivalent to st_region_xxx. For example, MM_MAKE_REGION() is equivalent to st_make_region(). MM_REGION_FREE(*ptr*) has no effects. When RALLOC is 0, all MM_REGION_xxx

things are redirected to the underlying allocator. For example, `MM_REGION_MALLOC(`*sz*`, `*rg*`)` ignores the region parameter and is equivalent to `MM_ALLOC(`*sz*`)`. `MM_REGION_FREE(`*ptr*`)` behaves like `MM_FREE(`*ptr*`)`.

4.4 Alloca is not supported (yet)

`alloca` is sometimes used to optimize memory allocation. It behaves like `malloc`, but automatically frees the allocated storage when the procedure that called it returns. It is normally implemented by allocating a block of memory on top of the stack, effectively extending the currently running frame.

I have to tell you that `alloca` is currently not supported with StackThreads/MP, as the standard implementation of `alloca` fundamentally conflicts with the execution scheme of StackThreads/MP. Please use above alternatives (malloc, SGC, or region-based allocator). Region-based allocator will be particularly useful when a single procedure calls `alloca` many times, as blocks allocated in a single procedure die together.

`stgcc` currently redefines `alloca` as the name of an undefined function, so that programs that use `alloca` fail to link. If you want to optimize allocation on thin ice, you can call the builtin alloca function (`__builtin_alloca`), **as long as you know it is safe**. Calling `__builtin_alloca` from a procedure is safe, as long as the procedure does not call any procedure compiled by `stgcc`. In other words, if you place all calls to `__builtin_alloca` to the procedure entry, you are safe.

We are planning to support `alloca` in future, although it will not allocate memory from stack but from heap. Like standard `alloca`, it automatically frees memory allocated by a procedure after the procedure exits. However, it will not be as efficient as `__builtin_alloca`; its overhead is likely to be similar to that of the region-based allocator.

5 Performance Profiler

StackThreads/MP has a simple performance profiler with which you can view the status of each processor after running a program. It is not a real time profiler and the interface is not sophisticated, but it is still very useful to identify performance bottleneck. To use this, you run your program with `-tp` option, convert the log into xgraph format, and then look at it using xgraph. You need xgraph to use this feature.

5.1 Profiling your program

The most convenient way is to run your program with `-tp` option. For example,

```
harp:367% ./fib -nw 10 -tp
pfib: 1091 ms on 10 processors, sfib: 2062 ms
```

It produces a lot of files whose names are `00stprof.xx.yy` (where xx and yy is a number) in the current directory. Using 10 processors, you typically have 10 files.

You may want to profile a particular section of your program. In such cases, add a call to `st_begin_profile()` where you want to begin profiling and `st_end_profile()` where you want to finish profiling. Currently, you can call them only once in a program run.

The resolution of profiling is, by default, 100 microseconds. Each processor measures how much time it spends in each state (busy, idle, etc.) and, at every 100 microseconds, calculates the dominating state of that period. The log file records a state of each period. You can change the resolution of profiling by command line option `--time_profile_resolution` S, where S specifies the length of a period in microseconds. Specifying a large number saves space but the result may be inaccurate. Specifying a smaller number makes result more reliable at the expense of space.

Each processor keeps a fixed sized buffer for accumulating profiles and saves it into a file when the buffer overflows (and when the profile is finished). It may introduce a large Heisenberg effect into your profiling. You can increase the size of the in-memory buffer by `--time_profile_buffer_size` N where N specifies the number of entry in an in-memory buffer. The default is 8100. N does not necessarily represent the number of periods you can tolerate without saving the in-memory profile into a secondary storage, because a single entry describes a number of consecutive periods in a single state. For example, a processor is busy most of the time, the necessary storage will be quite small.

5.2 Viewing profiled results

To convert generated log files into xgraph format, use `mkxgrp`. It takes two parameters. One is the prefix of filename (`00stprof` if you did not give one explicitly when you ran the application) and the other is the number of processors and writes to standard out. For example,

```
harp:367% mkxgrp 00stprof 10 > data.xg
```

produces an xgraph file called data.xg from files named `00stprof.xx.yy` in the current directory. Files `00stprof.xx.yy` are removed when `data.xg` the output is successfully generated.

6 Cooperating with Sequential Modules

This chapter describes issues that may arise when StackThreads/MP modules cooperate with sequential modules. In this chapter, a procedure is said to be a *StackThreads/MP procedure* when it is compiled via `stgcc` (or `stg++`). A procedure is otherwise said to be a *sequential procedure*. Whether a procedure actually does something in parallel is irrelevant. There is a set of rules for calling sequential procedure from within a StackThreads/MP procedure and another set of rules for calling in the other direction.

Remember that `stgcc` can almost always substitute `gcc`, therefore you can make any C procedure a StackThreads/MP procedure by recompiling it by `stgcc`. So you can normally avoid complication by recompiling everything with `stgcc`. You may still want to have non-StackThreads/MP module either because the source of the sequential module is not available or because recompiling for StackThreads/MP is undesirable.

6.1 Calling Sequential Procedure from StackThreads/MP Procedure

There are almost no issues in this direction. A StackThreads/MP procedure can call any sequential procedure, as long as it does not call back any StackThreads/MP procedure. Issues that arise due to callbacks are discussed in the next section.

More precisely, StackThreads/MP procedures can safely call a sequential procedure provided (1) it does not perform a call back to StackThreads/MP, and (2) it conforms to the calling standard on the platform. Most sequential compilers generate procedures that conform to the calling standard, therefore, as an approximation, you can call virtually any sequential procedure, whether it is written in C, Pascal, Eiffel or whatever.

By "you can call a sequential procedure", we simply mean that you can correctly pass arguments and get the result of the procedure call. Specifically, it is completely the responsibility of the programmer that guarantees its proper functioning in the pre-sense of multiple threads of control. If the sequential procedure is already MT-safe (i.e., works correctly in the presence of multiple threads of control), you do not have to do anything special. Otherwise, you may have to make it MT-safe or guard the procedure call by some sort of mutual exclusion.

6.2 Calling StackThreads/MP Procedure from Sequential Procedure

First of all, you cannot call any StackThreads/MP procedure from a sequential procedure by default. A special setup procedure is mandatory. This is primarily because StackThreads/MP procedures reserve a register and assume that the register always holds the pointer to the worker-local storage, and a sequential procedure does not know the fact. Even if a StackThreads/MP procedure does not call any StackThreads/MP primitive that explicitly accesses a worker-local storage, it does so in its epilogue code. Therefore, as an approximation, remember that you cannot call *any* StackThreads/MP procedure unless you follow the necessary steps described below.

We provide two ways for different situations. One is suitable when you want to extract parallelism in the callback. The other is suitable when the amount of work done in the callback is very small.

6.2.1 Create a Worker Group

StackThreads/MP provides a convenient way to create a worker group that executes a given StackThreads/MP procedure (`stf_create_sync_worker_group` and `stf_create_async_worker_group`). They serve as gates from a sequential procedure to any StackThreads/MP procedure. They create a specified number of workers, setup pointers to the worker local storages, and then call the specified entry procedure. The entry procedure is a StackThreads/MP procedure and thus is allowed to access any StackThreads/MP primitives.

```
void* stf_create_sync_worker_group(wgc, n, f, a0, a1, a2, a3)
        worker_group_conf_t wgc; int n;
        void * (*f)(void *, void *, void *, void*);
        void * a0, * a1, * a2, * a3;
```

creates n workers and creates a group of n workers. The job of the worker group is given by an entry procedure f and arguments to it ($a0$ ... $a3$). Specifically, the worker group's job is to complete a procedure call `f(a0, a1, a2, a3)`. `stf_create_sync_worker_group` returns when `f(a0, a1, a2, a3)` returns. The return value of `f(a0, a1, a2, a3)` becomes the return value of the `stf_create_sync_worker_group`. If non-zero value is given to *wgc*, it is a pointer to a structure that describes configuration options of the worker. If *wgc* is zero, a default configuration is used. `stf_create_sync_worker_group` can be called both from a sequential procedure and from a StackThreads/MP procedure. If you call this procedure from a sequential module (a module that is compiled by plain `gcc`), include `<st_foreign.h>`, which defines the correct prototype for the procedure.

For example, suppose you are writing an X Window application and wish to add a callback function that uses StackThreads/MP primitives. Also assume that you don't want to recompile the Xlib or your administrator didn't allow you to replace nor duplicate it, thus the Xlib itself must remain sequential. In this case, you first write a StackThreads/MP procedure that accomplishes your job.

```
... my_st_proc(...)
{
  /* do whatever you can do with
     StackThreads/MP */
  ST_THREAD_CREATE(...);
    ...
  ST_THREAD_CREATE(...);
    ...
}
```

You may want to write a wrapper procedure of the above function, so that the entry procedure conforms to the interface of `stf_create_sync_worker_group`. The wrapper's job is simply to pass parameters to the above procedure, possibly casting them to meaningful types. It must be a StackThreads/MP procedure.

```
void * my_st_proc_wrapper(void * a0, void * a1, void * a2, void * a3)
{
    /* you translate arguments somehow */
    ...
    /* and call my_st_proc to do a real job */
    my_st_proc(...);
}
```

Finally, you write a callback procedure that invokes the entry procedure and registers it. This must be a sequential procedure (compiled by a plain C compiler), because X library calls that procedure.

```
void my_callback(...)
{
  int n = ...;
  void * a0 = ..., * a1 = ...,
       * a2 = ...; * a3 = ...;
  stf_create_sync_worker(0, n, my_st_proc_entry, a0, a1, a2, a3);
}
   ...
  XAddCallback(my_callback, ...);
  ...
```

`stf_create_async_worker` is an asynchronous version of `stf_create_sync_worker`. See Section 3.9 [Workers and Worker Groups], page 27 for details.

6.2.2 Setup TLS Pointers

The method just described is suitable when the following conditions are satisfied:

- You do not care the overhead of creating a number of workers each time either `stf_create_sync_worker_group` or `stf_create_async_worker_group` is called. This will be the case when the job you wish to accomplish is fairly large (taking a second or more by a single CPU).
- Created workers are 'independent', in the sense that they need not interact with already-running workers. This will be the case when the application is mostly sequential and you only occasionally create a compute-intensive job that is done in parallel.

There are situations where the method is not very appropriate. For example, you may want to write a tiny procedure for comparing two items and pass it to `qsort` function or write a tiny callback procedure that just displays a menu. Such procedures have nothing to do with parallelism, hence creating a worker group for each callback is impractical.

StackThreads/MP provides a way to declare that a StackThreads/MP procedure may be called from a sequential procedure. Specifically, A StackThreads/MP procedure can be safely called from a sequential procedure when the following conditions are met:

- it calls:
  ```
  ST_CALLBACK_BEGIN();
  ```
  at the head of the procedure. This is a macro expanded into a variable declaration, so make sure you put it in a place where a variable declaration is allowed.
- it calls:
  ```
  ST_CALLBACK_END();
  ```
  immediately before returning from the procedure (more precisely, before returning from the procedure and after any procedure call performed within the procedure).
- it does not call, either directly or indirectly, any parallel primitives (including `ST_CREATE_THREAD`, `ST_POLLING`, and any blocking synchronization primitives such as `st_mutex_lock` and `st_mutex_unlock`).

Procedures written according to these rules may be called both from sequential procedures and from StackThreads/MP procedures. Here is an example:

```
int less_than_p(void * a, void * b)
{
  ST_CALLBACK_BEGIN();
  int xa = evaluate(a);
  int xb = evaluate(b);
  ST_CALLBACK_END();
  return xa - xb;
}
```

Here, `evaluate` may be a sequential procedure, a regular StackThreads/MP procedure, or another StackThreads/MP procedure that again surrounds its body by `ST_CALLBACK_BEGIN()` and `ST_CALLBACK_END()`.

With this setup, you can pass the function `less_than_p` to the `qsort`:

```
qsort(a, n, sizeof(element), less_than_p);
```

In the above three rules, the last one is very restrictive. In essence, it says that even though such a callback procedure is compiled as a StackThreads/MP procedure, it must stay sequential, in the sense it does not use any threading operation. In our example, `evaluate` may not use StackThreads/MP primitives such as `ST_POLLING`, `ST_THREAD_CREATE`, and other synchronization primitives.

You might ask, "Given a callback cannot call any StackThreads/MP primitives after all, why do I want to write it as a StackThreads/MP procedure?" As a matter of fact, you can always write all such procedures in a separate file and compile them using plain `gcc`. What the current StackThreads/MP currently supports is no more powerful than this solution. We provide these declarations to allow callbacks and proper StackThreads/MP procedures to be written in a single file.

7 Command Summary

7.1 STGCC/STG++: A wrapper for GCC

`stgcc` or `stg++` are shell scripts that call `gcc` or `g++`, respectively. Below we say `gcc` to refer to both `gcc` and `g++`, and use `stgcc` to refer to both `stgcc` and `stg++`.

What it basically does is to intercept the compilation process that outputs object file just after gcc emits assembly, extract some information about stack frame formats of each procedure from, and put some tricky pieces of code at each procedure epilogue sequence.

`stgcc` is intended to be usable wherever `gcc` is used, unless a command line option fundamentally conflicts with what StackThreads/MP assumes. Most command line options are passed as is to `gcc`, but it must recognize some of the parameters that controls compilation. Certainly you do not have to know what it is exactly doing and you should not even have to learn anything, as long as you already know `gcc`.

What you have to know, however, is things that might conflict with your tricky way of using `gcc`. At present, `stgcc` does not produce any error even when a conflicting option is given. So, it is possible that even when a compilation succeeds, the resulting program is erroneous. We are trying to improve this in future. To be sure, give `--show_commands` option to `stgcc`, which displays invoked subcommands.

1. A register is reserved for a OS-thread local storage and other must not be reserved. StackThreads/MP assumes that one register is reserved to access a thread-local storage (by `-ffixed-REG` in `gcc`). The reserved register is:
   - `%ebx` on i386,
   - `%l7` on SPARC,
   - `$23` on Mips, and
   - `$14` on Alpha.

   Options are automatically specified by `stgcc` so you do not normally notice this. But if you want to reserve this register or another for your purpose, this is likely to be a problem. Contact me if you want to do this anyway.
2. `stgcc` turns off inlining by giving `-fno-inline-functions` and `-fno-default-inline` to `gcc`. We assume that a procedure call written in `ST_CREATE_THREAD(`*e*`)` is translated into a procedure call in assembly level. In other words, *e* must not be inlined. This may severely

hurt performance of some programs, but for now there are no ways to prohibit inlining of a particular call site or a particular function.

You can instruct the compiler to explicitly inline some functions by giving `inline` directive before a procedure definition.

3. `stgcc` gives `-fno-omit-frame-pointer` to `gcc`, so that each procedure keeps a separate frame pointer. Therefore giving `-fomit-frame-pointer` to `stgcc` conflicts with this. This has no performance effects on i386 and SPARC, but may have some on Mips and Alpha.
4. On SPARC, `stgcc` turns off the use of register windows by giving `-mflat` to `gcc`. Hence giving `-mno-flat` to `stgcc` conflicts with this.

To summarize, the following options are prohibited in StackThreads/MP.

1. `-finline-functions`, `-fdefault-inline`, `-fomit-frame-pointer`, and `-ffixed-REG` are prohibited on all CPUs
2. `-mno-flat` is prohibited on SPARC

Here is the list of options supported by `stgcc`, in addition to all options supported by `gcc`.

- `--show_commands` shows all subcommands invoked by `stgcc`. This should be of no interest for you, but when you get strange compile-time errors, you might want to look at what `stgcc` is doing.
- `--leave_tmps` does not remove temporary files created during compilation. This should be of no interest for you, but when you get strange compile-time errors, you might want to look at temporary files.
- `--dbg` is equivalent to giving both `--show_commands --leave_tmps`.
- `--no_postprocess` does not postprocess the assembly file generated by `gcc`. Modules compiled with this option cannot use StackThreads/MP primitives. This option is primarily useful when you (1) when you want squeeze performance of some modules that never use StackThreads/MP primitives, or (2) investigate performance impact of postprocessing. Normally, compiling everything without this option will suffice.
- `--compiler=`*cc* specified the underlying C compiler invoked from `stgcc`. We currently assume the underlying C compiler is `stgcc` in many ways, so the presence of this option does not imply you can use any C compiler. Do not give anything except for `gcc` or `g++`.

Normally, `stgcc` defines `STHREADS` to 1. When `--no_postprocess` is given, `stgcc` does **not** define `STHREADS` but instead defines `STHREADS_NOPP` to 1.

7.2 STLINK: A Wrapper for Linker

`stlink` is a command that is invoked from `stgcc`. It is again a shell script that eventually calls a linker (it actually calls `gcc` or `g++`). When an assembly file is postprocessed, `stgcc` generates pieces of information about procedures in the file and attaches them in the assembly file. The assembly file is then compiled into an object file. At link time, `stlink` collects all these tables and generate a single table that point to tables of all files. The table is generated in a form of a C source file, compiled by `gcc` (by default), and then linked together with all other files. It also collects thread startup hooks and worker startup hooks and generate a table of them.

Since this command is implicitly invoked by `stgcc`, it should be usually of no interest for the programmer. We nevertheless describe it for its potential use in debugging strange link-time errors.

Accepted command line options are again the same as those of `gcc` or `g++` (specified by `--linker=` option described below). Most options are passed as is to `gcc` or `g++`. Here is the list of options specific to `stlink`.

1. `--dbg` shows the command lines of the invoked subcommands. If you get a strange link time error, run the same command with this option. You normally implicitly use `stlink` from `stgcc`, you may not know how `stlink` is invoked. In this case, first run `stgcc` with `--dbg` option and see its output. You will find a line that invokes `stlink`. Copy the line and add `--dbg` option.
2. `--linker=`*cc* specifies the underlying linker (which actually must be either `gcc` or `g++`) invoked by `stlink`. The default is `gcc`.
3. `--compiler=`*cc* specifies the compiler used to compile the generated table file. The default is `gcc` and I don't think of any reason why you have to change this (even if you are linking C++ programs).

We describe a little more details about its semantics and machinery. It collects all procedure tables and thread/worker startup hooks from files specified via explicit filenames (such as xxx.o or xxx.a) or libraries found under explicit search pathos (via -Lxxx). For example,

```
stlink a.o b.a c.o -L. -L../lib -lmylib
```

will collect tables and hooks from `a.o`, `b.a`, and `c.o` which are assumed to be found in the current directory and may collect them also from `libmylib.a`, if it is found in the current directory (`.`) or `../lib` directory. Tables and hooks in `libmylib.a` will not be collected when it is found in other directories that do not explicitly appear in the command line. Specifically, standard libraries such

as `libc` or `pthread` are not examined. In short, `stlink` does not know standard library search paths.

`stlink` uses `nm` command to collect tables and hooks from object files. For each file to be examined, it invokes `nm` and scans the output to find the table and hooks. Simple naming rules are assumed to find them. A surprising thing might occur when you accidently use names that are recognized as hooks or tables. To list them,

- Any static symbol in data segment (initialized static variable) whose name ends with `_proc_info` is assumed to be the name of the procedure table of the file.
- Any global symbol in text segment (external procedure) whose name begins with `sthreads_thread_start_hook_` is assumed to be a name of a thread startup hook.
- Any global symbol in text segment (external procedure) whose name begins with `sthreads_worker_start_hook_` is assumed to be a name of a worker startup hook.

7.3 Command Line Options Common for All StackThreads/MP Programs

Any StackThreads/MP program accepts the following command line arguments. A shorter synonym is provided for commonly used ones. These command line arguments are effective only when you do not write your own main function but write `st_main`.

- `--n_workers` P, or `-nw` P : specifies the number of processors by P.
- `--stack_size` S, or `-ss` S : specifies the stack size of an OS-thread by S. S is either an integer, or an integer followed by a `k` or `m`, which says the number should be interpreted as kilobytes and megabytes, respectively. For example, `-ss 4m` sets a stack size to 4 megabytes.
- `--print_toplevel_worker_stat,` or `-ps` : prints a simple statistics after a run. It prints statistics like how many threads were created and how many blocking occurred.
- `--time_profile`, or `-tp` : profiles processor status and generates log files. The file names begin with 00stprof by default. This can be changed by –time_profile_filename below.
- `--time_profile_filename` N : specifies the prefix of log files by N. For example, `--time_profile_filename my_important_logs` generate files of names `my_important_logs.`$xx.yy$ where xx and yy are numbers.
- `--time_profile_resolution` S : specifies the resolution of the profile. S is the length of a period in microseconds. Each period is represented by a state in which a processor spends its longest time during that period.

- `--time_profile_buffer_size` N : specifies the size of in-memory buffer size of each processor. N is the number of entries in the log file. Longer the N is, each processor can tolerate a long profiling run without saving the log into a secondary storage.

These command line arguments are removed from the command line before entering `st_main` and only remaining arguments are passed to `st_main`. For example, when you write:

```
./fib -nw 10 30 -ps
```

then the `st_main` of this program will receive only

```
{ "./fib" "30" }
```

as its second parameter (`argv`) and receive 2 as its first parameter `argc`.

8 Applying Safety Patches to GCC

Although StackThreads/MP can be practically used with unmodified GCC, there are certain cases where execution scheme of StackThreads/MP breaks the assumption that code generator of GCC exploits for optimization (See Chapter 10 [Known Limitations and Subtleties], page 73 for descrpitions about such cases). We provide simple extensions to gcc that fix the problem in a form of patches to gcc (2.7.2.3 and 2.8.1). Note that this is not a bug of gcc, but just a subtle mismatch between StackThreads/MP execution scheme and the sequential exeuction scheme that is assumed by the compiler. Patches are quite small (essential modifications are just a few lines) and the patched gcc supports a couple of additional command-line options. Without these command-line options, the patched gcc functions exactly the same as the original gcc. Even when these command-line options are given, generated code is different only in cases where the original generated code is unsafe. Of course, generated code is always compatible (inter-operable) with code compiled normally (code compiled with such an option can call functions compiled normally, and vice versa). Therefore, the patched gcc can safely replace the existing gcc, if you do not like having two copies of gcc. You need not recompile any sequential library compiled by the original gcc (such as X library and Tk library).

Modifications to source code are always guarded by:

```
#if !defined(STHREADS_SAFE) || STHREADS_SAFE
modified source
#else /* !defined(STHREADS_SAFE) || STHREADS_SAFE */
original source
#endif /* !defined(STHREADS_SAFE) || STHREADS_SAFE */
```

so that you can revert to the original code by defining flag `STHREADS_SAFE` to zero. Note that if `STHREADS_SAFE` is undefined (this is default), the modified source is used.

The patch is not a requirement for StackThreads/MP installation. In practice, you can start using the library with unmodified gcc and later apply the patch, when you like to use the library for serious purposes. If you do not want to patch gcc, you can workaround them by hand. Problematic cases are described below and you can avoid them (in particular, if StackThreads/MP is used as a compilation target, the frontend compiler are likely to be able to deal with them). If you want to worry about them and guarrantee safety, however, we recommend you to apply the patch to gcc someday. These patches are important only on SPARC and i386.

8.1 How to Apply Patches

Patches are placed in directory `gccpatch` in a form of diffs to sources of `gcc 2.7.2.3` and `gcc 2.8.1`. The name of the files are `gcc-patch.2.7.2.3` and `gcc-patch.2.8.1`, respectively. If you are going to use `gcc 2.7.2.3` or `gcc 2.8.1`, just copy the appropriate patch file to the source directory (the directory where files like `calls.c` and `flags.h` exist) and do:

```
patch -p 0 < patch-file
```

where *patch-file* is either `gcc-patch.2.7.2.3` or `gcc-patch.2.8.1`.

If the version number of your gcc source is neither 2.7.2.3 nor 2.8.1, you may still have a chance to apply the patches mechanically, as long as the major version number matches. Just try the same command and look at messages from patch command. Even if you fail, since patches are very small, it is practical to apply them by hand.

8.2 Using Patched GCC

Once you successfully patch gcc sources and rebuild gcc, you setup things so that `stgcc` exploits it.

- Set environment variable `ST_GCC_IS_PATCHED`. The value does not matter. When `stgcc` finds that this variable is defined, it gives necessary options when it calls `gcc` and `g++` (we hereafter only mention gcc, but the following statements also apply to g++).
- If the patched gcc replaces the original gcc or if you arrange your `path` variable so that you run the patched gcc by just typing `gcc` (`g++`), you are done.

  Recall that the patched gcc produces the exactly same output as the original does, unless explicit options are given, so replacing the original gcc is almost always practical. You may still want to call patched `gcc` only from `stgcc` and use the original gcc for other cases. If you are such a skeptical person, put the patched gcc to anywhere and tell `stgcc` the location of them by setting environment variable `ST_COMPILER_PATH`. For example, you may write the following line in your `.cshrc`.

  ```
  setenv ST_COMPILER_PATH /home/nomo/my_gcc/bin
  ```

  Then, `stgcc` calls `/home/nomo/my_gcc/bin/gcc` and `/home/nomo/my_gcc/bin/g++` instead of `gcc` and `g++`.
- To check if it is working, try some sample programs. On i386, compile and run `ex/nestcall`. On SPARC, compile and run `ex/strpass`. Supply `-O4` option to enable optimization. When

you run it with no arguments, if it says `OK`, the patched gcc is working. Before gcc is not patched, these programs do not say `OK` and will produce the following outputs:

– `strpass` on SPARC:

```
harp:1014% ./strpass
1 : st_main forks foo
2 : foo blocks
3 : foo blocked, restart foo
4 : foo restarted and finishes
************************************************************
Structure argument to ST_THREAD_CREATE(foo(p)) is corrupted.
This is a known problem on SPARC.
You may either patch gcc to fix this problem (see manual),
or just avoid passing struct args to procedure calls with
ST_THREAD_CRREATE
************************************************************
```

– `nestcall` on i386:

8.3 Detailed Description of Patches

Here we describe what these patches do in detail. The purpose is to give information to an expert. The following description makes sense only when you understand what is the problem with the original gcc. See Section 10.1 [Floating SP Problem], page 73 and Section 10.3 [Structure passing on SPARC], page 76 for the description of the problems with the original gcc.

- `calls.c` in function `expand_call`: There is a loop that determines, for each argument to a procedure call expression, whether the evaluated value of the argument should be placed on a temporary location or can be directly placed on the stack. The loop in gcc 2.7.2.3 looks like this (including preceeding comments).

```
 1: /* If we preallocated the stack space, and some ...
 2:
 3:                     ...
 4:
 5:                                     ... into the stack.  */
 6:
 7: for (i = 0; i < num_actuals; i++)
 8:   if (is_const
 9:    || ((args_size.var != 0 || args_size.constant != 0)
10:        && calls_function (args[i].tree_value, 1))
11:    || (must_preallocate && (args_size.var != 0 || args_size.constant != 0)
12:        && calls_function (args[i].tree_value, 0)))
13:     {
14:    /* If this is an addressable type, we cannot pre-evaluate it.  */
```

```
15:     if (TREE_ADDRESSABLE (TREE_TYPE (args[i].tree_value)))
16:       abort ();
17:         ...
```

This loop judges whether each argument in turn includes a nested procedure call or a call to `alloca` function. `calls_function(`*a*`, 1)` returns 1 if expression *a* includes a call to `alloca`, and `calls_function(`*a*`, 0)` returns 1 if expression *a* includes any procedure call. The essential modification is to change the second parameter to `calls_function` at line 10 to zero, which was originally one. This essentially treats any function call as if it were a call to `alloca`.

The patch passes zero to `calls_function` at line 10 only when a flag (`flag_calls_clobber_sp`) is set. It also guards the modification by a `#if` directive in the way described earlier in this chapter. The resulting code looks like this:

```
  /* If we preallocated the stack space, and some arguments must be passed
     on the stack, then we must precompute any parameter which contains a
     function call which will store arguments on the stack.
     Otherwise, evaluating the parameter may clobber previous parameters
     which have already been stored into the stack.  */
   for (i = 0; i < num_actuals; i++)
    if (is_const
#if !defined(STHREADS_SAFE) || STHREADS_SAFE
       || ((args_size.var != 0 || args_size.constant != 0)
           && calls_function (args[i].tree_value,
                              (flag_calls_clobber_sp ? 0 : 1)))
#else  /* !defined(STHREADS_SAFE) || STHREADS_SAFE */
       || ((args_size.var != 0 || args_size.constant != 0)
           && calls_function (args[i].tree_value, 1))
#endif /* !defined(STHREADS_SAFE) || STHREADS_SAFE */
       || (must_preallocate && (args_size.var != 0 || args_size.constant != 0)
           && calls_function (args[i].tree_value, 0)))
     {
       /* If this is an addressable type, we cannot pre-evaluate it.  */
       if (TREE_ADDRESSABLE (TREE_TYPE (args[i].tree_value)))
         abort ();
```

What this patch essentially does is to tell gcc that any function call may alter the value of stack pointer in an unexpected way, just as `alloca` does. On Pentium, where stack pointer is allowed to move to push and pop arguments to procedure calls, the compiler can push an argument on stack top as soon as it is evaluated, no matter whether other arguments have a nested procedure call. A nested procedure call can push its arguments on top of the arguments under construction for the outer procedure call. When the nested procedure call returns, its arguments can be freed and the rest of the arguments to the outer procedure call can then be pushed on the stack. There is one exception to this rule; when a nested procedure call is a call to `alloca`, the compiler cannot infer the value of stack pointer after the call, hence cannot place arguments evaluated before `alloca` on top of the stack at that point. It instead must perform calls to `alloca` first and evaluate the rest of the arguments. The results from `alloca`s

must be saved in temporary slots, which are addressed via frame pointer and then moved to stack top just before calling the outer function.

By treating any function call as if it were a call to `alloca`, we effectively force gcc to pre-evaluate all nested procedure calls.

- `flags.h`: Just add a declaration of flag `flag_calls_clobber_sp`, which is set when a command line option `-fcalls-clobber-sp` is given. Add the following piece of code at the end of the series of declarations of flags.

```
/* Non-zero any function call is supposed to clobber SP */
#if !defined(STHREADS_SAFE) || STHREADS_SAFE
extern int flag_calls_clobber_sp;
#endif /* !defined(STHREADS_SAFE) || STHREADS_SAFE */
```

- `toplev.c`: Just add a definition of flag `flag_calls_clobber_sp` and update the table of flags (`f_options`). Add the following piece of code at the end of the series of definitions of flags.

```
/* Non-zero any function call is supposed to clobber SP */
#if !defined(STHREADS_SAFE) || STHREADS_SAFE
int flag_calls_clobber_sp = 0;
#endif /* !defined(STHREADS_SAFE) || STHREADS_SAFE */
```

Also add the following at the end of the table `f_options`:

```
#if !defined(STHREADS_SAFE) || STHREADS_SAFE
  , {"calls-clobber-sp", &flag_calls_clobber_sp, 1}
#endif /* !defined(STHREADS_SAFE) || STHREADS_SAFE */
```

- `function.c` in function `assign_parms`: There is a code fragment that copies incoming parameters that are passed by an invisible reference, if necessary. Search `FUNCTION_ARG_CALLEE_COPIES`. You will find a fragment that looks like:

```
#if !defined(STHREADS_SAFE) || STHREADS_SAFE
#ifdef FUNCTION_ARG_MAY_BE_DESTROYED
        /* If we are passed an arg by reference and it may be destroyed
           for any unknown reason (even if the callee does not modify
           it by itself).
           PASSED_TYPE and PASSED mode now refer to the pointer, not the
           original argument, so we must recreate them in the call to
           FUNCTION_ARG_CALLEE_COPIES.
        */
         else if (passed_pointer
                  && FUNCTION_ARG_MAY_BE_DESTROYED (args_so_far,
                                                    TYPE_MODE (DECL_ARG_TYPE (parm)),
                                                    DECL_ARG_TYPE (parm),
                                                    ! last_named)
                  && ! TREE_ADDRESSABLE (DECL_ARG_TYPE (parm)))
          {
            rtx copy;
            tree type = DECL_ARG_TYPE (parm);
           /* This sequence may involve a library call perhaps clobbering
              registers that haven't been copied to pseudos yet.  */
```

```
                        ...
```

This is an `else if` branch of a larger `if` statement. Make a copy of the original `else if` branch, guard it by an `#if` directive, and insert it *before* the original one. Then, replace `FUNCTION_ARG_CALLEE_COPIES` with `FUNCTION_ARG_MAY_BE_DESTROYED` in the new branch. That is, the entire `if` statement now looks like:

```
              /* If we were passed a pointer but the actual value
                 can safely live in a register, put it in one.  */
              if (passed_pointer && TYPE_MODE (TREE_TYPE (parm)) != BLKmode
                            ...
                                              )
                {
                            ...
                }
#if !defined(STHREADS_SAFE) || STHREADS_SAFE
#ifdef FUNCTION_ARG_MAY_BE_DESTROYED
              /* ... */
              else if (passed_pointer
                       && FUNCTION_ARG_MAY_BE_DESTROYED (args_so_far,
                                                         ...)
                       && ! TREE_ADDRESSABLE (DECL_ARG_TYPE (parm)))
                {
                  rtx copy;
                  tree type = DECL_ARG_TYPE (parm);
                            ...
                }
#endif /* FUNCTION_ARG_MAY_BE_DESTROYED */
#endif /* !defined(STHREADS_SAFE) || STHREADS_SAFE */
 #ifdef FUNCTION_ARG_CALLEE_COPIES
              /* ... */
              else if (passed_pointer
                       && FUNCTION_ARG_CALLEE_COPIES (args_so_far,
                                                      ...)
                       && ! TREE_ADDRESSABLE (DECL_ARG_TYPE (parm)))
                {
                  rtx copy;
                  tree type = DECL_ARG_TYPE (parm);
                            ...
                }
#endif /* FUNCTION_ARG_CALLEE_COPIES */
```

The inserted branch handles the case in which an incoming argument passed by reference may be destroyed for any unknown reason. The callee copies such arguments to its own place. GCC happens to do the same thing for handling cases in which an argument is passed by reference and it is the callee's responsibility to copy it before modifying the location.

Although duplicating code in this way makes maintenance troublesome, and there is a way to handle both cases in a single `else if` branch, we currently treat them in two separate `else if` branches. Reasons are as follows:

  - The original branch (handling the case where `FUNCTION_ARG_CALLEE_COPIES(...)` is non-zero) may be improved in future; if gcc recognizes cases where the argument is not modified by the callee, the callee does not have to copy it. On the other hand, the inserted branch must copy it even when the callee itself does not modify it.
  - We want to maintain source code so that our addition can be syntactically separated from the original source, with a line-by-line granularity. The original source code guards the `else if` branch by `#ifdef FUNCTION_ARG_CALLEE_COPIES`. This makes it difficult to add anything that must be effective even if `FUNCTION_ARG_CALLEE_COPIES` is not defined, without modifying the line `#ifdef FUNCTION_ARG_CALLEE_COPIES`.

- `config/sparc/sparc.h`:
  - Define two macros `MASK_CALLEE_COPIES_STRUCT_ARGS` and `TARGET_CALLEE_COPIES_STRUCT_ARGS`. To determine the correct value for `MASK_CALLEE_COPIES_STRUCT_ARGS`, look through `sparc.h` and search all definitions that look like `#define MASK_xxx`. Any mask is a power of two and is a mask to check if a particular SPARC-specific flag is defined. Add a new mask to this series of masks. For example, if you find that the last mask defined in the file is `0x400000`, you can use `0x800000` for `MASK_CALLEE_COPIES_STRUCT_ARGS`, leading to the following definition:

    ```
    #define MASK_CALLEE_COPIES_STRUCT_ARGS 0x800000
    ```

    Always define `TARGET_CALLEE_COPIES_STRUCT_ARGS` like this:

    ```
    #define TARGET_CALLEE_COPIES_STRUCT_ARGS \
              (target_flags & MASK_CALLEE_COPIES_STRUCT_ARGS)
    ```

    To summarize, you add the following lines:

    ```
    #if !defined(STHREADS_SAFE) || STHREADS_SAFE
    #define MASK_CALLEE_COPIES_STRUCT_ARGS 0x800000
    #define TARGET_CALLEE_COPIES_STRUCT_ARGS \
              (target_flags & MASK_CALLEE_COPIES_STRUCT_ARGS)
    #endif /* !defined(STHREADS_SAFE) || STHREADS_SAFE */
    ```

  - Add a new command-line option `-mcallee-copies-struct-args` to `TARGET_SWITCHES` macro. Search `#define TARGET_SWITCHES` and add a line `{"callee-copies-struct-args", MASK_CALLEE_COPIES_STRUCT_ARGS} \` to an appropriate place. Do not forget the trailing backslash, which is necessary because it is in a macro definition. To guard the modification by an `#if` directive, we have to copy the entire definition, leading to the following code:

    ```
    #if !defined(STHREADS_SAFE) || STHREADS_SAFE
    #define TARGET_SWITCHES  \
      { {"fpu", MASK_FPU | MASK_FPU_SET},      \
        {"no-fpu", -MASK_FPU},          \
            ...
        {"64", MASK_64BIT},                    \
        {"stack-bias", MASK_STACK_BIAS},       \
        {"no-stack-bias", -MASK_STACK_BIAS}, \
        {"callee-copies-struct-args", MASK_CALLEE_COPIES_STRUCT_ARGS}, \
    ```

```
    SUBTARGET_SWITCHES                          \
    { "", TARGET_DEFAULT}}
#else /* !defined(STHREADS_SAFE) || STHREADS_SAFE */
#define TARGET_SWITCHES  \
  {{"fpu", MASK_FPU | MASK_FPU_SET},       \
    {"no-fpu", -MASK_FPU},          \
        ...
    {"64", MASK_64BIT},                      \
    {"stack-bias", MASK_STACK_BIAS},         \
    {"no-stack-bias", -MASK_STACK_BIAS}, \
    SUBTARGET_SWITCHES                          \
    { "", TARGET_DEFAULT}}
#endif /* !defined(STHREADS_SAFE) || STHREADS_SAFE */
```

– Add the definition of a macro `FUNCTION_ARG_MAY_BE_DESTROYED(CUM, MODE, TYPE, NAMED)` as follows:

```
/* non-zero if an argument passed by reference may be destroyed
   for any unknown reason, even if the callee does not write
   to it by itself. */
#if !defined(STHREADS_SAFE) || STHREADS_SAFE
#define FUNCTION_ARG_MAY_BE_DESTROYED(CUM, MODE, TYPE, NAMED) \
  TARGET_CALLEE_COPIES_STRUCT_ARGS
#endif /* !defined(STHREADS_SAFE) || STHREADS_SAFE */
```

This macro takes parameters just for possible future extensions and for consistency with existing macro `FUNCTION_ARG_CALLEE_COPIES`.

These patches essentially force a callee to copy incoming structure parameters to its private place. In the original SPARC v8 calling convention, when a procedure passes a structure to another procedure by value, it copies the structure to a place in the caller's frame and passes the pointer to it to the callee. The callee can assume that the structure persists throughout the lifetime of the procedure call. Setting `FUNCTION_ARG_MAY_BE_DESTROYED(CUM, MODE, TYPE, NAMED)` to non-zero tells gcc that this no longer holds, therefore the callee must copy its incoming structure parameters to its own place, no matter whether it modifies these arguments or not.

9 Advanced Topics

This chapter describes what happens on `ST_THREAD_CREATE` and how threads are distributed across processors. We then make a few remarks on programming practices that should be adopted and avoided, and on where should you insert `ST_POLLING`.

9.1 How does it work, basically?

Suppose procedure f calls `ST_THREAD_CREATE(`g`())`, f performs a normal C procedure call ``g`()`. What is important here is that it performs the procedure call just as if it were a sequential call. It also records the fact that the procedure call is a fork point. This way, the runtime system knows that the rest of f can progress even when g is blocked, or when an idle worker shows up. The exact method by which we record a fork point is unimportant; we note that it only takes a couple of instructions.

If ``g`()` terminates without blocking, it returns to f just as a normal procedure returns. What happens when g blocks? A thread blocks , for example, g calls `st_mutex_lock(`j`)`, but m is currently locked by another thread. In this case, we must be able to suspend execution of g for now and later resume it when m is unlocked. We achieve this by splitting the stack at the fork point where g was spawned; the activation frame of g is logically decoupled from the stack and the rest of f continues (when f is still not migrated to another worker). g's activation frame physically remains in the original stack, occupying the top portion of the stack. Stack pointer is kept above g, while frame pointer now points to f (we use a terminology as if a stack grows upward. A frame x is above y if x is nearer to the growing end of the stack than y). Subsequent procedure calls by f allocate a frame above g. When g later restarts its computation, g frame is logically connected to the current frame and becomes the child of it. This frame may of course be different from g's original caller. When g terminates, the control returns to the new parent, not the original caller.

In general, each worker maintains its stack pointer below any unfinished frame. A procedure call or `ST_THREAD_CREATE` allocates a new frame above stack pointer. Frames of blocked (yet unfinished) threads are logically decoupled from the stack, but physically stay within the stack. Frames of restarted threads are reconnected to the stack.

Leaving blocked frames in their original positions and making further allocations on top of them, we may leave free spaces in a middle of a stack. We do not try to reuse such holes within a stack. We always allocate a new frame on top of the stack. Once a region becomes free in a middle of a stack, it is reused only when no spaces are occupied above it. When such a situation occurs, stack

pointer is reset to top of the unfinished frames. Exact details as to how to reclaim such spaces is not described here; we only mention that such reclamation of space occurs when the programmer calls `ST_POLLING`. This is the first reason why you have to call `ST_POLLING` periodically.

What about thread migration? Interestingly, a thread migration is just a combination of thread blocks and a resume. Suppose that both f and g are in worker W's stack and W wants to migrate f to another (idle) worker U. At this point, W temporarily blocks g and resumes f. Next f also temporarily blocks and resumes the parent of f. The parent of f passes the context of f to U through shared-memory and then restarts g. U picks up the context of f and restarts it.

In general, to migrate a thread from W to U, W first blocks all frame above the migrating thread. Next W blocks the migrating thread too. W then restarts threads above the migrating thread. The migrating thread is picked up by U and restarted. This way, the migrating thread is pulled out from one stack and reconnected to another.

A thread migration also occurs when `ST_POLLING` is called. When a worker becomes idle, it randomly selects another worker and sends a request. `ST_POLLING` checks whether a request has arrived to the calling worker and if it has, performs the above operation to migrate a thread to the idle worker. This is the second (and the last) reason why you have to call `ST_POLLING` periodically.

The remaining question is, when a worker gets a request, how does it select a thread to migrate. We simply migrate the thread at the bottom of the stack. For example, suppose f performs `ST_THREAD_CREATE(g())`, `g` performs `ST_THREAD_CREATE(h())`, and `ST_POLLING()` finds a request from an idle worker. In this case, f migrates. When the second request soon arrives to the same worker, then g migrates. When call tree is fairly balanced, as in the Fibonacci example (See Section 2.1 [An Example], page 6), this strategy works very well; all workers will soon get a subtree which is fairly near to the root and concentrate on their local work. With this task distribution strategy in mind, let's go on to the next section.

9.2 A recommended programming style for performance

There are many ways to express parallelism that exist in your program. Consider you want to perform *100,000* procedure calls `f(`i`)` (i = 0, ... 99,999) in parallel. It is most likely to be written using a `for` loop if they were done in series:

```
for (i = 0; i < 100,000; i++) {
   f(i);
```

```
}
```

Therefore, you are likely to parallelize it as follows (actually, `f` may take extra arguments to achieve synchronization, and we may have to insert some `ST_POLLING`s, but we ignore them for simplicity):

```
for (i = 0; i < 100,000; i++) {
   ST_THREAD_CREATE(f(i));
}
```

This works only when each `f` involves a significant amount of work.

To directly get to the point, the recommended style is so-called divide-and-conquer method. We introduce an auxiliary function `f_aux(`p`, `q`)`, which performs `f(`p`)`, ... , `f(`q`-1)`, and perform recursive calls in parallel (again, synchronization and `ST_POLLING` are omitted).

```
f_aux(p, q) {
  if (p + 1 == q) f(p);
  else {
    m = (p + q) / 2;
    ST_THREAD_CREATE(f_aux(p, m));
    f_aux(m, q));
  }
}
```

This works well even when each `f(p)` does a small amount of work. The previous section has explained why this style works.

(This code performs roughly the same number of procedure calls to `f_aux` as the number of procedure calls to `f`. Therefore it adds the overhead of an extra procedure call to each `f`. If `f` is too tiny to tolerate this overhead, change the condition in the second line so that `f_aux` performs several calls to `f` in a sequential loop. For example:

```
f_aux(p, q) {
  if (p + T >= q) {
    for (i = p; i < q; i++) f(i);
  } else {
    m = (p + q) / 2;
    ST_THREAD_CREATE(f_aux(p, m));
```

```
        f_aux(m, q));
    }
}
```

Here, `T` is chosen so that calling `f` T times is likely to mask the overhead of a procedure call).

What's wrong with the first program? Well, it is not always wrong, and I know it is preferred in terms of maintenance cost and simplicity. Here I explain what happens when we execute the first program and in which circumstances is it OK and in circumstances should it be avoided.

When a worker, W, creates a thread for `f(0)`, W starts `f(0)`. W keeps its continuation (representing `f(1)` ... `f(99,999)` on its stack. A request from an idle worker, `X`, will soon arrive and `W` gives the continuation to `X`. Now, `X` creates a thread for `f(1)`, leaving the rest of the work `f(2)` ... `f(99,9999)` on its stack. A request from another idle worker then steals it and invokes `f(2)`, leaving `f(3)` ... `f(99,999)` on its stack, and so on. Observe that all these thread migrations occur in sequence. Assume for simplicity that each `f(`i`)` takes approximately the same time, this program assigns iterations in a round-robin manner, resulting in approximately 100,0000 sequential thread migrations. This places a lower bound of the time required to finish this loop, which is 100,000 times M, where M is the migration cost. In the original sequential loop, it takes 100,000 times F, where F is the cost of a *f(i)*. Therefore, the achievable speedup on any number of processors is F/M, which may be quite small when F performs only a tiny amount of work (M is an order of thousand instructions).

Here is a summary: if F represents a significant amount of work, or you have only a small number of processors so that the above limit will not be observed, the simple parallel loop will be OK.

9.3 Where should you insert ST_POLLING?

The final topic is where to insert `ST_POLLING`. Recall that `ST_POLLING` serves two purposes.

- Thread migration: Pick up, if any, request form an idle worker.
- Stack management: Free dead frames on top of the stack.

The constraint is that you must put enough `ST_POLLING` to make them work. Another constraint is that you want to minimize `ST_POLLING` to save your coding time (and to save some overheads, but the overhead is not large (like 10 instructions). When you are not sure, play safe; put it).

Here are rules as to where you should insert `ST_POLLING` to make thread migration and stack management work. Each imposes a similar but different constraints, but the total result is as follows:

1. When you have a non-leaf procedure (a procedure that may call another procedure) F, insert one at anywhere between the head of F and the first procedure call performed in F.
2. When you have a non-leaf procedure F, insert one at anywhere between the last procedure call in F and the end of F.
3. (Very strictly speaking) When you have a non-leaf procedure F, insert one at anywhere between two procedure calls performed in F (e.g., just after each procedure call). You can forget it in practice.
4. When you have a loop, insert one at anywhere in the body, so that each iteration calls it.

In the last case, you do not have to do it when you know the loop is very small (and therefore never spends a long time).

For example, in pfib, the original code without `ST_POLLING` is structured like this:

```
pfib (...)
{
  ... (a) ...
  if (...) { ... (b) ... return; }
  else {
    ... (c) ...
    pfib (...);
    pfib (...);
    ... (d) ...
  }
}
```

According to the rule 1, you insert ST_POLLING to: `(a)` or `(c)`. We advise `(c)`, because no procedure calls are made along the path `(a)->(b)->return`. This saves the overhead of `ST_POLLING` at leaves. According to the rule 2, you insert `ST_POLLING` to `(d)`.

Do you want to know why? Let me describe constraints more precisely for thread migration and stack management.

First, for thread migration, you must ensure that `ST_POLLING` is called often enough so that each worker is responsive to requests from idle workers. Following the above rules, the maximum interval in which no `ST_POLLING`s are called is bounded (in terms of instruction counts), regardless

of input to the program. If the first rule would not be followed, a recursive function may create an arbitrary long polling-free interval. For example,

```
f(int n)
{

  ...;

  f(n - 1);
}
```

If there would be no ST_POLLINGs in ..., the length of a polling-free interval is proportional to N, which is not constant across all inputs.

If the second rule would not be followed, a similar situation occurs during a chain of recursive calls are shrinking. For example,

```
f(int n)
{


  f(n - 1);

  ...;
}
```

If there would be no ST_POLLINGs in ..., the length of a polling-free interval is proportional to N, which is not constant across all inputs.

The third rule is irrelevant for thread migration.

The fourth rule will be obvious. In practice, you might want to save overhead of calling ST_POLLING, if the body of the rule is very small. If you know the entire loop does not take a long time, you may skip it.

For stack management, you must ensure that you do not miss a point where the entire space above the current frame (upto stack pointer) is free. Notice that a new frame is always allocated on top of stack pointer. Therefore, if you do not call ST_POLLING at such a point and make another procedure call, these frames do not reuse space that is otherwise reusable. In the worst case, this will result in stack overflow that can otherwise be avoided.

A frame becomes dead only when a procedure returns, thus it suffices to call `ST_POLLING` just after a procedure returns. This is where the third rule comes from. Strictly speaking, all you need for stack management is this rule, but it may seem impractical when manually inserted by the programmer. As a rough approximation, the second rule is in practice OK. This is because, in common programming styles, a procedure `F` finishes only if all its decedent have finished. When this is the case, it is guaranteed that, if procedure `F` performs the last `ST_POLLING` and returns, free space left on top of the stack is none or the frame of `F`, because all descendent of `F` have also been finished.

9.4 Implementing Synchronization Primitives

This section describes how to implement your own synchronizing data structure on top of the bottom-level primitives (`st_suspend_thread` and `st_resume_thread`). Before using bottom-level primitives, examine if it is possible to implement your synchronization just by combining higher-level primitives (mutex, semaphore, and condition variables). High-level primitives often save your time for debugging. In particular, condition variable is very general. You need bottom-level primitives only when they are necessary for performance or when your synchronization is most conveniently expressed with them.

Consider a simple example in which two threads, a producer (writer) and a consumer (reader), are synchronized through a location. The synchronization constraint is that the consumer must wait for the producer to put the result. We assume there is only a single writer and a reader. We must still resolve the races between the writer and the reader.

```
 1  typedef struct buffer
 2  {
 3    st_int_loc_t status; /* 0 for empty, 1 for full */
 4    int val;
 5    st_context_t waiter; /* waiting context */
 6  } * buffer_t;
 7
 8  void init_buffer(buffer_t b)
 9  {
10    ST_INT_LOC_INIT(&b->status, 0);
11    b->waiter = 0; /* no threads are waiting */
12  }
13
14  int GET(buffer_t b)
15  {
16    if (st_read_int(&b->status) == 1) {
17      return b->val; /* full */
```

```
18    } else {
19      /* atomically swap -1 and the current status */
20      int s = st_read_and_lock_int(&b->status);
21      if (s == 1) {
22        int val = b->val;
23        st_write_and_unlock_int(&b->status, s); /* unlock */
24        return val;
25      } else {
26        struct st_context c[1];
27        c->valid = 0;
28        b->waiter = c;
29        st_write_and_unlock_int(&b->status, 0); /* unlock */
30        st_suspend_thread(c);
31        return b->val;
32      }
33    }
34  }
35
36  int PUT(buffer_t b, int val)
37  {
38    /* atomically swap -1 and the current status */
39    st_read_and_lock_int(&b->status);
40    b->val = val;
41    if (b->w) {
42      /* somebody is waiting */
43      st_context_t w = b->w;
44      st_write_and_unlock_int(&b->status, 1); /* unlock */
45      st_resume_context(w);
46    } else {
47      st_write_and_unlock_int(&b->status, 1); /* unlock */
48    }
49  }
```

The buffer has three fields. The first one is `val`, the value written by the writer. The second is `waiter`, which is a pointer to a context. When a reader is blocked waiting for the value, the context is written to waiter. Waiter is zero when nobody is waiting. Finally it has `status` field to describe the status of the buffer. It also serves as a lock variable to make modifications to multiple words atomic. It is either full, in which the writer has already performed a `PUT`, empty, in which it has not.

`GET` checks `b->status` and if it is not full, it suspends the current thread by putting a context to `b->waiter` (the line 26—31). `PUT` checks `b->waiters` and if it is not zero, it resumes the waiting thread (the line 42—45).

To resolve possible race conditions between the writer and the reader, the above code looks more complicated than it ought to be. When the reader decides to suspend, we must guarantee that *the writer will see* `b->waiter` *after the reader writes to* `b->waiter`. We guarantee this by associating a lock with the status field. Before the reader decides to suspend, it first obtains the lock by `st_read_and_lock_int(&b->status)`.

Since the writer first obtains the lock before reading `b->waiter`, it is guaranteed that at line 26, the writer is not reading `b->waiter`. The rest of the code is straightforward.

Here is very important notes on the use of `st_read_and_lock_int`. First, do not forget unlock. Second, do not spend a long time when you are locking any location by `st_read_and_lock_int`. Once you obtain a lock, you must unlock it *as soon as possible*. Typically, what you do while locking something is a very short sequence of simple primitives like arithmetics. Do not call any function that may take a long time. For example, do not call malloc or printf. Finally, never try to obtain two or more locks at the same time. This is naturally avoided if you try to make critical sections as short as possible.

In short, `st_read_and_lock_int` itself should not be regarded as a thread synchronization primitive, but be a primitive to support atomic transactions to a single location. If you want to have an exclusive access to a data structure for a long time (this is already an unadvisable programming style, though) for some reasons, use synchronization primitives such as mutex.

9.5 What Does stgcc Command Do?

`stgcc` (`stg++`) is a simple script that drives compilation. It retains the same interface as `gcc` (`stg++`); it accepts the same set of command line options, except for those that fundamentally conflict with execution scheme of StackThreads/MP. The following descriptions about `stgcc` also apply to `stg++`.

`stgcc` *is not a preprocessor or a compiler* that understands C/C++ syntax and translates C/C++ programs in non-trivial ways. In fact, it feeds given C/C++ programs directly to `gcc`, passing command line (almost) as is. As a consequence, you can use almost all command line options and non-standard extensions supported by `gcc`.

What is `stgcc` then? It is a *postprocessor* of `gcc`. Whatever options you gave to `stgcc`, it lets `gcc` to generate assembly code and postprocess the assembly. The postprocessor is written in an AWK script and does the following jobs:

- It scans the assembly instructions of procedures and examines their stack frame formats. Information about stack frame formats are collected in a static table and attached to the original assembly file.
- It modifies the epilogue sequence of procedures for frame management purposes. It inserts about five instructions to each procedure.
- Upon linking, it calls an AWK script (called `stlink.awk`) to collect tables for all object files (Therefor it is mandatory to call `stgcc` to generate final executable).

Options of `gcc` that specifies where to stop compilation process (which are `-E`, `-S`, and `-c`) are also supported. Compilation transparently proceeds, if necessary, by compiling the postprocessed assembly file into an object file and by linking object files into executable.

There are other minor things that `stgcc` do:

- It automatically gives `-I$ST_DIR` to `gcc`, so that standard include files are automatically searched.
- It automatically links StackThreads/MP library.

9.6 What Does the Postprocessor Do?

Undocumented yet.

10 Known Limitations and Subtleties

While StackThreads/MP tries to support multithreading with unmodified GCC/G++, which does not have a notion of multiple threads of control, there are certain limitations which are hard to be fixed without changing the compiler or calling conventions. Here I list all the problems that I know of currently.

10.1 Floating SP Problem

If you apply patches under `gccpatch` to your gcc, you do not worry about the problem (see Chapter 8 [Applying Safety Patches to GCC], page 55 for how to apply patches).

In practice, this problem occurs only on Pentium, and only when you write a nested procedure call to a procedure that calls `ST_THREAD_CREATE`, `ST_POLLING`, `ST_STEAL_REQ_CHECK`, and any other synchronization primitives (mutex, semaphore, procedure compiled by `stgcc`). For example, if you write `f(g(x), 1, 2, 3, 4)` and if `g` may call `st_mutex_wait`, this is a nested procedure call. In such cases, outer procedures may receive wrong arguments. In this example, `f` may observe wrong arguments (more specifically, 1, 2, 3, and 4 may not be correctly passed to `f`).

To workaround this problem without patching gcc, avoid nested procedure calls (at least on Pentium), and do not call the above primitives within C++ constructors (because they are often implicitly called to cast arguments).

In theory, this problem may occur when the compiler would become cleverer. The execution scheme of StackThreads/MP manages stack pointer so that it always points to 'top of the stack,' above which no unfinished frames are allocated. In particular, the stack pointer may not point to the currently running frame. Therefore, the value of stack pointer may be different from gcc's idea it. This mismatch causes a problem if gcc exploits it to optimize things (see Section 9.1 [How does it work, basically?], page 63 for more info about how does StackThreads/MP manage stack pointer).

The calling convention on a particular CPU determines how to manage stack pointer. The compiler may assume that stack pointer after a procedure call has a certain statically-computable offset from its original value, accoding to the calling convention. More specifically, on most calling conventions, the value of stack pointer does not change across a procedure call; in other cases (e.g., 'Pascal' convention on Pentium), the callee is responsible for cleaning up arguments pushed on stack. Either case, the compiler knows where does stack pointer point to after a procedure call

(more precisely, it statically knows how much does it differ from its original value), and if it wishes, it may exploit the fact to optimize arguments passing in procedure calls.

Suppose, for simplicity, a hypothetical convention in which *i*th argument to a procedure is stored at SP[4(*i*-1)], where SP is stack pointer, and SP does not change across a procedure call. Given a procedure call `f(g(x), 1, 2, 3, 4)`, for example, the compiler may generate the following sequence:

```
 1       /* write 1, 2, 3, 4 */
 2       SP[ 4] = 1;
 3       SP[ 8] = 2;
 4       SP[12] = 3;
 5       SP[16] = 4;
 6       /* call g(x) */
 7       SP[ 0] = x;
 8       r = call g;
 9       /* write the result value*/
10        SP[ 0] = r;
11            call f;
```

Since `g` is nested inside `f`, it first performs a procedure call `r = g(x)` (line 8), and then `f(r, 1, 2, 3, 4)` (line 11). Before calling `g`, however, it puts a part of arguments to `f` at `SP[4]` ... `SP[16]` (line 2 – 5). When `g` returns (at line 10), the value of SP is the same as the value of SP at line 5, therefore `f` can correctly receive arguments 1 ... 4.

The fundamental problem here is that the argument-writing sequence for `f` was interrupted by another procedure call `g`, which was safe in the original convention because a procedure call does not change SP, but is not in StackThreads/MP convention in which SP points to whatever place happens to be top of the stack at that point. In general, we must tell the compiler that SP becomes 'unknown' after any procedure call.

As described, this problem in theory occurs whenever the compiler does a good job at placing arguments to procedures. It may in theory occur on any architecture and for almost any program, even if it does not use a literally nested procedure call (note that there are no reasons why the compiler cannot do the same optimization for non-nested version of the above example, which is, `{ r = g(x); f = (r, 1, 2, 3, 4) }`.

Let us now move our focus on what does the current gcc do, and let me describe why do I think this problem occurs only for literally nested procedure calls on Pentium. First of all, compilers use SP as the base register when they write arguments to procedures. For other purposes, frame pointer (FP) is used. This is because FP never changes within a single procedure, whereas SP in

general does because of `alloca`. Therefore named locations (e.g., local variables) are accessed via FP. On the other hand, arguments to procedures are going to be accessed via FP *of the called procedure*, which is SP of the current procedure. Therefore arguments are written via SP. This limits the situation in which a compiler exploits the fact that SP is known after a procedure call. To summarize, a compiler exploits the fact only to schedule arguments passing to procedure calls.

On all CPUs I know of except for Pentium, first several (typically 4 to 6 words) arguments are passed on registers. This makes opmization less important on RISC CPUs. Second, the optimization is potentially complex because `alloca` make SP essentially unknown. Therefore any compiler that tries to optimize arguments passing must take it into account. A compiler cannot simply assume "SP is constant."

10.2 Possible Slowdown by Disabling Inline

We must make sure that a procedure call expression that appears in `ST_THREAD_CREATE` is not inline-expanded. For example, when you write `ST_THREAD_CREATE(f(x))`, you must make sure procedure call `f(x)` is not inlined by `gcc`. When this condition holds, separate stack frames are allocated for f and the caller of f. StackThreads/MP runtime can then execute f and the caller in parallel by executing the two frames on different processors. This is impossible if f`(...)` is inlined, because their stack frames are merged. `stgcc` command automatically disables inlining by giving `gcc` suitable options (`-fno-inline-functions` for normal functions and `-fno-default-inline` for C++ member functions defined in a class definition). This is safe, but sometimes results in a noticeable slowdown compared to `gcc`, particularly for C++ programs.

Giving `-fno-inline-functions` and `-fno-default-inline` are very pessimistic, because we only have to make sure that inlining won't occur on parallel calls (`ST_THREAD_CREATE`). Inlining normal procedure calls are no problem, but the problem is we have no way to prohibit inlining of designated call sites. Here are a couple of suggestions to enable inlining sequential calls.

- You can explicitly add `inline` keyword for functions that are called only sequentially. These functions are subject of inlining even when `-fno-inline-functions` and `-fno-default-inline` are given.
- If writing explicit `inline` keyword is onerous (as is typically the case in C++ programs), and if you are sure that procedures called with `ST_THREAD_CREATE` are never inlined somehow, you can supply `--inline_functions_is_ok` and `--default_inline_is_ok` to `stgcc`. This enables inlining by `stgcc`. The remaining problem is how do you make sure that procedure calls in `ST_THREAD_CREATE` are never inline-expanded.

In practice, GCC/G++ never inline-expands procedure calls whose function part is not a simple function name. For example, in the following example, procedure call `f(x)` is likely to be expanded.

```
int f(int x) { return x + 1; }

g(int x)
{

  return 2 * f(x);
}
```

On the other hand, the following procedure call is, in practice, never inline-expanded. Therefore you can always make sure that inlining won't occur by calling a function through a function pointer.

```
int f(int x) { return x + 1; }
 g(int x)
{
   int (*pf)(int) = f;
   return 2 * pf(x);
}
```

In this way, for all procedures whose definitions are visible in the same compilation unit, you can make the call performed through a function pointer. The above code does not compile if the declaration of `f` were not given, but in this case the `gcc` does not inline-expand the call either.

To summarize, if the called function is not a simple function name, the procedure call won't be inlined. If the called function is a simple function name and its definition is not known in the compilation unit, again `gcc` does not inline it. Finally, if the definition is known, you can obscure the called function by transforming the function call in the way described above.

10.3 Structure passing on SPARC

If you apply patches under `gccpatch` to your gcc, you can forget about the problem.

The problem arises when a procedure performs two or more logically concurrent procedure calls that take a structure as an argument. The problem is, in such programs, structure arguments passed to a procedure call may be overwritten by subsequent ones.

For example,

```
1       typedef struct S { ... };
2       typedef struct T { ... };
```

```
 3
 4       void g(S x) { ... };
 5       void h(T y) { ... };
 6
 7       void f()
 8       {
 9         S s; T t;
10         ...
11         ST_THREAD_CREATE(g(s));
12         h(t);
13       }
```

Procedure calls `g(s)` and `h(t)` are logically concurrent and both take a structure argument. Procedure `g` may observe that its incoming parameter (`x`) is overwritten.

This problem occurs basically because SPARC calling convention uses the same storage to pass structure arguments to procedures. In this example, the storage for passing `s` to `g` and passing `t` to `h` are the same. The above program is equivalent to:

```
void f()
{
  union { S s; T t; } struct_passing_area;
  ...
  ST_THREAD_CREATE(g(&struct_passing_area.s));
  h(&struct_passing_area.t);
}
```

This is not a problem in sequential programs, because when `f` calls `h(t)`, procedure call `g(s)` has terminated. This is not the case when `g(s)` is evaluated by a new thread. What happens is that, concurrent activities `h` and `g` unconsciously share an unnamed storage.

To sidestep this problem, all procedures that take a structure parameter can copy the parameter into a local storage just after it is called. For example, the above procedure `g` can be rewritten as follows.

```
void g(S _x) { S x = _x; ... };
```

If such a copy is too expensive, and if you can change the signature of `g`, you may pass the structure by reference as follows.

```
void f()
{
  S s; T t;
  ...
  ST_THREAD_CREATE(g(&s));
  h(&t);
}
```

This does not work if `f` itself modifies `s` after calling `g`. Here we are facing essentially the same problem as before, but this time, the programmer can easily reason about the problem because the storage is explicitly named by the programmer. The real problem in the first program is that an unnamed storage is shared and thus the programmer cannot reason about it.

You may wonder why doesn't this problem occur on other CPUs or for scalar arguments? Procedures normally place arguments to a procedure call on registers and SP-relative locations (memory locations addressed via stack pointer). On Mips, Alpha, and Pentium, this is the case for any type of arguments. SPARC places scalar arguments on registers and SP-relative locations, but places structure arguments on FP-relative locations (memory locations addressed via frame pointer). For parameters passed via registers, the compiler-generated code, if necessary, saves them when the control departs from the procedure (by calling another procedure). For parameters passed via SP-relative locations, StackThreads/MP runtime system ensures that, if the control returns to `f` before `g` exits, the stack pointer is adjusted so that `h(t)` does use the same storage as `g(s)`. When the control returns to `f` before `g` exits, the values of the stack pointer at different at line 11 and line 12 are different, effectively renaming the storage for SP-relative locations.

10.4 C++ Global Constructors

Machine code generated by `stgcc` assumes that StackThreads/MP runtime has been initialized. StackThreads/MP library defines a `main` function that initializes the runtime before anything else and then calls a user-supplied entry point, `st_main`. When you have a C++ class that has a constructor and defines a global or static variable of that class, then the constructor is called before `main`. Therefore these constructors run before StackThreads/MP is initialized.

The problem occurs only when a constructor explicitly calls some StackThreads/MP primitives. If the constructor is a simple sequential constructor, as is typically the case, there are no problems.

If you ever wish to call a StackThreads/MP primitive from within a global constructor that runs before main, you must treat such a constructor call as a callback (see Section 6.2.2 [Setup TLS

Pointers], page 48 for more information). To summarize, when you have such a constructor, write `ST_CALLBACK_BEGIN()` at the entry and `ST_CALLBACK_END()` before return. For example, here is a class `Foo`, whose constructor creates a thread whenever an instance is created.

```
class Foo
{
  Foo {
    ST_CALLBACK_BEGIN();
    ST_THREAD_CREATE(fib(n, ...));
     ...
    ST_CALLBACK_END();
  }
}
```

10.5 Alloca

See Section 4.4 [Alloca is not supported (yet)], page 42 for information and possible workarounds.

10.6 Big arguments

This problem rarely occurs. When a procedure calls another procedure with extremely large arguments. In practice the problem only occurs when you have a structure that embeds a large array and passes an instance of the structure by value. Here is an example that may cause the problem.

```
typedef struct bigstr
{
  a[1000];
} bigstr;
void bigstr_eather(bigstr bs) { ... }
 void caller()
{
  bigstr b;
  bigstr_eater(b);
}
```

`stgcc` command invokes `gcc` and examines the assembly code generated by `gcc`. It collects information about stack frame formats of compiled procedures. Collected information includes, among other things, the size of the SP-relative region (the region accessed via stack pointer) of each stack frame. The size of SP-relative regions are determined by `gcc` depending on the size

of out-going parameters. `stgcc` must know the size of the region to do stack management. It computes the size by scanning the assembly instruction sequence and finding instructions that store something using stack pointer as the base register. For all such instructions, we check the offset + the operand size, and computes the maximum of these values. This is the size of the SP-relative region for that procedure.

In programs like this, the compiler may obscure such SP-relative stores. For example, if the location to be written cannot be reached by a store instruction using stack pointer as the base, the location may be computed to a general purpose register and then written using that register as the base. Even worse, arguments are written by calling functions like `memcpy`. Thus, `stgcc` may not be able to SP-relative regions accurately for such programs and as a consequence, stack management scheme may be broken.

I have never encountered this problem. You will not often encounter this problem either, because C/C++ programmers know that passing large structures are expensive in C/C++ and normally pass such structures by pointers.

There are some ways to make this problem even unlikely to occur, but I do not think of a perfect idea right now. To guarantee 100% safety, I guess we must eventually hack the compiler. We simply want to know gcc's idea about the size of SP-relative regions in a form of a comment or a directive in generated assembly instructions. Such information should be readily available *inside* a compiler and `gcc` seems to generate such information on some CPUs. Let's make it available all machines!

10.7 Interaction with pragma interface

The problem occurs when you are using a C++ header file that uses `#pragma interface`. When multiple C++ sources include such a header file and they are linked together, you observe a lot of erros saying symbols are duplicated. Symbols are names of member functions defined in the header file. This is probably a gcc bug (that exists at least in ver 2.7.2.1 on Solaris. I don't know situations in other platforms and on other versions of gcc), but I describe it here because the problem occurs faily often (most importantly when you use `iostream.h` that comes with `g++`).

`stgcc` command gives `-fno-default-inline` to `gcc`. When `gcc` compiles a C++ source that includes a header file that uses `#pragma interface`, it generates all the symbols defined in the header as *global symbols*, rather than file-scope or weak symbols. If multiple sources include such headers, they result in a link error. This is purely a gcc bug, I believe.

The problem does not occur without `-fno-default-inline`, in which case definitions of these symbols are not generated. The problem does not occurs without `#pragma interface` either, in which case symbols are defined as weak symbols. The manual of `gcc` says the specification about this pragma is in transition.

It would be nice if you would not encounter this problem unless you use this pragma. Unfortunately, however, library header files that come with `gcc` extensively use them! Most importantly, `iostream.h` uses it. A possible work around is to give `--default_inline_is_ok` to `stgcc`, which instructs `stgcc` not to pass `-fno-default-inline`. But then, you must make sure that `gcc` never inlines member function calls that appear in `ST_THREAD_CREATE`. You can achieve this by, for example, never calling member functions by `ST_THREAD_CREATE`. see Section 10.2 [Possible Slowdown by Disabling Inline], page 75 for more details about `--default_inline_is_ok`.

11 FAQ

11.1 Why am I interested in it?

Q: I am already accustomed to Pthreads, why must I use StackThreads/MP in the first place?

A: Pthreads and StackThreads/MP provide a similar set of basic primitives (thread create and synchronization). The most fundamental difference is that StackThreads/MP creates a thread very cheaply and tolerates a very large number of threads. This makes a significant difference in parallel programming styles.

In parallel programming with Pthreads, the programmer normally creates a constant number of threads and threads share a common task pool to distribute work (even more simply, each thread is statically assigned to a particular task at compile time). A piece of work is represented by a datum and it enters the task pool when it is found necessary. This explicit representation of a task is already onerous (Recall that in sequential programs, you simply call a function to perform a particular task. You don't have to define a data structure that represents a task). Worse, to achieve scalability, you are often forced to make the task pool concurrently accessible from multiple threads. This complicates the structure of the task pool, making overheads of queue manipulation larger and programming errors more likely. Most importantly, the resulting parallel program obscures the logic of the computation and looks very different from the sequential program, making the cost of maintenance significantly larger.

In parallel programming with StackThreads/MP, the programmer is encouraged to create threads dynamically when a piece of work is found necessary. It provides a primitive that attaches a thread to any procedure call in C and a thread creation costs only a little more than a sequential procedure call. The runtime system regards a thread as a unit of work and schedules whatever number of threads you created on whatever number of processors you have. Therefore, you need not manage a task pool explicitly. It is a library built on top of traditional thread libraries such as Pthreads. It internally spawns a fixed number of coarse-grain threads (i.e., threads in the Pthreads' sense) and each such thread creates arbitrary number of fine-grain threads (i.e., threads in StackThreads/MP's sense) and distributes them among the underlying coarse-grain threads. StackThreads/MP programming model is particularly attractive when starting from sequential source. Basically, some procedure calls in the sequential source are replaced by thread creations. In most cases, you can retain the overall structure of the sequential source. Everything comes from the fact that a thread creation is very cheap; therefore you do not worry about "too much threads."

There are some other reasons why you might be interested in StackThreads/MP.

- Portability among various parallel platforms. It encapsulates the underlying thread library.
- Stylized spin-locks: Tutorials of traditional thread libraries overly discourage spin-locks. They typically say "never do that." But they are necessary for performance and safe as long as care must be taken so that all critical sections are very short. Traditional thread libraries have no support for spin-locks, making them onerous and error-prone.
- Tools: It provides a simple performance profiler so that you can reason about how is your parallel program running well.
- Memory management primitives: It provides efficient alternatives to `malloc` in the `libc`. `malloc` is often a source of bottleneck in parallel programs. Alternatives include a simple replacement of `malloc` that are much more scalable than the `libc malloc`, a parallel conservative garbage collector, and explicit memory management based on regions (a.k.a. zones).

11.2 Does it subsume Pthreads?

No. It is intended to be a parallel programming tool. Pthreads, on the other hand, is intended to be both as a parallel programming tool and as a concurrent programming tool. Pthreads are sometimes used to spawn a 'background' job in programs that handle asynchronous inputs. For example, GUI programs may spawn a thread to redraw the screen while another thread is listening to events. Server programs may spawn a thread to process a request from a client while another thread is servicing another. These programs create threads not so much because they want to shrink the total computation time on multiprocessors, but because they want to make them responsive to inputs. In particular, it makes sense to create a thread even on uniprocessors, as long as threads are scheduled fairly.

Threads of StackThreads/MP do not replace such threads. In particular, it does not guarantee any fairness. It only guarantees that, as long as there is any thread in the system (and the main procedure is not returned yet), it schedules a thread. However, StackThreads/MP provides a way to directly create a thread (or a group of threads) of the underlying thread package. Therefore you can comfortably mix fine-grain threads, which are cheap to create but not scheduled fairly, and threads of the underlying thread package, which are expensive to create but are scheduled fairly by the OS (see Section 6.2.1 [Create a Worker Group], page 46 for details).

11.3 Compilation fails

Q: Compilation fails. Why?

A: `stgcc` simply calls `gcc` without any preprocessing. Therefore it virtually never happens that whatever errors `gcc` generated is caused by `stgcc`. Be sure that your program compiles with `gcc`.

There are still possibility that a program that successfully compiles with `gcc` does not successfully compile with `stgcc`.

- Environment variables: You have not set correct environment variables. You must define `ST_DIR`, `ST_CPU_TYPE`, `ST_OS_TYPE`, and `ST_THREAD_PKG`. See Section 1.3 [Installation], page 2 for details.
- Path: You must make sure your `PATH` environment variable contains `$ST_DIR/bin`.
- Command line errors: Some options that are normally accepted by `gcc` are prohibited in `stgcc`. When `stgcc` detects such an error, it displays an error message and does not call `gcc`. See Section 7.1 [STGCC/STG++: A wrapper for GCC], page 50 for the list of prohibited options.
- GNU awk: Make sure you have `gawk`.
- Postprocessing errors: `stgcc` calls `gcc` to generate assembly and postprocesses generated code. There may be errors (that I do not know of at this moment) in this process. When it happens, it displays a simple diagnostic message (that is useful at least for me :-) and aborts compilation. This is a bug of the postprocessor. When it happens, please send the assembly program before postprocessing and what did the postprocessor say. To obtain the assembly program before postprocessing, give `--dbg` flag to `stgcc`. It generates, among other things, `.t` file. Send it to me (see Section 7.1 [STGCC/STG++: A wrapper for GCC], page 50 for command line options of `stgcc`. See Section 1.1 [Reporting Bugs], page 1 for how to report bugs).

  The coding principle of the postprocessor is "never go a wrong way," meaning that it stops whenever it encounters something unfamiliar. It assumes a stereotyped code generation of `gcc` and aborts when it finds anything unexpected.

11.4 Link fails

Q: Link fails. Why?

A: First, see the previous item (See Section 11.3 [Compilation fails], page 83). In addition, `stgcc` invokes `nm` command at link time. Be sure that command invoked by typing `nm` displays contents of object files correctly.

11.5 How to obtain tools

Q: How to obtain necessary tools.

A: To use StackThreads/MP, you need `make`, `sh`, `nm`, `gawk`, and `gcc`. Virtually all UNIX platforms already have `make`, `sh`, and `nm`, although we recommend you to install GNU make. `gawk` is mandatory; `awk` cannot replace `gawk`. If you do not have `nm` or `nm` on your system doesn't work, you can install GNU nm. It is included in `binutils`.

For UNIXs, all these tools are available in a variety of sites. On Windows NT, all these tools are available as Cygnus GNUWIN32 package. If you have no idea, visit StackThreads/MP home page and follow links from there.

11.6 Why is compilation slow?

Q: Why does compilation take longer than `gcc`?

A: To be honest, the compilation is noticeably slower than plain `gcc` (as a rough estimation, `stgcc` takes twice longer than `gcc`). Probably, it comes mostly from the fact that the postprocessor is written in gawk. I want to retain it in awk for now because then it is very easy to fix bugs and add new platforms. I will eventually rewrite it for speed, possibly in compiled languages like C.

11.7 Segmentation fault

Q: When I ran a program compiled via `stgcc`, I got a segmentation fault. Why?

A: Ninty nine times out of hundred, it is not my fault :-) Here is some possible ways to isolate the problem.

- If the error occurs on a single processor, debug the program normally using GDB. If you suspect it is my bug, try to compile it using `gcc` if possible (by commenting out all StackThreads/MP primitives).
- When the error does not show up on a single processor but does on multiple processors, you are out of luck. The program is most likely to contain some race conditions.

- Some programs use a considerable amount of stack space when run on multiple processors. This happens when a program uses synchronizations between threads that are not in parent-child relationships. You can tell that your program caused a stack overflow by GDB. Run that program on GDB. If stack overflows, the program will stop at the opening brace of a procedure (This is most clearly seen when you run GDB from Emacs using `M-x gdb` command, because an arrow displays the location of the offending instruction).

  If this is the case, you can increase the stack size by giving `-ss` *S* option to your program, where S specifies a stack size in bytes (`-ss S` is automatically recognized by StackThreads/MP runtime. See Section 7.3 [Command Line Options Common for All StackThreads/MP Programs], page 53 for details). *S* can be a number or a number followed by a `k` (for Kbytes) or `m` (for Mbytes). For example, `-ss 4m` sets the stack size of each worker to 4Mbytes.

  Also See Section 3.6 [Stacks and Contexts], page 24 for examining stack size at various program points.

11.8 Why is my program slow on a single processor?

Q: When I run a program compiled by `stgcc` on a single processor, it is slower than its serial program compiled by `gcc`. Why?

A: Although StackThreads/MP tries to minimize the overhead of running parallel programs, there are several sources.

- `ST_THREAD_CREATE` adds something like ten instructions in addition to a normal procedure call.
- `ST_POLLING` costs something like ten instructions.
- Making your program runnable on multiprocessors, you must perform synchronization (join_counter, mutex, etc.). They roughly cost a procedure call even when they always succeed.

These overheads are insignificant when thread and synchronization granularity are large enough (several hundreds instructions between two events are large enough). Moreover, these overheads are fairly easy to reason about, because they all appear in the source program. There are other sources that are not directly visible in C/C++ sources.

- `stgcc` by default prohibits inlining. This may slowdown performance of C++ programs considerably. See Section 10.2 [Possible Slowdown by Disabling Inline], page 75 for details and workarounds.

- `stgcc` prohibits omitting frame pointers that are commonly adopted on Mips and Alpha. This may add a few instructions to a procedure call overhead.
- By inserting a few instructions at the end of each procedure, each procedure call now costs a few extra instructions at the epilogue.

The last two items are not noticeable unless the program calls a very small function (that perform, say, only 10 instructions in its body) too often. Such procedures are most likely to be inline-expanded if you attach `inline` keyword. This eliminates the procedure call overhead all together.

11.9 Why is my program slow on multiprocessors

Q: My program didn't speedup at all. Why?

A: I don't know :-) Again, it is not likely to be my fault, but here are some guidelines to shoot out the problem.

- A common mistake is that you forget to insert `ST_POLLING`. The program never speed up when you do that.
- Use profiler (on UNIX with X Window system and xgraph). It displays how much time each worker spends on which state (busy, idle, etc.) See Chapter 5 [Performance Profiler], page 43 for more information about profiler.

  If the graph is mostly `busy` (red), parallelization itself is good. If the graph contains a wide `steal` (green) area, there are something wrong in parallelization. One possibility is that you did not create enough threads in the green sections. If you are sure that there should be enough runnable threads there, check if you call `ST_POLLING` often enough.
- When your program does not speed up despite that the graph is mostly busy, before anything else, make sure you do not call the system's default malloc too often.

  In general, if the graph is mostly busy, and yet the program does not exhibit speedup, it implies that the total busy time increases as the number of processors increase. It in turn implies either that (1) your program performs extra work when run on multiple processors or (2) workers are invisibly descheduled by the OS. Note that when a worker is invisibly descheduled by, for example, a blocking system call, the event is not observed by the StackThreads/MP runtime system and hence counted as a busy time.

  When you are fairly sure that (1) is not the case, (2) is likely the case. Most frequently, such situations result when the program calls malloc often. Since some malloc implementations simply serialize all malloc request using mutex lock of the underlying thread library (such as

Pthreads), calling such mallocs within a parallel section is a bad practice. To our knowledge, Solaris malloc serializes all malloc calls. We provide several alternatives to this problem. See Chapter 4 [Memory Management], page 35 for more information.

You can tell how much does your busy time increase using profiler. When you run `mkxgrp` command, it displays a single line message that shows how much time did workers spend on each state. Each number is the total over all workers. You can run the same program with various numbers of workers (using `-nw` option) and see how does busy time increase. When (2) is the case, the typical behavior is that the busy time is roughly proportional to the number of processors.

11.10 What is ST_POLLING for?

Q: Why must I must insert ST_POLLING?

A: `ST_POLLING` does two things. One is to distribute threads among processors and the other is to manage stack. Without inserting `ST_POLLING` at all, your program never runs faster than on single processor. See Section 9.3 [Where should you insert ST˙POLLING?], page 66 for details.

11.11 Why both spin-locks and mutex?

Q: StackThreads/MP provides both spin-lock primitives and mutex locks. Why? And when should I use which?

A: Because both are useful on different situations. Mutex is useful when the critical section is not very short. When threads are contending on a mutex, these threads enter the waiting queue of the mutex and the workers schedule other runnable threads. The effect is to serialize computation within these critical sections and perform other tasks in parallel.

To achieve scalable speedup, you must be careful so that the total time spent on any critical section is far smaller than the ideal execution time on whatever number of processors you wish to scale. For example, if the program spends T seconds on a single processor and the total amount of time that must be spent in a critical section is X, you must make sure that $T/P << X$, where P is the maximum number of processors that you wish to run the program. Note that thread blocking takes a fairly long time, so the above X must account for blocking/unblocking threads on a mutex.

A common practice is to do your best to make critical sections as short as possible. A typical critical section is as short as 10 instructions. Spin-locks are useful for such short critical sections.

Table of Contents

1 Introduction 1

1.1 Reporting Bugs 1
1.2 Platforms 1
1.3 Installation 2

2 The Basics 6

2.1 An Example 6
2.2 Compiling 8
2.3 Running 9
2.4 Basic Primitives 10

3 APIs 14

3.1 Creating A Thread 14
3.2 Polling 15
3.3 Synchronization 15
3.3.1 Join Counter 16
3.3.2 Semaphore 17
3.3.3 Mutex 18
3.3.4 Condition Variable 18
3.4 Spin-Locks 19
3.4.1 Initialization, Lock, and Unlock 19
3.4.2 Reading and Checking Locations 22
3.4.3 Try Lock and Lock Any 22
3.4.4 Fetch and Add 23
3.5 Yield 23
3.6 Stacks and Contexts 24
3.6.1 Examine Current Stack Size 24
3.6.2 Show Stack Trace 24
3.6.3 Invoke Stack Management 25
3.6.4 Stack Pointers and Frame Pointers 25
3.7 Profiler 26
3.8 Timing 26
3.9 Workers and Worker Groups 27
3.9.1 Create A Group 27
3.9.2 Add a Slave Worker 28
3.9.3 Check Messages between Workers 28
3.9.4 Deschedule Worker 29

3.10 Callback 29
3.11 Graceful Exits 30
3.12 Thread Manipulation 30
3.12.1 Suspending A Thread 31
3.12.2 Resuming A Thread 32
3.12.3 Procedure Information 32
3.13 Thread and Worker ID 32

4 Memory Management 35

4.1 Parallel Conservative Garbage Collector (SGC) 35
4.1.1 GC_MALLOC and GC_FREE 36
4.1.2 Controlling GC behavior 36
4.2 Region-Based Memory Management 37
4.2.1 Region Basics 37
4.2.2 Creating A Region 38
4.3 How to Switch between Allocators 40
4.3.1 Changing the Underlying Allocator 40
4.3.2 Changing Region-based Allocator 41
4.4 Alloca is not supported (yet) 42

5 Performance Profiler 43

5.1 Profiling your program 43
5.2 Viewing profiled results 44

6 Cooperating with Sequential Modules 45

6.1 Calling Sequential Procedure from StackThreads/MP Procedure .. 45
6.2 Calling StackThreads/MP Procedure from Sequential Procedure .. 46
6.2.1 Create a Worker Group 46
6.2.2 Setup TLS Pointers 48

7 Command Summary 50

7.1 STGCC/STG++: A wrapper for GCC 50
7.2 STLINK: A Wrapper for Linker 52
7.3 Command Line Options Common for All StackThreads/MP Programs 53

8 Applying Safety Patches to GCC 55

8.1 How to Apply Patches 56
8.2 Using Patched GCC 56
8.3 Detailed Description of Patches 57

9 Advanced Topics 63

9.1 How does it work, basically? 63
9.2 A recommended programming style for performance 64
9.3 Where should you insert ST_POLLING? 66
9.4 Implementing Synchronization Primitives 69
9.5 What Does stgcc Command Do? 71
9.6 What Does the Postprocessor Do? 72

10 Known Limitations and Subtleties 73

10.1 Floating SP Problem 73
10.2 Possible Slowdown by Disabling Inline 75
10.3 Structure passing on SPARC 76
10.4 C++ Global Constructors 78
10.5 Alloca 79
10.6 Big arguments 79
10.7 Interaction with pragma interface 80

11 FAQ 82

11.1 Why am I interested in it? 82
11.2 Does it subsume Pthreads? 83
11.3 Compilation fails 83
11.4 Link fails 84
11.5 How to obtain tools 85
11.6 Why is compilation slow? 85
11.7 Segmentation fault 85
11.8 Why is my program slow on a single processor? 86
11.9 Why is my program slow on multiprocessors 87
11.10 What is ST_POLLING for? 88
11.11 Why both spin-locks and mutex? 88