Transparently Parameterizing Synchronization into a Concurrent Distributed Application

A Case Study of C++ Design Evolution

Douglas C. Schmidt
schmidt@cs.wustl.edu
Department of Computer Science
Washington University, St. Louis, 63130

An earlier version of this paper appeared in the July/August 1994 issue of the C++ Report.

1 Introduction

Many useful C++ classes have evolved incrementally by generalizing from solutions to practical problems that arise during system development. After the interface and implementation of a class have stabilized, however, this iterative process of class generalization is often de-emphasized. That is unfortunate since a major barrier to entry for newcomers to object-oriented design and C++ is (1) learning and internalizing the process of *how* to identify and describe classes and objects and (2) understanding when and how to apply (or not apply) C++ features such as templates, inheritance, dynamic binding, and overloading to simplify and generalize their programs.

In an effort to capture the dynamics of C++ class design evolution, the following article illustrates the process by which object-oriented techniques and C++ idioms were incrementally applied to solve a relatively small, yet surprisingly subtle problem. This problem arose during the development of a family of concurrent distributed applications that execute efficiently on uni-processor and multi-processor platforms. This article focuses on the steps involved in generalizing from existing code by using templates and overloading to transparently parameterize synchronization mechanisms into a concurrent application. Some of the infrastructure code is based on components in the freely available ADAPTIVE Service eXecutive (ASX) framework described in [1, 2, 3, 4, 5].

2 Motivation

The following C++ code illustrates part of the main event-loop of a typical distributed application (such as an object location broker [6] or a multi-threaded network file server):

Example 1

```
typedef unsigned long COUNTER;
COUNTER request_count; // At file scope

void *
run_svc (void *)
{
  Message_Block *mb;

  while (get_next_request (mb) > 0) {
    // Keep track of number of requests
    request_count++;

    // Identify request and
    // perform service processing here...
  }
  return 0;
}
```

This code waits for messages to arrive from clients, dequeues the messages from a message queue using `get_next_request`, and performs some type of processing (*e.g.,* database query, file update, etc.) depending on the type of message that is received.

This code works fine as long as `run_svc` runs in a single thread of control. However, incorrect results will occur on many multi-processor platforms when `run_svc` is executed simultaneously by multiple threads of control running on different CPUs. The problem here is that the auto-increment operation on the global variable `request_count` contains a race condition where different threads may increment obsolete versions of the `request_count` variable stored in their per-CPU data caches.

The remainder of this Section illustrates this phenomenon by executing the following C++ code example on a shared memory multi-processor running the SunOS 5.x operating system. SunOS 5.x is a version of UNIX that allows multiple threads of control to execute in parallel on a shared memory multi-processor [7]. This code is a greatly simplified version of the original distributed application.

```
// Manage a group of threads atomically
Thr_Manager thr_manager;

typedef unsigned long COUNTER;
COUNTER request_count; // At file scope

void *
run_svc (int iterations)
{
  Thr_Cntl t (&thr_manager);

  for (int i = 0; i < iterations; i++)
```

```
    request_count++; // Count # of requests

  return t.exit ((void *) i);
}

int
main (int argc, char *argv[])
{
  int n_threads  =
    argc > 1 ? atoi (argv[1]) : 4;
  int n_iterations =
    argc > 2 ? atoi (argv[2]) : 1000000;

  // Divide iterations evenly among threads
  int iterations = n_iterations / n_threads;

  // Spawn off N threads to run in parallel
  thr_manager.spawn_n (n_threads, &run_svc,
                       (void *) iterations,
                  THR_BOUND | THR_SUSPENDED);

  // Start executing all the threads together
  thr_manager.resume_all ();

  // Wait for all the threads to exit
  thr_manager.wait ();

  cout << n_iterations  << " = iterations\n"
       << request_count << " = request_count"
       << endl;
  return 0;
}
```

`Thr_Manager` is a class from the `ASX` framework. It contains a set of mechanisms for managing groups of threads that collaborate to implement collective actions (such as a pool of threads that render different portions of a large image in parallel). The `Thr_Manager::spawn_n` method creates n new threads of control. In the SunOS implementation of `Thr_Manager`, the `spawn_n` method calls the `thr_create` thread library routine to create a new thread. In this example, each newly created thread will execute the function `run_svc`, which iterates $\frac{n\_iterations}{n\_threads}$ times. Each thread is spawned using the `THR_BOUND` and `THR_SUSPENDED` flags. `THR_BOUND` indicates to the SunOS thread run-time library that each thread may run in parallel on a separate CPU in a multi-processor system. The `THR_SUSPENDED` flag creates each thread in the "suspended" state, which ensures that all threads are completely initialized before starting the tests with `Thr_Manager::resume_all`.

The `Thr_Manager::wait` method blocks the execution of the main thread until all the threads that are running `run_svc` have exited. When all the other threads have exited, the main thread prints out the total number of `iterations` and the final value of `request_count`.

Compiling this code into an executable `a.out` file and running it on 1 thread for 10,000,000 iterations produces the following:

```
% a.out 1 10000000
10000000 = iterations
10000000 = request_count
```

However, when executed on 4 threads for 10,000,000 iterations on a 4 CPU machine, the program prints:

```
% a.out 4 10000000
10000000 = iterations
5000000  = request_count
```

Clearly, something is wrong since the value of the global variable `request_count` is only one-half the total number of iterations! The problem here is that auto-increments on variable `request_count` are not being serialized properly. In particular, `run_svc` will produce incorrect results when executed in parallel on shared memory multi-processor platforms that do not provide *strong sequential order* cache consistency models. To enhance performance, many shared memory multi-processors employ "weakly-ordered" cache consistency semantics. For example, the SPARC V.8 and V.9 multi-processor family provides both *total store order* and *partial store order* memory cache consistency semantics. With total store order semantics, reading a variable that is being accessed by threads on different CPUs may not be serialized with simultaneous writes to the same variable by threads on other CPUs. Likewise, with partial store order semantics, writes may also not be serialized with other writes. In either case, expressions that require more than a single load and store of a memory location (such as `foo++` or `i = i - 10`) may produce inconsistent results due to cache latencies across CPUs. To ensure that reads and writes of variables shared between threads are updated correctly, programmers must manually enforce the order that changes to these variables become globally visible.

A common technique for enforcing a strong sequential order on a *total store order* or *partial store order* shared memory multi-processor is to protect the increment of the `request_count` variable by using some type of synchronization mechanism, such as a *mutex* (short for "mutual exclusion") [8]. Mutexes are used to protect the integrity of a shared resource that may be accessed concurrently by multiple threads of control. A mutex serializes the execution of multiple threads by defining a critical section where only one thread executes its code at a time.

One of the simplest and most efficient types of mutual exclusion mechanisms is a non-recursive mutex (this and other types of mutexes are discussed further in Section 4). SunOS 5.x implements non-recursive mutexes via the `mutex_t` data type and its corresponding `mutex_lock` and `mutex_unlock` functions. On SunOS 5.x, a thread may enter a critical section by invoking the `mutex_lock` function on a `mutex_t` variable. A call to this function will block until the thread that currently owns the lock has left the critical section. To leave a critical section, a thread invokes the `mutex_unlock` function on the same `mutex_t` variable. Calling `mutex_unlock` enables another thread that is blocked on the mutex to enter the critical section.

On SunOS 5.x, operations on mutex variables are implemented via adaptive spin-locks that ensure mutual exclusion by using an atomic hardware instruction. An adaptive spin-lock operates by polling a designated memory location using the atomic hardware instruction until (1) the value at this location is changed by the thread that currently owns the lock (signifying that the lock has been released by the previous owner and may now be acquired) or (2) the thread that is holding the lock goes to sleep (at which point the thread that is spinning also goes to sleep to avoid needless polling). On a multi-processor, the system overhead incurred by a spin-lock is relatively minor since polling affects only the local CPU cache of the thread that is spinning. A spin-lock is a simple and efficient synchronization mechanism for certain types of short-lived resource contention such as auto-incrementing the global `request_count` variable illustrated in the example above.

The following code illustrates how SunOS mutex variables may be used to solve the auto-increment serialization problem we observed earlier with `request_count`:

Example 2

```
typedef unsigned long COUNTER;
COUNTER request_count; // At file scope
mutex_t m; // mutex protecting request_count
           // initialized to zero...

void *
run_svc (void *)
{
  Thr_Cntl t (&thr_manager);

  for (int i = 0; i < iterations; i++) {
    mutex_lock (&m);
    request_count++; // Count # of requests
    mutex_unlock (&m);
  }

  return t.exit ((void *) i);
}
```

Although it solves the original synchronization problem, this approach is somewhat inelegant and error-prone since (1) it mixes C functions with C++ objects, (2) it leaves open the possibility that the programmer will forget to initialize the mutex variable,[1] or (3) forget to call `mutex_unlock`, and (4) it requires obtrusive changes to the code (in a larger system, managing these types of changes becomes a serious maintenance headache...).

3 C++ Solutions

C++ offers a number of language features that may be employed to solve the serialization problem more elegantly. This section illustrates a progression of C++ solutions, each one building upon insights from prior design iterations. As you read the examples, you might consider the point at which you would be satisfied with the solution and not contemplate any further enhancements.

3.1 An Initial C++ Solution

A somewhat more elegant solution to the original problem is to encapsulate the existing SunOS `mutex_t` operations with a C++ wrapper, as follows:

```
class Mutex
{
public:
  Mutex (void) {
    mutex_init (&this->lock, USYNC_THREAD, 0);
  }
  ~Mutex (void) {
    mutex_destroy (&this->lock);
  }
  int acquire (void) {
    return mutex_lock (&this->lock);
  }
  int release (void) {
    return mutex_unlock (&this->lock);
  }

private:
  // SunOS 5.x serialization mechanism
  mutex_t lock;
};
```

One advantage of defining a C++ wrapper interface to mutual exclusion mechanisms is that our code now becomes more portable across OS platforms. For example, the following code is an implementation of the Mutex class interface based on mechanisms in the Windows NT WIN32 API [9]:

```
class Mutex
{
public:
  Mutex (void) {
    InitializeCriticalSection (&this->lock);
  }
  ~Mutex (void) {
    DeleteCriticalSection (&this->lock);
  }
  int acquire (void) {
    EnterCriticalSection (&this->lock);
    return 0;
  }
  int release (void) {
    LeaveCriticalSection (&this->lock);
    return 0;
  }

private:
  // Win32 serialization mechanism
  CRITICAL_SECTION lock;
};
```

The use of the Mutex C++ wrapper class cleans up the original code somewhat and ensures that initialization occurs automatically when a `Mutex` object is defined, as shown in the code fragment below:

Example 3

```
typedef unsigned long COUNTER;
COUNTER request_count; // At file scope
```

[1] In SunOS 5.x, a zero'd `mutex_t` variable is considered to be implicitly initialized. However, other systems (such as Windows NT) do not make these guarantees, and all synchronization objects must be initialized explicitly.

```
Mutex   m; // mutex protecting request_count

void *
run_svc (void *)
{
  Thr_Cntl t (&thr_manager);

  for (int i = 0; i < iterations; i++) {
    m.acquire ();
    request_count++; // Count # of requests
    m.release ();
  }

  return t.exit ((void *) i);
}
```

However, the C++ wrapper approach does not solve the problem of forgetting to release the mutex (which still requires manual intervention by programmers) and it still requires obtrusive changes to the original source code.

3.2 Another C++ Solution

A straight-forward way to ensure the lock will be released is to leverage off the semantics of C++ class constructors and destructors to automate the acquisition and release of a mutex by supplying the following helper class for class `Mutex`:

```
class Guard
{
public:
  Guard (Mutex &m): lock (m) {
    this->lock.acquire ();
  }
  ~Guard (void) {
    this->lock.release ();
  }
private:
  Mutex &lock;
}
```

The `Guard` class defines a "block" of code over which a `Mutex` is acquired and then automatically released when the block is exited. It employs a C++ idiom (described in [10]) that uses the constructor of a `Guard` class to acquire the lock on the `Mutex` object automatically when an object of the class is created. Likewise, the `Guard` class destructor automatically unlocks the `Mutex` object when the object goes out of scope. By defining the `lock` data member as a reference to a `Mutex` object, we avoid the overhead of creating and destroying an underlying SunOS `mutex_t` variable every time the constructor and destructor of a `Guard` are executed.

By making a slight change to the code, we now guarantee that a `Mutex` is automatically acquired and released:

Example 4

```
void *
run_svc (void *)
{
  Thr_Cntl t (&thr_manager);

  for (int i = 0; i < iterations; i++) {
    {
      // Automatically acquire the mutex
      Guard monitor (m);
      request_count++;
      // Automatically release the mutex
    }
    // Remainder of service processing omitted
  }

  return t.exit ((void *) i);
}
```

However, this solution still has not fixed the problem with obtrusive changes to the code. Moreover, adding the extra '{' and '}' curly brace delimiter block around the `Guard` is inelegant and error-prone since a maintenance programmer might misunderstand the importance of the curly braces and remove them, yielding the following erroneous code:

```
  for (int i = 0; i < iterations; i++) {
    Guard monitor (m);
    request_count++;
    // Remainder of service processing omitted
  }
```

Unfortunately, this "curly-brace elision" has the side-effect of eliminating all concurrent execution within the system by serializing the main event-loop. Therefore, if computations may execute in parallel within that section of code, they will be serialized unnecessarily.

3.3 Yet Another C++ Solution

To solve the remaining problems in a transparent, unobtrusive, and efficient manner requires the use of two additional C++ features: parameterized types and operator overloading. We may use these features to provide a template class called `Atomic_Op`, a portion of which is shown below:

```
template <class TYPE>
class Atomic_Op
{
public:
  Atomic_Op (void) { this->count = 0; }
  Atomic_Op (TYPE c) { this->count = c; }
  TYPE operator++ (void) {
    Guard m (this->lock);
    return ++this->count;
  }
  TYPE operator== (const TYPE i) {
    Guard m (this->lock);
    return this->count == i;
  }
  void operator= (const Atomic_Op &ao) {
    // Check for identify to avoid deadlock!
    if (this != &ao) {
      Guard m (this->lock);
      this->count = ao.count;
    }
  }
  operator TYPE () {
    Guard m (this->lock);
    return this->count;
  }
```

```
  // Other arithmetic operations omitted...

private:
  Mutex lock;
  TYPE  count;
};
```

The Atomic_Op class transparently redefines the normal arithmetic operations (such as ++, --, +=, etc.) on built-in data types to make these operations work atomically. In general, any class that defines the basic arithmetic operators will work with the Atomic_Op class due to the “deferred instantiation” semantics of C++ templates.

Since the Atomic_Op class uses the mutual exclusion features of the Mutex class, arithmetic operations on objects of instantiated Atomic_Op classes now work *correctly* on a multi-processor. Moreover, C++ features such as templates and operator overloading allow this technique to work *transparently* on a multi-processor. In addition, all the method operations in Atomic_Op are defined as inline functions. Therefore, a highly optimizing C++ compiler should be able to generate code that ensures the run-time performance of this approach is no greater than using the mutex_lock and mutex_unlock function calls directly.

Using the Atomic_Op class, we can now write the following code, which is almost identical to the original non-thread safe code (in fact, only the typedef of COUNTER has changed):

Example 5

```
typedef Atomic_Op <unsigned long> COUNTER;
COUNTER request_count; // At file scope

void *
run_svc (void *)
{
  Thr_Cntl t (&thr_manager);

  for (int i = 0; i < iterations; i++) {
    // Actually calls Atomic_Op::operator++()
    request_count++;
  }

  return t.exit ((void *) i);
}
```

By combining the C++ constructor/destructor idiom for acquiring and releasing the Mutex automatically, together with the use of templates and overloading, we have produced a simple, yet expressive parameterized class abstraction that operates correctly and atomically on an infinite family of types that require atomic operations. For example, to provide the same thread-safe functionality for other arithmetic types we simply instantiate new objects of the Atomic_Op template class as follows:

```
Atomic_Op <double> atomic_double;
Atomic_Op <Complex> atomic_complex;
```

4 Extending Atomic_Op by Parameterizing the Type of Mutual Exclusion Mechanism

Although the design of the Atomic_Op and Guard classes described above yielded correct and transparently thread-safe programs, there is still room for improvement. In particular, note that the type of the Mutex data member is hard-coded into the Atomic_Op class. Since templates are available in C++, this design decision represents an unnecessary restriction that is easily overcome by parameterizing Guard and adding another type parameter to the template class Atomic_Op, as follows:

```
template <class MUTEX>
class Guard
{
// Basically the same as before...

private:
  MUTEX &lock; // new data member change
};

template <class MUTEX, class TYPE>
class Atomic_Op
{
  TYPE operator++ (void) {
    Guard<MUTEX> m (this->lock);
    return ++this->count;
  }
  // ...

private:
  TYPE  count;
  MUTEX lock; // new data member
};
```

Using this new class, we can make the following simple change at the beginning of the file:

```
typedef Atomic_Op <Mutex, unsigned long> COUNTER;
COUNTER request_count; // At file scope

// ... same as before
```

Before making this change, however, it is worthwhile to analyze the reasons *why* using templates to parameterize the type of mutual exclusion mechanism used by a program is beneficial. After all, just because templates exist does not necessarily make them useful in all circumstances. In fact, parameterizing and generalizing the problem space via templates without clear and sufficient reasons may increase the difficulty of understanding and reusing a class.

One motivation for parameterizing the type of mutual exclusion mechanism is to increase portability across OS platforms. Templates decouple the formal parameter class name “MUTEX” from the actual name of the class used to provide mutual exclusion. This is useful for platforms that already use the symbol Mutex to denote an existing type or function. By using templates, the Atomic_Op class source code would not require any changes when porting to such platforms.

However, a more interesting motivation arises from the observation that there are actually several different flavors of mutex semantics one might want to use (either in the same program or across a family of related programs). Each of these mutual exclusion flavors share the same basic protocol (*i.e.,* `acquire`/`release`), but they possess different serialization and performance properties. Five flavors of mutual exclusion mechanisms that I have found useful in practice are described below.

• **Non-Recursive Mutexes:** Non-recursive mutexes provide an efficient form of mutual exclusion. They define a critical section in which only a single thread may execute at a time. They are non-recursive since the thread that currently owns a mutex may not reacquire the mutex without releasing it first. Otherwise, deadlock will occur immediately. SunOS 5.x provides support for non-recursive mutexes via its `mutex_t` type. The `ASX` framework provides the `Mutex` C++ wrapper shown above to encapsulate the `mutex_t` semantics.

• **Readers/Writer Mutexes:** Readers/writer mutexes help to improve performance for situations where an object protected by the mutex is read far more often than it is written. Multiple threads may acquire the mutex simultaneously for reading, but only one thread may acquire the mutex for writing. SunOS 5.x provides support for readers/writer mutexes via its `rwlock_t` type. The `ASX` framework provides a C++ wrapper called `RW_Mutex` that encapsulates the `rwlock_t` semantics.

• **Recursive Mutexes:** Recursive mutexes are a simple extension to non-recursive mutexes. A recursive mutex allows calls to `acquire` to be nested as long as the thread that owns the `Mutex` is the one that re-acquires it. For example, if an `Atomic_Op` counter is called by multiple nested function calls within the same thread, a recursive mutex will prevent deadlock from occurring.

Recursive mutexes are particularly useful for callback-driven C++ frameworks [11, 3, 4], where the framework event-loop performs a callback to arbitrary user-defined code. Since the user-defined code may subsequently re-enter framework code via a method entry point, recursive mutexes may be necessary to prevent deadlock from occurring on locks held within the framework during the callback. The mutual exclusion mechanisms in the Windows NT WIN32 subsystem provide recursive mutex semantics.

The following C++ class implements recursive mutexes for SunOS 5.x, whose native mutex mechanisms do not provide recursive mutex semantics:[2]

```
class Recursive_Mutex
{
public:
  // Initialize a recursive mutex.
  Recursive_Mutex (const char *name = 0
                   void *arg = 0);

  // Implicitly release a recursive mutex.
  ~Recursive_Mutex (void);

  // Explicitly release a recursive mutex.
  int remove (void);

  // Acquire a recursive mutex (will increment
  // the nesting level and not deadmutex if
  // owner of the mutex calls this method more
  // than once).
  int acquire (void) const;

  // Conditionally acquire a recursive mutex
  // (i.e., won't block).
  int try_acquire (void) const;

  // Releases a recursive mutex (will not
  // release mutex until nesting level == 0).
  int release (void) const;

  thread_t get_thread_id (void);
  // Return the id of the thread that currently
  // owns the mutex.

  int get_nesting_level (void);
  // Return the nesting level of the recursion.
  // When a thread has acquired the mutex for the
  // first time, the nesting level == 1.  The nesting
  // level is incremented every time the thread
  // acquires the mutex recursively.

private:
  void set_nesting_level (int d);
  void set_thread_id (thread_t t);

  Mutex nesting_mutex_;
  // Guards the state of the nesting level
  // and thread id.

  Condition<Mutex> lock_available_;
  // This is the condition variable that actually
  // suspends other waiting threads until the
  // mutex is available.

  int nesting_level_;
  // Current nesting level of the recursion.

  thread_t owner_id_;
  // Current owner of the lock.
};
```

The following code illustrates the implementation of the methods in the `Recursive Mutex` class:

```
Recursive_Mutex::Recursive_Mutex
  (const char *name, void *arg)
  : nesting_level_ (0),
    owner_id_ (0),
    nesting_mutex (name, arg),
    lock_available_ (nesting_mutex_, name, arg)
{
}

// Acquire a recursive lock (will increment
// the nesting level and not deadlock if
// owner of lock calls method more than once).

int
Recursive_Mutex::acquire (void) const
{
  thread_t t_id = Thread::self ();

  Guard<Mutex> mon (nesting_mutex_);

  // If there's no contention, just
  // grab the lock immediately.
  if (nesting_level_ == 0)
    {
      set_thread_id (t_id);
      nesting_level_ = 1;
    }
  // If we already own the lock,
  // then increment the nesting level
```

[2] Note that POSIX Pthreads and Win32 provide recursive mutexes in their native thread libraries.

```
  // and proceed.
  else if (t_id == owner_id_)
    nesting_level_++;
  else
    {
      // Wait until the nesting level has dropped to
      // zero, at which point we can acquire the lock.
      while (nesting_level_ > 0)
        lock_available_.wait ();

      set_thread_id (t_id);
      nesting_level_ = 1;
    }

  return 0;
}

// Releases a recursive lock.

int
Recursive_Mutex::release (void) const
{
  thread_t t_id = Thread::self ();

  // Automatically acquire mutex.
  Guard<Mutex> mon (nesting_mutex_);

  nesting_level_--;
  if (nesting_level_ == 0)
    // Inform waiters that the lock is free.
    lock_available_.signal ();

  return 0;
}
```

• **Intra-Process vs. Inter-Process Mutexes:** To optimize performance, many operating systems provide different mutex mechanisms for serializing (1) threads that execute within the same process (*i.e.,* intra-process serialization) vs. (2) threads that execute in separate processes (*i.e.,* inter-process serialization). For example, in Windows NT, the `CriticalSection` operations define a mutual exclusion mechanism that is optimized to serialize threads within a single process. In contrast, the Windows NT mutex operations (*e.g.,* `CreateMutex`) define a more general, though less efficient, mechanism that allows threads in separate processes to serialize their actions. In SunOS 5.x, the USYNC_THREAD flag to the `mutex_init` function creates a mutex that is valid only within a single processes, whereas the USYNC_PROCESS flag creates a mutex that is valid in multiple processes. By combining C++ wrappers and templates, we can create a highly-portable, platform-independent mutual exclusion class interface that does not impose arbitrary syntactic constraints on our use of different synchronization mechanisms.

• **The Null Mutex:** There are also cases where mutual exclusion is simply not needed (*e.g.,* we may know that a particular program or service will *always* run in a single thread of control and/or will not contend with other threads for access to shared resources). In this case, it is useful to parameterize the `Atomic_Op` class with a "`Null_Mutex`." The `Null_Mutex` class in the `ASX` framework implements the `acquire` and `release` methods as "no-op" inline functions that may be removed completely by a compiler optimizer.

Often, selecting a mutual exclusion mechanism with the appropriate semantics depends on the context in which a class is being used. For instance, consider the following methods in a C++ search structure container class that maps external identifiers (such as network port numbers) onto internal identifiers (such as pointers to control blocks):

```
template <class EX_ID, class IN_ID, class MUTEX>
class Map_Manager
{
public:
  int bind (EX_ID ex_id, const IN_ID *in_id) {
    Guard<MUTEX> monitor (this->lock);
    // ...
  }

  int unbind (EX_ID ex_id) {
    Guard<MUTEX> monitor (this->lock);
    // ...
  }

  int find (EX_ID ex_id, IN_ID &in_id) {
    Guard<MUTEX> monitor (this->lock);

    if (this->locate_entry (ex_id, in_id)
      /* ex_id is successfully located */
      return 0;
    else
      return -1;
  }

private:
  MUTEX lock;
  // ...
};
```

One advantage to this approach is that the `Mutex` lock will be released regardless of which execution path exits a method. For example, `this->lock` is released properly if either arm of the `if/else` statement returns from the `find` method. In addition, this "constructor as resource acquisition" idiom also properly releases the lock if an exception is raised during processing in the definition of the `locate_entry` helper method. The reason for this is that the C++ exception handling mechanism is designed to call all necessary destructors upon exit from a block in which an exception is thrown. Note that had we written the definition of `find` using explicit calls to acquire and release the `Mutex`, *i.e.*:

```
  int find (EX_ID ex_id, IN_ID &in_id) {
    this->lock.acquire ();

    if (this->locate_entry (ex_id, in_id) {
    /* ex_id is successfully located */
      this->lock.release ();
      return 0;
    }
    else {
      this->lock.release ();
      return -1;
    }
  }
```

that not only would the `find` method logic have been more contorted, but there would be no guarantee that this->lock was released if an exception was thrown in the locate_entry method.

The type of MUTEX that the Map_Manager template class is instantiated with depends upon the particular structure of parallelism in the program code when it is used. For example, in some situations it is useful to be able to declare:

```
typedef Map_Manager <Addr, TCB, Mutex>
        MAP_MANAGER;
```

and have all calls to find, bind, and unbind automatically serialized. In other situations, it is useful to turn off synchronization without touching any existing library code by using the Null_Mutex class:

```
typedef Map_Manager <Addr, TCB, Null_Mutex>
        MAP_MANAGER;
```

In yet another situation, it may be the case that calls to find are *far* more frequent than bind or unbind. In this case, it may make sense to use the Readers/Writer Mutex:

```
typedef Map_Manager <Addr, TCB, RW_Mutex>
        MAP_MANAGER;
```

By using templates to parameterize the type of locking, little or no application code must change to accommodate new synchronization semantics.

5 Discussion

I frequently encounter several questions when discussing the use of templates in the Atomic_Op class. The first is "what is the run-time performance penalty for all the added abstraction?" The second is "aren't you obscuring the synchronization properties of the program by using templates and overloading?" The third question is "instead of templates, why not use inheritance and dynamic binding to emphasize uniform mutex interface and to share common code?" Several of these questions are related and I'll discuss my responses in this section.

The primary reason why templates are used for the Atomic_Op class involve efficiency. Once expanded by an optimizing C++ compiler during template instantiation, the additional amount of run-time overhead is minimal. In contrast, inheritance and dynamic binding often incur more overhead at run-time in order to dispatch virtual method calls.

Figure 1 illustrates the performance exhibited by the mutual exclusion techniques used in Examples 2 through 5 above.[3] This figure depicts the number of seconds required to process 10 million iterations, divided into 2.5 million iterations per-thread. The test examples were compiled using the -O4 optimization level of the Sun C++ 3.0.1 compiler. Each test was executed 10 times on an otherwise idle 4 CPU Sun SPARCserver 690MP. The results were averaged to reduce the amount of spurious variation (which proved to be insignificant).

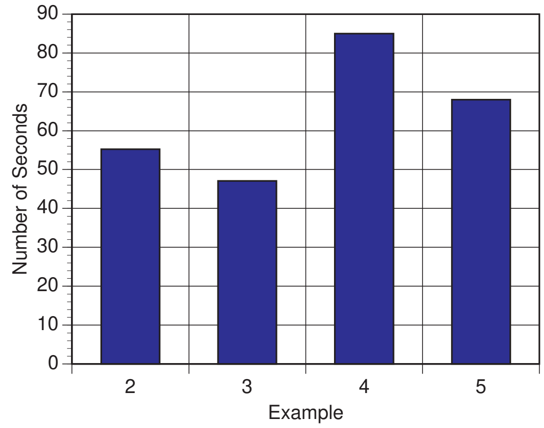

Figure 1: Number of Seconds Required to Process 10,000,000 Iterations

Example	*u*secs per operation	Ratio
Ex. 2	2.76	1
Ex. 3	2.35	0.85
Ex. 4	4.24	1.54
Ex. 5	3.39	1.29

Table 1: Serialization Time for Different Examples

Example 2 uses the SunOS mutex_t functions directly. Example 3 uses the C++ Mutex class wrapper interface. Surprisingly, this implementation consistently performed better than Example 2, which used direct calls to the underlying SunOS mutex functions. Example 4 uses the Guard helper class inside of a nested curly brace block to ensure that the Mutex is automatically released. This version required the most time to execute. Finally, Example 5 uses the Atomic_Op template class, which is only slightly less efficient than using the SunOS mutex functions directly. More aggressively optimizing C++ compilers would likely reduce the amount of variation in the results.

Table 1 indicates the number of micro-seconds (*u*secs) incurred by each mutual exclusion operation for Examples 2 through 5. Recall that each iteration requires 2 mutex operations (*i.e.,* one to acquire the lock and one to release the lock). Example 2 is used as the base-line value since it uses the underlying SunOS primitives directly. The third column of Examples 3 through 5 are normalized by dividing their values by Example 2.

An argument I have heard against using templates to parameterize synchronization is that it hides the mutual exclusion semantics of the program. However, whether this is a problem or not depends on how one believes that concurrency and synchronization should be integrated into a program. For class libraries that contain basic building-block components (such as the Map_Manager described above), allowing syn-

[3] Example 1 is the original erroneous implementation that did not use any mutual exclusion operations. Although it operates extremely efficiently (approximately 0.09 seconds to process 10,000,000 iterations), it produces results that are totally incorrect!

chronization semantics to be parameterized is often desirable since this enables developers to *precisely* control and specify the concurrency semantics that they want. The alternatives to this strategy are (1) don't use class libraries if multi-threading is used (which obviously limits functionality), (2) do all the locking outside the library (which may be inefficient or unsafe), or (3) hard-code the locking strategy into the library implementation (which is also inflexible and potentially inefficient). All these alternatives are antithetical to principles of reuse in object-oriented software systems.

An appropriate synchronization strategy for designing a class library depends on several factors. For example, certain library users may welcome simple interfaces that hide concurrency control mechanisms from view. In contrast, other library users may be willing to accept more complicated interfaces in return for additional control and increased efficiency. A layered approach to class library design may be quite useful to satisfy both groups of library users. In such an approach, the lowest layers of the class library would export most or all of the parameterized types as template arguments. The higher layers would provide reasonable default type values and provide an easier-to-use application developer's programming interface.

The new "default template argument" feature recently adopted by the ANSI C++ committee will facilitate the development of class libraries that satisfy both types of library users. This feature allows library developers to specify reasonable default types as arguments to template class and function definitions. For example, the following modification to template class `Atomic_Op` provides it with typical default template arguments:

```
template <class MUTEX = Mutex,
          class TYPE = unsigned long>
class Atomic_Op
{
// Same as before
};

// ...

#if defined (MT_SAFE)
// default is Mutex and unsigned long
Atomic_Op request_count;
#else /* don't serialize */
Atomic_Op<Null_Mutex> request_count;
#endif /* MT_SAFE */
```

Due to the complexity that arises from incorporating concurrency into applications, I've found the C++ template feature to be quite useful for reducing redundant development effort. However, as with any other language feature, it is possible to misuse templates and needlessly complicate a system's design and implementation. Currently, the heuristic I use to decide when to parameterize based on types is to keep track of when I'm about to duplicate existing code by only modifying the data types it uses. If I can think of another not-too-far-fetched scenario that would require me to make yet a third version that only differs according to the types involved, I typically generalize my original code to use templates.

6 Concluding Remarks

The example described in this paper was derived from a much larger distributed application that runs on a high-performance shared memory multi-processor. The `Atomic_Op` class and `Mutex`-related classes are some of the components available in the ADAPTIVE Communication Environment (ACE), which is a freely available object-oriented toolkit designed to simplify the development of distributed applications on shared memory multi-processor platforms [12]. ACE may be obtained via anonymous ftp from ics.uci.edu in the file `gnu/C++_wrappers.tar.Z` and `gnu/C++_wrappers_doc.tar.Z`. This distribution contains complete source code and documentation for the C++ components and examples described in this article. Components in ACE have been ported to both UNIX and Windows NT and are currently being used in a number of commercial products including the AT&T Q.port ATM signaling software product, the Ericsson EOS family of PBX monitoring applications, and the network management portion of the Motorola Iridium mobile communications system.

References

[1] D. C. Schmidt, "ACE: an Object-Oriented Framework for Developing Distributed Applications," in *Proceedings of the 6^{th} USENIX C++ Technical Conference*, (Cambridge, Massachusetts), USENIX Association, April 1994.

[2] D. C. Schmidt and P. Stephenson, "An Object-Oriented Framework for Developing Network Server Daemons," in *Proceedings of the 2^{nd} C++ World Conference*, (Dallas, Texas), SIGS, Oct. 1993.

[3] D. C. Schmidt, "The Reactor: An Object-Oriented Interface for Event-Driven UNIX I/O Multiplexing (Part 1 of 2)," *C++ Report*, vol. 5, February 1993.

[4] D. C. Schmidt, "The Object-Oriented Design and Implementation of the Reactor: A C++ Wrapper for UNIX I/O Multiplexing (Part 2 of 2)," *C++ Report*, vol. 5, September 1993.

[5] D. C. Schmidt, "IPC_SAP: An Object-Oriented Interface to Interprocess Communication Services," *C++ Report*, vol. 4, November/December 1992.

[6] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, 1.2 ed., 1993.

[7] J. Eykholt, S. Kleiman, S. Barton, R. Faulkner, A. Shivalingiah, M. Smith, D. Stein, J. Voll, M. Weeks, and D. Williams, "Beyond Multiprocessing... Multithreading the SunOS Kernel," in *Proceedings of the Summer USENIX Conference*, (San Antonio, Texas), June 1992.

[8] A. D. Birrell, "An Introduction to Programming with Threads," Tech. Rep. SRC-035, Digital Equipment Corporation, January 1989.

[9] H. Custer, *Inside Windows NT*. Redmond, Washington: Microsoft Press, 1993.

[10] G. Booch and M. Vilot, "Simplifying the Booch Components," *C++ Report*, vol. 5, June 1993.

[11] M. A. Linton and P. R. Calder, "The Design and Implementation of InterViews," in *Proceedings of the USENIX C++ Workshop*, November 1987.

[12] D. C. Schmidt, "The ADAPTIVE Communication Environment: An Object-Oriented Network Programming Toolkit for Developing Communication Software," in *Proceedings of the 12^{th} Annual Sun Users Group Conference*, (San Jose, CA), pp. 214–225, SUG, Dec. 1993.