OpenContainers: A Study of Portable Techniques for Thread-Heavy Applications

Richard T. Saunders
Rincon Research Corporation
IRAD Division
Tucson, Arizona 85711 USA
rts@rincon.com

ABSTRACT

The OpenContainers library is a open-source collection of portable C++ containers classes (arrays, hash tables, trees, etc.) written specifically to perform well in threaded applications that use hundreds of threads. OpenContainers explores the notion of collateral damage and thread-neutrality, employing a variety of techniques to allow the collection classes to perform well in the presence of many threads: lock elimination, lookaside caches, stack preference and allocation coalescing. All of these techniques are general purpose and can be applied to other thread-heavy applications. The techniques used to be implement the library are portable across shared-memory architectures (Alpha, Opteron, Xeon) and operating systems (Linux, Tru64). Real applications using the OpenContainers have been ported from early Shared-Memory Multiprocessors (SMP) to current hardware and scale well.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Scalability

General Terms

Thread, Portable

Keywords

thread neutrality, collateral damage, interference, lookaside cache, allocation coalescing

1. INTRODUCTION

As multi-core applications become more pervasive, the shared resources of such machines are become increasingly constrained. The shared memory bus/caches of a shared-memory multiprocessor (with multiple cores and multiple dies) are becoming an increasingly pressured bottleneck in modern applications as more processors share the bus. Techniques to relieve this pressure are important to acheieve scalability: without some attention to the problem, a multi-core machine can effectively become a single-core machine. Although hardware improvements are constantly being introduced to address the scalability problem[1], software techniques are also needed. This paper explores portable techniques in scalability via a C++ case study of the OpenContainers collection library.

Threadsafe container libraries are a critical tool for building multi-threaded applications in the modern multi-core world.[23][24][15][14] Arrays, hash tables, trees, strings, are typically important tools for building any software: combining them with threads can lead to problems in correctness and scalability[22]. *OpenContainers*, an open-source containers library, represents one solution for dealing with the issues of containers and concurrency. See the *Related Work* section for more discussion of different approaches.

The OpenContainers library is active and in current maintenance, although the distribution has been completely subsumed by the PicklingTools library: For more information, see http://www.picklingtools.com.

WHIST 2011 Tucson, Arizona

2. HISTORY

The OpenContainers containers classes were originally part of a Digital Signal Processing (DSP) framework called Midas 2k[5] that relied heavily on threads: many major applications utilizing hundreds of threads (i.e., *thread-heavy applications*), were successfully built using the Midas 2k framework[17]. But rather than the OpenContainers being written from scratch simply because they could, the collections classes of OpenContainers were written as a reaction to problems in scalability. Real world performance problems were the driving force in the development of the OpenContainers collection classes.

2.1 Midas 2k

Midas 2k applications rely heavily on hundreds of threads to do their job. Early in the development of Midas 2k, the applications scaled poorly in the presence of hundreds of threads. The collection classes of Midas 2k, originally the threaded RogueWave collection classes[19][20], were identified as the major bottleneck. The OpenContainers collection were born to replace the RogueWave classes (preserving the interface), thus allowing the Midas 2k applications to scale:

[1]we have seen the scalability of a program increase linearly with the speed of the frontside bus

The choice to use RogueWave libraries (and not C++ STL) was already deeply embedded in the framework by the time the framework was discovered to have scalabilty issues.

It is worth noting that, at the time of this rewrite (1997), the C++ STL still hadn't gained full acceptance. From Midas 2k's point of view, the C++ STL had been a non-starter because it was completely silent on the issue of threads[28] (until even recently, the C++ stance on thread guarantees has been very weak[2]). The RogueWave collections classes were originally chosen because they at least had a threading philosophy: lock all accesses to the containers for thread safety. Since threads were an instrinsic part of the Midas 2k framework, the RogueWave classes were the logical choice. The rationale (at the time) was that thread-level locking was much cheaper than process-level locking, so the RogueWave containers classes should give acceptable performance. Unfortunately, this was one of the major performance issues with the Roguewave classes.[2]

Through experimentation with several of the large Midas 2k applications, the OpenContainers utilized several techniques to achieve scalability.

3. COLLATERAL DAMAGE

Writing thread-safe code is usually a straight-forward proposition: shared resources are protected with a lock or similar mechanism before access[1][8]. This ensures that shared resources are always in a known state and updated atomically. The problem is that in systems involving hundreds of threads, any kind of locking mechanism has *collateral damage*: the unintentional consequence of affecting a different threads or processes (potentially limiting scalability).

1. *Serializing Access:* Many synchronization mechanisms work by serializing access (i.e., queueing, allowing access one at a time) to shared resources. The monitor, mutex and sempahore are typical examples[1][8]. Although this makes code thread-safe, by definition, it has the collateral damage of disallowing other threads to proceed in parallel. The collateral damage is the lack of opportunity for other threads: threads that could be running are idle in a queue.

   This may seem an easy trap to avoid, and in general it is with good design, but one typical culprit for resource serialization is the heap (aka dynamic memory pool, free storage, malloc pool, new pool). Any C++ application of any significance will use the heap extensively to allocate memory for all of its co-operating classes. But since there is essentially one heap, and many threads may be trying to allocate memory from the heap *at the same time,* access to the heap must be controlled. At the very least, there must be a lock around the malloc/new and free/delete calls (this was the solution of older systems, such as Digital Equipment Corporartions Tru64[4]). There are many much better techniques to allow the heap to work well with multiple threads[1][8][18], but in the end, some amount of locking[18], or a lock-free mechanism controlling access[9], has to occur: the heap, in a general purpose application is a shared resource and the requests for memory need to be serialized. This limits the amount of parallelism.

   There are techniques to allow threads to allocate memory without locks[9], such as per-thread heaps which don't require locking[18], but these are not general purpose techniques (as they tend to waste address space, which can be precious in an application with many threads).

2. *Locks and Cache Coherence:* The main mechanism for implementing mutexes and semaphores on modern shared memory architectures rely on synchronization primitives that require caches to be synchronized: in order for "atomic" access to be implemented correctly, memory between caches across CPUS must be synchronized.

   It depends on how the architecture implements these things, but locks are typically not cheap: many architectures require memory barriers[10] which are instructions enforcing memory-ordering constraints. These are specialized instructions which cause the machine to "wait" for memory to become coherent (different machines have different memory consistency protocols). The key here is that the system has to wait. And these memory barrier instructions usually have to be there to preserve correctness: These low-level constructs are typically hard to program correctly and optimizing them away is difficult[10].

   While the hardware enforces memory barriers and performs cache coherence protocols between processors, typically things are stalled. It may take on the order of 10-100 clock cycles to enforce the coherency and the locking. See [10]. During this time, not only is the CPU issuing the memory barrier stalled, but also the other processors who may to partipate in the cache coherence protocol are stalled. That stalling limiting the amount of parallelism.

   False sharing[16] is another type of collateral damage seen in multi-threaded applications. Although false sharing could easily fit under the *Locks and Cache Coherence* header, it tends to be different enough collateral damage that it could be classified specially. There are techniques to avoid false sharing[16], but the damage effects of false sharing are usually only pronounced in tightly-coupled threads.

One of the more frustrating things about collateral damage is that it tends to be more difficult to diagnose. This is the very nature of collateral damage, as the damage affects the running time of a *different* thread, not the one under consideration. Profiling tools tend to time the thread under consideration, not the effects. For example, *false sharing* has to be diagnosed by looking for excessive level one cache expulsions in a tool like Intel's *vtune*[6] or the open-source *oprofile*. Or you have to use special special tools to avoid creating any false sharing altogther[16]. There are other tools such as the open-source*valgrind* (via tooltype *drd*)[13], and the commenercial Intel Thread Checker that can be useful for finding collateral damage.

The key to making the OpenContainers scale was to avoid programming constructs that cause collateral damage.

[2]The author wants to emphasize that he had no control over the original architectural decisions of Midas 2k: RogueWave vs. C++ STL, locking architecture in the baseline, etc.: His job was to help fix it.

```
// Three container operations: length, contains, []
if (container.length() > 0 && container.contains("something")) {
    container["something"] = "newthing";
}
```

Figure 1: Typical HashTable Container Operations

4. THREAD NEUTRALITY

Thread neutrality is a simple concept: a thread running a piece of code should not impede the progress of another thread. If thread A could not run because it was waiting for the output of thread B, the code would not be thread-neutral. The notion of thread neutrality is a continuum, as everything a thread does affects the performance of another thread to some degree. In an ideal world, no thread could affect the performance of another thread. Unfortunately, issues with collateral damage (see above) all affect thread-neutrality.

Note that thread-neutrality implies a lack of collateral damage, although a lack of collateral damage does not necessarily imply thread-neutrality: some threads must wait on each other for results, even if they don't indirectly affect the others running times.

The thread-neutrality concept was coined in the early stages of Midas 2k: In test cases where multiple threads were running concurrently and completely independently (sharing no visible resources), it was surprising to see one running thread slowing the others. That pointed to some major flaw(s) somewhere in the Midas 2k architecture. In order for the Midas 2k to be a framework that worked well with hundreds of threads, it needed to promote *thread-neutrality* throughout. Part of achieving this thread neutrality was to avoid the things that caused the collateral damage mentioned in the previous section. It turns out the major flaws were frequently one of two things:

1. Pressure on the heap (as too many threads were requesting memory allocations at the same time)

2. Excessive locking (as things that didn't need to be locked were)

The concept of thread-neutrality is explored in depth in [16], although the paper doesn't actually use the term.

4.1 Approach

After identifying the fundamental classes (the collections classes) as the major impediment to scaling, the OpenContainers developed several techniques to achieve thread neutrality: eliminating locks, stack preference, lookaside caches and allocation coalescing. These techniques were developed in an attempt to make the fundamental classes of Midas 2k (the OpenContainers) thread-neutral so that the framework would scale in presence of hundreds of threads. Several thread-heavy applications already written in Midas 2k (namely GALACTUS and EARTHBOUND)[17] used the new OpenContainers as a testbed. The new techniques allowed the applications to scale well on the 4 CPU (Tru64 Alpha) shared-memory architectures, then later on 2-8 CPU (XEON Linux) shared-memory architectures. Although this approach has the danger of optimizing for a particular application, GALACTUS and EARTHBOUND were different enough applications (and ported to different architectures easily) that the results were general enough to give a strong confidence of the scalability of the techniques. Later experience with other very large applications written in Midas 2k (by other developers and other companies) cemented this belief.

The major *qualitative* test that thread-neutrality had been achieved (at least for the collections classes: the major applications GALACTUS and EARTHBOUND were a good testbed for confirming basic scalability) was simply allowing n threads on a *n-CPU* machine to run in parallel performing container actitivies: insertion, deletion, etc. in parallel. If the run-time of each of the n threads in that environment (with all sorts of potential collateral damage) was the same runtime as a single thread on an unloaded machine, we achieved success. It took all the techniques listed in the next sections to achieve this goal.

5. LOCK ELIMINATION

The first obvious problem with the RogueWave classes was simply that every operation on the classes locked a mutex. For several reasons, this degrades performance:

1. *Many operations on containers span multiple related operations:*

   Consider the hypothetical use of a hash table indexed by strings in Figure 1: The "lock" for the container would be held and released three times when, strictly speaking it really it should be locked once as part of a greater transaction. Sutter discusses this point further in [22].

   The argument can be made that if there's no contention for the lock, three locks/releases are trivial. However, the collateral damage of cache coherence protocol across caches can be surprisingly expensive. Most locks have to be implemented with memory barriers, forcing cache coherence protocols. A "trivial" lock is at least 10-100 cycles of clock time. See discussion above.

2. *The container locks may be completely unecessary:*

   The major building block of Midas 2k is the *component:* the component does an intrinsic Digital Signal Processing operation (filter, tune, decimate, Fast Fourier Transform, etc.). Each component is a implemented as a class that contains a thread and an associated lock for the data in the component. Most threads never leave their own component, and even if they do, the lock of the component protects the internals.

   In other words, the lock of the component can usually be used to protect all container accesses. Since most containers in Midas 2k are used for component helpers, the locks *in the containers* are not strictly necessary.

```
// OpalValue is a heterogenous container
OpalValue value = Opalize(int_u4(123));  // contains int_u4
value = Opalize(3.1415f);                // ... now contains float

// OpalTable is a recursive, key-value structure
OpalTable t;
t.put('key', value);                  // contains key-value pair
```

Figure 2: OpalValues and OpalTables in Midas 2k: Hetergeneous Containers

This may seem a trivial first step, but the fact the programming framework (Midas 2k) allowed the user to "think" in terms of sequential programming was a major step for easy-of-use for the users. The callback-based nature of the Midas 2k based framework was what enabled this simpler paradigm: the users were given their data through a callback without having to worry about the synchronization themselves. There is more discussion of this in the *Related Work* section.

It appears, at first blush, that rewriting the containers to get rid of the internal container locks (and using the component lock for synchronization) is all that is necessary. Unfortunately, more work is needed: the constructs of OpenContainers need to be revisited all the way to the core to achive thread-neutrality.

6. STACK PREFERENCE

Java's object model is a simplification of C++'s object model in many ways: this was one of the design goals of Java. For example, Java has no multiple inheritance, the syntax is cleaner, and all objects are heap-based. The last item is of particular interest: All objects in Java are (theoretically) allocated on the heap[3]. Consider, in Java:

```
// Allocates instance on heap:
// n is implemented as a pointer to the heap object
Someobj n = new Somobj;

// Empty object:
// empty is implemented as a NULL pointer
Someobj empty;
```

Objects in Java are always allocated on the heap. This means that *every* allocation has to pass through the heap, which increases the heap pressure, potentially causing collateral damage.

In C++, objects can be allocated on the stack.

```
// Allocated on stack
Someobj s;

// Allocated on heap
Someobj* sp = new Someobj;
```

Java avoids the stack-based object for several reasons: to avoid object slicing[11], to simplify garbage collection (objects on stacks have to be treated differently during garbage collection), and to simplify the language.

However, in a multi-threaded program, stack-based objects reduce collateral damage significantly: they promote data locality (preserving local caches and avoiding the shared memory bus), they reduce the pressure on the heap, and they reduce locking of the heap. Most memory allocations from the heap involve significant overhead: locking/unlocking as well as code to update a complex data structure[9][18] that manage freelists of memory. Stack-based allocation is much cheaper: typically *less than one machine add instruction* as all objects in a stack frame are allocated at once (of course, the objects themselves may allocate heap memory in their constructors, but the basic skeleton of the class can be allocated easily and quickly). And a stack allocation is lock-free and contention-free, as each thread has complete control over its own stack frame pointer. Stack-based allocation can completely eliminate locking and heap pressure if done correctly.

Inside of Midas 2k and the OpenContainers, users are "encouraged" to use stack-based classes. This is a pervasive philosophy, as all OpenContainers classes are written with stack preference usage in mind. The documentation for both Midas 2k and OpenContainers show stack-based examples, discouraging using the containers classes with "new". For example, see Figure 2. The Resouce Acquisition Is Intialization (RAII) idiom[22], a standard C++ idiom, encourages stack-based allocations as well.

As an example of stack-based encouragement in Midas 2k, consider the OpalValue and OpalTable of Midas 2k in Figure 2. The OpalValue is the generic heterogeneous container that can contain primitive types like ints, floats, and complexes as well as aggregate types like OpalTables. The OpalTable is a recursive, heterogeneous key-value hash table[5][17] very similar to the Python dictionary or the Perl hash. All thoughout M2k, its usage is encouraged via plain stack-based entities.

Using code like `OpalValue* ov = new OpalTable()` tends to be discouraged because abusing raw pointers tends to cause memory leaks (one of the very reasons Java doesn't have pointers), but also because the classes are written to be stack-based to promote thread-neutrality. See the discussion below on *Lookside Caches* for more details on the OpalValue.

7. LOOKASIDE CACHES

The lookaside cache is a small cache, as part of the data section of a class instance, for storing some data of the instance without having to go to the heap. Consider Figure 3, a very simple class to store an array of ints. If the array is small enough, it can be stored entirely within the class instance: the entire array and overhead can be allocated with a single allocation, either on the stack (see above) or in a single `new` call.

Lookaside caches are useful for avoiding collateral damage because they can reduce the number of allocations, reducing

[3]Optimizations by the Java compiler may do tricks, but the basic model is that the object is on the heap.

```
class ArrayOfInts {
   int* data_;  // points to heap data if array is larger
                //             than 32 elements
                //  ... points to lookaside_ cache otherwise.
   int lookaside_[32];
};
```

Figure 3: Simple Array of Ints Illustrating the Lookaside Cache Idea

```
class string {
      char* str;  // points to lookaside_ if under 32 bytes,
                  // ... otherwise points to heap
      char lookaside_[32];
};

// fits on stack, under 32 characters
string s("Some human-readable string");
```

Figure 4: String with Lookaside Cache

the heap pressure. Inside of Midas 2k and OpenContainers, there are three major places where we used the lookaside cache: the Midas 2k `OpalValue` class and the OpenContainers `string` and `Val` class.

7.1 String

The `string` class is an important part of the Midas 2k framework. Most interactions with the Midas 2k components are through the shell, which uses string-based processing to use and connect components. Messages sent back and forth between components are heavily string-based. Reporting messages to logs is string-based.

The original `string` class of Midas 2k was a reference-counted string, where the reference count was protected by a mutual exclusion lock (since it is a thread-based system). Naively, this appears to be an excellent implementation, as memory usage is reduced, and copying a string is simply a few instructions:

1. `lock ref-count mutex`
2. `increment reference-count (so as to "share" the string representation)`
3. `point to shared repr.`
4. `unlock ref-count mutex`

At first glance, this appears to be much faster than a memory copy of a string. There are, however, several issues with a reference-counted string in a threaded environment:

1. *Locks are not cheap:* As discussed earlier, the cost of locking a mutex is quite expensive (10-100 clock cyles), plus the collateral damage of invalidating caches across a shared-memory archicture.

2. *Many strings are small:* This is an observation that is applicable to Midas 2k, but of course, other applications may have different profiles. Many of the strings flowing in M2k (names, short messages) are very short, under 32 characters.

   Copying a small string can be a matter of 3 or 4 machine instructions, which is as fast or faster than a reference-counted scheme. More important is the scalabilty: there's no collateral damage from locking (because there is no locking) or cache coherence (as we shouldn't be doing explit sharing across CPUS: copying stays within the thread).

3. *Strings exercise the heap (dynamic memory):* The heap is a precious resource in a shared-memory, threaded application. Many C++ classes use "new" or "malloc" to get memory from the heap. With hundreds of threads potentially competing for memory simultaneously, the heap *has to be* thread-safe.

   Most Midas 2k applications have intense string usage, which causes extreme pressure on the heap.

The combination of all three of these problems causes string usage to be a major bottleneck in M2k applications. The fix:

1. *Don't use reference counting for strings:* Many strings in M2k can be copied cheaper than a reference-counted implementation because they tend to be shorter. This avoids all the collateral damage of excessive locking for string copies. Even if a straight-forward memory copy of a string is slower than a reference counted copy, it won't have the collateral damage effects.

2. *Use a look-aside cache for short strings:* This is frequently called the *small string optimization*. See [12] and [22]. Since most strings in M2k are short, they can be stored in a small cache inside the instance.

The major win with the "small string" is that many strings do not touch the heap at all. If small enough, the string fits perfectly on the stack, completely avoiding the heap and the collateral damage thereof. See Figure 4.

Although we independently discovered a lot of these issues in Midas 2k, [22] also outlines the problems with a reference-counted string in a threaded environment.

This one optimization was one of the the biggest wins in scalability for the Midas 2k system, as it drastically reduced heap pressure.

```
{
  "timestart"="2009:20:10::00", // OpalTable contains key-value pairs
  "data"=D:<1,2,3,4>,
  "axis"={
     0: "time",                 // ... values can be other OpalTables
  }
  "delta"=0.1
}
```

Figure 5: Typical OpalTable

7.2 OpalValue

As discussed earlier, the Midas 2k OpalValue is a heterogeneous container for a single entity, whether it's a primitive type (like strings, ints, floats, etc.) or a aggregate type (like an OpalTable or Vector). Note the Midas 2k OpalValue and OpenContainers Val class are very similar, so we limit our discussion to the OpalValues.

The original implementation of the Midas 2k OpalValue was type-tag and a void pointer. The actual data would be allocated from the heap, and the type-tag would identify what type of data stored through the pointer. This implementation was identified early on as a major bottleneck in early Midas 2k, because it caused tremendous heap pressure.

The OpalValue was replaced with a tag and a lookaside cache of 32 characters. All types that the OpalValue could contain would rewritten to fit within those 32 bytes: the primitive types all fit easily (the largest primitive type is a `complex_16`, which is 16 bytes: the ints and floats were smaller than this). The string and OpalTable were also optimized to fit within the 32 bytes (see string discussion above). Essentially, the OpalValue became a giant union.

Now, any OpalValue could be allocated with reduced heap pressure, as every type could fit in the lookaside cache without an extra allocation. This in turn reduced the collateral damage of an OpalValue significantly.

Any OpalValue allocated on the stack reduced the heap pressure to zero (unless it was a long string or an OpalTable, but see below for further reductions).

8. ALLOCATION COALESING

Some data structures, by their very nature, must use the heap—they are dynamic and must be allocated at run-time. In that case, they will exert heap pressure and cause collateral damage.

One main tool the OpenContainers uses to deal with very dynamic data structures is *allocation coalesing*. Rather than perform many small heap allocations, they perform fewer large allocations. The tradeoff here is that collateral damage is reduced (fewer allocations), but memory is wasted (and possibly fragmented) from over-allocation. Since the amount of memory wasted is usually small (at worst 476 bytes per OpalTable, but usually much less), this was seemed an acceptable cost to reduce collateral damage.

Note that using extra memory to avoid collateral damage seems to be a standard tradeoff: lookaside caches uses extra memory, coallesing allocations uses extra memory, per-thread allocators uses extra memory, false sharing padding uses extra memory. This major theme pervades the OpenContainers.

8.1 OpalTables and HashTables

One of the major data structures used by Midas 2k is the OpalTable (like Python dictionaries, see previous discussion). Most Midas 2k components pass OpalTables around as messages: they are the *currency* of Midas 2k. The key-value nature of the OpalTable makes it easy to pass dynamic, complex data around the system. An example OpalTable might look like Figure 5.

The keys of the OpalTables are strings, and the values of the OpalTables are OpalValues (which can in turn be other OpalTables). Note that the previous discussion optimized both strings and OpalValues to reduce collateral damage, but the OpalTable class itself can cause collateral damage.

The OpalTable is basically just a wrapper around the OpenContainers `HashTableT` class. The `HashTableT` is, by its very nature, quite dynamic. Since the OpalTable is the major currency of Midas 2k, it has to be adaptable to store very large or very small tables.

The `HashTableT` is implemented by dynamically allocating each key-value pair, putting it on a singly-linked list that is indexed by the hashvalue into an array: this is called *open-hashing* or *separate chaining* implementation of hash tables[27]. The link, key and value are all allocated in one chunk. To avoid excessive allocation, these chunks are allocated in groups of N, where the N is actually a compile-time template parameter so it can generate optimal code. This default for N is 8. The first time a key-value is inserted into the table, 8 chunks are allocated from the heap with one allocation: this completely eliminates the heap pressure for the next 7 insertions, as each insertion uses one of the remaining chunks. On the 9th insertion, the heap will have to be revisited.

Note that each individual table must have its own list of chunks: the chunks cannot be shared across tables because sharing across threads implies locks which implies collateral damage.

Most of the dynamic data structures within OpenContainers (`AVLHash`, `AVLTree`, `HashTableT`) use the allocation coalescing technique to achieve better thread-neutrality.

9. PUTTING IT ALL TOGETHER

All the fundamental classes of the OpenContainers (and M2k) were rewritten to avoid collateral damage.

Philosophically, the classes were written to encourage stack-based allocation.

From the top-level, all locks were removed from the containers classes to avoid excessive locking.

From the bottom-level, all classes were rewritten/reorganized to avoid/reduce heap pressure. The string uses lookaside caches to avoid heap pressure. The OpalValue (which

may contain strings) use lookaside caches to avoid heap pressure. The OpalTable (which contains strings and OpalValues) coalesces allocations to avoid heap pressure.

Consider the simple OpalTable operation inserting a string value into the table:

```
OpalTable table;
table.put(string("key"),
          OpalValue(string("value")));
// table["key"] = "value";
```

The original implementation operation would have caused up to three lock/unlocks and up to 10 memory allocations (new OpalValue, OpalValue, string, OpalValue, string, table, string, string, OpalValue). Now it has been reduced to 0 locks and at worst 3 memory allocations (string, string, hash entry). In this case, the strings fit within the lookaside caches, reducing the memory allocations to 1. And if the OpalTable has already seen a few insertions, the heap activity is reduced to zero, potentially eliminating all collateral damage.

The climate in which Midas 2k was developed was the late 1990s when most developers and vendors were struggling with multi-threaded programs and abstractions. Granted, memory allocators (which contributed to a lot of thread-neutrality problems of M2k) have come a long way since that time[9]. But the fact remains that even the best memory allocators can easily become a bottlenecks in a system as the number of threads increases. As the number of processors on multicore chips increases, the number of threads in modern applications will likely increase as programmers struggle to get the performance from threads[23]. The portable techniques outlined here in the OpenContainers work continues to be relevent, as they show how to mitigate the effects of memory pressure.

10. RELATED WORK

There are multiple aspects to building thread-safe containers:

1. *internal vs. external locking:* Are mutexes locked by user code outside the container (external) or by the container itself inside the container code (internal)?

2. *loosely-coupled vs. tightly-coupled threads:* Are threads managed by the user outside of the containers (loosely-coupled) or so the data structures directly managed the threads (tightly-coupled)?

3. *amount of thread-neutrality:* Do the data structures of the container interfere with other threads?

The simplest approach is to use a single threaded collections library and have the programmer use external locks around all accesses to the containers collection. This will basically work as long the classes lack global state (i.e., no globals). This is the model of the C++ STL. Until the recent C++ standard (C++ 0x)[3], however, C++ was silent on the issue of threads, so the simplest assmuption of "no global state" wasn't even clear. The GNU C++ package, however, is a de-facto standard and is packaged with the SGI STL, which has well-defined thread assumptions[21]. Note that in this model, threads are loosely-coupled to the containers: the threads are external entities and not part of the containers. As long as the user has to lock manually, this is a clumsy and error-prone technique for thread safety.

A more traditional approach, used by Roguewave[19][20] containers, some Intel Threading Building Blocks (TBB)[15] containers, some STAPL containers[26][25][24], and MTL[14] is to have a lock built into the class so that each instance has its own lock to protect the instance data. The Java `synchronized` methods provide a related mechanism. In this model, threads are still loosely-coupled: they are not part of the data. The data structures lock themselves, but depend on the user to provide threads to operate on the data. This model also suffers from the "Many operations on containers span multiple related operations" problem where the same lock will be locked/unlocked multiple time, when one lock would do.

Parts of STAPL libraries[24] and the Intel Threading Building Blocks (TBB)[15] take a more tightly-coupled approach to the containers collection: the thread/distributed unit is more tightly bound to the collection and the data inside. For example, in the Parallel Container Framework (PCF)[26] of STAPL, the programmer sets-up data inside the container for distribution. In this model, the container itself helps manage both the concurrency and threads so that user can concentrate on writing the algorithm. Although the algorithms (*i.e.* `pAlgorithms` in STAPL) are actually separate entities, they co-operate with the locks of the containers classes, doing things such as *method forwarding*[24]. Tightly-coupled threads with internal locks tend to force the user into a programming abstraction, but usually with a simpler programming model.

The OpenContainers spans all three major models, but with an emphasis on the core: The core of OpenContainers is an externally locked, loosely-coupled container library, but unlike other containers libraries, it is thread-neutral: it suffers no performance penalties for being single-threaded or multi-threaded. Furthermore, the simple techniques of the OpenContainers tend to preserve locality of caches, decreasing memory bus/caches pressure. The OpenContainers acheives this by avoiding collateral damage. The techniques developed by the OpenContainers library allow the program to stay "out of the way" of other cores on the bus.

There has been a lot of work in the newest versions of C++ to make the libraries behave well in threaded environments[28], most of the work based upon the Boost libraries[7]. Unfortunatelty, besides the C++ standards[2][3], documentation is sparse: [28] (the only book available at the time of this writing) has be in pre-order for a year.

The STAPL and TBB libraries represent an agressive tact of splitting data (`pRanges`) and the algorithm (`pAlgorithm` of STAPL and `pipeline` of TBB). The containers are written to work actively with multiple threads, but the single threaded implementation may suffer. The user, however, is given an abstraction where the locks and threads are handled more easily (internal locks with tightly-coupled threads). Although not the emphasis of this paper, the OpenContainers also has overlapping abstractions with the TBB: a pipeline and a concurrent queue. TBB has a `parallel_for` which is similar to the barrier synchronizer in OpenContainers.

Although the core OpenContainers requires external locking and a user-defined abstraction for threads, this is not necessarily a negative: many frameworks (Midas 2k, Qt, any callback-based system) present the abstraction of single thr-

eaded programming and the framework automatically locks necessary structures. For example, in Midas 2k, the locks of the higher-level *component* lock needed data structures, allowing the user (in a callback) to write what amounts to single-threaded code. Thus, the OpenContainers work well in that model, as they suffer little or no performance penalties for multiple cores.

Perhaps most important fact is that the core OpenContainers are thread-neutral: in both the single and multi-threaded scenario, the use of the OpenContainers will not cause any performance slowdowns due to collateral damage. In other words, the core OpenContainers will not introduce any hard to trace performance bottlenecks.

Historically, the issues for OpenContainers were simple: Midas 2k needed a library that was portable, had the same interface as the RogueWave libraries, were loosely-coupled libraries (like the Roguewave), and had the same performance characteristics across multiple architectures (Alpha True64 4-CPU machines, Linux Xeon dual CPUs). The C++ STL libraries still tend to have different implementations across different operating systems, as each compiler provides its own version STL. The OpenContainers library is portable and has been used on multiple architectures, with the same implementation and guarantees.

11. CONCLUSION

The OpenContainers collection represents a series of techniques for helping thread-heavy applications scale. There are several applications (using hundreds of threads) in use today that use the OpenContainers as their main containers collection: GALACTUS and EARTHBOUND are two such examples[17]. These applications were written in the late 1990s and have been ported through many different versions of Tru64 UNIX and Linux (RedHat, Fedora), using the newer multicore machines of today. These applications have ported to the new architectures, retaining their scalability under the multicore architectures. This demonstrates that the techniques continue to be portable and scalable, even in today's multicore rich environment.

Currently, the heaviest user of the OpenContainers is the PicklingTools library[17], a collection of C++/Python code allowing users to build hybrid applications easily, where C++ is used for performance-critical and multi-threaded code, Python for simpler control-flow code.

Although the techniques used here were used for a very particular containers library, all the techniques are general purpose and portable and should be applicable to any thread-heavy applications.

12. REFERENCES

[1] D. R. Butenhof. *Programming with POSIX threads*. Addison-Wesley, Boston, MA, 1997.

[2] C. S. Committee. *C++ International Standard ISO/IEC 14882, First Edition 1998-09-01*. ISO/IEC, New York, NY, 1998.

[3] C. S. Committee. *ISO/ISC DTR 19769 (February 28, 2011) Working Draft, Standard for Programming Language C++*. ISO/IEC, New York, NY, 2011.

[4] D. E. Corporation. *Digital Unix Guide to DECthreads*. Digital Equipment Corporation, Maynard, MA, 1997.

[5] S. F. Derek Jones. *Midas 2k Documentation*. Rincon Reserach Corporation, Tucson AZ, 1996-2000.

[6] R. Gerber and A. Binstock. *Programming with Hyper-Threading Technology*. Intel Press, 2004.

[7] B. Karlsson. *Beyond the C++ Standard Library: An Introduction to Boost*. Addison-Wesley, Boston MA, 2005.

[8] M. Maged. High performance dynamic lock-free hash tables and list-based sets. In *The 14th Annual ACM Symposium on Parrallel Algorithms and Architectures*, pages 73–82. ACM, 2002.

[9] M. Maged. Scalable lock-free dynamic memory allocation. In *The 2004 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 35–46. ACM, 2004.

[10] P. E. McKenney. Memory barriers: a hardware view for software hackers. In *Linux Technology Center*, pages 1–29. IBM, 2002.

[11] S. Meyers. *More Effective C++*. Addison-Wesley, Boston MA, 1996.

[12] S. Meyers. *Effective STL*. Addison-Wesley, Boston MA, 2001.

[13] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation*, 2007.

[14] D. W. P. Gottschling and M. Adams. Representation-transparent matrix algorithms with scalable performance. In *Proceeedings from Internal Conference on SuperComputing*, 2007.

[15] J. Reinders. *Intel Threading Building Blocks*. O'Reilly & Associates, Inc., Sebastopol, CA, 2007.

[16] R. T. Saunders. A portable framework for high-speed parallel producer/consumers on real cmp, smt and smp architectures. In *Parallel and Distributed Processing Symposium, IPDPS 2007, IEEE International*, pages 1–8. ACM, 2007.

[17] R. T. Saunders. Complex software systems in legacy and modern environments: A case study of the picklingtools library. In *New Application Areas in Open Source Software (NAOSS) Minitrack at 43th Hawaii International Conference of System Sciences*, 2010.

[18] D. S. N. Scott Scheider, C Christos D. Antonopoulos. Scalable locality-concious multithreaded memory allocation. In *Proceedings of the 5th international symposium on Memory management*, pages 84–94. ACM, 2006.

[19] R. Software. *Tools.h++: Foundation Class Library for C++ Programming, Class Reference, Version 7*. Rogue Software, Corvallis, Oregon, 1996.

[20] R. Software. *Tools.h++: Foundation Class Library for C++ Programming, User's Guide, Version 7*. Rogue Software, Corvallis, Oregon, 1996.

[21] R. Stallman. Thread-safety for sgi stl. In *The GNU C++ Library Manual*, pages 277–288. OSF, 2011.

[22] H. Sutter. *More Exceptional C++*. Addison-Wesley, Boston MA, 2002.

[23] H. Sutter. A fundamental turn toward concurrency in software. *Dr. Dobbs Journal*, 30(3), Mar 2005.

[24] G. Tanase, A. Buss, A. Fidel, Harshvardhan, I. Papadopoulos, O. Pearce, T. Smith, N. Thomas,

X. Xu, N. Mourad, J. Vu, M. Bianco, N. M. Amato, and L. Rauchwerger. The stapl parallel container framework. In *Proceedings on eleventh ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*. ACM, 2011.

[25] G. Tanase, C. Raman, M. Bianco, N. M. Amato, and L. Rauchwerger. *Languages and Compilers for Parallel Computing: Associative Parallel Containers in STAPL*. Springer-Verlag, Berlin, Heidelberg, 2007.

[26] N. Thomas, G. Tanase, O. Tkachyshyn, J. Perdue, N. Amato, and L. Rauchwerger. A framework for adaptive algorithm selection in stapl. In *Proceedings on the tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 277–288. ACM, 2005.

[27] M. A. Weiss. *Data Structures and Algorithm Analysis in C*. Benjamin/Cummings, Redwood City, CA, 1993.

[28] A. Williams. *C++ Concurrency in Action*. Manning, Corvallis, Oregon, 2009.