Princeton Plasma Physics Laboratory

PPPL-5046

PPPL-5046

NSTX-U Advances in Real-time C++11 on Linux

Keith G. Erickson

MAY 2014

Prepared for the U.S. Department of Energy under Contract DE-AC02-09CH11466.

Princeton Plasma Physics Laboratory
Report Disclaimers

Full Legal Disclaimer

Trademark Disclaimer

PPPL Report Availability

Princeton Plasma Physics Laboratory:

http://www.pppl.gov/techreports.cfm

Office of Scientific and Technical Information (OSTI):

http://www.osti.gov/scitech/

Related Links:

U.S. Department of Energy

Office of Scientific and Technical Information

NSTX-U Advances in Real-time C++11 on Linux

Keith G. Erickson, *Princeton University*

***Abstract*—Programming languages like C and Ada combined with proprietary embedded operating systems have dominated the real-time application space for decades. The new C++11 standard includes native, language-level support for concurrency, a required feature for any nontrivial event-oriented real-time software. Threads, Locks, and Atomics now exist to provide the necessary tools to build the structures that make up the foundation of a complex real-time system. The National Spherical Torus Experiment Upgrade (NSTX-U) at the Princeton Plasma Physics Laboratory (PPPL) is breaking new ground with the language as applied to the needs of fusion devices. A new Digital Coil Protection System (DCPS) will serve as the main protection mechanism for the magnetic coils, and it is written entirely in C++11 running on Concurrent Computer Corporation's real-time operating system, RedHawk Linux. It runs over 600 algorithms in a 5 kHz control loop that determine whether or not to shut down operations before physical damage occurs. To accomplish this, NSTX-U engineers developed software tools that do not currently exist elsewhere, including real-time atomic synchronization, real-time containers, and a real-time logging framework. Together with a recent (and carefully configured) version of the GCC compiler, these tools enable data acquisition, processing, and output using a conventional operating system to meet a hard real-time deadline (that is, missing one periodic is a failure) of 200 microseconds.**

Index Terms—**Computer languages, Real-time systems, Software design**

I. Introduction

C++11 introduced new paradigms to the language [1] that will benefit the real-time (RT) software world given careful usage. Previous versions of the language relied on third party libraries often written in C (e.g. pthreads) or non-portable operating system specific routines to provide features like threading, synchronization, and communication. The new version moves those requirements into the language itself, requiring that compliant compilers provide the necessary features. Thus, concurrent code can now become portable, reusable code.

This manuscript has been authored by Princeton University under Contract Number DE-AC02-09CH11466 with the U.S. Department of Energy. The United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this manuscript, or allow others to do so, for United States Government purposes.

K. G. Erickson is with the Princeton Univeristy Plasma Physics Lab, Princeton, NJ 08540 (e-mail: kerickso@pppl.gov).

But concurrent code is not the same as RT code. Concurrency refers to situations where multiple threads execute in parallel, possibly with shared data structures. Real-time instead describes the timing requirements and, specifically, the determinism of the code, regardless of how many threads are running. For instance, the new National Spherical Torus Experiment Upgrade (NSTX-U) system will operate at 5 kHz, one cycle every 200 µs [4]. Since missing one cycle is a failure, the system is *hard* real-time.

Making the new C++ concurrency features also support hard real-time is, ironically, *hard*. By design, most of the language components trade latency for throughput. The NSTX-U system, like most RT systems, instead prefers the lowest reachable latency to reduce the risk of failing to meet hard deadlines. The total work possible is therefore less than that achievable on a non-real-time system [7]. The implication here then is that facilities that would otherwise work well in a non-RT application may become harmful.

A. Reference Platform

Because the NSTX-U Digital Coil Protection System (DCPS) is the primary customer for the ideas described in later sections, its platform and associated characteristics will provide the backdrop for any implementation specific details such as timing metrics or compiler feature availability. It is a 32-core AMD Opteron 6386SE 64-bit system with 64 GB of registered ECC memory running on SuperMicro hardware with a Concurrent RedHawk GNU/Linux operating system. It runs custom software written entirely in C++11 whose purpose is to protect the NSTX-U coil system. It executes 600 algorithms [5] every 200 µs, computing various stresses, forces, and combinations thereof on the machine. Should any algorithm trip any of the predetermined upper or lower limit values, the entire machine will instantly shut down. [6][7]

B. GNU/Linux, RedHawk Kernel

The NSTX-U DCPS test platform uses Concurrent RedHawk as the real-time version of Linux [8]. It is based on RedHat Enterprise Linux (RHEL) version 6, amplified with a modified, recent kernel and a comprehensive tool suite specifically engineered for real-time development and deployment. The tools include a debugger with code injection support, simulator, system tuner, kernel tracer, and more. The advanced kernel delivers superior performance with respect to scheduling and latency vs. a stock kernel. RedHawk provides a simple method of shielding CPU cores from any combination of interrupts, other processes, and the kernel

timer. This gives user applications exclusive use of a given set of cores, guaranteeing that core contention stays within the application itself. Later sections will continually describe ways to exploit this feature with the gratuitous use of spinwaits.

C. GNU Compiler Collection (GCC) [9]

C++11 is a recent language. Unfortunately, the base operating system, RHEL6, uses an old version of the compiler (4.4) that does not support anything past C++03. DCPS instead uses a manually built version of a stock GCC v4.8.3 [10] upstream release. It is configured with only one important option described later, whose purpose is to streamline certain timing system calls. With the recent release of GCC v4.9.0 [11], plans are in place to upgrade to the new version, which should allow the usage of two features missing in 4.8: regular expressions, and generic lambda functions. The latter feature should simplify some of the proposed implementations below.

D. Scheduling and Priority

Typical Linux scheduling involves modifying a process "nice" value. A low value reduces the allotted share of the CPU time, while a high value increases the share. Any user can reduce the share, while only the super user can increase it. This is know as the Time Share scheduling policy, and is the de facto standard to manage CPU time.

However, Linux actually supports three scheduling policies for user processes: Time Share (TS), First In First Out (FIFO, though the kernel uses the designation "FF"), and Round Robin (RR). The first one as stated runs virtually every process. The latter two are the RT policies. They are both preemptive, in that they will force a context switch with a lower priority thread. They differ with respect to threads within the same priority level. The FIFO scheduling policy will cause the scheduler to block all other threads at the same priority until the running thread yields (either manually or via a system call). The RR policy will do the same, up to a maximum time slice determined by the thread's "nice" level. If a thread exceeds its time slice, the scheduler will place it at the back of the run queue for that priority level. Since the time slice sizes range from 10 ms to 780 ms on the reference platform, the RR scheduler is not an option for the remainder of this discussion; it is here only for completeness. Consequently, every thread in DCPS uses the FIFO scheduling policy.

Priorities are rather simple in the Linux system. They range from 0 to 99. 0 is reserved for the TS policy, and 1 through 99 are for the RR and FF policies. Arbitration between threads at different levels is simple: for any threads on a CPU that are currently runnable, the thread with the highest priority runs.

II. Key C++ Concepts

Object oriented programming techniques apply to virtually all object oriented languages. However, there are certain nuances specific to writing C++ code that affect the implementation or even practical application of generic concepts and approaches.

A. Resource Acquisition Is Initialization (RAII)

In C++, the constructor and destructor approach to class design allows for clear distinction of object and resource lifetime. In general, an object should acquire all resources in the constructor, and release all resources in the destructor. This is a very safe design idiom, since the language guarantees that the destructor always executes, even in the face of exceptions. This is markedly different from other languages that instead encourage the use of explicit "getters" and "setters" to tell the object instance when to conduct acquire and release operations.

In the world of C or C-like C++ code, the typical approach is to acquire resources, test for some error condition, and free the resources after handling the error. C++ makes this much easier. Consider these two scenarios:

```
// Manual management
int a = new int[10];
try {
  f(a);
} catch (...) {
  std::cout << "Error" << std::endl;
}
delete[] a;
```

vs:

```
// Automatic management
std::vector<int> a(10);
f(a);
```

In the first example, the user must take care of handling all possible errors to avoid leaking memory. In the second, the object allocates memory in its constructor and deletes the memory in its destructor, all transparently to the user. Good class design follows this paradigm or variations thereof whenever possible. Sections that follow will use this extensively to show simple ways to lock and unlock protected regions.

B. Memory Allocation In Real Time

As shown in the previous section, RAII provides a powerful way to ensure safe resource management. The drawback is that sometimes the objects in use may unexpectedly do things that affect real-time applications. In the example above, the vector started with enough space pre-allocated for ten elements large enough to hold an integer. This will have no impact on a real-time scenario, until the vector runs out of space.

The vector implementation that comes with GCC has a greedy allocator. When it runs out of space, it doubles the allocated size. That means that if a user pushes an eleventh element into the vector above, it will grab ten additional elements from the free store. That allocation can be very time consuming, especially in the case of large vectors. Consider a vector holding one million 8-byte double precision floating-point numbers. Adding one more element to that vector will cause a memory allocation of 8 MB. With 4 KB default page

sizes, that will entail substantial page faulting.

Page faults introduce another potential for real-time disaster. The obvious solution to the above problem is to always overcommit allocated memory. In the case of DCPS, each real-time event (called a "test shot") cycles at 5,000 times per second for 11 seconds total, yielding 55,000 cycles. Any buffer used during the test shot therefore requires a minimum of 55,000 elements. DCPS overcommits this to 100,000 elements because, frankly, memory is inexpensive.

Obvious solutions sometimes have unobvious problems. The Linux memory allocator is a lazy allocator, and it will not actually fault in the allocated pages. Page faults occur on first use instead of on first allocation. Since each page fault takes a minimum of 2 ms to complete, the result is catastrophic. A single page fault blocking the RT execution path will cause immediate shutdown of the NSTX-U system.

Fortunately, there is an easy solution. Borrowing from the C world, DCPS makes use of memory locking [12] to instruct the operating system to lock all current and future pages of allocated memory. This not only prevents swapping to disk, but forces page faults at first allocation instead of on first use.

The one thing to keep in mind with all of the solutions posed in this section is that each one increases determinism by reducing efficient memory usage. Said inversely, each solution wastes memory. DCPS accounts for this with a very large memory footprint at surprisingly little cost. [6]

III. New C++ Version: 2011

In September of 2011, the International Organization for Standardization (ISO) released a new version of the C++ programming language standard as ISO/IEC 14882:2011, or colloquially as C++11 [1]. This replaced the previous C++03 (ISO/IEC 14882:2003) [2] and C++98 (ISO/IEC 14882:1998) [3] versions. The next language versions are tentatively named C++14 and C++17, and these will decouple the language standard from upcoming experimental technical specifications to encourage shorter timespans between released versions.

The 2011 version of the language added several new capabilities that are critical for the advancements present in DCPS. These include both library changes as well as base language changes.

A. Atomics

```
#include <atomic>
```

The most important concurrency tool present in the new language is support for native atomic operations. Previous versions of C and C++ required non-portable inline assembly (possibly via a third party library) or special compiler "builtins" to work with atomic types. The new version contains a complete set of atomic tools, including base types, generic overrides, and an entire memory model with many options for clarifying the desired read/write semantics.

The atomic type that can serve as the basis for any other atomic structure is `std::atomic_flag` [13]. Beyond that, all of the language defined simple data types have corresponding overrides: `std::atomic_bool`, `std::atomic_int`, `std::atomic_size_t`, etc. Even pointers work atomically, using generics:

```
template<class T> struct std::atomic<T*>
```

Without knowing anything else about memory models, atomicity, or concurrency principles, a user can take an atomic type, share it between threads, and use conventional operations as if it were a normal type (add, subtract, compare/exchange, etc.) The language will guarantee that the memory backing the object will maintain integrity. For example, the object will not encounter a situation where one thread writes to two bytes of a four-byte integer, while a second thread thread reads the partially written type. Instead, the first write will complete entirely before the second read starts.

B. Threading

```
#include <thread>
```

By definition, concurrency requires running multiple execution paths in parallel. While these parallel operations could be entirely separate processes, the use of threads instead tends to increase performance gains. Thread creation is faster than process creation, and communication between threads is significantly easier. Each thread in a process shares the entire memory space, so there is no need to use external interprocess communication (IPC) methods such as shared memory, pipes, message queues, etc. Should a thread need a given object to stay local to the thread instead of global to the process, C++11 introduces the `thread_local` keyword. This keyword makes the most sense when applied to a member variable that should remain static within the lifetime of the thread. [13]

The reference platform is highly multithreaded. All tools described in later sections use the threading facilities provided by the language. Native tools provide only what the language cannot, e.g. setting the processor affinity, scheduler priority, and scheduler type. Note that an important feature, thread pools, is unavailable in C++11. DCPS implemented custom thread pools instead, in a replaceable way should the language adopt official thread pools in the future.

C. Move Semantics and RValue References

Copying data between objects is a potentially slow and invasive operation. Sometimes, depending on the actual characteristics of a constructor, it can have unacceptable impact on real-time determinism. Recall that no real-time system should ever allocate memory during a real-time event. Recall also that the most common container, the vector, will allocate memory during construction. This leads to a challenging situation when trying to return a vector from a function. Without careful considerations, a seemingly innocuous function call could, in the absence of return value optimization (RVO), lead to several vector copies through the use of temporaries.

Move semantics are a new tool in C++11 that elide some of these inefficiencies entirely. Consider a function that returns a vector by value. The compiler must generate code that copies that vector using the copy constructor. It most likely allocates

memory for a temporary in the process, only to deallocate the memory moments later. The new move operation instead allows using a move constructor to invalidate the internal meta data of the source vector and populate the meta data of the target vector without ever allocating temporary storage. The "rvalue reference" naming in this case refers to the common situation where the temporary mentioned above sits on the right hand side of an equals sign, though this is not a required prerequisite. [13]

The real-time queue described later uses this new feature to optimize moving data into a queue virtually instantly.

IV. Automatic Real-time Atomic Lock

The easiest concurrency mechanism is a mutex lock. These are unfortunately inherently slow, requiring up to 65 µs to release a contended lock on an otherwise idle CPU. With the NSTX-U 5 kHz system, spending a third of each cycle unlocking just one mutex is a nonstarter. The solution involves careful process yielding with an RAII atomic lock.

C++11 introduces the `std::atomic_flag` object that has only two members: `test_and_set()`, and `clear()`. This is enough, however, to serve as the foundation for atomic devices at any level of complexity. It will also become a vital tool for subsequent constructs in later sections.

The design is simple, and uses several important features of C++:

```
struct AtomicLock {
  std::atomic_flag & flag;
  AtomicLock(std::atomic_flag & f): flag(f) {
    while (flag.test_and_set())
      std::this_thread::yield();
  }

  ~AtomicLock() {
    flag.clear();
  }
};
```

First, the constructor takes in a reference, and initializes the local member reference with it, putting the onus on the caller to ensure that all instances share the same flag. The constructor then tries to repeatedly set the flag, and only exits when the old state is zero, indicating that no other thread owns the lock. RAII kicks in with the destructor, to guarantee that the lock is exception safe. If an exception ever occurs, the destructor will always unlock the flag. Therefore, a simple usage of this lock follows:

```
void reentrantFunction() {
  static std::atomic_flag f(ATOMIC_FLAG_INIT);
  AtomicLock lock(f);
  // Do work
}
```

Every call to the function above will share the same flag instance, and will block waiting for the atomic lock to instantiate. The first call to the function will initialize the flag (so-called "lazy initialization") with a safe value. When the function exits, either normally or because of an exception, the destructor will clear the flag so that the next waiting function can proceed.

There are several caveats with this simplistic approach. Multiple waiters will fight with the scheduler and each other to win the next slot. Starvation is possible, but easily mitigated in practical scenarios where the amount of contention is low with respect to the time required for each thread to finish work. Waiters in this example use a spinwait, which will tie up the CPU, blocking any process with a lower priority. This is a tradeoff, avoiding the context switches at the expense of one less processing core. Customization is possible at the expense of complexity, tailored to the needs of the user application. As an example, it would be feasible to pass in a lambda function to the constructor that it would call inside the while loop, with a reasonable default set to either a no-op, or the existing yield. This way, each thread could take a possibly different action when the lock is unavailable.

V. Real-time Queue

Adding items to a `std::queue` on most implementations requires allocating memory from the heap. This is dangerous in a real-time loop, and introduces unallowable system calls. This particular queue design is also not thread safe, so users must graft concurrency mechanisms on top of the underlying data structure. The example that follows combines a modification to the lock described in Section 3 with preallocated storage and ring buffer iterators:

```
template<class T, std::size_t size = 1000>
struct RealtimeQ {
  RealtimeQ():
    flag(ATOMIC_FLAG_INIT),
    q(size), front(q.begin()), back(q.end())
  {}

  void enqueue(T && x) {
    AtomicLock lock(flag);
    *front++ = std::move(x);
      if (front == q.end())
        front = q.begin();
  }

  T dequeue() {
    AtomicLock lock(flag);
    T & x = *back++;
    if (back == q.end())
      back = q.begin();
    return x;
  }

  std::atomic_flag flag
  std::vector<T> q;
  std::vector<T>::iterator front, back;
};
```

This queue uses the atomic lock from Section 3, modified in a way to block the thread calling `enqueue()` if the queue is empty. The size defaults to 1000 elements, configured at instantiation. There is no safety in overflow; it is the caller's responsibility to choose an appropriate size. This is a tradeoff between safety and real-time determinism. It would be possible, however, to add checks to test for overflow and throw an exception should it occur.

VI. Real-time Logging

Writing log information is an important tool for any reasonably complex piece of software. It gives immediate feedback when problems arise, and provides a timeline of events for later reconstruction. Typical characteristics include timestamps to show when events occur, and logging thresholds or severities to filter important events.

A. Reference Platform Characteristics

The DCPS system uses timestamps with microsecond resolution, and the following 5 log levels: Fatal, Warning, Informational, Debug, and Trace.

1) Fatal

Fatal entries trigger immediate programmatic shutdown, as something catastrophic has occurred. Typically this involves some situation from which the software cannot recover.

2) Warning

Warning entries report that something bad transpired, but do not result in shutdown actions. This is useful if for instance some internal limit such as a buffer size reaches a given percentage of capacity.

3) Informational

Informational entries represent the typical default level of logging, and provide a high level sequence of the real-time event as it progresses through various stages, from startup to configuration to execution to post-event recording. No informational messages occur during the actual RT execution to avoid unnecessary system perturbation.

4) Debug and Trace

The last two levels provide an increasing level of detail into the inner workings of important operations. These tend to increase intrusiveness with detail, and so only serve any purpose when troubleshooting a problem.

B. Real-time Constraints

To maintain determinism, a real-time application must refrain from triggering non-essential interrupts in the underlying system. The most common activity causing intrusive interrupts outside of memory management is writing to the filesystem, either to the underlying disk storage or to a file based device such as /dev/console or /dev/pty. Writing to a log file directly or writing log information to stdout can lead to devastating effects depending on the timing constraints of the application. Therefore, an ideal real-time logger will not obstruct the main real-time threads should any interrupt occur.

Further, the act of preparing the log entry may introduce overhead while constructing the information string. There are two sources of this overhead: processing logger function arguments, and converting arguments to a character representation from some other data type. The first can be problematic if an argument is not a Plain Old Data (POD) type, or if it requires a nontrivial function call. The second becomes an issue when the argument is not already a character type, and conversion to a character type involves potentially long operations. Therefore, the logger must delay handling arguments until required, which may imply never depending on the current allowable log level threshold.

C. Class Design

Writing a log message requires calling through an external interface using a preprocessor macro. While macros are generally bad form in the C++ world [14], they are unavoidable in this use case. There is no other way to short circuit log entry processing. The macro loosely expands to the following:

```
#define LOG(log, level, ...) \
  if (level > log.minLevel) \
    log.append(level, __VA_ARGS__);
```

This makes use of variadic preprocessor arguments to forward along the entire log statement, however the user intends. The if-statement prevents ever touching any of the arguments if the stated level is too low (for instance, if a log entry contains detailed trace data useful only in certain situations.) Forcing the user to write the same if-statement for every call defeats the purpose of abstraction and object oriented design, but the language has no support for delayed evaluation or partial evaluation of function call arguments. Using the preprocessor is typically a last resort; in this case, it is the only choice.

There are three phases that must happen for any log entry not disabled by the logging threshold level: 1) argument evaluation, 2) timestamp injection, and 3) string conversion and storage.

If the log threshold is low enough, the variadic argument list propagates to the internal logging methods. If this were a traditional C program or a C++ program using the C interfaces, the function would invoke the variadic methods of `<stdarg.h>` or `<cstdarg>` respectively. Instead, C++11 provides new variadic templates that allow processing the arguments recursively using variadic template arguments. Each level of recursion pulls off another argument in the list, until a final call with no arguments. The following code illustrates this technique as it applies to handling a log entry:

```
template<class T, typename... Args>
void append(T x, Args... msg) {
  // append x to the log;
  append(msg...);
}

void append() {
  // msg... is empty, so append a newline
}
```

The compiler creates an instance of the first function for every type in every call used throughout the program not explicitly overridden (such as the final one), making use of the C++ idiom, Substitution Failure Is Not An Error (SFINAE). Using this pattern, it is also possible to inject intermediary handlers that move the data along without knowing the exact contents. This last technique provides a way to move the data off of the real-time execution path and onto different threads scheduled elsewhere.

The Logger class as implemented contains two RealTimeQueue members to handle moving the data, one

running at an elevated FIFO priority and the other running at a normal TS priority. The queues themselves use a `std::function` template type, allowing any function call signature as the queue contents through tools such as `std::bind`. The high priority queue serves only one purpose. It injects a timestamp at the front of the log entry before forwarding everything to the low priority queue, which performs the remainder of the work. When writing a log entry, for the log to be of any practical use, the timing information contained therein must be accurate and reliable. Therefore, the call to determine the current time must happen as close as possible to the point in the real-time execution path where the diagnostic exists. Moreover, concurrent log entries must be sequenced in the logical order they occurred. The logger implementation presented sacrifices negligible timing accuracy for guaranteed ordering correctness.

To actually acquire the time stamp requires careful use of the new `<chrono>` library. Despite the implementation agnostic nature of the standard libraries, calls to `std::chrono::high_resolution_clock::now()` can have wildly different results. The solution chosen on NSTX-U was to configure GCC v4.8.2 with the `--enable-libstdcxx-time=rt` option, which uses a Linux Virtual Dynamically Linked Shared Object (vDSO) to get time in roughly 200 ns instead of 2000 ns, an order of magnitude difference.

Therefore, the sequence described above follows the following timeline:

1. Determine if logging is enabled. Short-circuit all remaining processing if not.
2. Collect all arguments into a std::bind object (including hitting all side effects, if any), and pass to the high priority queue.
3. Inject a timestamp to the front of the argument list, and pass to the low priority queue.
4. Recursively peel off arguments using a variadic template with SFINAE, appending each to the log.
5. Write a new line to the log.

VII. (Bi-directional) Atomic Synchronization

As mentioned earlier, mutexes provide an easy yet invasive way to coordinate multiple threads of execution. Since the timing constraints of the DCPS reference platform preclude using mutexes during the real-time event execution path, the only remaining choice for any thread coordination mechanism is the use of atomic types to signal readiness between threads. Unfortunately, atomics do not support complex communication mechanisms out of the box like mutexes do.

A. Motivating Use Case: Algorithm Pools

As with the other design patterns shown thus far, the NSTX-U DCPS wastes resources in exchange for virtually zero latency. The same is true when implementing algorithm thread pools. Section 3 mentions briefly that pools are not present in C++11, but they are a common tool used to group similar execution threads together. In fact, Linux supports grouping threads and processes together natively.

To meet the 5 kHz timing deadline while executing all of the algorithms, DCPS divides the work into discreet tasks assigned to one of 18 threads each occupying a dedicate CPU core. Therefore, each thread contains a fixed but unequal chunk of work. The threads need to coordinate the beginning and end of each cycle. Since each thread will finish at a different time, they all must rendezvous at a barrier released only when the last thread is ready.

B. Atomic Barriers and Signaling

There is no predefined way to use atomics as signals between threads. Therefore, the point of this design is to create a signaling mechanism that is extensible beyond the specific requirements of DCPS.

1) Initial Starting Barrier

Bootstrapping the algorithm threads occurs outside of the real-time event, so no specific requirements block the usage of conventional signaling means. DCPS uses a `std::mutex` coupled with a `std::condition_variable`, with each thread waiting after thread creation and a notification sent until all threads have checked in.

2) Atomic Counters

In early designs, the worker threads communicated with a master control thread using atomic booleans. However, this proved ineffective, as a simple boolean cannot convey enough information to signify bidirectional information. A counter works better, but the final product requires two counters: one for each thread to signal that it is ready to start work, and another to signal that it has finished the work. The master thread then waits for the ready counter to indicate that all threads are ready before opening the gate on all threads at once. It then waits for the finished counter to indicate that all threads have reached their own finish lines. At that point, it can make the next dataset ready for consumption, and repeat the process.

Each thread, be it a worker thread or the master thread, can increment or decrement a counter using atomic operations without fear of corrupting another thread. They can also spin waiting for another thread to change a particular value. For instance, if the master thread knows that there are 18 worker threads, the following is a reasonable communication idiom:

```
// Master
ready = 19;
while (ready > 1)
  std::this_thread::yield();
--ready;

// Worker
--ready;
while (ready > 0)
  std::this_thread::yield();
doWork();
```

The worker threads decrement the counter and wait for the final decrement from the master thread. The master thread waits until all worker threads have decremented the counter, then it decrements the counter itself. This is the bi-directional nature of the synchronization mechanism. At first glance,

there may appear to be a race condition if the final worker decrements the counter to 1 and the master thread decrements it to zero before the worker can check that it is still greater than zero. That is why the contents of the while loop – by design – must be optional. In this example, a yield is surely optional. Variations on this approach must take care to maintain the optional nature of the while loop contents.

There is a similar barrier at the end of each periodic cycle to allow each worker to signal to the master that processing is complete. This creates a zone between two atomic barriers during which the master can easily check that processing completes in a timely fashion. Individual workers are free to finish at any time before the expiration period. The only guarantee on timing is that by the end of the cycle period, either every worker has finished, or the system has triggered a fatal abort.

3) Final Cleanup

Once the real-time event has ended, the worker threads must exit gracefully. Doing so requires testing for an exit condition at particular interruption points, and setting that exit condition externally. In this case, there is an atomic boolean shared between the worker threads and the master thread that the workers check twice near the beginning of the cycle. The design borrows from the popular Double Checked Locking Optimization design pattern [15]. In the example shown earlier, the worker would check the shutdown flag both before decrementing the ready counter as well as after exiting the while loop. Once the worker enters a cycle, it is impossible to stop until the cycle completes. In practice, other checks through the system ensure that this poses no threat to system integrity, and in any unforeseen case, at most one cycle will run. This keeps the design metrics within the specification tolerance.

VIII. Conclusion and Future Work

The NSTX-U DCPS successfully uses C++11 in a hard real-time machine protection environment. Doing so requires careful use of language features and customized devices that exploit the concurrency mechanisms in a real-time safe way. Atomic types provide the basic building block for all objects presented here, such as the bi-directional atomic synchronization and the RAII atomic lock. The latter lock enables a real-time safe queue, which further serves as the groundwork for a complete logging framework. All of these tools and processes operate without affecting the determinism of the overall system, such that it can still make very tight microsecond deadlines on standard commercial off the shelf hardware.

Version 2 of the DCPS will see improvements in all real-time devices. Modifications to the Real-time Queue will allow greater control over the actions taken while waiting for lock acquisition through the use of lambda functions and function objects. The Logger class will make use of new features in GCC 4.9 to allow generic lambda functions, greatly simplifying the framework required to push a `std::function` onto the high and low priority delayed execution queues. The Atomic Lock will in the future no longer require repeated local implementations, instead benefitting from physical reusable code instead of the current situation of a reusable idiom. This allows for instance the easy adaptation and implementation of a mixture of both global and local atomic flags, which can allow locks to function in more complex capacities without sacrificing run-time determinism. Finally, the atomic synchronization class could use an overall polishing to better handle more generic situations, though this is more of an evolutionary rather than revolutionary change.

References

[1] *Information Technology – Programming Languages – C++*, ISO/IEC 14882:2011, 2011

[2] *Programming Languages – C++*, ISO/IEC 14882:2003, 2003

[3] *Programming Languages – C++*, ISO/IEC 14882:1998, 1998

[4] Menard, J.; Caniky, J.; Chrzanowski, J.; Denault, M.; Dudek, L.; Gerhardt, S.; et al., "Overview of the physics and engineering design of NSTX upgrade," *Fusion Engineering (SOFE), 2011 IEEE/NPSS 24th Symposium on*, pp.1,8, 26-30 June 2011 doi: 10.1109/SOFE.2011.6052355

[5] Woolley, R.D.; Titus, P.H.; Neumeyer, C.L.; Hatcher, R.E., "Digital Coil Protection System (DCPS) algorithms for the NSTX centerstack upgrade," *Fusion Engineering (SOFE), 2011 IEEE/NPSS 24th Symposium on* , vol., no., pp.1,6, 26-30 June 2011

[6] Erickson, K.G.; Tchilinguirian, G.J.; Hatcher, R.E.; Davis, W.M., "NSTX-U Digital Coil Protection System software design," *Fusion Engineering (SOFE), 2013 IEEE 25th Symposium on*, vol., no., pp.1,6, 10-14 June 2013 doi: 10.1109/SOFE.2013.6635500

[7] Erickson, K.G.; Tchilinguirian, G.J.; Hatcher, R.E.; Davis, W.M., "NSTX-U Digital Coil Protection System Software Detailed Design," *Plasma Science, IEEE Transactions on*, vol.42, no.6, pp.1811,1818, June 2014 doi: 10.1109/TPS.2014.2321106

[8] Brosky, S.; Rotolo, S., "Shielded processors: guaranteeing sub-millisecond response in standard Linux," *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, pp.9 22-26 April 2003 doi: 10.1109/IPDPS.2003.1213237

[9] *GCC, the GNU Compiler Collection*, Available at http://gcc.gnu.org/

[10] *GCC 4.8 Release Series*, Available at https://gcc.gnu.org/gcc-4.8/

[11] *GCC 4.9 Release Series*, Available at https://gcc.gnu.org/gcc-4.9/

[12] The Single UNIX ® Specification, Version 2 (1997) by The Open Group

[13] Williams, A., *C++ Concurrency in Action, 1st* ed. Shelter Island: Manning, 2012.

[14] Kumar, A.; Sutton, A.; Stroustrup, B., "Rejuvenating C++ programs through demacrofication," *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, pp.98,107, 23-28 Sept. 2012 doi: 10.1109/ICSM.2012.6405259

[15] Schmidt, D., Stal, M., Rohnert, H., Buschmann, F., "Synchronization Patters," *Pattern-Oriented Software Architecture Vol. 2: Patterns for Concurrent and Networked Objects*, vol. 2, New York: Wiley, 2000. pp. 353-363

Princeton Plasma Physics Laboratory
Office of Reports and Publications

Managed by
Princeton University

under contract with the
U.S. Department of Energy
(DE-AC02-09CH11466)

P.O. Box 451, Princeton, NJ 08543
Phone: 609-243-2245
Fax: 609-243-2751

E-mail: publications@pppl.gov
Website: http://www.pppl.gov