Local threads
A programming model that prevents data races while increasing performance.

Tobias Rieger

May 12, 2014

Abstract

Data races are evil and must be prevented[1][2].
A data race is defined as two threads accessing the same memory location at the same time with one writing. This leads to four ways of preventing data races: Single threaded programming, making sure threads use distinct memory locations, using mutual exclusion to prevent concurrency or not modifying the values.
This paper proposes a new programming model that prevents data races while improving performance compared to the common model of sequential consistency under the condition that no data races exist. This paper uses C++ for code examples[3]. The principles, however, also apply to other imperative programming languages such as C, Python and Java.

[1]Boehm, "How to Miscompile Programs with "Benign" Data Races".

[2]Boehm, "Position Paper: Nondeterminism is Unavoidable, but Data Races Are Pure Evil".

[3]`#includes`, the required `main` function and the `std` namespace are intentionally left out.

1 Local thread definition

Local threads are threads that use `thread_local` storage (TLS) for every variable. TLS is supported in C++11 through the storage modifier `thread_local`[4]. A variable with the `thread_local` modifier exists once per thread, each thread accessing its own separate version. For local threads this behavior is the default for all global variables. Consider example 1:

Example 1: Race from zero to non-zero

```
int var = 0;

void inc{
    for (int i = 0; i < 10000; i++)
      var++;
}

void dec{
    for (int i = 0; i < 10000; i++)
      var--;
}
int main(){
  async(inc);
  async(dec);
  wait_until_threads_finished();
  return var;
}
```

There are three threads involved in this example: The main thread and the two threads created by `async`[5]. In the C++11 memory model this would be a data race since `var` is modified in two threads at the same time and thus this would be an ill-formed program. While no guarantees can be made about the behavior of such a program the typical effect is that even though `var` starts at 0, is incremented 10000 times and decremented 10000 times, `var` may have a seemingly random value at the end, for example -2131832621.

When using local threads, however, this program is completely legal, because `var` is automatically `thread_local` and exists three times, once for each thread. Consequently each thread accesses its own copy[6] of `var` and no two threads access the same variable. Therefore no data race

[4]It will turn out later that the `thread_local` keyword is insufficient for local threads, so this does not impose a limitation on the languages local threads can be applied to.

[5]For C++ experts: Assume for these examples that the issue of the destructor of the future returned by `async` blocking until the thread finishes does not apply and that the compiler always chooses the launch::async policy.

[6]The starting value of the `thread_local` variable could be either the value defined at program start (as done by C++11) or the value of the variable of the creating thread. In this example it makes no difference.

exists in this program. The first thread created by `async` counts its `var` up to 10000, meanwhile the second thread counts its different `var` down to -10000. Finally the main thread returns its unmodified version of `var` with the value of 0.

Local threads always automatically prevent all data races since no two threads can ever[7] access the same memory location, which is a precondition for data races. However, this introduces a problem, because now threads are not able to communicate through shared memory anymore. To solve this problem the keyword `sync` is introduced. The `sync` keyword revokes the `thread_local` nature of the global variable (so there is only one variable for all threads) and also adds mutual exclusion for the `sync`-modified variables similarly to `atomic` in C++. Mutual exclusion means that no two threads can access a variable at the same time. Typically this behavior is implemented using a mutex, but may also be generated by special assembler instructions without an actual mutex.

Local threads therefore make sure that all variables are either `thread_local` or properly synchronized and thus data race free without the need of manual synchronization by the programmer. It needs to be possible to manually lock the implicit mutex of one or more `sync` variables[8] though to implement for example a compare-and-swap-function and complex data structures. In the following examples a `sync_lock` function is used that takes an arbitrary number of `sync` variables and locks their mutex.

The mentioned C++ keyword `thread_local` is insufficient for local threads in two aspects. First it is only a storage specifier (such as static, register and extern), not an access specifier (such as private, protected and public). It only specifies how a variable is stored, not who can access it. A thread can access another thread's `thread_local` variable, provided it somehow gets a pointer or reference to that variable or just guesses the address. For local threads this must be disallowed, resulting in undefined behavior[9] if bypassed[10]. The other in-

[7] It is possible to attempt to access any memory location in C++. Doing so with another thread's `thread_local` copies would result in undefined behavior.

[8] In C++ `atomics` cannot be locked. Therefore `sync` variables can only be implemented as `atomics` if locking of the implicit mutex is not required.

[9] The term "undefined behavior" comes from the C++ standard and means that any behavior is legal and no guarantee can be made.

[10] In C++ it is not possible to prevent code from attempting to access arbitrary memory locations. In some other languages such as Java this problem does not occur.

sufficiency is the lack of `thread_local` dynamic memory. In the C++11 standard it is not possible to allocate memory with the `new` operator that is `thread_local`[11]. Dynamic `thread_local` memory needs to be added in order to use local threads effectively. To be able to compare the performance of local threads to established forms of programming the established models will be introduced.

2 Established memory models[12]

2.1 Strict sequential consistency (S-SC)

Sequential consistency seems to be the dominating memory model as of today[13]. It is the only memory model for Java and the default memory model for C++11. Sequential consistency requires that the code needs to appear to execute in the order it was written in with some interleaving for threads. Sequential consistency is generally not used in its strict form, because it is very inefficient. Consider the following example:

Example 2: Optimization potential

```
int va, vb;

int foo(){
  va = 1;
  vb = 6;
  va++;
  vb--;
  return min(va, vb);
}
```

This code has some optimization potential. For one the `va = 1` and `va++` could be combined to `va = 2` and the `vb = 6` and `vb--` to `vb = 5`. Additionally the minimum of 2 and 5 is 2, so the call to `min` (returning the minimum of `va` and `vb`) could be optimized away as shown in the following example:

[11] The Windows API includes the functions `TlsAlloc`, `TlsGet`, `TlsSet` and `TlsFree` since Windows XP that do implement dynamic `thread_local` memory.

[12] If you are very familiar with the various versions of sequential consistency you may skip the section on established memory models (since they do not contain new information) and continue reading "Optimizations in local threaded sequential consistency" on page 10.

[13] Other memory models such as causal or eventual consistency memory models could also be supported by local threads.

Example 3: Faster, more elegant and wrong

```
int va, vb;
int foo(){
  va = 2;
  vb = 5;
  return 2;
}
```

It is possible to tell that this optimization is not sequentially consistent by using a debugger or other thread to monitor the value changes of `va` and `vb`. One could see that the assignments `va = 1` and `vb = 5` are never executed. Additionally, if the value of `va` would be changed to 9 by a debugger or other thread during the assignment to `vb`, the return value would change from 2 to 5 in the original code, but it would stay 2 in the optimized code. Optimized code returning a different result than the original code is not correct, thus this transformation is not legal in a strictly sequential consistency model. One may object that looking at or changing the variable `va` creates a data race, which is correct. S-SC does not forbid data races though, so it is legal to do so. A system running with the S-SC model needs to work correctly even in the presence of data races. This can be achieved by not reordering any code and using memory barriers to synchronize every memory access. Such a system would be very slow and almost impossible to optimize and thus is generally never implemented.

2.2 Single threaded sequential consistency (ST-SC)

While understanding why such optimizations are not correct, one may feel that they *should* be correct. A much better[14] system is single threaded sequential consistency (ST-SC). It adds the rule that only a single thread may be used. In this model the above optimizations become correct, because it is impossible to tell if the optimization has been done or not. A second thread or debugger could tell the difference, but different threads have been explicitly forbidden. Furthermore a debugger is an activity that is not part of the one and only allowed main thread and thus forbidden. The only observable actions of `foo` are the change of the variables `va` and `vb` and the return value of `foo`[15], which are identical between the original and the optimized version of `foo`, thus making the optimization legal. Now consider a less commonly seen optimization:

[14] From an optimization possibility point of view.

[15] One may object that one could measure the run time of the function and figure out if the function executes faster than it should. However, the point of optimizing code is to make it run faster, so this effect is intended.

Example 4: Protected

```
int var;
mutex varmut;

void moo(){
  varmut.lock();
  var++;
  varmut.unlock();
}
```

Here a mutex `varmut` is used to protect the access to the variable `var`. Remember that a single threaded environment is used, so there is no other thread that can be mutually excluded. Mutexes do not seem to make sense in a single threaded environment. However, common functions, operators and objects such as `printf`, `cout`, `malloc`, `new` and `shared_ptr` are synchronized using some form of mutual exclusion, because they need to be thread safe. Not all environments and libraries offer different implementations for single- and multithreaded situations. Thus being able to optimize synchronization mechanisms away is important for single threaded performance and reduction of code duplication and programming effort. On a first attempt one may just remove all mutex related code. However, if `varmut` is already locked before `moo` is called, it would result in a deadlock (which interestingly single threaded applications are able to do). A deadlock and incrementing `var` are not the same result, thus removing all mutexes is not correct[16]. If reentrant mutexes[17] were used, just removing all mutex related code would be correct, since reentrant mutexes never have any effect in a single threaded environment. But even non-reentrant mutexes may be optimized by implementing them with simple boolean variables and normal memory accesses instead of the usually more expensive[18] synchronizing instructions. Another interesting optimization is to move `var++` out of the critical section. This may seem odd, since `varmut` is meant to protect `var`, which is defeated by moving `var` out of the `varmut.lock-varmut.unlock` area. However, there is no other thread to synchronize with. Moreover, `moo` has one of two effects: It either increments `var` or it deadlocks. It is impossible to tell if `var` was incremented before or after a deadlock, since the only legal thread that could figure it out is deadlocked, so it does not matter when `var` is incremented.

[16]For an example why one would insist on producing a deadlock consider a nuclear power plant controller application that rather deadlocks than making the wrong decision.

[17]Reentrant mutexes are those that may be locked repeatedly by the same thread. The idea is that if a thread already locked a mutex, locking it again can automatically be allowed.

[18]This is interestingly not as often the case as one might expect.

2.3 Data race free sequential consistency (DRF-SC)

While the optimization possibilities for single threaded code are astounding, it is believed for some time now that its performance will not increase significantly in the future[19]. When more performance is required multiple processors (cores) need to be utilized by multithreaded code. This also changes the memory model, since ST-SC does not allow multiple threads, while S-SC allows multiple threads, but does not deliver the required performance. The problem is that unknown threads[20] may read or write any variable at any time, thus making optimizations near impossible. In data race free code, however, one can tell if and when a variable will be accessed, because it must be protected by some mutual exclusion mechanism. Mutual exclusion mechanisms and protected variables may therefore not be reordered to preserve sequential consistency of those variables. The optimization rules for DRF-SC state that code cannot move out of mutexes anymore, but can still move into mutexes[21]. Consider the following example:

Example 5: Mutex optimizations

```
int var;
mutex varmut;

void lu(){
  var = 1;
  varmut.lock();
  varmut.unlock();
  var = 2;
}
```

In this example `var` is needlessly being assigned twice. The first assignment should be optimized away. Optimizing across mutexes is usually incorrect, since mutexes potentially allow another thread to look at the variables without producing a data race, in which case sequential consistency must be maintained. Using the rule that code can move into the mutex, however, both assignments can be put into the critical section and then the first assignment can be eliminated:

[19] Sutter, "The free lunch is over - A Fundamental Turn Toward Concurrency in Software".

[20] A compiler may not see the whole code because source files may be compiled separately or libraries may be linked to the program.

[21] Moving code into critical sections is legal but not advisable since one wants to keep the time of locking a mutex to a minimum to allow for concurrent execution. Having all but one thread wait to get a mutex defeats the purpose of multithreading.

Example 6: Mutex optimized

```
int var;
mutex varmut;
void lu(){
  varmut.lock();
  var = 2;
  varmut.unlock();
}
```

One may get the idea that this optimization is incorrect since the assignment `var = 1` has been removed. Also `varmut` may have been locked, stopping the thread executing `lu` at the exact position where the assignment of 1 to `var` should have happened and the assignment of 2 has not happened. To attempt to proof that the above optimization is incorrect a test function can be written:

Example 7: Challenge

```
void lutester(){
  var = 0;
  varmut.lock();
  async(lu);
  while (var != 1){
  //wait until lu changes var to 1
  }
  varmut.unlock();
}
```

The function `lutester` sets `var` to 0, locks `varmut`, starts a thread executing `lu` and waits for `var` to change to 1. Meanwhile the unoptimized function `lu` will set `var` to 1 (allowing `lutester` to continue), then get stuck on locking `varmut`. Then `lutester`'s thread continues to unlock `varmut` which lets both threads finish. If the optimized version of `lu` is used instead, `var` will not be set to 1 and both threads get stuck, creating a deadlock. So this was obviously not a sequentially consistent transformation. The reason for that is that the example does not not obey the only restriction imposed in DRF-SC, which is to not write data races (if you did not see the data race immediately consider switching to local threads where this mistake cannot happen). In the above code, `var` is read in the `while` loop, meanwhile it is written inside `lu`. This is a read-write data race, making the program ill-formed and thus voiding (among other things) the guarantee of sequential consistency.

To fix a data race on an integer, `atomic`[22] seems an appropriate choice, so one can change the declaration of `int var` to `atomic<int> var`. In the `lu` function variable `var` cannot be moved into the mutex anymore, because every access to `var` is now protected by mutual exclusion, which may not be reordered. So now two as-

[22]Variables declared `atomic` are automatically accessed in a mutually exclusive way. This is typically implemented more efficient than actually having, locking and unlocking a mutex. In C++11 only primitive data types can be declared `atomic`, including arrays.

signments to `var` are required. Additionally, in `lutester` before making `var` `atomic` in the `while` loop, `var` may have been register promoted[23]. So even if `lu` had changed `var` to 1, `lutester` may not have noticed since only the memory of `var` changed, not the register. This optimization was legal, because no other thread could change `var`, because that would be a data race, which is forbidden. Now, because `var` is accessed mutually exclusively, it is not possible to keep `var` in a register, instead it needs to be reloaded from memory every time, because it may have been changed by another thread. Without the data race the code again behaves sequentially consistent, but fewer optimizations are possible. Consider another example[24]:

Example 8: Not obvious

```
int va = 0, vb = 0;
void Fa(){
  if (va != 0)
      vb++;
}
void Fb(){
  if (vb != 0)
      va++;
}
```

The functions `Fa` and `Fb` run concurrently. First note that this code does not contain a data race. Since `va` and `vb` are both 0, the function `Fa` reads only `va`, while `Fb` reads only `vb`. Since the code is data race free, all optimizations must be sequentially consistent. A possible optimization is speculative execution.

Example 9: Data race insertion

```
void Fa(){
  //same optimization for Fb
  vb++;
  if (va == 0)
      vb--;
}
```

First the work is done, then, if it was wrong to do so, the mistake is undone. Similar optimizations are known as lock free algorithms. This would be a legal optimization in ST-SC, however, in DRF-SC this optimization creates a data race[25]. `Fa` as well as `Fb` now access `va` and `vb` concurrently. When executed at the same time `Fa` will increment `vb` while `Fb` increments `va`. Then they see that their speculative execution paid off, since they do not need to undo it, since both `va` and `vb` are not 0. The problem

[23]Register promotion means to keep a variable in a CPU register instead of in memory. This speeds up execution.

[24]Example from "atomic Weapons: The C++ Memory Model and Modern Hardware" 2013-02-11 H. Sutter

[25]This is not necessarily a problem, because in assembler code it is possible to write benign data races, which is not really possible for C++ code.

is, that the result is incorrect, `va` and `vb` ending up with the value 1 is impossible in the original code. Thus this optimization is not legal in DRF-SC. The above examples show that the switch from ST-SC to DRF-SC made some optimizations illegal, thus decreasing performance, which may or may not be compensated by utilizing multiple cores or processors. Still one may argue that DRF-SC is optimal since it keeps the shared variables in sequentially consistent order while allowing optimizations on non shared variables[26]. I will now return to local threads to show that it can be done better.

3 Optimizations in local threaded sequential consistency

The precondition of local threaded sequential consistency (LT-SC) is that only local threads are used, which use `thread_local` storage by default and mutual exclusion when specified with `sync`. Traditional non-local threads without the limitation of not being able to access other threads' data are not allowed in LT-SC. There is no rule not to write data races since it is inherently impossible[27] to do so when using local threads. The optimization rules are similar to those for DRF-SC. Accesses to `sync` variables as well as the `sync_lock` function are not allowed to be reordered to preserve sequential consistency, but everything else can be optimized. The previously mentioned limitations of DRF-SC, however, do not apply to LT-SC. While DRF-SC cannot reorder around mutexes, LT-SC can. Consider the following example:

Example 10: Locked

```
sync int va;
int vb;

void mo(){
  sync_lock(va){
      vb *= 2;
      va = vb * vb;
  }
}
```

Note that `va` is a `sync` variable and therefore has an implicit mutex[28] associated with it while `vb` is a `thread_local` variable. The function

[26]The error in this argument is that DRF-SC cannot tell which mutex protects which variable, so it assumes that all variables are protected by all mutexes, which results in unnecessary restrictions in possible optimizations.

[27]It actually is possible to produce data races by accessing another threads local storage or using a debugger. This is not allowed in the LT-SC model.

[28]The mutex may or may not be implemented by an actual mutex. In this case assembler instructions similarly to an `atomic<int>` would be sufficient.

`sync_lock` will acquire the implicit mutex for `va`, execute the code inside the `sync_lock` and release the mutex. To optimize this code the time the mutex is held can be minimized, which maximizes concurrency and thus performance. Even though `vb` is inside a mutex, `sync_lock` clearly states that it protects `va` and not `vb`, which cannot be accessed by other threads. Therefore `vb` can be moved out of the mutex protected area.

Example 11: Unlocked

```
sync int va;
int vb;

void mo(){
  vb *= 2;
  //let r be a register
  r = vb * vb;
  sync_lock(va){
      va = r;
  }
}
```

The calculations have been moved out of the lock and only an assignment remains. Therefore the synchronization is minimized. It turns out that LT-SC can do all ST-SC transformations except that it needs to keep accesses to `sync` variables and the `sync_lock` function in order and cannot optimize the implicit mutexes away. Note that local threads can not prevent all errors. Consider the following example:

Example 12: What to do?

```
sync int var = 0;

void inc(){
  for (int i = 0; i < 10000; i++){
      var = var + 1;
  }
}

int main(){
  async(inc);
  async(inc);
  wait_for_threads_to_finish();
  return var;
}
```

Local threads make sure that this program is properly synchronized. However, the program may not do what the programmer intended it to do. A programmer may assume that the program always returns 20000, but in fact it may return any number between 10000 and 20000 inclusively (but no other numbers like in the first example). Consider one thread running first, incrementing `var` to 9999, then reading the 9999, incrementing it to 10000 and then get preempted before writing the changed `var` back. The other thread runs to completion, incrementing the 9999 to 19999. Finally the first thread finishes writing its 10000. The final result is only 10000, not 20000. This is known as the lost update problem which is a determinacy race. In this case it is a logic error, which cannot be prevented[29]. The program does exactly what the code says it should do, it just may not do what the programmer wanted. The program has no way of knowing if it was intended to return 20000 or any number between 10000 and 20000 (which I intended it to do, so it may actually do exactly what it is meant to do, depending on the perspective). To fix this problem one would either use `sync_lock` or `var++`, which prevent this lost update.

4 A model for comparing memory models

Comparing memory models seemed unnecessary for programmers before LT-SC. If an application is single threaded choose ST-SC, for multithreaded applications choose DRF-SC. Now that LT-SC is an option one needs to decide if DRF-SC or LT-SC is to be preferred. Additionally, since ST-SC and LT-SC have the same optimizations in case no `sync` variables are used (which are unnecessary in single threaded mode), one could always pick LT-SC regardless of threading concerns. Additionally the future may bring more memory models to choose from. Thus an objective metric to compare memory models is needed. The core problem is preventing data races. Just letting the system deal with it by choosing S-SC is not an option from a performance point of view. There are four preconditions for data races: Two threads, concurrent access, same memory location and one thread modifying data. This definition directly leads to the four ways to prevent data races: Allowing only one thread, mutually excluding concurrent access, keeping memory locations disjunct for all threads and not allowing modifications. Every variable must have at least one of these DRF strategies at any time. A memory model should support all of the above strategies on a per variable per logical time frame basis. Note that these strategies may change over time (example in DRF-SC):

[29] Boehm, "Position Paper: Nondeterminism is Unavoidable, but Data Races Are Pure Evil".

Example in DRF-SC:

Example 13: DRF strategy change

```
int var;
mutex mvar;
void inc(){
  for (int i = 0; i < 10000; i++){
      mvar.lock();
      var++; //lock required
      mvar.unlock();
  }
}

int main(){
  async(increment);
  async(increment);
  wait_for_threads_to_finish();
  return var; //lock not required
}
```

In LT-SC the example would look like this:

Example 14: Local thread imperfection

```
sync int var;

void inc(){
  for (int i = 0; i < 10000; i++){
    var++;
    //lock required and
    //automatically acquired
  }
}

int main(){
  async(increment);
  async(increment);
  wait_for_threads_to_finish();
  return var;
  //automatically acquires lock
  //for var, which is not
  //required, but cannot be
  //avoided by the programmer
}
```

In this example two threads increment the same variable. To prevent a data race a mutex is locked, either explicitly for DRF-SC or implicitly when accessing a `sync` variable for LT-SC. Note that while `var` is a shared variable inside `inc`, in the `main` function it is not shared anymore. Inside the `inc` function mutual exclusion is used to prevent data races while inside the `main` function single threaded execution is utilized to prevent a data race.

The data race prevention strategy changes over time and one can imagine that a more sophisticated program might change its DRF-strategy multiple times for various variables. The DRF-strategy needs to be part of the type system to prevent accidental data races, however, since in C++ data types do not change over time[30], changes can not be expressed through types. Consider another example in DRF-SC:

Example 15: Strategy composition

```
mutex strmutex;

void makePalindrome(string &s){
  //thread safe due to
  //mutual exclusion
  strmutex.lock();
  s += string(rbegin(s), rend(s));
  strmutex.unlock();
}
bool isPrintable(const string &s){
  //thread save due to
  //non-modifiability
  for (const auto &c : s)
    if (!isprint(c))
      return false;
  return true;
}
```

The above code runs two functions on the same string concurrently (main function starting the threads and passing the same string to both functions is not shown). One function uses mutual exclusion to protect the string while the other function uses the string in a non-modifying way to prevent data races. Yet even though mutual exclusion as well as not modifying any variable are valid ways to prevent data races, the above example contains a data race and is thus illegal in DRF-SC. The data race prevention strategies only work if all threads use the same strategy per variable per time frame. The model also needs to ensure that at least one DRF strategy is used to guarantee that the code is data race free and thus correct and fast.

[30]Polymorphism as implemented in C++ does not help.

To summarize, here are the seven features that an optimal memory model needs to support:

- thread local variables
- Mutual exclusion
- Constant variables
- Single threading
- Switching strategies
- Enforcing the existence of one strategy for all threads per variable per time frame
- Allow optimizations

One may argue that ease of use or understandability is another important feature, but that is difficult to evaluate objectively[31]. Let us see how the mentioned memory models would be evaluated.

S-SC allows data races, as such it does not require any DRF strategy, so it would get full points for the first four features. Switching strategies or enforcing strategies is not required either, so full points there also. In the last category, however, S-SC does terrible. Basically no optimizations are possible resulting in very poor performance.

ST-SC enforces the single threaded DRF strategy. It does not allow multithreading.

Functional programming languages can be seen as enforcing all variables to be non modifiable as their DRF strategy, but do not allow switching to other strategies.

DRF-SC allows all DRF strategies as well as switching between them, but does not enforce a DRF strategy and thus makes it possible to compile illegal (as in not DRF) code. DRF-SC has fewer optimization possibilities than ST-SC, but more than S-SC.

Finally let's look at LT-SC. It excels at `thread_local` variables, which is its main feature. Mutual exclusion is built into the `sync` variables and is also available through `sync_lock`, while other forms of mutexes can be optimized away. Single threaded mode is supported by means of not using `sync` variables, which bares no overhead over using ST-SC. It also has the added benefit of working correctly

[31] I would argue in favor of LT-SC, that in this model it is not necessary to understand what memory models or data races are. Having all variables be `thread_local` by default seems somewhat arbitrary, but not difficult. Forgetting synchronization may still result in a determinacy race, but that cannot be avoided. The guarantee of sequential consistency and absence of data races make debugging easier than in DRF-SC when data races are present.

[32] Initially every thread has its own constants, but that can be optimized away by the compiler or linker so that all threads use the same constant.

even in the presence of other threads. Constants are also supported[32]. Adding `const` to a `sync` variable makes it unnecessary to lock it, otherwise a local copy can be used. Switching strategies works with some overhead. Switching to single threaded mode can be done by making local copies of all `sync` variables and only access the local variables. The same applies to the non modifiable strategy. An unnecessary synchronized read per variable is required to perform the switch to single threaded or the non modifying strategy as well as keeping possibly unnecessary copies of the data. Switching to a mutual exclusion based approach is also easily done with `sync` variables. LT-ST also forces every variable to either be `thread_local` or properly synchronized. By maliciously attempting to access other threads' `thread_local` storage or modifying top level constants one may still manage to create a data race, but it seems unlikely that this happens accidentally. Finally LT-SC allows ST-SC optimizations with the limitation of not being able to reorder memory accesses to `sync` variables and `sync_lock` function calls and not being able to optimize away implicit mutual exclusion.

Overall LT-SC looks very promising. What it keeps from being optimal is the overhead of switching strategies, which generally costs one unnecessary synchronized read of variables and some unnecessary copies of thread local constants. Additionally a memory model may exist that can reorder `sync` variables and `sync_lock` operations without for-fitting sequential consistency in some cases, which LT-SC is unable to do.

5 Cache coherency protocols

Modern computers have multiple processors or cores. For performance reasons they each have their own cache. This makes it possible that one processor[33] has a different view of the memory than another, since one processor may see a variable having one value while another sees it having a different value, both looking at different stale cached values. Cache coherency protocols (CCPs) make sure that changes in one cache will propagate to all other caches and an inconsistent view as described above is not possible. CCPs do not scale well since for n caches any CCP must send n^2 messages between caches. With enough processors, they will spend most of their time invalidating each others caches.

LT-SC provides an approach to solve this problem. Software can now precisely tell for every memory location if it is shared or not. All variables fall in two categories: thread local and

[33]In this section processor, thread, core and cache can be used interchangeably.

synchronized. The thread local memory can be cached without the need for invalidating others' caches since only one thread can access that memory. The global memory can be divided into mutexes and global data. Caching mutexes is useless[34]. Caching global data only pays off if the same thread accesses the same global data twice in a row without another thread making changes, which should not happen often. So caching of global data can also be given up without significant performance loss. Consequently CCPs can be either removed, reduced or made more efficient when running under the LT-SC model.

6 Weaknesses and improvement potential

Local threads do not prevent deadlocks. It may be possible to specify local thread semantics in a way to do that while keeping sufficient flexibility.

Local threads bind mutexes on variables which becomes inefficient when one mutex could protect multiple variables or different variables at different times. This can be mitigated by compiler optimizations.

7 Future Work

When programming in LT-SC one will find that declaring and using complex data structures which use mixed strategies is non-trivial and should be explored further. One should figure out what should happen when a CPU switches execution to another thread when using the thread local information to boost performance. An idea is to just invalidate the whole cache or to push it into a backup cache that allows efficient cache swapping. LT-SC should be implemented, which should be easy since LT-SC is very similar to ST-SC, which is already implemented. Once a good implementation exists further studies on efficiency such as the average speedup of common problems is possible. A metric on ease of use and understandability with a comparison of DRF-SC and LT-SC would be of interest. Local threads do not require sequential consistency and could be combined with causal or delta consistency.

8 Conclusion

Local threads potentially have significant improvements in terms of performance over the common DRF-SC model, albeit still having

[34] If the state of a mutex is known it still needs to be locked and it also needs to be made sure concurrent locking of one mutex by multiple threads is prevented. Knowing the state of a mutex ahead of time provides no advantage.

some unnecessary overhead. LT-SC makes debugging of synchronization errors easier, since it guarantees sequential consistency even in the presence of errors, unlike DRF-SC with data races. Local threads also offer new potential for building more efficient caches. In the future ST-SC and DRF-SC should be replaced by either LT-SC or an even better system.

References

Boehm, Hans-J. “How to Miscompile Programs with "Benign" Data Races”. In: HotPar’11.

Boehm, Hans-J. “Position Paper: Nondeterminism is Unavoidable, but Data Races Are Pure Evil”. In: RACES ’12.

Sutter, Herb. “The free lunch is over - A Fundamental Turn Toward Concurrency in Software”. In: *Dr. Dobb’s Journal* (Mar. 2005).

9 Appendix - Implementation specification for C++

The implementation of local threads depends on the programming language used. Local threads restrict which and how memory can be accessed. Integrating local threads into C++ seems to be more difficult than in other languages, because C++ generally does not make such restrictions and gives the programmer tools to bypass restrictions. Some other languages do not allow unsafe accesses[35] and may instead have trouble with garbage collection since deleting a `sync` variable requires locking that variable which may indefinitely block. The following section explores the problems and possible solutions specifically for C++ and assumes the reader has a basic understanding of C++11. This is only an example specification for academic purposes. A real specification should be done by an appropriate standardization committee including peer review which is beyond the scope of this paper.

9.1 Pointers[36]

Local threads are based on the idea that all variables and variable accesses are either thread local or properly synchronized. Any implementation of local threads must guarantee this. In C++ one problem is to specify pointers. Either a pointer points to thread local data or it points to a sync variable. C++ has strongly typed pointers so a pointer declaration includes the description of what the pointer points to (called the pointee).

[35] Unsafe access means that the compiler may not be able to prove or check if a valid memory is accessed leading to undefined behavior, which Java for example does not allow.

[36] References behave similarly to pointers when integrating them with local threads and are not specified separately. The only difference is that pointers can be `sync` while references cannot, because references in C++ are immutable (same as `const`).

Example 16: Pointer declaration syntax

```
const int *pci; //pci is a pointer to a const int
int *const cpi; //cpi is a const pointer to an int
const int * const cpci; //cpci is a const pointer to a const int
int *const *pcpi; //pcpi is a pointer to a const pointer to an int
int **const cppi; //cppi is a const pointer to a pointer to an int

sync int *psi; //psi is a pointer to a sync int
int *sync spi; //spi is a sync pointer to an int
sync int *sync spsi; //spsi is a sync pointer to a sync int
int *sync *pspi; //pspi is a pointer to a sync pointer to an int
int **sync sppi; //sppi is a sync pointer to a pointer to an int
```

Example 17: Pointer assignment rules

```
int i;
sync int si;
int *pi;
pi = &i;
pi = &si; //error: sync int * cannot be assigned to int *
sync int *psi;
psi = &i; //error: int * cannot be assigned to sync int *
psi = &si;
int *sync spi;
spi = &i;
spi = &si; //error: sync int * cannot be assigned to int * sync
```

When accessing a `sync` variable its implicit mutex will be locked for the duration of the access. When accessing a `sync` variable through a pointer the implicit mutex of the `sync` variable will be locked as if it was accessed directly without a pointer. If a pointer does not point to a valid variable and is dereferenced the behavior is undefined as with regular pointers. A `sync` pointer will automatically lock its implicit mutex before its address is read. The lock is held until evaluation of the expression is complete. If a pointer is dereferenced to a `sync` variable the `sync` variable is locked after which the pointer is unlocked, possibly allowing a pointer to unlock before an expression has been evaluated.

Example 18: Sync pointer access rules

```
1 sync vector<int> sv;
2 sync vector<int> *psv = &sv;
3 psv->push_back(42);
```

In the above example `psv` is a thread local pointer pointing to the `sync` variable `sv`. In the expression in line 3 `psv` is dereferenced, `sv` is locked, the `vector` member function `push_back` is executed and then the lock of `sv` is released.

```
4 sync vector<int> *sync spsv = &sv;
5 spsv->push_back(43);
```

In this example `spsv` is itself a `sync` variable pointing to the `sync` variable `sv`. The expression in line 5 locks the `sync` pointer `spsv`, dereferences it, locks the pointee `sv`, unlocks pointer `spsv`, executes the `push_back` and unlocks pointee `sv`.

```
6 vector<int> v;
7 vector<int> *sync spv = &v;
8 spv->push_back(44);
```

The `sync` pointer `spv` points to the thread local variable `v`. In the expression in line 8 `spv` is locked, the `push_back` is executed and `spv` is unlocked. Great care must be taken that `v` can only be accessed through `spv`, otherwise a data race on `v` can occur. There are different ways to ensure this. One way is to forbid `sync` pointers to thread local variables entirely. A C++11-specific way is to make `sync` pointers to thread local variables only take rvalues[37], making it undefined behavior if the original value is accessed. Another way is to move the thread local variable to a new global location, which is difficult to do since pointers may point to the first element of an array.

[37] An rvalue is a temporary variable. It is used in C++11 to move objects instead of copying them and then destroying the old one which can be more efficient. A non-temporary can be made a temporary with std::move. After a variable has moved from one place to another, the old location generally contains garbage and accessing it results in undefined behavior, though it may be assigned a valid value again.

9.2 Legacy code compatibility - unstorables

Legacy code (code that has been written without local threads) should still work when using local threads to allow backwards compatibility. Specifically code that takes non-`sync` objects should still work with `sync`-objects. In the following example[38] a `sync char *` is used that works with the unaltered standard library's `strlen`-function.

Example 19: Legacy code compatibility `strlen`

```
1
2 int main(){
3   const sync char *str = "Hello world!";
4   return strlen(str);
5 }

6 size_t strlen(const char *s){
7   size_t retval = 0;
8   while (*s++)
9     retval++;
10  return retval;
11 }
```

There are different ways to implement `strlen`, this is only an example. The problem with the example is that `str` from the `main` function is passed to `strlen` in line 4, which converts a `const sync char *` to a `const char *`, which loses the `sync`. This is only legal if `str` is locked before calling `strlen`, unlocked after the call and not stored during the call. This leads to the term unstorable pointer[39]. When a `sync` pointer is turned into a thread local pointer the `sync` pointer is locked for the life time of the thread local pointer and the thread local pointer becomes unstorable, which means its life time must not exceed that of the `sync` pointer it was created from. Copying an unstorable pointer

[38] The example code has some issues. In C++ the `string` class should be preferred to C-strings and `strlen` returns a `size_t`-object which may differ in size and differs in signedness from `int`. This, however, is not relevant to local threads.

[39] This applies to references as well.

makes the copy unstorable as well. This allows the above example to work correctly.

A different implementation of `strlen` may look like this:

```
size_t strlen(const char *s){
  const char *c = s;
  while (*c++){
  }
  return c - s;
}
```

In this example `c` is initialized with `s` which is unstorable so `c` also becomes unstorable.

```
1  size_t strlen(const char *s){
2    static const char *lastseen;
3    if (*s == 'H')
4      lastseen = s;
5    const char *c = s;
6    while (*c++){
7    }
8    lastseen = "hello";
9    return c - s;
10 }
```

In the above implementation `lastseen` is also unstorable because it may be assigned an unstorable value in line 4. Furthermore overwriting it by a regular thread local storable value in line 8 does not make it storable to reduce code analysis and compilation time for more complex examples. Since the unstorable pointer `lastseen` exceeds the life time of the passed pointer it causes a compilation error[40].

[40]It is somewhat uncharacteristic in C++ to require such an analysis. C++ does not for example required a compiler to produce a compilation error when a references to a function local variable is `return`ed, instead it is just undefined behavior. However, a compilation error on stored unstorable objects is required to keep up the data race free guarantee and it has no runtime cost.

9.3 Arrays

There are different ways `sync` works with arrays. Either every array element is `sync`, only the array itself is `sync` or both. For multidimensional arrays every dimension should be specifiable with `sync`.

Example 20: Array declarations

```
sync int asi[10]; //array with 10 sync ints
int sai sync [10]; //sync array with 10 ints
sync int sasi sync[10]; //sync array with 10 sync ints
int asai [10] sync [10]; //array of 10 sync arrays of 10 ints
int saai sync [10][10]; //sync array of 10 arrays of 10 ints
int aais [10][10] sync; //syntax error
```

When accessing an element of a `sync` array the whole array is locked while accessing a `sync`-element only locks the specific element. This allows the programmer to choose between the overhead of locking a whole array when only one element is required or having one implicit mutex per element. Locking both the array as well as the element is possible but not useful. However, when using `vectors` of `vectors` this becomes useful.

9.4 Complex data structures - syncable and early unlocks

It is easy to just put the `sync` modifier on a C++-container such as `vector` to fit it into the thread local model. However, that will lock the whole container for every access. It would be nice if different threads could instead work on different elements of a container concurrently. To prevent code duplication another keyword `syncable` is used. It must be used on a non-`static` member or member function of a `class`[41]. When `syncable` is used outside of a `class` or on a `static` member it should result in a compilation error. If the object created from a `class` is `sync`, `syncable` has the same effect as `sync`, otherwise it has no effect. The following example uses a singly linked list to show the general idea of how to work with `sync` and complex data structures. It intends to show how `sync` works with complex data structures and does not aim to be otherwise useful.

[41] There is no local thread specific difference between `class` and `struct`, therefore they are used interchangeably here.

Example 21: Syncable singly linked list

```
1 struct Node{
2   int value;
3   syncable Node *next;
4   Node(int v) : next(nullptr), value(v){}
5   void append(int v) syncable{
6     if (!next)
7       next = new syncable Node(v);
8     else
9       next->append(v);
10  }
11  //Destructor, copy constructor, move ctor, ...
12 };

13 int main(){ //usage
14   Node n(1);
15   n.append(2);
16   sync Node sn(3);
17   sn.append(4);
18 }
```

First note that the `next` pointer in line 3 is `syncable`. For `Node n` in line 14 the member `Node::next` is of type `Node *` whereas the `Node::next` for sn in line 16 is of type `sync Node *`. The member function `append` in line 5 first tests if `next` is empty. If so it will create a new `syncable Node` that is assigned the passed value `v`. If `next` is not empty the pointee pointed to by `next` will append the value `v`. Note that dereferencing `next` will produce a `sync` variable. According to the pointer rules `next` will be unlocked after the pointee of `next` is locked. That way multiple threads can append to the same `Node` concurrently.

In C++ non-`static` member functions have an implicit `this`-pointer parameter that points to the object the member function was invoked on. This is necessary for the member function to know which object's members to access. Line 9 in the above example can be written as `this->next->append(v);` with no semantic difference. Dereferencing the `this`-pointer to access `next` happens implicitly. Note that since the `Node`-object is `sync`, the type of the `this`-pointer is `sync Node *`[42]. Special rules apply for unlocking the `this`-pointer after it has been dereferenced to a `sync` variable, because the `this`-pointer should not be allowed to change while executing a member function on an object. Otherwise it would be difficult to keep an object in a consistent state. However, in this specific case it is easy to prove that the `this`-pointer will not be accessed inside `append` after line 9, so early unlocking is possible. Unlocking can only happen after the last access to the `this`-pointer of a member function to satisfy the constraint that the object and the `this`-pointer will not change while a member function runs on it.

Note the `syncable` keyword after `append`. This `syncable` refers to the pointee of the `this`-pointer. The same syntax is used in standard C++ to specify the pointee of the `this`-pointer as `const`. Since it is not harmful to treat a non-`const` object as a `const` object it is allowed to call a `const` member function on a non-`const` object. It is, however, potentially harmful to modify a `const` object, so calling a non-`const` member function on a `const` object will not compile. Similarly when using local threads calling a non-`sync` member function on a `sync` object is correct if the caller locks the object before passing it to the member function and unlocking it after the member function returns. Inside the member function the object will be treated as a thread local object with the `this`-pointer and the object being unstorable. Passing a non-`sync` object to a `sync` member function cannot work correctly since the member function will attempt to lock mutexes that do not exist. In the above example the `syncable` keyword is used to specify that this member function works with either `sync` or non-`sync` objects, but this actually duplicates the function into two different functions, one with `sync` and the other without. While the duplicated bodies of the functions are identical (except for `syncable` being interpreted either as `sync` or nothing), the code generated for them will most likely be different, because the compiler must insert locking instructions for the `sync` version when accessing the `this`-pointer which happens implicitly when accessing any member while it must not do so for the non-`sync` version.

[42] It is also an rvalue so it cannot be modified even though there is no `const` in the type of the `this`-pointer.