MiniCS: Critical Section Minimisation in Concurrent Programming

Hyunsu Kim
KAIST
Daejeon, Rep. of Korea
hyunsu.kim00@kaist.ac.kr

Jaemin Yu
KAIST
Daejeon, Rep. of Korea
platinant@kaist.ac.kr

Doam Lee
KAIST
Daejeon, Rep. of Korea
ehdkacjswo@kaist.ac.kr

Jaemin Choi
KAIST
Daejeon, Rep. of Korea
jmchoi98@kaist.ac.kr

Heeju Wi
KAIST
Daejeon, Rep. of Korea
bb0711@kaist.ac.kr

ABSTRACT

Modern machines consume massive amount of data and deals with drastic amount of computations, as the era of artificial intelligence. In the hardware manufacturing industry, developing SoC (System on a chip) optimized to the computations is one of the main on-going trends. As a result, multi-core machines are prevalent nowadays. Still, thanks to the end of Moore's law and Dennard's scaling, software implementation that synchronizes multiple threads is required in order to achieve reasonably scalable computing. However, it is often a fear to write/refactor multi-threaded code due to its non-deterministic behavior. In order to tame the parallelism with scalability, using synchronization primitives such as lock and semaphore is mandatory to prevent race conditions. While which API to use is rather trivial, it is usually known that setting critical sections (when to acquire a lock, then when to release the lock) is not trivial at all. In this work, we tackle such intractability by applying combination of heuristics for search and static analysis for rigorous evaluation and correctness. We use genetic algorithm to traverse over combinatorially explosive search space and analyze the code statically to detect possible data race. Further descriptions about the experiments can be found in section 3, and the evaluation of our progress can be seen in section 4. In sections 1 and 2, we state the problem in detail and cover basic concepts which might be unfamiliar to most readers, however crucial to understand our work. Concluding remarks and possible improvements for the research are stated in section 5.

1 INTRODUCTION

As the era of artificial intelligence and big data, a program that runs on single core machine has fundamental limitation in terms of computation power. In hardware manufacturing industry, developing SoC optimized to such computations is already on-going process. Still, thanks to the end of Moore's law and Dennard's scaling, software implementation that synchronizes multiple threads is unavoidable in order to achieve reasonably scalable computing.

There has been an aggressive development of libraries and languages that exploits concurrency such as TensorFlow, PyTorch, CUDA, OpenCL [1], and etc. Inside each implementation, concurrent programming enables multi-threading, and this is what really drives the high-level logic to be correct and efficient. Some libraries hide the complicated logic for concurrency underneath, and some provides basic API as directives to let programmers fill in the gaps with their desired logic. Developer of the library in former case, while any programmer who tries to use the language in latter are those who really meets the nitty-gritty details of concurrency. Traditionally, it has been a fear for most system developers to develop multi-threaded codes. Such aspect can be easily understood by observing how many stale bugs are still present in many projects (e.g. Mozilla's Firefox) related to concurrency.

Multi-threaded codes often comes with the problem called *data race*. It indicates the situation of multiple threads accessing the same memory location simultaneously which is not a intention of a programmer. Data race results in non-deterministic executions, which eventually gives wrong output depending on the specific interleaving order of threads. Such issue can be resolved by imposing exclusive locks to prevent simultaneous access from more than desired number of threads. However, abusing lock will block other threads' execution, then lead to longer execution time or even a deadlock, which is very severe. Further details can be found in 2.1. In our work, we do not rigorously deal with the problem of deadlocks which is left as one of our possible improvement.

It is important to protect shared memory location using locks while minimising the interval between acquiring the lock and releasing it. However, it is hard to manually achieve the sweet spot without data races that probably reveals on runtime. Our project, **MiniCS** is a solution tackling the problem in natural setup. We generate and evolve the population of candidate programs equipped with synchronization primitives. Then, apply genetic algorithm to get the best performing program without data race. With that being said, we come up with the following research questions:

RQ1 How well MiniCS detects the data races?

RQ2 How does MiniCS measure the performance of each multi-threaded code rigorously?

RQ3 How long does it take to find the best performant multi-threaded code using MiniCS?

By answering to these questions, we suggest our solution in a more detailed and convincing manner.

2 BACKGROUND

Minimising critical section is one of the most important aspect when refactoring multi-threaded code. However, it is impossible

to compare the performance of any two distinct version of code by just measuring the execution time. It is hard to guarantee those measured durations solely depend on each code execution, unless we take full control over thread scheduling, which is totally up to operating systems. In order to overcome such issues, we took approach counting machine instructions especially in Intel x86 architecture. Further details about how our evaluations went through will be mentioned in section 3 and 4. Here, we briefly go over two concepts that needs to be explained before moving on to details.

2.1 Data race

A *data race* is a situation where multiple threads try to access shared data at once, and the result depends on the execution order of the threads. For example, consider the following code:

```
1  int a = 0;
2  int foo() {
3      a++;
4  }
```

Suppose two threads call the `foo` function at once. If the threads do not run concurrently, then the value of `a` will be 2. However, this is not always the case when there is concurrency, because `a++` actually consists of three steps: reading the value, incrementing the value, and writing back the value. The following sequence of actions may occur as a result:

(1) Thread 1 reads the value of `a`, which is 0.
(2) Thread 2 reads the value of `a`, which is 0.
(3) Thread 1 increments its value, making it 1.
(4) Thread 2 increments its value, making it 1.
(5) Thread 1 writes 1 into the value of `a`.
(6) Thread 2 writes 1 into the value of `a`.

We use locks to prevent this situation. When a thread tries to acquire a lock, but another thread owns the lock, then it must wait before the other thread releases the lock. For example, in the modified code below, two threads would wait for each other to read the value of verb|a|, increment it, and write it back. This keeps the global variable from being accessed twice at the same time.

```
1  int a = 0;
2  int foo() {
3      pthread_mutex_lock(&mutex_lock);
4      a++;
5      pthread_mutex_unlock(&mutex_lock);
6  }
```

2.2 Machine instructions

In compiled languages like *C/C++*, each line of code is translated into multiple machine instructions. For example, consider ***a++*** located at third line of code snippet in 2.1. It is adding one to integer variable ***a***. Although it is a single line of code in *C*, a machine equipped with Intel x86 processors translates it into the following instructions:

```
ld  $r1, a
add $r1, $r1, 1
st  $r1, a
```

It first loads the value of ***a*** from memory into register *$r1*. Then, add one to the register value. Finally, it stores the result back to the memory location of ***a***. Note that these machine instructions are the commit point for each computation within a machine. It means that we can ensure there is no thread interleaving while executing each instruction. In fact, this is the reason why we are safe to count the number of instructions to measure performance of target program independent from other programs being executed.

3 EXPERIMENT DESIGN

3.1 Environment setting

MiniCS runs on Linux, requires Python (3.6 or higher) and Clang (v6.0.0 or higher). Note that `gm.cpp` requires an external library, however one can also use pre-compiled binary file, `gm`. To manually compile the source code, `llvm-dev` must be installed into Clang [2]. Our project code is open sourced in Github repository [3], and collaborators are as follows: Hyunsu Kim (*hyunsukimsokcho*), Jaemin Yu (*planetarynebula*), Doam Lee (*ehdkacjswo*), Jaemin Choi (*jh05013*), and Heeju Wi (*bb0711*).

3.2 Gene representation

We could use AST format generated from Clang as our gene, however, we made the reprsentation simpler for efficient computation. Our gene is a list of 2-tuples where each tuple indicates location (line number) for acquiring lock (s_i), and location for releasing it (e_i). Here is the gene representing a program with n many locks:

$$G = \{(s_1, e_1), (s_2, e_2), \cdots, (s_n, e_n)\}$$

Generation and manipulation of genes are further described in the following sections.

3.3 Population generation

As a prototype, we assume the following constraints:

(1) All scopes are explicitly given with curly braces.
(2) In the `if`-branches, `for`-branches, and `while`-branches, there is no line break between the closing parenthesis and the opening brace. There is no line break between the braces and `else`.

Under these assumptions, we find a *valid range* $[a, b]$, and modify the code so that a lock is held at the end of the a-th line and released at the end of the b-th line. A valid range to lock must satisfy the following four conditions:

(1) The range must be contained in a function.
(2) There must be a reference to a global variable inside the range. Otherwise there would be no need to place a lock.
(3) Every scope must either fully contain a range, or be fully contained in a range. This is to prevent a thread from holding its own lock or releasing a lock that it does not hold.
(4) There should be no `return` statement in the range. This is to prevent a thread from finishing while holding a lock.

For example, consider the following code:

```
1  int arr[100];
2  int bar(int n) {
```

```
3       int x = 0;
4       for (int i = 0; i <= n; ++i) {
5           arr[i] = x;
6           x+= i;
7       }
8       return arr[n];
9   }
```

A range [4, 5] is valid; in this case, we hold and release a lock right before and after executing `arr[i] = x`. On the other hand, [1, 9] is invalid because it is not contained in a function. [5, 6] is invalid because there are no global variables. [2, 5] is invalid because the scope inside the `for` loop neither contains [2, 5] nor is contained in [2, 5]. Finally, [7, 8] is invalid because it contains `return`.

We use Clang AST to detect the references to global variables. Each condition is checked by the AST matcher with the following procedures:

- A global variable reference is detected by a `DeclRefExpr` node, which has an ancestor `FunctionDecl` node, and refers to a `VarDecl` node with the `hasGlobalStorage` property.
- A scope in a function is detected by a `CompoundStmt` node.
- A return statement in a function is detected by a `ReturnStmt` node.

After finding all references to global variables, all return statements, and all scopes, we output all possible valid ranges in the program. To generate a population, we choose zero or more valid ranges such that no two ranges with the same kind of lock do not intersect.

3.4 Mutation

We forced the genes used in the genetic algorithm to have only a valid lock range. For this reason, a difficulty arises in mutation. If any of the lock ranges are changed randomly, invalid lock ranges can be created. As such, mutations should not change the given gene freely but should give only limited changes. To solve this problem, we need to look at the following facts.

(1) Gene is a set that has a lock range as an element.
(2) If there is a set U with all possible lock ranges, then all genes are a subset of U.
(3) The elements of the gene do not intersect with each other.

We generate U using clang AST. And, U is needed for the mutation to work properly. After making changes to the gene, the changed gene must still be a subset of U for this mutation to be valid. We used two methods under these conditions.

The first way is to add a lock. Let's say G is the gene that we want to mutate. Create $G^{'}$ by selecting an element of U that does not intersect G and adding it to G. $G^{'}$ is still a subset of U and the elements do not intersect with each other. Thus, $G^{'}$ is a valid gene.

The second way is to remove one lock. Create $G^{'}$ by removing one of the elements of G. Since $G^{'}$ is a subset of G, $G^{'}$ is also a subset of U. And the elements of $G^{'}$ do not intersect each other. Thus, $G^{'}$ is a valid gene.

3.5 Crossover

Because we force only valid genes as in mutations, we need to prove that the altered gene is still valid at the crossover. We used a single-point crossover. We have sorted the range of genes to make implementation easier. Assume that applying the crossover to the following two genes G_0 and G_1.

$$G_0 = \{(s_1, e_1), (s_2, e_2), \cdots, (s_n, e_n)\}$$
$$G_1 = \{(l_1, r_1), (l_2, r_2), \cdots, (l_m, r_m)\}$$

If split point x is set at G_0, G_0 is split to $\{(s_1, e_1), \cdots, (s_i, e_i)\}$ and $\{(s_{i+1}, e_{i+1}), \cdots (s_n, e_n)\}$, to satisfy $e_i \leq x \leq s_{i+1}$.

G_1 is divided in the same way. However, if G_1 has (l_j, r_j) which satisfies $l_j \leq x \leq r_j$, there is a problem. We can't divide a lock range with (l_j, x), (x, r_j). In the code below, you can see that if you divide $(1, 6)$ by $(1, 4)$, $(4, 6)$, you get an invalid lock.

```
1  int i = 0;
2  cnt = 0;
3  while(i < 10) {
4      cnt = cnt + i;
5      i = i + 1;
6  }
```

We decided to merge them instead of split. Assume that we got the following children after crossover G_0 and G_1.

$$\{(s_1, e_1), \cdots, (s_i, e_i), (l_{j+1}, r_{j+1}), \cdots, (l_m, r_m)\}$$
$$\{(l_1, r_1), \cdots, (l_j, r_j), (s_{i+1}, e_{i+1}), \cdots, (s_n, e_n)\}$$

There is a possibility of overlap between (s_i, e_i) and (l_{j+1}, r_{j+1}). The same is true between (l_j, r_j) and (s_{i+1}, e_{i+1}). In this case, the problem can be solved by combining the two ranges together. If $(x0, y0)$ and $(x1, y1)$ are in U, $(min(x0, x1), max(y0, y1))$ also are in U. There is no `return` or part of the `for`-statement between $(x0, y0)$ and so on between $(x1, y1)$. For this reason, even if the two ranges are combined, the `return` or `for`-statement is not included and all of the valid lock conditions are satisfied.

3.6 Fitness evaluation

Since our goal is to build code "without data race" and has "minimum lock interval", we have two objective for fitness evaluation. Between two of them we have to make the code data race free and then reduce the lock interval, so we are going to give priority to data race. It means that if we have fitness (a, b) which a stands for data race and b stands for lock interval, we compare a first and then compare b if they are same.

3.6.1 Data race free (RQ1). First objective is number of "racing sets". The term "racing set" means the set of two lines that have data race. Since we need data race free code, we need to minimize the number of racing sets to 0. And this number can be obtained by data race detector, ThreadSanitizer [5]. It's part of Clang that can detect data race for given code. The detection report will be offered like the sample below:

```
1  WARNING: ThreadSanitizer: data race
2    Write of size 4 at 0x7fe3c3075190:
3      #0 bar1() simple_stack2.cc:16
4      #1 Thread1(void*) simple_stack2.cc:34
5
6    Previous read of size 4 at 0x7fe3c3075190:
7      #0 bar2() simple_stack2.cc:29
8      #1 main simple_stack2.cc:41
```

This report gives us four racing sets (16, 29), (16, 41), (34, 29), (34, 41) that each (a, b) means a-th line and b-th line have data race. Since the detection result of ThreadSanitizer depends on execution order of threads, we execute ThreadSanitizer for every possible execution orders (for n threads, execute for $n!$ times) and count the number of distinct racing sets.

3.6.2 Lock interval (RQ2). Second objective is number of machine instructions between locks(including themselves). Measuring actual execution time is noisy and can be affected by environment, so we use the number of machine instruction instead of execution time.

The number is measured by using GDB `stepi` function. It runs the program step by step(step means machine instruction) and count the number of them between locks. "Between locks" means containing locks themselves two, because locking and unlocking affects runtime too. More specifically, for given part of code:

```
16  int i;
17  pthread_mutex_lock(&mutex_lock);
18  for (i=0; i<3; i++) {
19      cnt ++;
20  }
21  pthread_mutex_unlock(&mutex_lock);
22  return NULL;
```

It locks on line 17, and unlock on line 21. So on GDB, we set breakpoint on line 17 and execute `stepi` until the execution of unlock on line 21 ends. Now if we count every instructions, it will be the number of machine instructions between locks. On this sample code, it executes 148 instruction between two locks. Additionally, if we change thread during execution the number of execution won't be measured properly, so it's important to execute only one thread at a time.

4 EVALUATION

For the evaluation, we will check whether the program works as expected for given test codes and compare the result with optimal solution. The result of program is given in list $[(s1, e1), (s2, e2), \ldots, (sn, en)]$ which each (si, ei) states for locking range. And the fitness will be given in tuple (a, b) where a is number of racing sets detected by ThreadSanitizer and b is the number of machine instructions executed between locks.

4.1 easytest

"easytest.cpp" is short test to check whether the program considers the number of machine instructions of locks themselves. It has two versions, one iterates `for` statement for one time, and the other iterates three times. The part of first version is like below:

```
16  int i;
17  for (i=0; i<1; i++) {
18      cnt ++;
19  }
20  return NULL;
```

And this is the second version:

```
16  int i;
17  for (i=0; i<3; i++) {
18      cnt ++;
19  }
20  return NULL;
```

For two tests, line 20 causes data race so we have to protect it by lock. There are two choices: place lock inside or outside of `for`-statement. Since locking and unlocking acquires some machine instructions too, for the second version if we place lock inside `for`-statement the lock range will cover smaller amount of code. But lock interval will be longer because locking and unlocking will be performed for three times. As a result, the first version produced lock range [(17, 18)] and second produced [(16, 19)]. It shows that the program considers locking instructions properly.

4.2 global

Since "easytest.cpp" is very short, it easily came out with optimal lock range. Now we have longer test "global.cpp" with 106 lines. For this test we are going to check that the program successfully creates code without data race and reduces lock interval. Then we'll compare the output with optimal lock range.

First, let's see whether the program works properly: prevent data race and minimise lock interval. Initial population starts with best lock ranges [(49, 56), (57, 65), (65, 67), (71, 82), (82, 85), (89, 90), (94, 97), (97, 99)] with 0 data race and 3540 machine instructions. Usually the program starts with lock range that covers almost every part of the code to keep it data free. After 10 generations with 10 candidates, the program comes out with output lock range [(51, 58), (64, 65), (65, 67), (72, 78), (78, 82), (82, 84), (89, 90), (90, 94), (94, 97)] with 0 data race and 2274 machine instructions. We can easily observe that the program excluded the ranges that don't have to be protected and reduce the lock interval properly.

Next, let's compare the best output with optimal lock range. The optimal solution is [(51, 67), (72, 84), (89, 97)]. The optimal solution and program's output have two differences:

(1) Program did not detect the range (60, 64) that causes data race.
(2) Adjacent ranges were merged in optimal solution.

First difference is critical because missing data race can result as wrong code that has potential data race but reported to be data race free. This problem is caused by ThreadSanitizer, so it can be solved by using better data race detector. Second difference is problem of our GA method. Separating adjacent ranges that can be merged causes more locks to be used, and results in increase of lock interval. To resolve this problem, adding merge to mutation operator would be helpful.

4.3 Running time

We ran miniCS using an Ubuntu virtual machine using 4GB memory and 2 processor cores, on a laptop with Intel(R) Core(TM) i7-8550U CPU 1.80GHz and 8.00GB RAM. (*RQ3*)

We ran easytest.cpp three times. The running time was 422, 394, and 512 seconds, the average being 442.67 seconds (7 minutes 22.67 seconds). We also ran dining.cpp three times, with the running

time 175, 126, and 169 seconds, the average being 156.67 seconds (2 minutes and 36.67 seconds). When we ran global.cpp, however, less than 10% progress was done in 20 minutes.

5 CONCLUSION

On evaluation section we observed that our program produced data race free code with short lock interval successfully. On "global.cpp", it reduced the lock interval to 2/3 of initial lock range that covers every code while keeping the code data race free. Since we have observed that our code works fine, now it's time to discuss problems and possible improvements of our program.

5.1 Observed problem

There are two problems of our program: ThreadSanitizer misses data race, significant slowdown of the program. For ThreadSanitizer error, as discussed on evaluation section we may use better data race detector(will be discussed later).

The program is basically slow because ThreadSanitizer detects data race on runtime and causes 5x to 15x of time overhead. But still, there are possible way to avoid slowdown. There are two reasons of slowdown that we can deal with:

(1) ThreadSanitizer should be executed for every possible execution order of threads.
(2) GDB runs the program step by step between locks.

Now let's discuss the solution of these problems. For the missing data race and first problem of slowdown, we may use other data race detection program. There are two choices: Static(code) analyzer and RaceFuzzer [4]. First, static analyzer detects data race depending on the code, not on runtime. So it has no time overhead at all and faster than dynamic detector. But it has problem that it can come up with false alarm that detects data race can't be observed on runtime. And second, RaceFuzzer is similar with ThreadSanitizer but it dynamically chooses random thread unlike original one. So it does not depend on execution order and we can execute the program only once. But it has problem too, that detection is probabilistic and takes more time on one execution because it dynamically schedules threads.

And the second problem of slowdown can be resolved by using "Intel process tracing", namely "Intel-pt". It stores the trace of machine instruction execution with very low time overhead, so we can measure lock interval faster than GDB using Intel-pt. Since machine instruction counting with GDB takes a lot of time, it can significantly reduce time overhead for overall project.

5.2 Possible improvements

First, we may use multiple kinds of locks. Our current version uses only one kind of lock for any kind of locks, but it's inefficient because it will make two different locks that protects different memory location to wait for each other. So using multiple kinds of locks will make multithreaded program even more faster. It will need to figure out which variable the lock is protecting, so this improvement will need more code analysis and harder GA.

Second, we need to detect deadlock. Our current version does not detect deadlock, but deadlock is very serious problem for multithreaded programs. To deal with it, we can use ThreadSafetyAnalysis of Clang. It's static test, so we can detect deadlock within short time and drop the candidate codes that has deadlock issue.

Lastly, but not least, we can try coevolution with thread scheduling. This needs support from control over kernel code which may be necessary to use certain emulator like Qemu or Bochs. While GA part remains the same, we add one more step to generate and evolve schedule of thread interleavings that cause more lost update problems or even deadlocks.

REFERENCES

[1] Bobby R. Bruce and Justyna Petke. 2018. RN / 18 / 04 Towards automatic generation and insertion of OpenACC directives April 12 , 2018.
[2] Clang. 2018. Clang official website. https://clang.llvm.org/docs/HowToSetupToolingForLLVM.html. [Online; accessed November-2019].
[3] Hyunsu Kim, Jaemin Yu, Doam Lee, Jaemin Choi, Heeju Wi. 2019. MiniCS. https://github.com/hyunsukimsokcho/miniCS. [Online; accessed December-2019].
[4] Koushik Sen. 2008. Race Directed Random Testing of Concurrent Programs. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '08)*. ACM, New York, NY, USA, 11–21. https://doi.org/10.1145/1375581.1375584
[5] Konstantin Serebryany and Timur Iskhodzhanov. 2009. ThreadSanitizer: Data Race Detection in Practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications (WBIA '09)*. ACM, New York, NY, USA, 62–71. https://doi.org/10.1145/1791194.1791203