Random Testing for Concurrency Compiler Bugs

A Thesis Submitted

in Partial Fulfillment of the Requirements

for the Degree of

Master of Technology

by

Pankaj Pawan

Supervised By

Francesco Zappa Nardelli [†] *&* **Prof. Amey Karkare** [*]

[†] INRIA Paris-Rocquencourt

[*] IIT Kanpur

to the

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY KANPUR

June 2012

CERTIFICATE

This is to certify that the work contained in the thesis entitled "*Random Testing for Concurrency Compiler Bugs*" by *Pankaj Pawan* has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

[illegible]

Francesco Zappa Nardelli
Research Scientist
INRIA

May 2012

Amey Karkare

Prof. Amey Karkare
Dept. Of Computer Sc. and Engg.
IIT Kanpur

[illegible]

Abstract

The new C++ standard has been recently introduced. Along with many other features, the C++11 standard adds a concurrency model to the language. Many standard optimisations have now become invalid due to precise definition of the *language memory model*.

The concurrency model is complicated and its not easy to implement for compiler developers. Moreover, the current compilers will be used as source to add the notion of concurrency into them. This can lead to bugs as they were written in absence of any memory model.

This motivated us to test for the correctness of compiler optimizations with respect to the C++11 memory model. For that, we have designed and implemented a framework based on trace comparison of sequential code. The key idea is that, since C/C++ compilers must support separate compilation, correctness of compilation of concurrent programs can be tested by analysing which optimizations are applied to sequential code, and then relying on a general theorem stating if those optimizations are sound when the code runs in a concurrent environment.

We already found and reported a subtle concurrency bug due to write introduction in the latest version of GCC, thus validating the relevance of our approach.

Acknowledgements

First and foremost, I would like to extend my enormous gratitude towards my supervisor Francesco Zappa Nardelli for presenting me with the opportunity to work with him. He has been an immeasurable source of inspiration, motivation and knowledge for me. I learnt invaluable lessons of perseverance and focus from him. I would also like to thank him for taking pains to provide me with an excellent academic environment where I could continue my thesis without any roadblocks.

I would also like to thank my co-advisor Prof. Amey Karkare for showing confidence in my work and helping me to pursue my research without worrying about any administrative aspects.

I would also like to thank Peter Sewell, Jaroslav Sevcik for the various useful discussions we had. Their insight proved to be instrumental for our progress.

I owe thanks to Hardik Patel for the technical discussions and handholding he provided towards the thesis. I thank Tanmay Mogra for patiently reviewing my thesis. I am thankful to my friends in IIT Kanpur for the wonderful time spent together and making my stay at IIT Kanpur such a memorable experience. I thoroughly enjoyed every discussion, useful or useless, over the past five years.

I extend my sincere gratitude toward IIT Kanpur and the Department of Computer Science and Engineering for providing a warm and friendly learning environment. I am grateful to the Institute for its excellent infrastructure and freedom. I would also like to thank INRIA for providing financial assistance during my stay in Paris.

Last but not the least, I would like to thank my sister and my parents for the encouragement and unconditional love they have shown throughout my life.

Contents

Chapter 1

Introduction

Compilers are complex pieces of software and complex pieces of software are always prone to bugs: bugs in compilers are inevitable. For instance Regehr et al. [6, 10] reported more than 400 bugs in widely-used C compilers. Most of these bugs were in the compilers' middle-ends, where most of the optimizations take place. When it comes to optimizing potentially concurrent code, compilers must be extremely careful, as even standard optimizations may be incorrect.

Consider the following shared memory concurrent code:

requestReady = responseReady = data = 0

Thread 1	Thread 2
`data = 1;`	`if (requestReady == 1) {`
`requestReady = 1;`	`  data = 2;`
`if (responseReady == 1)`	`  responseReady = 1`
`  print data`	`}`

If all shared locations, i.e. `data`, `requestReady` and `responseReady`, are initialised to `0` then this program cannot output `1` in any execution. However, an optimizing compiler will propagate the constant `1` from the `data:=1` statement and replace the `print data` statement with `print 1`. While this optimization is correct for sequential code, it introduces a new, unexpected behaviour in the original concurrent program.

To enable a compiler to implement aggressive optimization without introducing unexpected behaviours, language designers specify the *language memory model*, that define which *memory writes* each *memory read* is allowed to see. The language memory model can be seen as a contract between what programmers can expect from their code and the ways in which a compiler/hardware can reorder or eliminate memory accesses.

The recent C++11 standard precisely defines a concurrency and memory model for the C++ language, and the next revision of the C language will adopt the same memory model to preserve interoperability. The C/C++11 memory model guarantees that the programmer can expect an interleaved semantics for “well-synchronized” programs, also referred to as *data-race free* programs [5]. This model, as shown by Sevcik [9], enables most compiler optimizations as they can be observed only by programs with races. At the same time, the C/C++11 model provides escape mechanisms called *low-level atomics* to write high-performance code, with a complex semantics [1].

The latest version of GCC, the forthcoming version of CLang/LLVM, and other commercial compilers, claim to implement the C/C++11 memory model. In this thesis we investigate if and how the random-testing approach of Regehr et al. [10] can be extended to find concurrency bugs in the optimizers of two modern compilers, GCC and CLang/LLVM.

1.1 Random Testing for Compiler Bugs

Currently GCC and LLVM adopt regression testing for finding bugs. Regression testing involves testing against known code-snippets that used to trigger bugs in past revisions, and is effective in guaranteeing that known bugs are not reintroduced by work on the codebase. However it is ineffective in finding unknown bugs.

Regehr et al. successful approach to compiler testing is based on the following three steps:

- generate a random well-defined C program (using a tool called CSmith), that at the end of the execution prints a checksum of its computation;
- compile it with a variety of compilers at different optimization levels;
- compare the output of the executions of the compiled programs: if one compiler/optimization combination ends up with a surprising checksum, then it has a bug.

The naive approach to test the compilation of the memory model, would require to generate randomly interesting *concurrent* programs, compile them and capture *all the outcomes* (as concurrent programs are often non-deterministic), and then compare the set of outcomes. However generating *interesting* concurrent program is non trivial, as the invariants must be subtle to trigger behaviours allowed by corner cases of the specifications. Also, capturing *all* the behaviours of a concurrent program is a difficult task, as it requires having a precise understanding of (and control over) the interesting schedules of the program.

Our Approach Our approach is based on two observations. First, C and C++ compilers must support separate compilation, and the concurrency model states that basically any function can be spawned as an independent thread. As a consequence, compilers must always assume that the sequential code they are optimizing can be run in a concurrent context, and can only apply optimizations which are sound with respect to the concurrency model. Second, it is possible to characterise which optimizations are correct in a concurrent setting by observing how they eliminate or reorder memory accesses in the traces of the sequential code with respect to a reference trace. If combined, these remarks imply that testing the correctness of compilation of concurrent code can be reduced to validating the traces generated by running optimized sequential code against a reference (unoptimized) trace for the same code.

Concretely, we modify the Regehr et al. testing strategy as follows:

- we use a modified version of Csmith to generate a random well-defined sequential program that includes low-level atomic memory accesses and lock/unlock statements;
- we compile the generated program without and with optimizations, and we use binary instrumentation to record the execution traces of the two binaries;
- we compare the two sequential traces and decide if the optimized one can be obtained from the unoptimized by a combination of valid eliminations and reordering of memory accesses;
- if this is not possible, we conclude that the optimizer performed an unsound optimization, and we attempt to perform testcase reduction.

The most interesting and original contribution of this work is the comparison of the optimized trace against the reference trace. For this we extended the theory of safe optimizations of Sevcik [9] to take into account the peculiarities of the C/C++11 memory model (in particular low-level atomic accesses), and we design an algorithm that puts this theory at work and can be used to match traces captured by running realistic programs.

At the time of writing, using our tool we have been able to find and report a subtle concurrency bug in GCC, thus validating the relevance of this approach.

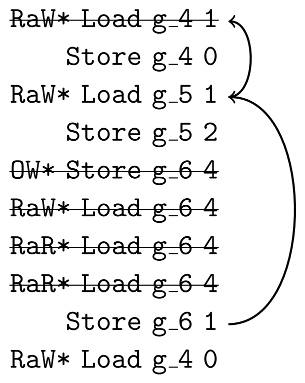
1.2 Example

In this section, we present a basic overview of how our trace matching algorithm works using an example. Consider the following C program.

```
const unsigned int g_3 = 0UL;
long long g_4 = 0x1;
int g_6 = 6L;
volatile unsigned int g_5 = 1UL;
void func_1(void){
    int *l_8 = &g_6;
    int l_36 = 0x5E9D070FL;
    unsigned int l_107 = 0xAA37C3ACL;
    g_4 &= g_3;
    g_5++;
    int *l_102 = &l_36;
    for (g_6 = 4; g_6 < (-3); g_6 += 1);
    l_102 = &g_6;
    *l_102 = ((*l_8) && (l_107 << 7)*(*l_102));
}
int main (int argc, char* argv[]){
    func_1();
    return 0;
}
```

The above program is compiled with and without optimizations, and then we trace all the memory events which take place. The left side represents the unoptimized memory accesses while the right side represents the memory accesses in the optimized code.

```
Init g_6 1
Init g_4 1
Init g_5 1
```



```
 Load g_5 1
Store g_4 0
Store g_6 1
Store g_5 2
 Load g_4 0
```

We try to map all the events in the optimized trace by finding a corresponding equivalent event in the unoptimized trace. We perform valid transformations (eliminations and re-orderings) in the unoptimized trace to obtain the mapping. We discuss the validity of such transformations in Chapter 3. In the above example, the eliminated events are *struck-off* while the reorderings are represented with the help of arrows.

1.3 Organization of the Thesis

In Chapter 2 we recall the technology we use, including binary instrumentation and parsing debugging in information, and we review the C/C++ memory model and the effect of optimizations on concurrent programs. In Chapter 3 we describe the testing infrastructure, the CPPTEST tool, and we explain the trace matching algorithm. In Chapter 4 we report the bug we found in GCC and we discuss the future directions for this research project.

Chapter 2

Background

In this chapter we describe the technology used to record the execution traces, and how we parse the relevant debugging information from a binary. We discuss the Csmith and Delta tools, as our testing framework relies on extensions of this third-party software. We briefly review the new C/C++11 Memory Model and we end with some remarks about the x86 instruction set which are relevant for our work.

2.1 Binary Instrumentation

We built two different tools to record the execution traces. The former is based on the Valgrind [8] binary instrumentation framework, the latter on the Pin [7]. Dynamic binary instrumentation (DBI) is a technique that enables injecting code into an executable to collect run-time information. DBI is not just limited to collecting information but can also perform certain actions (analysis calls) such as to modify the behaviour or outcome of the program execution. Examples of actions include deleting an instruction, modifying the return value of a load instruction, log certain type of instructions (e.g. calls to specific functions, global memory accesses).

There are two major approaches in DBI when adding the instrumentation code. One approach is known as a 'Disassembly and Resynthesise', in which the executable is first disassembled and analysed, then an IR is built, on top of which the instrumentation code is added, and finally the complete code containing both the original application and the instrumentation code is compiled back to the machine level. The other approach is known as ‘Copy and Annotate’, in which the incoming instructions of the executable are copied. Each incoming instruction is annotated with its effects by an instruction querying API, as in Pin. Instrumentation tools use the annotations to guide the instrumentation, and instrumentation code is inter-leaved with the original application binary.

Valgrind uses the D&R approach, whereas Pin uses the C&A approach. Each approach has its own advantages and disadvantages. D&R is heavy-weight, and can add to runtime execution overhead. C&A is lightweight, and can be faster. However the instrumentation granularity in D&R approach is much finer (IR level) giving more control to the user while in C&A approach the instrumentation granularity is at the assembly instruction level. However, Pin provides various useful levels at which instrumentation can be inserted. These include, but not limited to, instruction level, routine level, Basic Block level.

2.2 Extracting Information from Binary

Our testing framework is specific to the x86 architecture, and in some cases the binary instrumentation does not give us enough information. In these cases, we extract the required information directly from the ELF binary.

2.2.1 Initialization values

Initialization values of all variables are stored in the `.data` section of the executable, except those which are declared as constants or are zero-initialized. To save space zero-initialized variables are stored in the `.bss` section while the constant variables are stored in the `.rodata` section. Variables declared as `static` are harder to trace as they can have namespace clashes, because they are valid only in their scope and cannot be destroyed even if they go out of scope during execution. Compilers usually convert the static variables in a token called *stab* which stores the namespace where the actual variables belongs. GCC and Clang both have an option (`-fno-zero-initialized-in-bss`) which prevent the zero initialized variables to be stored in the `.bss` section which makes parsing easier.

2.2.2 Type Information

We parse the debugging information stored in binaries to extract certain information such as variable names, variable size, array type, struct information. The debugging information is stored in the `.debug_info` section of the executable. The original source must be compiled with the `-g` attribute to tell the compiler to add the debug information.

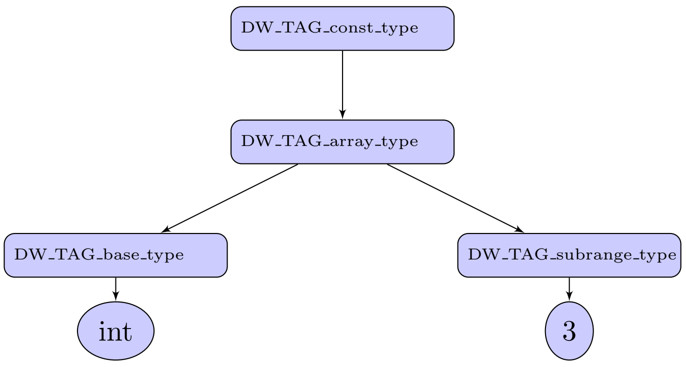
There are several debugging formats namely stabs, COFF, PE-COFF, OMF, IEEE-695 and DWARF, to name some of the common ones. GCC and Clang by default use DWARF version 2 format. DWARF uses a series of debugging information entries to define a low-level representation of a source program. Each debugging information entry is identified by a tag and contains a series of attributes. The tag specifies the class to which an entry belongs, and the attributes define the specific characteristics of the entry. Some example tags are :

- `DW_TAG_base_type`: A base type is a data type that is not defined in terms of other data types.

- `DW_TAG_variable` : Program variables are represented using this tag. Its attributes include `DW_AT_name`, which contains the string name as defined in the source program. It has other attributes like `DW_AT_type` which contains the type of the variable, `DW_AT_external` flag which tells whether the variable is a global or a local variable.

All this information is stored in a tree format with each entry assigned an id. For example:

```
const int g_1[4];
```



We critically rely on the debugging information for the proper tracing of array variables and structs. This is mainly because it is not possible to infer the addresses of struct fields by parsing the source file. It becomes even more difficult when compiling bitfields as they might be packed differently by different compilers. We can also extract the offset and padding of various fields of the structs.

The debugging information is stored in a highly complex and optimized way, hence decoding it is non trivial. There are several tools which are able to dump the debug information and certain sections of the executable which we can parse to get the required output.

Parsing the `.debug_info` section is much harder that parsing the `.data`, `.rodata`, or `.bss` sections. For these we use `objdump` and `readelf` to dump the symbol table and sections of the executable to obtain the initialization values. Both come as a part of standard binutils library. For the `.debug_info` section, at the time of writing, we rely on a naive tool that dumps the section in a human readable format, which we then re-parse. However the naive tool is not adapted to dump information on bitfield structures, and we are now writing our own dwarf parser.

2.3 Csmith

Csmith (http://embed.cs.utah.edu/csmith/) is a tool that generates random C programs that statically and dynamically conform to the C99 standard. In particular, programs generated by Csmith do not have any undefined or compiler dependent behaviour. Csmith proved useful for stress-testing compilers, static analyzers and other tools that process C code. It generates C programs with thousands of lines and complex expressions. The programs generated by Csmith follow these conventions:

- global variables begin with `g_[id]`;
- atomic variables begin with `a_[id]`;
- local variables begin with `l_[id]`;
- function names begin with `func_[id]`.

2.4 Delta

Delta (http://delta.tigris.org/) is a test-case reducer: given a program and a predicate on the program, it attempts to find the smallest sub-program of the input that does not satisfy the predicate. As such it is extremely useful when attempting to isolate a small failure-inducing snippet in a large program: in our case, the predicate tests for the failure (usually the presence or absence of a given bug). In our case, our testing framework is the predicate: whenever a Csmith generated program does not pass the testing, we run it within Delta, looking for the smallest subprogram that still does not pass the testing. Technically, Delta simply tries to remove out chunks of code from the testcase; it is not guaranteed to find the smallest sub-program, and in some cases it can introduce undefined behaviours. However it was fundamental to make sense of testing failures, as directly making sense of the large traces of the programs generated by Csmith is simply not possible.

2.5 The C/C++11 Memory Model

Originally C and C++ were designed without thread support; threads were available through a separate library. This turned out to be a bad design [4]. The latest revision of the C++ standard explicitly defines a concurrency model [2] [3]. The model is basically a so-called DRF model: programs that do not exhibit *data-races* (that is, programs which are well-synchronized) are guaranteed to expose only sequentially consistent executions (that is, executions obtained by interleaving the actions of each thread). A *data race* occurs when two or more threads in a single process access the same memory location concurrently and at least one of the accesses is a write. For instance, in the program below

```
int x = 0;
void foo(){ x = 2;}
void bar(){print x;}
int main() {
  thread t1(foo);
  thread t2(bar);
  t1.join();t2.join();
  return 0;
}
```

thread `t1` performs a write to `x` while thread `t2` performs a read of `x`, and these two accesses are not synchronized (for instance with lock/unlock or other mechanism). According to the standard, data-races make the program behaviour *undefined*.

The C++11 standard also features a complex escape mechanism, called *low-level atomics* for expert use; this escape mechanism is useful for high-performance low-level programming. Atomic objects are of integral type (byte, int, long, etc) and operations on atomics (reads, writes and read-writes) do not race with each other, so in the above program changing `x` to be *atomic_int* would make the program race-free. By default, the behaviour of atomic operations is as one would expect in a sequentially consistent semantics. However, to allow high performance code, the atomic operations include weaker variants and are parametrised by a memory order which specifies how much synchronization and ordering is required.

The strongest ordering is *memory_order_seq_cst*, which provides a sequentially consistent semantics, and the weakest is *memory_order_relaxed*, which is usually compiled as a hardware memory access and as such provides weak ordering guarantees; these are defined for both loads and stores. In between there are *memory_order_release* and *memory_order_acquire* semantics for stores and loads respectively. To understand the semantics of these attributes, consider the following implementation of a common message passing pattern. Here one thread writes some data into a large object `x` and then sets a flag `y` while the other spins until the flag is set and then reads the data.

```
                        atomic_int y = 0
// sender                               // receiver
x = ...                                 while (0 == y.load(memory_order_acquire);
y.store(1,memory_order_release);        r = x;
```

The *memory_order_release* semantics ensures that all the writes before it are propagated to the memory before the store of $[y = 1]$ is visible. The *memory_order_acquire* acts as a barrier that prevents other memory operations from being moved above it. Hence the receiver is guaranteed to see the data writes of the sender. Observe that the program above is not racy, and no expensive lock/unlock synchronization have been required.

2.6 Remarks about the x86 instruction set

Our tracing tool support all the x86 instructions targeted by GCC and Clang. For our purposes, instructions can be classified according to the memory operations they perform. If we ignore the instructions with string operands and rep prefixes, any given memory instruction can either perform no memory access, or perform a memory write, a memory read, or a memory read and write to a single memory address. To the best of our knowledge, no instruction performs a memory read and write to different addresses in the same instruction.

String instructions are more complicated, as two memory reads of different addresses can be observed when comparing two strings using string instructions. These instructions are generally used in conjunction with rep (repeat) prefixes. The repeat prefixes tell the processor to do a multi-byte string operation. We encountered the x86 string instructions when manipulating arrays. We observe that if the local array comprises of a few elements, the compiler puts each element of the array elements sequentially on the stack. However if the size of local arrays is large, to reduce the code size, the compiler puts the entire array in the `.rodata` section and then puts a string instruction with rep prefixes to perform multiple move instructions on the stack. Some precautions were taken instrument instructions with rep prefixes using Pin, as the above instructions are reported to perform a memory read and write at different addresses.

Chapter 3

The CPPTEST tool

In this chapter we describe the design and implementation of our testing tool. A typical usage is

```
 $ cpptest -generate c -locks -trace -analyse
```

which causes cpptest to generate a C program with locks, compile it at different optimization levels, trace the binaries, and then try to match the unoptimized and optimized traces. The tool returns *true* if the traces match successfully, and *false* otherwise.

The tool has a rich interface, for instance, instead of generating a C/C++ program, it can also perform the analysis on a list of input C/C++ files or traces, it supports several timeout options, it can invoke the testcase reducer, and can loop until it finds a program for which the matching fails. For reference, below we report the list of options accepted by the tool.

```
usage: cpptest [options] <file1> .. <filen>
  -generate <c|cpp>       generate a c/cpp file via Csmith
  -expr_complexity <n>    set Csmith expr_complexity to n
  -timeout <n>            set timeout for analyse (in sec)
  -max_funcs <n>          set Csmith max_funcs to n
  -atomics                generate cpp code with low-level atomics
  -locks                  generate c code with locks
  -trace                  trace the input files
  -tool <pin|valgrind>    specify how to instrument the binary
  -asm                    generate the optimized and unoptimized assembly
  -dump_traces            dump traces
```

```
-baseline               compare the reference traces generated by GCC and LLVM
-analyse                run the analyser on the generated file
-count_only             just compare the no. of atomic actions during analysis
-delta                  run delta if analyse fails
-timeout_delta consider a timeout as a failing analysis for the purpose of delta
-no_relax_size          forbid size relaxation from unopt to opt when matching traces
-debug output           some debugging information
-clean                  clean up if matching returns true
-quiet                  disable the gcc/llvm warnings and errors
-repeat                 compatible only with -generate, repeats until the analyse
                        returns false for the generated program
-unsound                ignore loads used in jumps and do not mark as relevant
-no_arrays              disable arrays when generating programs using Csmith
-compiler               <clang|gcc> specify compiler
-help                   Display this list of options
```

In what follows we formalise the execution traces, we describe our tracing algorithm and program analysis, and then we describe the processing we perform on the traces to decide if they match or not.

3.1 Setup

Definition 3.1.1. *An action can be one of the following :*

- *a load/store of a memory location, specified by the value and size being read or written;*
- *a load/store of an atomic location, specified by the value, size being read or written and the atomic attribute;*
- *a lock or unlock specified by the mutex location, or a memory flush.*

We refer lock, unlock, flush, and release/acquire events as synchronization actions.

Definition 3.1.2. *A trace is a finite sequence of actions. All actions in a trace belong to the same thread.*

Definition 3.1.3. *A traceset is a prefix closed set of traces.*

Definition 3.1.4. *An interleaving is a sequence of thread id and action pairs. Given an interleaving we can reconstruct the trace of each thread id. Conversely, given a traceset, we can build all the interleavings of the traceset.*

Definition 3.1.5. *An interleaving is sequentially consistent if each read sees the most recent value written in the interleaving. A sequentially consistent (and well-locked) interleaving is called an execution.*

A traceset denotes a program, by enumerating all the traces that can be performed by each thread. We focus on sequential programs, and ensure that the optimizers apply only optimizations correct in a potentially concurrent environment, as characterised by Sevcik [9].

As an example, here is a simple program and a possible trace.

Program

```
int x,y = 0;
void foo (){
  for(int i = 0;i < 2;i++){
    y = x++;
  }
}
```

Trace

Op	var	size	value
Init	x	4	0
Init	y	4	0
Load	x	4	0
Store	y	4	0
Store	x	4	1
Load	x	4	1
Store	y	4	1
Store	x	4	2

3.2 Capturing the traces

Memory Accesses We use Pin to capture the execution traces of a binary. We instrument each memory load and store, fence instruction, and instruction with lock prefix, as well as pthread library functions. We are only interested in accesses to potentially shared global memory, so we must filter out all the stack accesses (although a program might copy the address of a local stack-allocated variable into a global location, our generated programs do not, so we can safely ignore stack accesses). On x86, a stack access can be easily identified by checking if the stack pointer register (`esp`) or the base pointer register (`ebp`) is used in the address computation. However due to the complexity of x86 instruction set, this analysis is not complete. For example consider the following example

```
int g_1 = 0;


void foo(){
  int *l_1 = g_1
}
```

and a common compilation (usually performed by GCC):

```
1. lea [EAX] <- offset[ESP];
2. mov [EBX] <- g_1;
3. mov (EAX) <- [EBX];
```

1. `lea` stands for "load effective address". It computes the stack address (`offset + esp`) and stores them in the `eax` register;
2. load of `g_1` to register `ebx`;
3. store of `ebx` into the address contained in register `eax`.

The last instruction effectively does a stack write but does not directly involve `esp` or `ebp` in the instruction. As such, Pin doesn't identify it as a stack write, and reports it as a global store. We add a mask to identify these accesses in our analysis routines.

Fences and Lock Prefix A lock prefix causes the processor's `LOCK` signal to be asserted during execution of the accompanying instruction, i.e it turns the instruction into an atomic instruction. In a multiprocessor environment, the `LOCK` signal ensures that the processor has exclusive use of any shared memory while the signal is asserted.
An x86 instruction can have up to 4 prefixes each 1 byte long, in any order.

- Prefix Group 1

  - 0xF0: LOCK prefix
  - 0xF2: REPNE/REPNZ prefix
  - 0xF3: REP or REPE/REPZ prefix

- Prefix group 2
  - 0x2E: CS segment override
  - 0x36: SS segment override
  - 0x3E: DS segment override
  - 0x26: ES segment override
  - 0x64: FS segment override
  - 0x65: GS segment override

- Prefix group 3
  - 0x66: Operand-size override prefix

- Prefix group 4
  - 0x67: Address-size override prefix

The lock prefix can occur in any of the 4 bytes. Note that the size of x86 instructions are not fixed, nor is the prefix length. Also since `xchg` instructions are locked by default, we also treat them as locked instructions. They are relatively easy to detect as their first byte is always `0x87`. In our traces and trace matching, we do not yet support natively locked actions, so we log these as the accesses they perform followed by a `mfence`.

We explicitly trace the `mfence` instructions as they act as synchronization operations. They always begin with `0xAE0F` as their prefix.

Pthread Calls We also trace the `pthread_mutex_lock/unlock` calls. This is straightforward except that certain `.so` files used when linking the executable might also contain calls to these functions: we have to be careful not to instrument them except which are present in the user code. We thus instrument all the pthread library calls but maintain a dedicated flag (`RUN_ANALYSIS`), and the analysis logs the function calls only if the flag is set. The flag is set when we first enter the `main` function and is unset when we leave `main`. This ensures that none of the library specific calls are traced.

To obtain the mutex location, we simply log the argument which is being passed at the called function (following the x86_64 ABI, this can be done by reading a specific register).

3.3 Computing the IR Set

3.3.1 Introduction

We will formalise *irrelevant reads* in Section 3.4, but, intuitively, a read is irrelevant if the returned value is not used by the program to perform an observable action. Equivalently, the value read does not influence the behaviour of the program. Compilers try hard to remove irrelevant reads. For instance, consider the following example:

```
int g_1  = 0;
{
  int l_1 = g_1;
  ...           //no use of l_1 after this
}
```

The load of `g_1` is irrelevant and it is sound to remove such an access while optimizing. Interestingly, in some cases, irrelevant reads are introduced (rather than removed) by the optimizer as it attempts to pre-fetch some values that are possibly never used.

Irrelevant reads cannot be characterised by looking only at one execution trace, so we must implement a program analysis to identify them. More in detail, we perform a liveness analysis on the loads performed and we (attempt to) check whether the load is actually affecting the global memory state.

The idea behind finding irrelevant reads is to find those reads which do not change the state of global memory. This can be performed by tracing each load and recording the destination registers of each load. We keep tracking each target register until either it is overwritten, which implies that the previous load was not used (and thus irrelevant), or the register is stored into some global memory location (and thus the read is relevant). The above analysis performs well on a simple instruction set, but the complex nature of x86 instructions requires accounting for all the possible ways in which a value being read can be propagated.

3.3.2 Information Flow

The information from an instruction can propagate through registers, stack or both.

Registers We extract all the operands involved in any given instruction. We then separate out the registers involved from the set of operands. An operand can be (but are not limited to) an immediate operand or a memory address (stack or global). For x86 instruction set, we can also have certain implicit registers involved which are not encoded in the instruction itself but through which the effect of load can be propagated forward. For example:

```
mov   [EAX] <- g_1
xor   [EAX], [EAX]    //flags are affected
cmove [EBX] -> g_2
```

In the above example, the load of `g_1` is used to decide whether we can write to `g_2` (if the flag `ZF` is non zero) and hence is relevant.

Also we must be careful with instructions in which an address is computed. For example in x86, while calculating performing operations involving array indexes, we generally have a base and an index register which are implicitly used in the address computation when performing an operation on an array index.

Stack Operations It is also necessary to trace stack operations because a register can be spilled to the stack, which is then reloaded and used to perform some operation which writes to global memory. The simplest example is

```
int g_1 = 0;
int g_2 = 0;

{
 ...
 int l_1 = g_1;
 g_2 = l_1;
 ...
}
```

A naive compilation would be

```
MOV [EAX] <- g_1
MOV offset(ESP) <- [EAX] //stack write
MOV [EBX] <- offset(ESP) //stack read
MOV g_2 <- [EBX]
```

and the read of `g_1` is propagated to `g_2` using stack write followed by a stack read.

3.3.3 Data Structures

The algorithm performs IR analysis on the executable and returns a list of integers which are indices of irrelevant loads in the generated trace. We explain the various data structures used for performing the analysis.

- *indexCount* maintains the index of the current event. The analysis return a list of indexes of load events which are irrelevant.
- *usedIndexSet* is a set of indexes of load events which affect the global memory state.
- *loadIndexSet* is the set of indexes of all loads events.
- *dependenceTable* maps a register and a stack address to the list of load events on which the current value depends.
- *loads2Stores* maps a load event to the set of store events for which the load was used.
- *ReadSet* stores the registers read by an instruction. This can be computed during instrumentation i.e. before the instruction is executed.
- *WriteSet* stores the registers written by an instruction. It can also be computed during instrumentation. A register can appear in both the sets.
- *AddressGenerated* stores the memory address (stack or global) which will be accessed by an instruction. This cannot be pre-computed as stack operations depends on the `esp` register.

3.3.4 Analysis

In this section we discuss the analysis for computing the irrelevant reads and provide pseudo code for the various cases which need to be handled.

Depending on the type of each instruction and the way (registers and/or stack) information can flow, we compute all the loads used in it. We call this set to be *dependentLoads* (see Algorithm 3.3.1). Once the analysis is complete for the instruction, we propagate the loads forward by setting every register in the write set to be dependent on the *dependentLoads* in the *dependenceTable* (see Algorithm 3.3.2).

Algorithm 3.3.1: *GenDependentLoads*()

```
$dependentLoads \leftarrow \emptyset$
for all $reg \in readSet$ do
  for all $i \in dependenceTable[reg]$ do
    $dependentLoads \leftarrow dependentLoads \bigcup i$
  end for
end for
if $stackRead$ then
  for all $i \in dependenceTable[stackAddress]$ do
    $dependentLoads \leftarrow dependentLoads \bigcup i$
  end for
end if
```

Algorithm 3.3.2: *PropagateLoads*()

```
for all $index \in writeSet$ do
  if $|dependentLoads| > 0$ then
    $dependenceTable[index] \leftarrow dependentLoads$
  else
    $dependenceTable.erase(index)$
  end if
end for
```

We now discuss the algorithms for handling different classes of instructions based on the type of memory operation they perform.

Memory Read & Memory Write We consider two cases. If the memory is a stack address, then it is added to the *dependenceTable* and the loads are propagated. Else, if it is a read-write to a global location, then all the indices in *dependentLoads* are relevant.

Also we add the store event to all the corresponding loads in the *loads2Stores*.

Algorithm 3.3.3: *MemoryRW*()

```
if not $stackWrite$ then
  $loadIndexSet \leftarrow loadIndexSet \cup indexCount$
  $dependentLoads \leftarrow dependentLoads \cup indexCount$
  for all $index \in dependentLoads$ do
    $usedIndexSet \leftarrow usedIndexSet \cup index$
    $loads2Stores[index] \leftarrow loads2Stores[index] \cup (indexCount + 1)$
  end for
else
  $dependenceTable[stackAddress] \leftarrow dependentLoads$
end if
$PropagateLoads()$
```

Memory Write If it's a stack write, then we propagate the address in the *dependenceTable* else we include this store event in *loads2Stores* for each load event in *dependentLoads*.

Algorithm 3.3.4: *MemoryW*()

```
if not $stackWrite$ then
  for all $index \in dependentLoads$ do
    $usedIndexSet \leftarrow usedIndexSet \cup index$
    $loads2Stores[index] \leftarrow loads2Stores[index] \cup (indexCount)$
  end for
else
  if $|dependentLoads| > 0$ then
    $dependenceTable[stackAddress] \leftarrow dependentLoads$
  else
    $dependenceTable.erase(stackAddress)$
  end if
end if
$PropagateLoads()$
```

Memory Read If we have a stack read then we just propagate the loads. Otherwise, we add *indexCount* to the *dependentLoads*.

Algorithm 3.3.5: *MemoryR*()

```
if not $stackRead$ then
  $loadIndexSet \leftarrow loadIndexSet \cup indexCount$
  $dependentLoads \leftarrow dependentLoads \cup indexCount$
end if
$PropagateLoads()$
```

The main analysis function, called *IRAnalysis*, combines all the routines discussed above. This function is called for each x86 instruction during the instrumented execution. In the end, ($loadIndexSet \setminus usedIndexSet$) gives the set of irrelevant reads.

Algorithm 3.3.6: *IRAnalysis*()

```
$GenDependentLoads()$
**if** $memoryRead$ & $memoryWrite$ **then**
  $MemoryRW()$
**else if** $memoryRead$ **then**
  $MemoryR()$
**else if** $memoryWrite$ **then**
  $MemoryW()$
**else**
  $PropagateLoads()$
**end if**
```

3.4 Eliminable Actions

An eliminable event is an event that can be removed without introducing any new visible behaviour in any concurrent context with respect to the reference trace. In this section we describe a list of sound rules to detect eliminable events.

Read after Read (RAR) A read of a variable is a redundant read after read if it follows a read of the same variable with the same value , and provided that there is no write to the variable and/or no synchronization occurs in between the two reads. This transformation is often performed as a part of common sub-expression elimination or constant propagation optimizations in compilers.

```
Load g_1 42
...             //no access to g_1 and no synchronization
Load g_1 42    (*RAR*)
```

Read after Write (RAW) A read of a variable is a redundant read after write if it follows a write of the same variable and reads the same value as the previous write, and provided that there is no other access to the same variable and/or no synchronization occurs between the two events.

```
Store g_1 42
...             //no access to g_1 and no synchronization
Load g_1 42    (*RAW*)
```

Write after Read (WAR) A write of a variable is a redundant write after read if it follows a read of the same variable and writes the same value as the previous read, and provided that there is no other access to the same variable and/or no synchronization between the two events.

```
Load g_1 42
...             //no access to g_1 and no synchronization
Store g_1 42   (*WAR*)
```

Overwritten Write (OW) A write of a variable is an overwritten write if it precedes a write to the same variable, provided all accesses in between to the same variable are eliminable and/or there are no synchronization between the two events. If an event is marked as OW then it also carries with it a list of eliminable events which when eliminated, makes the write as OW. Two snippets of traces illustrate these two cases:

- Case 1:

```
Store g_1 42  (*OW*)
...           //no access to g_1 and no synchronization
Store g_1 47
```

- Case 2:

```
Store g_1 42  (*OW*)
Load  g_1 42  (*RAW*)
...           //all accesses to g_1 are eliminable and no synchronization
Store g_1 47
```

Irrelevant Read (IR) To understand irrelevant reads, consider the example:

```
int x = 0;
int y = 0;
void foo (){
    y = x+1;
}
```

A naive compilation would look somewhat like

```
mov [EAX] <- x;
add $1, [EAX];
mov [EAX] -> y;
```

The trace generated by the above compilation is:

```
Load  x 0
Store y 1
```

and more in general, for an arbitrary value `v`:

```
Load  x v
Store y (v+1)
```

Now we see that on changing the value of the read value the trace after the load event gets changed: the first load is not irrelevant as it directly effects the value being stored at a global location.

Consider now another example:

```
int x = 0;
int y = 0;
void foo (){
    int l = x;
    y = x+1;
}
```

A naive compilation would look somewhat like

```
mov [EAX] <- x;
mov offset(ESP) <- [EAX];   //stack write of the local l
mov [EAX] <- x;
```

```
add $1, [EAX];
mov [EAX] -> y;
```

The trace generated by the above compilation :

```
Load  x 0
Load  x 0
Store y 1
```

and more in general, if the first load reads an arbitrary value v, we get:

```
Load  x v
Load  x 0
Store y 1
```

In this case, the remaining trace remains unchanged even after modifying the value read by the first load: the first load is irrelevant. IR reads are treated as eliminable events as they do not affect the global state of memory.

This can be formalised as follows: given a trace, we define a read event $A_i(R[x = \alpha])$ as an irrelevant read provided that if A_i is replaced by $A_j(R[x = \beta])$ $s.t$ $\alpha\ ! = \beta$, the remaining trace does not get affected.

3.5 Reorderings

A transformation that changes the order of occurrence of some actions is known as reordering. While optimizing, compilers reorder lots of actions but must do so without affecting the semantics of the programs. Hence compilers can reorder only certain actions with each other.

Next, we define reorderable actions in a trace. The notion of reordering does not imply that the two actions in the trace get swapped. Also this definition of reordering differs slightly with the general notion about reordering as swapping of two events used by Sevcik[9].

Definition 3.5.1. *Two actions A_i and A_j are said to be conflicting if :*

- *A_i and A_j both access the same location and either of A_i or A_j is a store;*

- *Either A_i or A_j is an atomic operation;*
- *either A_i or A_j is a synchronization operation (we ignore the so-called roach motel reorderings).*

Definition 3.5.2. *An action A_j can be moved before A_i in the trace if it satisfies the following properties:*

- *A_i and A_j are both non atomic accesses.*
- *$\forall$ k s.t $i \leq k < j$, A_j does not conflict with A_k.*

Reorderings interact with eliminations; the following examples illustrate sound reorderings and highlight how reordering and eliminations spice up things. Consider for instance

```
 ***unopt***          ***opt***
Load  x 1            Load  y 1
Store x 2    ==>     Load  x 1
Load  y 1            Store x 2
```

The above reordering is correct. However note that the two elements being reordered are not swapped, instead the load of y now precedes the load of x. Clearly swapping the two loads would be incorrect as it violates sequential consistency:

```
 ***unopt***          ***opt***
Load  x 1            Load  y 1
Store x 2     =/=>   Store x 2
Load  y 1            Load  x 1
```

Consider now

```
              Init   g_6 0
 ***unopt***                ***opt***
OW * Store  g_6 1
RaW* Load   g_6 1
RaR* Load   g_6 1   ==>   Store g_6 2
RaR* Load   g_6 1
     Store  g_6 2
```

Clearly the above eliminations are sound. Note that the first store of `g_6` is marked as OW only because all memory accesses until the next store of `g_6` are eliminable. Hence all store action marked as OW also carries with itself the list of possible eliminable actions, which make it an over-written write. In this example the first store to `g_6` is an overwritten write once the redundant reads in the middle have been removed (and should not be considered an overwritten write if the reads are not removed by the optimizer).

A more complex example:

```
                  Init   g_6  0
                  Init   g_11 0
      ***unopt***                ***opt***

  RaW* Load   g_11 0         Load   g_11 0
       Store  g_11 1   ==>   Store  g_6  2
  OW * Store  g_6  1         Store  g_11 1
  RaW* Load   g_6  1
  RaR* Load   g_6  1
  RaR* Load   g_6  1
       Store  g_6  2
```

The last store of `g_6` must be reordered with the store `g_11`, but this is correct only if all the intermediate accesses to `g_6` are removed.

3.6 Irrelevant Action Introduction

We observed irrelevant actions being introduced by the optimizers. Consider the following C program and associated traces generated by Clang:

```
int g_2  = 0;
int g_7  = 0;
int g_90 = 0;
void  func_1(void) {
  for (g_2 = 15; (g_2 != 14); g_2--) {
    if (g_7){}
    else
      if (g_90)
      break;
    }
  }
```

```
                           Init   g_2  0
                           Init   g_7  0
                           Init   g_90 0
           ***unopt***                     ***opt***
       OW * g_2 F 4  Store         IR * g_90 0 4 Load
       RaW* g_2 F 4  Load    ==>   IR * g_7  1 4 Load
       RaW* g_7 1 4  Load               g_2  E 4 Store
       RaR* g_2 F 4  Load
            g_2 E 4  Store
       RaW* g_2 E 4  Load
```

Observe the load of g_90 introduced by the compiler in the optimized trace. Observing such read introductions are quite common and compiler try to pre-fetch the loads for faster processing. Although, introducing such loads is safe in general, they are simply irrelevant (marked as IR). We will take this into account in the matching algorithm.

Consider the following C program and associated traces generated by GCC 4.7:

```
int  g_22[1] = {0x4L};
int  g_167 = 0;
int  g_168 = 1;
int  g_207 = 2;
char g_244 = 0;
int func_1(void)
{
  for (g_167 = 0; (g_167 != 28); g_167++) {
    for (g_244 = (-8); g_244 < (-15); g_244 -= 2);
    if(g_168 &= g_22[0])
      g_12 &= 0L;
    else
      return g_207;
  }
}
```

```
          ***unopt***                     ***opt***
          g_167 0  4 Store             g_167 0  4 Store
     RaW* g_167 0  4 Load              g_244 F8 1 Store
          g_244 F8 1 Store        IR * g_22  4  2 Load
     RaW* g_244 F8 1 Load    ==>  IR * g_168 1  4 Load
     RaW* g_22  4  2 Load              g_168 0  4 Store
     RaW* g_168 1  4 Load              g_167 0  4 Store
          g_168 0  4 Store             g_244 F8 1 Store
     RaW* g_168 0  4 Load
     RaW* g_207 2  4 Load         IR * g_207 2  4 Load
```

Note the two newly introduced stores of g_167 and g_244 which were not present in the unoptimized trace. However introducing such stores is valid because any context that we try to build will always have a data race present. Our tool is currently unable to deal with this case.

3.7 Trace Processing

In this section we describe the details about the representation of traces in *cpptest*. We explain how traces are enriched with initialization values and why and how we identify pointer variables. We also explain about how eliminable events are identified and marked.

Representation Internally a trace is represented as a list of events.

```
type loc = int
type size = int
type value = NonPointer of Int64.t | Pointer of loc


type event =
  | Init of loc * value * size
  | Load of loc * value * size
  | Store of loc * value * size
  | ALoad of atomic_attribute * loc * value * size
  | AStore of atomic_attribute * loc * value * size
  | Flush
  | Lock of loc
  | Unlock of loc

type rr =
  | IRR
  | RAW | RAR
  | OW of int list
  | WAR
  | NotRedundant

type annot_event = {
  evt         : event;
  redundant   : rr ref;
  deleted     : bool ref;
}
```

Initialization Values The traces generate by Pin do not contain the initialization values of global variables, which are necessary to mark the eliminable events. We augment the generated traces to add the initialization values. This information is extracted from the `.data` section of the executable using the `objdump` and `readelf` tools.

Getting the initialisation values is easy for simple types. We need to know the size of variable for which we are trying to find the initial value, because the data is dumped in a compact hex form which can be parsed by taking the address and the size of the variable into account. We extract the size of simple variables from the symbol table stored in the executable.

For arrays, we need to know the type of elements of the array. We cannot extract this from the symbol table because it only stores the total size which is not sufficient for tracing the individual elements of the array. We get this information by parsing the debugging information (explained in 2.2.2).

Handling Pointers Pointers contain the address of a variable, which may differ between the unoptimized and optimized traces as they are compiled separately (and thus the memory layout can be different), but they might point to the same variable. Consider,

```
...
int  g_1 = 0;
int* g_2 = &g_1;
...
```

Now `g_2` contains the address of `g_1` which might be different in optimized and unoptimized compilation. Since we compare values of the variables, this resulted in false positives. Hence the value a variables can hold is either *NonPointer* or *Pointer* type.

```
type loc = int  // entry in the symbol table
type value = NonPointer of Int64.t | Pointer of loc
```

The pointer variables are identified in the following way. In the first pass, all addresses are mapped to the corresponding variables. In the second pass, for each variable if the initialization value corresponds to an address of any variable, the current variable is marked as *Pointer* type.

Marking Eliminable Events The events are processed to convert them into *annot_event*: eliminable events are now annotated with the corresponding redundancy annotation (described in Section 3.4), or marked as *Not_Redundant.* For unoptimized traces, we consider all the eliminable events, while in the optimized traced we mark only the IR as eliminable. This enables us to take into account irrelevant read introductions.

Annotating IR events is easy, as the analysis in performed in our pintool, which returns the list of indices of the load events which are irrelevant. We simply mark those events as IR.

The other eliminable events are marked by performing a simple data-flow analysis: for each trace point we keep a track of details of the last access made to each location. These details include last value of the location, type of access (Store, Load or Init), and the size of the access. This is sufficient for marking all eliminable events except OW.

We store the information about the previous access made to a variable in a map called *info_var.* It contains the size, value and type of the relevant event mapped to the location.

Init events are handled as normal stores, they are inserted in the *info_var* table with their initial size and initial value of the variable.

For a store event, if there is no previous entry for the variable, then it is just added to *info_var*. Otherwise, there are two outcomes depending on whether the previous entry was a load event or a store event. If the previous access was a load, then we need to compare the values from both the accesses. If the values are different, then we just insert the new entry replacing the old entry. Otherwise, the current store is marked as redundant write after read (WAR) and the previous access is marked to be a store in *info_var*. Note that we need to take into account the size of accesses when comparing the values.

Algorithm 3.7.1: *Store*$(S(loc, value, size))$

```
if $info\_var[loc]$ then
  $(oldS, op, oldV) \leftarrow info\_var[loc]$
  if op = R then
    if oldS $\geq$ size then
      if $compare\_val(value, size, oldV, oldS)$ then
        $S \leftarrow WAR$
        $info\_var[loc] \leftarrow (oldS, W, oldV)$
      else
        $info\_var[loc] \leftarrow (size, W, value)$
      end if
    else
      $info\_var[loc] \leftarrow (size, W, value)$
    end if
  end if
else
  $info\_var[loc] \leftarrow (size, W, value)$
end if
```

For load events, if we do not find any previous entry for the variable, then we just add an entry for the event. Else, there are two possibilities. First if the previous event was a load then the current event is marked as a redundant read after read (RAR) and if it was a store then the current event is marked as a redundant read after a write (RAW); in both cases the new event replaces the previous entry for this location. Note that comparison of values is not required in this case as we test sequential programs and hence if there is any previous entry the program cannot read any other value. However the size of access must be taken into account.

Note that while all eliminable events except OW can be marked by a forward processing of the traces, for marking OW we need a second pass over the trace (for examples see Case 2 of the OW paragraph in Section 3.4). In the first pass we mark all the eliminable events

Algorithm 3.7.2: *Load*$(L(loc, value, size))$

```
if $info\_var[loc]$ then
  $(oldS, op, oldV) \leftarrow info\_var[loc]$
  if oldS $\geq$ size then
    if op = R then
      $L \leftarrow RAR$
    else
      $l \leftarrow RAW$
      $info\_var[loc] \leftarrow (oldS, R, oldV)$
    end if
  else
    $info\_var[loc] \leftarrow (size, R, value)$
  end if
else
  $info\_var[loc] \leftarrow (size, R, value)$
end if
```

except overwritten writes. In the second pass we know the eliminable events and hence mark the special case discussed above. An OW write also contains a list of indices of the integers which represent the events which need to be eliminated in order to make the event as OW. To compute this we scan the trace, and whenever we find a store followed by redundant reads and another store to the same location (and possibly other actions at different locations), we mark the previous store as OW with the list of indices of redundant reads in between.

The dataflow can be killed by synchronization actions such lock or unlock or release/acquire pairs; in these cases we simply delete the *info_var* table.

3.8 The Trace Matching Algorithm

After the pre-processing, we get two traces, represented as two lists of annotated events. One trace is obtained from the unoptimized compilation i.e. with `-O0`, and the other from optimized compilation i.e. with `-O2` or `-O3`. Note that while the unoptimized trace is annotated with all the redundant annotations mentioned in Section 3.4, the optimized trace is annotated only with irrelevant reads.

We recursively try to match the optimized trace with the unoptimized trace, i.e. finding the same events in the unoptimized trace as in the optimized trace. After the matching is complete, no ineliminable events must be left in the unoptimized trace. While performing the search for events we also check for valid reorderings across reorderable events as discussed in Section 3.5.

3.8.1 Comparing Events

Two events are said to be equal if they have the same value, they operate on the same location and they perform the same operation. However there are cases in which a compiler may make different size accesses which refer to the same event. For example, consider

```
int g_1 = 0xB4E1A362;
int g_2 = 0;

char func_1(){
  return g_1;
}

int main(){
  g_2 = func_1();
  return 0;
}
```

The generated traces[1] are as follows:

```
                    g_1  B4E1A362 4 Init
                    g_2         0 4 Init


            ***unopt***                          ***opt***
       RaW* g_1 B4E1A362 4 Load          RaW* g_1 62 1 Load
            g_2 62       4 Store              g_2 62 4 Store
```

Notice the difference in the size of load of `g_1`: in unoptimized trace, the compiler reads 4 bytes of data but uses only 1 byte (via register manipulation) whereas the optimizers inlines the code and infers that only 1 byte of data will be used, and hence performs a load of only 1 byte.

[1]using GCC 4.7 with -O0 and -O2

Traces do not carry enough information to establish if the two loads are both correct, and we consider such events to be equivalent (at the risk of having an unsound analysis). When comparing two events which perform the same action and operate on same location but with different sizes, we compare values only for the minimum of the two sizes. To reduce the risk of unsound analysis, we allow this size relaxation only when matching the unoptimized event against the optimized event, not vice versa.

Algorithm 3.8.1: $CompareVal(x, size\_x, y, size\_y)$

```
if $size\_x = size\_y$ then
  return $x == y$
else if $size\_x < size\_y$ then
  $mask \leftarrow (1 << (size\_x * 8)) - 1$
  return $(y \& mask) == x$
else
  $mask \leftarrow (1 << (size\_y * 8)) - 1$
  return $(x \& mask) == y$
end if
```

3.8.2 Matching Traces

We try to find a mapping between all the events in T_{opt} to the events in T_{unopt}. If a matching exists, the algorithm returns true, else, returns false. After the mapping has been established, all the events remaining in T_{unopt} should be eliminable, and the mapping should only reorder reorderable events. To take into account irrelevant read insertion, the matching can be partial and ignore the eliminable events in T_{opt}.

The algorithm proceeds in a natural, recursive, way. We describe it here, ignoring several corner cases, to give an insight of its inner working.

The algorithm picks up the first element from T_{opt} and tries to match it with the first element of T_{unopt}. If both events are equal, then it recursively continues to match the remaining elements from both traces. If this fails, then it backtracks and tries to eliminate the first element of T_{unopt}, and to match the remaining elements. Else, the traces cannot be matched.

If the events being compared initially are different, then again there are two possibilities to find a matching. If the first event in T_{unopt} is eliminable then eliminate it and recursively continue to match the remaining elements in both traces. If this fails, we attempt to reorder events in the unoptimized trace. For this, the algorithm searches in T_{unopt} for an event equal to the first event of T_{opt} which is reorderable with the first event of T_{unopt}. Recall that two events in a trace are said to be reorderable if there are no conflicting events in between them (see 3.5.1 for details). If a reorderable event exists, we move it to the top of T_{unopt} and we continue to match the reordered T_{unopt} against T_{opt}.

If at the end, all the events in T_{opt} are matched to T_{unopt} and all remaining elements in T_{unopt} are eliminable, then the matching is successful. The pseudo-code for the algorithm is reported in the figures below, with the main entry point being the *matchTrace* function. It outlines only the major cases considered, ignoring some corner cases.

It is not clear which is the best order to search either for an elimination or a reordering. A simple heuristic we implemented is to prefer elimination when traces differ greatly in size, and reordering instead.

We also remark that, unlike what happens with other eliminable events, overwritten writes require a special treatment. Consider the traces:

```
                    g_1  0 8 Init

      ***unopt***                        ***opt***

   OW* g_1 2 4 Store

   OW* g_1 3 4 Store                   g_1 2 4 Store

       g_1 2 4 Store
```

Intuitively the above traces match. However the first and last events in T_{unopt} are not reorderable, so the algorithm described above would fail. Hence when searching for a store, we return the entire list of stores i.e. until we find the first non redundant store to that location. This is detailed in *findOwList.*

The algorithm described above has exponential complexity in the size of the T_{unopt} trace. In practice, heuristics allow it to match traces of hundred of events and in some cases much bigger traces.

3.9 Other tool support

3.9.1 Extending CSmith

We extended Csmith to generate sequential programs that include atomic variables and low-level atomic accesses. We also extended Csmith to generate sequential programs with pthread mutex locks. This is not entirely trivial as we need to ensure that the locks are well balanced. For example preventing programs which re-acquire the lock on a mutex. Also this has to be ensured across *for* loops and the conditional statements. For example, the operations performed on the mutex within an *if* statement should be identical to the operations performed on the same mutex inside the matching *else* block. Also we must be careful when propagating locks across loops.

To implement this, we augmented the context information already built inside Csmith by adding a mutex state . The mutex state contains the current state of the mutex variables. It also records the operations performed on the mutex variables in the current block, for example inside an *if* block. This information should be passed on when generating the *else* block and it must be ensured that the final context after the *if* and *else* blocks are the same, as only one of them will execute at any given time.

Algorithm 3.8.2: findOwList(i, T)

```
Input  : trace $T$ with index $i$ of an OW type
Output: a list of redundant store events until we find first non redundant store
        which can match instead $T^i$
$j \leftarrow (i + 1)$;
$res \leftarrow i$;
while ($T(j)$) do
    if $T(j).loc = T(i).loc$ && $T(j).op = Store$ then
        if $T(j).redundant = None$ then
            if $T(i).val = T(j).val$) then
                return $res \cup j$
            else
                return $res$
        else
            $res \leftarrow res \cup j$
return $res$
```

Algorithm 3.8.3: Search(i, e, T)

```
Input  : trace $T$ with index $i$ and an event $e$
Output: searches for an event $e$ in $T$ which can be reordered with $T^i$
$j \leftarrow (i + 1)$
while $j \leq End_T$ do
    if $T^j \neq e$ then
        $j \leftarrow (j + 1)$
    else
        if $T^j.redundant = OW$ then
            $owList \leftarrow$ findOwList(j,T)
            $(res, elimSet) \leftarrow$ Validate($owList, j, T$)
        else
            $(res, elimSet) \leftarrow$ Validate($i, j, T$)
        if $res$ then
            return $(j, elimSet)$
        else
            return $(None, \emptyset)$
```

Algorithm 3.8.4: Validate(i, j, T)

```
Input  : trace $T$ with indices $i$ and $j$
Output: checks if $T^j$ can be reordered with $T^i$ with possible eliminations
$deleted \leftarrow \emptyset$;
for $k = i$ to $j - 1$ do
    if $conflicting(T^k, T^j)$ then
        if $T^k.redundant = None$ then
            return $(False, \emptyset)$
        else
            $deleted \leftarrow deleted \cup k$
return $(True, deleted)$
```

3.9.2 Changes to Delta

Delta performs test-case reduction by removing chunks of lines of the input program. As such it completely ignores the semantics of the program, and might generate code that hits one of the many undefined behaviours of the C language, making the reduced test-case incomprehensible and unusable. Among the most common wrong reductions, there was the removal of return statements and initialization of arrays. We modified Delta so that it considers only those subsets of code which contain all the return and initialization statements.

Algorithm 3.8.5: matchTrace(i, j)

```
Input  : index $i$ of $T_{unopt}$ and index $j$ of $T_{opt}$
Output: matches $T_{unopt}$ and $T_{opt}$ starting from index $i$ and $j$ respectively
if $T^i_{unopt}$ is deleted then
    return matchTrace($i + 1, j$)
if $i \leq End_{T_{unopt}}$&& $j \leq End_{T_{opt}}$ then
    if $T^i_{unopt} = T^j_{opt}$ then
        return matchTrace($i + 1, j + 1$)
        $\vee$ (eliminable $T^i_{unopt} \wedge$ matchTraces($i + 1, j$))
    else
        (eliminable $T^i_{unopt} \wedge$ matchTrace($i + 1, j$))
        $\vee$
        (($res, elimSet$) $\leftarrow$ search($i, T^j_{opt}, T_{unopt}$)
        if $res \neq None$ then
            mark $T^e_{unopt}$ as deleted $\forall\ e \in elimSet \cup res$
            matchTrace($i, j + 1$)
        )
```

Chapter 4

Assessment and Perspectives

4.1 Assessment

In the last month we have been running *CPPTEST* continuously on an Intel Xeon workstation. The tool is robust enough to run unattended for days and leaves the generated programs for which the trace matching fails in a directory for subsequent manual investigation.

In our testing we could replicate several already known bugs of GCC and clang; for example our tool was able to detect the bug (http://llvm.org/bugs/show_bug.cgi?id=12189) which was already reported by Regehr et al. More interestingly, we found a previously unknown concurrency bug in GCC 4.7. The trace matching algorithm does not yet deal with all the complexity and subtleties encountered in trace optimizations, and in some cases it fails to match equivalent traces. Also, the trace matching algorithm has exponential complexity in the worst case, and it may fail to match traces with about one thousand memory accesses.

A GCC 4.7 bug: write introduction is incorrect in the C/C++11 Memory Model The program below:

```
int g_1 = 1;
int g_2 = 0;
int func_1(void) {
  int l;
  for (l = 0; (l != 4); l++) {
    if (g_1)
      return l;
    for (g_2 = 0; (g_2 >= 26); ++g_2)
      ;
  }
}
int main (int argc, char* argv[]) {
 func_1();
}
```

is miscompiled by GCC version 4.7

Observe that the inner loop of `func_1` is never executed, and this program should never perform any read/write to `g_2`. This means that `func_1` might be executed in a thread in parallel with another thread that performs:

```
 g_2 = 42;
 printf ("%d",g_2)
```

The resulting system is data-race free and the only value that should be printed is 42. However gcc -O2 generates the following x86-64 assembler for `func_1`:

```
func_1:
      movl    g_1(%rip), %edx
      movl    g_2(%rip), %eax
      testl   %edx, %edx
      jne     .L2
      movl    $0, g_2(%rip)
      ret
.L2:
      movl    %eax, g_2(%rip)
      xorl    %eax, %eax
      ret
```

and this code always performs a write to `g_2`. If this asm code runs in parallel with `g_2 = 42; print g_2`, then the system might also print 0: this behaviour is introduced by the compiler and should not have happened.

It might be the case that in the C++11 memory model it is safe for the compiler to introduce a write provided that there is an earlier write to the same location, but this testcase shows that introducing a write is unsafe whenever there are no previous writes.

We filed a bug report (http://gcc.gnu.org/bugzilla/show_bug.cgi?id=52558), and GCC developers are working to fix it (see the discussion associated to bug report).

4.1.1 Current False Positives

Some limitations of the current trace-matching algorithm, make it fail to match equivalent traces. We discuss here the outstanding issues.

Size Issues Related memory accesses can differ in the size of the data being written or read between the reference and optimized trace. We have already discussed some access size discrepancies in Section 3.8.1: those are taken into account by the trace matching algorithm. Some others are not. For instance we frequently observe a store to a `long long` (8 bytes) variable being spilt up into two long accesses of 4 bytes each in the unoptimized trace. As an example, consider:

```
                    g_1  0 8 Init


        ***unopt***                         ***opt***
      ...
      g_1   AAAAAAAA 4 Store        g_1 BBBBBBBBAAAAAAAA 8 Store
      g_1+4 BBBBBBBB 4 Store
      ...
```

Recall that `g_1+4` represents the word at the address `g_1 + 4`. Here the optimizer has coalesced the two 4 byte accesses into a single 8 byte access. This optimization is correct (one could argue here that the unoptimized trace is not a real reference trace), but we did not implement this matching in the algorithm as it would drastically slow down the matching.

The actual size of the memory accesses can vary with the optimization level. With GCC, in a few programs we could observe a reduction in the size of load accesses between the unoptimized trace and the -O2 optimized trace, but the -O3 optimized trace uses the same size as the unoptimized one (and then perform size conversion in registers).

With Clang we also observe coalesced write accesses for contiguous array locations.

```
                    char g_14[8] = 0;
  ***unopt***                         ***opt***


 g_14    B1 1 Store
 g_14+1  B1 1 Store
 g_14+2  B1 1 Store
 g_14+3  B1 1 Store
 g_14+4  B1 1 Store         g_14 B1B1B1B1B1B1B1B1 8 Store
 g_14+5  B1 1 Store
 g_14+6  B1 1 Store
 g_14+7  B1 1 Store
```

Notice that all the 1 byte array accesses have been merged into a single 8 byte store. Similar behaviours can be observed with load accesses with Clang.

Irrelevant Load and Store Introductions Quite frequently we have observed irrelevant load and store introductions by GCC in optimized traces. We believe it is correct to introduce a store in a path where there is already a store present, because any context that attempts to exploit the newly introduced store to generate an unexpected behaviour must exhibit a data-race with the unoptimized program. Here is a snippet of a trace where a second store to g_60 is introduced by the optimizer.

```
      ***unopt***                         ***opt***
     g_60  0   1 Store                g_60  0   1 Store
RaW* g_60  0   1 Load                 g_37  FFF 4 Load
     g_37  FFF 4 Load                 g_172 FFF 4 Store
     g_172 FFF 4 Store                g_60  0   1 Store
```

Note that this is different from the bug that was described, where a store was introduced in a path where there was none. The tool is currently unaware of store introductions and reports them as errors. We have had some success in dealing with irrelevant read introductions by running the IR analysis on the optimized trace. However, as for irrelevant read eliminations, this is subject to the limitations of IR analysis described below.

Irrelevant Read Analysis We perform the analysis of irrelevant reads as part of the binary instrumentation. This means that the analysis can observe only one execution path. As a consequence, we cannot implement a correct and complete analysis for memory loads whose value is used to decide the control flow. For instance, consider:

```
...
if (g_1){
  ...       //no global store.
}else{
  g_2 = 5;
}
```

Note that in the above example if the value of `g_1` is not zero then the *if* block executes, and it has no global store. So the analysis might deduce that the load of `g_1` is irrelevant. However this would be incorrect in a concurrent context where some other thread modifies `g_1` to 0, and the *else* block that is executed; in which case the load of `g_1` would be relevant. If we consider loops, deciding which loads used in the control flow are relevant becomes even more complicated. We experimented both a sound analysis (which in the example above consider `g_1` as relevant because it cannot test the *else* branch), which ends up missing several irrelevant reads, and an unsound analysis (which would deduce that the `g_1` read above is irrelevant), but both end up in false positives.

Test Case Reduction Whenever the trace matching algorithm returns false, we obtain a fairly complicated program and big traces which are unsuitable not only for filing a bug report but also to understand why the trace matching failed. We rely on test-case reduction as performed by delta: remember that delta keeps on removing lines from the source file until when the trace matching succeeds, and returns the smallest example for which the trace matching fails. The problem of this approach is that it does not guarantee that the source and the reduced case fail for the same reason. It might well be that programs exhibiting interesting bugs are reduced into programs that exhibit known bugs, or even false positives. Currently test case reduction is likely to be the bottleneck of our testing infrastructure.

4.2 Perspectives

The subtle bug we found in GCC validates our approach based on random sequential program generation and trace matching. However, at the time of writing, this research project is not yet complete.

We have built most of the infrastructure but our tool should be extended to support some missing C features, most notably `structs`, unions and bitfields, where most of the concurrency bugs are likely to lurk. We recently added support for arrays but we must yet perform serious testing and analyse the results. Preliminary investigations show lots of write introduced and/or merged in the optimized trace. Clang does not yet support the source compilation of C++11 low level atomics so we could not test their compilation (but the LLVM bytecode already includes special annotations for the atomic attributes, so it should not take long before low-level atomics are supported by Clang). We do not support `read_modify_write` C/C++11 instructions: these are usually compiled with locked instructions as `XCHG`, and our tracing and matching infrastructures ignore that the read and write performed by locked instructions are guaranteed to execute atomically.

The trace matching algorithm should be extended to support merging of writes, and clever heuristics must be baked into the algorithm to tackle longer and more complex traces. We also need a finer analysis of irrelevant reads, and improving the testcase reduction strategy, so that we minimise the number of false positives, as described in the previous section. We also believe that CSmith might be tailored to generate more interesting programs, by sprinkling release/acquire pairs in complex expressions (so that it is more likely that the optimizer gets confused).

Once the tool will have been extended and polished, the GCC developers have expressed interest in integrating it in their standard test suite.

Our work could also be used to understand what kind of optimizations a compiler can perform and can also be used as a guide to the compiler developers. For example we found an example were a variable was never read in the source program but we observed a load of the variable in the optimized trace.

```
int g_2 = 5L;
int g_13 = 0xFAL;
int g_18 = 0xCDEDL;
int  func_1(void) {
  for (g_2 = 0; (g_2 >= (-17)); g_2 -=  1)  {
    if (g_2)
      g_13 = g_2;
    else
      g_18 = 16;
  }
  return 1;
}
int main (int argc, char* argv[]) {
    func_1();
    return 0;
}
```

If we compile the above program, with -O2 using GCC a load of `g_13` is (surprisingly) introduced in the optimized trace. There are several more cases in the compiler introduces many more unnecessary loads. Our setup can thus be used to provide feedback to the compiler developers providing insight into what we actually get in the traces and find better ways to optimize concurrent code.

Bibliography

[1] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. Mathematizing c++ concurrency. In *POPL*, pages 55–66, 2011.

[2] P. Becker, editor. *Programming Languages — C++.* 2011. ISO/IEC 14882:2011. A non-final but recent version is available at `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3242.pdf`.

[3] H.-J. Boehm and S.V. Adve. Foundations of the C++ concurrency memory model. In *Proc. PLDI*, 2008. doi: http://doi.acm.org/10.1145/1379022.1375591.

[4] Hans-J. Boehm. Threads cannot be implemented as a library. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 261–268. ACM, 2005.

[5] Laura Effinger-Dean, Hans-J. Boehm, Dhruva Chakrabarti, and Pramod Joisha. Extended sequential reasoning for data-race-free programs. In *Proceedings of the 2011 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*, pages 22–29. ACM, 2011.

[6] Eric Eide and John Regehr. Volatiles are miscompiled, and what to do about it. In *EMSOFT*, pages 255–264, 2008.

[7] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 190–200, 2005.

[8] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, pages 89–100, 2007.

[9] Jaroslav Sevcík. Safe optimisations for shared-memory concurrent programs. In *PLDI*, pages 306–316, 2011.

[10] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. In *PLDI*, pages 283–294, 2011.