Project no. 644235

REPHRASE

Research & Innovation Action (RIA)
REFACTORING PARALLEL HETEROGENEOUS RESOURCE-AWARE APPLICATIONS – A SOFTWARE ENGINEERING APPROACH

Software for testing, initial quality assurance, verification, detection of catastrophic failures and detection of property violations D3.1

Due date of deliverable: 31.03.2016

Start date of project: April 1st, 2015

Type: Deliverable
WP number: WP3

Responsible institution: IBM
Editor and editor's address: Sharon Keidar-Barner, IBM

Version 0.1

Project co-funded by the European Commission within the Horizon 2020 Programme		
Dissemination Level		
PU	Public	√
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

Executive Summary

This document is the initial deliverable from WP3 "Reliability, Robustness and Software Integrity of Parallel Software". It hosts the development of tools to *i)* test functional and extra-functional properties of parallel data-intensive software of the **RePhrase** against their requirements; *ii)* detect catastrophic failures in parallel data-intensive applications, such as deadlocks and race conditions; *iii)* provide validation and verification mechanisms for parallel implementations of patterns; *iv)* support quality assurance; and *v)* detect extra-functional property violations, such as performance and energy consumption problems.

The deliverable is the result of the first phases of WP3 (T3.1: "Testing Parallel Software", T3.2: "Detection of Catastrophic Failures including Race Conditions and Deadlocks", T3.3: "Verification of Patterned Code", T3.4: "Quality Assurance Analysis" and T3.5: "Detection of Extra Functional Property Violations") where, according to the DoW, we will

- *extend the existing technologies for testing, verification and debugging parallel applications and integrate them into the implementation and testing/verification phases of the* ***RePhrase*** *methodology*
- *develop tools for testing parallel data-intensive applications, detecting possible failures and violations of functional and extra-functional requirements and verifying that parallel versions of the applications have the same functionality as their sequential versions, supporting the implementation and testing/verification phases*
- *develop a set of tools for testing parallel applications and discovering violations of functional and extra-functional requirements, ensuring that the software produced by RePhrase is reliable, robust, resilient, and adaptive*

This deliverable consist of a review of existing approaches for failure detection of parallel applications (Helgrind, DRD, ThreadSanitizer), and a set of software tools developed by the consortium which were enhanced with initial capabilities to address the objectives of WP3.

Contents

1. Introduction

As we pave the way towards Exascale computing, the use of multi- and many-core architectures, with possibly one or more co-processors/accelerators, working together to efficiently solve scientific problems becomes a complex challenge that the HPC community needs to face [2, 3]. The adoption of high-level parallel programming models relieves the developers from some of the burden typically involved in designing and implementing parallel applications from scratch. Despite this, much of the current software is not yet fully accommodated to run on recent parallel platforms. In most cases, hardware design progresses faster than the parallelization and optimization processes of existing software. To deal with this issue, the use of building blocks implementing core functionalities has been a well accepted approach in the HPC area [8]. Indeed, many of scientific parallel applications leverage efficient parallel kernels from highly-tuned libraries. However, these kernels must guarantee correctness and thread-safety in order to generate correct global results.

While parallel programming techniques have been broadly adopted in implementing large scientific applications, concurrency bugs, especially data races and deadlocks, have become more frequent. The adversity in finding data races and deadlocks is a well known problem [4]. Detecting catastrophic errors has been recognized as an arduous task, given that errors may occur only during low-probability sequences of events and may also depend on the external factors such as the current machine load. This makes their detection extremely sensitive to timing, I/O operations, compiler options and differences in memory models. Data races are especially difficult to observe, since often they quietly violate data structure invariants rather than cause immediate crashes. Although data race detectors alleviate debugger's task in finding these issues, they are still not perfect [1, 4]. Furthermore, most of the testing and verification mechanisms are still aimed at sequential applications, and there is not much work in extending these to address the above mentioned problems that arise in parallel settings. Finally, there is also very little work in mechanisms for detecting violation of extra-functional requirements of applications, such as finishing execution in a certain time period or staying within certain energy budget. Violating these requirements may render the parallel code as useless as if it had actual errors in it.

This documents makes several contributions to the state-of-the-art in detecting failures and testing/verification of parallel applications:

- we provide a methodology and automated prototype tool support for testing both different parts of the overall **RePhrase** tool-chain, and the patterned parallel data-intensive applications produced by the **RePhrase** technology, aiming to improve the integrity of the parallel software;
- we extend the IBM FOCUS test planning tool, the Static Analysis tool PRL QA-Verify, and the Formal verification ExpliSAT tool to provide test planning and verification for parallel, and specifically patterned, applications;
- we evaluate three well-known tools for failure detection of parallel applications: the Helgrind and DRD tools, from the Valgrind environment, and the ThreadSanitizer detector, a tool compliant with the Clang C/C++ and GCC-GNU compilers, demonstrating some of their shortcomings and pointing to their proper use for detecting errors and catastrophic failures;
- we describe extensions to the ParaFormance refactoring tool described in D2.2 that i) detect the race conditions in parallel applications; ii) calculate cost of the execution of loops in application, indicating hotspots for parallelisation and potential granularity issues; and, iii) detect communication hotspots in threaded code

This document is organized as follows: Chapter 2 describes the extension to the IBM FOCUS planning and reduction tool to provide test planning for parallel, and specifically patterned, applications. Chapter 3 describes ExpliSAT, formal verification tool enhanced to support concurrent applications. Chapter 4 describes the extension of the Static Ananlysis tool, PRL QA-Verify to find catastrophic failures in parallel programs and points to experimental results. Chapter 7 describes the Helgrind, DRD and ThreadSanitizer tools, enumerate their advantages and drawbacks. Chapters 5 and 6 describe the prototype extensions to the ParaFormance refactoring tool for detection of race conditions and extra-functional property violations.

2. The IBM Functional Coverage Unified Solution (IBM FOCUS) Test Planning Tool

IBM FOCUS is an advanced test planning tool for improving the testing of an application. FOCUS uses Combinatorial Test Design (CTD) to generate an efficient test plan that provides consistent coverage across the test space at a known depth, while significantly reducing the required resources. FOCUS is independent of the application's domain, and can be applied at different levels of testing. FOCUS can also read existing tests, analyze their functional coverage, select a subset of the tests that maintains the same coverage, and generate new tests to close the coverage gaps. FOCUS requires a user definition of the test space, and provides advanced review and debugging capabilities to verify that the test space was defined correctly. IBM FOCUS has been extended and can handle concurrent software.

IBM FOCUS tool can be downloaded from
`https://ctd.haifa.il.ibm.com/downloads/download.html`

For access information please contact Yonit Magid at "yonit@il.ibm.com". To install IBM FOCUS, unpack the downloaded zip file into any folder. To run FOCUS follow the instructions in "FOCUS/README.txt". To learn more about FOCUS follow the FOCUS tutorial in "FoCuS/focusTutorial/tutorial.html".

IBM FOCUS tool contains several sample models. We have also included a sample model for a concurrent application. To view in the tool open the file "samples/Concurrent/Concurrent.model" This model describes the test space for a concurrent system containing four threads, at least one of which has failed in some way. System recovery is attempted by restarting the failing threads in a certain order. The model has attributes representing the threads states and the system state at the time of the failure, and attributes representing the order in which the threads are restarted.

3. Verification of patterned code - IBM ExpliSAT Tool

3.1 Symbolic Interpretation

Symbolic analysis methods examine program computations while representing data symbolically. This is a broad category of formal technologies with several different flavors, including concolic testing [11, 18], bounded model checking [9], symbolic execution [6, 16], and symbolic interpretation [5][1]. The tool we use, ExpliSAT [7], is a symbolic interpreter. While the results we show can theoretically be demonstrated by any symbolic analysis method, we find that the symbolic interpretation approach was instrumental in achieving the high levels of automation and scalability required to attack real-life applications such as OpenSSL. We present here only an informal overview of symbolic interpretation. A comprehensive description of the technology is beyond the scope of this deliverable.

The idea behind symbolic analysis is to inject *symbols* in place of program inputs. Symbols are special placeholders that represent all possible values a variable can get (rather than a concrete value). During the traversal of a program execution path, computations over symbols result in program variables containing symbolic expressions instead of specific values. In addition, each control-flow path in the program is associated with a symbolic expression called the *path condition*, which defines the space of input values that would result in this particular path being executed. There are two major ways to implement this – by instrumenting the program or by building a symbolic interpreter for the language being analyzed. Symbolic execution techniques (as well as most concolic testing techniques) work by instrumenting the program code with the necessary modifications and then executing it. In symbolic interpretation, however, we do not change the program code but rather build an interpreter that mimics the execution of the program on a real machine, only using expressions. In what follows we focus on symbolic interpretation.

Figure 3.1 shows an example C function next to the results of a symbolic interpretation run along the program path 1-2-3-4-5-8. At every point in time the

[1]The precise classification into these different flavors of symbolic program analysis may be arguable, and in particular the distinction between "concolic testing", "symbolic execution" and "symbolic interpretation", that are sometimes used interchangeably.

interpreter keeps track of the current value of each variable as an expression over input symbols. We start by assigning a fresh symbol for each of the inputs (with an appropriate type). Program statements are interpreted one at a time along a specific control-flow path. The assignment on line 2 results in the symbolic expression $S_1 - S_2$ being assigned to `tmp`. When line 3 is reached the interpreter uses the current expression in `tmp` to compute the expression for `tmp` after incrementing it. When a branch is reached in line 4, the interpreter creates the symbolic expression for the branch condition (in this case $S_1 - S_2 + 1 > 0$) and uses a decision procedure to determine whether it is possible for the computation to continue along one or both sides of the branch. Notice that the symbolic expression for the branch precisely defines all input data values that would result in the "then" side being taken, and its negation defines all input values that would continue along the "else" side. If both sides are feasible the interpreter will heuristically choose one side to continue along first. Lets say we continue to line 5. The branch expression that lead us to this line is conjuncted to the path condition, to maintain the invariant that at any point in time the path condition precisely defines the set of inputs that would lead the computation to follow the path that was executed so far. On line 5, we encounter an assertion. The interpreter builds an expression that conjuncts the path condition with the negation of the assertion and checks its satisfiability using an appropriate decision procedure. If this expression is satisfiable it means that there exists a valuation of the input symbols that results in this assertion evaluating to $false$. This result is then reported to the user as an assertion violation. If not, the computation continues, in this case to line 8 where the current run terminates.

In this way the symbolic interpreter will traverse only feasible paths. In order to achieve completeness the symbolic interpreter will attempt to traverse all feasible control-flow paths. Whenever both sides of a branch are feasible the side that is not traversed is saved in a queue. The saved state includes all the information needed to continue the traversal, including the current symbolic expressions for all variables, the path condition, and the program counter location. Thus, when the computation terminates on line 8 the interpreter will pop from the queue the state that was saved when reaching the branch on line 4, and will continue the symbolic run through line 7.

The above is a very simplistic explanation of the core technology. It shows how a single control-flow path can be symbolically executed very efficiently. However, since the number of different control-flow paths is exponential it is difficult to achieve complete coverage with this naive approach. There are various ways to achieve scalability – by controlling the search, learning between runs, summarizing loops and functions, and so on. With all these improvements that have been suggested in recent years symbolic analysis tools are now able to offer considerable coverage of real-life programs, and often even prove correctness.

```
foo(int x, int y) {                 x ← $S_1$, y ← $S_2$
    int tmp = x-y;                  tmp ← $S_1 - S_2$
        tmp++;                          tmp ← $S_1 - S_2 + 1$
        if (tmp > 0)
           assert(tmp > x);             $\mathcal{P} \leftarrow S_1 - S_2 + 1 > 0$
        else
             return 0;
        return tmp;                     return $S_1 - S_2 + 1$
}
```

Figure 3.1: Example of symbolic run

3.2 ExpliSAT

ExpliSAT is a C/C++ software model checking tool that uses symbolic interpretation technology described in Section 3.1 for the verification of the code correctness. ExpliSAT is capable of detecting violations of user-defined assertions or existence of some generic programming errors (a.k.a built-in checks), or proving the absence of such violations or errors. Given a program (or function) under test, ExpliSAT explores each feasible execution path of a program, and if no path violating the error condition found, proven correctness result is generated.

3.2.1 Installation

To download the tool, please visit the following page:

https://ctd.haifa.il.ibm.com/downloads/download.html

The access is secured with user name and password. Please contact Dmitry Pidan (pidan1@il.ibm.com) or Sharon Keidar-Barner (sharon@il.ibm.com).

ExpliSAT is built for Linux OS, kernel 2.6.32 and newer. Below are step-by-step installation instructions:

1. Go to the downloading page and follow the link “ExpliSAT tool zip file” to save “explisat.zip” to your computer.

2. Create a folder where to put a tool (e.g. /home/ExpliSAT) and copy the downloaded archive there.

3. Being in the folder, open the archive, e.g. “tar zxvf explisat.zip”.

4. Set environment variable SMC_BASE in your shell to point to ExpliSAT folder, e.g. in K-Shell, do “export SMC_BASE=/home/ExpliSAT”

ExpliSAT uses Flexlm [2] licensing mechanism to ensure authorized usage. Please contact Dmitry Pidan (pidan1@il.ibm.com) for obtaining the license and instructions how to install it.

[2]https://en.wikipedia.org/wiki/FlexNet_Publisher

3.2.2 Working flow

Assume the goal is to verify the correctness of the function $hex2dec$ having the following signature:

```
int hex2dec(unsigned char* hex)
```

To accomplish the task of preparing a complete verification problem, a unit test that defines function inputs and provides checkers for expected results should be written:

```
#include <formal.h>
extern int hex2dec(unsigned char*);
int main()
{
   int i;
   unsigned char buf[8];
   for (i=0; i<8; ++i) {
      buf[i] = nondet_uchar();
   }
   int value = hex2dec(buf);
   if (buf[0]>='0' && buf[0]<'8')
      fv_assert(value>=0);
   else
      fv_assert(value<0);
   return 0;
}
```

The bold keywords $nondet\_uchar$ and fv_assert in the unit test code above are special keywords recognized by ExpliSAT. The first one is a placeholder for *symbolic* value of type unsigned char, where symbolic means "any possible value in the domain" (0 to 255 in this case) [3]. The second keyword (fv_assert) defines the condition that must be satisfied while the execution of the program reaches the location where this keyword appears.

Now we have a complete code consisting of the function under test and the wrapping unit test in "main" function that drives the function inputs with symbolic values, and checks the function output. The next step is to build the code with the ExpliSAT-provided compiler and then verify it using the tool:

```
1 >> fv-compile hex2dec.c main.c -o hex2dec
2 >> smc --halt-on-error hex2dec
3 Starting ExpliSAT
4 Writing testcase to file __expliSAT_ce_main_17.c
5 Assertion failure found in file main.c, line 17
6 VERIFICATION FAILED
```

The result of the run shown in lines 4-6 indicates that a concrete valuation of the function inputs was found such that the assertion condition was not satisfied. Its now up to the user to take this information (the concrete valuation found by the tool a.k.a "counter example" can be extracted from the concrete test produced by the tool as indicated in line 4), debug and fix the code under test. If no violation of either assertions or built-in checks is found, "VERIFICATION PASSED" result is issues, which means that ***the code is correct with respect to provided assertions and built-in checks for every possible value of the input***.

[3]Symbolic values of any primitive type can be defined using the same pattern $nondet\_ < type >$

3.2.3 Built-in checks

In addition to user-defined assertions, ExpliSAT is capable of verifying the absence of so-called generic programming errors, using a rich set of built-in checks that are able to verify the absence of those errors without a need for any additional action to be taken by the user. Below is a list of programming errors ExpliSAT is capable to verify autonomously:

- ***Use of uninitialized variable*** - the variable is used before it is assigned any value
- ***Non-null terminated string*** - in libc string function (e.g. strcpy, strlen, strcmp), a non-null terminated string is given as an input
- ***Exception is not handled*** - exception was thrown, but no catch block for it for found
- ***Wrong pointer dereference*** - on dereferencing, an invalid pointer value was detected (e.g. null value), or pointer does not point to any valid memory block (e.g. released memory)
- ***Wrong release of memory*** - releasing of pointer that does not point to any previously allocated block (e.g. double-free)
- ***Memory leak*** - on the end of particular execution path, a non-released memory was detected
- ***Buffer overflow / underflow*** - access to a memory block out of its previously allocated bounds
- ***Wrong function pointer dispatch*** - on an access to a function via a pointer, function was not found (e.g. can happen when input value propagates to a function pointer)
- ***Division / Modulo by zero*** - zero value was detected in denominator of division or modulo operation

4. Static Analysis - PRL QA-Verify

During the first phase on this task we have integrated existing tools for multi-threading static analysis (Helgrind, DRD, ThreadSanitizer) into PRL QA-Verify software. We have created instance of QA-Verify on the server with full access for the consortium members (http://172.16.100.171:9080). For provided samples of the use cases we have set up automatic nightly analysis using these tools and PRL QA-C and QAC++ analysers. All the results automatically uploaded into QA-Verify server.

Additionally we have developed the first version of PRL Multi-Threading and Resource analyser (MTR) and currently added this analysis to the automatic nightly analysis with results uploaded to QA-verify server. MTR module is available for evaluation together with the PRL main software at the PRL web site (http://www.programmingresearch.com/resources/request-evaluation/). (For delivery review purpose please complete the evaluation form and put the comment that this is for review purpose for EU project. After that you will receive a direct link to the download).

Results of analysis have shown a serious number of the catastrophic failures including race conditions and deadlocks - reports available at: https://rephrase.programmingresearch.com/rephrase/index.html. Currently we have started the work for expansion of MTR to implement analysis of initial patterns developed in T2.2 to detect specific patterns related issues.

5. Detection of Catastrophic Failures including Race Conditions and Deadlocks

5.1 Safety

The refactorings for WP2 (and outlined in D2.2) should preserve the correctness of the functional semantics of the C++ program under refactoring. The meaning of preserving correctness is: given the same input value(s), the program should produce the same output value(s) before and after a refactoring, *up to a given ordering*. These functional semantics are separated from extra-functional ones, such as performance and memory usage. When we talk about *safety* we are particularly interested in the preservation of the program's functional semantics. For this deliverable, there are two approaches we can take for safety checking: *static* and *dynamic*.

5.1.1 Static Checking

Static safety checking has a number of advantages and limitations over dynamic checking.

- Static checking does not require the application to be parallelised before proceeding and can run on sequential code.
- Checks code before and after a refactoring is applied at the source level.
- Ensures a lack of race conditions, deadlocks and other problems that may affect safety.
- We can introduce additional refactorings to remove these problems.
- Static analysis is generally faster than dynamic analysis, as the application does not need to be executed in order to analyse for safety problems. However, for some analyses, problems of scalability arise, where the time and/or space required by the analysis grows unacceptably for large input programs.

- Generally, static analysis involves much more sophisticated and complicated procedures to check for safety of code.
- A carefully designed static analysis can *prove* that a program is safe, in that no possible input can cause undesirable behaviour. This is a delicate matter though, as it is well-known that interesting program properties are typically *undecidable*, meaning that it is impossible to produce an analysis that can decide whether or not the property is satisfied. The practical import of this is that if an analysis can successfully detect all of the actual problems in a given program then it must also sometimes report *false positives*: suspected errors that are in fact unproblematic. Alternatively, the tool could occasionally report that it can't decide whether (part of) a program is safe or not. Careful design and tuning may be required to ensure that good results are achieved for a majority of programs.

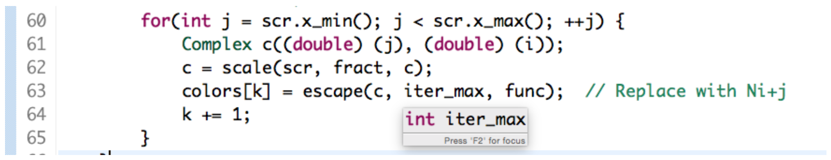
There are a number of different properties that a static analysis system can check for to determine potential safety violations. For this deliverable, we only consider *updated variables*. We leave the other properties for future work.

Side Effects. We must be able to capture and control any side-effects. These can include (but are not limited to) global destructive updates of variables, IO reading or writing, etc. It is permissible for the refactoring tool to error on detection of a side-effect in the user-highlighted code.

IO Files and Streams. Updating and writing to files, printing to the screen and reading in data are generally considered to be unsafe. It is possible in some cases for the user to flag some of these calls as permissible but in general it is a good idea for the tool to highlight any call to the `iostream` as being unsafe.

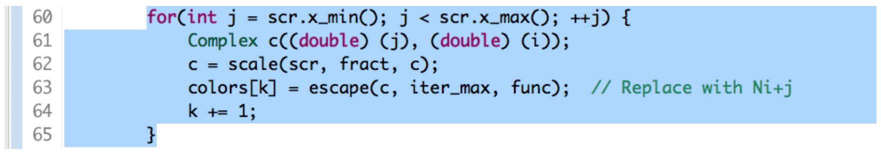
Updated Variables. Updating global or free variables in threaded code causes race conditions. These can occur in the following situations:

- **Function calls.** A function or method is called which updates a global/free variable.
- **Objects/Arrays.** In C++ it is possible to call an object directly (see function calls); alternatively, array elements could be updated. This is problematic if the same element is being updated in different iterations of a loop, for example.
- **No Pointers.** Pointers cause lots of issues. Particularly the use of pointer arithmetic.
- **Global and Free Variables.** Updating variables can either arise from a thread updating a global variable or one that is *free* (i.e., bound outside of



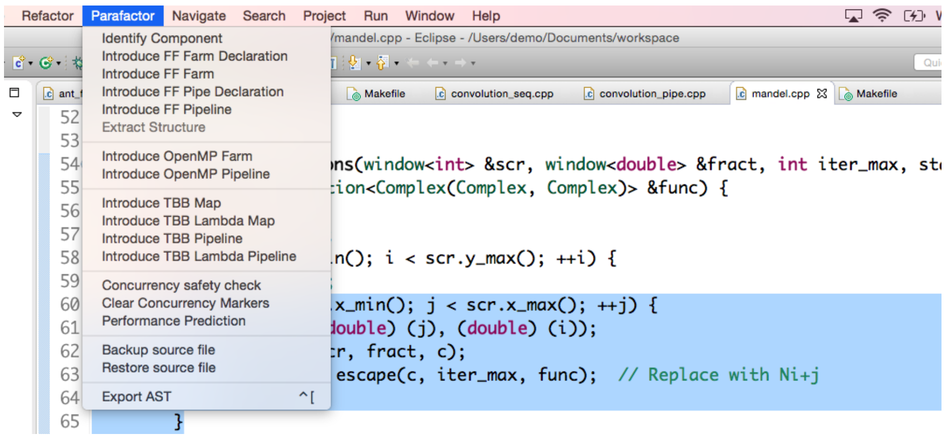
```
60        for(int j = scr.x_min(); j < scr.x_max(); ++j) {
61            Complex c((double) (j), (double) (i));
62            c = scale(scr, fract, c);
63            colors[k] = escape(c, iter_max, func);  // Replace with Ni+j
64            k += 1;
65        }
```

Figure 5.1: Sequential code that shows a potential data race on Line 64



```
60        for(int j = scr.x_min(); j < scr.x_max(); ++j) {
61            Complex c((double) (j), (double) (i));
62            c = scale(scr, fract, c);
63            colors[k] = escape(c, iter_max, func);  // Replace with Ni+j
64            k += 1;
65        }
```

Figure 5.2: Sequential code is highlighted for refactoring within Eclipse



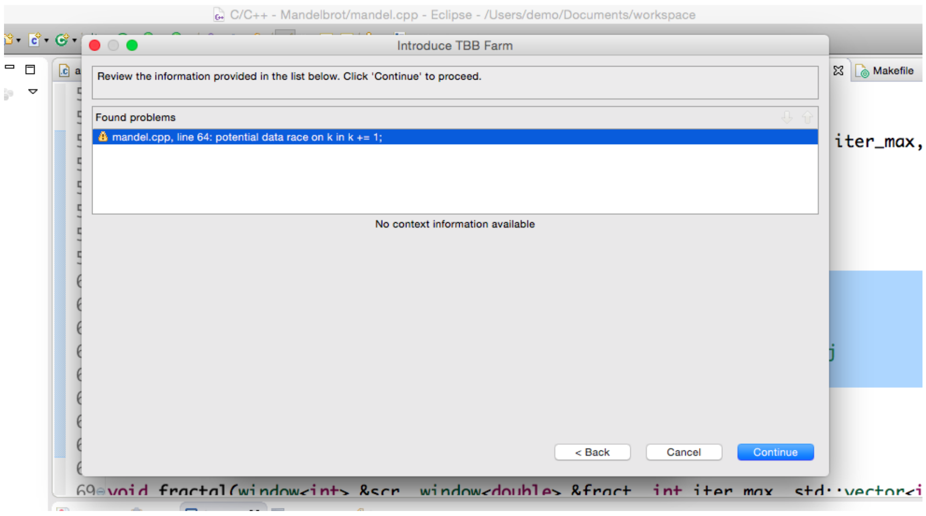
```
54 ns(window<int> &scr, window<double> &fract, int iter_max, st
55 tion<Complex(Complex, Complex)> &func) {
58 n(); i < scr.y_max(); ++i) {
60 x_min(); j < scr.x_max(); ++j) {
61 ouble) (j), (double) (i));
62 r, fract, c);
63 escape(c, iter_max, func);  // Replace with Ni+j
65        }
```

Figure 5.3: A menu of refactoring choices is presented to the programmer

the scope of the thread). Both of these instances causes issues that the variable may be accessed by other parts of the program or by other threads at the same time.

To give an example of the static safety checking in the refactoring tool, consider Figure 5.1. In the figure, we show a block of sequential code that is targetting for parallelisation by the programmer. The sequential code as stated runs correctly, but when executed in parallel produces an incorrect result. This is due to the fact that at Line 64, there is an assignment to a global variable, `k`.

The programmer highlights this code without realising that there is a potential data race (as shown in Figure 5.2 and chooses to introduce a TBB map pattern from the refactoring drop down menu (show in Figure 5.3; this work is also described in Deliverable D2.2). The refactoring tool then displays a warning message to the programmer, stating *madel.cpp, line 64: potential data race on k in k += 1*. This warning message is illustrated in Figure 5.4.


C/C++ - Mandelbrot/mandel.cpp - Eclipse - /Users/demo/Documents/workspace
Introduce TBB Farm
Makefile
< Back
Cancel
Continue

Review the information provided in the list below. Click 'Continue' to proceed.

Found problems

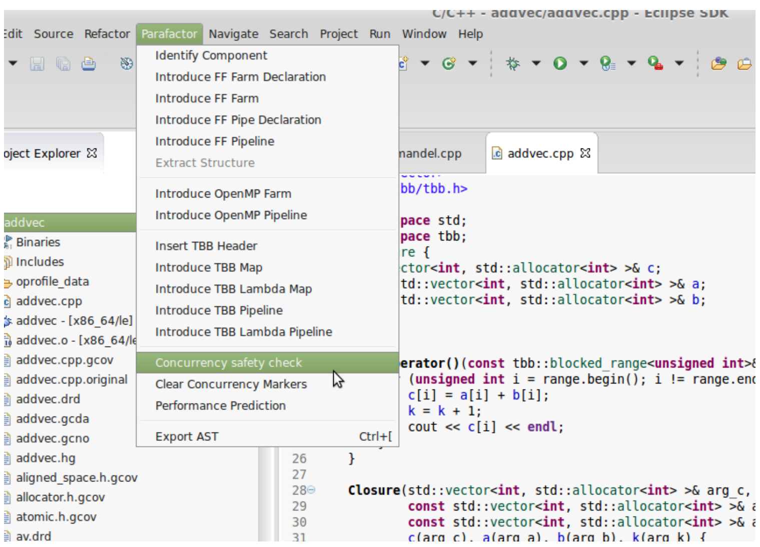
mandel.cpp, line 64: potential data race on k in k += 1;

No context information available

Figure 5.4: A safety violation warning from the refactoring tool stating that Line 64 has a potential data race on `k += k + 1`

5.1.2 Dynamic Checking

- Requires application to be parallelised before proceeding with analysis.
- Dynamic checking uses profiling information obtained from running an execution instance of the parallelised application.
- Most dynamic systems are limited to checking only for race conditions and deadlocks in parallelised code.
- Generally, dynamic checking is easier to implement than static checking, as an execution profile can contain more information about how the program has executed, memory accesses, etc.
- There are a number of existing tools available that perform dynamic checking to some degree.
- Dynamic checking is generally slower than static checking as it requires the programmer to execute an entire instance of their parallelised application.
- Dynamic techniques are less general that static techniques. They are usually dependent on a particular threading model, working at a very low level, making it hard to relate the problems back to the source code in a high-level way.
- For most properties of interest, dynamic techniques are incapable of definitively proving that bad behaviour cannot occur. It may be the case that a particular error only arises for some very rare combination of program inputs, or



```
bb/tbb.h>

pace std;
pace tbb;
re {
ctor<int, std::allocator<int> >& c;
td::vector<int, std::allocator<int> >& a;
td::vector<int, std::allocator<int> >& b;

erator()(const tbb::blocked_range<unsigned int>&
 (unsigned int i = range.begin(); i != range.enc
  c[i] = a[i] + b[i];
  k = k + 1;
  cout << c[i] << endl;
}

Closure(std::vector<int, std::allocator<int> >& arg_c,
        const std::vector<int, std::allocator<int> >& a
        const std::vector<int, std::allocator<int> >& a
        c(arg_c), a(arg_a), b(arg_b), k(arg_k) {
```

Figure 5.5: Our refactoring tool built into Eclipse, with a menu showing the safety (concurrency checking) feature

perhaps when some random hardware-dependent factors are satisfied, such as two threads finishing their execution in a particular order. In cases such as this, it could be very unlikely that dynamic analysis would reveal the problem, as we can't examine all possible executions of the program.

Helgrind. Helgrind is a Valgrind tool for detecting synchronisation errors in C, C++ and Fortran programs that use POSIX pthreads threading primitives. The main abstractions of POSIX pthreads are: a set of threads sharing a common address space, thread creation, thread joining, thread exit, mutexes (locks), condition variables (inter-thread event notifications), reader-writer locks, spinlocks, semaphores and barriers.

Helgrind can detect three kind of POSIX errors:

1. Misuses of POSIX pthread, such as invalid uses of locks that may introduce deadlocks/race conditions.
2. Potential deadlocks arising from lock ordering problems.
3. Data races, i.e., accessing memory without any adequate synchronisation or locking.

For this deliverable, we have integrated Helgrind into the refactoring tool to offer dynamic safety checking of the refactored code. This entailed adding a menu option to check the refactored code for safety (as illustrated in Figure 5.5). This then executes the code, runs helgrind, and then extracts from helgrind's report the line numbers and type of safety condition associated with each line number. These line numbers will then be presented to the user by highlighting each line red in the Eclipse IDE and displaying the helgrind safety conditions as warnings to the programmer.

As an example, consider the following program with a global variable, `x`.

```
...
int x=4;
...
/********* Intel TBB specific part **********/
class CPU_Solve_Farm_Component {
public:
  void operator()(const blocked_range<size_t>& r) const {
    for (size_t i=r.begin(); i!=r.end(); i++) {
      cost[i] = solve(i);
      x = x + 1;
    }
  }
  CPU_Solve_Farm_Component() {};
};
/*******************************************/
...
  for (j=0; j<num_iter; j++) {
    /******* Intel TBB farm ********/
    parallel_for(blocked_range<size_t>(0, num_ants, min_chunk_size),
       CPU_Solve_Farm_Component());
    /******************************/
    best_t = pick_best(&best_result);
    update(best_t, best_result);
  }
```

Executing this program with the following valgrind command:

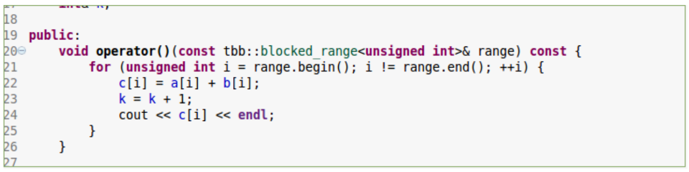
```
valgrind --tool=helgrind ./ant_farm 2 1 2 inputs/wt1000.txt 1
   &> helgrind.output.two
```

We are able to obtain a report, containing errors relating to the source code. This is an example of one such error:

```
==93581== Possible data race during write of size 4 at 0x1000096D4 by thread #2
==93581== Locks held: none
==93581==    at 0x100007CF1: CPU_Solve_Farm_Component::operator()
(tbb::blocked_range<unsigned long> const&) const (ant_farm.cpp:213)
==93581==    by 0x10000718E: tbb::interface7::internal::start_for
<tbb::blocked_range<unsigned long>, CPU_Solve_Farm_Component,
tbb::auto_partitioner const>::run_body
(tbb::blocked_range<unsigned long>&) (in ./ant_farm)
```

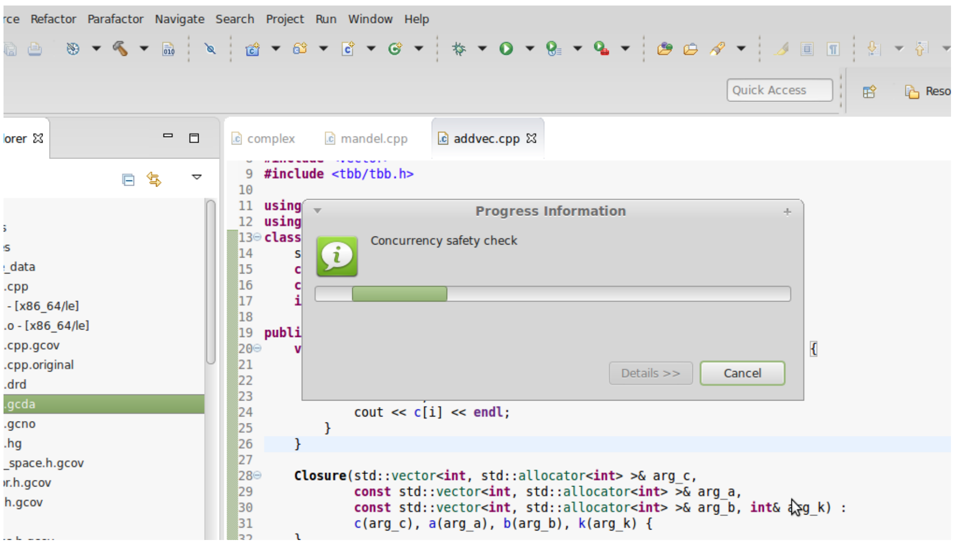
This error (pointing to Line 213) shows we have a data race in the original code:

```
x = x + 1;
```



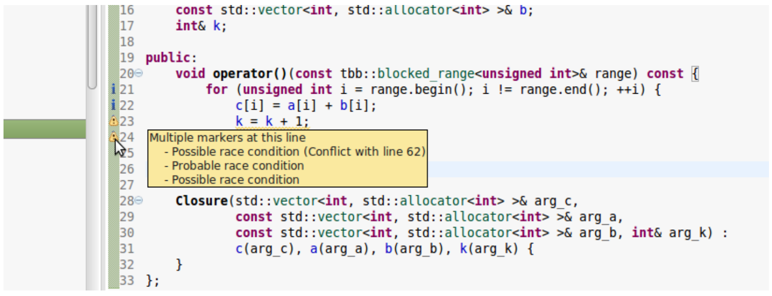
```
18
19 public:
20     void operator()(const tbb::blocked_range<unsigned int>& range) const {
21         for (unsigned int i = range.begin(); i != range.end(); ++i) {
22             c[i] = a[i] + b[i];
23             k = k + 1;
24             cout << c[i] << endl;
25         }
26     }
```

Figure 5.6: Our refactoring tool built into Eclipse, complete with a menu of refactorings for FastFlow, OpenMP and TBB



```
9  #include <tbb/tbb.h>
10
11 using
12 using
13 class
14     s
15     c
16     c
17     i
18
19 publi
20     v
21
22
23
24             cout << c[i] << endl;
25         }
26     }
27
28     Closure(std::vector<int, std::allocator<int> >& arg_c,
29             const std::vector<int, std::allocator<int> >& arg_a,
30             const std::vector<int, std::allocator<int> >& arg_b, int& arg_k) :
31             c(arg_c), a(arg_a), b(arg_b), k(arg_k) {
```

Figure 5.7: Safety violations are shown in amber at the left of each line



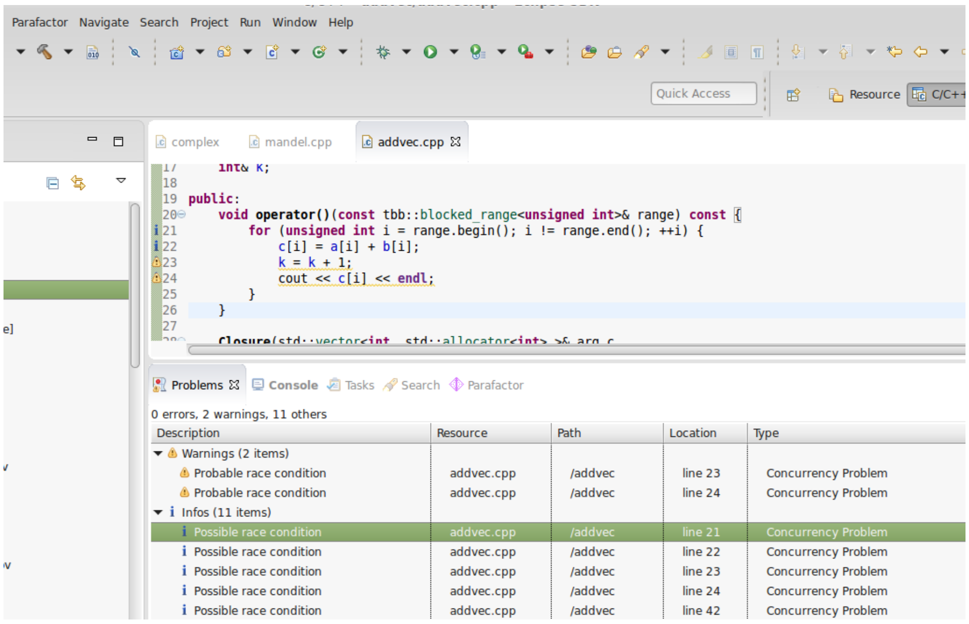
```
16     const std::vector<int, std::allocator<int> >& b;
17     int& k;
18
19 public:
20     void operator()(const tbb::blocked_range<unsigned int>& range) const {
21         for (unsigned int i = range.begin(); i != range.end(); ++i) {
22             c[i] = a[i] + b[i];
23             k = k + 1;
24
25
26
27
28     Closure(std::vector<int, std::allocator<int> >& arg_c,
29             const std::vector<int, std::allocator<int> >& arg_a,
30             const std::vector<int, std::allocator<int> >& arg_b, int& arg_k) :
31             c(arg_c), a(arg_a), b(arg_b), k(arg_k) {
32     }
33 };
```

```
Multiple markers at this line
  - Possible race condition (Conflict with line 62)
  - Probable race condition
  - Possible race condition
```

Figure 5.8: Mouse over showing a more detailed description of the safety violation at the current line number

This is due to each thread trying to update the variable, `x`.

To illustrate how this feature can be used to check the safety of parallelised code dynamically in the refactoring tool, consider the example code illustrated in Figure 5.6. Here we show an already refactored program, with a TBB parallel-for inserted into the sequential code. As it can be seen from the figure, Line 23 has a race condition, `k=k+1`. This is because a variable that is declared *free* to the block



```
17      int& k;
18
19  public:
20      void operator()(const tbb::blocked_range<unsigned int>& range) const {
21          for (unsigned int i = range.begin(); i != range.end(); ++i) {
22              c[i] = a[i] + b[i];
23              k = k + 1;
24              cout << c[i] << endl;
25          }
26      }
27
28      Closure(std::vector<int, std::allocator<int> >& arg_c
```

0 errors, 2 warnings, 11 others

Description	Resource	Path	Location	Type
Warnings (2 items)				
Probable race condition	addvec.cpp	/addvec	line 23	Concurrency Problem
Probable race condition	addvec.cpp	/addvec	line 24	Concurrency Problem
Infos (11 items)				
Possible race condition	addvec.cpp	/addvec	line 21	Concurrency Problem
Possible race condition	addvec.cpp	/addvec	line 22	Concurrency Problem
Possible race condition	addvec.cpp	/addvec	line 23	Concurrency Problem
Possible race condition	addvec.cpp	/addvec	line 24	Concurrency Problem
Possible race condition	addvec.cpp	/addvec	line 42	Concurrency Problem

Figure 5.9: A detailed analysis report of potential safety violations in the refactoring tool

of code that will be executed in parallel (Line 20) is being updated. This kind of error is often overlooked by novice and even expert programmers alike. We can then run the concurrency safety check from Figure 5.5. The result of this check is shown in Figure 5.7. In the figure, line numbers in the Eclipse IDE corresponding to potential safety violations are marked in amber on the left side. The programmer can then hover the mouse over each amber warning to display a detailed report. Figure 5.8 shows a mouse-over action. Here the tool explicitly warns that Line 23 has a *possible race condition*. A more detailed analysis of the whole application can be viewed as a report instead, as shown in Figure 5.9.

6. Detection of Extra-Functional Property Violations

In this chapter, we briefly describe the work done in T3.5 and we give overview of a preliminary version of the tool for detecting violations of extra-functional properties in parallel applications. As opposed to the catastrophic failures described in 5, here we deal with the issues in the application code that do not produce errors that prevent the application from finishing or returning the correct result, but rather that violate some of the extra-functional requirements for the application, such as performance or energy consumption. Amongst others, the issues of this type include i) too fine or too coarse granularity, which can result in too big overheads in thread creation or load imbalance; ii) communication hotspots, which can result in too big synchronisation overheads where, for example, threads are waiting for a long time to obtain locks; iii) bad data placement, which can result in too much time being spent in accessing remote data etc. In this deliverable, we focus on helping the detection of the first two issues, granularity and communication hotspot problems.

We have extended the ParaFormance refactoring tool with a mechanism to detect locations in the application source code which are the potential sources of granularity and communication problems. It is built on top of the Callgrind profiling tool, which itself is a part of the Valgrind suite. Callgrind can log low-level events, such as the number of instructions executed and bus locking, and can link these events to the lines in the application source code where they occur. Our tool processes the Callgrind output, detects undesirable conditions and present the output in the user-friendly way. The following two sections describe the parts of our tool that deal with detecting granularity issues and hotspot detection, respectivelly.

6.1 Granularity Analysis

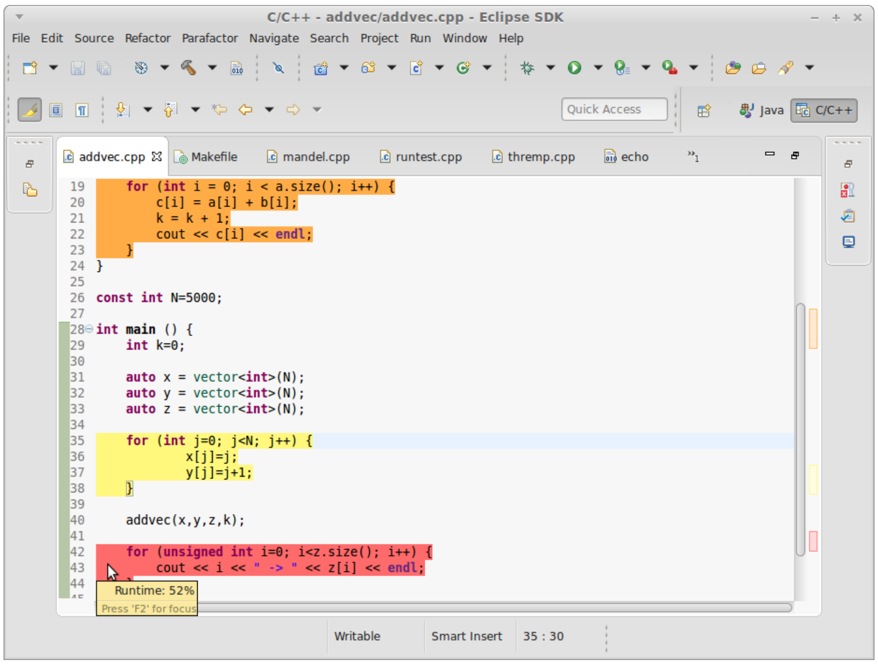
Granularity analysis part of the ParaFormance refactoring tool detects the loops in the sequential or parallel code and calculates how much of the execution time is spent in each of them. A command line tool runs Callgrind on the user application, analyses its output, parses the source code files extracting the loops from them and calculating the accumulated cost (in terms of a number of instructions that they use) for each of these loop. This tool is linked to the ParaFormance refactoring tool, so that the programmer can see highlighted loops in their source code, together with visual indication of how much of the execution time program spends in these loops. This information can be used both to detect the hotspots for parallelisation, as it detects the most expensive loops in the application on which parallelisation effort should be focused, and for detecting the graularity issues in paralle programs. The latter is possible because we can obtain cost information for individual threads in a parallel program, so we can detect issues that come from, for example wrapping relatively inexpensive for loops into Parallel For pattern, which can result into poor performance due to overheads in creating and managing Parallel For thread outweighting the benefits of parallelisation.

Below is an example of the output given by the command line tool for the Ant Colony example. Each line in the output shows the information about one loop of the application, with the columns being, respectivelly, the file where the line is, line numbers in the file where the loop starts and ends and the percentage of the execution time spent in the loop.

```
"ant_seq.cpp" 265 271 0.992
"ant_seq.cpp" 266 268 0.988
"ant_seq.cpp" 127 167 0.985
"ant_seq.cpp" 130 135 0.463
"ant_seq.cpp" 96 101 0.308
"ant_seq.cpp" 141 146 0.058
"ant_seq.cpp" 218 224 0.004
"ant_seq.cpp" 219 223 0.003
"ant_seq.cpp" 84 88 0.002
"ant_seq.cpp" 182 188 0.001
"ant_seq.cpp" 85 87 0.001
"ant_seq.cpp" 170 172 0.000
"ant_seq.cpp" 123 125 0.000
"ant_seq.cpp" 75 77 0.000
"ant_seq.cpp" 69 71 0.000
"ant_seq.cpp" 72 74 0.000
"ant_seq.cpp" 80 82 0.000
"ant_seq.cpp" 226 228 0.000
"ant_seq.cpp" 237 242 0.000
```

The lines 265 – 271 of the `ant_seq.cpp` file are shown below:

```
for (j=0; j<num_iter; j++) {
  for (i=0; i<num_ants; i++) {
    cost[i] = solve (i);
  }
  best_t = pick_best(&best_result);
  update(best_t, best_result);
}
```



```
    for (int i = 0; i < a.size(); i++) {
        c[i] = a[i] + b[i];
        k = k + 1;
        cout << c[i] << endl;
    }
}

const int N=5000;

int main () {
    int k=0;

    auto x = vector<int>(N);
    auto y = vector<int>(N);
    auto z = vector<int>(N);

    for (int j=0; j<N; j++) {
            x[j]=j;
            y[j]=j+1;
    }

    addvec(x,y,z,k);

    for (unsigned int i=0; i<z.size(); i++) {
        cout << i << " -> " << z[i] << endl;
```

Figure 6.1: Visualisation of granularity analysis in the ParaFormance refactoring tool

The output of the granularity analysis suggests that the parallelisation effort should focus on the inner for loop in the code above, where the function `solve` is executed a number of times, since this loops accounts for 98% of the program execution time. This also mean that trying to parallelise the `pick_best` or `update` functions is not going to yield any performance improvement, as their cost is negligable.

Figure 6.1 shows the visualistaion of the granularity analysis in the ParaFormance refactoring tool. As we can see, all loops in the code are coloured according to the amount of time the program spends in them during execution. The most expensive loops, where there is the biggest potential for parallelisation, are coloured in red. On the image, the loop in red takes 52% of the execution time. Loops that take less execution time, but that are still worth parallelising are coloured in orange (e.g. the one on the picture takes 34% of the execution time). Finally, the loops that only account for marginal proportion of execution time are coloured in yellow.

6.2 Communication hotspot detection

We have also extended the ParaFormance refactoring tool with a mechanism for detecting communication hotspots in parallel code. This analysis can detect instructions that result in locking of a bus, such as various synchronisation primitives in threaded code (e.g. `pthreads_mutex_lock`). In this way, we can discover at which point in the code most of the synchronisation is happening. While this is most useful for code that uses low-level synchronisation, such as purely threaded code, it is also useful for patterned applications as the synchronisation bottlenecks can happen at unexpected places, such as calls to library functions (e.g. `rand`).

Below is the output of the detection of communication hotspots on an example application, Fluid Animation taken from a ParSec benchmark suite of parallel programs. It shows the file name, line lumber of a synchronisation command and percentage of bus locking events that the command accounts for.

```
"pthreads.cpp" 1160 99.710
"pthreads.cpp" 1139 48.399
"pthreads.cpp" 1135 48.399
"pthreads.cpp" 846 12.111
"pthreads.cpp" 844 12.111
"pthreads.cpp" 744 12.111
"pthreads.cpp" 742 12.111
"pthreads.cpp" 837 12.089
"pthreads.cpp" 835 12.089
"pthreads.cpp" 735 12.089
"pthreads.cpp" 733 12.089
"pthreads.cpp" 1131 2.655
"pthreads.cpp" 616 1.328
"pthreads.cpp" 604 1.328
```

We can see that 99% of the synchronisation occurs on line 1160, which is a function call. Further analysis reveals that the bottlenecks lie in four `pthread_mutex` function calls on lines 742, 744, 844 and 846. Below is a code snipet for one of them:

```
if( border[indexNeigh]) {
  pthread_mutex_lock(&mutex[iN][iparN % MUTS]);
  neigh->a[iparN % PARTS] -= acc;
  pthread_mutex_unlock(&mutex[iN][iparN % MUTS]);
}
```

The complete code for the application contains 315 routines that can cause synchronisation problems, so using the ParaFormance tool we are able to isolate just 4 of these that are responsible for most of the synchronisation problems in the code.

7. Review of tools for detecting catastrophic failures

In this chapter we review three well-known tools for failure detection of parallel applications. In the following we summarize features and limitations of DRD and Helgrind, from Valgrind, and ThreadSanitizer, a tool within the Clang compiler part and from the LLVM infrastructure.

7.1 Related work

In this section we review a few additional works about race detection tools and techniques. In general, they can be classified into two different groups: static and dynamic. Static algorithms base their results on a static analysis of the application' source code using compiler internals, i.e., Abstract Syntax Tree (AST) to analyze dependencies between data structures and synchronization operations [10,22]. On the contrary, dynamic approaches produce results at run-time using Lamport's happens-before relation, checking for conflicting memory accesses from different threads without any synchronization mechanism [12]. Many examples of tools that fall in this group can be found in the literature [14, 15, 17]. Two consolidated examples of applications that detect errors in multithreaded C and C++ programs using the POSIX threading primitives are DRD [20] and Helgrind [21]. However they are implemented on the top of Valgrind, i.e., driven by a simulator, and thus, making them slower than other approaches. A faster alternative is ThreadSanitizer [19], a race detector that leverages LLVM's infrastructure to instrument code at compile-time and detect races at run time, without performing any simulation.

7.2 Helgrind

Helgrind is a Valgrind tool for detecting synchronization errors in C, C++ and Fortran programs that use the POSIX p-threads threading primitives. More information about Helgrind can be found on [21].

7.2.1 Detection of errors

Helgrind can detect the following errors of errors:

- **Data races.** Helgrind is capable of encountering data races in multithreaded applications by using the happens-before relation. Listing 7.2.1 shows an example of warnings printed by Helgrind when it detects a data race. Basically, it shows the stack traces of the threads involved in the data race. Furthermore it also identifies the name, the size, the address and the location in the code.

```
Thread #1 is the program's root thread

Thread #2 was created
   at 0x511C08E: clone (in /lib64/libc-2.8.so)
   by 0x4E333A4: do_clone (in /lib64/libpthread-2.8.so)
   by 0x4E33A30: pthread_createGLIBC_2.2.5 (in /lib64/libpthread-2.8.so)
   by 0x4C299D4: pthread_create* (hg_intercepts.c:214)
   by 0x400605: main (simple_race.c:12)

Possible data race during read of size 4 at 0x601038 by thread #1
Locks held: none
   at 0x400606: main (simple_race.c:13)

This conflicts with a previous write of size 4 by thread #2
Locks held: none
   at 0x4005DC: child_fn (simple_race.c:6)
   by 0x4C29AFF: mythread_wrapper (hg_intercepts.c:194)
   by 0x4E3403F: start_thread (in /lib64/libpthread-2.8.so)
   by 0x511C0CC: clone (in /lib64/libc-2.8.so)
```

  Internally, Helgrind builds a directed acyclic graph represented the collective happens-before dependencies and monitors memory accesses in order to detect data races. If a location is accessed by two different threads, but Helgrind cannot find any path through the happens-before graph from one access to the other, then it reports a race.

- **Misuses of the POSIX p-threads API.** Helgrind intercepts calls to many POSIX p-threads functions, and is therefore able to report on various common problems. The detected errors are:
  - Unlocking an invalid mutex, a not-locked mutex or a mutex held by a different thread.
  - Destroying an invalid or a locked mutex.
  - Recursively locking a non-recursive mutex.
  - Deallocation of memory that contains a locked mutex.
  - Passing mutex arguments to functions expecting reader-writer lock arguments, and vice versa.
  - When a POSIX pthread function fails with an error code that must be handled.
  - When a thread exits whilst still holding locked locks.
  - Calling `pthread_cond_wait` with a not-locked mutex, an invalid mutex, or one locked by a different thread.
  - Inconsistent bindings between condition variables and their associated mutexes.
  - Invalid or duplicate initialization of a pthread barrier.
  - Initialization of a pthread barrier on which threads are still waiting.

  - Destruction of a pthread barrier object which was never initialized, or on which threads are still waiting.
  - Waiting on an uninitialized pthread barrier.
  - For all of the p-threads functions that Helgrind intercepts, an error is reported, along with a stack trace, if the system threading library routine returns an error code, even if Helgrind itself detected no error.

- **Deadlocks.** For the detection of deadlocks Helgrind monitors the order in which threads acquire locks. This allows it to detect potential deadlocks which could arise from the formation of cycles of locks. Detecting such inconsistencies is useful because, whilst actual deadlocks are obvious, potential deadlocks may never be discovered during testing and could later lead to hard-to-diagnose failures. To detect this issues, Helgrind builds a directed graph indicating the order in which locks have been acquired in the past. When a thread acquires a new lock, the graph is updated, and then checked to see if it now contains a cycle. The presence of a cycle indicates a potential deadlock involving the locks in the cycle. In general, Helgrind chooses two locks involved in the cycle and shows how their acquisition ordering has become inconsistent. Listing 7.2.1 shows a report from an example involving two locks causing a deadlock situation.

```
Thread #1: lock order "0x7FF0006D0 before 0x7FF0006A0" violated

Observed (incorrect) order is: acquisition of lock at 0x7FF0006A0
   at 0x4C2BC62: pthread_mutex_lock (hg_intercepts.c:494)
   by 0x400825: main (tc13_laog1.c:23)

 followed by a later acquisition of lock at 0x7FF0006D0
   at 0x4C2BC62: pthread_mutex_lock (hg_intercepts.c:494)
   by 0x400853: main (tc13_laog1.c:24)

Required order was established by acquisition of lock at 0x7FF0006D0
   at 0x4C2BC62: pthread_mutex_lock (hg_intercepts.c:494)
   by 0x40076D: main (tc13_laog1.c:17)

 followed by a later acquisition of lock at 0x7FF0006A0
   at 0x4C2BC62: pthread_mutex_lock (hg_intercepts.c:494)
   by 0x40079B: main (tc13_laog1.c:18)
```

  Furthermore, Helgrind can detect situations where there are more than two locks in the cycle. A current Helgrind limitation is that it does not show the locks involved, sometimes because that information is not available.

7.2.2 Features

7.2.2.1 Code instrumentation

It is also possible to mark up the effects of thread-safe reference counting using the `ANNOTATE_HAPPENS_BEFORE`, `ANNOTATE_HAPPENS_AFTER` and `ANNOTATE_HAPPENS_BEFORE_FORGET_ALL`, macros. More information about Helgrind instrumentation can be found on [21].

7.2.2.2 Debugging OpenMP

Helgrind supports OpenMP from GCC versions 4.2 and 4.3. However, the GNU OpenMP runtime library constructs its own synchronization primitives using combinations of atomic memory instructions and the futex syscall, which is not supported by Helgrind as it cannot see them. This can be solved using a configuration-time option for disabling Linux futex.

7.2.3 Requirements and limitations

The requirements and limitations of Helgrind can be listed as follows:

- It currently supports glibc-2.3 or later, i.e., it supports the NPTL threading implementation. Older LinuxThreads implementations are not supported.
- It is aware of all the pthread abstractions and tracks their effects as accurately as it can. On x86 and amd64 platforms, it understands and partially handles implicit locking arising from the use of the `LOCK` instruction prefix. On PowerPC/POWER and ARM platforms, it partially handles implicit locking arising from load-linked and store-conditional instruction pairs.
- It works best when the application uses only the POSIX p-threads API. However, if custom or other threading primitives are used, their behavior should be described using Helgrind `ANNOTATE_*` macros defined in `helgrind.h`.
- It is advisable to avoid POSIX condition variables. They should be better replaced by POSIX semaphores to do inter-thread event signalling. Otherwise, Helgrind may miss some inter-thread synchronization events and report false positives.
- If the application is using thread local variables, Helgrind might report false positive race conditions on these variables, despite being of very probably race free. On Linux, the option `-sim-hints=deactivate-pthread--stack-cache-via-hack` can be used to avoid such false positive error messages.
- Helgrind tracks the state of memory in detail, and memory management bugs in the application are liable to cause confusion. It is recommended to use application Memcheck-clean before using Helgrind.
- Slowdowns using Helgrind are in the order of 100×.

7.3 DRD

DRD is a Valgrind tool for detecting errors in multithreaded C and C++ programs. The tool works for any program that uses the POSIX threading primitives or that uses threading concepts built on top of the POSIX threading primitives. While Helgrind can detect locking order violations, for most programs DRD needs less memory to perform its analysis. More information about DRD can be found on [20].

7.3.1 Detection of errors

The DRD tool works for any program that uses the POSIX threading primitives or that uses threading concepts built on top of the POSIX threading primitives and is capable of detecting the following problems:

- **Data races**. The algorithm used by DRD to detect data races is based on the happens-before algorithm. In the way the tool has been implemented, DRD prints a message every time it detects a data race. Listing 7.1 shows an example of warnings printed by DRD when it detects a data race. Basically the number in the column of the left hand side is the process ID of the process being analyzed by the tool. The first lines indicate the thread causing the data race and the kind of operation performed (load or store) along with the start address and the number of bytes involved in the conflicting accesses. Next, the call stack of the conflicting access is also displayed. Furthermore, DRD provides information about the allocation context for the conflicting variables and the data allocated in the calling stack.

Listing 7.1: Example of warnings printed by DRD for data races.

```
$ valgrind --tool=drd --read-var-info=yes drd/tests/rwlock_race
...
==9466== Thread 3:
==9466== Conflicting load by thread 3 at 0x006020b8 size 4
==9466==    at 0x400B6C: thread_func (rwlock_race.c:29)
==9466==    by 0x4C291DF: vg_thread_wrapper (drd_pthread_intercepts.c:186)
==9466==    by 0x4E3403F: start_thread (in /lib64/libpthread-2.8.so)
==9466==    by 0x53250CC: clone (in /lib64/libc-2.8.so)
==9466== Location 0x6020b8 is 0 bytes inside local var "s_racy"
==9466== declared at rwlock_race.c:18, in frame #0 of thread 3
==9466== Other segment start (thread 2)
==9466==    at 0x4C2847D: pthread_rwlock_rdlock* (drd_pthread_intercepts.c:813)
==9466==    by 0x400B6B: thread_func (rwlock_race.c:28)
==9466==    by 0x4C291DF: vg_thread_wrapper (drd_pthread_intercepts.c:186)
==9466==    by 0x4E3403F: start_thread (in /lib64/libpthread-2.8.so)
==9466==    by 0x53250CC: clone (in /lib64/libc-2.8.so)
==9466== Other segment end (thread 2)
==9466==    at 0x4C28B54: pthread_rwlock_unlock* (drd_pthread_intercepts.c:912)
==9466==    by 0x400B84: thread_func (rwlock_race.c:30)
==9466==    by 0x4C291DF: vg_thread_wrapper (drd_pthread_intercepts.c:186)
==9466==    by 0x4E3403F: start_thread (in /lib64/libpthread-2.8.so)
==9466==    by 0x53250CC: clone (in /lib64/libc-2.8.so)
...
```

Listing 7.2: Example of warnings printed by DRD for data races.

```
$ valgrind --tool=drd --exclusive-threshold=10 drd/tests/hold_lock -i 500
...
==10668== Acquired at:
==10668==    at 0x4C267C8: pthread_mutex_lock (drd_pthread_intercepts.c:395)
==10668==    by 0x400D92: main (hold_lock.c:51)
==10668== Lock on mutex 0x7fefffd50 was held during 503 ms (threshold: 10 ms).
==10668==    at 0x4C26ADA: pthread_mutex_unlock (drd_pthread_intercepts.c:441)
==10668==    by 0x400DB5: main (hold_lock.c:55)
...
```

- **Improper use of the POSIX threads API**. In some cases, implementations of the POSIX threads API have been optimized for runtime speed and do not consider certain errors, e.g. when a mutex is being unlocked by another thread than the thread that obtained a lock on the mutex. The DRD tool is able to detect and report the following misuses of the POSIX threads API:
  - Passing the address of one type of synchronization object (e.g. a mutex)

to a POSIX API call that expects a pointer to another type of synchronization object (e.g. a condition variable).
  - Attempts to unlock a mutex that has not been locked or locked by another thread.
  - Attempts to lock a mutex of type `PTHREAD_MUTEX_NORMAL` or a spinlock recursively.
  - Destruction or deallocation of a locked mutex.
  - Sending a signal to a condition variable while no lock is held on the mutex associated with the condition variable.
  - Calling `pthread_cond_wait` on a mutex that is not locked, that is locked by another thread or that has been locked recursively.
  - Associating two different mutexes with a condition variable through `pthread_cond_wait`.
  - Destruction or deallocation of a condition variable that is being waited or a locked reader-writer synchronization object.
  - Attempts to unlock a reader-writer synchronization object that was not locked by the calling thread.
  - Attempts to recursively lock a reader-writer synchronization object exclusively.
  - Attempts to pass the address of a user-defined reader-writer synchronization object to a POSIX threads function.
  - Attempts to pass the address of a POSIX reader-writer synchronization object to one of the annotations for user-defined reader-writer synchronization objects.
  - Reinitialization of a mutex, condition variable, reader-writer lock, semaphore or barrier.
  - Destruction or deallocation of a semaphore or barrier that is being waited upon.
  - Missing synchronization between barrier wait and barrier destruction.
  - Exiting a thread without first unlocking the spinlocks, mutexes or reader-writer synchronization objects that were locked by that thread.
  - Passing an invalid thread ID to `pthread_join` or `pthread_cancel`.

- **Lock contention**. DRD is also able to check for lock contention, i.e., when a thread blocks the progress of one or more other threads by holding a lock too long. Listing 7.2 shows an example of DRD report due an excessive lock contention in a given thread. The report describes that the lock acquired at line 51 in source file `hold_lock.c` and released at line 55 was held during 503 ms for a threshold of 10 ms.

7.3.2 Features

7.3.2.1 Code instrumentation

As for other Valgrind tools, it is possible interact with the DRD tool through client requests, however it requires to instrument the code in order to guide such requests. Specifically, DRD define a set of macros in the header file `<valgrind/drd.h>` so as to instrument the user code and aid DRD during the detection of errors. For example, the macro `ANNOTATE_HAPPENS_BEFORE(addr)` identifies that a variable will be accessed before the same from another thread, and thus, it allows to improve DRD happens-before algorithm for data race detection. Other macros, for instance, allow the definition of barriers `ANNOTATE_BARRIER_INIT`, while others, such as `DRD_TRACE_VAR(x)`, let the user to trace specific variables. More information about these macros can be found on the DRD website [20].

7.3.2.2 Debugging OpenMP and C++11 applications

DRD supports the use of OpenMP and C++11 (along with the C++11 class `std::thread`) multithreaded applications. For that case, DRD requires to annotate the `std::shared_ptr<>` objects used in the implementation of that class.

7.3.3 Requirements and limitations

The requirements and limitations of DRD can be listed as follows:

- DRD requires that Linux distributions contain symbol information in `ld.so`.
- DRD requires the use of `gcc` version 3.0 or later, older versions are not supported.
- Using `gcc` version 4.4.3 and before, DRD may report data races on the C++ class `std::string` in a multithreaded program.
- Only the NPTL (Native POSIX Thread Library) POSIX threads implementation is supported, older POSIX implementations are not supported.
- DRD requires, by default, between 1.1–3.6× more memory compared to a native run of the client program. More memory may be required if debug information has been enabled. On the other hand, DRD allocates some of its temporary data structures on the stack of the client program threads, this amount of data varies between 1 and 2 KB.
- Slowdowns using DRD range between 20–50× compared with a single-threaded run. These slowdowns may be noticeable for applications that perform frequent mutex lock/unlock operations.

7.4 ThreadSanitizer

ThreadSanitizer (TSan) is a data race detector that works together with the Clang C/C++ from the LLVM infrastructure. TSan instruments the application code at compile time and allows, at execution time, to check all non-race-free memory access. Contrary to tools that work along with Valgrind and need to perform a simulation run to detect failures, such as Helgrind or DRD, TSan takes advantage of the compile-time instrumentation to operate right away at runtime and detect potential data races. More information about TSan can be found on the developers site [13] and [19].

7.4.1 Detection of errors

- **Data races.** TSan can detect a variety of data races: normal data races, races on C++ object `vptr`, use after free races, races on mutexes, races on file descriptors, races on `pthread_barrier_t`, leaked threads, signal-unsafe malloc/free calls in signal handlers, signal handlers spoiling `errno`. TSan leverages detection algorithms to track both lock-sets and the happens-before relations, allowing to switch between the pure happens-before and the hybrid modes.

  When TSan detects a bug it prints a report and its format differ depending on bug type. Listing 7.4.1 contains a set of blocks describing the threads causing the data race containing the thread ID, the stack frame with the function, file name and line and column, if available. The report also contain a description of the conflicting memory accesses, being read or writes. Note that the first memory accesses is the current access causing the data race, while the second is the previous memory access. Race happening on heap memory locations also contain the allocation address and parameters of the heap block.

```
WARNING: ThreadSanitizer: data race (pid=9337)
  Write of size 4 at 0x7fe3c3075190 by thread T1:
    #0 foo1() simple_stack2.cc:9 (exe+0x000000003c9a)
    #1 bar1() simple_stack2.cc:16 (exe+0x000000003ce4)
    #2 Thread1(void*) simple_stack2.cc:34 (exe+0x000000003d99)

  Previous read of size 4 at 0x7fe3c3075190 by main thread:
    #0 foo2() simple_stack2.cc:20 (exe+0x000000003d0c)
    #1 bar2() simple_stack2.cc:29 (exe+0x000000003d74)
    #2 main simple_stack2.cc:41 (exe+0x000000003ddb)

  Thread T1 (tid=9338, running) created at:
    #0 pthread_create tsan_interceptors.cc:683 (exe+0x00000000de83)
    #1 main simple_stack2.cc:40 (exe+0x000000003dd6)
```

- **Deadlocks.** The current Clang trunk version of TSan has an experimental detector of lock order inversions (potential deadlocks). Only `pthread_mutex`, `pthread_rwlock` and `pthread_spin` are supported but only partially tested. Also, the bug reports are not as informative as they could be. The algorithm internally used for deadlock detection maintains a directed graph of lock acquisitions. If a lock B is acquired while a lock A is being held by the same thread, a directed edge A $\Rightarrow$ B is added to the lock acquisition graph. A potential deadlock is reported when there is a cycle in the graph. To use the TSan lock detector, one should enable `TSAN_OPTIONS=detect_deadlocks=1` when running a TSan-instrumented program.

7.4.2 Features

7.4.2.1 Supressions

TSan supports suppressions, i.e., if there is a bug report that cannot be fixed right away, it may be useful to temporary suppress it. A suppressions file can be created using a specific file TSan file. There are different kind of suppressions that can be treated for: data races and use-after-free reports, threads, mutexes (destruction of a locked mutex), signal handlers (handler calls malloc()), and lock inversion reports.

7.4.2.2 Debugging OpenMP and C++11 applications

TSan supports 64-bit architectures using POSIX threads and C++11 threading (within the LLVM `libc++` library) as parallel execution models, being compliant with pthread synchronization primitives, compiler-built-in atomics and synchronization C++11 primitives. However, it is not yet compliant with C++ exceptions.

7.4.3 Requirements and limitations

The requirements and limitations of TSan can be listed as follows:

- It is supported only on Linux x86_64 along with the Clang C/C++ and `gcc` 4.8 compilers.
- TSan does not support to statically link `libc/libstdc++` libraries into the program. They should be linked dynamically.
- The cost of race detection varies by program, on a typical application, overheads are about 2–20$\times$. It is recommended to use the `-O2` flag to get reasonable performance.
- Overheads for memory usage are in the rage of 5–10$\times$, as it maps (but does not reserve) virtual address space.
- TSan needs shadow memory for application memory, so TSan maps it at startup, and then access it when the application accesses own memory. The shadow memory is 4–8$\times$ in size relative to application memory.

8. Conclusions

This deliverable introduced prototype set of tools for improving reliability, robustness and integrity of parallel software. With these tools, we aim to address critical issues of debugging, testing and verification of parallel applications, tasks that are currently very hard due to the complexity of parallelism and dependence of application behaviour on extrenal events such as variable system load. Another issue that we are adressing is the identification of problems in the application code that can lead to the violation of *extra-functional* properties, such as finishing the execution in a given time limit or staying within a given energy budget. Violation of these properties can render applications completely unusable.

We introduced the IBM Focus (Chapter 2) and ExpliSAT (Chapter 3), tools for test planning and verification. Currently, these tools work on a sequential code, but we are in the process of extending them to parallel patterned programs. We have also described the initial work of extending the static code analysis tool QA-Verify (Chapter 4), developed by PRQA, that will enable us to verify that the parallel code conforms to the common coding standards, as well as the standards that will be developed in the remainder of the **RePhrase** project.

One of the key problems with developing parallel code is ensuring that the code is free of parallelism-specific issues and bugs, such as race conditions and deadlocks. In this deliverable, we described the state-of-the-art in tools for race condition detection (Chapter 7), including the detailed description of DRD, Helgrind and ThreadSanitizer, pointed out to the proper way to use them and to some of their drawbacks. We then described the extensions to the ParaFormance refactoring tool that enables detection of race conditions in parallel code (Chapter 5) and detection of granularity problems and excessive communication (Chapter 6), which can lead to violation of extra-functional properties of parallel applications.

Bibliography

[1] Cyrille Artho, Klaus Havelund, and Armin Biere. High-level data races. *Software Testing, Verification and Reliability*, 13(4):207–227, 2003.

[2] S. Ashby and *et al.* The opportunities and challenges of Exascale computing. Summary Report of the Advanced Scientific Computing Advisory Committee (ASCAC) Subcommittee, November 2010.

[3] S. Borkar. The exascale challenge. In *VLSI Design Automation and Test (VLSI-DAT), 2010 International Symposium on*, pages 2–3, April 2010.

[4] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *ACM SIGPLAN Notices*, volume 37, pages 211–230. ACM, 2002.

[5] Cristian Cadar, Daniel Dunbar, and Dawson R Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[6] Cristian Cadar, Patrice Godefroid, Sarfraz Khurshid, Corina S Păsăreanu, Koushik Sen, Nikolai Tillmann, and Willem Visser. Symbolic execution for software testing in practice: preliminary assessment. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 1066–1071. ACM, 2011.

[7] Hana Chockler, Dmitry Pidan, and Sitvanit Ruah. Improving representative computation in ExpliSAT. In *Hardware and Software: Verification and Testing*, pages 359–364. Springer, 2013.

[8] Jee Choi, M. Dukhan, Xing Liu, and R. Vuduc. Algorithmic time, energy, and power on candidate hpc compute building blocks. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 447–457, May 2014.

[9] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In Kurt Jensen and Andreas Podelski, editors, *Tools and*

Algorithms for the Construction and Analysis of Systems (TACAS 2004), volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.

[10] Dawson Engler and Ken Ashcraft. RacerX: Effective, Static Detection of Race Conditions and Deadlocks. *SIGOPS Oper. Syst. Rev.*, 37(5):237–252, October 2003.

[11] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed automated random testing. In *ACM Sigplan Notices*, volume 40, pages 213–223. ACM, 2005.

[12] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[13] LLVM-Project. ThreadSanitizer, a tool to detect data races. `http://clang.llvm.org/docs/ThreadSanitizer.html`, 2016.

[14] Daniel Marino, Madanlal Musuvathi, and Satish Narayanasamy. Literace: Effective sampling for lightweight data-race detection. *SIGPLAN Not.*, 44(6):134–143, June 2009.

[15] Robert O'Callahan and Jong-Deok Choi. Hybrid dynamic data race detection. *ACM SIGPLAN Notices*, 38(10):167–178, 2003.

[16] Corina S Păsăreanu and Willem Visser. A survey of new trends in symbolic execution for software testing and analysis. *International journal on software tools for technology transfer*, 11(4):339–353, 2009.

[17] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems (TOCS)*, 15(4):391–411, 1997.

[18] Koushik Sen and Gul Agha. CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools. In *Computer Aided Verification*, pages 419–423. Springer, 2006.

[19] Konstantin Serebryany, Alexander Potapenko, Timur Iskhodzhanov, and Dmitriy Vyukov. Dynamic Race Detection with LLVM Compiler. In *Proceedings of the Second International Conference on Runtime Verification*, RV'11, pages 110–114, Berlin, Heidelberg, 2012. Springer-Verlag.

[20] Valgrind-Project. DRD: A Thread Error Detector. `http://valgrind.org/docs/manual/drd-manual.html`, 2009.

[21] Valgrind-Project. Helgrind: A Data-Race Detector. `http://valgrind.org/docs/manual/hg-manual.html`, 2009.

[22] Jan Wen Voung, Ranjit Jhala, and Sorin Lerner. RELAY: Static Race Detection on Millions of Lines of Code. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE ’07, pages 205–214, New York, NY, USA, 2007. ACM.