Project no. 644235

REPHRASE

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

Combined report describing testing, verification, catastrophic failures detection and prop- erties violation detection for the initial set of patterns D3.2

Due date of deliverable: 30.09.2016

Start date of project: April 1^{st}, 2015

Type: Deliverable
WP number: WP3

Responsible institution: IBM
Editor and editor's address: 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 second deliverable from WP3 “Reliability, Robustness and Software Integrity of Parallel Software”. It presents the scientific progress beyond the state-of-the-art, including the novel algorithms implemented in the set of tools presented in deliverable D3.1. The report covers methods for *i)* testing functional and extra-functional properties of parallel data-intensive software of the **RePhrase** project against their requirements; *ii)* detecting catastrophic failures in parallel data-intensive applications, such as deadlocks and race conditions; *iii)* providing validation and verification mechanisms for parallel implementations of patterns; *iv)* providing methodology for equivalence checking between the program before and after **RePhrase** refactoring; *v)* supporting quality assurance; and, *vi)* detecting reasons for violations of extra-functional properties of applications, such as performance.

The deliverable is the result of the second 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*; and,
- *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*.

Contents

1. Introduction

Parallel patterns [8] represent a very useful programming model for bridging the gap between the complexity of modern (heterogeneous) multi-core/many-core hardware and the high-level of abstraction that programmers want to use in programming this hardware. While they allow programmers to avoid many concurrency bugs that are almost unavoidable with lower-level models, such as pthreads, it is still possible to introduce these kind of bugs in the resulting parallel code, even if patterns are the only source of parallelism. Therefore, automated tools for detecting the most common concurrency bugs, such as data races and deadlocks, are still critical to the development of parallel software. The adversity in finding data races and deadlocks is a well known problem [5]. 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 the debugger's task in finding these issues, they are still not perfect [4,5].

An additional problem for the development of parallel software is that most of the state-of-the-art tools for *testing* and *verification* of parallel software are still aimed at sequential applications. Other than the work we are undertaking in the **RePhrase** project, there is not much work in extending these mechanisms to the parallel settings, dealing with specific issues that arise there, such as non-deterministic ordering of events in multi-threaded setting. Finally, even when the parallel code is free of concurrency bugs and has been propertly tested/verified, we still do not have guarantees that its extra-functional properties, such as performance or energy consumption, will be acceptable. A seemingly "perfect" parallel program can often run even slower than its sequential equivalent due to various problems, such as too fine granularity, load imbalance due to too coarse granularity, bad placement of data, cache problemts etc. These problems do not impact the semantics of an application, so the result of the execution is still valid, but can severely impact its performance. Therefore, tools that can pinpoint to the reasons for violations of extra-functional properties of parallel applications would be an invaluable in a tool-chain for parallel software development.

In the deliverable D3.1, we have described a basic **RePhrase** infrastructure for testing and verification of patterned applications, static analysis of the patterned code for ensuring that it conforms to code standards, detection of data races, and detection of reasons for violation of extra-functional properties of the parallel application. In this deliverable, we report how different parts of this infrastructure can be applied both to *parallel pattern libraries*, such as Intel TBB and FastFlow, and to the parallel code obtained by using these libraries on sequential code. In this way, are make several contributions to the state-of-the art in detecting failures and testing/verification of parallel applications:

- we demonstrate the use of QAC++ and QAV tools on the state-of-the-art parallel pattern libraries Intel TBB and FastFlow, detecting potential problems in their latest versions and proposing mechanisms for fixing these problems;
- we propose a test generation method for patterned applications and demonstrate it using IBM FOCUS tool on the Mandelbrot benchmark application;
- we describe an algorithm for verification of the equivalence between the sequential and parallel version of the code, whereas parallel version is obtained using refactorings described in D2.2, and its application on the Mandelbrot benchmark application;
- we describe novel mechanisms for detecting race conditions in parallel applications that use Single-Produces/Single-Consumer parallel data structure (e.g. FastFlow parallel library) and its integration into the TSan race detection tool;
- we demonstrate the usage of the PAPI library for the use of performance counters for detection of violation of extra-functional application properties.

This document is organized as follows. Chapter 2 provides analysis of the FastFlow and Intel TBB libraries using the QA-Verify tools and tests whether they conform to the **RePhrase** coding standards described in D5.2. Chapter 3 describes the combinatorial test design (CDT) planning techinque and its application, using the IBM FOCUS tool, for generating tests for patterned applications. Chapter 4 describes a verification method for a code refactored to use the parallel patterns and demonstrate it using the IBM ExpliSAT testing tool. We demonstrate how we can chech whether the refactored application is equivalent to the original one. Chapter 6 describes the use of the PAPI library for detection of violation of extra-functional properties of patterned applications. We restrict our attention to the patterns that belong to the initial **RePhrase** pattern set, described in D2.1. Finally, Chapter 8 concludes.

2. Verification of Pattern Libraries

2.1 Introduction

Libraries such as Thread Building Blocks (TBB) and FastFlow (FF) provide abstractions to aid developers in the creation of parallel programs. The quality of the overall software system is the sum of the quality of the code using the library and the library itself. In this chapter, we describe the analysis of the FF and TBB libraries, in terms of whether they conform to the RePhrase coding standard rules described in D5.2. We use a prototype version of the QA-Veirfy tool for multi-threaded applications, described in D3.1. This analysis also highlights false positives (false alarms) with the mapping or implementation currently provided by the rules.

2.2 Methodology

We have analysed the libraries and reviewed violations of the set of RePhrase coding standard. Our goal was to determine:

1. the status of the violation, ie. true or false positive diagnostic;
2. the severity of any true positive diagnostics identified;
3. what can be done to rectify or reduce the number of mapping false positives;
4. a possible set of changes to the RePhrase coding standard, the tool diagnostic mapping, and the library source code.

The review of diagnostics in steps 1 and 2 is tabulated with the status of the diagnostics categorised under the following headings:

- Mapping/Tool False Positive (MFP)
- Low Severity True Positive (LSTP)
- High Severity True Positive (HSTP)

Scope Of Review: The selected libraries make considerable use of C++ templates. It is not sufficient to analyse the library code in isolation as this will include uninstantiated template definitions. Due to dependent types and values, analysis of uninstantiated templates is generally only possible at a syntactic level.

The libraries are, therefore, analysed using the examples and tests that are shipped as part of the project. However, analysis results generated directly against the test source and test header files will be excluded.

2.2.1 Coding Standard Enforcement Status

The RePhrase coding standard from deliverable D5.2, lists 43 rules split into 3 sections. These rules have been selected based on correct use of standard library concurrency and parallelism features and the detection of non-hygienic language features [12].

In general, any use of non-hygienic language features will make it more difficult to refactor the code into a parallelised version using the parallel patterns described in deliverable D2.1 as non-hygienic properties will usually violate preconditions of those patterns.

Of the 43 rules, as of 30th August 2016, 17 are automatically enforced with the QAC++ and the QAV analysis tool. The following table shows the rule id and text along with the tool message providing the enforcement:

Table 2.1: RePhrase Coding Standard Tool Enforcement

Rule	***Rule and Tool Diagnostic(s)***
reachable	**Ensure that all statements are reachable.** 2880 *This code is unreachable.* 2991 *The value of this 'if' controlling expression is always 'true'.* 2992 *The value of this 'if' controlling expression is always 'false'.*
bounds	**Ensure that pointer or array access is demonstrably within bounds of a valid object.** 2840 *Constant: Dereference of an invalid pointer value.* 2841 *Definite: Dereference of an invalid pointer value.* 2842 *Apparent: Dereference of an invalid pointer value.* 2843 *Suspicious: Dereference of an invalid pointer value.* 2930 *Constant: Computing an invalid pointer value.* 2931 *Definite: Computing an invalid pointer value.* 2932 *Apparent: Computing an invalid pointer value.* 2933 *Suspicious: Computing an invalid pointer value.*

continued on next page

Table 2.1 – continued from previous page

Rule	*Rule and Tool Diagnostic(s)*
div-by-zero	**Ensure that the right hand operand of the division or remainder operators is demonstrably non-zero.** 2831 *Definite: Division by zero.* 2832 *Apparent: Division by zero.* 2833 *Suspicious: Division by zero.*
lifetime-0	**Do not return a reference or a pointer to an automatic variable defined within the function.** 4026 *This returns a handle to local data.*
lifetime-1	**Do not assign the address of a variable to a pointer with a greater lifetime.** 2515 *This is assigning the address of an object to a pointer that has greater lifetime.*
side-effects	**Do not code side effects into the right-hand operands of: `&&`, `||`, `sizeof`, `alignof`, `typeid` or a function passed to `condition_variable::wait`.** 3230 *The right hand side of this operator has side effects.* 3231 *The right hand side of this operator calls a function.*
return-value	**Ensure that execution of a function with a non-void return type ends in a return statement with a value.** 2888 *This function has been declared with a non-void 'return' type but ends with an implicit 'return ;' statement.*
postpone-declaration	**Postpone variable definitions as long as possible.** 4230 *Scalar type control variable '%1s' not declared here.* 4113 *The declaration of object '%1s' can be moved to this nested scope.*
use-const	**Use `const` whenever possible.** 2081 *This copy assignment operator has a non-const reference parameter.* 2142 *Prefer to have copy constructors and copy assignments with reference to the const class.* 4213 *This pointer to non const parameter is not modified.* 4283 *This non const reference parameter is not modified.*
use-asm	**Do not use the `asm` declaration.** 1100 *This is an asm declaration.*

continued on next page

Table 2.1 – continued from previous page

Rule	***Rule and Tool Diagnostic(s)***
no-unset	**Do not access an invalid object or an object with indeterminate value.** 2961 *Definite: Using value of uninitialised automatic object '%s'.* 2962 *Apparent: Using value of uninitialised automatic object '%s'.* 2963 *Suspicious: Using value of uninitialised automatic object '%s'.* 2971 *Definite: Passing address of uninitialised object '%s' to a function parameter declared as a pointer to con st.* 2972 *Apparent: Passing address of uninitialised object '%s' to a function parameter declared as a pointer to const.* 2973 *Suspicious: Passing address of uninitialised object '%s' to a function parameter declared as a pointer to const.* 4056 *Uninitialised member used as initialise.* 4058 *Address of uninitialised object used in initialise.* 4231 *The starting value of the scalar type control variable '%1s' is not set in the initialisation expression.* 4238 *The starting value of class type control variable '%1s' is not set in the initialisation expression.*
mk-static-or-const	**Declare `static` any member function that does not require `this`. Alternatively, declare `const` any member function that does not modify the externally visible state of the object.** 4211 *This non const member function does not modify any member data.* 4212 *This non static member function does not access any member data.*

continued on next page

Table 2.1 – continued from previous page

Rule	***Rule and Tool Diagnostic(s)***
encapsulate-mbrs-const	**Do not return non-const handles to class data from const member functions.** 4625 *This function returns a non-const handle to data pointed to by a private or protected member, in a class performing shallow copy.* 4626 *This const function returns a non-const handle to data pointed to by a member, in a class performing shallow copy.* 4627 *This function returns a non-const handle to data pointed to by a private or protected member.* 4628 *This const function returns a non-const handle to data pointed to by a member.*
platform-specific	**Do not use platform specific multi-threading facilities.** 6002 *Do not use platform specific multi-threading facilities*
use-lock-guard	**Objects of type `std::mutex` shall not be accessed directly.** 6001 *Objects of type `std::mutex` shall not be accessed directly.*
use-async-for-tasks	**Use `std::async` instead of `std::thread` for task-based parallelism.** 6003 *Use `std::async` instead of `std::thread` for task-based parallelism*
static-locals	**Shared global data shall be provided through static local objects.** 2310 *This non-local object will be initialised at runtime.*

2.3 Analysis Results

The analysis tool displays the diagnostics generated, if any, against the source code listing of the project. Every rule violation, highlighted by a tool diagnostic was reviewed to determine the category of issue detected. For brevity, only rules for which violations of the coding standard have been detected will be included in the results tables.

2.3.1 Analysis of FF

The FF library was analysed and rule violations on the test code were excluded. The results[1] are shown in table 2.2.

Table 2.2: FastFlow Analysis Results

Rule	***Msg***	***Num. MFP***	***Num. LSTP***	***Num. HSTP***	***Total***
static-locals	2310	0	0	1	1
reachable	2880	2	17	4	23
	2991	0	1	0	1
	2992	0	7	4	11
bounds	2841	4	0	0	4
side-effects	3230	1	6	0	7
	3231	18	7	0	25
postpone-declaration	4113	1	1	0	2
	4230	0	11	0	11
mk-static-or-const	4211	7	19	0	26
	4212	1	20	0	21
use-const	4213	33	8	0	41
	4283	0	23	0	23
no-unset	2961	4	2	0	6
	4283	0	23	0	23
encapsulate-mbrs-const	4625	0	2	0	2
	4626	2	16	0	18
	4627	0	1	0	1
use-lock-guard	6001	0	3	2	5
platform-specific	6002	0	1	0	1

2.3.2 Analysis of TBB

As for FF, the TBB library was analysed and rule violations on the test code were excluded. The results are shown in table 2.3.

Table 2.3: Thread Building Blocks Analysis Results

Rule	***Msg***	***Num. MFP***	***Num. LSTP***	***Num. HSTP***	***Total***
static-locals	2310	11	20	0	31

continued on next page

[1]The full analysis for both FF and TBB can be viewed in the QA Verify tool.

Table 2.3 – continued from previous page

Rule	***Msg***	***Num. MFP***	***Num. LSTP***	***Num. HSTP***	***Total***
reachable	2880	25	22	0	47
	2991	1	10	0	11
	2992	3	20	0	23
bounds	2841	4	0	0	4
	2842	0	1	0	1
side-effects	3230	3	16	0	7
	3231	170	16	0	186
use-const	4213	34	18	0	52
	4283	13	31	0	44
mk-static-or-const	4211	74	41	0	115
	4212	0	99	0	99
postpone-declaration	4113	0	8	0	8
	4230	1	22	0	23
no-unset	2961	10	0	0	10
	2963	1	0	0	1
	4056	0	6	0	6
	4058	1	0	0	1
	4104	0	1	0	1
encapsulate-mbrs-const	4625	2	5	0	7
	4626	3	11	0	14
	4628	0	1	0	1

2.4 Future Work

As a result of pattern library analysis, we have identified the parts of the library implementations that should be improved to make them conform to the RePhrase coding standard and, at the same time, re-evaluate the coding standard based on the benefit of impact of individual rules. We also need to address the problem of false positives reported by our tool and to work on automation of enforcement of the proposed coding rules.

2.4.1 Improvements to Libraries.

The analysis of the TBB library, 2.3.2 uncovered no *High Severity True Positives* of the RePhrase Coding Rules. Addressing *Low Severity True Positives*, in many cases will help improve the maintainability and robustness of the source.

The analysis of the FF library however, 2.3.1, found a few issues that are considered to be *High Severity True Positives*. These were violations of the `reachable`, *use-lock-guard* and `static-locals` rules. The former two being particularly interesting where the target system has limited memory resources available.

Shared global data shall be provided through static local objects: Although not a true violation of the rule in question the logic appears at best overly complex. In the examples in question are two global objects with internal linkage:

```
// fftree.hpp:47-48
static std::mutex treeLock;
static std::set<fftree *> roots;
```

These variables are declared `static` and so every translation unit that includes the header will have a separate copy of the objects. For example:

```
// tu1.cc
#include "fftree.hpp"
void f1 (fftree * t, fftree * c) {
  //
  // 'c' possibly erased from tu1.cc:roots
  t->add_child (c);
}

// tu2.cc
#include "fftree.hpp"
void f1 ((fftree * t, fftree * c);
void f2 (fftree * t, fftree * c) {
  //
  // 'c' possibly erased from tu2.cc:roots
  t->add_child (c);
  f1(t, c);
}
```

Depending on which translation unit calls `add_child` the node `c` may or may not be removed from the correct `roots` object. Using a file local static, as well as guaranteeing synchronised initialisation will also result in the same objects being used from all translation units.

Ensure that all statements are reachable: The `reachable` rule, enforced through diagnostics 2880 and 2992, highlights what at first seems to be a problem introduced through a maintenance change. However, the problem in question has been present from the first revision of the file. The issue is that a test for an allocation failure is being performed against a variable that is already known to have a non-null value:

```
// ff/allocator.cpp:1721 - 1726
if (ptr) {
  // ...
  void * newptr= ::realloc(...
  if (!ptr) return 0;
```

It is more likely that the check for a null is intended to take place against the value returned by `realloc`, ie. `newptr` and not `ptr`.

Objects of type `std::mutex` shall not be accessed directly: Message 6001 highlights calls to the `lock` member of a `std::mutex` type. As for other uses of the *Resource Allocation Is Initialisation (RAII)* idiom, in addition to simplifying the structure of the code, RAII guarantees that when an exception is thrown the resources will be freed.

If the `lock` and `unlock` members of `std::mutex` are accessed directly, then an exception may result in a missed call to `unlock`, which in turn could result in deadlock.

```
// fftree.hpp:77-87
treeLock.lock();
// ...
roots.insert(this);
treeLock.unlock();
```

An exception thrown by `roots.insert(this)` will result in the call to `treeLock.unlock()` not being made, leaving the mutex locked. This code should be changed to use `std::lock_guard`.

2.4.2 Re-evaluate the tool messages used to enforce the rule set.

Messages 4625 to 4628 highlight where a non-const handle is returned from a member function:

4625 *This function returns a non-const handle to data pointed to by a private or protected member, in a class performing shallow copy.*

4626 *This const function returns a non-const handle to data pointed to by a member, in a class performing shallow copy.*

4627 *This function returns a non-const handle to data pointed to by a private or protected member.*

4628 *This const function returns a non-const handle to data pointed to by a member.*

Messages 4625 and 4626 highlight the special case where the tool has determined that the pointer does not refer to a resource. This is achieved by checking how copying takes place, i.e. does the class perform a *shallow* or *bitwise* copy of the member. A *shallow* copied handle returned from a member function is just like any other member and so these messages should be removed from the enforcement of *encapsulate-mbrs-const*.

2.4.3 Re-evaluate the rule set based on the benefit of impact of individual rules.

There are no rules that are recommended for removal in the next version of the RePhrase Coding Standard.

2.4.4 Address mapping and tool false positives to improve the signal to noise ratio.

The following are examples of diagnostics which can be enhanced to improve the the signal to noise ratio for the enforcement of the rules.

3231 ***The right hand side of this operator calls a function.*** Diagnostic 3231 is issued when a function call without side-effects, or an unknown function call is the *conditionally evaluated* operand of a logical operator.

```
bool f1 ( ) { return true; }
bool f2 ( );

void f3 (bool b) {
  b && f1 () ; // 3231
  b && f2 () ; // 3231
}
```

It is acceptable for a function without side-effects to be conditionally evaluated and therefore these functions should not be treated the same as functions for which the effect is unknown.

The suggestion here is to update the tool to differentiate between these cases and then remove the non side-effect case from enforcing the rule.

4211 ***This non const member function does not modify any member data.*** The C++ language enforces `const` at the bitwise level. A consequence of this is that a const member function will still be capable of modifying the program state:

```
struct A {
  int * p;
  void f1 ( ) const
  {
    *p = 0;
  }
};
```

Enforcing logical const increases the semantic information provided by the type, that is, calling a const member will not modify the state of the program. The recommendation is to modify diagnostics such as *This non const member function does not modify any member data* to further distinguish between functions that are logical vs bitwise const.

2.4.5 Increase the automated enforcement of the proposed coding standard rules.

The following RePhrase Coding Standard rule do not yet have automated enforcement:

- General Rules
  - *for-each-loop* Implement a loop that only uses element values using range-for or an STL algorithm

    This removes unhygienic properties from the loop and is usually a precondition for refactoring the loop into parallel code by applying parallel patterns.
- Concurrency Rules

- *specify-async-launch-policy* Always explicitly specify a launch policy for `std::async`.
- *same-lock* Synchronise access to data shared between threads using the same lock.
- *atomic-mutable* Members declared `mutable` and shared between threads shall be `std::atomic`.
- *protect-volatile* Access to volatile data shared between threads shall have appropriate synchronisation.
- *call-once* Use `std::call_once` to ensure a function is called exactly once.
- *lambda-thread-ctor* Use a `noexcept` lambda as the argument to a thread constructor.
- *thread-ctor-lifetime* The lifetime of data passed to the thread constructor must exceed the lifetime of the thread.
- *no-static-deadlock-path* Within the scope of a lock, ensure that no static path results in a lock of the same mutex.
- *locks-form-dag* Ensure that order of nesting of locks in a project forms a DAG.
- *no-recursive-mutex* Do not use `std::recursive_mutex`.
- *mutex-not-dynamic* Objects of type `std::mutex` shall not have dynamic storage duration.
- *relaxed-atomics* Do not use relaxed atomics.
- *no-condition-variable-any* Do not use `std::condition_variable_any` on objects of type `std::mutex`.
- *unlock-mutex-not-held-by-current-thread* Do not unlock a mutex which is not already held by the current thread.
- *do-not-destroy-locked-mutex* Do not destroy a locked mutex.
- *exiting-thread-holding-lock* Before exiting a thread ensure all mutexes held by it are unlocked.

• Parallelism Rules

- *parallel-algorithms* Use higher-level standard facilities to implement parallelism

  Higher-level facilities to implement parallelism are described in deliverable D2.1.
- *non-pure-read-write* Detect functions that are not pure.

  Most of the parallel patterns require functions to be pure as a precondition.

- *hygienic-0* Loops and Functors should not exhibit unhygienic properties.

  Any unhygienic properties in loops or functors for sequential algorithms make it more difficult to refactor the code into a parallel version using parallel patterns.
- *hygienic-1* This hygienic sequential algorithm can be replaced with a parallel version.
- *non-returning-function* Functor used with parallel algorithm shall always return.

  Using a non-returning functor in a parallel algorithm would be violating a pre-condition of the parallel pattern.
- *noexcept-parallel-algorithm-0* Functors used with parallel algorithms shall be `noexcept(true)`.

  Similar to a non-returning functor, throwing an exception from a functor in a parallel algorithm would be violating a pre-condition of the parallel pattern.
- *catch-include-bad-alloc* Catch handlers enclosing algorithms with execution policies shall include `std::bad_alloc`.
- *algorithm-associative-commutative* The `binary_op` used with `std::reduce` or `std::transform_reduce` shall be demonstrably associative and commutative.
- *no-iterator-invalidate-0* The `Function` argument used with an algorithm shall not use a non-const iterator or reference to the container being iterated over.

The remainder of this section suggests some potential heuristics for adding automated enforcement on a selection of these rules.

2.4.6 Heuristics for Implementation of General Rules

for-each-loop The goal is to identify loops that do not depend on a value of the loop control variable and to recommend that they are rewritten using the range based for syntax.

```
void foo(std::vector<int> & v)
{
  for (auto iter = v.begin (), end = v.end ()
       ; iter != end
       ; ++ iter)
  {
    ++(*iter);
  }

  //
  // The above loop is better written as:
  for (auto & e : v)
  {
    ++e;
  }
}
```

Removing the dependency on the loop control variable is a pre-condition for parallelisation of the loop using the parallel map pattern as described in deliverable D2.1.

2.4.7 Heuristics for Implementation of Concurrency Rules

specify-async-launch-policy Highlight calls to a `std::async` overload that don't have `std::launch` as its first parameter type.

```
void foo(int i)
{
  //
  // Warn:  First parameter does not have
  //        type 'std::launch'
  auto a1 = std::aync ([i] () { doStuff(i); });
}
```

lambda-thread-ctor The `std::thread` constructor uses a template parameter for the function to be executed in the new thread. In the case of passing a lambda expression as the first argument to the constructor, the type deduced will be the closure type of the lambda expression.

```
void foo(int i)
{
  std::thread t([i] () { doStuff(i); });
  // ...
  t.join();
}
```

This approach also works if the lambda expression is created in a separate step by assigning to a local "auto" variable (or passing through a template wrapper):

```
void foo(int i)
{
  auto fn = [i] () { doStuff(i); };
  std::thread t(fn); // type of "fn" is closure type
  // ...
  t.join();
}
```

However, type information is lost when the lambda expression is assigned to a `std::function`, e.g.

```
void foo(int i)
{
  std::function<void()> fn = [i] () { doStuff(i); };
  std::thread t(fn); // type of "fn" is closure type
  // ...
  t.join();
}
```

thread-ctor-lifetime In the case of a lambda function being passed to the `std::thread` constructor, any captures by reference can be highlighted.

```
void foo()
{
  int i = 0;

  //
  // 'i' is captured by reference and may be
  // destroyed before the thread has completed.
  return std::thread ( [&i] () { doStuff(i); });
}
```

If the thread object has automatic storage duration an exception can be made if the `join` is called on that object.

```
void foo()
{
  int i = 0;
  std::thread t( [&i] () { doStuff(i); });

  //
  // OK, lifetime of 't' shorter than 'i'.
  t.join ();
}
```

locks-form-dag The call graph of the entire program shall be walked to ensure that there is no path between two calls to `std::mutex.lock()` on the same mutex object.

```
std::mutex m;
void f1()
{
  std::lock_guard (m);
}
void f2()
{
  std::lock_guard (m);
  //
  // Warn
  f1 ();
}
```

mutex-not-dynamic Highlight *new-expression*s or calls to `std::make_shared` with type `std::mutex`.

```
void f1()
{
  // Warn
  auto a = new std::mutex;

  // Warn
  auto b = std::make_shared<std::mutex> ();
}
```

2.4.8 Heuristics for Implementation of Parallelism Rules

relaxed-atomics Assuming that the memory order will be passed in as a constant, an explicitly specified function call argument other than `std::memory_order_seq_cst` for a parameter of type `std::memory_order` can be highlighted.

noexcept-parallel-algorithm-0 The functor is passed as a template parameter to the parallel algorithm. For a closure type it can be highlighted if the function call operator isn't declared noexcept.

Throwing an exception from a functor is an unhygienic property and makes the function non-pure. As a consequence this is violating a common precondition of most parallel patterns as described in deliverable D2.1.

3. Validation of Patterned Code

3.1 Combinatorial Test Design (CTD)

Combinatorial test design (CTD), a.k.a. combinatorial testing, is an effective test planning technique for coping with the verification challenge of increasingly complex software systems. As the size of test spaces continuously grows, exhaustive verification methods become impractical. Functional testing, on the other hand, is prone to omissions, as it always involves a selection of what to test from a possibly enormous test space. CTD addresses this challenge by a systematic selection of tests, which is based on the observation that in most cases, the appearance of a bug depends on the interaction between a small number of features, or parameters, of the system under test. Experiments show that a test set that covers all possible pairs of parameter values can typically detect 50% to 75% of the bugs in a program [17, 19]. Other experimental work has shown that typically 100% of bugs can be revealed by covering the interaction of between 4 to 6 parameters [13]. In CTD, the test space is manually modelled by a set of parameters, their respective values, and restrictions on the value combinations. The aggregate of parameters, values, and restrictions is called a combinatorial model. A valid test in the test space is an assignment of one value to each parameter without violating restrictions. A subset of the space is automatically constructed so that it covers all valid value combinations of every parameters, where is usually a user input. The most common application of CTD is known as pairwise testing, in which the interaction of every pair of parameters must be covered, but, in general, one can require different levels of interaction for different subsets of parameters. Each test in the result of CTD is a combination of parameter values that represents a high level test, or a test scenario. These high level scenarios need to be translated into concrete tests that can be executed. An example of a required translation is the need to generate actual test data to match the data that may be captured in model parameters. Another example is the need to perform test setup operations to reach a state required by the CTD test.

CTD is very effective for a variety of system types and testing domains [6, 7, 10, 20]. It was recently shown in [21] that the application of CTD to industrial products can result in significant improvement in field quality. Successful application of CTD in practice is, however,challenging. One challenge relates to the manual process of defining combinatorial models and maintaining them, i.e., as the system under test evolves, since it requires both domain knowledge of the system under test and testing expertise. This is further exacerbated when testing for concurrency-related issues, as useful testing requires understanding concurrency and its affects.

3.1.1 CTD for validating Rephrase parallel patterned code

The Rephrase concurrency patterns enhance the ability to successfully and quickly introduce parallelism into an application's code. It is important to also validate the correctness of the application's operation after the Rephrase concurrency related transformations. We have developed both a methodology and tool support, implemented as a new IBM FOCUS CTD feature, to greatly ease this validation task, while allowing for the benefits of test planning and optimisation of Combinatorial Test Design.

3.1.2 IBM FOCUS CTD concurrency patterns validation methodology

The validation goal is to provide confidence in the correctness of the Rephrase concurrent patterns transformations. We assume that the operation of the original sequential code, before undergoing the transformation is the desired behaviour. This behaviour could be functional, or model any desired well defined metric, such as performance or size metrics. The validation methodology can either utilise an existing test suite, or a FOCUS CTD model that is created for the sequential code.

3.1.2.1 Recommendation on how to build a model for the sequential code

A simple model would cover the input space of the relevant code (the sequential code that then was automatically parallelised using one of the Rephrase parallel patterns). As the behaviour is expected to remain unaffected after the transformation, there is no need to model the output. Instead, the sequential code's behaviour is used as an oracle, or a reference point. The simplest implementation would compare the results of running a test on the sequential code with the results of running an enhanced test, as described next, on the parallelised code. A more sophisticated comparison could define a user function with equivalence classes, to allow more flexibility in the comparison.

To model the input space, the modeller should look at the function inputs and their type. Each input should be an attribute or a small set of attributes, with values determined by the input variable type. It is recommended to abstract the input to a few equivalence classes. For example, if a function has an *'int Age'* variable, there could be a FOCUS CTD model attribute named *'Age'* with values *'young', 'middle-aged', 'elderly'*. It may be the case that some input values do not affect the output of the function, and these can be marked as 'don't cares' or 'NA' attribute values.

3.1.2.2 A template-like model for testing the Rephrase parallel patterned code

We make the important observation that to test any concurrent implementation, what matters the most is the execution order of the different threads. Therefore, our testing methodology is to insert, in the parallelised implementation, a wait function before each write operation of a variable that is either global, or defined outside the scope of the parallelised code. For example, if a loop implementation is parallelised, as in the following example, any variable that is defined outside that loop and changed within the loop should have a wait function added before the write operation (see example). The number of waits will depend on the parallelised implementation. However, the parallelism FOCUS CTD model pattern is very simple and remains the same for any pattern.

3.1.2.3 IBM FOCUS CTD parallelism model for any Rephrase parallelism pattern implementation

Each inserted wait, as described above, is a model attribute. The value of each wait attribute could be binary: 1 (wait executed) or 0 (wait skipped), or more elaborate, for example, specifying also various wait duration.

3.1.3 Merging the original code's model or tests with the parallel patterns model

The IBM FOCUS CTD model of the sequential code is merged with the IBM FOCUS CTD parallelism model that contains the correct number of wait attributes per the specific Rephrase parallelism pattern transformation that was applied. IBM FOCUS CTD optimises the combined model. The result is a test plan that covers (n-wise coverage, defined as part of the model) both the important behaviour of the original code, and the different potential orders in which the code may be executed after undergoing the Rephrase transformation.

If a test suite exists, it is still best to create an IBM FOCUS CTD model for it, and proceed as above. However, this may not always be feasible. For example, people familiar with the functionality of that specific piece of code might not be available. In that case, it is possible to use the existing tests, and only optimise the wait model, using IBM FOCUS CTD. For each existing test, we would recommend to implement all the resulting wait options, per the CTD test plan created for the wait model .

3.1.3.1 IBM FOCUS CTD tool support: the 'add model' feature

We implemented assistance for combining models. The new IBM FOCUS CTD 'add model' feature enables the user to work on one model, then select another model, and add all of it (attributes and values, restrictions, and coverage requirements) or only a subset of its attributes at once. Any restrictions of the added model that contain added attributes are automatically added to the combined model. Conflicts are highlighted graphically so the user can resolve them. This way, the user may specify the wait model, add in the behavioural model of the sequential code, and run CTD on the combined model. The resulting optimised test plan will contain both the behavioural attributes values and the wait attributes values to implement tests for.

This approach greatly eases the task of validating the Rephrase transformations. The major effort remains in validating the sequential code—creating a FOCUS CTD model for it. This effort is reused in testing the code after the Rephrase parallel pattern transformation. It is also possible to reuse just the existing test cases, without requiring any additional modelling, as described above.

Figures 3.1,3.2,3.3 that follow provide screen captures of the IBM FOCUS CTD 'add model' feature over the following example.

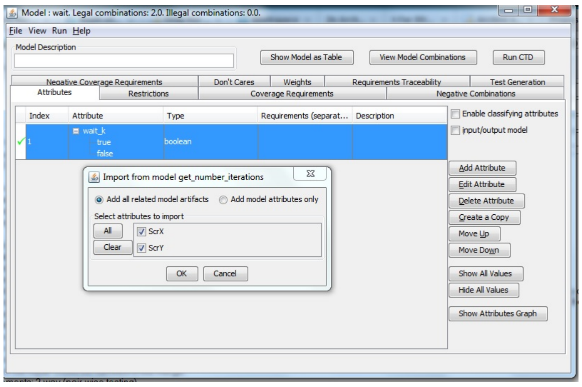
3.1.3.2 An example: The Mandelbrot inner-loop map pattern example with FOCUS CTD new/merge models feature

We use the same example as presented in describing the IBM Explisat verification—the Mandelbrot example, shown in Figure 4.2 (a) and 4.2 (b).

The first IBM FOCUS CTD model would be an input-space model for the original sequential function. Lets call it get_number_iterations model. The IBM FOCUS CTD model attributes and values for the get_number_iterations function shown in Figure 4.2 (a) (the original sequential function) could be:

1. ScrX (attribute)
    (a) ScrX_min (value)
    (b) ScrX_max
    (c) OutOfRangeXMin
    (d) OutOfRangeXMax
2. ScrY (attribute)
    (a) ScrY_min (value)
    (b) ScrY_max
    (c) OutOfRangeYMin
    (d) OutOfRangeYMax

Similarly, we could potentially add model attributes for the other code function parameters. To keep the example small and simple, let us assume that the other parameters do not affect the output, and therefore need not be optimised as part of the test planning process. When generating test cases from the IBM FOCUS CTD test plan, there would be a need to assign values to these parameters as well. Therefore, we leave the following out of the model Attribute: Fract defined as



Model : wait. Legal combinations: 2.0. Illegal combinations: 0.0.

Model Description

	Index	Attribute	Type	Requirements (separat...	Description
✓	1	wait_k true false	boolean		

☐ Enable classifying attributes
☐ input/output model

Import from model get_number_iterations

● Add all related model artifacts ○ Add model attributes only

Select attributes to import

☑ ScrX
☑ ScrY

Figure 3.1: While working on the wait model, select the 'add model' File menu option, select the get_number_iterations model, and import all attributes.

NA, Attribute: Iter_max defined as NA, Attribute: Function defined as NA. Notice that there is an additional parameter to the function—Colors—that is an output parameter, and therefore is not part of the IBM FOCUS CTD model.

Restrictions: none in our case. Could be handled in the merge.

Coverage requirements: 2-way (pair-wise testing)

The second model would be a wait model, for the parallelised code, following the Rephrase pattern transformation. Lets call it the wait model. The IBM FOCUS CTD model attributes and values for the code shown in Figure 4.2 (b) could be:

1. Attribute: Wait_k

2. Values: 1, 0

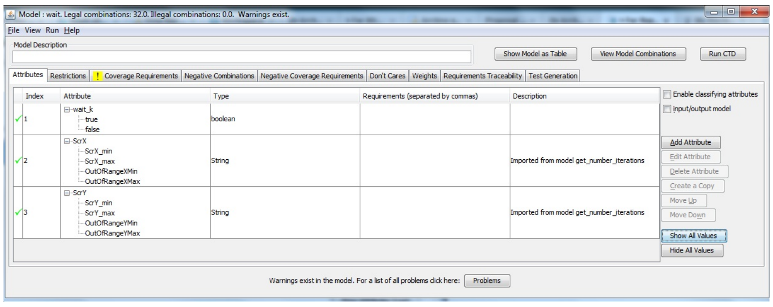
For testing, a wait should be inserted into the code depicted in Figure 4.2 (b) after line 11.

Notice that there are numerous options for implementing an IBM FOCUS CTD model for the sequential code, The user may define any attributes, values, restrictions, and coverage requirements they see fit. We choose a simple approach that is both intuitive and easy to implement for any code function—modelling the code function input space.

As we noted, the tests output is not explicitly specified, as we have the sequential code implementation as our oracle.

The next step is to open the wait model in IBM FOCUS CTD, then utilise the new IBM FOCUS CTD 'add model' feature to merge this model with the add_number_iterations model created for the sequential code. The resulting model will contain all attributes. After running IBM FOCUS CTD test plan optimisation we get the optimised test plan for testing the code shown in Figure 4.2 (b).

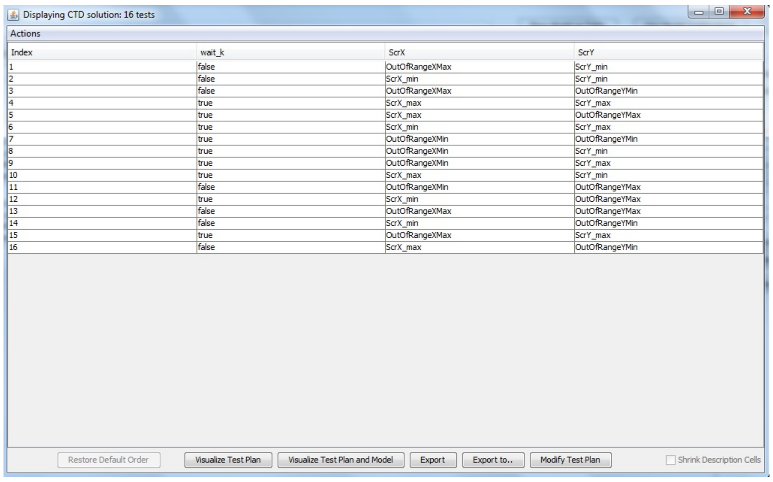
Figures 3.1,3.2,3.3 depict the IBM FOCUS CTD interface for merging the wait model with the get_number_iterations model.



Index	Attribute	Type	Requirements (separated by commas)	Description
✓ 1	wait_k true false	boolean		
✓ 2	ScrX ScrX_min ScrX_max OutOfRangeXMin OutOfRangeXMax	String		Imported from model get_number_iterations
✓ 3	ScrY ScrY_min ScrY_max OutOfRangeYMin OutOfRangeYMax	String		Imported from model get_number_iterations

Warnings exist in the model. For a list of all problems click here:

Figure 3.2: The resulting merged model. Notice the automated warning indication about the difference in coverage requirements.



Index	wait_k	ScrX	ScrY
1	false	OutOfRangeXMax	ScrY_min
2	false	ScrX_min	ScrY_min
3	false	OutOfRangeXMax	OutOfRangeYMin
4	true	ScrX_max	ScrY_max
5	true	ScrX_max	OutOfRangeYMax
6	true	ScrX_min	ScrY_max
7	true	OutOfRangeXMin	OutOfRangeYMin
8	true	OutOfRangeXMin	ScrY_min
9	true	OutOfRangeXMin	ScrY_max
10	true	ScrX_max	ScrY_min
11	false	OutOfRangeXMin	OutOfRangeYMax
12	true	ScrX_min	OutOfRangeYMax
13	false	OutOfRangeXMax	OutOfRangeYMax
14	false	ScrX_min	OutOfRangeYMin
15	true	OutOfRangeXMax	ScrY_max
16	false	ScrX_max	OutOfRangeYMin

Figure 3.3: We resolve the warning of the previous step, run CTD, and get the optimised merged test plan for the code after transformation (the code depicted in Figure 4.2 (b). Notice that the number of combinations to test went down from 32 (title of prev screenshot to 16 (title of this screeshot).

4. Verification of Patterned Code

4.1 Equivalence Check

This section provides a verification method for a code refactored by the patterns from initial pattern set described in D2.1 using the initial refactoring tool described in D2.2. The verification is done by the ExpliSAT tool described in Section 3 of D3.1. The method is explained for the case of introducing a single refactoring into the code. In the case of many patterns the process can be applied iteratively one pattern at a time. The verification method will be explained on Data Parallel Patterns (D2.1 Section 3.3) and demonstrated on the `Map` pattern also known as "`parallel for`" pattern (D2.1 Section 3.3.1).

4.1.1 Preliminaries

Refactoring with pattern replaces component (or task), which is a piece of code, by another piece of code, that will be executed in parallel as part of a parallel pattern (see D2.2 Chapter 4). Therefore, there always can be found a contiguous segment of a code S_o called the original segment, which is replaced by another continuous segment of a code S_p called the replacing segment. One can assume that there is a single entry to the original segment and a single exit from it, otherwise the segment can be enlarged to meet this requirement. The variables carrying the input to the segments can be identified, and they are identical for both segments. The variables, which values are altered by the segment and they are used outside the segment, are the output variables and they as well are identical for the both segments. Of cause there may be variables that are both input and output.

The segments should be semantically equivalent, which means to produce the same output when taking the same input. Testing of such equivalence between two segments is enough to deduce the equivalence between two programs, which differ only on these segments. Thus if many patterns are introduced in a program, the process can be applied iteratively one pattern at a time.

The ExpliSAT tool interprets a program for all possible input values on all possible control paths (see D3.1 Chapter 3.1). The new equivalence verification method accomplishes equivalence checking between S_o and S_p through interpreting them by ExpliSAT one after another in the same piece of code, and then comparing all possible output variables values. First, for combining the segments together, the input and output variables are duplicated to assure no mutual impact. Second, duplicated input variables should be initialised. Section 4.1.4 is dedicated to the question how to choose the initial values for equivalence check. But once the values are chosen, the duplicated input variables should be initialised with exactly the same values. This is obvious for constants, but the values can be *symbols* as well (see D2.1 Section 3.1). One way to inject symbols is to use the pattern $nondet\_ < type >$ (see D2.1 Section 3.3.2). For example, for initialisation of a variable of type `unsigned char`, $nondet\_uchar()$ is used. For injecting **the same symbol** in place of different variables, the same call to $nondet\_uchar()$ should be used for both as demonstrated in the example in Figure 4.1, which is further explained.

In the example of Figure 4.1 the original and replacing segments are presented in sub-figures (a) and (b) respectively. `result` is an input/output variable, and `Max` is input variable for these segments. The equivalence check program is presented in Figure 4.1 (c). The duplicates of `result` (`result1` and `result2`) are initialised with `0` (line 1). Suppose that `Max` can be initialised to `0`, `1`, or `2` only. The variable `MaxSymbol` gets the *symbol* as an input (line 2), fv_assume ensures the restrictions on the symbol (line 3). Then the same symbol is used to initialise `Max1` and `Max2` (line 4), which are the duplicates of `Max`. The program will be further explained in the next section.

4.1.2 The Algorithm

When ExpliSAT interprets a program (see D2.1 Section 3.3.1) whenever it faces a branch and both sides of the 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 path condition. Thus, when the computation terminates, the interpreter will pop from the queue the state that was saved when reaching the branch, and will continue the symbolic run.

For equivalence check ExpliSAT works in a special mode. It gets a specially structured piece of code P containing the both segments S_o and S_p. The new mode introduces a new keyword $fv\_end\_equivalence\_segment$. This keyword is inserted to P at the end of each segment and indicates the end of the segment for ExpliSAT. An example for P and its structure can be seen in Figure 4.1 (c): the initialisations (lines 1 to 4) were explained in Section 4.1.1; first segment (lines 5 and 6); call to $fv\_end\_equivalence\_segment$ (line 7); second segment (lines 8 to 13); call to $fv\_end\_equivalence\_segment$ (line 14); a comparison part (line 15), where for each original output variable u an equality of the values for u's duplicates is tested using fv_assert (see D2.1 Section 3.2.2).

Notice that S_o, S_p and their input and output variables may be identified already during the refactoring process and may be automatically obtained from the refactoring tool. After the input variables values are found, P may be created automatically as well and then ExpliSAT will be invoked on P in equivalence check mode. Therefore, if finding input variables values is automatic, the entire equiva-

(a)

```
unsigned char result=0;
for(int i=1;i<=Max;i++)
    result=result+i;
```

(b)

```
unsigned char result=0;
unsigned char j=1;
for(int i=1;i<Max;i++){
    result=result+i;
    j++;
}
result=result+j;
```

(c)

```
unsigned char result1=0, result2=0;
unsigned char MaxSymbol = nondet_uchar();
fv_assume(MaxSymbol<=2);
unsigned int Max1=MaxSymbol, Max2=MaxSymbol;
for(int i=1;i<=Max1;i++)
    result1=result1+i;
fv_end_equivalence_segment();
unsigned char j=1;
for(int i=1;i<Max2;i++){
    result2=result2+i;
    j++;
}
result2=result2+j;
fv_end_equivalence_segment();
fv_assert(result1==result2);
```

Figure 4.1: Equivalence check example

lence test process may be executed fully automatically.

Now it will be shown how ExpliSAT accomplishes the equivalence test. Thanks to the P's structure, any control path of P will pass through both

$fv\_end\_equivalence\_segment$ calls. This enables the special equivalence check mode for ExpliSAT. This mode has three stages: interpreting of the first segment, interpreting of the second segment, and comparing of the output result variables values. In the first stage ExpliSAT runs as usual, but considers $fv\_end\_equivalence\_segment$ as path termination. Consequently, when $fv\_end\_equivalence\_segment$ is reached, the next state is popped from the queue. When the queue is empty, ExpliSAT interprets the second segment in exactly the same way. When the queue is empty again, ExpliSAT interprets the comparison part.

In addition during each of the first two stages ExpliSAT builds for each output variable v of the segment the conditional expression expressing its value. It is not to be forgotten that v is a duplicate of some original variable. The expression has the following form $c_1 ? v_{1\_true} : (c_2 ? v_{2\_true} \ldots (c_n ? v_{n\_true} : illegal\_value) \ldots)$, where c_i is the path condition of the path terminated by $fv\_end\_equivalence\_segment$ and v_{i_true} is the value of v at the end of this path. Notice, that v_{i_true} is not necessarily a constant, but may be a symbolic expression. ExpliSAT creates the expression for v as follows: when the path with path condition c_i reaches $fv\_end\_equivalence\_segment$, "$c_i ? v_{i\_true}$: (" is appended to the expression. When the queue is empty, this means that there are no more path conditions that can be satisfied, therefore the expression can be closed. But the last appended conditional expression $c_n ? v_{n\_true}$: contains only a true part. Since the false condition cannot be satisfied, an expression symbolising illegal value is used as a false part in the expression. Therefore the last appended conditional expression has the following form: $c_n ? v_{n\_true} : illegal\_value$. Now the brackets for the whole expression can be closed. The example of the variable value expression is explained at the end in this Section.

Path condition defines the space of input values that would result in this particular path being executed. After the first two stages each output variable duplicate holds an expression containing all possible values that it can get in specific space of input values. Using of the same symbols and constants for the input variable duplicates forces the inputs to be identical. If the segments are equal, for each set of input values they produce an identical output. Let us see why in the case of equivalence the result of comparison for any output variable will be true. Suppose, it is false for some output variable v_{out}. We denote by $E_{v_{out\_1}}$ and $E_{v_{out\_2}}$ expressions for two duplicates of v_{out}. According to the form of the conditional expression, there has to be at least one set P of input values, such that P satisfies some path condition c_i in $E_{v_{out\_1}}$ and some path condition c_j in $E_{v_{out\_2}}$, and the values v_{i_true} and v_{j_true} are different. This contradicts the assumption that the segments are equivalent. On the other hand, if the segments are not equivalent, there is at least one set S of input values that causes at least for one output variable v_{out} difference in values produced by the segments. Suppose the comparison for the duplicates of this variable returns true. This means that for each set of variables under the satisfied path condition the values of duplicates of v_{out} are equal. But this contradicts the assumption for S.

Now the algorithm is demonstrated on the example in Figure 4.1 (c). $MaxSymbol$ can get three possible values, which induces three path conditions. In this example the conditions are identical for both segments and can be expressed in C style by: $sym == 0$, $sym == 1$, and $sym == 2$. Thus the value of `result1` after the first stage will be: $sym == 0 ? 0 : (sym == 1 ? 1 : (sym == 2 ? 3 : illegal\_value))$. The value of `result2` after the second stage will be: $sym == 0 ? 1 : (sym == 1 ? 1 : (sym == 2 ? 3 : illegal\_value))$. Comparison of `result1` and `result2` will return false, since the outputs are different when `MaxSymbol` is `0`.

4.1.3 Extension to the parallel programs

Any of the segments designated for equivalence check can be parallel. The algorithm should be extended to deal with parallel programs. From inputs and outputs point of view parallelism makes no difference. The distinction is that on the same input the code may be executed differently depending on threads interleavings and then to produce a different output. Consequently for each path condition there may be a set of possible values. Therefore ExpliSAT manages a conditional expression of a new type, where for each condition a set of all possible values is added. Thus the previous expression is the private case of the new expression, where the set sizes are always 1. The comparison of new expressions will return true if and only if for any input the sets are identical.

4.1.4 Initialisation

Choosing of correct values for the input variables is crucial for equivalence checking. ExpliSAT can simulate all possible values using symbols. But if the equivalence test fails, it is not clear whether the failure was caused by the input values, which are feasible in the original program. The best practice is to obtain the space of input values from the code analysis. The values also can be obtained by running ExpliSAT on the original program with one change: every time when the entrance to S_o is reached, for each input variable its value is stored. Since ExpliSAT interprets a program for all possible input values on all possible control paths, at the end of the run, for each input variable v all stored values for v are all possible values. Therefore the disjunction of these values may be used as the initial value of v for the equivalence check.

4.2 The Mandelbrot example

We demonstrate our method on the code of the Mandelbrot project [1] is used. The pattern `Map` is applied inside the function `get_number_iterations`. Figure 4.2 (a) shows the code before the refactoring and Figure 4.2 (b) shows the code after the refactoring. The segment in the example may be naturally chosen as body of the `get_number_iterations`. From the analysis of the code the input variables to the segments are: `scr`, `fract`, `iter_max`, and `func`; and the output variable is the vector `colors`. All input variables values are constants in Mandelbrot example, which leads to the same single path condition in the both segments. Since the segment (a) is not parallel, the set V_o of values for `colors` under this path condition contains a single constant. Segment (b) is parallel, as it comprises $parallel\_for$ implemented by Intel Threading Building Block library. Therefore the set V_p of values for `colors` under the same path condition may contain numerous values. In this case the duplicates of colors will not be equal. Indeed, the equivalence test fails for this example on $fv\_assert(colors1 == colors2)$, where `colors1` and `colors2` are duplicates of `colors`. For more refined in-

(a)

```
1  void get_number_iterations (window<int> &scr, window<double> &fract, int
2  iter_max, std::vector<int> &colors, const std::function<std::complex<double>
3  (std::complex<double>, std::complex<double>)> &func) {
4      int k = 0;
5      for(int i = scr.y_min(); i < scr.y_max(); ++i) {
6          for(int j = scr.x_min(); j < scr.x_max(); ++j) {
7              std::complex<double> c((double)j, (double)i);
8              c = scale(scr, fract, c);
9              colors[k] = escape(c, iter_max, func);
10             k++;
11         }
12     }
13 }
```

(b)

```
1  void get_number_iterations (window<int> &scr, window<double> &fract, int
2  iter_max, std::vector<int> &colors, const std::function<std::complex<double>
3  (std::complex<double>, std::complex<double>)> &func) {
4      int k = 0;
5      for(int i = scr.y_min(); i < scr.y_max(); ++i) {
6          tbb::parallel_for (tbb::blocked_range<int>(scr.x_min(), scr.x_max()),
7          [&](const tbb::blocked_range<int>& r){
8              for (int j = r.begin(); j != r.end(); ++j){
9                  std::complex<double> c((double)j, (double)i);
10                 c = scale(scr, fract, c);
11                 colors[k] = escape(c, iter_max, func);
12                 k++;
13             }
14         });
15     }
16 }
```

Figure 4.2: `get_number_iterations` before and after the refactoring

formation the segment can be reduced to the second `for`: lines 6 to 11 in Figure 4.2 (a) and lines 6 to 14 in Figure 4.2 (b). For the reduced segments the input value `i` and input/output variable `k` should be added. Even if the equivalent test will be executed with inputs from the first outer iteration, which means `i=0,k=0`, $fv\_assert(k1 == k2)$ fails (here `k1`, `k2` are duplicates of `k`). Now the reason is clear. While the value of `k` at the end of the segment (a) is unambiguous, in the segment (b) different threads update `k`, and its value is unpredictable.

4.3 Verification of parallel programs by ExpliSAT

In order to support parallel patterns verification, ExpliSAT was obliged to interpret parallel code. This section presents the changes in ExpliSAT to deal with this kind of code.

We added support for commonly used POSIX library functions. The multithreading is simulated by ExpliSAT during the interpretation of a multithreaded program. Consequently, threads and their context switch information become part of the saved states (see D2.1 Section 3.3.1). ExpliSAT analyses the code and finds control points where different interleavings may lead to different output. At these points ExpliSAT adds non-determinism to the process of thread choosing. The list of unchosen threads in a specific control point is included in the saved state, giving an opportunity to choose another thread, when the state will be popped.

5. Improving detection of lock-free data structure data races via semantics

5.1 Introduction

Nowadays, parallel programming is a de-facto standard to write efficient software on today's machines based on the multi-/many-core technology. Despite the diffusion of parallel programming frameworks offering parallel patterns at different abstraction levels, write efficient and correct parallel code still poses challenging issues, as concurrency bugs, especially data races, have become more frequent if the pattern semantics is not well understood by the high-level programmer.

The adversity in finding data races is a well-known problem []. It has been recognised as an arduous task, given that errors may occur only during low-probability sequences of events and may also depend by external factors such as the current machine load. These facts make data races extremely sensitive in terms of time, the presence of print statements, compiler options, or differences in memory models. The benefits of race detectors are formidable in this sense, nevertheless they may emit false positives and are not able to detect semantics misuses.

In particular, race detectors may generate too many false positives in case of programs that use non-blocking lock-free data structures, where no high-level atomic instructions are used. This makes the debugging process even harder when tracing back the main cause of the problem. Furthermore, current race detectors are not able to detect wrong uses of lock-free data structures, thus violating their semantics and possibly generating undefined results.

The use of lock-free concurrent data structures has been adopted as an effective technique to write highly scalable implementations of fine-grained parallel problems. Examples of such data structures are shared queues with FIFO semantics, linked lists, stacks and many others. They can be used directly by the programmer to write the parallel application, or by the run-time system of a parallel programming framework to implement efficiently high-level parallel patterns. This second case has been adopted in the run-time systems of FastFlow and Intel TBB, and in efficient libraries like Boost.

In this chapter we present our work aimed at extending the semantics of a well-known race detector tool in order to successfully detect data races in parallel programs making use of lock-free data structures. Without loss of generality, we exemplify our approach in the analysis of lock-free concurrent queues used according to the producer-consumer concurrency paradigm. We choose this type of concurrent objects because they represent a key part of the run-time system of the FastFlow parallel framework. However, the same methodology can be systematically applied to solve data race detection for other lock-free data structures.

5.2 The FastFlow parallel programming framework

In this section, we give an overview of the two main software component that have been used to carry out the contribution of this chapter. Apart from, the TSan failure detection tool for identification of undefined and suspicious behaviour of threads, we have leveraged FastFlow parallel programming framework. In this section we review this framework and put particular emphasis on the lock-free structures used underneath.

The FastFlow parallel programming model is a framework that provides a series of defined, general purpose, customisable and modular parallel patterns conceived as algorithmic skeletons [3]. It pursues the use of high-level, platform-independent structured parallel programming patterns and gives support to develop portable and efficient applications for shared-memory (multi-/many-core, hybrid, GPGPU, FPGA, etc.) and distributed platforms. FastFlow's architecture has been designed using a building blocks approach: each layer implements structures built using simpler ones from lower layers. Specifically, its patterns are organised in three main layers: *i)* high level patterns, implementing parallel loops, streams, data-parallel algorithms, workflows of tasks, etc.; *ii)* core patterns, implementing pipelines, farms and feedback structures that allow processes/threads exchanging data among them; and *iii)* building blocks, providing queues, process/thread containers and threads/processes intermediators.

One of the most characterising aspects of the FastFlow run-time system is that it is by default completely lock-free[1]. All the parallel patterns are implemented as acylic/cyclic networks of threads cooperating by exchanging memory pointers to shared data structures through lock-free Single-Producer-Single-Consumer (SPSC) queues which represent the unique case of concurrent data structures present in the runtime. All the patterns (e.g., farm, pipeline, map, divide & conquer, parallel-for) have been written according to this non-blocking lock-free model. For the previous reasons, in this work we have focused on SPSC queues and their race detection analysis, owing to their importance for the FastFlow runtime design.

FastFlow lock-free queues are influenced by FastForward queue [9] and Lamport's wait-free protocols, thus having the ability to build up streaming networks

[1]in alternative, a blocking behaviour can be used in applications that generate long periods of inactivity, e.g., to prevent the CPU from constantly polling, and thus, saving energy.

whose implementation is guaranteed to be correct and efficient through lock-free Single-Producer-Single-Consumer (SPSC) bounded and unbounded queues storing memory references [2].

Different combinations of these SPSC queues can generate more complex streaming networks, e.g., N-to-1, 1-to-M, and N-to-M channels. Although some of these structures require synchronisation mechanisms to keep their correctness, FastFlow implements them using helper threads that serialise communications between producers and consumers and avoids the use of expensive synchronisation primitives. This results again in wait-free, non-blocking structures that allow to reduce cache coherence overheads.

5.3 The Lock-free Single-Producer/Single-Consumer Queue

In this section we describe formally the Single-Producer/Single-Consumer (SPSC) bounded queue and its semantics for the concurrent lock-free version. This allows us to proceed further with our rationale in order to develop semantics and rules for the proper use among entities of the lock-free parallel version.

5.3.1 Formal definition

Consider a SPSC bounded queue $\mathcal{Q}$ the tuple

$$\mathcal{Q}(buf, pread, pwrite, M),$$

where buf is a circular buffer, $pread$ plus $pwrite$ are internal read and write pointers for the buffer, respectively, and M is a set comprising the following methods:

- `init`: Initialises the buffer buf, allocating space of possibly aligned memory and resetting the internal ($pread$ and $pwrite$) pointers by placing them at the beginning of buf. If buf has already been allocated, the method does nothing.
- `reset`: Resets and places the read and write pointers to the beginning of the buffer buf.
- `push`: Enqueues the item into the buffer buf.
- `available`: Returns true if there is at least one room in the buffer buf.
- `pop`: Removes and returns the first item on the buffer buf.
- `empty`: Returns true if the buffer buf is empty.
- `top`: Returns the first item on the buffer buf.
- `buffersize`: Returns the size of the internal queue buffer buf.
- `length`: Returns the number of items currently held by buffer buf.

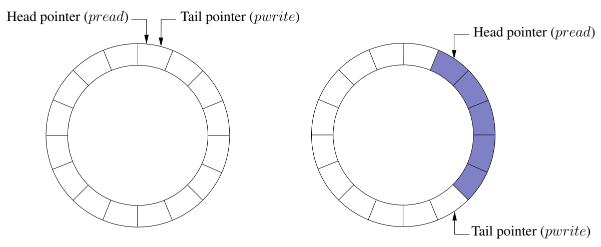


Figure 5.1: Circular buffer of the Single-Producer/Single-Consumer bounded queue.

Figure 5.1 depicts the internal working of buffer buf from the First-In-First-Out (FIFO) SPSC queue. Initially $pread$ and $pwrite$ point to the initial position of the buffer, while afterwards some elements have been added at the terminal position through `push` calls and others removed from the head position by means of `pop` calls.

5.3.2 Semantics of the concurrent lock-free SPSC queue

The implementation of the concurrent SPSC queue for shared cache multi-core systems requires coordination between producer and consumer to ensure proper operation. While different kinds of shared data structures provide different fairness levels requiring diverse synchronisation primitives, Lamport in [14, 15] proposes a wait-free Single-Producer/Single-Consumer queue implementation using a circular buffer without requiring any explicit synchronisation mechanism. Although Lamport considers a Sequential Consistency memory model, a slightly modified version of this approach is still valid under Total-Store-Order (TSO) and weaker consistency memory models [2]. The importance of these algorithms is that they meet with both wait-free and lock-free qualities, neglecting the use of blocking constructs, and thus, improving their performance. For our specific case, we leverage the SPSC lock-free bounded queue implementation from the FastFlow programming framework [3], however the methodology presented can be applied to any other implementation featuring this specific data structure.

Nevertheless, the correctness of this specific algorithm is only ensured if several conditions are met. We define these requirements as the following semantics rules:

1. *Fixed roles*. A lock-free concurrent SPSC queue instance can be shared by three different entities if one of them acts as the constructor, the other as the producer and the latter as the consumer; each of them calling only to methods allotted to their specific role. In certain cases, the producer or the consumer can perform the role of the constructor, being only two different entities sharing the same queue. If none of these cases are given, we consider that the queue is misused, thus having an undefined behavior due to potential

data races.

2. *Initialisation methods*. The constructor can call to methods belonging to:

$$Init = \{\texttt{init}, \texttt{reset}\} : \; Init \subset M.$$

3. *Producer methods*. The producer thread should only invoke a subset of methods in M,

$$Prod = \{\texttt{push}, \texttt{available}\} : \; Prod \subset M.$$

4. *Consumer methods*. The consumer thread should only invoke a subset of methods in M,

$$Cons = \{\texttt{pop}, \texttt{empty}, \texttt{top}\} : \; Cons \subset M.$$

5. *Common methods*. There are methods that can be invoked by both producer and consumer:

$$Comm = \{\texttt{buffersize}, \texttt{length}\} : \; Comm \subset M.$$

Particularly, all subsets allotted to different roles of the queue fulfill $M = Init \cup Prod \cup Cons \cup Comm$. Note also that, methods internally using the $pwrite$ pointer are those assigned to the producer, while those using the $pread$ pointer are related to the consumer. Methods using none of these pointers or only static parameters, such as the buffer size, can be called by both producer and consumer.

A formalisation of the aforementioned semantics is possible if a set C of entity IDs is added as an attribute to each of the preceding subsets of methods in the queue $\mathcal{Q}$. By inserting the ID of the calling entity (or thread) to the corresponding set C of the subsets $Init$, $Prod$ or $Cons$, each time a method belonging to it is invoked, it is possible to control the proper use of the lock-free SPSC queue checking two simple requirements. The first ensures that the cardinality of the C set of the initialisation, producer and consumer subsets should always be less or equal than one,

$$|Init.C| \leq 1 \;\wedge\; |Prod.C| \leq 1 \;\wedge\; |Cons.C| \leq 1, \tag{5.1}$$

hence, only one and the same entity should use methods allotted to its role. The second guarantees that both producer and consumer are performing the right role, i.e.,

$$Prod.C \;\cap\; Cons.C = \emptyset. \tag{5.2}$$

Note also that all these requirements are valid for the concurrent SPSC queue; in other words, if $|Prod.C \;\cup\; Cons.C| > 1$ is met, as stated in semantic rule (1).

Listing 5.1 illustrates an execution sequence of a correct use of the lock-free SPSC queue. In this example, 3 threads use the same queue, each of them calling only to its assigned methods, therefore meeting requirements (5.1) and (5.2).

Listing 5.1: Example of execution sequence of a correct use of the SPSC queue.

```
Thread 1 call to init      C={1}
Thread 1 call to reset     C={1}
Thread 2 call to empty     C={2}
Thread 2 call to pop       C={2}
Thread 3 call to available C={3}
Thread 3 call to push      C={3}
```

Listing 5.2 shows a misuse of the SPSC queue: it violates requirements (5.1) and (5.2). The cardinality of set C of methods in $Prod$ and $Cons$ ends up equating to 2 and producer thread with ID 2 calls to methods allotted to the consumer, e.g., $\texttt{push}.C \cap \texttt{pop}.C \neq \emptyset$.

Listing 5.2: Example of execution sequence of a misuse of the SPSC queue.

```
Thread 1 call to init      C={1}
Thread 1 call to reset     C={1}
Thread 2 call to available C={2}
Thread 2 call to push      C={2}
Thread 3 call to available C={2,3} (Violated req. 1)
Thread 3 call to push      C={2,3} (Violated req. 1)
Thread 4 call to empty     C={4}
Thread 4 call to pop       C={4}
Thread 2 call to empty     C={4,2} (Violated req. 1,2)
Thread 2 call to pop       C={4,2} (Violated req. 1,2)
```

5.4 Implementation

In this section we describe implementation details considered to integrate semantics of the Single-Producer/Single-Consumer parallel data structure into the race detection tool TSan.

To illustrate the workings of TSan, we leverage both execution sequences shown in Listings 5.1 and 5.2. Assuming a machine supporting the TSO memory model, both examples would have reported data races in instructions that read and write simultaneously from the same memory address. Listing 5.4 shows a data race report generated by TSan when executing `push` and `empty` FastFlow functions presented in Listing 5.3. However, this is only true to some extent. While data races reported for the execution sequence in Listing 5.2 would have been real due to misuse of the concurrent SPSC queue, those reported in the execution sequence in Listing 5.1 would have resulted in false positives, as semantics of the concurrent SPSC queue are accomplished at any time. Being the lock-free SPSC queue a thread-safe concurrent data structure, as defined in Section 5.3, no data races should occur. Therefore, our aim is to endow the race detector with the semantics of this specific data structure in order to filter those "benign" data races and reduce the amount of warnings generated.

Listing 5.3: FastFlow's SPSC bounded lock-free queue producer and consumer implementation.

```
/************ PRODUCER FUNCTIONS *************/
bool available() {  return  (buf[pwrite] == NULL)
   (cont.);  }

bool push(void* data) {
   if  ( !data ) return false;
   if  ( available() ) {
      WMB(); // write-memory-barrier
      buf[pwrite] = data;
      pwrite += (pwrite+1 >= size) ? (1 - size)
         (cont.): 1;
      return true;
   }  return false;
}
```

```
/************ CONSUMER FUNCTIONS *************/
void* top() const {  return buf[pread];  }

bool empty() {  return  (buf[pread] == NULL);  }

bool pop(void** data) {
   if  ( !data || empty() ) return false;
   *data = buf[pread];
   buf[pread] = NULL;
   pread += (pread+1 >= size) ? (1-size): 1;
   return true;
}
```

Listing 5.4: Example of `empty-push` SPSC data race report from TSan.

```
==================
WARNING: ThreadSanitizer: data race (pid=5181)
Read of size 8 at 0x7d5c0000fc48 by thread T1:
    #0 ff::SWSR_Ptr_Buffer::empty() fastflow-2.0.4/ff/buffer.hpp:186:17 (testSPSC+0x0000004f965a)
    #1 ff::SWSR_Ptr_Buffer::pop(void**) fastflow-2.0.4/ff/buffer.hpp:325 (testSPSC+0x0000004f965a)
    #2 consumer(void*) fastflow-2.0.4/tests/testSPSC.cpp:74:19 (testSPSC+0x0000004f933a)

Previous write of size 8 at 0x7d5c0000fc48 by thread T2:
    #0 ff::SWSR_Ptr_Buffer::push(void*) fastflow-2.0.4/ff/buffer.hpp:239:25 (testSPSC+0
       (cont.)x0000004f9527)
    #1 producer(void*) fastflow-2.0.4/tests/testSPSC.cpp:54:19 (testSPSC+0x0000004f91ba)

Location is heap block of size 800 at 0x7d5c0000fc00 allocated by main thread:
    #0 posix_memalign llvm/projects/compiler-rt/lib/tsan/rtl/tsan_interceptors.cc:758 (testSPSC+0
       (cont.)x000000423967)
    #1 getAlignedMemory(unsigned long, unsigned long) fastflow-2.0.4/ff/sysdep.h:200:9 (testSPSC+0
       (cont.)x0000004f9807)
    #2 ff::SWSR_Ptr_Buffer::init(bool) fastflow-2.0.4/ff/buffer.hpp:170 (testSPSC+0x0000004f9807)
    #3 main fastflow-2.0.4/tests/testSPSC.cpp:92:6 (testSPSC+0x0000004f939c)

Thread T1 (tid=5183, running) created by main thread at:
    #0 pthread_create llvm/projects/compiler-rt/lib/tsan/rtl/tsan_interceptors.cc:849 (testSPSC+0
       (cont.)x000000423ddf)
    #1 main fastflow-2.0.4/tests/testSPSC.cpp:95:11 (testSPSC+0x0000004f93b5)

Thread T2 (tid=5184, finished) created by main thread at:
    #0 pthread_create llvm/projects/compiler-rt/lib/tsan/rtl/tsan_interceptors.cc:849 (testSPSC+0
       (cont.)x000000423ddf)
    #1 main fastflow-2.0.4/tests/testSPSC.cpp:96:11 (testSPSC+0x0000004f93cc)

SUMMARY: ThreadSanitizer: data race fastflow-2.0.4/ff/buffer.hpp:186:17 in ff::SWSR_Ptr_Buffer::
   (cont.)empty()
==================
```

5.4.1 Supporting semantics of the SPSC queue into TSan

There are several approaches in order to embed SPSC semantics into TSan's runtime. A naive but wrong approach to filter "benign" data races is to use the predefined `no_sanitize_thread` TSan's attribute in order to avoid instrumentation of routines that have been avowed to generate false data races. This, however, misses real data races given from improper uses of the concurrent SPSC queue. Alternatively, our approach to implement semantics of the concurrent SPSC queue requires the following modifications in TSan's runtime internals:

- Given that a multithreaded application can use multiple lock-free SPSC queues simultaneously, with possibly each thread performing different roles in diverse queue instances, it is necessary to univocally identify those in reports obtained by TSan. This can be solved by printing out the C++ implicit `this` pointer associated to the SPSC queue instance involved in the data race report. To do so, it is necessary to design a mechanism to get this pointer from TSan runtime. This is, though, not easy to implement, since the TSan's thread responsible for detecting data races is not the same to that incurring the data race, possibly in a different context. There exist several approaches to implement this approach using communication mechanisms between threads, such as sockets or shared memory. However, all of them require modifications to the user code, something not desirable in our case.

  Our approach follows a different path. We leverage the `libunwind` library [16] to read the stack and base pointers (`sp` and `bp`, respectively) so as to walk backwards on the stack until the parameter is found in the appropriate stack frame.[2] Note that if there are inlined functions within the calling stack, it is not possible to retrieve the desired frame by walking a fixed number of steps back in the stack. Thus, our approach requires Clang's `noinline` attribute on inlined user functions and the `-O0` flag to suppress automatic inlining at compile time.

- Afterwards, we implement the semantic rules defined in Section 5.3. To support this functionality, we leverage a C++ STL's `map` container that stores `this` pointers from SPSC queue objects along with `set` structures for each member function of the queue, as key/values pairs. Next, this structure is used to check the semantic requirements of the lock-free SPSC bounded queue, in order to determine whether a data race within a function member of this queue class is benign or not. Finally, we print out only those reports related to real data races.

While in this work we only included the semantics of the SPSC queue, the current implementation within TSan internals can be easily extended to other lock-free

[2]To the best of our knowledge and throughout the experimentation, an object's `this` pointer can be found at position $\texttt{bp} - 1$ in the class member function's stack frame of a Clang-compiled C++ program running on a x86_64 architecture.

data structures where “benign” data races can occur. We believe this can tremendously aid developers to focus on debugging real data races.

5.5 Experimental results

After the implementation of the semantic requirements of the SPSC bounded lock-free queue into ThreadSanitizer, we evaluate the detection of data races using a set of μ-benchmarks and real parallel applications from the FastFlow parallel programming framework. In the following we describe the target platform, software and benchmarks sets adopted during our evaluation process.

- ***Target platform***. The analysis and the evaluation has been carried out on a server platform comprised of 2$\times$ Intel Xeon Ivy Bridge E5-2695 v2 with a total of 24 cores running at 2.40 GHz, 30 MB of L3 cache and 128 GB of DDR3 RAM, while running Linux Ubuntu 14.04.2 LTS OS on a 3.13.0-57 Linux kernel. We denote this platform as IVY.

- ***Software***. We leveraged FastFlow v2.0.4 as for the parallel programming framework to test the implementation of the SPSC bounded lock-free queue. As mentioned, for the compilation and race detection process we used the LLVM compiler infrastructure v3.7.0 together with the compiler runtime libraries (`compiler-rt`) including the race detector ThreadSanitizer. Furthermore, we installed the respective LLVM’s `libc++` and `libc++` ABI in order to support the C++11 standard library for the FastFlow examples.

Featuring the aforementioned hardware and software, we perform our evaluation using the following benchmarks as part of the FastFlow framework. While the first item in the list refers only to the tests included in our μ-benchmarks set, the subsequent application examples are part of the applications set.

- *μ-benchmarks*: They primarily focus to test internal structures and performance of specific FastFlow features. These tests are written in tutorial style and can be effectively used as FastFlow μ-benchmarks. With this, we aim at testing internal workings of FastFlow, therefore testing all possible ways in which a SPSC is used in FastFlow’s core. We run this applications with the default parameters.

- *Cholesky factorization*: The Cholesky factorisation of a symmetric positive definite matrix $A \in \mathbb{R}^{n \times n}$ is given by $A = U^T U$, where the Cholesky factor $U \in \mathbb{R}^{n \times n}$ is upper triangular. Optimisation of this algorithm are possible using block-partitioning in conjunction with highly-tuned (BLAS-2 and BLAS-3) kernels, since they reduce the latency associated to the data movement between memory and LLC. During our evaluation, we run both the classic `cholesky` and blocked `choleskyBlock` Cholesky factorisation on a matrix consisting of 20,480$\times$20,480 elements with 40 streams, i.e, a block size of 512$\times$512.

- *Fibonacci*: The Fibonacci numbers is sequence of starting with one or zero and followed by a one, where each subsequent number is the sum of the two previous values, as stated in [11]. Within the FastFlow examples a version of this problem using skeletal programming on a simple stream parallel approach is implemented under `ff_fib`. We set to 100 the length of the Fibonacci series and to 20 the number of streams.

- *Matmul*: This set of applications implement the matrix-matrix multiplication make use of FastFlow parallel pattern structures. The first two tests, `ff_matmul` and `ff_matmmul_v2` use *farms* of tasks each of them computing an element of the output matrix. While the third, `ff_matmul_map`, leverages the *map* construct. All these tests were run using 24 worker threads on matrix sizes of 512×512.

- *Quicksort*: The parallel version of the sorting algorithm Quicksort has also been included in the FastFlow examples. In this case we only run the version of the algorithm based on the farm pattern (`ff_qs`) over an array of 10,000 entries and a threshold of 10.

- *Jacobi*: This algorithm provides the solution of the indefinite Helmholtz equation that is a two-step generalisation of the classic Jacobi iteration using complex parameters. It solves the Poisson equation on rectangular grid assuming uniform discretisation in each direction and Dirichlet boundary conditions. The FastFlow versions of this application leverage parallel for/reduce and stencil patterns in the `jacobi` and `jacobi_stencil` applications examples, respectively. We set the grid dimensions in both x and y directions to 5,000, the Helmholtz constant to 0.8, the error tolerance to 1.0 and the maximum number of iterations to 1,000.

- *Mandelbrot*: This test is a simple parallel version of the application that computes and displays the Mandelbrot set leveraging FastFlow. Indeed, it is an example of an embarrassingly parallel application. The Mandelbrot set is represented as a matrix of pixels, whose coordinates are orchestrated by a scheduler that dispatches streams to worker threads in a round robin fashion. We run the implementation using FastFlow with and without memory allocator (`mandel_ff` and `mandel_ff_mem_all`, respectively) with a resolution of 640 k-pixel and 1024 maximum number of iterations.

- *n-queens*: The n-queens problem is a generalisation of the well-known 8-queens problem supporting a $n \times n$ board-size chess with the purpose of counting all possible solutions. There exists multiple implementations of the n-queens problem using recursive divide and conquer and iterative approaches. We use the fast iterative but parallel FastFlow's implementation adapted from the sequential code in [18]. During the evaluation, we executed the both `nq_ff` and `nq_ff_acc` versions computing a board of 21×21 positions.

Table 5.1: Number of SPSC data races caused by pairs of functions for the μ-benchmarks and applications sets.

Benchmark set	SPSC data races		
	`push-empty`	`push-pop`	SPSC-other
μ-benchmarks	177	2	4
Applications	60	0	0

Note that all the aforementioned benchmarks were compiled using Clang compiler with the following additional flags `-std=c++11`, `-stdlib=libc++`, `-fsanitize=thread`, `-lc++abi` and `-O0`. Afterwards, we executed them using a fixed pool of 24 worker threads, i.e., to fully populate the cores of IVY. Throughout the execution of these examples, we aim at internally enforcing the use of multiple SPSC queue instances in multiple ways, within different FastFlow parallel constructs. Simultaneously, we collect data races reports generated by TSan for further analysis.

5.5.1 Analysis of individual SPSC data races

Our first experiment analyses the frequency of occurrence of SPSC queue-related data races detected by TSan for both μ-benchmarks and application sets. Table 5.1 combines frequency of the most common function pairs responsible for the data races occurred in both benchmarks tested. As seen, for the μ-benchmark set, a big portion of the data races were caused by the pair of functions `push-empty`. Referring at the code in Listing 5.3, this data race arose when the producer thread was writing data to the buffer at position $pwrite$ (line 8) of the `push` function while the consumer thread was reading from the same position (pointed by $pread$) in `empty` function (line 16). Actually, this was the only type of data race detected for the application set, but not the only for the μ-benchmark set. In this case, we also detected two other pairs of data races. One of them is the tuple `push-pop`, occurring at lines 8 and 20 of the code in Listing 5.3, also when both threads were writing and reading from the same buffer entry.

Finally, there exists four more occurrences of data races for the μ-benchmarks ("SPSC-other" column) in which only one side involved in the data race was related to a member function of the SPSC queue FastFlow's class. Analysing data race reports we observed that, at some point, a thread executing `posix_memalign` or `malloc` POSIX functions, caused in total 4 possible races when another thread was running `inc`, `pop` or `empty`. This effect was probably caused during the SPSC queue buffer's allocation, nevertheless we cannot ensure at this point whether they could be benign or not. A deep analysis of this section of the code is ongoing.

5.5.2 Analysis of global and SPSC data races

In this experiment we analyse statistically data races occurred during the execution of both μ-benchmark and application sets with special focus on those related to the SPSC queue.

Figure 5.2 shows (percentage-wise) the portion of SPSC queue-related data races with respect to the others for both μ-benchmark and application sets. Note that we consider part of the SPSC races those in which only one side was related to a function member of the SPSC queue class. As observed for the μ-benchmark set, roughly 47 % of the data races, on average, were due to SPSC queues. A slightly smaller figure can be appreciated for the application set, decreasing to 34 %. Generally, these percentages give a notion of the importance of this kind of data races occurring in SPSC queue-related functions used to stream data between FastFlow worker threads.

As stated in Section 5.3, the aim is to filter, whenever possible, those "benign" SPSC queue-related data races according to the requirements (5.1) and (5.2). Taking advantage of our implementation, in Figure 5.3 we classify SPSC data races in three different groups: benign, undefined and real. Benign data races represent those complying both requirements. Undefined data races stand for those in which TSan failed to restore the stack of one of the threads involved in the data race, and thus, the aforementioned requirements could not be checked. Finally, real data races stand for those in which, at least, one of the requirements was violated. Analysing the percentage breakdown of the different groups, we observe that large percentages (about 50% and 20% for the μ-benchmarks and application set, respectively), were classified as undefined. Since we are not aware of the specific cause that prevented TSan from restoring the stack, we are not confident to classify these data races as benign or real. A deep understanding of the TSan implementation is needed in order to investigate these undefined data races. This further step will be done as future work.

To gain insights into this issue, we performed an extra experiment considering the FastFlow's implementation of the bounded and unbounded SPSC queues plus the Lamport version.[3] These tests, `buffer_SPSC`, `buffer_uSPSC` and `buffer_Lamport`, corroborate that percentages of the undefined data races are independent of the queue version. Considering that all implementations are semantically correct, but data races still occurs when using TSan, we assume that they are all false positive and related to internal issue of TSan.

Similarly, Table 5.2 combines the breakdown for the different types of data races at SPSC and application levels for both benchmark sets. Additionally, it incorporates figures representing the total number of data races, average of data races per test, and the corresponding percentages over the total data races detected on the application, regardless of their source. Note that the analysis of the SPSC

[3]The codes of these structures can be found in the FastFlow's SVN repository https://sourceforge.net/p/mc-fastflow/code/HEAD/tree/, more specifically in file `ff/buffer.hpp`.

Table 5.2: Statistics of SPSC and application's total data races for the μ-benchmarks and applications sets.

Benchmark set	**Metrics**	**SPSC level**			**Application level**			**w/o SPSC semantics**	**w/ SPSC semantics**
		Benign	Undefined	Real	SPSC	FastFlow	Others		
μ-benchmarks	Total	187	93	0	280	213	102	595	408
	Per test	4.79	2.38	0.00	7.18	5.46	2.62	15.26	10.46
	Percentage	31.43 %	15.63 %	0.00 %	47.06 %	35.80 %	17.14 %	100.00 %	68.57 %
Applications	Total	60	12	0	72	55	83	210	150
	Per test	4.62	0.92	0.00	5.54	4.23	6.38	16.15	11.54
	Percentage	28.57 %	5.71 %	0.00 %	34.29 %	26.19 %	39.52 %	100.00 %	71.43 %

Table 5.3: Statistics of SPSC and application's unique data races for the μ-benchmarks and applications sets.

Benchmark set	**Metrics**	**SPSC level**			**Application level**			**w/o SPSC semantics**	**w/ SPSC semantics**
		Benign	Undefined	Real	SPSC	FastFlow	Others		
μ-benchmarks	Total	72	62	0	134	170	58	362	290
	Per test	1.85	1.59	0.00	3.44	4.36	1.49	9.28	7.44
	Percentage	19.89 %	17.13 %	0.00 %	37.02 %	46.96 %	16.02 %	100.00 %	80.11 %
Applications	Total	19	9	0	28	44	45	117	98
	Per test	1.46	0.69	0.00	2.15	3.38	3.46	9.00	7.54
	Percentage	16.24 %	7.69 %	0.00 %	23.93 %	37.61 %	38.46 %	100.00 %	83.76 %

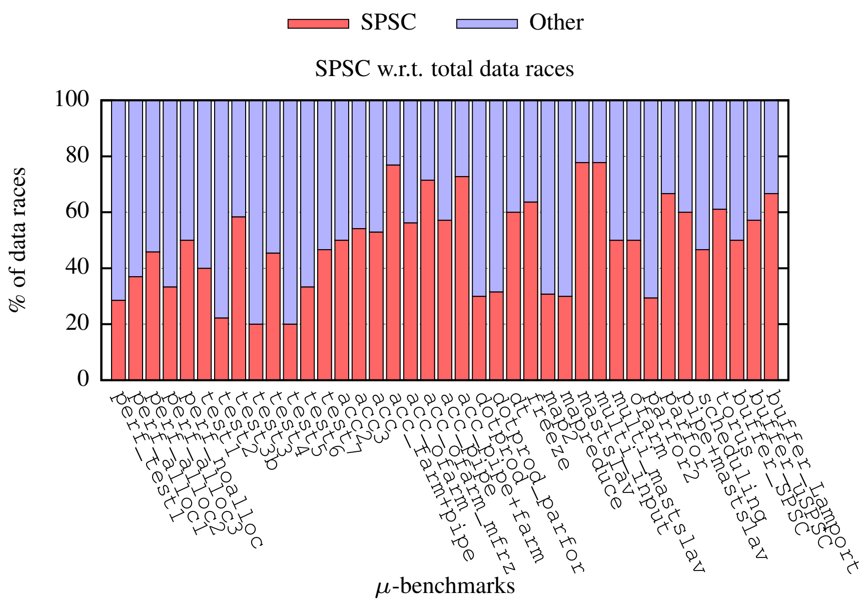
breakdown for the μ-benchmarks and applications sets has already been performed through Figure 5.3 and 5.2, respectively. In this table, however, we added a subdivision of data races related to FastFlow, but not to SPSC queues. As observed, this percentage represents approximately a third of total reports.

Finally, the last two columns of Table 5.2 present figures without and with the data race filtering technique, respectively. As can be seen, we reduce about a third of the number of warnings of data races for both sets tested. Being aware that in this case the filtering technique was only considering the SPSC queue, we are confident that more false positives could have been reduced if semantics for other parallel lock-free data structures would have been considered.

5.5.3 Analysis of unique data races

To conclude our analysis, we perform a third analysis in which we have filtered redundant data races out from the reports, thus keeping only unique entries. Results for the aforementioned metrics (total, average per test and percentage) are shown in Table 5.3. Focusing on the SPSC queue breakdown, we detect that portions of benign data races has been reduced to approximately 20 % and to 16 % for the μ-benchmark and application sets, respectively. However, percentages for the undefined SPSC data races data races are almost unchanged. Looking at the application level, the percentages of the SPSC data races have dropped accordingly, representing 37 % and 23 % over the total for both benchmark sets. This indicates that almost a 10 % of these data races occurred more than once. Therefore, the fact of dropping SPSC data races, affects positively to reduce noise in debugging traces.

For the data races related to the FastFlow framework, we note that the per-



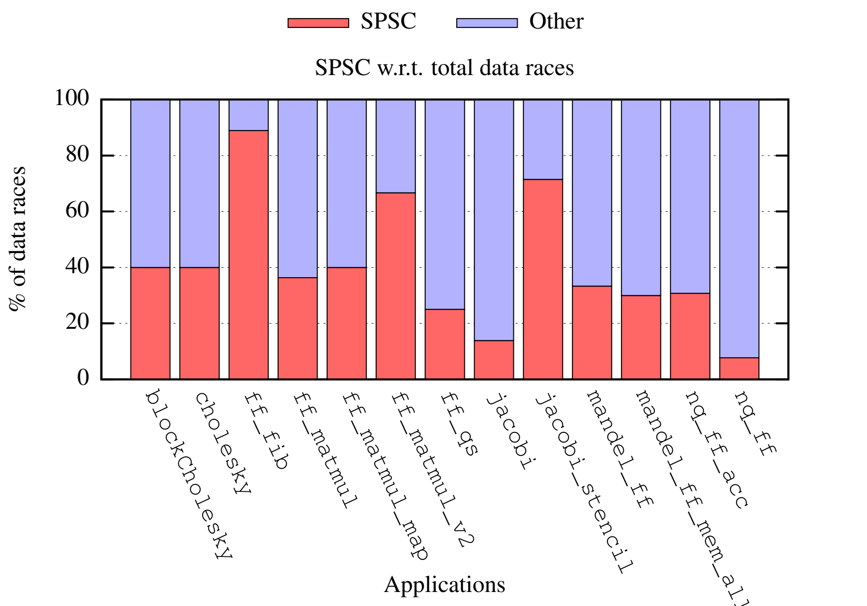
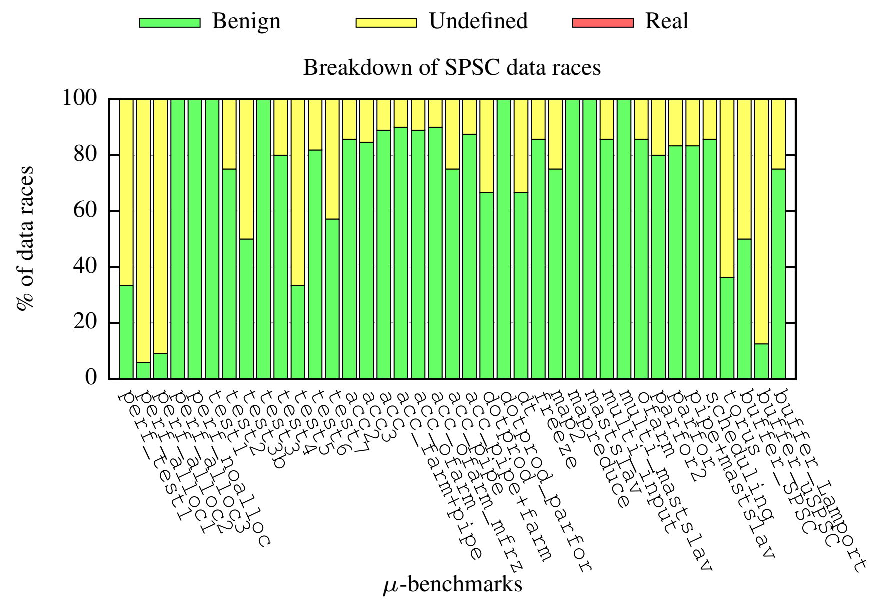


Figure 5.2: Percentage of SPSC data races with respect to the total for the μ-/benchmarks and applications sets.



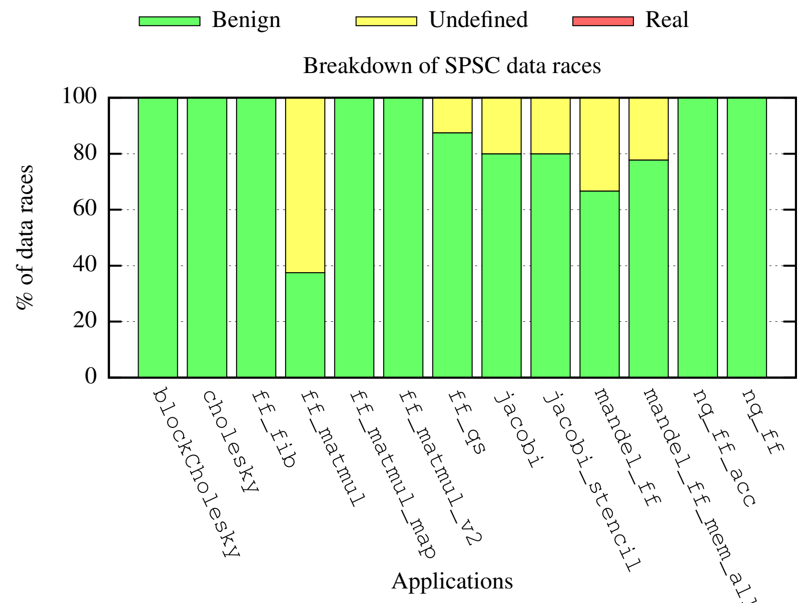


Figure 5.3: Breakdown of SPSC data races between benign, undefined and real for the μ-benchmarks and applications sets.

centages have increased, being almost a half and a third, for both benchmark sets, respectively. Thus, redundancy of this kind of data races was smaller than that for the SPSC queue (occurring mostly in the same pair of routines, as shown in Figure 5.1). Other data races remain constant, being again quite heterogeneous and stable with respect to the figures in Table 5.2.

5.6 Conclusions

Data race detectors aid to a great extent developers to easily identify data races in parallel applications. Several post-mortem and dynamical approaches for data race detection have been implemented among a range of tools and plug-ins for compilation infrastructures. However, none of them is aware of the semantics behind the data races. The concurrent use of a shared resource within a correct lock-free parallel structure should not always imply a data race, unless its semantics are violated. In this chapter, we have taken the Fastflow implementation of the general Single-Producer/Single-Consumer lock-free queue as a use case, to formalise its semantics for determining whether a queue instance has been properly used or not. The implementation of these semantics into ThreadSanitizer, allows us to filter false positives and prevent overwhelming users due to excessive false warning reports. Furthermore, we also ensure that data races detected are real through a second level of verification semantics. Through a series of experiments, we observe that this technique able to discard, on average, 66 % and 83 % of the data races detected set for the set of selected benchmark. Although for the sake of simplicity we have only covered the case of the SPSC bounded queue, in future work we plan to extend and formalise the semantics of other lock-free parallel data structures. For the case of FastFlow, we plan to consider queues and communication channels build on the top of the SPSC bounded queue, i.e., SPSC unbounded, one-to-many (SPMC), many-to-one (MPSC), and many-to-many (MPMC) queues.

6. Detection of Extra-Functional Property Detection

In this chapter, we describe the work done in T3.5, the task that deals with detection of violations of extra-functional properties, such as performance or energy consumption. Violation of these properties does not impact the results that the application produces, but can render a parallel application essentially useless, if the performance is unacceptable or energy consumption is higher than the energy budget. Here, we focus solely on issues that lead to decrease in the application performance, in terms of the delivered speedup and we plan to consider the issues related to energy in future deliverables. In the previous deliverable, D3.1, we described a preliminary version of the tool for detecting these violations and for discovering the reasons behind them, such as granularity problems and communication hotspots. This tool was based on *Callgrind*, a part of the *Valgrind* profiling suite, which means it provided a very accurate simulation of the application execution and very precise results, at the expense of potentially very high overheads. In this deliverable, we describe the state-of-the-art Performance Application Programming Interface (PAPI) library Interface (PAPI) that provides interface to the hardware *counters* which can record certain events related to how well the processors and memory perform, such as number of cache hits/misses, number of cycles stalled on various resources etc. We also describe how this library can be used for detection of violations of extra-functional properties of the application and discovering reasons for that. This allows us, for example, to discover automatically the situations where basic versions of patterns should be replaced with the versions that do data-replication (see Deliverbale D4.1). In Section 6.1 we give a brief overview of PAPI and how it can be used for detection of extra-functional properties violations. In Section 6.2 we demonstrate the use of PAPI on detection of extra-function property violation on a simple toy example. Section 6.3 outlines the future work.

6.1 Performance Application Programming Interface (PAPI)

Performance Application Programming Interface (PAPI) is a cross-platform library that provides tool designers and application programmers with a consistent interface and methodology for use of the *performance counter* hardware found in most major microprocessors[1]. Performance counters use a small set of hardware registers that count occurrences of specific signals related to the processor's function, such as number of cache misses and number of clock cycles. PAPI provides both simpler, *high-level* interface for acquiring simple measurements from these counters and more sophisticated *low-level* interface that provides more control of what signals are measured and how. Using these interfaces, a programmer can instrument his/her application to analyse critical pieces of code and discover reasons for performance bottlenecks. PAPI also works in multithreaded environment, where separate counters can be associated to separate threads to analyse each thread individually. This allows us to apply it to the applications parallelised using the **RePhrase** pattern set and discover reasons for violations of extra-functional properties.

6.1.1 PAPI Events

PAPI works by offering to the programmer a set of *events* that can be logged and counted using performance counters, and grouping them into *event sets*. The user creates an event set, adds events to this set and then starts and stops the event set to obtain counters for the events in it. There are two different kinds of events that can be counted – *native events*, which are hardware specific and are accessed via platform specific counters and *preset events*, which are generic events that are architecture independent. Preset events allow portability to different architectures by implementing the same, high-level event using different native events (or a combination of these events). Table 6.1.1 shows some of the more commonly used preset events that can be counted using PAPI. Figure 6.1 shows an example of the code that counts the number of Level 3 cache misses in a simple for loop.
Using the events from the table, we can discover a number of different problems that can severely impact the extra-functional properties of an application.

- Large number of cache misses (relative to the total number of cache accesses) can be an indication of bad data placement;
- Large number of cycles stalled waiting for memory access (relative to the total number of cycles) can point to congestion of the memory bus because many threads may be trying to access the same data, indicating that data-replication may be needed, or that data needs to be spread around different memory modes of a NUMA machine;
- Total number of cycles can be used to measure granularity of computations, pointing to too low/high granularity that can impact performance;
- Large number of requests for exclusive access to a shared cache line can point to the problem of *false sharing*, where data is shared in the cache,

[1]http://icl.cs.utk.edu/papi/index.html

```
retval = PAPI_library_init(PAPI_VER_CURRENT);
if (retval != PAPI_VER_CURRENT) {
  printf("PAPI␣library␣init␣error!\n");
  exit(1);
}

/* Create an EventSet */
retval = PAPI_create_eventset(&EventSet);
if (retval != PAPI_OK) {
  fprintf (stderr, "Error␣creating␣event␣setRROR\n");
  exit(1);
}

/* Check if the event is available at our hardware */
retval = PAPI_query_event(PAPI_L3_TCM);
if (retval != PAPI_OK) {
  fprintf (stderr, "Event␣cannot␣be␣counted\n");
  exit(1);
}

retval = PAPI_add_event(EventSet, PAPI_L3_TCM);
if (retval < 0) {
  fprintf (stderr, "Error␣adding␣event\n");
  exit(1);
}

retval = PAPI_start (EventSet);
if (retval != PAPI_OK) {
  fprintf (stderr, ``Error starting event set\n'');
  exit(1);
}

for (int i=0; i<10000; i++) {
    ...
}

retval = PAPI_stop (EventSet, values);
if (retval != PAPI_OK) {
  fprintf (stderr, "Stoping␣counters␣failed\n");
  exit(1);
}
```

Figure 6.1: An example of instrumenting the code using PAPI to measure the number of L3 cache misses

PAPI Event Name	Description
PAPI_L1_TCM	Level 1 cache misses
PAPI_L2_TCM	Level 2 cache misses
PAPI_L3_TCM	Level 3 cache misses
PAPI_L3_TCA	Level 3 cache accesses
PAPI_TOT_CYC	Total number of cycles
PAPI_CA_SHR	Requests for exclusive access to shared cache line
PAPI_MEM_SCY	Cycles stalled waiting for memory accesses
PAPI_TOT_INS	Instructions completed
PAPI_RES_STL	Cycles stalled on any resource
PAPI_LD_INS	Load instructions
PAPI_LST_INS	Load/store instructions completed
PAPI_FPU_IDL	Cycles floating point units are idle

Figure 6.2: PAPI Preset Events.

but independent threads frequently update the values that are in the same cache line, therefore requesting frequent need for exclusive access and thus hampering the performance.

6.2 Using PAPI to detect violations of extra-functional properties of patterned applications

In this section we show how PAPI can be used for performance debugging of patterned applications.

6.2.0.1 Toy Example - RandomArray

In this section, we consider an example of the application of a `map` (or `parallel_for`) pattern to the computation over an array, where the indices of the array that are accessed by the same thread are not necessarily adjacent (or even close). This kind of behaviour is manifested when, for example, a large shared data structure is kept in memory and different parts of it are accessed by different threads mapped to the cores of the same processor, for example Matrix Multiplication or Ant Colony Optimisation described in D2.2. This example shows how seemingly "perfect" parallelisation of an application may result in subpar performance. In our example, we apply a simple addition over random elements of a (possibly large) array;

```
1   int worker_function (int worker_index, int *array, int array_dim, int num_additions)
2   {
3     double x=0;
4     for (int i=0; i<num_additions; i++) {
5       int ind = rand();
6       x += array[ind % array_dim];
7     }
8     return x;
9   }
10
11  int main(int argc, char **argv)
12  {
13    int array_dim = atoi(argv[1])
14    int *a = (int *) malloc (sizeof(int)*array_dim);
15    int nr_iterations = atoi(argv[2])
16    int num_additions = atoi(argv[3])
17    double *res = (double *) malloc (sizeof(double)*nr_iterations);
18    for (int i=0; i<array_dim; i++)
19      a[i] = 1;
20
21    for (int i=0; i<nr_iterations; i++) {
22      res[i] = worker_function (i, a, num_additions);
23    }
24  }
```

The obvious parallelisation of this example is introducing a parallel for in place of the main for loop at line 19:

```
class WorkerClass {
public:
  void operator()(const blocked_range<size_t>& r) const {
    for (size_t i=r.begin(); i!=r.end(); i++) {
      res[i] = worker_function (i, a);
    }
  }

  WorkerClass() {}
};
```

```
int worker_function (int worker_index, int *array, int array_dim, int num_additions)
{
  double x=0;
  for (int i=0; i<num_additions; i++) {
    int ind = rand();
    x += array[ind % array_dim];
  }
  return x;
}

int main(int argc, char **argv)
{
  int array_dim = atoi(argv[1])
  int *a = (int *) malloc (sizeof(int)*array_dim);
  int nr_iterations = atoi(argv[2]);
  int num_additions = atoi(argv[3]);
  int nr_cpu_w = atoi(argv[4])
  double *res = (double *) malloc (sizeof(double)*nr_iterations);
  for (int i=0; i<array_dim; i++)
    a[i] = 1;

  task_scheduler_init init(nr_cpu_w);
  chunk_size = chunk_size/nr_cpu_w;
  parallel_for(blocked_range<size_t>(0, nr_iterations, chunk_size), WorkerClass());

}
```

We can see that the elements of the array that are accessed from the same loop iteration in the worker function may be distant. For large arrays, this can trigger many cache misses, especially when multiple threads execute on the cores of a multicore system that share the same cache memory. This, in turn, can result in many accesses to the memory bus by different threads and, in the case of larger number of threads, congestion of the memory bus since there is only one copy of the array in the memory, and many threads are competing for it. Indeed, we have conducted experiments with this application on the machine with 4 16-core AMD Opteron 6376 processors, with 512 GB RAM. This machine has 8 different NUMA memory regions, and cores of each processor are able to access the memory closest to them much faster than the more distant memory regions. We have noticed a drop in performance when larger number of cores is used, most notable drops happening when we run more than 32 threads and, hence, using processor cores from 4 or more NUMA regions.

To discover the reasons for decrease in performance when a larger number of cores is used, we instrument the parallelised version using PAPI. First thing that we measure is the number of cache misses, to verify that we indeed have caching problem. We consider the execution when 56 threads are used, and we measure the number and percentage of L3 cache misses for each thread separately when 56 threads are used. The instrumented code is very similar to the code show in Figure 6.1. A snapshot of the output of the instrumented version of the code is given below.

```
...
L3 accesses: 19971182, L3 misses: 8957195, Pct L3 misses; 0.448506
L3 accesses: 19911258, L3 misses: 8946280, Pct L3 misses; 0.449308
L3 accesses: 20163984, L3 misses: 9032903, Pct L3 misses; 0.447972
L3 accesses: 20368578, L3 misses: 9272790, Pct L3 misses; 0.455250
L3 accesses: 20124252, L3 misses: 9026775, Pct L3 misses; 0.448552
L3 accesses: 20290565, L3 misses: 9153340, Pct L3 misses; 0.451113
...
```

We can observe quite a high percentage of cache misses, which indicates that the applications might be frequently accessing the main memory. Ideally, the next events we would record would be `PAPI_MEM_SCY` events, to give us information about how many cycles the execution of the application has been stalled on memory accesses, and the high percentage of these would indicate the congestion of memory bus. However, this event was not available on the hardware that we had. So, instead of that, we recorded the `PAPI_FPU_IDL` evens to see how many cycles was floating point unit idle for. The following listing shows the output when 8 threads are used:

```
FPU Idle cycles: 4017978869, Total cycles: 8300908470, Percentage: 0.484041
FPU Idle cycles: 4022145700, Total cycles: 8339572507, Percentage: 0.482296
FPU Idle cycles: 4052507969, Total cycles: 8359054144, Percentage: 0.484805
FPU Idle cycles: 4032508812, Total cycles: 8343636547, Percentage: 0.483304
FPU Idle cycles: 4088243268, Total cycles: 8385287566, Percentage: 0.487550
FPU Idle cycles: 4087480383, Total cycles: 8411225642, Percentage: 0.485955
FPU Idle cycles: 4091774581, Total cycles: 8397389804, Percentage: 0.487267
FPU Idle cycles: 4094840546, Total cycles: 8392106566, Percentage: 0.487940
```

However, when we use 56 threads, we can observe a significant increase in the percentage floating point unit was idle. The following listing shows the snipped of the output when 56 threads are used:

```
...
FPU Idle cycles: 14133210759, Total cycles: 17919352381, Percentage: 0.788712
FPU Idle cycles: 14341535154, Total cycles: 18134246881, Percentage: 0.790854
FPU Idle cycles: 14209644722, Total cycles: 18019536941, Percentage: 0.788569
FPU Idle cycles: 14288537675, Total cycles: 18084440503, Percentage: 0.790101
FPU Idle cycles: 14285845428, Total cycles: 18107911613, Percentage: 0.788928
FPU Idle cycles: 14136104336, Total cycles: 18049862547, Percentage: 0.783170
FPU Idle cycles: 14331723742, Total cycles: 18150046791, Percentage: 0.789625
...
```

Due to relative simplicity of the application, where essentially the only operations are memory operations and a simple addition, we can conclude that, indeed, the main bottleneck for the performance when larger number of cores is used lays in expensive memory accesses. Therefore, instead of the regular TBB `parallel_for`, we can use the version that uses data-replication (described in D4.1). The following are the relevant parts of the new parallelisation of the application

```
...
std::vector<data_t> in_array(1);
data_t arr = {.p_data = (void *) a,
              .size = sizeof(int)*array_dim};
in_array[0] = arr;
...
parallel_for(blocked_range<size_t>(0, nr_iterations, chunk_size),
             MapReplicationClass<double>(worker_function,
                                         nr_cpu_w,
                                         in_array,
                                         res,
                                         chunk_size));
...
```

In this version, the array `a` is replicated over all NUMA nodes, and the threads mapped to the cores of different processor access different copies of the same array, therefore reducing the memory congestion and, therefore, improving performance.

6.3 Conclusions

In this section, we addressed the problem of detecting the reasons for violations of extra-functional properties of applications, focusing on the performance. We described the PAPI library that provides interface to hardware counters that can, with very little overhead, count some important events that give the indication of how well the processor and memory are performing. We then showed, on a simple example, how PAPI can be used for our purposes. Compared to the tool described in D4.1, this approach brings much less overheads in terms of the execution time of the instrumented application, while still providing good accuracy of the results. Ultimately, we will use a combination of the detailed profiling tool described in D4.1 and the instrumentation using PAPI to get different kind and level of measurements and pinpoint to different kind of problems with extra-functional behaviour. In the rest of the project, we plan to build a tool on top of the PAPI which will be able to automatically detect reasons for violations of extra-functional properties, and additional refactorings that will automatically insert the code for instrumenting applications in order to obtain relevant information. We also plan to demonstrate these mechanisms on larger benchmarks and use cases, including the industrial use cases provided by industrial partners in the **RePhrase** project.

8. Conclusion

This deliverable describes the application of the **RePhrase** framework for testing, verification and debugging of patterned application both to the libraries of parallel patterns, such as FastFlow and Intel TBB, and to the applications parallelised using patterns from these libraries. We restrict our attention to the applications that use the patterns from the initial pattern set described in D2.1. This demonstrates how the tools that we have provided in the software deliverable D3.1 can be used for testing, verification and debugging of parallel applications and, therefore, how we can increase their reliability, robustness and software integrity.

In Chapter 2 we described how we used the QA-Verify tools to detect potential problems in the FastFlow pattern library and proposed methods to avoid the detected problems. In Chapter 3, we described the adaptation of the Combinatorial Test Design (CDT) test planning technique to patterned applications and demonstrated it on the Mandelbrot example that uses the map pattern. This application is not trivially parallelisable, since if we start from the intuitive sequential version and parallelise it, we get a race condition. In Chapter 4, we described an algorithm for verifying equivalence between the sequential and patterned versions of the same program, where patterned version was obtained using refactorings described in D2.2. This algorithm is used by the IBM ExpliSAT to automatically validate the patterned applications, which we demonstrated on the same example, Mandelbrot, as in Chapter 3. Chapter 5 describes detection of data-races in the applications that use Single-Producer/Single-Consumer parallel data structure and the implementation of this technique in the Thread Sanitizer detection tool. We then demonstrated our findings when Thread Sanitizer was applied to benchmarks and real-world applications parallelised using FastFlow. We were able to discard large percentages, between 66% and 83%, of large positives reported by Thread Sanitizer in these examples, therefore significantly easing the programmer's task of finding the real data races. Finally, in Chapter 6 we addressed the problem of violation of extra-functional properties of parallel applications and proposed methods how to discover bottlenecks that are causing these violations.

As future work, we plan to integrate these tools more tightly into the overall **RePhrase** methodology and tool chain. We will also consider domain-specific patterns and will analyse additional refactorings that introduce these patterns. Finally, until the end of the project, we will extend the mechanisms for detecting violations of extra-functional properties to heterogeneous systems, comprising combinations of CPU and GPU processors.

Bibliography

[1] The mandelbrot set in c++11. http://solarianprogrammer.com/2013/02/28/mandelbroot-set-cpp-11.

[2] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, Massimiliano Meneghin, and Massimo Torquati. An Efficient Unbounded Lock-Free Queue for Multi-core Systems. In Christos Kaklamanis, Theodore Papatheodorou, and PaulG. Spirakis, editors, *Euro-Par 2012 Parallel Processing*, volume 7484 of *Lecture Notes in Computer Science*, pages 662–673. Springer Berlin Heidelberg, 2012.

[3] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, and Massimo Torquati. Fastflow: high-level and efficient streaming on multi-core. In *in Programming Multi-core and Many-core Computing Systems, ser. Parallel and Distributed Computing, S. Pllana*, page 13, 2012.

[4] Cyrille Artho, Klaus Havelund, and Armin Biere. High-Level Data Races. In *Journal On Software Testing, Verification and Reliability (STVR*, volume 13, page 2003. 207–227, 2003.

[5] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe programming: Preventing data races and deadlocks. *SIGPLAN Not.*, 37(11):211–230, November 2002.

[6] Kirk Burroughs, Aridaman Jain, and Robert L Erickson. Improved quality of protocol testing through techniques of experimental design. In *Communications, 1994. ICC'94, SUPERCOMM/ICC'94, Conference Record,'Serving Humanity Through Communications.'IEEE International Conference on*, pages 745–752. IEEE, 1994.

[7] Myra B Cohen, Joshua Snyder, and Gregg Rothermel. Testing across configurations: implications for combinatorial testing. *ACM SIGSOFT Software Engineering Notes*, 31(6):1–9, 2006.

[8] Murray Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, Cambridge, MA, USA, 1991.

[9] John Giacomoni, Tipp Moseley, and Manish Vachharajani. Fastforward for efficient pipeline parallelism: A cache-optimized concurrent lock-free queue. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, pages 43–52, New York, NY, USA, 2008. ACM.

[10] Mats Grindal, Birgitta Lindström, Jeff Offutt, and Sten F Andler. An evaluation of combination strategies for test case selection. *Empirical Software Engineering*, 11(4):583–611, 2006.

[11] Godfrey Harold Hardy and Edward Maitland Wright. *An introduction to the theory of numbers*. Oxford Science Publications. Clarendon Press, Oxford, 1979.

[12] ISO/IEC. Working draft, standard for programming language c++, 2016.

[13] D Richard Kuhn, Dolores R Wallace, and Albert M Gallo. Software fault interactions and implications for software testing. *IEEE transactions on software engineering*, 30(6):418–421, 2004.

[14] L. Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Trans. Comput.*, 28(9):690–691, September 1979.

[15] Leslie Lamport. Concurrent Reading and Writing. *Commun. ACM*, 20(11):806–811, November 1977.

[16] Savannah Project. The libunwind project. http://www.nongnu.org/libunwind/.

[17] N. Karunanithi J. M. Leaton C. M. Lott G. C. Patton S. R. Dalal, A. Jain and B. M. Horowitz. Model-based testing in practice. Proc. 21st International Conference on Software Engineering (ICSE'99), pages 285–294, 1999.

[18] Jeffrey Somers. The N Queens Problem, a study in optimization. http://www.jsomers.com/.

[19] Kuo-Chung Tai and Yu Lie. A test generation strategy for pairwise testing. *IEEE Transactions on Software Engineering*, 28(1):109, 2002.

[20] Alan W. Williams. Determination of test configurations for pair-wise interaction coverage. In *Proceedings of the IFIP TC6/WG6.1 13th International Conference on Testing Communicating Systems: Tools and Techniques*, TestCom '00, pages 59–74, Deventer, The Netherlands, The Netherlands, 2000. Kluwer, B.V.

[21] Paul Wojciak and Rachel Tzoref-Brill. System level combinatorial testing in practice–the concurrent maintenance case study. In *2014 IEEE Seventh*

International Conference on Software Testing, Verification and Validation, pages 103–112. IEEE, 2014.