Grand amphi IMAG

JEUDI 30 AOÛT

9h00
Paul Raynaud
Towards the automatic translation from C program to Horn clauses

9h40
Enzo Brignon
Runtime verification of logico-temporal properties for embedded C software

10h35
Florian Barrois
Combining Path and Cache Analysis for WCET estimation improvement

13h30
Maxime Calka
Semi-automatic segmentation of surgical instruments in minimally invasive surgery videos

14h10
Maxence Grand
Integrating lexical constraints to K-Means with Deep Learning

15h05
Christopher Ferreira
A System-Wide study of performance Issues in FaaS platforms

15h45
Antoine Delise
Deciding multivariate polynomials inequalities by combining factorization, Euclidian division and Handelman's theorem

VENDREDI 31 AOÛT

9h00
Alexandre Borthomieu
Automatic grading

9h30
Nils Defauw
Large scale traces analysis : multi-scale patterns

10h15
Adelina Prokhorova
Knowledge base mining using compressed data structures

10h45
Fabien Lefebvre
Natural language generation : Comparison of language models

11h30
Thomas Vandendorpe
Certifying answers of boolean SAT-solvers with C

UNIVERSITÉ Grenoble Alpes

MAGISTÈRE D'INFORMATIQUE - 2018

Le magistère est une option pour les étudiants de L3 à M2 souhaitant avoir de l'expérience dans le domaine de la recherche

MIGA'2018

Proceedings du Magistère d'Informatique 2018 de l'Université Grenoble-Alpes

Organized by Michaël Périn and Cyril Labbé

August, 30-31, 2018

Program

Towards the automatic translation from C program to Horn clauses [*]

Paul Raynaud
VERIMAG
Grenoble, France
paul.raynaud66@hotmail.fr

Supervised by: Michaël Périn.

I understand what plagiarism entails and I declare that this report is my own, original work.
Raynaud Paul, 24/08/2018 : [illegible]

Abstract

Automatically checking if a program realizes its specification allows us to prove something stronger than the mere absence of certain errors ; we are outright capable of proving the correction of the program. Therefore we are trying to prove, based on a source program, whether it's capable of satisfying the properties that are attached to it or not with an entirely automatic procedure.

We will rely on already known hypothesis such as Hoare Logic, and control flow graph study to generate Horn clauses. Once those logical formulas are generated, we resort to a SMT (Satisfiability modulo theories) solver to resolve the so-called formulas.

1 Related work

1.1 The formal verification and CompCert

Tools capable of realizing programs verification have been developed, like SeaHorn. However, as will be discussed later, those tools are not compatible with the "recent" compiler CompCert C. With the unique semantic of CompCert, being capable of automatically verifying a program would be an interesting feature to be added. Thereafter, on a code, the specification of which has been proven, we could have a proved program during the compilation without integrating errors by a compiler of a "hazardous" semantic.

1.2 SeaHorn

A tool like SeaHorn ([Gurfinkel *et al.*, 2015]) is practically based on the same hypothesis that are used in this internship, with the only difference that SeaHorn takes C and C++ programs and uses the bitecode generated by Clang [Fan, 2010]. Clang compiler is different than ComCert C. Therefore it doesn't have the same semantic. As a result, the generated code is different. It is for this exact reason that we cannot use SeaHorn.

The general reason why we cannot re-use the programs verification tools that already exist is not only that those tools were not developed with the same semantic as C but also that the in dependant development of a tool attached to CompCert C is preferable if there are modifications to be made.

1.3 CompCert

Yang and al [Yang *et al.*, 2011] realized a study with the purpose of showing the miscompilation of the different C compilers. All the compilers tested within their study generated incorrect codes for certain tests (including CompCert). Let us note that CompCert was not entirely developed in 2011, but it was already reducing bugs in comparison with the other compilers :

> "The striking thing about our CompCert results is that the middle- end bugs we found in all other compilers are absent. As of early 2011, the under-development version of CompCert is the only compiler we have tested for which Csmith cannot find wrong-code errors. This is not for lack of trying : we have devoted about six CPU-years to the task. The apparent unbreakability of CompCert supports a strong argument that developing compiler optimizations within a proof framework, where safety checks are explicit and machine-checked, has tangible benefits for compiler users".

CompCert is a project initiated in 2005 that aims to create the first proven C compiler [Blazy and Leroy, 2005]. It is a project with a complex structure that has been entirely proven. It is mainly developed in Coq. Using the Coq code, it is possible to extract Ocaml code. It is precisely on this Ocaml code that we are working.

2 The concept of Translation

2.1 Expected result

The global goal that is expected from our tool is specified in the Figure 1. We will go through the detailed translation, step by step.

[*]. These match the formatting instructions of IJCAI-07. The support of IJCAI, Inc. is acknowledged.

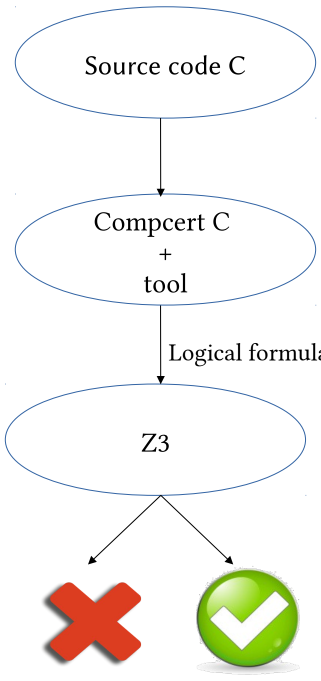


FIGURE 1 – The goal of our tool

2.2 The description of our example

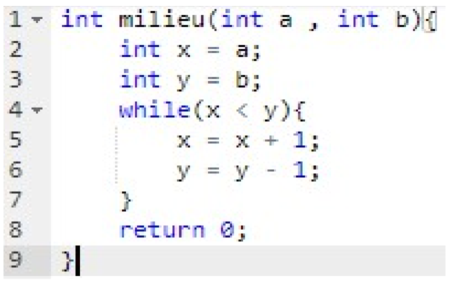
To illustrate our statements all along this paper, we will use an example of C code "middle.c".

The algorithm shown in Figure 2 calculates the middle value of an interval simply by increasing x and decreasing y while y is greater than x.

At the end of the algorithm we have the following properties

y $\geq (a+b)/2 \geq x \wedge 0 \leq y-x \leq 1$

that frame the middle value.



```
int milieu(int a , int b){
    int x = a;
    int y = b;
    while(x < y){
        x = x + 1;
        y = y - 1;
    }
    return 0;
}
```

FIGURE 2 – Our example program "middle.c"

However, it is not trivial to "manually" prove this algorithm correction.

This algorithm is interesting because if any small change is made to the loop

$$< by \leq$$

the properties stated previously are no longer valid since y-x cannot be worth 2.

Therefore, making a small change in our logical formulas is enough to judge whether our tool works correctly or not.

2.3 From a Control Flow Graph to Logical Formulas

Let's take a simple example, the C code and RTL translation of which are defined in Figure (...).

We will first recall the formulas of Hoare's logic :

A triplet of Hoare is :

$$\{\rho\}program\{\gamma\}$$

where ρ represents the pre-condition, and γ the post-condition of the program.

There are 2 methods for generating the logical formulas of a program :

- Strongest Postcondition, that generates the formulas forward.

- Weakest Precondition, that generates the formulas backward. We will be using the Weakest Precondtion.

We will now explain the rules concerning Weakest Precondition : A program to be verified must satisfy :

$$WP : \frac{\rho \Longrightarrow WP(instr, \gamma)}{\{\rho\}\, instr\, \{\gamma\}}$$

$\gamma\, is\, the\, preconditionand \rho\, the\, postcondition.$

The sequence :

$$\frac{\{\rho\}\, S1\, \{\theta\}\, \{\theta\}\, S2\, \{\gamma\}}{\{\rho\}\, S1;\ S2\, \{\gamma\}}$$

where $\theta\, a\, descending\, property\, from\, S1\, and\, S2$

The affection :

$$\frac{}{\{\rho\}\, x\ :=\ e\, \{\gamma[x/e]\}}$$

The condition :

$$\frac{\{\rho \wedge C\}S1\{\gamma\} \wedge \{\rho \wedge \neg C\}S2\{\gamma\}}{\{\rho\}if\, C\, then\, S1\, else\, S2\{\gamma\}}$$

The implication :

$$\frac{\{\rho\} \Rightarrow \{\rho'\} \wedge \{\gamma'\} \Rightarrow \{\gamma\} \wedge \{\rho'\}i\{\gamma'\}}{\{\rho\}i\{\gamma\}}$$

We must now apply these formulas to our graph from the RTL representation Figure.

3 Contribution

Theoretical principle

We were integrated in CompCert as soon as it created the first CFG. We were able to use the principles mentioned previously, along with an already defined Hoare Logic. To apply them in CompCert, we will need to re-write the rules and adapt them following the intermediate language instructions that we are tapping : RTL (Register Transfer Language). Afterwards, we will pass the obtained logical formulas to a SMT solver to solve them Z3 [de Moura and Bjørner, 2007].

3.1 The work provided

CompCert C data recovery

First of all, we have to relate to CompCert, an industrial compiler. We have to fully understand the global architecture of the software of CompCert and the way the different files are linked before working on it. Despite the fact that we have had a relatively easy access to the needed data, it was not enough for a smooth manipulation. In fact, to modify something in the purpose of testing it, we had to recompile CompCert. For this reason, we used a particular Caml library (ppx_deriving.show) to retrieve the data structure handled by CompCert.

The principle of the library is to automatically generate printers for all the types : sum, record or other. In order to ensure this, we needed to modify not only the files where the types that we wanted to display had been defined but also the ones where we have defined all the basic types.

In Figure 3 we have the classic display of RTL by CompCert C. Below we detail an instruction and show the structure of a RTL instruction.

9 : x4 = x2

Here is the 9th line in our program. This is the way CompCert displays the following instruction, which is the real structure that we get from the library ppx$_{deriving.show}$.

(RTL.Iop (Op.Omove, [(BinNums.Coq_xO BinNums.Coq_xH)], (BinNums.Coq_xO (BinNums.Coq_xO BinNums.Coq_xH)), (BinNums.Coq_xO (BinNums.Coq_xO (BinNums.Coq_xO BinNums.Coq_xH)))))

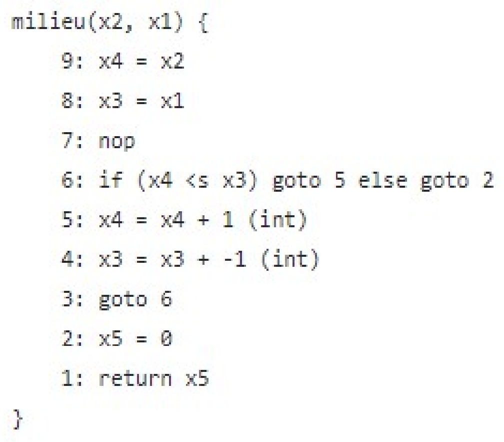
Now let's see in details what it means :

(RTL.Iop $\longrightarrow$ indicates that this is an operation

(Op.Omove, $\longrightarrow$ same as a move in assembler code

(BinNums.Coq_xO BinNums.Coq_xH)

$\longrightarrow$BinNums defined binear 1 and 0, here we have $10_2 = 2_{10}$, for x2



```
milieu(x2, x1) {
    9: x4 = x2
    8: x3 = x1
    7: nop
    6: if (x4 <s x3) goto 5 else goto 2
    5: x4 = x4 + 1 (int)
    4: x3 = x3 + -1 (int)
    3: goto 6
    2: x5 = 0
    1: return x5
}
```

FIGURE 3 – Display of the representation RTL of "middle.c" during its compilation by CompCert C

(BinNums.Coq_xO (BinNums.Coq_xO BinNums.Coq_xH))$\longrightarrow$ we have $100_2 = 4_10$, for x4

(BinNums.Coq_xO (BinNums.Coq_xO (BinNums.Coq_xO BinNums.Coq_xH))))) $\longrightarrow$ we have $1000_2 = 8_10$, 8 for the next instruction

Implementation of a first structure of CFG

The compilation was a bit complicated but once the structure of the compiled code by CompCert was extracted, we obtain a list of the Control Flow Graphs, each one represents a function/procedure.

By sorting and modifying the tree instructions we have implemented a first new structure that is better suited to the representation of a CFG, a structure by triplet.

Once our starting tree transformed into a CFG as a triplet structure, we have developed a printer in the format .dot allowing the display of a graph using the graphviz software [Ellson *et al.*, 2003].

Generation of Logical Formulas

Once the CFG recovered, we can start to generate the logical formulas, each one independently, by following Hoare rules. The formulas obtained for our program are the following :

$\forall x_1, x_2, x_3, x_4, x_5,$

precond(x$_1, x_2, x_3, x_4, x_5) \Rightarrow P9(x_1, x_2, x_3, x_2, x_5)$

$\forall x_1, x_2, x_3, x_4, x_5,$

P9(x$_1, x_2, x_3, x_4, x_5) \Rightarrow P8(x_1, x_2, x_3, x_2, x_5)$

$\forall x_1, x_2, x_3, x_4, x_5,$

P8(x$_1, x_2, x_3, x_4, x_5) \Rightarrow P7(x_1, x_2, x_1, x_4, x_5)$

$\forall x_1, x_2, x_3, x_4, x_5,$

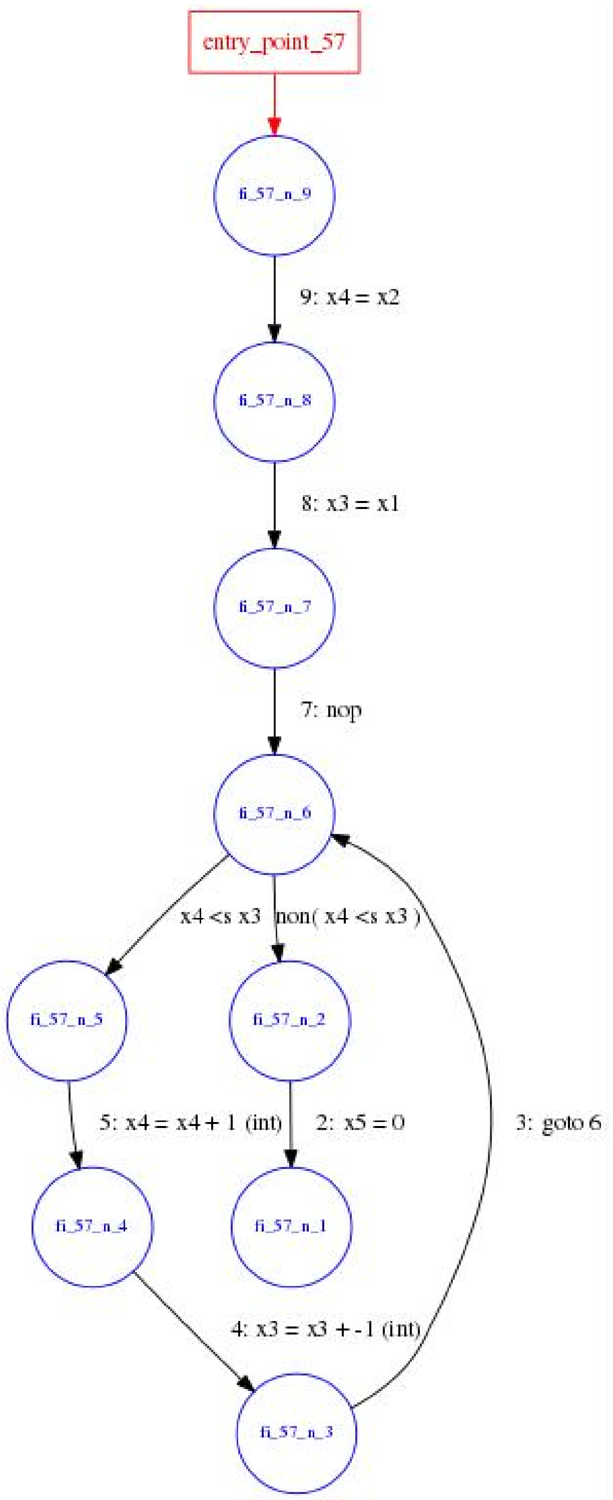


FIGURE 4 – Control Flow Graph of the program "middle.c"

P7($x_1, x_2, x_3, x_4, x_5) \Rightarrow P6(x_1, x_2, x_3, x_4, x_5)$
$\forall x_1, x_2, x_3, x_4, x_5,$

P6($x_1, x_2, x_3, x_4, x_5) \wedge (x_4 < x_7) \Rightarrow P5(x_1, x_2, x_3, x_4, x_5)$
$\forall x_1, x_2, x_3, x_4, x_5,$

P6($x_1, x_2, x_3, x_4, x_5) \wedge \neg(x_4 < x_7) \Rightarrow P2(x_1, x_2, x_3, x_4, x_5)$
$\forall x_1, x_2, x_3, x_4, x_5,$

P5($x_1, x_2, x_3, x_4, x_5) \Rightarrow P4(x_1, x_2, x_3, x_4 + 1, x_5)$
$\forall x_1, x_2, x_3, x_4, x_5,$

P4($x_1, x_2, x_3, x_4, x_5) \Rightarrow P3(x_1, x_2, x_3 - 1, x_4, x_5)$
$\forall x_1, x_2, x_3, x_4, x_5,$

P3($x_1, x_2, x_3, x_4, x_5) \Rightarrow P6(x_1, x_2, x_3, x_4, x_5)$
$\forall x_1, x_2, x_3, x_4, x_5,$

P2($x_1, x_2, x_3, x_4, x_5) \Rightarrow P1(x_1, x_2, x_3, x_4, 0)$
$\forall x_1, x_2, x_3, x_4, x_5,$

P1($x_1, x_2, x_3, x_4, x_5) \Rightarrow postcond(x_1, x_2, x_3, x_2, x_5)$

These are the formulas (simplified) resulting of the CFG ref. Figure 4 Pi ⇒ Pj indicates that the node i is pointing at the node j and the transformation is done in the parameters of Pj.

The choice of the SMT Solver

To resolve the formulas that we have generated, we can choose between many SMT solvers to do that task. However, since we are working on a project similar to Gurfinkel's that uses Z3 and works just fine, we naturally chose to simply use the same SMT solver.

Adaption of the logical formulas to Z3

Once the formulas generated, we need to transform them so that it suits Z3. A function definition (predicate) in Z3 is defined as follows :

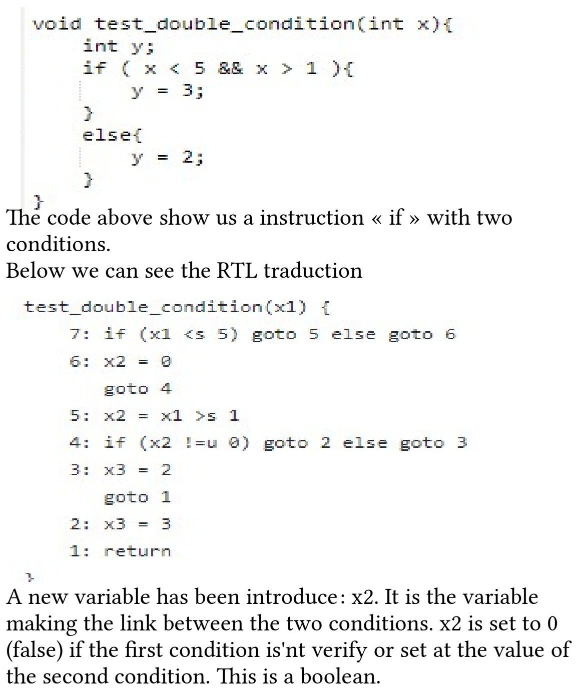
(declare-fun((type_var1) (type_var2)... (type_varN)) Bool)

Here we have a predicate declaration (Bool) with N variables.

To declare Z3 predicates, we have to get the types of all the variables of the program.

However, the language RTL is close to the Assembler language and manipulates pseudo-registers. Therefore the variables (or pseudo-registers) are not typed.

To find the types of the variables in RTL, we have to rebuild the types using the operations in which each variable participated.



```
void test_double_condition(int x){
    int y;
    if ( x < 5 && x > 1 ){
        y = 3;
    }
    else{
        y = 2;
    }
}
```

The code above show us a instruction « if » with two conditions.
Below we can see the RTL traduction

```
test_double_condition(x1) {
    7: if (x1 <s 5) goto 5 else goto 6
    6: x2 = 0
       goto 4
    5: x2 = x1 >s 1
    4: if (x2 !=u 0) goto 2 else goto 3
    3: x3 = 2
       goto 1
    2: x3 = 3
    1: return
}
```

A new variable has been introduce: x2. It is the variable making the link between the two conditions. x2 is set to 0 (false) if the first condition is'nt verify or set at the value of the second condition. This is a boolean.

FIGURE 5 –

x3 = x3 + -1 (int)
from this operation we can deduce
that x3 is an integer variable

This is possible thanks to the multitude of possible operations defined in the RTL representation of CompCert which make it possible to find the types.

However to have a better use of Z3 we could decide to use Booleans instead of the variables that only realize conditions. Particularly useful on intermediate registers added by the compiler making the link between 2 conditions (as defined in the figure 5.

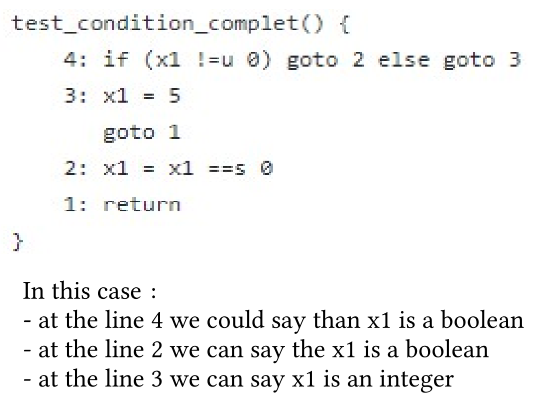
It is nevertheless possible for an integer to be used as a boolean temporarily (as defined in Figure 6)

We have done a hierarchy of types $Min < Bool < Int < Float$. to proceed afterwards to a homogenization of the types by creating a link between all the instructions of the program. After that, the types of the arguments given to the function must be compared. It's crucial that everything remains consistent. (as defined in Figure 7).

Thereby, we can find the types of all the program's variables.

In our example, it is simple because as it turns out there are integers only. (voir Figure 8).

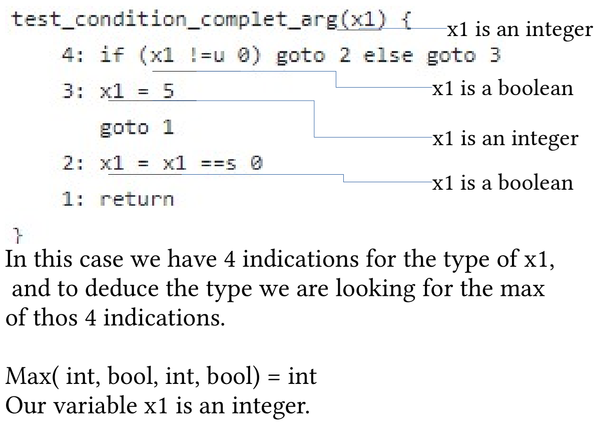
Give an example of 2 functions definitions and the logical formula that makes the link between them. In our case, we will write the formula that draws the link between the lines 9 and 8 :



```
test_condition_complet() {
    4: if (x1 !=u 0) goto 2 else goto 3
    3: x1 = 5
       goto 1
    2: x1 = x1 ==s 0
    1: return
}
```

In this case :
- at the line 4 we could say than x1 is a boolean
- at the line 2 we can say the x1 is a boolean
- at the line 3 we can say x1 is an integer

FIGURE 6 –

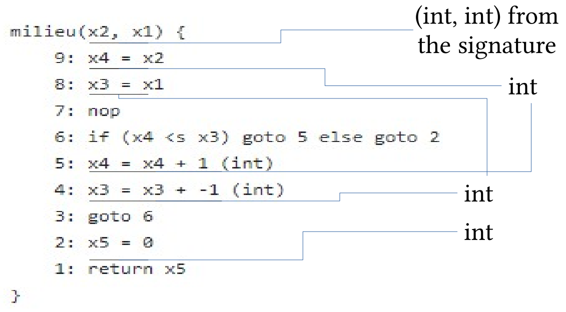


```
test_condition_complet_arg(x1) {
    4: if (x1 !=u 0) goto 2 else goto 3
    3: x1 = 5
       goto 1
    2: x1 = x1 ==s 0
    1: return
}
```

In this case we have 4 indications for the type of x1, and to deduce the type we are looking for the max of thos 4 indications.

Max(int, bool, int, bool) = int
Our variable x1 is an integer.

FIGURE 7 –



```
milieu(x2, x1) {
    9: x4 = x2
    8: x3 = x1
    7: nop
    6: if (x4 <s x3) goto 5 else goto 2
    5: x4 = x4 + 1 (int)
    4: x3 = x3 + -1 (int)
    3: goto 6
    2: x5 = 0
    1: return x5
}
```

Therefore x1,x2,x3,x4,x5 are integer variables

FIGURE 8 –

$$\forall\, x_1, x_2, x_3, x_4, x_5,$$

P9(x$_1, x_2, x_3, x_4, x_5) \quad \Rightarrow \quad P8(x_1, x_2, x_3, x_2, x_5)$
par
(declare-fun P_9 (Int Int Int Int Int) Bool)
(declare-fun P_8 (Int Int Int Int Int) Bool)

(forall ((x1 Int) (x2 Int) (x3 Int) (x4 Int) (x5 Int))
  (⇒
    (P_9 x1 x2 x3 x4 x5)
    (P_8 x1 x2 x3 x2 x5) ; from x4 = x2
)
)

Once the types are determined, we write the predicates as well as the logical formulas associated to the program while transcribing solely our formulas into the format Z3.

Adding of the pre/post condition

Whatever concerns the C code formulas is therefore written automatically.

We need to define thereafter the true properties at the beginning of the program (pre-condition) and the ones that we wish to prove at the end of the program (post-condition).

The writing of the pre-condition and the post-condition is left to the developer because the properties that he wishes to show are the ones that should be written. The links between the pre-condition od the program and the post-condition are written automatically, only the bodies of the pre-condition and the post-condition should be written.

Interpretation of Z3 results

When everything is written, it only remains to launch Z3 and wait for the result. With the way the formulas have been written, if Z3 responds with sat then the program is proven. It's not proven if it responds with unsat.

We would have preferred having more information when Z3 responds with unsat, unfortunately we couldn't dig further in the returns of Z3. Get-proof returns a proof that there is no way to satisfy the formulas but it is illegible.

Z3 can also respond with unknown or timeout. And in those cases, we don't have further information.

We remain very dependent on the solver. If it can't solve our formulas, even if they are correct, we need another way to proceed.

Observation on Z3

The pre-condition has a significant impact on the response of Z3.

We fix the properties $0 \leq x \leq y \leq parameter$ We have to put the results into perspective since the tests weren't completely under control. Realized on Z3 online (https ://rise4fun.com/Z3) , based on the formulas defined earlier (written in Z3).

parameter	time
3	2
5	3
7	7
8	7.7
9	timeout (12)
10 and +	timeout

Given that the experiment was not quite normalised, I was not able to go further in an estimation in function on the size of the parameters, however it is obvious that the bigger the space of the pre-condition, the longer the response will be.

If Z3 give us a time-out, we can restrain our pre-condition space to do local verification. That can give us indication about the program completeness but that don't prove the completeness.

4 Conclusion

4.1 Prove a small program

It was possible to prove a small program. We have also proven that if we did a small modification $\leq$. the program was no longer valid. Since the proof is annoying to do by hand, we can consider that the internship mission is accomplished.

4.2 The representation RTL

The real goal of the internship was to check if the representation RTL had the necessary information to realize this verifier. During this internship, I haven't had time to get into the details of all the RTL language instructions, however, based on what i have done I estimate that there are enough information to implement the program verifier from this representation. It will not be inevitably immediate but the necessary information are there.

+ : . The CFG
   . The functions have a lot of information
   . Possibility of finding the unavailable information

- : . Not everything is straight forward, there is some work that has to be done.

5 Possible and likely extensions

5.1 Addition of a property to test in the program

The idea is to replace the functions calls assume() and assert() of the C source code by some properties to verify in the program, exactly where there are functions calls.

Try to recover information of SMT solver when it returns unsat to see which properties remain unverified.

Apply the ideas of other projects that are aiming to prove code. Some of them are in the bibliography [Temesghen kahsai, 2015].

5.2 Mid-term objective

It is necessary to eventually complete this tool to treat all the RTL language instructions. By doing so, my work will be completed and a verdict on the relevance of the intermediary RTL representation for the realization of a verifier will be reached.

If it is not possible to treat certain instructions from the RTL representation then the tool will not be usable. It may be necessary to transmit other information from the previous steps into the compilation. If all this is done, the tool will be the equivalent of Seahorn or Boogie, adapted to the CompCert C compiler.

5.3 To be synchronized with CompCert

To remain in the same logic as CompCert, we have to prove this software, the same way as CompCert and develop this checker (verifier) in Coq, which is, according to Mr. Périn, would be the work of a thesis.

If the project came to an end, then a program compiled by CompCert C (and our tool) would be able to have properties, and maintain them throughout the compilation.

Références

[Blazy and Leroy, 2005] Sandrine Blazy and Xavier Leroy. Formal verification of a memory model for C-like imperative languages. In *International Conference on Formal Engineering Methods (ICFEM 2005)*, volume 3785 of *Lecture Notes in Computer Science*, pages 280–299. Springer, 2005.

[de Moura and Bjørner, 2007] Leonardo de Moura and Nikolaj Bjørner. Efficient e-matching for smt solvers. *21st International Conference on Automated Deduction, Bremen, Germany,*, volume 4603 of Lecture Notes in Computer Science :183–198, July 2007.

[Ellson *et al.*, 2003] John Ellson, Emden R. Gansner, Eleftherios Koutsofios, Stephen C. North, and Gordon Woodhull. Graphviz and dynagraph – static and dynamic graph drawing tools. In *GRAPH DRAWING SOFTWARE*, pages 127–148. Springer-Verlag, 2003.

[Fan, 2010] *Clang/LLVM Maturity Report*, Moltkestr. 30, 76133 Karlsruhe - Germany, June 2010. *See* `http://www.iwi.hs-karlsruhe.de`.

[Gurfinkel *et al.*, 2015] Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A. Navas. The seahorn verification framework. *27th International Conference on Computer Aided Verification (CAV 2015); 18-24 Jul. 2015; San Francisco, CA; United States*, pages 1–17, 2015.

[Temesghen kahsai, 2015] Dejan Jovanovic Martin Schäf Temesghen kahsai, Jorge A. Navas. finding inconsistencies in programs with loops. pages 1–15, 2015.

[Yang *et al.*, 2011] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. *Proceedings of the 2011 ACM SIGPLAN Conference PLDI, San Jose*, pages 1–17, June 2011.

Runtime verification of logico-temporal properties for embedded C software [*]

Enzo Brignon
TIMA Lab. (Univ. Grenoble Alpes, CNRS)
46 Av. Felix Viallet, Grenoble, France
Enzo.Brignon@univ-grenoble-alpes.fr

Supervised by: Laurence Pierre

Abstract

The reliability of C and C++ software is a main concern in many application domains such as embedded systems, in particular in critical systems. In this context, our team has developed OSIRIS, a tool that automatically instruments C programs with temporal property checkers designed for runtime verification of properties specified in PSL. The long term goal of this project is to give the ability to express properties over software execution for debugging purpose and fault tolerance analysis. The purpose of the work presented here is to propose alternative instrumentation solutions that increase portability and minimize the CPU time overhead induced by the current observation model.

1 Introduction

The reliability of C or C++ software is a main concern in many application domains, in particular embedded and critical systems such as avionics, automotive and aerospace. Indeed malfunction in firmware embedded in critical systems can cause huge problems. Hence it is of utmost importance to provide methods to prevent runtime errors and to ensure that the behavior of programs does not differ from their specification.

Verifying properties on firmware is a complex task because of the interactions between the software and the hardware. Embedded software uses hardware mechanisms, such as interrupts, that make debugging difficult to perform. There is a need of methods for verification of C firmware. One way to do that is to use *logico-temporal properties* that facilitate debugging through assertions and that can also ease fault tolerance analysis. A simple example is the following : "The variable v becomes greater than 100 before the function f is called". This example takes into account **temporal aspects** as *something happens before something else* and has to **observe** variable assignments with certain values, and function calls.

Static analysis is a commonly used method for program validation. It uses formal models of the system under verification (finite state machines for example) in order to automatically verify if these models satisfy the specification. Model checking [7] has the advantage to give an exhaustive view of the program behavior but has difficulty to scale when the number of variables becomes large. Others methods are proposed that do not face the scalability limit by doing dynamic verification (i.e., runtime verification), yet they focus on instances of the problem. There exists some tools that make such analysis among them [8] [13] [16]. Our team has developed the OSIRIS [5] tool (inspired by the ISIS tool [10]) that offers automatized methods for instrumentation of C programs (from source or binary) for assertion-based verification of properties specified in PSL [2].

This paper presents the adaptation of the OSIRIS tool to support two new instrumentation solutions that are proposed for binary firmware compiled for ARM processors, for bare metal context (using software interrupts) and for both contexts (OS and bare metal) using assembly code insertion. It also reports some experimental results for some significant properties dedicated to two use cases.

2 Temporal properties for software

To monitor a program, properties have to be specified in order to check if something wrong happens. These properties involve events and values to get temporal and quantitative aspects.

For example, a property can be expressed in natural language as follows : "every time variable v_1 is reset then the value of the variable v_2 becomes greater than 100 before the function f is called". In this property we can identify the quantitative and temporal aspects, the reset of v_1, the assignments to v_2 and the calls to f are **events** that are used to construct the execution trace (see Fig. 1) on which we have evaluation points which are where we have to evaluate the property. This is the

[*]These match the formatting instructions of IJCAI-07. The support of IJCAI, Inc. is acknowledged.

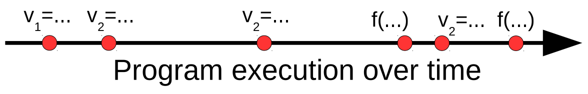


Figure 1: Example execution trace

temporal aspect. The quantitative part is denoted by the "greater than 100" that means that, in addition to event occurrence, we have to check the value contained in the variable v_2.

2.1 Brief overview of PSL

In order to specify such properties we use the standard Property Specification Language (IEEE std. 1850) [2] for multiple reasons. It is a standard with intuitive keywords for temporal logic. In addition to that, PSL can be used with any programming language because of the fact that Boolean expressions use the syntax of the language on which it is applied (here, Boolean expressions will borrow the C syntax). All that makes it easy and intuitive to employ.

Some commonly used operators for the specification of temporal assertion are the following :

always φ is satisfied on the trace if φ remains satisfied all along the trace.

next! φ is satisfied if φ is satisfied from the next evaluation point.

φ **until!** b is satisfied on the trace if there exists an evaluation point where b is satisfied, and φ is satisfied until that point.

$b1$ **before!** $b2$ is satisfied in the trace if $b1$ is satisfied at least once strictly before $b2$.

eventually! φ is satisfied on the trace if there exists an evaluation point from which φ is satisfied.

$b \rightarrow \varphi$ is the logical implication.

In addition to this "Temporal layer", the PSL "Modeling layer" allows to declare and give behavior to auxiliary signals and variables. For a C flavor, it consists of declarations and statements which would be legal within a C function.

2.2 Specific observer events

To express events in PSL assertions, we extend the language with 4 operators [5]. Given that v is a variable, f is a function and $f.v$ is a local variable or parameter of the function f :

v**#SET()** is true when an assignment to the variable v is being executed.

v**#GET()** is true when a reading of the variable v occurs.

f**#START()** is true when the function f is called.

f**#END()** is true when the function f returns.

Given the natural language example stated earlier "every time variable v_1 is reset then the value of the variable v_2 becomes greater than 100 before the function f is called", we can specify a PSL assertion that characterizes the property as follows :

```
always((v1#SET() && v1 == 0) ->
          ( (v2#SET() && v2 > 100)
            before! f#START()) );
```

2.3 OSIRIS monitors

Once PSL formulae are written, OSIRIS automatically generates monitors that will check these assertions along the execution. A monitor is a software component able to check a property, which observes the instrumented program in order to evaluate this property.

Since our logic model uses temporal assertions, a monitor has to resume any time an event that involves its property occurs. These events are defined using the operators of section 2.2. In our example, the generated monitor has to be updated upon any occurrence of either an assignment to the variable v_1 or v_2 (due to v_1#SET() and v_2#SET() conditions) or the call of the function f (due to the f#START() condition).

Events are the main element of the observation mechanism since the evaluation is performed only when they occur. The work presented here focuses on the definition and the implementation of solutions for this observation process, in the context of embedded systems, in the presence of an operating system or not.

3 Existing solution for software instrumentation (OSIRIS tool)

Once the monitors are generated, we still need to set an *observation mechanism* in order to notify them at the right time and with the right values. To do that, the original OSIRIS solution implements two alternative methods [5]. We recall them in this section after explaining the use of the observer design pattern.

3.1 Observer design pattern

To monitor the program, OSIRIS uses an adaptation of the Observer design pattern [11] that enables objects to notify observers each time they are modified. Observers are used to make the connection between a monitor and a variable. There are four component used to implement this design pattern, the *subject*, the *observer*, the *wrapper* and the *monitor*. The subject contains a list of wrappers (that inherits from the observer) that contains a list of monitors. Each variable or function that is involved in a property is bound to a subject, that notifies all its observers each time an event that relies on the corresponding variable occurs, thus, it triggers the evaluation of each property that relies on the variable associated to it.

3.2 Method that instruments the source code

A first solution consists in modifying the C source code such that every observed event provokes the notification of its observers. It means that the call to the notification function is automatically inserted in the C code, close to the corresponding statement.

To do that, OSIRIS automatically associates a *subject* `S` with each observed variable `V`, and its observers are the wrappers for the assertions that involve this variable. In addition, a specific function `obs_write_V` is produced, it enables the observation of the events that update `V`. Similarly, a function `obs_read_V` is produced for read actions. Source code instrumentation simply consists in calling function `obs_write_V` or `obs_read_V` everywhere the variable `V` is updated or read.

To observe that a *function* `f` starts or returns, OSIRIS produces a function `obs_f_START` or `obs_f_END`. This function has the same parameters as `f`. It calls `f`, notifies the *subject* associated with `f`, and returns the return value of `f`. Source code instrumentation simply consists in calling function `obs_f_START` or `obs_f_END` in place of `f`.

This method is simple but its disadvantage is that it is intrusive in the source code. Another solution consists in dynamically instrumenting the executable code.

3.3 Method with observing processes

The first method proposed for dynamic binary instrumentation used an additional UNIX process for observation [5]. The principle is to have two processes that are executed in parallel, the first is the targeted application on which we want to evaluate properties (the tracee), and the second is the "observing process" (the tracer). Using two processes permits to preserve the targeted application in the same state that is given to OSIRIS. The ptrace [4] libprary (originally used by gdb [3] for debugging) provides systems calls to define the tracer and tracee processes for instrumentation and memory inspection, it is used to enable the checkers to interact with the targeted application.
To be able to recognize events, the tracer will dynamically modify instructions in the tracee. These instructions are the reads and writes in variables, function calls and return that we want to observe. They are replaced by a special instruction called INT 3 which is a breakpoint. When a breakpoint is reached during the execution, the execution of the tracee process is interrupted and the tracer process is notified. This way the tracer can execute the corresponding property checkers. The breakpoints are differentiated by the value contained in the program counter at breakpoint. The fact that instructions are replaced implies that they are not executed by the tracee. Therefore, the tracer makes the tracee execute the replaced instruction as a single step when it reaches a breakpoint. The detail of the observation mechanism is the following (see fig. 2) :

1. When the tracee execution is interrupted due to INT 3, the tracer resumes.
2. The tracer replaces the breakpoint on which the tracee has stopped by the replaced instruction.
3. The tracer makes the tracee do a single step (executing only the instruction that was replaced at the previous step).
4. The tracer evaluates the property that corresponds to the event encountered.
5. The tracee replaces the breakpoint where it was previously (before the second step).
6. The tracee resumes its execution.

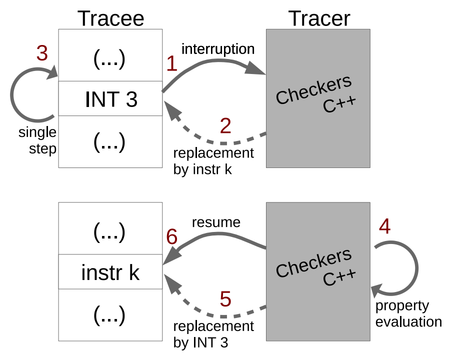


Figure 2: Mechanism using observing process

OSIRIS generates the source code of the tracer process and a new "main" function that creates the two processes of the instrumentation mechanism.
This method has the advantage of not being intrusive because the binary file of the program has not been modified and is instrumented dynamically at runtime. Nevertheless, the use of multiple processes and the ptrace library implies the presence of an Operating System (POSIX) on the system that is executing the instrumented program. We proposed two other instrumentation strategies explained in the following section.

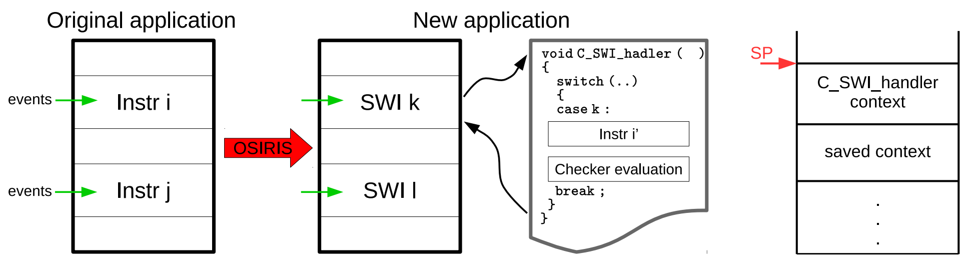
4 Alternative observation methods

To overcome the drawbacks of the methods of section 3, we have designed to new observation techniques.

4.1 Method using software interrupt

This method aims at using software interrupts, in particular for ARM processors. The Software Interrupt (SWI[1]) [1] is a special instruction that emits a signal to the processor which has to change mode to supervisor and to branch to the interrupt vector table. This table is used for exception handling, it contains one word instruction per exception type (usually a branch instruction to the corresponding handling function or "interrupt handler"). SWI allows to program to request privileged operations in supervisor mode, it is commonly used to perform system calls.
The SWI instruction takes a parameter called interrupt number, it can be used to specify the system call number (deprecated in recent ARM ABIs[2]). This number is given as parameter to the interrupt handler (named C_SWI_handler) which can define different behaviors according to this number.

[1]Or SVC for supervisor call.
[2]Application Binary Interface



(a) Interrupt handling using C_SWI_handler (b) Representation of the stack

Figure 3: C_SWI_handler mechanism

Similarly to the solution of section 3.3, we automatically replace instructions in the binary file of the targeted application. Here the instrumentation is done statically by replacing the instructions by a software interrupt SWI (see fig. 3(a)). When the interrupt is triggered, the execution is rerouted to the interrupt handler (pointed by the interrupt vector table) that will execute the replaced instruction, and then call the checker evaluation.
The execution of the replaced instruction requires to save on the stack the state of the program when the exception occurs in order to retrieve it in the handler context (see fig. 3(b)). To do that we have defined a wrapper for the interrupt handler, which saves the content of all the registers on the stack and calls the interrupt handler with two arguments : the interrupt number and the addresses of the copied context on the stack.
The instruction replacement is performed automatically by OSIRIS after finding the events on which the property must be evaluated. Given the replaced instructions, the interrupt handler is also automatically generated.
The evaluation mechanism remains the same as the one of solution of section 3.3 : once the subject is updated, it notifies its observers that triggers the evaluation of the properties by notifying the monitors.
This method has the advantage of not requiring an operating system, that makes it more convenient for some categories of embedded systems. On the other hand, with an operating system, modifying the interrupt vector table and the interrupt handlers could enter into conflict with the system calls, which is not desirable. Hence this solution is well adapted to bare metal contexts.

4.2 Method using jump

This second solution has been proposed to be independent from the presence of an Operating System or not. Instead of using special instructions to trigger the property evaluation we use branch instructions to jump directly to a handling function (called jump handler) used to trigger the property evaluation (see fig 4).

1. When the execution reaches an event that must trigger a property evaluation after its instruction, the program jumps to the jump handler.
2. The properties that involve this event are evaluated.
3. The program resumes after the event.

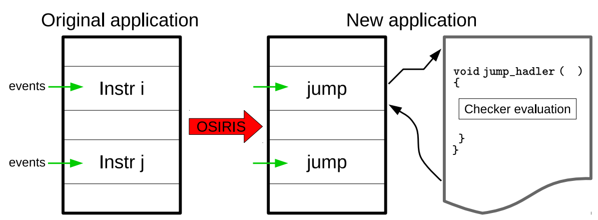


Figure 4: Observation mechanism using jumps

In order to use branch instructions we have to insert instructions in the binary file without modifying all the addresses. To do that we work on binary *relocatable* file, that is the object file before linking. The three main elements that we have to modify in the file (in addition to the instruction insertions) are the *section header table*, the *symbol table* and the *relocation table*.

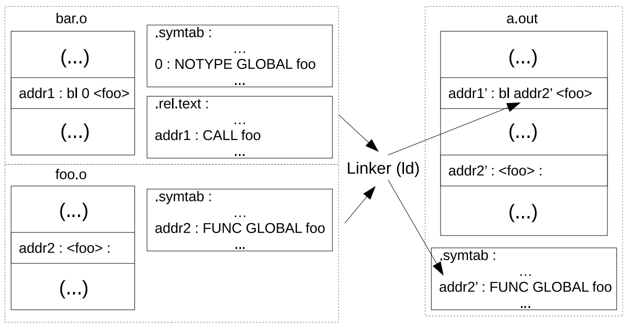


Figure 5: Relocation phase during linking

The section header table (`.symtab`) contains the addresses, size, name and type of every section headers in the file. The symbol table contains identifiers that are visible from external files as function and variables name and addresses (see fig 5). The relocation table (`.rel.text`) contains pointers that have to be resolved during the loading phase, such that it will point to the correct locations.
To insert instruction blocks at a given point in the program, OSIRIS proceeds as follows:

1. Replace the instruction that we want to observe with a branch instruction to a "handler wrapper"
2. Insert the symbol that corresponds to the targeted

	Bare metal (A9)		Linux (A7)		Numbers	
			CPU time = 522 ms			
	source (3.2)	binary (4.2)	source (3.2)	binary (4.2)	activations	evaluations
Property (P_1)	-	-	685 ms	692 ms	597 326	47 929
Property (P_2)	-	-	717 ms	735 ms	597 326	127 326
Property (P_3)	-	-	764 ms	788 ms	1 017 558	77 551
Property (P_4)	-	-	890 ms	911 ms	1 017 558	392 449

Table 1: Experimental results for the simulated annealing application

functions of the inserted branch instruction (foo in fig 5) with address 0 in the symbol table, and modify the relocation table in order to add the inserted instruction to the pointers that have to be resolved (addr1 in `.rel.text`)

3. Generate the corresponding "handler wrapper", in assembly code, that will first execute the replaced instruction and then backup the current state of the program in order to call the jump handler (see Fig. 4).

Moreover, OSIRIS must recognize the replaced instructions. These are data processing and load/store instructions. The parameters that are given to the jump handler are: an integer that specifies the observed event, the current frame pointer, and the value of the R_0 register (the register that contains function return values).

This method has the advantage that it does not require to be in an operating system or bare metal context. Nevertheless it is more invasive for the targeted program because of the modifications done in the program.

5 Experimental results

We now present the results of some experiments on two use cases that are freely available C applications. The first one is an implementation of simulated annealing that comes from the GSL (GNU Scientific Library) [12], and the second one is a path following lateral controller implemented on an autonomous car [15]. Experiments are realized in two contexts: bare metal execution on a Zybo board (ARM Cortex A9 processor), and Linux context on a Raspberry Pi 2 (ARM Cortex A7 processor).

Since the solution of section 4.2 is more general than the one of section 4.1, we only perform experiments with this method, as well as with the source code instrumentation described in section 3.2.

5.1 Simulated annealing

The first use case is a package dedicated to solving optimization problems using simulated annealing. It also provides a test program that uses a simulated annealing algorithm to solve the Travelling Salesman Problem (TSP). Let us briefly recall the principles of simulated annealing. It is based on the computation of an "energy function" E(s) on the system state space. The algorithm first selects a solution s_0 from the state space, then it iterates by visiting neighbouring solutions. When the selected solution `new_E` is worse than the previous one `E`, a probabilistic function decides whether the algorithm shall choose another neighbour or accept the solution and step forward. The final solution is at least a local minimum of the energy function.

We have considered properties that were already specified in [5] (note that all the variables are local variables of a function called `gsl_siman_solve`):

- (P_1) everytime the algorithm finds a new solution with an energy `new_E` lower than the energy of the best solution `best_E`, then `best_E` will be updated with this `new_E`
- (P_2) in addition to the first property, if `best_E` is updated, it will not be modified again (no alteration) until `new_E` is recomputed
- (P_3) everytime the algorithm finds a better solution than the previous one (i.e., `new_E` < `E`), this solution will unconditionnaly get accepted (i.e., the "boltzmann" function will not be called until `new_E` is recomputed)
- (P_4) everytime the algorithm finds a worse solution than the previous one, the "boltzmann" function must be called before determining the next solution.

As an example, we give below the PSL formalization of property (P_4). It means that everytime the local variable `new_E` is updated with a value that is greater than the one of `E`, then the function `boltzmann` will be called before the next assignment to `new_E`:

```
always((gsl_siman_solve.new_E#SET() &&
       (gsl_siman_solve.new_E > gsl_siman_solve.E))
    => next(boltzmann#START()
            before gsl_siman_solve.new_E#SET())));
```

Table 1 summarizes the CPU times without and with instrumentation, for the execution of the TSP for a dozen of towns. For this use case, experiments have not been performed in the bare metal context due to the fact that the GNU Scientific Library cannot be compiled (dependencies to some specific functions are missing). For the Linux context (on the Raspberry Pi), the table first gives the raw CPU time for this application (i.e., without any instrumentation). Then it summarizes the CPU times for the execution with each property, for the source instrumentation and for the binary instrumentation using jumps. The last column gives the number of activations (notifications on events) and of actual evaluations (the

	Bare metal (A9)		Linux (A7)		Numbers	
	CPU time = 21.93 s		CPU time = 4.17 s			
	source (3.2)	binary (4.2)	source (3.2)	binary (4.2)	activations	evaluations
Property (P_1)	21.98 s	21.97 s	4.28 s	5.09 s	7920	1164
Property (P_2)					7920	1266
Property (P_3)	21.97 s	21.97 s	7.1 s	7.96 s	9901	486
Property (P_4)					9901	510
Property (P_5)	21.97 s	21.96 s	4.92 s	5.68 s	13861	1981
Property (P_6)					5941	1981

Table 2: Experimental results for the path controller

checker is notified and the premise of the implication holds i.e., an evaluation actually starts) of each property. We can remark that the CPU time overhead is rather comparable for the source code instrumentation and for the binary instrumentation, and that it is roughly around 40%.

5.2 Path following lateral controller

This use case has also already been described in [5]. It is a path following lateral controller implemented on an autonomous car [15]. The idea is that the car has some desired path to follow, and sensors below the car detect the location of this path (see Figure 6).

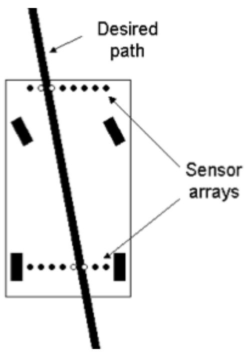


Figure 6: Sensors on the autonomous car [15]

First, the values captured by the sensors are analyzed by a DSP (Digital Signal Processor) which ultimately sends an `angle` to the controller, such that the controller can determine how to turn the steering wheels in order to follow the correct path. The DSP first computes the `error` between the desired path and the car, then it computes the expected `angle`, copies it to a variable `output` that is used to communicate with the controller. To send the value of this variable to the controller, the DSP uses a fonction `putMem`. In some other phases of its algorithm, it computes and sends a new velocity, instead of an angle.

The following correctness properties have been considered:

- (P_1) if the front sensors detect the path on the left, then the next assignment to `output` will be such that a correction is applied i.e., the difference between the previous value of `output` and the current one is greater than 0,
- (P_2) symmetrically, if the front sensors detect the path on the right, then the next assignment to `output` will be such that a correction is applied i.e., the difference between the previous value of `output` and the current one is lesser than 0,
- (P_3) if the front sensors detect the path on the left and then do not detect the path, then the next assignments to `output` will be such that a correction is applied until the front sensors detect the path again,
- (P_4) symmetrically, if the front sensors detect the path on the right and then do not detect the path, then the next assignments to `output` will be such that a correction is applied until the front sensors detect the path again.

Property ($P1$) for instance is formalized as follows, it uses the PSL "Modeling layer".

```
// Modeling layer:
  bool back_blackout;
  bool front_left=false;
  long zero_angle=32;
  long delta = 0;
  long prev_angle = 32;
  if(front_array#SET()){                  // path on
    if ((front_array & 0xF80) != 0xF80)  // the left
      front_left=true;
    else
      front_left=false;
  }
  if (back_array#SET()){
      back_blackout = back_array == 0xFFF;
  }
  if (output#SET() && ((output & 0x40) == 0)){
    // i.e., if output receives a new angle
      delta = prev_angle - (output & 0x3f);
      prev_angle = output & 0x3f;
  }
// Assertion:
  assert
     always((!back_blackout && front_left)
           =>next_event!(output#SET())(delta >= 0));
```

Property ($P3$) expresses that the correction is applied *until* the path is detected again, it has the following PSL formalization:

```
// Modeling layer:
  bool front_to_left;
  bool front_blackout;
  bool back_blackout;
  long delta = 0;
  long prev_angle = 32;
  if (front_array#SET()) {
     if ((front_array & 0xF80) != 0xF80){  // path on
        front_to_left = true;              // the left
        front_blackout = false;
     }
     else if (front_array == 0xFFF){  // no path
         front_blackout = true;
         front_to_left = false;
         }
         else {
            front_to_left = false;
            front_blackout = false;
          }
      }
  if (output#SET() && ((output & 0x40) == 0)){
      delta = prev_angle - (output & 0x3f);
      prev_angle = output & 0x3f;
  }
  if (back_array#SET()){
     back_blackout = back_array == 0xFFF;
  }
// Assertion:
assert
  always
    ((front_array#SET() &&
      front_to_left && !back_blackout) =>
     (next (next_event!
             (front_array#SET())
             ((front_blackout &&
               !back_blackout) =>
                  (next ((angle#SET() -> (delta >= 0))
                        until! (front_array#SET()
                               && !front_blackout)))));
```

Regarding fault tolerance (e.g. w.r.t. electromagnetic perturbations), it is also important to check that, once an appropriate value is computed for the angle, it remains unchanged until it is sent to the microcontroller. To that goal we have introduced the complementary properties (P_5) and (P_6) below:

- (P_5) every last assignment to `angle` before an assignment to `output` actually corresponds to the value that is put into `output`,
- (P_6) when the algorithm is computing a new `angle`, then the value assigned to `output` indicates that it corresponds to an angle, and the next call to `putMem` actually receives this value as parameter.

To check the relevance of these properties, we have performed various fault injections. They inject faults (bit flips or stuck-at) either in least significant bits which take part in the value of the angle, or in most significant bits which represents some flags. One of these flags indicates whether the value is an angle or a velocity. Another flag should always be true to indicate that the car is enabled. Here is the summary of the results of these fault injections:

- Fault injection in the value of angle:
  - ($P5$) is violated (because it deals with `angle`)
  - ($P6$) is not violated (because it only deals with `output`)
- Fault injection in the angle/velocity flag of angle:
  - ($P5$) is violated (because it deals with `angle`)
  - ($P6$) is violated (because the value seems to be a velocity)
- Fault injection in the enabled flag of angle:
  - ($P5$) is not violated (because this bit is ignored)
  - ($P6$) is not violated (because this bit is overwritten when output is assigned)

Table 2 summarizes the CPU times without and with instrumentation, for the execution of this application. For the bare metal and the Linux contexts, the table first gives the raw CPU time (i.e., without any instrumentation). Then it summarizes the CPU times for the execution with each property, for the source instrumentation and for the binary instrumentation using jumps. The last column gives the number of activations and of actual evaluations of each property. We can remark that the CPU time overhead is comparable for the source code instrumentation and for the binary instrumentation, and that it is generally negligible in the bare metal context.

6 Related Work

C. Watterson and D. Heffernan present in their paper [18] different approaches for monitoring embedded systems. The first approach is to use *hardware monitoring* that minimize intrusion in the target software. There is also *software monitoring* that can be source code instrumentation, modifications of the operating system, or separate process monitor. The later approach may have the disadvantage to slow down the target application. Then there is the combination of the two first approaches to get *hybrid monitors* that could minimize the disadvantages of the two first methods. In addition the observation could be performed by a remote hybrid monitor called an on-chip monitors. For our project we decided to use software monitors to be able to automatize the monitor generation for embedded software.

6.1 Event-triggered monitoring

Thaker [17] proposed an assertion-based verification solution for Java programs that uses aspect oriented programming for instrumenting the source code of the target application. They used Linear Temporal Logic (LTL) [14] that involves events (that they called interest points) and values of variables to specify properties that will be translated into assertions. These interest points are global variables reads and writes and method begins and ends. The observation mechanism of OSIRIS also consider this kind of events but they are extended to the use of the local variables and parameters of functions.

Most solutions suffer from the lack of quantitative oriented verification. This is the point of view of Drusinsky [8, 9] with the Temporal Rover that uses a Linear-time Temporal Logic (LTL) extended with Metric Temporal Logic (MTL) assertion-based verification. In addition to that it enables to use specification of time-series constraints. This means that the properties are specified over constraints on modifications of variables over time. This tool takes as input source code of C, C++, Java, VHDL or Verilog program/model with properties written within comments, and returns the instrumented source code. The instrumentation works using an external software that runs on a host machine and gets the instrumentation information sent by the target program through HTTP, socket or serial port. The logic model is quite close to our PSL logic model by the fact that properties involve changes of values and event triggered over the whole execution. Nevertheless the observation mechanism is dependant to an external machine and is not well-suited for embedded software.

The RMOR framework developed by the Jet Propulsion Laboratory [13] is also an event-triggered verification tool, but it does not use temporal properties as input. It generates monitors for properties specified as Finite State Machines (FSM). These FSM have to be defined in such a way that they correspond the expected behavior of the targeted application. The inputs of the FSMs are temporal events declared with the FSMs. These monitors are translated to CIL (C Intermediate Language) an aspect oriented programming language that is used to instrument the source code.

6.2 Some other approaches

The RiTHM tool [16] takes another direction than Event-Triggered Runtime Verification (ETRV) and takes the Time-Triggered Runtime Verification (TTRV) model. In this model, the monitors are invoked periodically based on a predefined frequency. The tool uses LTL properties to instrument C program source code that will be evaluated in between polling time. Polling is the mechanism where the process waits for a device to be available, here the process waits for the timer (that defines the period) to trigger. The property verification is handled either by the GPU, or by another CPU core. It is then not very adapted for embedded systems that does not support multiprocessing. This solution has fixed time overhead (because of the fact that the monitors are called periodically), and it is more time-consuming than with ETRV. Either the polling time is short and the precision over property evaluation is good but the latency is big or, the polling time is longer and the evaluation quality decreases.

Cheung and Forin [6] give a C-Language Binding for PSL that enables to do simulation based verification of properties specified in PSL. It uses a hardware/software simulator to perform analysis of the events involved in the properties that are specified. A verification unit (vunit) must be defined to set where and when the property has to be verified. Vunit is also used to set the block in which the variables and events can be found. In OSIRIS, property specification is not limited to blocks and the scope is the whole program. We can define a property that will be verified on the entire application including the use of local variables and function parameters (using the $f.v$ notation).

7 Conclusions

The four proposed observation methods are complementary. The source code instrumentation is simple and efficient, but it is intrusive. The "observing process" method is less invasive because it does not require any modification in the binary file but it is only suitable in an Operating System context. The version that uses software interrupts is not much invasive and is designed only for bare metal systems. And the fourth solution works whatever the context is (OS or bare metal) but is more intrusive than the one that uses software interrupts.

The methods described in section 4 have been implemented in OSIRIS, and checked on few examples. It required to mechanize static binary instrumentation, generation of C source code (for the interrupt and jump handlers), and generation of assembly code (for the jump handler solution, in order to mimic a kind of interrupt mechanism).

References

[1] *SWI, ARM Developer Suite, Assember guide*, 2000 - 2001.

[2] IEEE Standard for Property Specification Language (PSL). *IEEE Std 1850-2010 (Revision of IEEE Std 1850-2005)*, pages 1–182, April 2010.

[3] *gdb, http://www.man7.org/linux/man-pages/man1/gdb.1.html*, 2018.

[4] *ptrace, http://man7.org/linux/man-pages/man2/ptrace.2.html*, 2018.

[5] Martial Chabot, Kevin Mazet, and Laurence Pierre. Automatic and Configurable Instrumentation of C Programs with Temporal Assertion Checkers. In *Proc. MEMOCODE*, September 2015.

[6] Cheung, Ping Hang and Forin, Alessandro. A C-Language Binding for PSL. In Lee, Yann-Hang and Kim, Heung-Nam and Kim, Jong and Park, Yong-wan and Yang, Laurence T. and Kim, Sung Won, editor, *Embedded Software and Systems*, pages 584–591, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[7] Clarke, Edmund and Grumberg, Orna and Peled, Doron. *Model Checking.* MIT Press, Cambridge, MA, USA, 1999.

[8] Doron Drusinsky. The Temporal Rover and the ATG Rover. In *Proc. International SPIN Workshop.* Springer-Verlag (LNCS 1885), 2000.

[9] Doron Drusinsky and Man-Tak Shing. Monitoring temporal logic specifications combined with time series constraints. 2003.

[10] L. Ferro and L. Pierre. ISIS: Runtime verification of TLM platforms. In *2009 Forum on Specification Design Languages (FDL)*, pages 1–6, Sept 2009.

[11] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Abstraction and Reuse of Object-Oriented Design*, pages 701–717. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002.

[12] GNU Scientific Library. http://www.gnu.org/software/gsl/.

[13] Klaus Havelund. Runtime Verification of C Programs. In *Proc. TestCom'2008*. Springer-Verlag (LNCS 5047), 2008.

[14] Zohar Manna and Amir Pnueli. *The temporal logic of reactive and concurrent systems: Specification.* Springer Science & Business Media, 2012.

[15] Patricia Mellodge. Feedback control for a path following robotic car. Master's thesis, Virginia Tech, 2002.

[16] Samaneh Navabpour, Yogi Joshi, Chun Wah Wallace, Shay Berkovich, Ramy Medhat, Borzoo Bonakdarpour, and Sebastian Fischmeister. RiTHM: A Tool for Enabling Time-triggered Runtime Verification for C Programs. In *Proc. FSE*, St. Petersburg, Russia, 2013.

[17] Sahil Thaker. Runtime monitoring temporal property specification through code assertions. *Department of Computer Science, University of Texas at Austin*, 2005.

[18] C. Watterson and D. Heffernan. Runtime verification and monitoring of embedded systems. *IET Software*, 1(5):172–179, October 2007.

Combining Path and Cache Analysis for WCET estimation improvement

Florian Barrois
Université Grenoble Alpes
Grenoble, France
florian.barrois@etu.univ-grenoble-alpes.fr

Supervised by : Valentin Touzeau, Catherine Parent-Vigouroux, Claire Maïza
Laboratory VERIMAG, Grenoble, France

I understand what plagiarism entails and I declare that this report is my own, original work.
Name, date and signature: Florian Barrois [illegible]

Abstract

The Worst Case Execution Time (WCET) of a program is an important parameter in the field of hard real-time systems. A static path analysis shows that its estimation can be improved by using Linear Relation Analysis to consider programs semantics in addition to programs structure. Furthermore, some static survey about cache memory was conducted in the scope of determining whether cache memory accesses of a program would result in a cache hit or a cache miss. The idea is to add a model checking step to the use of abstract interpretation to refine more block accesses as hit/miss.

Our proposal consists in exploiting programs semantics, and more precisely infeasible paths, to refine the state of the art about the cache memory accesses of programs. By injecting the infeasible paths found into a model checker analysis, we may find out that some undefined block accesses are always a cache hit/miss. We propose two methods for incorporating infeasible paths into the study: modifying the program Control Flow Graph or expressing these paths as constraints using temporal logic. The knowledge newly acquired on the cache memory accesses then allows to enhance the WCET static estimation either by obtaining a better value for it, or by gaining confidence on its actual value.

Key words: WCET, infeasible path, cache memory, control flow graph

1 Introduction

As the technology expands, technical devices become ubiquitous and the need of assurance on the systems behaviour gets stronger. In particular, the safety aspect requires a specific attention in the case of critical and hard real-time systems. For instance, in critical devices such as an aircraft landing gear or a car airbag, it is essential to ascertain that the mechanism deploys on time.

The Worst Case Execution Time (WCET) refers to the greatest amount of time needed for a task to execute on a specific physical platform. Therefore, finding an upper bound of the WCET allows to guarantee the safety requirements of a system. Estimating it is not a simple task since it depends on the hardware platform and the executed program. Usually, measuring the program's execution time on representative inputs and adding a safety margin may suffice, but in safety-critical systems, one may wish for a higher degree of assurance and use static analysis to cover all cases.

The programs meaning is generally not considered when estimating the WCET. A study conducted at the laboratory VERIMAG of Grenoble [7] presents a way of statically obtaining a WCET estimation by taking into account both the program structure and semantics. From this information, the WCET value is found considering operations cost but also the memory access cost. This last factor thus leads to the question of knowing whether the desired information remains in the cache memory.

Cache memories are small memories very close to the processor which allow faster accesses than the main memory. Caches store a copy of each accessed memory block. Thus, when accessing a block, the cache is examined to determine whether the block is present in the cache (*cache hit*) or not (*cache miss*). Memory blocks are placed in cache parts called *cache lines*. In order to guarantee an efficient look-up, each block can be only stored in a certain number of different cache lines called a *cache set*. The number of blocks in a cache set is named the *cache associativity*. As the cache is much smaller than the main memory, a *replacement policy* must be chosen to decide which cache block is to be evicted when the cache has no more space to store the last accessed block. In this paper, we focus on instruction caches using the commonly used Least Recently Used (LRU) policy.

Ideally, in the scope of precising the WCET estimation of a program, one could proceed to a static cache analysis that would determine whether each block access of the program will result in a cache hit or a cache miss. Unfortunately no analysis can perfectly give such information. Thus, most cache analyses use a sound but incomplete technique of analysis called *abstract interpretation* [4] to evaluate the classification of block accesses. It has been shown that some memory block accesses left unclassified by abstract interpretation could be refined as hit or miss by model checking [10]. However, this survey ignores the semantic aspect of the program.

Our aim, in this article, is to supply this refined cache analysis with the programs semantics by highlighting some infeasible execution paths found from the mentioned study about the WCET estimation and exploiting them to categorize as cache hit or cache miss some cache accesses of kind still unknown. Depending on the input program, the benefit of this survey may vary from reducing the uncertainty of the original WCET estimation to obtaining a better one.

2 Background and related work

2.1 Input data representation: Control Flow Graphs

Programs are exploited in their control flow graph (CFG) form, where vertices correspond to the instructions blocks and edges model transitions of the program (conditions, loops, jumps...). Throughout the survey, we base findings upon many kinds of control flow graphs detailed below.

Binary CFG This scheme (like in Figure 1a) represents the program structure obtained after the compilation step. *Basic blocks* (BB) correspond to sets of binary instructions that are executed. The binary CFG remains very close to the CFG of the original C program besides some reorganizations due to compiler optimizations (*traceability* [7]).

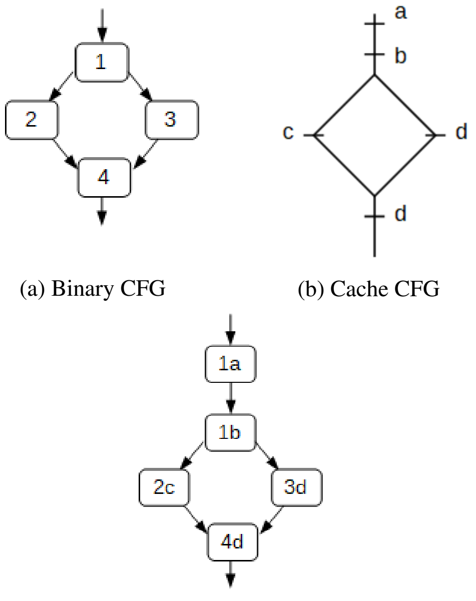
Cache CFG Accesses to cache memory blocks are displayed in all possible execution paths and labeled with a letter representing the block they refer to, as shown in Figure 1b. Memory blocks may be accessed several times.

Memory CFG In our approach we need to consider basic blocks (BB) as well as cache block accesses. Memory CFGs gather both information and reveal which cache blocks are accessed in each basic block. We associate one memory CFG node per unique couple basic block/cache block. Basic and cache blocks can be identified respectively by numbers and letters so that the memory representation of the program appears clearer, as can be seen in Figure 1c.

2.2 Path analysis

The existing path analysis we rely on is based upon the use of the following tools and techniques.

WCET estimation with OTAWA
OTAWA [1] is an academic framework dedicated to gathering and comparing many methods related to WCET analysis. The implicit path enumeration technique (IPET [2]) is the most common approach to estimate the WCET. By applying this method to the binary CFG of a program, OTAWA is able to



(a) Binary CFG (b) Cache CFG

(c) Memory CFG

Figure 1: Graph examples on if-then-else structure

establish what is called *flow constraints*, i.e. invariant structural properties about the program execution, expressed in the form of equations and inequalities involving the basic blocks and the edges of the CFG. We then solve the system of equations using a Linear Programming solver in order to obtain an overestimation of the WCET.

The initial C program (Figure 2a) is first pictured as its binary CFG (Figure 2b) before being analyzed by OTAWA. In Figure 2c, a variable x<*nb*> corresponds to the number of times the basic block of the binary CFG identified by *nb* is traveled.

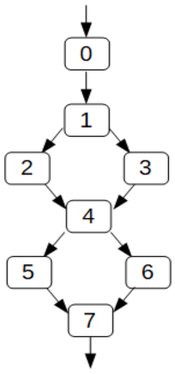
Program instrumentation with counters
The idea for injecting programs semantics into the WCET analysis is to instrument C programs with counters covering all the possible execution paths. Variables are initialized to zero and each of them is incremented in a different CFG block. This code arrangement can then be used for a static inspection with program analyzers like PAGAI [6]. This tool is a static analyzer that aims to generate numerical invariants of a program using abstract interpretation techniques such that Linear Relation Analysis (LRA) [5].

In our case, this technique is employed in order to predict the value of the variables throughout the program. Since each variable equals the number of times a path would be taken during the execution, this process highlights the infeasible paths which are the semantic element exploited in the WCET estimation refinement.

```
if (a > 0){
     a = -2 * a;
     b = a + b;
}
else
     a = a + 4;

if (a < 0)
     a = 60;
else{
     b = b * a;
     a = a + 3;
}
```

(a) C program



(b) Binary CFG

```
x0 = 1
x0 = x1
x1 = x2 + x3
x4 = x2 + x3
x4 = x5 + x6
x7 = x5 + x6
```

(c) Flow constraints

Figure 2: Establishing flow constraints with OTAWA

```
int cptr1 = 0;
int cptr2 = 0;
int cptr3 = 0;
int cptr4 = 0;
int cptr5 = 0;
int cptr6 = 0;
cptr1++;
if (a > 0){
     cptr2++;
     a = -2 * a;
     b = a + b;
}
else
     cptr3++;
     a = a + 4;

if (a < 0)
     cptr4++;
     a = 60;
else{
     cptr5++;
     b = b * a;
     a = a + 3;
cptr6++;
}
```

(a) instrumented C program

```
cptr2 + cptr5 <= 1;
cptr4 +cptr5 = 1;
cptr3 +cptr2 = 1;
cptr3 <= 1;
cptr4 <= 1;
```

(b) counter constraints

Figure 3: Finding counter constraints from instrumented C program

In Figure 3b, many constraints correspond to structural constraints found by OTAWA and thus do not bring any additional information. However, the first invariant implies that no execution can pass both through the block that increments cptr2 and the one where cptr5 is incremented since this case would give a result higher than one for the sum of both counters. This invariant is semantically equivalent to stating that the path composed of the if branch of the first condition and the else branch of the second one is an infeasible path (Figure 3a)[1].

Taking advantage of infeasible paths in WCET computation

The structural formulas of OTAWA associated to the numerical constraints found by PAGAI may result in a tighter overestimation i.e. a better estimation of the WCET. In the previous example (Figure 2a), the first if and the second else branches are more costly in terms of execution time due to operation types (multiplication). Consequently, they form the longest execution path of the program. Knowing the unreachability of this path thus leads to estimate the WCET with paths requesting cheaper operations which reduces the final value obtained as can be seen on Figure 4.

```
Estimation of WCET without PAGAI : 118 cycles
Estimation of WCET with PAGAI : 100 cycles
Gain : 15.25%
```

Figure 4: WCET improvement with PAGAI on Figure 2a example

2.3 Cache analysis

May and Must analyses

Most static cache analyses are realized using abstract interpretation [4] to categorize memory blocks as cache hit or cache miss. By this means, one may find that, independently from the program input, some accesses always result in the same cache classification. The so-called *may* and *must* analyses [8] respectively establish that a memory access is an *Always Miss* (AM) or an *Always Hit* (AH). The analysis also shows if a block access is a miss in some cases and a hit in some others. Hence these accesses are classified *Definitely Unknown* (DU). Finally, there remains blocks that cannot be placed in any of the three mentioned classes after running may and must analyses and that are left *Unknown*[2].

[1]Here the infeasible path is trivially visible to illustrate our words. The benefit of this study appears more clearly on more realistic programs.

[2]In some cases, we can find out that there exists cache hits (resp. misses) but we have a lack of information about the other possible executions.

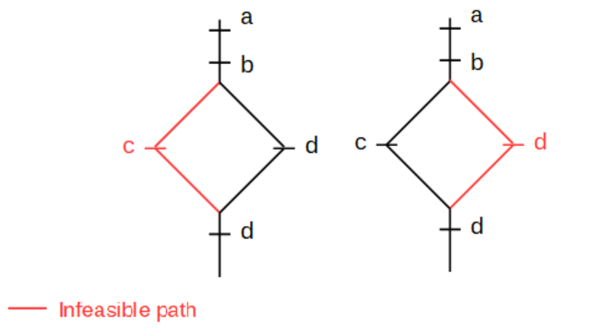
Adding model checking to abstract interpretation
The model checking method consists in giving logical propositions to a model checker, together with program and cache models, that will respond true if and only if the proposition is always true, and false otherwise. While this method allows refining block accesses more precisely, it remains more costly in time than abstract interpretation. For this reason, the latter is first employed and the use of the model checker is kept for the blocks left Unknown. One can then choose an expression semantically equivalent to an Always Hit or to an Always Miss to give to the model checker and deduce the belonging of the target block to one of the three classes.

3 Exploiting infeasible paths knowledge for cache accesses refinement

Finding infeasible paths brings us information about the program that reduces the domain of the possibilities as for the cache accesses classification. Indeed, since infeasible paths avoid some memory blocks from being accessed; the blocks used in the program that were found Definitely Unknown may be refined either as Always Hit or Always Miss.

For instance, in Figure 5a, the left execution path is found infeasible, thus the right branch is always taken and the block d is accessed and cached. Consequently, when reaching the second access to the block d, it is necessarily in the cache. This access can now be classified as Always Hit instead of Definitely Unknown.

Similarly the Figure 5b shows the refinement of the last memory block access from Definitely Unknown to Always Miss.



(a) DU-to-AH refinement (b) DU-to-AM refinement

Figure 5: Graph structures for memory block refinement

In our study, we thus wish to obtain such cache configurations to improve our classification of accesses. Once reached, the memory CFG corresponding to this cache disposition can be transmitted to the model checker. The objective then consists in incorporating someway the infeasible paths into the model checker analysis in order to find out some block accesses refinements. Thereafter the logical proposition expressing an AH/AM may be passed as input and become correct, what can be translated into the assurance that the target access has turned from Definitely Unknown into AH/AM. Note that there also exists cases where the infeasible paths found do not change any memory access categorization.

3.1 Playing on the input memory CFG

An idea for simulating infeasible paths is to modify the CFG structure given to the model checker. Instead of the whole memory CFG, we can provide only the subgraphs corresponding to each realizable path.

By this means, the analysis is able to conclude on the expected results, i.e. the change of classification of the studied memory block access. However, the number of paths is exponential in the number of conditions in the program, what involves a massive input model. Moreover, since these subgraphs are analyzed separately, all the memory blocks common to all the possible paths are re-examined in each subgraph although some of them are guaranteed not to be refined differently with the infeasible paths found, due to the program structure and the cache associativity.

The costly use of the model checker to analyze partly redundant data thus makes this approach relatively inefficient.

3.2 Working with temporal logic

Another solution is to express infeasible paths directly as constraints for the model checker. This is possible by using an extended logic form called *temporal logic* [3], that includes temporal operators in addition to the ones existing in the classical logic. Let us consider an example to understand how this is used.

Figure 6 shows the memory CFG of a C program containing two if/then/else conditions with its infeasible path colored in red.

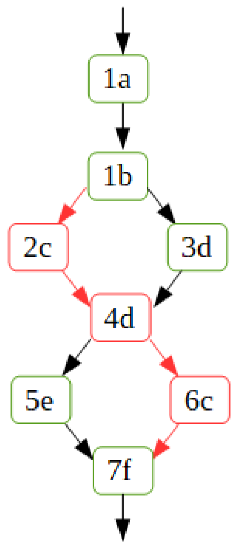


Figure 6: Example of memory CFG with infeasible path

Let us assume that we want to check if the block 6c can be refined as Always Miss. The logical proposition to give to the model checker is

6c => *cache miss*

and means that when reaching the block 6c, the memory block accessed is not in the cache. Without regard to the infeasible paths, the block 6c is categorized as Definitely Unknown since there exists an execution path for which the memory block c is previously cached and another one where it is not[3]. Consequently, the model checker would evaluate the previous logical proposition to false.

Modeling infeasible paths with temporal operators

Temporal operators allow to express properties involving some of the blocks traveled along the possible paths during the execution. We focus on four temporal operators to establish these properties:
- X(p): the property p is true at all the states following the current state;
- F(p): the property p will be true at some future state of the current state (i.e. any of the states accessible from the current state in one or several steps);
- G(p): the property p is true at all future states of the current state;
- p U q : the property q will be true at some future state, and the property p is true in the current state and will be true at least until q becomes true.

The infeasible path of our example can be incorporated to the study thanks to the temporal logic, by modeling it with the blocks it is composed of, what gives a formula of the form

((state = 6c) => *cache miss*) OR (the execution passes through the state 2c).

The associated temporal formula is

G((state = 6c) => *cache miss*) OR F(state = 2c).[4]

The objective is to assess the following statement: "Either the access to the block 6c causes a cache miss, or the execution path taken is infeasible".

In this example, the model checker would evaluate this assertion to true, which entails that the block 6c can be classified as Always Miss.

This method presents the advantage of avoiding an intensive use of the model checker while keeping the memory CFG we give it in its complete form. Furthermore, the above formula can easily be generalized. Indeed, employing several logical ORs allows to cover programs that comprise many infeasible paths, while the unrealizable paths involving more than two blocks can be treated using logical conjunctions in the dedicated part of the formula.

[3]We consider that the memory block c has not yet been evicted from the cache when reaching the block 6c, i.e. the cache associativity is higher than 1.

[4]Note that the current state is the initial state of the graph.

It can be noticed that this memory CFG of Figure 6 has many non-adjacent blocks accessing the cache block c. Given that cache blocks gather contiguous memory addresses, the corresponding C program necessarily contains memory jumps. In order to consider a larger panel of programs in our survey, we have examined the case of function calls.

Single calls to many functions

Function calls slightly modify memory CFGs. Indeed, the block containing the call is divided in two blocks that we will call the calling block and the return block. Function calls are then represented simply by adding two edges: one from the calling block to the first block of the function called, and the other one from the last block of the function to the return block of the calling function.

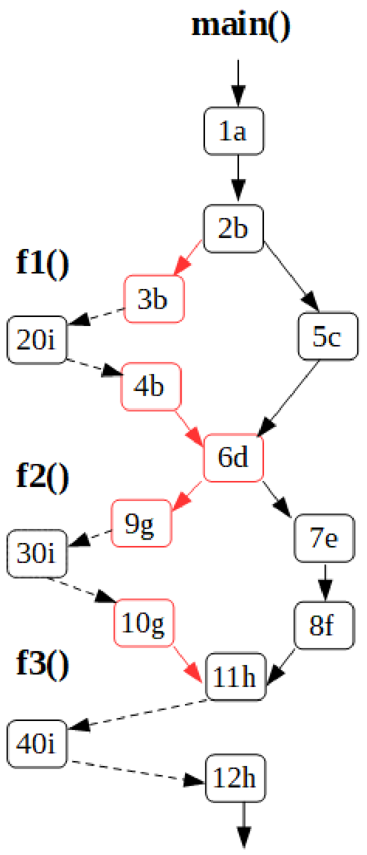


Figure 7: Memory CFG with single calls to many functions stored in a same cache block

In the case of single calls to many functions all stored in different cache blocks, all the accesses to the functions remain Always Misses. The benefit of our study is thus more visible on programs gathering many functions located in a same cache block. Figure 7 shows the memory CFG of the C program joined in Appendix I.

The main function makes calls to three other functions, all composed of a single memory block accessing the cache block i. Let the block 30i the one we wish to refine. Although the memory CFG looks a little different from the ones showing a unique function, the same form of logical formula is able to express the infeasible executions: the proposition

G((state = 30i) => *cache miss*) OR F(state = 20i)

proves to be correct and the block 30i may be refined as Always Miss.

The block 40i, however, cannot be refined as AH nor as AM since reaching the call of one of the two other functions remains possible and makes the last function call capable of inducing either a cache hit or a cache miss.

Many calls to a same function
The programs containing many calls to a same function are harder to study. Indeed, the first block of the function possesses as many input edges as the function is called, and similarly as for the last block of the function. This creates loop structures that are considered feasible by the model checker, which might skew the analysis results.

For example on Figure 8, the different calls make the following loop structure: 8f, 20k, 21k, 4c, 7e, 8f. As a result, if we consider a cache memory of size four, this path leads to a cache hit when accessing the block 20k, which makes the model checker classify it as Definitely Unknown, while the actual program semantics would lead to refine it as Always Miss. The logical formula to analyze via model checking thus needs to prevent these "fake loops" from being considered realizable.

Our solution to do this is to force the right *calling block-return block* associations by following this reasoning: when calling a function, after the first block of this function has been traveled, it will not be traveled again until the proper return block has been reached.

For instance, the logical proposition modeling the loop created by the first function call is as follows:

G((state = 3c) => X(X(not(state = 20k) U (state = 4c)))).

This means that if the current block is 3c, then once the execution arrived to the block located two steps ahead, that is to say the block following the block 20k, this block 20k will not be attained until the return block has been executed.

This forces the right path to be followed since the Until temporal operator states that the condition must be reached at some point during the execution.

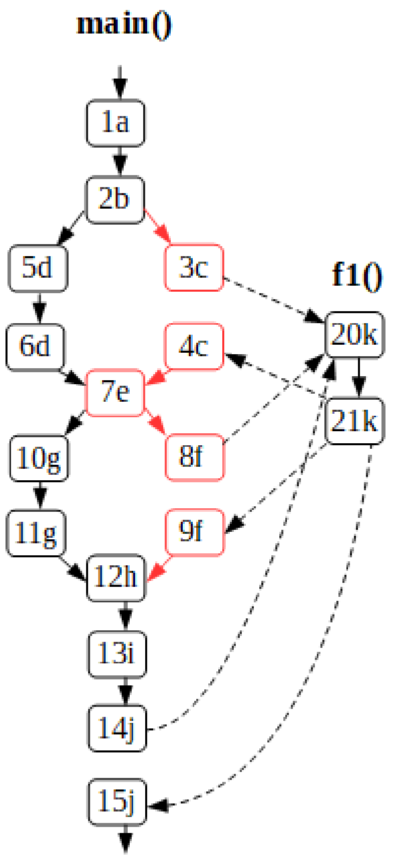


Figure 8: Memory CFG with many calls to a single function corresponding to the C program in Appendix II

Since the formula to give to the model checker is the disjunction of the refinement expression with all the paths that are infeasible, the previous logical proposition must be included in its negative form. As regards example of Figure 8, the formula to be assessed to refine the block 20k must include the three loops created by the function calls in addition to the infeasible path, what gives

G(state = 20k => *cache miss*)
OR
F(state = 3c) AND F(state = 8f)
OR
not(G((state = 3c) => X(X(not(state = 20k) U (state = 4c)))
OR
not(G((state = 8f) => X(X(not(state = 20k) U (state = 9f)))
OR
not(G((state = 14j) => X(X(not(state = 20k) U (state = 15j))).

Nevertheless, if this formula generally solves the loop problem coming from the function calls, it does not allow to cover to case of the calls to recursive functions, due to the Until condition.

3.3 Application to the WCET computation

The memory accesses that we have been able to refine may lead to enhance the original WCET static analysis. Indeed, since this one considers all worst possible values for memory accesses, any *Definitely Unknown-to-Always Hit* transformation of a block access located on the longest execution path is likely[5] to result in a reduction of the WCET because some accesses to the main memory are avoided.

On the other hand, finding Always Misses means that we need to access the main memory so *Definitely Unknown-to-Always Miss* refinements usually do not help to lower the WCET computed. However, some accesses that were considered by default as misses to keep a sound estimation are now proved to be misses. In that way, our study makes the WCET analysis gain in precision.

4 Analysis of an existing program

The method to incorporate infeasible paths into the model checking phase found along this survey have been tested on a realistic program taken from an available benchmark[6].

The C code analyzed is a fourty-line long function composed of nested if/then/else and switch/case structures. Unfortunately, although this function contains many infeasible paths, it is not sufficient to refine any of the memory accesses. However, this analysis has showed that, despite its shortness, its nested conditions make it hard to analyze.

5 Conclusion and future work

Our survey shows that the semantic aspect of programs can be used to refine our knowledge of the access type to the cache memory. This provides a significant impact in particular on WCET static analysis as long as at least one memory access can benefit from it. The first technique employed to simulate infeasible paths was sufficient to establish that access refinement was possible, but it appears hardly adapted to realistic programs since it is very program-size-dependent.

The temporal logic allows further researches, however we still have not found any way to work on some main programming statements like loops and recursive functions. Furthermore, the tests realized on an existing program suggest to make our analysis automatized in order to be capable of considering more complex programs.

A possible perspective of work from this survey, in particular in the case of loops, is the study of the persistence of memory accesses, that is to say the ability for a memory access to always cause the same kind of cache access after the first access to this cache block.

[5]except for timing anomalies [9]

[6]https://github.com/tacle/tacle-bench/blob/master/bench/sequential/statemate/statemate.c

References

[1] Ballabriga, C., Cassé, H., Rochange, C., Sainrat, P.: OTAWA: An open toolbox for adaptive WCET analysis. In: SEUS (2010)

[2] Ballabriga, C., Cassé, H.: Improving the WCET computation time by IPET using control flow graph partitioning. In: WCET (2008)

[3] Banieqbal, B., Barringer, H., Pnueli, A. (eds.): Temporal Logic in Specification, Altrincham, UK, April 8-10, 1987, Proceedings, *Lecture Notes in Computer Science*, vol. 398. Springer (1989). DOI 10.1007/3-540-51803-7. URL https://doi.org/10.1007/3-540-51803-7

[4] Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: 4th ACM Symposium on Principles of Programming Languages, POPL'77. Los Angeles (1977)

[5] Halbwachs, N., Proy, Y., Roumanoff, P.: Verification of real-time systems using linear relation analysis. Formal Methods in System Design **11**(2), 157–185 (1997)

[6] Henry, J., Monniaux, D., Moy, M.: Pagai: A path sensitive static analyser. Electr. Notes Theor. Comput. Sci. **289**, 15–25 (2012)

[7] Raymond, P., Maiza, C., Parent-Vigouroux, C., Jahier, E., Nicolas, H., Carrier, F., Asavoae, M., Boutonnet, R.: Improving wcet evualuation using linear relation analysis. To be published (2018)

[8] Reineke, J.: Caches in WCET analysis: Predictability - competitiveness - sensitivity. Ph.D. thesis, Saarland University (2009)

[9] Reineke, J., Sen, R.: Sound and efficient WCET analysis in the presence of timing anomalies. In: N. Holsti (ed.) 9th Intl. Workshop on Worst-Case Execution Time Analysis, WCET 2009, Dublin, Ireland, July 1-3, 2009, *OASICS*, vol. 10. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany (2009). URL http://drops.dagstuhl.de/opus/volltexte/2009/2289

[10] Touzeau, V., Maïza, C., Monniaux, D., Reineke, J.: Ascertaining uncertainty for efficient exact cache analysis. In: R. Majumdar, V. Kuncak (eds.) Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II, *Lecture Notes in Computer Science*, vol. 10427, pp. 22–40. Springer (2017). DOI 10.1007/978-3-319-63390-9_2. URL https://doi.org/10.1007/978-3-319-63390-9_2

Appendices

Appendix I

```
void f1();
void f2();
void f3();

int main(int argc, char* argv[]){
  int a = 3;
  int b = 5;
  int c;

  a = a + 2;


  if (c != 3){
    f1();
    a = a + 1;
    b = b + 1;
    c = c + 1;
  }

  else{
    c = c - 3;
    a = a + 4;
    a = 1651768716*a;
    b = 5*(b*8);
  }

  a = *argv[1];
  b = 10;
  b = b + 5;
  a = a + 1;


  if (c == 0){
    a = a + 1;
    b = b + 1;
    c = c + 1;
    f2();
    a = *argv[1];
  }

  else{
    a = a * 1487+(96874+a*b);
    b = b + 654*(a+c*8576);
    c = 1;
  }

  f3();
  a = a + 3;
  b = b + 3;
  b = 1;
  c = c + 1;
  return 0;

}



void f1(){}
void f2(){}
void f3(){}
```

Appendix II

```



void f1(int x);
void f2();

int main(int argc, char* argv[]){
  int a = 3;
  int b;
  int c;

  a = a + 2;

  if (c == 3) {
    c = b + c;
    b = b + 2;
    a = a + 1;
  }

  else{
    f2();
    b = c - b;
    a = a + 4;
    b = b + 1;
    b = b + 1;
    c = 0;
  }

  a = *argv[1];
  b = 10;
  b = b + 5;
  a = a + 1;

  if (c != 0){
    a = a + 1;
    b = b + 1;
    c = c + 1;
    f2();
    a = *argv[1];
  }

  else{
    a = a * 3;
    b = a + b;
    b = b*b;
    c = a + 3;
  }

  a = a + b;
  a = a + c;
  b = b + c;
  b = b*(a*3+c);
  c = 65*((a-b)*b-5);
  c = 8;
  f2();
  a = b + c;
  b = a + 2;
  c = 5;

  return 0;
}

void f2(){}
```

Semi-Automatic Segmentation of Surgical Instruments in Minimally Invasive Surgery Videos

Maxime CALKA
University Grenoble-Alpes, France
Maxime.Calka@etu.univ-grenoble-alpes.fr

Supervised by: Sandrine VOROS, Katia CHARRIERE, Arthur DERATHE.

I understand what plagiarism entails and I declare that this report is my own, original work.
Name, date and signature: CALKA, 05/06/2018

Abstract

In the past 30 years, surgical practices have evolved a lot with the appearance of Minimally Invasive Surgery (MIS). These techniques offer a lot of benefits to the patient, but are complex for the surgeon, and the learning curve is important compared to open surgery. The automatic analysis of surgical endoscopic videos can help to understand and analyze surgical gestures, and in term it can help to develop tools to assist surgeons during laparoscopy (Fig. 1(a)), especially during the learning period. For instance, surgical tool detection, localization or segmentation in the surgical scene provide precious information about the surgeon's gestures. This project proposes to develop a semi-automatic method based on tracking, color and shape methods to segment the instruments inside sleeve surgery videos. This semi-automatic approach allows to accelerate the constitution of a database to investigate deep learning detection approaches. We propose a method that searches a ROI in the whole image, then segment the tool inside of the ROI. The results demonstrate that the template matching algorithm allow to find a relevant ROI in $\simeq 1/4$ of the image that we wish to segment. In addition, our segmentation method gives a correct result compared to our ground-truth of the image where a relevant ROI is found.

Index terms — Image Processing, Segmentation, Tracking, Template Matching, Surgical Tools

1 Introduction

1.1 Minimally invasive surgery

Minimally invasive surgery (MIS) knows a large success in many application domains: digestive, urological, bariatric surgery [1]. This modern surgical technique reduces the operative trauma to the patient compared to open surgery, but is more complex for the clinician.

However, a problem with this kind of surgery is the difficulty to operate in a restricted field of vision, the loss of depth and haptic notion due to the use of a video feedback with limited access to the surgical scene. For all those reasons, MIS requires a long learning curve to the beginner.

That is why, analysis of the video flow can be helpful to the clinician in order to reduce the learning curve and improve quality of the surgical practice. Indeed, it helps to better understand the surgical skills or provide new intra-operative assistance tools.

1.2 Video analysis

In this context, the scientific community is interested by video analysis because it provides a large amount of relevant information which can be extracted to characterize MIS among which it finds the automatic tool detection.

Some example, like workflow analysis [2], evaluation of surgical skill ([3], [4]) or control surgery robot, use tool detection to characterize the surgery.

Workflow Analysis

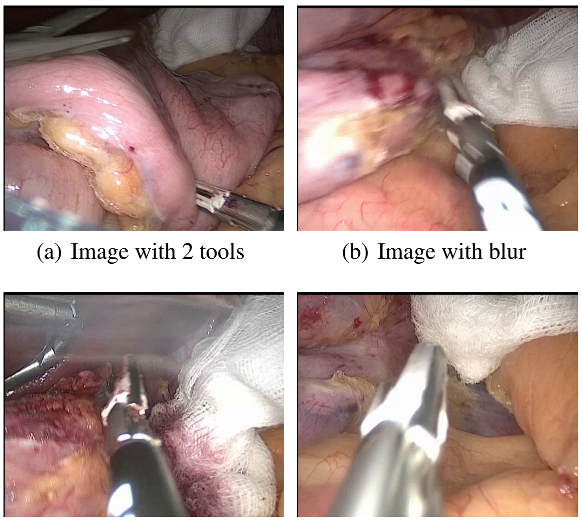
A first interesting video analysis application is the workflow analysis. This process decomposes a surgery or surgical step into phases, step and actions based on manual annotation. In this domain, the detection and recognition of surgical instruments can provide a precious information for surgical phase recognition and can enable some automatization of the annotations [2].

Quality evaluation

Another example is surgical skills analysis and quality evaluation of the surgical practice [3]. Various techniques are used for surgical skill assessment. This kind of evaluation is critical to ensure a high quality of care. That is why more and more objective computer aided technical skill evaluation (OCASE-T) emerge to help to evaluate skill quality [4]. A previous work, developed at TIMC-IMAG, proposes a method for the quantification of the surgical gesture quality. The approach uses an automatic tool tracking (in a test bench setup) to automatically compute a validated "quality" score called GOALS [5].

More recently, the review [3] showed that the tool tracking is used in a lot of benchtop simulation. Moreover, the growing availability of quantitative data documenting surgical performance and recent developments in machine learning methods, has allowed to quickly increase the development of OCASE-T methods.

Finally, he difficulties encountered in tool segmentation from laparoscopic images are caused by blur, smoke, occlusion or specular reflections on the tools, as illustrated by Fig. 1.


(a) Image with 2 tools (b) Image with blur

(c) Image wih smoke (d) Image with reflect

Figure 1: Set of frame in laparoscopic video

2 Related Work

2.1 Machine Learning

Recently, machine learning and particularly deep learning approaches shown great success to extract the information from the surgical video flow for a better analysis of the surgery, such as surgical tool detection and tracking ([2], [6], [7], [8], [9]). To this, two main processes are lots of used to this kind of work.

Random Forest

One of the most used is the random forest (RF) classification. For example, [10] measures the variable importance and detect four different colors features while [11] uses the RF classification to separate instruments pixels and background pixels. In [12], tools are tracked by building random forests and addressing the task of modelling the instruments as an articulated object.

Convolutional Neural Network

In recent approaches, Convolutional Neural Networks (CNNs) are used for the segmentation task. Some approaches based on CNNs search to localize the tools by detecting interest points ([7], [6]) or a bounding box [8].

Deep learning methods require a large number of annotated videos serving as ground-truth which are difficult to collect in the medical community.

Machine Learning Drawback

As of now, the databases are complex to create because the laparoscopic surgery videos are rare and can have limited relevance caused by none realistic setups, and a manual segmentation which is time consuming. This is illustrated by the annotated videos available in the MICCAI challenges [1] (a reference conference in the field), where the instruments are slowly moving in front of a static liver which is not very representative of the surgical reality.

2.2 Tracking & color and shape based methods

Today, the color and shape based techniques are less used than machine learning techniques. Nevertheless, they could be useful in the context of a semi-automatic segmentation tool, with the aim of help building the databases.

The approach [13] developed at the TIMC-IMAG is based on a color and shape method and the research of regions of interest (ROI). It proposes to use Lab color space to split color (a,b) and brightness and compute a grayscale image based on this both channels. This is followed by a binarization with the Otsu threshold and a morphological filter erosion to extract candidate regions corresponding possibly to the tool, then one region is selected as corresponding to the tool and become a ROI in which a Hough method is used to find the edges of the tool. Finally, a Kalman filter is applied to track the tools all along of the video. One advantage of this approach is its low computational load, leading to real-time performance.

Many other methods have been compared in the review [14], and exhibit a set of methods to detect surgical tools like feature representation, color, gradient, texture, shape, temporal tracking, tools constraint, etc.

3 Semi-Automatic Segmentation Method

As of now, TIMC-IMAG has constituted a database composed of 30 videos of laparoscopic surgery videos weakly annotated to investigate machine learning approaches for tool detection. Annotate this database manually takes a large amount of time. This paper proposes to expand and improve the method [13] to develop a semi-automatic surgical instrument segmentation method to support the database building. The goal of the project is to develop a tool which, based on a manual segmentation of a key frame, is able to automatically segmented a few subsequent frames.

4 Method

This work is based on a semi-automatic method such that the user provides a segmentation at time t that help to segment the tool at time $t+n$ with $n \in [1, ni]$n where ni is the number of frames automatically segmented between two manually segmented frames. The first step of our work is to track the tool all along of the video. Approaches evaluated in this paper are based on template matching in order to find a ROI fitted around the tool. The second step is the segmentation of the tool inside the ROI. One constraint is to segment the tool faster than a manual segmentation. Our method is described in the Fig. 2.

[1]https://endovissub201-roboticinstrumentsegmentation.grand-challenge.org/results/

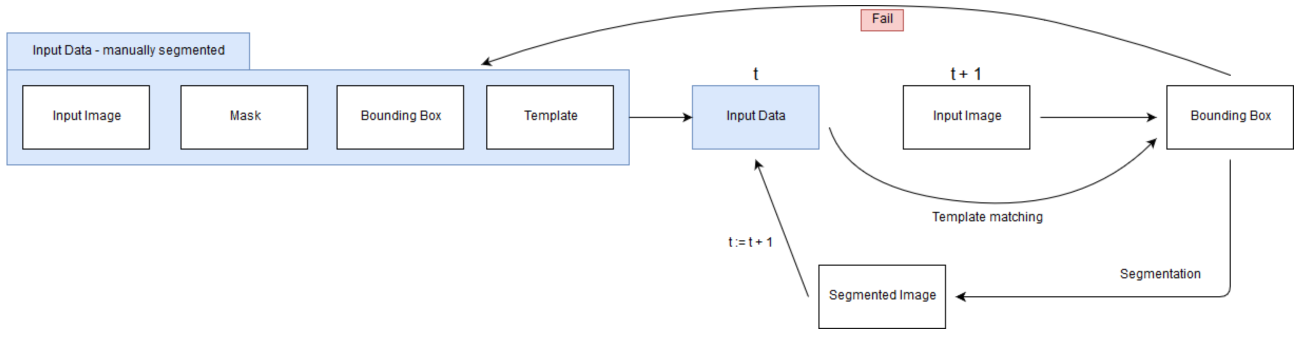
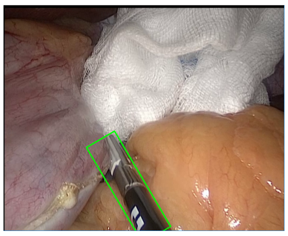


Figure 2: Semi-automatic method for surgical tool tracking and segmentation

4.1 Tracking

The first step of our method is a tracking system that defines a ROI that fit around of the tool all along of the video. A refinement of the segmentation space from the total image to a ROI that around around the tool is really important, because it allows to suppress a huge number of pixels belonging to the background.

We proposed a semi-automatic methodology, so the user provided an already segmented image at time t. This segmented image allows us to define automatically a ROI that fit around the tool. A method [13] previously developed at TIMC-IMAG try to find a ROI containing the tool, but it fails in some of difficult cases illustrated Fig. 1. However, this method works well on an already segmented mask (the mask of Fig. 3(b)), so we use these techniques to find a relevant ROI in time t. These two information (Fig.3(a) and Fig.3(b)) given by a manual user segmentation provides relevant information to track the tool.


(a) ROI

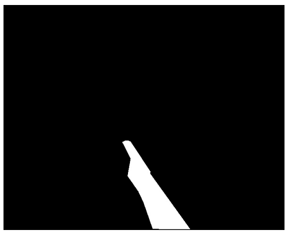

(b) Mask of a tool

Figure 3: Data provided by the user

The tracking process is initialized with a user manual segmentation at time t containing a relevant ROI. This ROI provides the template tracked in the following frames. The tracker is defined with a warping function, a similarity function, an acceptability function and some parameters. The warping function defines the type of transformation of the ROI between t and $t + n$, the similarity function is a score that give a quantitative result on the ROI detected, the acceptability condition decides if a ROI detected is a success (positive ROI) or a fail (negative ROI). If the ROI is positive the tracker is relaunched at $t + n$ else if the ROI is negative a new manual segmentation is asked to the user. This process is described Fig. 4.

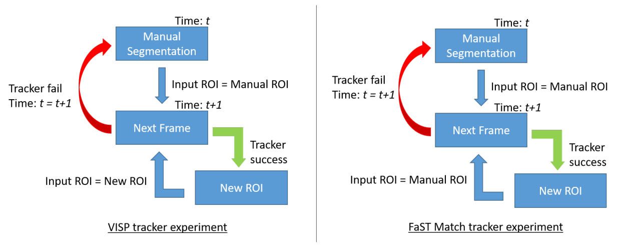


Figure 4: Tracking Process

This paper compares two methods proposed in the literature to track the ROI from the user data along with the video and find a precise ROI.

The first investigated method [15] is a template matching approach (Fig. 5) where the computation time is reduced, compared to complete space exploration methods, thanks to a sampling of the tested points within the template and a branch and bounds scheme. The warping function is an affine transformation where the shearing is removed because a tool can not undergo this kind of transformation. The best transformation is acquired by minimizing the similarity function defined as the Sum-Of-Absolute-Distance (SAD) between the template sampled points and the original image. An acceptability condition is defined as a threshold on the SAD score to detect incorrect ROI. The tracking of the tools with this technique is realized once every 25 frames as our objective is to segment one image every second.. Parameters of the method are the size of the points sample $\epsilon \in]0, 1]$, the step between the different transformations $\delta \in [0, 1]$, the consideration of the photometric appearance $p \in true, false$, the rotation and scaling range respectively r, s. If the tracker found a positive ROI, the ROI use for the next frame is, yet again, the manual ROI obtain at t.

The second method is a template tracker provided in the VISP library, that track a template by using an image registration algorithm [16]. The warping function is an homographic transformation and the tracking is realized on each frame. Here, the similarity function used is the Sum of Square Differences (SSD) with the inverse compositional algorithm [17]. Parameters to be assessed for this method are the following : the sampling that consider only one pixel from $n \times m$ pixel in

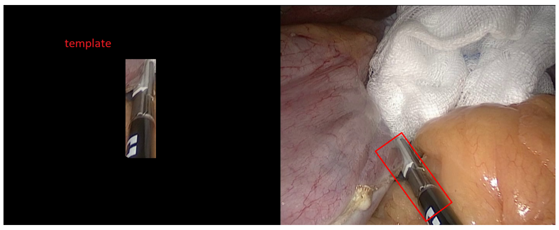


Figure 5: Example of template matching method

the reference image, so change this value increase/decrease the efficiency and the running time of the tracking phase, the number of iteration maximum to reduce the time necessary to minimizing the cost function, the number of level that allows to speed up the algorithm, and a parameter that forces the tracker to only consider the pixels with a high gradient value in the reference template. Here, our acceptability condition is only when an exception is catchy. Here, if the tracker found a positive ROI, this ROI is now use as template by the tracker for the next frame.

4.2 Segmentation

Once a ROI is detected at time $t+n$, we can restrain the segmentation space. This segmentation step consists to define an automatic method to binarize the image in two classes (background and tool) using the ROI previously found.

In this paper, we compare two segmentation processes that are an automatic thresholding method based on the study of color spaces, binarization processes and an homographic registration method.

Automatic Clustering-based segmentation Method We compare various color spaces and two automatic clustering-based approaches for the binarization step. The binarization is applied to the entire image, but only positive pixels inside the ROI are considered as representing a tool. While pixels outside are automatically considered as representing the background as shown in Fig. 6.

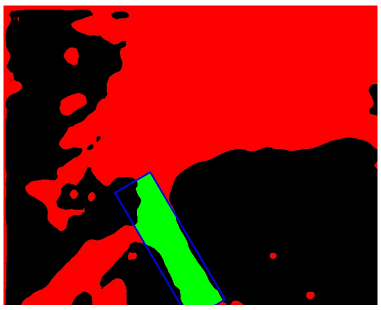
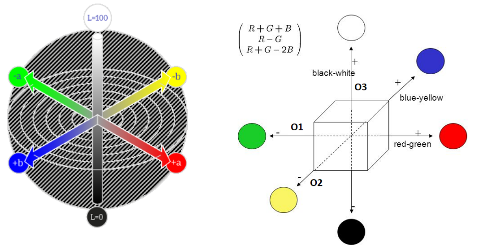


Figure 6: Only pixels inside the ROI are considered as positive. Green pixels represent the final result of the segmentation while red pixels are detected as tools but represent the background as the black pixels

Color Space and Channel Choice The goal of this step is to choose the channel which has the histogram with the best discriminant division between pixels representing tools or background to improve the binarization process. Two color spaces, frequently used in the State-of-Art, are compared : color space that mix brightness and color such that RGB, and color space that unmix them such that Lab or Opponent. The main difference between Lab and Opponent is their representations. Lab is represented like a sphere compared to Opponent that represent a cube and can be deduced by a linear combination between the different channels of RGB. These two color spaces are interesting because they unmixed brightness and color and will be allowed to avoid specular reflect by removing the brightness information. These both color spaces are represented in Fig. 7.



(a) Lab (b) Opponent

Figure 7: Two color space that split brightness and color

We choose, for each of these color space, the most valuable channel (grayscale image) among the three channels of each color space. In Lab and Opponent the channel representing brightness is not used to avoid the problem of specular reflect. In addition, we choose the channels representing the red component such as *RGB Red*, *Opponent 1* and *Lab a* because this color is most represented in the organs and less on the tools.

Binarization process A binarization process is then applied to the retained colorplane. For this step, the pixels of the image are divided into two classes (tools and background), so it is possible to evaluate which binarization method is the closest to the ground-truth provided by the manually segmented mask using some metrics introduced later.

Two methods are compared for the binarization process: the Otsu's method [18], and a Bayesian classification method developed at TIMC-GMCAO [19] in order to make the best clustering. The advantage of these two methods is the automatic separation of pixels in two classes from the histogram.

The Otsu's method, described Fig. 8, is an automatically clustering-based method using the histogram of a grayscale image to split in two classes (in our case tool, background) the images by computing the best threshold value. The main idea of the algorithm is the choice of an optimal threshold between the two classes, such that the variance intra-classes is minimal. The complexity of the Otsu's method is $n \times n$, with n the number of classes of the histogram.

The second method uses a Bayesian classifier to assign the most likely class to a given feature, so each pixel is classified like a tool if its tool posterior probability is larger than its

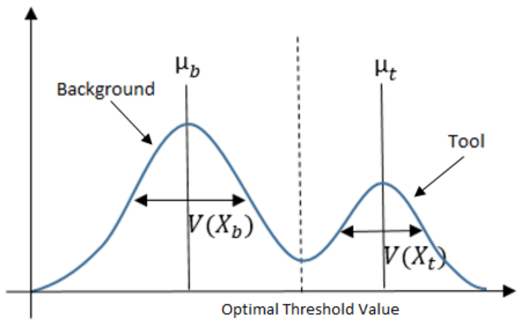


Figure 8: Description of the Otsu method

background posterior probability. This probabilistic method considers the histograms of pixel values as a mixture of two Gaussian functions (Fig. 9). Parameters of the method are therefore: mean (noted μ), variance (noted V), prior probability (denote pt and pb, for tool and background respectively) for the two histograms which are optimized with the expectation-maximization algorithm (EM). The complexity of the Bayesian classification method is $O(k \times n \times m)$, where k is the number of iterations, n the number of rows in the image and m the number of columns.

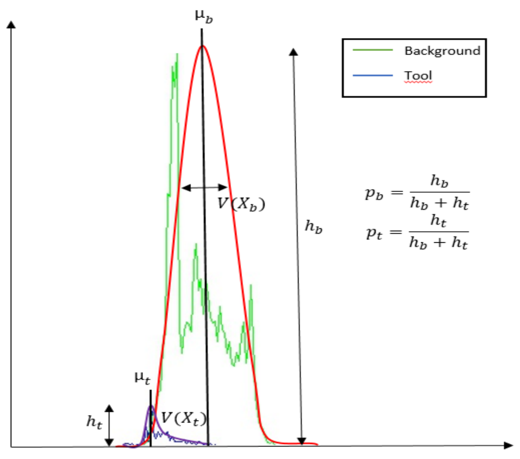


Figure 9: Gaussian approximation of a histogram

Homographic registration

A second approach evaluated is the homographic registration to segment the frame. we compute here the homography matrix H between the ROI at time t (manually segmented) and the ROI at time $t+n$ (output of the tracker). Then, it is possible to compute the homography matrix H between the two ROIs. Then, H is used to compute the same transformation (1) for each pixel of the mask at time t, so a new mask is obtained in the frame $t+n$.

$$\begin{bmatrix} x'/z \\ y'/z \\ z \end{bmatrix} = H \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}, with\, H = \begin{bmatrix} h11 & h12 & h13 \\ h21 & h22 & h23 \\ h31 & h32 & h33 \end{bmatrix} \quad (1)$$

5 Experimental Protocol

5.1 Tools

Our implementation of the method is integrated in the CamiTK framework[2] (version 4.0.4). The image processing task was performed thanks to OpenCV 3.1 library[3].

5.2 Data

This semi-automatic segmentation method was applied and evaluated on the practical case of the sleeve gastrectomy surgery. Our experiments were realized on the only already segmented video (= 426 frames with a resolution of 720 × 576 pixels) that provides a ground-truth segmentation (ground-truth mask, ground-truth ROI). These videos were gathered at the Grenoble CHU in collaboration with an expert surgeon. In our case, the annotation was performed at a frequency of one frame per second on a key step of the surgery named Fundus dissection.

5.3 Parameters

Tracking

For the two trackers, some parameters required to be fixed.

For the FAsT Match algorithm the parameters ϵ, the number of points and δ, the step of the transformation was respectively set to 0.2 and 0.5. These parameters were empirically defined with the aim of obtaining a sufficient number of configurations and a sufficient sample of points to obtain a compromise between quality of the ROI and acceptable running -time. For the affine transformation parameters, the rotation range was studied and shows that a range $r \in [-90.0°, 90.0°]$ allows to include a large amount of cases. The scaling range was empirically defined with $r \in [1.0, 1.2]$ allowing to prevent the increase of the size of the objects. The photometric invariance parameter has enabled because the laparoscopic images are sensitive to illumination. Our algorithm stops when the SAD score is the same for two consecutive iterations or the number of iterations is upper than twenty and a ROI is validated when the SAD score is lower than 0.075. To define this score, a study of the average score for a relevant ROI was performed.

For the VISP tracker, the parameters was defined to make a trade-off between running-time and efficiency. We defined a sampling 2 × 2, a maximum, at the maximum number of iterations to 200, the pyramidal parameters with 3 levels from 3 to 0 and the parameter that force the tracker to only consider the pixels with a high gradient value is enabled.

Segmentation

For the both methods, the maximization of the parameters is computed on the whole image. The Otsu's method chooses automatically the threshold by separating in two classes the histogram Fig. 8. For the Bayesian method, the EM algorithm optimal mean and variances for the two Gaussian functions are estimated thanks to the EM algorithm in 10 iterations. However, to converge the algorithm needs initials

[2]http://camitk.imag.fr/
[3]https://opencv.org/

parameters values to start. Those initials values are automatically computed from the two Gaussian, so the parameters are: the mean as the center of the histogram the variance as the value situated at 68% on it and the prior probabilities as a ratio between the maximum values of the two histograms. This process is described Fig. 9.

5.4 Metrics

Tracking

The tracker detects if a ROI is positive or negative (success or fail of the tracker). We use the IoU score (2) that corresponds to the overlap between the ROI detected and the ground-truth provided. This score allows to define if the result of the tracker is coherent with the ground-truth (true if the result is coherent or false otherwise). A ROI is consider as a coherent with the ground-truth (true) if the overlap between the two ROI is superior to 50%. This metric allows to define the number of true positive ROI and know the percentage of automatically tracked image (3).

$$IoU = \frac{Area\,of\,Overlap}{Area\,of\,Union} \tag{2}$$

$$\%\,of\,TP\,ROI = \frac{\#\,of\,TP\,ROI}{\#\,image\,available} \tag{3}$$

Thereafter, some metrics are used to evaluate the pertinence of the tracker: the accuracy (4), the recall (5), the specificity (6). The percentage of automatically tracked image that helps us to assess the time saving for the manual segmentation, the accuracy that corresponds to the proportion of pixels well classified, the recall is the proportion of selected items that are relevant and the specificity is the proportion of none relevant item selected truly none relevant.

$$Accuracy = \frac{TP + TN}{P + N} \tag{4}$$

$$Recall = \frac{\#TP}{\#TP + \#FP} \tag{5}$$

$$Specificity = \frac{\#TN}{\#TN + \#FP} \tag{6}$$

Finally, the running-time is computed to evaluate the speed of the tracker.

Segmentation

We realize the quantitative comparison of the different methods we use the ”ground-truth” provided. The metrics used to compare each method are: the balance accuracy (7), the recall (5), the specificity (6) and the DICE/F1 score (8). The balance accuracy is a metric more precise that the accuracy because they balance the number of true negative and true positive. The DICE/F1 represents the size of the overlap between the segmentation and the ground-truth over the total size of the both objects. These metrics are used in the literature to evaluate automatic segmentation machine learning methods [6]. Note that the metrics are computed on the whole image.

$$B.Accuracy = \frac{Specificity + Recall}{2.0} \tag{7}$$

$$DICE = \frac{2\#TP}{2\#TP + \#FP + \#FN} \tag{8}$$

5.5 Experiments

Tracking

To evaluate the best tracking method, each method is evaluated on the whole video. Three characteristics are study the efficiency of the tracker, the pertinence of the tracker and his running-time. The efficiency is define like the percentage of the video segmented, the pertinence like the ability of the tracker to detect correct or incorrect ROI. For each ROI (positive or negative), we compute the IoU score to know if the result of our tracker is true or false and evaluate his efficiency and her pertinence. For the running-time, we make an average of the time to find each ROI. The process is relaunched if the ROI is negative.

Segmentation

The segmentation experiment compares two kinds of methods. One using automatic clustering and the other one using homographic registration. We evaluated the two automatic clustering-based methods, for the binarization process, on each channel from our three color spaces and the homographic registration method. To obtain a relevant evaluation, we made the hypothesis that the ROI is known, so this part was firstly evaluated independently of the tracking part.

Whole Process

Finally, for the whole process, the segmentation process is applied of each true positive ROI.

6 Results

6.1 Tracking

Results for the comparison of the both trackers are presented in Table 1. We show that the VISP tracker tracks automatically more images and is faster than the FAsT Match tracker. In addition, when the VISP tracker detects an incorrect ROI, this ROI is always incorrect (according to the IoU score) in contrary of the FAsT Match tracker. However, the FAsT Match tracker is better to detect an incorrect ROI than the VISP tracker. The accuracy shows that the VISP tracker has a better classification rate that the FAsT Match tracker.

The percentage of successfully tracked ROI and the running-time are better for the VISP tracker than for the FAsT Match tracker. In addition, the VISP tracker owns a better classification rate. The only advantage of the FAsT match is for the detection of the incorrect ROI, thus the segmentation methods will be evaluated with the VISP method.

6.2 Segmentation

Results for the comparison of the segmentation approach are presented in Table 2. We see that the channels a and O1 that belonging to color space that unmix brightness and color have better results that the red channel except for the specificity that is homogeneous for each method. In addition, the

Table 1: Quantitative comparison between the tracking methods

	% of TP ROI	Accuracy	Recall	Specificity	Running-time (in s.)
VISP Tracker	23.47	83.57	100.0	78.53	4.93401
FAsT Match Tracker	14.08	66.43	36.14	85.76	72.9919

Table 2: Quantitative comparison between the different automatic clustering based segmentation methods

	B.Accuracy	Recall	Specificity	DICE
Otsu a	95.87 (± 4.054)	93.76 (± 8.171)	*97.98 (± 2.097)*	87.93 (± 7.354)
Otsu O1	***96.40 (± 4.384)***	***94.83 (± 8.858)***	97.97 (± 2.294)	***88.68 (± 8.052)***
Otsu R	84.15 (± 9.322)	71.47 (± 18.40)	96.83 (± 2.663)	69.62 (± 13.76)
Bayesian a	92.58 (± 11.63)	87.13 (± 23.33)	98.03 (± 2.085)	82.23 (± 21.34)
Bayesian O1	*93.71 (± 8.196)*	*89.22 (± 16.70)*	98.20 (± 2.278)	*85.75 (± 14.26)*
Bayesian R	77.19 (± 17.80)	56.13 (± 36.88)	***98.25 (± 1.990)***	54.77 (± 30.64)
Homography	89.11 (± 8.103)	79.98 (± 15.12)	98.23 (± 2.733)	81.06 (± 14.40)

Table 3: Quantitative result of the segmentation with VISP Tracker

	B.Accuracy (in %)	Recall (in %)	Specificity (in %)	DICE (in %)
Otsu O1	90.82	83.50	98.14	81.72

results based on the channel O1 are better for the both binarization methods. The homographic method is outperformed by the method with a and O1 channel on the DICE score, the balance accuracy and the recall. However, it is more efficient than the method using the R channel.

The Fig 10 and Fig 11 show some quantitative results with the Otsu's method and Opponent 1 channel. This both images show that our method can realize relevant segmentation. The Fig 11 is representative of a common problem on our dataset, where the tip of the tool has similar color with the compress, and that for each color space evaluated.

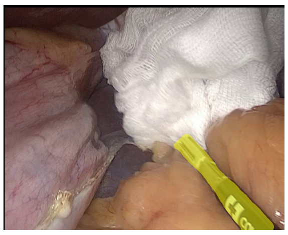


Figure 10: Relevant Electrothermal Biforceps segmentation with Otsu's method on O1 channel

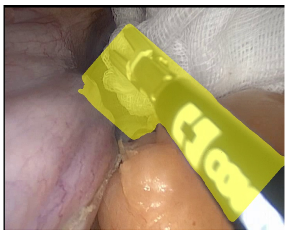


Figure 11: Incorrect Electrothermal Biforceps segmentation with Otsu's method on O1 channel

6.3 Whole process

Finally, the tables 3 give the results of our whole process. The tracker is able to find a relevant ROI for 23.47% of the image that we wish to segment (Table 1). In addition, the overlaps between the GT mask and our segmentation is in average of 81.72%. On average, our method well classified the pixel in 90.82% of the cases. Note that this result are obtained in the image where the ROI is TP.

7 Discussion

7.1 Tracking

The first experiments of our study investigate, compare and evaluate two tracking methods. The tracking experiments described earlier show that the most efficient tracker is the VISP tracker because it is faster and more efficient than the FAsT Match tracker. In addition, a ROI detected is not always relevant as show in the table 1, so the VISP better detect the number of irrelevant ROI. This first experiment responds at a question of our problematic, i.e the VISP tracker is able to obtain a relevant ROI for 23% of the image that we wish to segment.

A limitation of our results is the use of only one annotated video for the evaluation. However, many results are available and allow to choose a tracking method. In addition, they show interesting results on the quality of the ROI and the efficiency of the tracker to detect irrelevant ROI.

7.2 Segmentation

The segmentation methods investigate two kinds of methods (automatic clustering, Bayesian classification). The results of the automatic clustering-based segmentation give some interesting information. Results in Table 2 valid that in the case of laparoscopic images, color spaces that unmixing brightness and color are more discriminant than color spaces mixing the both. This could be caused by the specular reflects that illuminate the tools and organs and harmonize the value of the pixels to high intensities. Thereafter, the use of color space unmixing brightness and color for clustering-based segmentation is justified. In addition, the Opponent O1 channel seems to be more discriminant channel. The Otsu's method own better score than a Bayesian classification. In addition, these results are satisfying because the complexity of the Otsu's method is inferior to the Bayesian classification complexity. Finally, the comparison between the Otsu's method with O1 channel and the homographic registration show that the clustering-based segmentation is more efficient. This difference is caused by the high difference in the shape and the size of the tools between two frames, so it is difficult to find an homographic registration between two frames sampled every second (25 frames).

A limitation in our experiment is to compute the metrics on the whole image because the number of true negative pixels is very important compared to true positive pixels in the image, so some metrics are distorted.

7.3 Whole process

Our method is able to segment $1/4$ of the video. In addition, the results of the whole process detailed in Table 3 show that the segmentation is close to the bounding box. However, this segmentation stays imperfect, so the user needs to make few change on the segmentation to correct it. However, the segmentation is restricting to the ROI and implies only small deformation to realize for the user.

In this paper, our method was not compared to State-of-Art approaches. This project tries only to investigate some methods to reduce the time for manual segmentation, not to outperform the other segmentation methods. In addition, our method is a semi-automatic one compared to the others that are automatic.

8 Conclusion and Future work

In this paper, we propose a general method to segment semi-automatically the tool inside laparoscopic surgery videos. We validate a method able to automatically segment 23% of the required images, with some acceptable performances.

In future work, the most important work is to increase the number of videos to realize the experiments. In addition, another work will be to improve the tracker to increase the percentage of automatically segmented images. Finally, our final aim, is to develop a complete software allowing to modify the segmentation manually and thus reduce the manually segmentation time.

References

[1] C. Schaaf, A. Iannelli, and J. Gugenheim, "État actuel de la chirurgie bariatrique en france," *e-mémoires de l'Académie Nationale de Chirurgie*, vol. 14, no. 2, pp. 104–107, 2015.

[2] A. P. Twinanda, S. Shehata, D. Mutter, J. Marescaux, M. de Mathelin, and N. Padoy, "Endonet: A deep architecture for recognition tasks on laparoscopic videos," *IEEE transactions on medical imaging*, vol. 36, no. 1, pp. 86–97, 2017.

[3] S. S. Vedula, M. Ishii, and G. D. Hager, "Objective assessment of surgical technical skill and competency in the operating room," *Annual review of biomedical engineering*, vol. 19, pp. 301–325, 2017.

[4] M. C. Vassiliou, L. S. Feldman, C. G. Andrew, S. Bergman, K. Leffondré, D. Stanbridge, and G. M. Fried, "A global assessment tool for evaluation of intraoperative laparoscopic skills," *The American journal of surgery*, vol. 190, no. 1, pp. 107–113, 2005.

[5] R. Wolf, *Quantification de la qualité d'un geste chirurgical à partir de connaissances a priori*. PhD thesis, Université de Grenoble, 2013.

[6] I. Laina, N. Rieke, C. Rupprecht, J. P. Vizcaíno, A. Eslami, F. Tombari, and N. Navab, "Concurrent segmentation and localization for tracking of surgical instruments," in *International Conference on Medical Image Computing and Computer-Assisted Intervention*, pp. 664–672, Springer, 2017.

[7] T. Kurmann, P. M. Neila, X. Du, P. Fua, D. Stoyanov, S. Wolf, and R. Sznitman, "Simultaneous recognition and pose estimation of instruments in minimally invasive surgery," in *International Conference on Medical Image Computing and Computer-Assisted Intervention*, pp. 505–513, Springer, 2017.

[8] D. Sarikaya, J. J. Corso, and K. A. Guru, "Detection and localization of robotic tools in robot-assisted surgery videos using deep neural networks for region proposal and detection," *IEEE transactions on medical imaging*, vol. 36, no. 7, pp. 1542–1549, 2017.

[9] A. Shvets, A. Rakhlin, A. A. Kalinin, and V. Iglovikov, "Automatic instrument segmentation in robot-assisted surgery using deep learning," *arXiv preprint arXiv:1803.01207*, 2018.

[10] M. Allan, S. Ourselin, S. Thompson, D. J. Hawkes, J. Kelly, and D. Stoyanov, "Toward detection and localization of instruments in minimally invasive surgery," *IEEE Transactions on Biomedical Engineering*, vol. 60, no. 4, pp. 1050–1058, 2013.

[11] S. Bodenstedt, M. Wagner, B. Mayer, K. Stemmer, H. Kenngott, B. Müller-Stich, R. Dillmann, and S. Speidel, "Image-based laparoscopic bowel measurement," *International journal of computer assisted radiology and surgery*, vol. 11, no. 3, pp. 407–419, 2016.

[12] N. Rieke, D. J. Tan, C. A. di San Filippo, F. Tombari, M. Alsheakhali, V. Belagiannis, A. Eslami, and

N. Navab, "Real-time localization of articulated surgical instruments in retinal microsurgery," *Medical image analysis*, vol. 34, pp. 82–100, 2016.

[13] A. Agustinos, *Navigation augmented fluorescence informations for the laparoscopic surgeryrobot-assisted*. Theses, Université Grenoble Alpes, Apr. 2016.

[14] D. Bouget, M. Allan, D. Stoyanov, and P. Jannin, "Vision-based and marker-less surgical tool detection and tracking: a review of the literature," *Medical image analysis*, vol. 35, pp. 633–654, 2017.

[15] S. Korman, D. Reichman, G. Tsur, and S. Avidan, "Fast-match: Fast affine template matching," in *Computer Vision and Pattern Recognition (CVPR), 2013 IEEE Conference on*, pp. 1940–1947, IEEE, 2013.

[16] A. Dame and E. Marchand, "Video mosaicing using a mutual information-based motion estimation process," in *Image Processing (ICIP), 2011 18th IEEE International Conference on*, pp. 1493–1496, IEEE, 2011.

[17] S. Baker and I. Matthews, "Lucas-kanade 20 years on: A unifying framework," *International journal of computer vision*, vol. 56, no. 3, pp. 221–255, 2004.

[18] N. Otsu, "A threshold selection method from gray-level histograms," *IEEE transactions on systems, man, and cybernetics*, vol. 9, no. 1, pp. 62–66, 1979.

[19] H. Younes, S. Voros, and J. Troccaz, "Automatic needle localization in 3d ultrasound images for brachytherapy," in *2018 IEEE 15th International Symposium on Biomedical Imaging (ISBI 2018)*, pp. 1203–1207, April 2018.

Integrating lexical constraints to K-Means with Deep Learning

Grand Maxence
Supervised by :
Gaussier Éric and Thonet Thibaut
and Tommasi Marc and Bellet Aurélien and Moradi Fard Maziar

Abstract

We study in this paper the problem of thematic clustering. Thematic clustering is the set of lexical constraints skewing the text clustering towards different themes. These lexical constraints are represented by a set of keywords given by the user. To perform K-Means with these constraints, we need a representation where data about documents and constraints are present. For this reason, deep K-Means can be used to learn a latent space showing all constraints, and perform K-Means in the latent space. We propose here an approach to integrate constraints to K-Means algorithm by introducing penalties in the Non Clustering loss of the Deep K-Means model[4].

Keywords Clustering, K-means, DNN, Autoencoder, Deep Clustering

1 Introduction

Clustering is one of the most fundamental tasks in data mining and machine learning. K-Means algorithm is a clustering method using centroid models, it represents each cluster by a single mean vector. K-Means clustering sorts n objects into k clusters in which each observation belongs to the cluster with the nearest centroid. Then, K-Means is often used in practice and it is easy to interpret.
In real application domains, users may want to introduce constraints to finding useful properties for clustering data. Traditional K-Means algorithms have no way to take advantage of this information.
The difficulty with integration of constraints into K-Means algorithm is to find a good representation for data taking into account constraints. The Deep Learning and Auto-Encoder can be used to learn this representation. With Auto-Encoder we have to perform the K-Means in the latent space learned, and this latent space must be K-Means friendly.
In this study, we specifically focus on the k-Means algorithm with lexical biases. Lexical biases are represented by a set of keywords given by the user. We use an Autoencoder to learn a latent space taking into account biases. The loss of the Autoencoder is divided in two parts, (a) the reconstruct loss L_{rec} and (b) different penalties skewing the representation. Then, the representation must be K-Means friendly, to do this, we use the Deep K-Means model [4].
In the next section, we provide some background on the K-Means algorithm and deep learning. In section 3, we proposed a method to introduce constraints to the K-Means algorithm. And we are experimenting our method in section 4.

2 Background

2.1 K-Means

Given a corpus C, where each document X is a d-dimensional real vector, k-means clustering aims to partition the n documents into K S_k clusters represented by centroids R = $r_1, r_2, ..., r_K$. Formally, the objective is to minimize :

$$\sum_{k=1}^{K} \sum_{X \in S_k} ||X - r_k||_2^2$$

We can see K-Means algorithm in algorithm1

```
input  : Corpus C, the number of cluster K
output: Assignment matrix S
Let $r_1^0, r_2^0, ..., r_k^0$ be the initial centroids
$t \leftarrow 1$
repeat
    forall $X \in C$ do
        $S_k^t \leftarrow \{X : ||X - r_k^{t-1}||_2^2 \leq ||X - r_l^{t-1}||_2^2 \forall l \neq k, 1 \leq l \leq K\}$
    end
    foreach centroids $r_k$ do
        $r_k^t \leftarrow \frac{1}{|S_k^t|} \sum_{X \in S_k^t} X.$
    end
    $t \leftarrow t + 1$
until Convergence;
return $S$
```

Algorithm 1: K-means

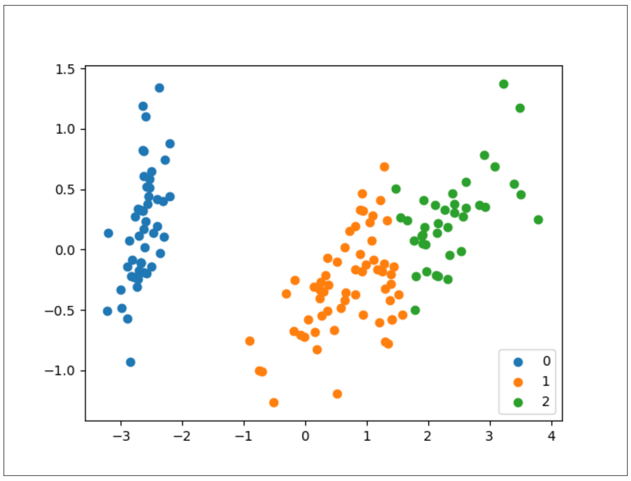


Figure 1: K-Means algorithm result for the Iris Flower Dataset [6]

2.2 Autoencoder

The Autoencoder [7] allows to learn a latent space with significant information for the clustering without loss of information. The Autoencoder tries to learn a function $f(X,\theta) = X$. In other words, it is trying to learn an approximation of the identity function. An Autoencoder is composed in two parts an encoder function g, and a decoder function f. To learn representations from which it is possible to reconstruct input data, we use a reconstruct loss L_{rec} using euclidean distance :

$$L_{rec}(X;\theta) = \sum_{X} ||X - A(X,\theta)||_2^2 \tag{1}$$

where $A(X,\theta)$ is the Autoencoder output. We denote $h_\theta(X) = g(X,\theta)$ the encoder output. The reconstruct loss allows to minimize the euclidean distance between input and output. In other word, after the training of our Autoencoder, the output will be close to the input.
Minimize the reconstruct loss allows to have a latent space where essential data are kept. In addition to reconstruct loss we can add different penalties Ω to bias the latent space. If we add penalties we need to introduce different hyperparameters Λ to balance the different losses.
Generally, the Autoencoder loss with m penalties takes the form :

$$L(X;\theta) = \sum_{X} ||X - A(X,\theta)||_2^2 + \lambda_0\omega_0 + \ldots + \lambda_m\omega_m \tag{2}$$

2.3 Deep Clustering

A several approaches for the deep K-Means propose to jointly learn the representation and perform the K-Means algorithm [2]. In these approaches, network's loss is divided in two parts : The non-clustering loss and the clustering loss. The non-clustering loss does

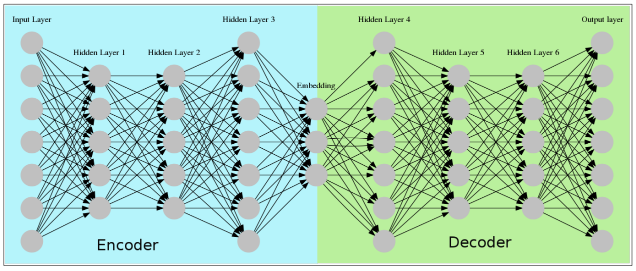


Figure 2: Autoencoder

not take into account of the clustering parts. In general, the non-clustering loss is the reconstruction loss of the auto-encoder. We can add additional information in the non-clustering loss to bias the representation. For example, in our case, we add penalties to integrate lexical constraints . The clustering loss allows to learn a K-means friendly representation. Moradi Fard, Thonet and Gaussier [4] proposed a method for deep K-Means clustering based on a continuous reparametrization of the objective function that leads to a truly joint solution. The problem takes the form :

$$\begin{aligned} L(C,\alpha;\theta,R) = & \sum_{X\in C} ||X - A(X;\theta)||_2^2 \\ & +\lambda_0 \sum_{X\in C} ||h_\theta(X) - c(h_\theta(X);R)||_2^2 \end{aligned} \tag{3}$$

where $c(g(X;\theta);R) = \underset{k=1..K}{\arg\min} ||h_\theta(X) - r_k||_2^2$ is a non differentiable function that assigns the document X to its nearest centroid.
They transform this representation as follows :

$$\begin{aligned} L(C,\alpha;\theta,R) = & \sum_{X\in C} ||X - A(h_\theta(X))||_2^2 + \\ & \lambda_0 \sum_{X\in C}\sum_{k=1}^{K} ||h_\theta(X) - r_k||_2^2 G_k(h_\theta(X),\alpha;R) \end{aligned} \tag{4}$$

where G_k is a differentiable function such that :

$$\lim_{\alpha\to\alpha_0} G_k(h_\theta(X),\alpha;R) = \begin{cases} 1 & \text{if } r_k = c(h_\theta(X);R) \\ 0 & \text{Otherwise.} \end{cases} \tag{5}$$

where α play the role of an inverse temperature. In [4] , G_k was chosen to be a parameterized softmax.
The softmax function take as input a K-dimensional vector z of real values and return K-dimensional vector $\sigma(z)$ of positive real values and all the entries add up to 1. The softmax function takes the form :

$$\forall k, \sigma(z)_k = \frac{e^{z_k}}{\sum_{k'=1}^{K} e^{z_{k'}}} \tag{6}$$

In the case of deep K-Means, they want :

$$\sigma(z)_k \to \begin{cases} 1 & \text{if } r_k = c'(h_\theta(X);R) \\ 0 & \text{Otherwise.} \end{cases} \tag{7}$$

Finally, G_k is defined as follows :

$$G_k(h_\theta(X), \alpha; R) = \frac{e^{-\alpha||h_\theta(X)-r_k||_2^2}}{\sum_{k'=1}^{K} e^{-\alpha||h_\theta(X)-r_{k'}||_2^2}} \tag{8}$$

To update (θ, R), they used the stochastic gradient descent (SGD) as follows :

$$(\theta, R) \leftarrow (\theta, R) - \epsilon \frac{1}{|\widetilde{C}|} \nabla_{(\theta,R)} L(C, \alpha; \theta, R) \tag{9}$$

where $\widetilde{C}$ is a random mini batch of C, and ϵ the learning rate.
SGD is an iterative method for optimizing a differentiable objective function. If we find a local minimum (respectively maximum) of an objective function using stochastic gradient descent, one takes steps proportional to the negative (respectively positive) of the gradient of the function at the current point. It is called stochastic because samples are shuffled.
Algorithm 2 summarizes the deep K-Means algorithm :

```
input  : Corpus C , number of clusters K, balancing
         parameter $\lambda_0$, scheme for $\alpha$, number of
         epochs T , number of minibatches MB ,
         learning rate $\epsilon$
output: Autoencoder parameter $\theta$, cluster
         representative R
Initialize $\theta$ and $r_k$, $1 \leq k \leq K$ (randomly or through
  pretraining)
foreach $\alpha = m_\alpha : M_\alpha$ do
    foreach $t = 1 : T$ do
        foreach $n = 1 : MB$ do
            Draw minibatch $\widetilde{C} \subseteq C$
            Update $(\theta, R)$ using SGD
        end
    end
end
```

Algorithm 2: Deep K-Means

2.4 Seed Words classification

Li, Xing, Sun and Ma [12] proposed a classification algorithm using Keywords. They assume that keywords could be semantically or statistically related to the seed words of the same category. In this paper, they use a function p to compute the co-occurrence of each word of a document with keywords of each category :

$$p(i|w) = \frac{df(w,i)}{df(w)} \tag{10}$$

where $df(w)$ is the number of the documents containing keywords w, $df(w,i)$ is the number of the documents containing both word i and keywords w. Then,

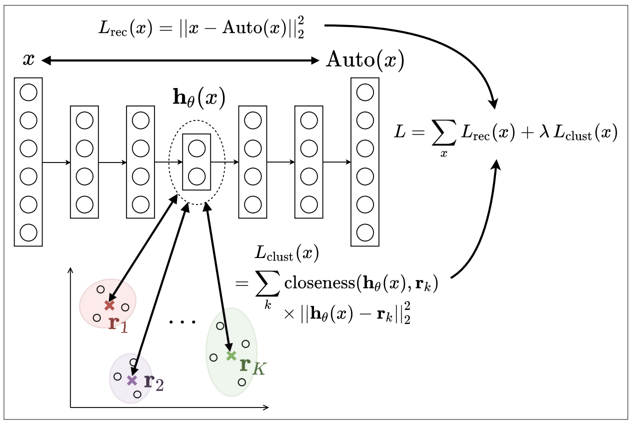

$L_{\text{rec}}(x) = ||x - \text{Auto}(x)||_2^2$
x
Auto(x)
$\mathbf{h}_\theta(x)$
$L = \sum_x L_{\text{rec}}(x) + \lambda L_{\text{clust}}(x)$
$L_{\text{clust}}(x) = \sum_k \text{closeness}(\mathbf{h}_\theta(x), \mathbf{r}_k) \times ||\mathbf{h}_\theta(x) - \mathbf{r}_k||_2^2$
$\mathbf{r}_1$
$\mathbf{r}_2$
$\mathbf{r}_K$
...

Figure 3: Overview of the Deep k-Means approach

they calculate the relevance score $rel(i,k)$ for each word i and category k as follows :

$$rel(i,k) = \frac{1}{|k|} \sum_{w \in k} s(w,i) \tag{11}$$

$$v(i,k) = max \left(\frac{rel(i,k)}{\sum_{k' \in KW} rel(i,k')} - \frac{1}{|KW|}, 0 \right) \tag{12}$$

In Equation 12, they normalize the relevance score $rel(i,k)$ and subtract it by the average relevance score for each category. It is expected that word i is a category word for category k only if i and k have a high $rel(i,k)$ value. Therefore subtracting the relevance scores by the average is necessary to filter out irrelevant categories.

3 Integrating Lexical Biases to Deep K-Means Algorithm

The idea is to learn a latent space taking into account lexical biases.
We denote X a document of size d, C the corpus, N the size of C, and K the number of cluster.
We denote :

$$KW = \begin{pmatrix} KW_1 \\ ... \\ KW_k \\ ... \\ ... \\ KW_K \end{pmatrix} \tag{13}$$

where KW_k the set of keyword for the k^{th} class. Also, we denote X' a biasing version of X. X' can be defined in two different ways :

1. **Masked Document** :

$$X'_k = mask_k(X) \tag{14}$$

where $mask_k(X)$ is the document X masked by keywords from the k^{th} class

$$\forall i, mask_k(X)_i = \begin{cases} X_i & \text{if } i \in KW_k \\ 0 & \text{Otherwise.} \end{cases} \tag{15}$$

Masked documents allow to have a representation biased by keywords. The risk with this method is to have a large number of documents containing no keywords, and therefore to have many X' such that $\forall k, X'_k = \vec{0}$.

2. **Similarity function** :
   For the similarity function we use the score defined in equation 12 :

$$X'_k = \forall i, X_i \nu(i,k) \tag{16}$$

The similarity function allows to minor the words being semantically distant from keywords. In addition, unlike hidden documents, words that are semantically close to keywords will not have a null score. Nevertheless the risk with this method is to have words with little importance for clustering have high score.

3.1 Lexical Constraints

We want a latent space where document X is close to the closest X'_k :

$$||h_\theta(X) - h_\theta(c'(X;X'))||_2^2 \tag{17}$$

where $c'(X;X') = \underset{X'_k \in X'}{\arg\min} ||h_\theta(X) - h_\theta(X'_k)||_2^2$

However, the function c' is not differentiable, and we cant use SGD algorithm [5] to learn this function. To approximate the arg min function we can use the parameterized softmax function [17]. The softmax function can be used as a differentiable substitute to arg min.
Indeed, want representation where document X is close to the closest, so we want a function ω_k such that :

$$\omega_k = \begin{cases} 1 & \text{if } r_k = c'(h_\theta(X);R) \\ 0 & \text{Otherwise.} \end{cases} \tag{18}$$

Then if we minimize

$$\sum_{X \in C} \sum_{k=1}^{K} \omega_k ||h_\theta(X) - h_\theta(X'_k)||_2^2 \tag{19}$$

we minimize the distance between the document $h_\theta(X)$ and the closest $h_\theta(X'_k)$ allowing to have a latent space where document X is close to the closest X'_k. Let's take the equation 6 and set $z_k = -\alpha||h_\theta(X) - h_\theta(X'_k)||_2^2$. So we have :

$$\lim_{\alpha \to \alpha_0} \sigma(z)_k \to \left\{ \begin{array}{ll} 1 & \text{if } r_k = c'(h_\theta(X);R) \\ 0 & \text{Otherwise.} \end{array} \right\} \approx \omega_k \tag{20}$$

Finally, the penalty takes the form :

$$\sum_{X \in C} \sum_{k=1}^{K} \frac{e^{-\alpha||h_\theta(X) - h_\theta(X'_k)||_2^2}}{\sum_{k'=1}^{K} e^{-\alpha||h_\theta(X) - h_\theta(X'_{k'})||_2^2}} ||h_\theta(X) - h_\theta(X'_k)||_2^2 \tag{21}$$

3.2 Deep K-Means

For the Deep K-Means, we can use the approach proposed by Moradi Fard, Thonet and Gaussier [4] see in section 2.3 introducing lexical biases with equation 21.
We denote $R = (r_1 \quad r_2 \quad ... \quad r_K)$ the vector of centroids.
The loss function is :

$$\begin{aligned} L(C,\alpha;\theta,R) = \quad & \sum_{X \in C} ||X - A(X;\theta)||_2^2 + \\ & \lambda_0 \sum_{X \in C} \sum_{k=1}^{K} ||h_\theta(X) - r_k||_2^2 G_k(h_\theta(X),\alpha;R) + \\ & \lambda_1 \sum_{X \in C} \sum_{k=1}^{K} \frac{e^{-\alpha||h_\theta(X) - h_\theta(X'_k)||_2^2}}{\sum_{k'=1}^{K} e^{-\alpha||h_\theta(X) - h_\theta(X'_{k'})||_2^2}} ||h_\theta(X) - h_\theta(X'_k)||_2^2 \end{aligned} \tag{22}$$

with hyperparameters $\lambda_0 \geq 0, \lambda_1 \geq 0$.

3.3 Learning Algorithm and Pretraining

For the learning algorithm we can use the Deep K-Means algorithm with pretraining (algorithm 2) see in section 2.3.

Pretraining
The pretraining we performed here simply consists in initializing the weights by training the auto-encoder then, petraining allows to initialize centroïds. We use two methods for pretraining :

1. **simple pretrain** : minimizing only reconstruct loss : $\sum_{X \in X} ||X - f(h_\theta(X))||_2^2$. It allows to initialize with the least loss of information about documents.
2. **lexical pretrain** : minimizing

$$\sum_{X \in X} ||X - f(h_\theta(X))||_2^2 + \lambda_1 \sum_{X \in C} \sum_{k=1}^{K} \frac{e^{-\alpha||h_\theta(X) - h_\theta(X'_k)||_2^2}}{\sum_{k'=1}^{K} e^{-\alpha||h_\theta(X) - h_\theta(X'_{k'})||_2^2}} ||h_\theta(X) - h_\theta(X'_k)||_2^2$$

   It allows to initialize centroïds with bias representation.

Centers Initialization
After pretraining we need to initialize centers. We denote $S^{(k)}$ as follows :

$$\forall i, S_i^{(k)} = \begin{cases} log\left(\frac{N}{df(i)}\right) & \text{if } i \in KW \\ 0 & \text{Otherwise} \end{cases} \tag{23}$$

Then we initialize R as follows :

$$\forall k, r_k = h_\theta(S^{(k)}) \tag{24}$$

4 Experiment

4.1 Data

To experiment our algorithm we use the dataset 20Newsgroups [1]. The 20 Newsgroups data set is a collection of approximately 20,000 Newsgroups documents, partitioned evenly across 20 different newsgroups. We use also the RCV1 dataset [11]. The RCV1 dataset is a collection of over 800,000 text documents.

For RCV1 dataset, we use only a subset of 10,000 documents from RCV1 such that each document belongs to only one of the root classes in the class hierarchy. This was detailed in [[4]].
Each document are represented by a vector using term frequency-inverse document frequency (TFIDF) representation [9]. The term frequency-inverse document frequency is a method of weighting depicting the significance of each word of a document in relative to a corpus.

$$TF(t, X) = \frac{f_{t,X}}{max_{t' \in C} f_{t',X}} \quad (25)$$

$$IDF(t, C) = log(\frac{N}{|X \in C : t \in X|}) \quad (26)$$

$$TFIDF(t, X, C) = TF(t, X).IDF(t, C) \quad (27)$$

For each dataset there is a preprocessing step. We remove stopword and keep only the 2000 words with the top TFIDF scores. We use also a stemming step [14].

4.2 Keywords Extraction

To generate the set of keywords KW we rank each word of each document of each classes using TFIDF according to algorithm 3. We add for each document of each classes the TFIDF of each word, in this manner we find the most important word of each class. Furthermore, for discriminative keywords, we substract the TFIDF of other class, so that the keywords are the most discriminating.

```
input : Corpus C, The number of keywords per classes $P$
output: KW
$KW \leftarrow \{\}$
foreach Class $c_i \in C$ do
    $rank_i \leftarrow$ [0...0]
    foreach Document $X \in c_i$ do
        foreach Word $w \in X$ do
            $rank_{i,w} \leftarrow rank_{i,w} + TFIDF(w, X, C)$
        end
    end
end
if Discriminating_Extraction then
    foreach Class $c_i \in C$ do
        $rank'_i \leftarrow rank_i - \sum_{\forall c_j, c_j \neq c_i} rank_j$
    end
    $rank \leftarrow rank'$
end
foreach Class $c_i \in C$ do
    $KW \leftarrow KW \cup \{\{w_1, w_2...w_P\} : \nexists (v_1, v_2) | v_1 \notin \{w_1, w_2...w_P\}, v_2 \in \{w_1, w_2...w_P\}, rank_{i,v_1} \geq rank_{i,v_2}\}$
end
return $KW$
```

Algorithm 3: Extract Keywords

4.3 Evaluation

Baseline Algorithm

As our goal in this work is to study the k-Means clustering algorithm with constraints, we focus on the family of K-Means-related models and compare our approach against state-of-the-art models from this family, using both standard and deep clustering models. For the standard clustering methods, we used: the K-Means clustering approach, denoted **KM**; an approach denoted as **AE-KM** in which dimensional reduction is first performed using an auto-encoder followed by K-Means applied to the learned representations. For AE-KM, we can use only the reconstruct loss (denoted **AE-KM SP**) or integrate lexical constraints loss (denoted **AE-KM LP**).
For the deep clustering models, we use the Deep K-Means Model see in section 2.3 denoted **DKM** with pretraining.

Metric

To evaluate our algorithm and compare results with reference algorithms we can use the NMI Metric, Accuracy Metric [3], and Adjusted Rand index[20].

- NMI is an information-theoretic measure based on the mutual information of the ground-truth classes and the obtained clusters, normalized using the entropy of each. The NMI Metric is defined as follows

$$NMI(S, C) = \frac{I(S, C)}{[H(S) + H(C)]/2}$$

  with $I(S, C) = \sum_k \sum_f \frac{|s_k \cap c_f|}{N} log \frac{N|s_k \cap c_f|}{|s_k||c_f|}$ and $H(S) = -\sum_k \frac{|s_k|}{N} log \frac{N|s_k|}{|s_k|}$

- The Accuracy is the proportion of true results among the total number of cases examined. The Accuracy metric is defined as follows :

$$ACC(S, C) = \frac{1}{N} \sum_k max_j |s_k \cap c_j|$$

- Let a be the number of pairs of document in C that are in the same cluster in the predicted partition and in the same cluster in the real partition, and b be the number of pairs of document in C that are in different clusters in predicted partition and in different cluster in real partition. The Adjusted Rand index is defined as follows :

$$ARI = \frac{a + b}{\binom{N}{2}}$$

4.4 Experimental Setup

Autoencoder Architecture

We use the same architecture used in [4]. The encoder is a fully-connected multilayer perceptron formed by 3 hidden layers (with dimensions 500, 500, 2000) and an embedding layer (with dimension K, the number of cluster). The decoder is a mirrored version of the encoder 2. All layers except the one preceding the embedding layer and the one preceding the output layer are applied a ReLU activation function [16] before being fed to the next layer. For the layer preceding the embedding layer and for the layer preceding the output layer we apply the identity function.

Experimental Protocol

The purpose of the experiment is to rediscover the different classes of datasets with keywords.
To add noise to 20 newsgroups dataset we divide the dataset into two corpus C_1, C_2. Each corpus contains ten classes. We generate keywords 3 from C_1. Then we concatenate document from corpus C_1 with document from corpus C_2. The clustering processed on corpus C_1. The tests were carried out in 3 steps :

1. Discriminative Keywords : We test our algorithm with 3 discriminative keywords (see algorithm 3).
2. Non Discriminative Keywords : We test our algorithm with 3 non discriminative keywords (see algorithm 3).
3. Robustness : We test the robustness of our algorithm. To test this, we can vary the number of keywords by classes. In addition we compare the results of each version of CDKM. We generate discriminative keywords with algorithm 3 for this test.

For all tests we select hyperparameters with 3 discriminative keywords per classes see section 4.5.

Hyperparameters Selection

The hyperparameters λ_0 and λ_1 , that define the trade-off between the lexical constraint error and clustering error in the loss function, were determined by performing a grid search on the set $\{10^i | i \in [-6, -1]\}$. To do so, we randomly split each dataset into a validation set (10% of the data) and a test set (90%). Each model is trained on the whole data and only the validation set labels are leveraged in the grid search to identify the optimal λ_0 and λ_1.
We select hyperparameters which maximize the Accuracy Metrics for Validation Set. The results of Grid Search are reported in table 1.

Table 1: Best results of Grad Search for the optimization of hyperparameters for each dataset.

	λ_0	λ_1
Deep K-Means	10^{-2}	▨
AE + KM, LP mask	▨	10^{-4}
AE + KM, LP sim	▨	10^{-2}
CDKM, LP mask	10^{-1}	10^{-3}
CDKM, LP sim	10^{-1}	10^{-3}
CDKM, SP mask	10^{-1}	10^{-3}
CDKM, SP sim	10^{-1}	10^{-2}

RCV1

	λ_0	λ_1
Deep K-Means	10^{-1}	▨
AE + KM, LP mask	▨	10^{-2}
AE + KM, LP sim	▨	10^{-2}
CDKM, LP mask	10^{-6}	10^{-1}
CDKM, LP sim	10^{-3}	10^{-1}
CDKM, SP mask	10^{-1}	10^{-6}
CDKM, SP sim	10^{-1}	10^{-5}

20 Newsgroups

	λ_0	λ_1
Deep K-Means	10^{-1}	▨
AE + KM, LP mask	▨	10^{-2}
AE + KM, LP sim	▨	10^{-2}
CDKM, LP mask	10^{-1}	10^{-3}
CDKM, LP sim	10^{-1}	10^{-3}
CDKM, SP mask	10^{-1}	10^{-4}
CDKM, SP sim	10^{-1}	10^{-5}

20 Newsgroups without noise

4.5 Results

To present the tests, we denote **CDKM MASK** (respectively **CDKM SIM**) the version of the algorithm using masked document (respectively similarity function), and **LP** (respectively **SP**) when we use the Lexical Pretrain (respectively Simple Pretrain).
The results for the evaluation of the compared clustering methods on the different benchmark datasets are summarized in tables 2 3 and in figures 4. The clustering performance is evaluated with respect to three standard measures Normalized Mutual Information (NMI), the Adjusted Rand Index (ari) and the clustering accuracy (ACC) see in section 4.3. We report for each dataset/method pair the average and standard deviation of these metrics computed over 10 runs and conduct significance testing as described in section 4.4. Bold Result in each column of tables 2 3 corresponds to the best result for the corresponding method/metric.

Discriminative Keywords

Results are reported in table 2.
We can observe that cdkm is always better than baselines, whatever the method to define X′ and the type of pretraining. We also observe that Lexical Pretrain is generally better than Simple Pretrain. Finally, the results for the mask method and for the sim method are similar.
The strongest progression is for the dataset RCV1, and the weakest progression is for the dataset 20 newsgroup with noise.
While observing results for **AE+KM mask LP** and **AE+KM sim LP**, we notice that lexical biases give better results than K-Means or Deep K-Means.

Non Discriminative Keywords

Results are reported in table 3.
We can observe that cdkm is always better than baselines, whatever the method to define X′ and the type of pretraining.

While observing results for **AE+KM mask LP** and **AE+KM sim LP**, we notice that lexical biases give better results than *K*-Means or Deep *K*-Means. It is with the method sim that we have the lowest performance decrease.
We also notice that both types of pretraining have similar performance.

Robustness

We can see results in figures 4.
We observe that when we use only one keyword the results are bad for each metrics.
We observe that the sim method is more stable. Indeed, from two keywords, performances increase. While for the mask method we see a peak for 3 keywords, then performances decrease. Moreover, we note that for the sim method, the two methods of pretraining have fairly similar results. While for the mask method there is a big difference in terms of performance.

5 Conclusion

We have presented and tested in this study an approach to integrate lexical constraints to the *K*-Means algorithm by introducing in the non clustering loss penalties biasing the representation. To the best of our knowledge, this is the first approach that integrates lexical constraints to the *K*-Means algorithm.

6 Future Work

In future work, we plan to add priorities to classes In other words, we would like some classes to be more important than others, to bias clustering to some classes considered more important by the user.
Also, because users do not always know the entire corpus of documents, it might be useful to add a trash class system that groups documents that have no links to the user's given keyword classes.
In addition, the initialization of the centers is only taking into account keywords, we could use a similarity function and initialize the centers with the words closest to the keywords in the corpus.
Moreover, our tests do not allow us to know if the performance gain is due to lexical biases or centroid initialization. Indeed, for Deep *K*-Means algorithm, centroid are initialized after the pretraining with *K*-Means++ algorithm. It could be interesting to add two baselines :

1. **DKM LP** : We could test a version of the DKM algorithm where pretraining would take lexical bias into account. In other words, we could use the lexical pretrain with Deep *K*-Means algorithm. For this test, we continue to use the *K*-Means++ algorithm for the centroid initialization.

2. **DKM SI** : We could test a version of the DKM algorithm the centroid initialization defines in section 3.3. For this test, we continue to use the petraining defined in [4] for Deep *K*-Means algorithm.

Table 2: Clustering results for *K*-Means applies to different learned latent space to measure the efficiency of lexical constraints for *K*-Means algorithm. Performance is measured in terms of NMI, Adjusted Rand Index and clustering Accuracy, higher is better. Each cell contains the average and the standard deviation computed over 10 runs.

	ACC	ARI	NMI
K-Means	48.8 ± 6.6	18.4 ± 6.0	29.7 ± 5.8
AE + KM, SP	51.3 ± 3.5	17.5 ± 5.8	24.5 ± 5.1
Deep *K*-Means	54.4 ± 4.9	23.9 ± 3.5	29.6 ± 3.6
AE + KM, LP mask	65.0 ± 6.7	35.6 ± 7.3	37.5 ± 6.5
AE + KM, LP sim	62.7 ± 6.1	35.1 ± 5.7	38.4 ± 4.1
CDKM, LP mask	$\mathbf{72.7 \pm 4.0}$	$\mathbf{43.8 \pm 5.4}$	$\mathbf{44.1 \pm 3.9}$
CDKM, LP sim	72.5 ± 4.5	43.7 ± 5.7	44.0 ± 4.3
CDKM, SP mask	70.3 ± 4.2	41.0 ± 5.1	42.1 ± 4.0
CDKM, SP sim	70.4 ± 5.1	41.8 ± 7.1	43.0 ± 5.2

RCV1

	ACC	ARI	NMI
K-Means	36.1 ± 2.2	13.3 ± 1.7	40.9 ± 1.6
AE + KM, SP	53.1 ± 2.3	35.0 ± 1.4	49.3 ± 1.0
Deep *K*-Means	54.9 ± 1.7	37.6 ± 1.1	51.6 ± 0.6
AE + KM, LP mask	56.8 ± 1.7	38.4 ± 1.1	51.6 ± 0.6
AE + KM, LP sim	56.0 ± 2.3	37.6 ± 1.6	50.8 ± 0.7
CDKM, LP mask	$\mathbf{61.3 \pm 0.6}$	$\mathbf{41.2 \pm 0.8}$	52.7 ± 0.5
CDKM, LP sim	60.6 ± 1.3	40.5 ± 1.3	53.0 ± 0.8
CDKM, SP mask	60.2 ± 1.8	$\mathbf{41.2 \pm 0.9}$	$\mathbf{53.5 \pm 0.5}$
CDKM, SP sim	60.5 ± 1.2	40.7 ± 0.8	52.9 ± 0.5

20 Newsgroups

	ACC	ARI	NMI
K-Means	33.1 ± 2.7	8.7 ± 1.3	26.3 ± 1.6
AE + KM, SP	45.2 ± 3.3	23.0 ± 2.5	30.0 ± 2.0
Deep *K*-Means	44.5 ± 2.4	23.6 ± 2.4	30.2 ± 2.0
AE + KM, LP mask	45.9 ± 2.2	24.0 ± 1.8	31.8 ± 1.5
AE + KM, LP sim	46.2 ± 2.6	24.6 ± 1.9	32.3 ± 1.3
CDKM, LP mask	50.0 ± 2.1	26.0 ± 1.6	31.7 ± 1.7
CDKM, LP sim	49.4 ± 2.1	25.3 ± 1.9	31.1 ± 1.7
CDKM, SP mask	49.5 ± 1.7	26.1 ± 1.1	32.0 ± 0.9
CDKM, SP sim	49.3 ± 1.1	25.9 ± 1.2	32.0 ± 1.1

20 Newsgroups with noise

Finally, one of the major problems of our algorithm is that we cannot use as keywords only words present in the corpus, nevertheless it is possible that the user wants to give keywords that are not part of corpus. To do this, we could use words embedding [15]. Word embedding is a method that focuses on learning a representation of words. This technique allows to represent each word of a dictionary by a corresponding real number vector. This facilitates the semantic analysis of words. This new representation is unique in that words appearing in similar contexts have corresponding vectors that are relatively close.

Table 3: Clustering results for *K*-Means applies applies to different learned latent space to measure the efficiency of lexical constraints for *K*-Means algorithm. Performance is measured in terms of NMI, Adjusted Rand Index and clustering Accuracy, higher is better. Each cell contains the average and the standard deviation computed over 10 runs.

	ACC	ARI	NMI
K-Means	48.8 ± 6.6	18.4 ± 6.0	29.7 ± 5.8
AE + KM, SP	51.3 ± 3.5	17.5 ± 5.8	24.5 ± 5.1
Deep *K*-Means	54.4 ± 4.9	23.9 ± 3.5	29.6 ± 3.6
AE + KM, LP mask	66.1 ± 5.0	38.2 ± 5.3	40.5 ± 4.3
AE + KM, LP sim	62.3 ± 4.8	35.2 ± 5.0	38.7 ± 3.5
CDKM, LP mask	$\mathbf{68.5 \pm 5.6}$	$\mathbf{39.6 \pm 6.4}$	$\mathbf{41.3 \pm 4.6}$
CDKM, LP sim	65.8 ± 5.0	36.2 ± 5.5	38.9 ± 3.7
CDKM, SP mask	68.5 ± 5.6	39.6 ± 6.4	41.3 ± 4.6
CDKM, SP sim	67.5 ± 6.3	38.6 ± 6.4	40.7 ± 4.4

RCV1

	ACC	ARI	NMI
K-Means	36.1 ± 2.2	13.3 ± 1.7	40.9 ± 1.6
AE + KM, SP	53.1 ± 2.3	35.0 ± 1.4	49.3 ± 1.0
Deep *K*-Means	54.9 ± 1.7	37.6 ± 1.1	51.6 ± 0.6
AE + KM, LP mask	57.5 ± 2.8	39.6 ± 1.7	51.6 ± 1.4
AE + KM, LP sim	55.7 ± 1.7	37.6 ± 1.0	50.8 ± 0.4
CDKM, LP mask	58.8 ± 1.4	38.5 ± 1.4	52.6 ± 0.6
CDKM, LP sim	$\mathbf{60.1 \pm 2.1}$	$\mathbf{40.2 \pm 1.4}$	$\mathbf{52.8 \pm 1.0}$
CDKM, SP mask	58.9 ± 2.0	38.9 ± 1.6	$\mathbf{52.8 \pm 0.9}$
CDKM, SP sim	60.1 ± 1.7	40.1 ± 1.5	52.7 ± 0.7

20 Newsgroups

	ACC	ARI	NMI
K-Means	33.1 ± 2.7	8.7 ± 1.3	26.3 ± 1.6
AE + KM, SP	45.2 ± 3.3	23.0 ± 2.5	30.0 ± 2.0
Deep *K*-Means	44.5 ± 2.4	23.6 ± 2.4	30.2 ± 2.0
AE + KM, LP mask	45.2 ± 2.2	24.2 ± 1.6	32.4 ± 1.1
AE + KM, LP sim	42.3 ± 2.4	21.7 ± 2.0	29.1 ± 1.3
CDKM, LP mask	48.8 ± 2.3	$\mathbf{25.7 \pm 1.9}$	$\mathbf{31.6 \pm 1.9}$
CDKM, LP sim	47.8 ± 2.6	24.3 ± 1.8	30.8 ± 1.7
CDKM, SP mask	48.8 ± 2.3	25.7 ± 1.9	31.6 ± 1.9
CDKM, SP sim	$\mathbf{49.2 \pm 2.3}$	25.2 ± 2.0	31.5 ± 1.7

20 Newsgroups with noise

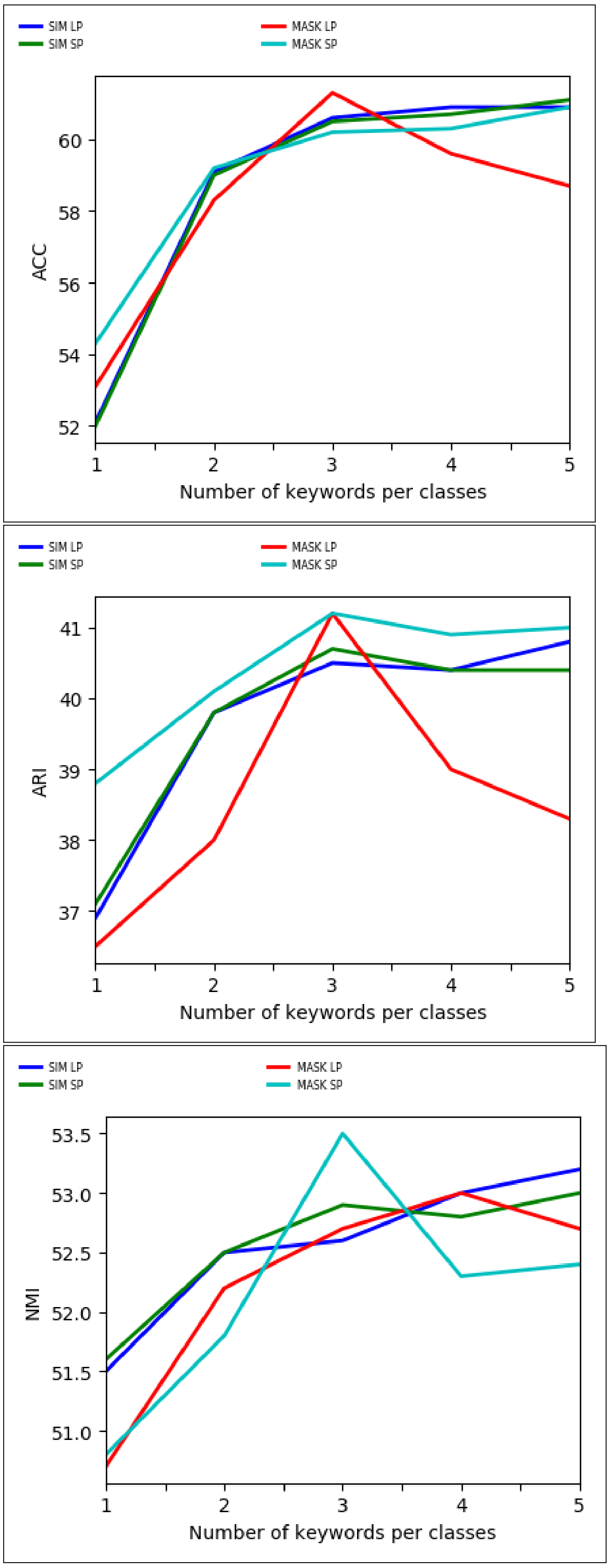


Figure 4: Results for 20 newsgroups dataset

References

[1] *20 Newsgroups Dataset*.

[2] E. Aljalbout et al. "Clustering with Deep Learning: Taxonomy and New Methods". In: *ArXiv e-prints* (Jan. 2018). arXiv: `1801.07648 [cs.LG]`.

[3] Deng Cai, Xiaofei He, and Jiawei Han. "Locally consistent concept factorization for document clustering". English (US). In: *IEEE Transactions on Knowledge and Data Engineering* 23.6 (Mar. 2011), pp. 902–913. ISSN: 1041-4347.

[4] Maziar Moradi Fard, Thibaut Thonet, and Eric Gaussier. *Deep K-Means : An End-to-End, Annealing-based Approach for jointly Clustering with K-Means and Learning Representations*. Submitted to 32nd Conference on Neural Information Processing Systems (NIPS 2018).

[5] Thomas S. Ferguson. "An Inconsistent Maximum Likelihood Estimate". In: *Journal of the American Statistical Association* 77.380 (1982), pp. 831–834.

[6] Ronald A. Fisher. *UCI Machine Learning Repository: Iris Data Set*. Jan. 2011.

[7] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[8] Y.-C. Hsu and Z. Kira. "Neural network-based clustering using pairwise constraints". In: *ArXiv e-prints* (Nov. 2015). arXiv: `1511.06321 [cs.LG]`.

[9] KAREN SPARCK JONES. "A STATISTICAL INTERPRETATION OF TERM SPECIFICITY AND ITS APPLICATION IN RETRIEVAL". In: *Journal of Documentation* 28.1 (1972), pp. 11–21.

[10] S. Kullback and R. A. Leibler. "On Information and Sufficiency". In: *Ann. Math. Statist.* 22.1 (Mar. 1951), pp. 79–86.

[11] David D. Lewis et al. "RCV1: A New Benchmark Collection for Text Categorization Research". In: *J. Mach. Learn. Res.* 5 (Dec. 2004), pp. 361–397. ISSN: 1532-4435.

[12] Chenliang Li et al. "Effective Document Labeling with Very Few Seed Words: A Topic Model Approach". In: *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*. CIKM '16. Indianapolis, Indiana, USA: ACM, 2016, pp. 85–94. ISBN: 978-1-4503-4073-1.

[13] S. Lloyd. "Least squares quantization in PCM". In: *IEEE Transactions on Information Theory* 28.2 (Mar. 1982), pp. 129–137. ISSN: 0018-9448.

[14] Julie Beth Lovins. "Development of a stemming algorithm." In: *Mech. Translat. and Comp. Linguistics* 11.1-2 (1968), pp. 22–31.

[15] T. Mikolov et al. "Efficient Estimation of Word Representations in Vector Space". In: *ArXiv e-prints* (Jan. 2013). arXiv: `1301.3781 [cs.CL]`.

[16] Vinod Nair and Geoffrey E. Hinton. "Rectified Linear Units Improve Restricted Boltzmann Machines". In: *Proceedings of the 27th International Conference on International Conference on Machine Learning*. ICML'10. Haifa, Israel: Omnipress, 2010, pp. 807–814. ISBN: 978-1-60558-907-7.

[17] Nasser M. Nasrabadi. "Pattern Recognition and Machine Learning". In: *Journal of Electronic Imaging* 16 (2007).

[18] Lorenzo Rosasco et al. "Are Loss Functions All the Same?" In: *Neural Computation* 16.5 (2004), pp. 1063–1076.

[19] W.H. Swann. "A survey of non-linear optimization techniques". In: *FEBS Letters* 2 (1969), S39–S55. ISSN: 0014-5793.

[20] Nguyen Xuan Vinh, Julien Epps, and James Bailey. "Information Theoretic Measures for Clusterings Comparison: Variants, Properties, Normalization and Correction for Chance". In: *J. Mach. Learn. Res.* 11 (Dec. 2010), pp. 2837–2854. ISSN: 1532-4435.

[21] Kiri Wagstaff et al. "Constrained K-means Clustering with Background Knowledge". In: *Proceedings of the Eighteenth International Conference on Machine Learning*. ICML '01. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001, pp. 577–584. ISBN: 1-55860-778-1.

A System-Wide Study of Performance Issues in FaaS Platforms
Magistère M1 Report

Christopher Ferreira
LIG, ERODS Team

Supervised by: **Renaud Lachaize** and **Vivien Quéma**
With the help of: **Hugo Guiroux**

Abstract

The Function as a Service (FaaS) paradigm introduces a new programming model that reduces the size of the execution unit managed by a Cloud provider's infrastructure to its bare minimum: the request. As a result, the resource management logic of a FaaS infrastructure is involved in each and every request, exacerbating its overhead. Moreover, strong evidence shows that a significant portion of these Cloud functions are likely to have low execution times, in the order of 10–100 milliseconds. Overall, the FaaS model introduces extremely demanding requirements for the underlying software stack in terms of efficiency.

This Master thesis aims at studying the behavior of the whole software stack of FaaS infrastructures, with the end goal of improving their performance and resource usage. More precisely, our work aims at identifying the main performance bottlenecks within an existing FaaS platform, with a particular interest for the software stack of the machines that execute the Cloud functions. We describe three main performance bottlenecks identified in the OpenWhisk architecture: (i) garbage collection within the internal components of the infrastructure, (ii) scheduling interferences between the Linux kernel, the container subsystem and the FaaS middleware, (iii) protocols used to interact with the containers hosting the user functions. Additionally our study reveals some of the challenges that exist for the understanding of the root causes of performance issues in today's Cloud infrastructures. In particular, we highlight that the heterogeneity and isolation mechanisms of these infrastructures reveal some limitations of existing profiling tools for such situations. Finally, we discuss possible mitigation strategies for the above-mentioned bottlenecks.

1 Introduction

One of the most recent developments of Cloud computing platforms is the introduction of so-called *serverless* services. In the present work, we focus on the most popular Serverless service: *Functions as a Service (FaaS)* platorms. Since the release of Amazon Web Service (AWS) Lambda [55] in 2014 (joined notably by Microsoft Azure Functions [56], Google Cloud Functions [57] and IBM Cloud Functions [58] in 2016), FaaS platforms have gained significant traction in the industry [1] and are raising the interest of the research community [2, 3, 4, 5, 6].

1.1 Functions as a Service

Essentially, the term "serverless" in "serverless computing" refers to the lack of server management duties from the point of view of the user. For FaaS platforms, this means that users do not upload their applications as virtual machines or containers running longstanding servers (as in Infrastructure as a Service (IaaS) and Containers as a Service (CaaS)) but as simple functions[1] written using one of the supported languages and their API (for example AWS Lambda supports Java, Javascript, C#, Python and Go). The serverless/FaaS computing model shares many similarities with Platform as a Service (PaaS), however current PaaS offerings still operate in a longstanding server fashion while functions in the FaaS model are designed to be ephemeral — quickly spawned and dismissed. Besides, the billing principles of FaaS platforms are designed to closely follow the actual resource consumption: customers are billed with a fixed cost for each execution of the function and an additional cost proportional to the duration of the function (as an example, for an AWS Lambda function having its memory consumption capped to 1GB, a customer pays \$0.20 for each million of invocations and \$0.00001667 per second accounted from the start of the execution of each function to its end). This proportional cost is usually billed at a granularity of 100 milliseconds, which can result in much less idle time being charged, in contrast to other models for which the granularity is much coarser (a typical order of magnitude is one hour [35]).

FaaS platforms hide away most of the system stack (hardware, Operating System (OS), virtualization), leaving only the language runtime exposed and the application layer to the hands of the user. This trade-off of convenience over

[1]These functions have different names in different contexts: they are named *lambdas* in the AWS ecosystem, *actions* with IBM Cloud Functions and simply *functions* or *Cloud functions* in other contexts.

flexibility and control is easily justified: server configuration and management is a well-known pain point. Setting up and maintaining a custom software stack (e.g., addressing performance, fault tolerance and security issues) requires a non-negligible amount of time and expertise (these software stacks generally offer a large set of configuration options for performance tuning and regularly require to be updated with security fixes). With FaaS platforms, most of the server operations are delegated to the Cloud provider. Additionally, elasticity – the capacity of an online service to quickly allocate and deallocate resources to meet the variation of service demands – is no longer a concern. Effectively, the elasticity of the service is also entirely delegated to the Cloud provider: the Cloud provider automatically allocates and dellocates new resources for the functions when the service demand increases and decreases.

Characteristics & Use Cases

Cloud functions are attached to events that trigger their execution. For example a function can be executed as the handler of an HTTP request with its return value being used as the HTTP response. As another example, when a new entry has been added to a database table, a function can be consequently called and this function can propagate the new entry to a search engine indexer. The functions written for FaaS platforms are *stateless*: the platform offers no guarantee that a global state is shared from one execution of the function to the next execution. Consequently applications written with these functions tend to rely heavily on other Cloud services, i.e., typically the functions fetch and store data from and to databases or other Cloud storage services.

There are typically only a few configuration options available for these functions: (i) the maximum execution time (ranging from 1s to 5 minutes on AWS Lambda) and (ii) the maximum amount of memory available (from 128MB up to 3GB). Other options like CPU frequency and number of CPUs are automatically derived from the memory cap: with AWS Lambda, if the allocated memory is greater or equal to 1.5GB, the function has access to a second CPU.

Given the rapid pace of innovation of the Web/mobile software industry, the FaaS model is an appealing solution for consumers thanks to its lower time-to-market. FaaS platforms can be perceived as a response from the Cloud providers to the move from monolithic applications towards *microservice architectures* [36]. In a microservice architecture, instead of a single monolithic application, the application is written as a set of small dedicated services running independently and cooperating via network communications (usually using HTTP, with each microservice exposing an API). These kinds of architectures have gained popularity because, among others reasons, they simplify the process of scaling the application (both on a day-to-day basis to adapt to demand variations and in the long term, when a service gains popularity) when only one component actually needs more resources. FaaS platforms are a good fit for the deployment of such microservice-based applications, for example each microservice can be deployed as its own function and transparently benefits from the automatic elasticity featured by the FaaS platform. Additionally, the FaaS model simplifies the process of setting up a service on the Cloud, potentially allowing a new type of users to access the computing facilities of the Cloud without the complexity of server operations. For example, Jonas et al. [7] show that FaaS platforms can be used to run a variety of large-scale data processing computations, using the Map Reduce model, in a simpler manner than what is currently offered by existing solutions, which require to set up and configure a complex data processing system stack. Another example is ExCamera [8], which implements a highly parallel video transcoder by invoking the same function more than a thousand times in parallel .

Server Consolidation

The FaaS model gives Cloud providers more control over the management of their resources: it offers a new opportunity to maximize server consolidation, i.e., maximize utilization of the machines of their data centers. One of the main reasons that explains the widespread adoption of Cloud computing is the cost-effectiveness of its model. In contrast to the use of dedicated hardware, Cloud computing enables the physical resources to be shared between consumers (i.e., multi-tenancy) and to remain at a higher utilization (server consolidation). Given that the expenses related to data centers contain a non-negligible portion of fixed costs (i.e., initial purchase of hardware) and that energy consumption is non-proportional to the utilization (higher utilization have better energy efficiency – energy efficiency being defined as the energy consumption divided by the utilization [9]), Cloud computing allows a better mitigation of these expenses. However, most data centers report an average utilization that is sub-optimal (typically between 20% and 40%) [10, 11]. This generally low utilization leaves room for further improvement towards cost-effectiveness. A sensible target utilization would be to have an average of server utilization around 70%-80% which is closer to the maximum cost-efficiency while leaving some room to handle unexpected bursts . Increasing the utilization requires an infrastructure that is able to balance the workload on the machines intelligently and adapt to their variations over time. FaaS offers a good opportunity to tackle this issue. In contrast to virtual machines and containers migrations which are costly and complex operations , functions, due to their ephemeral and stateless nature, can be executed virtually anywhere in the data center. This offers more flexibility for optimizations in load balancing and server consolidation leading to a better utilization of the physical resources of the data center.

Cold Start

Conversely this new model creates a new challenge. The overhead incurred before starting a function should be kept low as it would directly impact every function request. Current implementations of FaaS platforms rely internally on operating system containers to execute the function. Even though container startup is considered lightweight in comparison to the same operations for a virtual machine, this startup time (around 100ms for a lightweight container [12]) is high in proportion for a function that may run for a duration of the same order of magnitude. In order to address this issue, FaaS platforms reuse already running containers for further executions of the same functions. Consequently, this introduces two types of function startup profiles, a *cold start* when the container has to be started from scratch and a *warm start* for functions that are executed on an already available container.

1.2 Problem Statement

The FaaS model introduces a new programming model that reduces the size of the execution unit managed by the Cloud provider's infrastructure, from the long-standing application server in the IaaS model to its bare minimum: the request. As a result, the resource management logic (load balancing, resource pooling) of the FaaS infrastructure is involved in each and every request, exacerbating its overhead. Moreover, strong evidence suggests [2] that a significant portion of these Cloud functions are likely to have low execution times in the order of 10–100 milliseconds (or even below 1 ms if a function acts as a basic gateway to a simple service like a key-value store). Such a low order of magnitude further exacerbates the overhead of the FaaS platform. Overall, the FaaS model introduces extremely demanding requirements for the underlying software stack in terms of efficiency.

Besides, most of the existing FaaS platforms have been developed by the Cloud providers in order to provide their customers/users with a functional substrate allowing them to develop and host simple and cost-effective "glue code" in order to connect the different Cloud services that they use (e.g. Web frontends, storage backends, data analytics pipelines). Most of the publicly available information about these industrial platforms is related to their functional aspects and use cases rather than their internal design and performance optimizations. Furthermore, beyond the specific considerations regarding the available FaaS platforms operated by commercial Cloud providers, there is to date (to the best of our knowledge) no study that provides a detailed analysis of the performance challenges within the software layers of a FaaS infrastructure. Such insights can be of interest both for the scientific community (operating system researchers) and the designers and maintainers of FaaS platforms used in production. Arguably, we can draw a parallel with the history of the system-level infrastructures that support the World Wide Web: FaaS platforms are still in their infancy and achieving highly efficient operating conditions for such complex infrastructures will likely require years of research and performance engineering efforts.

In the above-described context, this Master thesis aims at studying the behavior of the whole software stack of FaaS infrastructures, with the end goal of improving their performance and resource usage. More precisely, our work aims at identifying the main performance bottlenecks within an existing FaaS platform (OpenWhisk, the main open-source FaaS platform to date), with a particular interest for the software stack of the machines that execute the Cloud functions. Our study focuses on performance in terms of throughput and latency[2]. We study the impact of the software infrastructure at different levels, from the components in charge of request routing and resource allocation and the language runtimes running the applications to the Linux kernel and its container abstraction. Additionally our study reveals some of the challenges that exist for the understanding of the root causes of performance issues in today's Cloud infrastructures. In particular, we highlight that the heterogeneity (i.e., mixture of various programming languages, language runtimes, concurrency models) and mechanisms of isolation of these infrastructures (i) introduce complex interactions between software components that can degrade performance in subtle, yet significant ways, and (ii) reveal some limitations of existing profiling tools for such situations. Our contributions can be summarized as follows:

- We highlight several profiling challenges that are introduced by the characteristics of modern Cloud infrastructures and describe how we have overcome them;
- We describe three main performance bottlenecks identified in the OpenWhisk architecture: (i) Garbage collection within the internal components of the infrastructure, (ii) scheduling interferences between the Linux kernel, the container subsystem and the FaaS middleware, (iii) protocols used to interact with the containers hosting the user functions;
- We discuss possible mitigation strategies for the above-mentioned bottlenecks;

1.3 Contents of the Report

The remainder of this document is organized as follows:

- Chapter 2 introduces and discusses related works.
- Chapter 3 describes the OpenWhisk platform, which is the focus of our empirical case study.
- Chapter 4 describes our experimental methodology.
- Chapter 5 describes the results of our study.
- Chapter 6 concludes this report and discusses future work.

2 Related Work

The FaaS model is still relatively new and the research community is only starting to tackle the new challenges that it introduces. We discuss these challenges in Section 2.1. Section 2.2 discusses more general contributions related to the performance of Cloud infrastructures.

2.1 Research Challenges of the FaaS Model

Hendrickson et al. [2] perform an early study of FaaS platforms with the expressed intent to discuss a research agenda for these platforms. The authors compensate for the absence of public FaaS workloads by providing an ad-hoc empirical evaluation of the characteristics of the requests generated by the Google Gmail Web client. This empirical evaluation notably highlights that a significant portion of the requests generated by the Gmail client have an execution time that is less than 100 milliseconds; Cloud functions are likely to have the same range of execution times or even lower execution times for some workloads. The authors then discuss their vision of the research challenges introduced by FaaS platforms.

[2]Energy efficiency, while a growing and important concern, is left for future work.

The first of these challenges is the startup overhead of the container engine. FaaS platforms currently leverage containers for the execution (in an isolated environment) of functions. Unfortunately, current container runtimes have large startup times, in the order of hundreds of milliseconds. To mitigate this issue, current platforms pause and reuse already started containers for the following executions of the same functions. However, both running and paused containers consume memory space, thus the maximum number of running and paused containers that a machine can keep is limited. Therefore, improving the startup time of containers or decreasing their memory sizes are both means to reduce the overhead of the container engine. As an illustration, Oakes et al. [13] propose PipSqueak, a system that combines prestarted containers with a python libraries cache to reduce startup time.

Other works discuss the same issue, but instead, propose alternative solutions to container engines to remove their overhead. Given the relatively large startup time of containers, Koller and Williams [12] state that the container abstractions of the Linux kernel are simply inadequate for FaaS workloads, and that it is preferable to introduce new abstractions designed for this context. Furthermore, they claim that implementing these new abstractions directly in the kernel would be counter-productive as it would increase the kernel code complexity (which is already a concern both for performance and safety/security reasons), mentioning as an example the pervasive modifications that have been required by the introduction of container-related mechanisms in the recent past. To support their claim, they show that it is possible to obtain faster startup times while conserving isolation guarantees by using a lightweight virtual machine manager running a unikernel (i.e., an operating system kernel stripped-down to the only necessary components for the execution of one single application). In a similar fashion, Manco et al. [14] propose *LightVM*, a version of the Xen virtual machine manager that is optimized for the executions of a large number of concurrent virtual machines running unikernels. The authors validate their approach by showcasing that *LightVM* can be used to design a new FaaS platform. In this design, functions are executed in virtual machines running unikernels enabling startup times that are less than 10 milliseconds. However, in both cases, the use of unikernels introduces new challenges. Unikernels are fast to boot because their are written specifically for their use cases and contains only what is necessary for the application, as a consequence they are complex to create and maintain. For a similar reason, debugging support on unikernels is usually problematic [37]. Furthermore it is not clear how these solutions could be integrated in existing Cloud infrastructure where machines usually run a mixed set of workloads (virtual machines of IaaS services, containers of CaaS services, and functions of FaaS)[3].

Another interesting approach to replace container engines for function execution is to implement the isolation at the the level of the language runtime. With this approach, only one language runtime would have to run on a machine and would process all requests assigned to the machine. However, this solution also introduces new challenges, e.g., (i) the language runtime should support multiple programming languages to have feature-parity with current platforms, (ii) it is not clear how to implement performance isolation at the level of the language runtime. We know of at least two initiatives working towards this kind of solutions [6, 15].

Other challenges highlighted by Hendrickson et al. include the reuse of JIT information across containers running the same functions (for language runtimes like Java and Node.js) and mitigation of the overhead of using third-party libraries (given that transporting the code of these libraries over the network can take a significant time). Furthermore, at the cluster level, the performance of FaaS platform can be improved by implementing intelligent load balancing strategies that are locality-aware, relatively to different locality dimensions: (i) locality of code, such as JIT information and third-party libraries, (ii) locality of session, for cases when the sessions share long-lived network connections, (iii) locality of data, notably for data processing function patterns such as a Map/Reduce implementation on top of Cloud functions. For example, Abad, Boza, and Eyk [16] report early results of on ongoing work that shows that, using a cluster-level scheduler that favors code locality, it is possible to improve the latency of requests by up to 66%.

Other challenges are related to new opportunities identified by the authors to extend the FaaS model: (i) improving the interoperatibility with databases, notably the support for the processing of batches of data from the database, (ii) defining a more adequate billing model for functions that have a long polling behavior, (iii) proposing metrics reporting the billed cost of each individual execution of a functions to facilitate cost optimization for the developers.

Finally, Hendrickson et al. discuss their intent to create a benchmark suite, *LambdaBench*, to facilitate the performance evaluation of research contributions for FaaS platform. Eyk et al. [4] also express their intent of developing a benchmark in the context of the SPEC Research Group. Unfortunately, these benchmark suites have not been released as of today.

2.2 Performance of Cloud Infrastructures

Our study focuses on the impact of the FaaS infrastructures on the performance of Cloud function executions. Although, to the best of our knowledge, no work has studied this problem specifically in regards to the performance of FaaS platforms (except for the overhead of the container engine), several works have studied it in the more general context of Cloud infrastructures.

In the recent years, the focus of these studies have shifted towards improving the utilization of Cloud infrastructure and improving the tail latencies. The latter is motivated by a phenomenon known nowadays as the *"Tail at Scale" problem* [17]. For one request, a 99 percentile latency describes the worst case over 99 other better cases. However, in the context of Cloud platforms and microservices, a single user-facing request (e.g., loading a page from a Web application such as Facebook) may result in hundreds of parallel and sequential

[3]For example, Google is known to use Linux as a common base for all its servers/workloads, which among other benefits, provides high flexibility for resource allocation within a datacenter and unified tooling for performance monitoring/analysis.

sub-requests (e.g., to fetch and aggregate data from several distributed servers). The latency of the request from the point of view of the user is thus determined by the slowest of all these sub-requests. Consequently, the worst case for one sub-request becomes the common case for the end-user. For example with 10 sub-requests, the probability that one of these sub-requests has a latency that is greater or equal to its 99 percentile latency is roughly 10%; for 100 sub-requests, this probability becomes 63%.

Among the studies of tail latency in existing systems, one of the major works is the study of Li et al. [18]. Li et al. provide a systematic study of the causes (at the hardware, operating system and application levels) of high tail latencies for latency-sensitive services. The authors start by deriving an ideal model relying on queuing theory in order to use it as a baseline. Then, they evaluate the latency of three relatively simple online services (a null-RPC server, an instance of Nginx serving a static web page and a Memcached key-value store server) and show that their performance are significantly worse than predicted by the theoretical model. The study thus proceeds to uncover and fix six causes of high tail latency with these applications.

The first cause of high tail latency is the presence of background processes (e.g., the SSH and Network Manager daemons), which (when scheduled) can delay the execution of the latency-sensitive tasks. Their solution is to give a higher priority of execution to these latency-sensitive tasks, enforcing the fact that an incoming latency-sensitive job should be run as soon as possible, potentially preempting a currently running background task. This is achievable by configuring these tasks to be scheduled using the real-time FIFO scheduling policy implemented in Linux while the background tasks are still scheduled using the standard scheduling policy.

Similarly, Leverich and Kozyrakis [19] study the co-location of latency-sensitive tasks (in their study, a Memcached server) and other tasks (notably long-running, batch/best-effort tasks such as data analytics jobs) on the same machine to increase the utilization of data center machines (a common setup used by Google on its server farm). The authors identify three sources of latency degradation that are caused by the co-location of these tasks and provide solution to address them: (i) interference on shared hardware resources (caches or memory) that can be mitigated with a more careful/conservative provisioning of the machines for the latency-critical service (to avoid pathological queuing delays), (ii) threads imbalance on cores that the authors solve with a careful thread-to-core mapping of the latency-sensitive tasks, (iii) scheduling delays that are best handled with tuning of the scheduler behavior (by assigning more weight to latency-sensitive tasks, e.g, a higher niceness or a real-time priority) or a new scheduler, such as the Borrowed Virtual Time scheduler [20] that implements specific mechanisms to boost latency-sensitive tasks.

In the study of Li et al., the use of the FIFO scheduling policy also introduces another benefit for the null-RPC server that uses a multi-threaded execution model. In this model, each request is assigned to one thread, thus the order in which the threads (hence the requests) are processed is determined by the operating system scheduling policy. By applying a FIFO policy to these threads, the requests are processed in a FIFO order, which has be shown to yield lower tail latencies.

The advantage of using a FIFO scheduling policy is also discussed more in-depth by Asyabi et al. [21] in the more specific context of multi-threaded servers using the Linux Completely Fair Scheduler (CFS). Asyabi et al. instrument the Linux kernel and show that the fair scheduling policy implemented by the Linux CFS scheduler behaves in practice closer to a Last-Come, First-Served (LCFS) scheduling policy than a First-Come, First-Served (FCFS) policy. The authors more specifically pinpoint the strategy of the scheduler that tries to schedule I/O-bounded tasks as early as possible. In the context of a server that processes requests by assigning each one of them to a dedicated thread (such as the Apache Web Server), this LCFS behavior translates into a server that processes requests in a non-FIFO order. In order to address this problem, Asyabi et al. propose a modification of the Linux kernel that adds a new layer to the scheduler: *CTS*. With their modification, the CFS policy is still responsible for scheduling the processes running on the machine in a fair fashion. However, the CFS policy is no longer responsible for the scheduling of threads. This responsibility is delegated to the new CTS layer that schedules the threads pertaining to the same process in an FCFS fashion. The empirical evaluation of the authors shows that CTS improves the 99 percentile latency of an Apache Web Server by up to 51%.

As a third cause of high latency, Li et al. study the impact of using a multicore machine. Theoretical results shows that a single shared queue of ready tasks (in contrast to multiple queues, one for each core) offers better tail latencies because it ensures a better load balance. However, in practice, a single shared queue is difficult to implement efficiently because of the overhead of synchronization and contention. For example, the Linux scheduler uses one queue of ready tasks for each core to minimize contention but tries to mimic the performance of a single queue system by implementing a work-stealing strategy [22]. The null-RPC server, with its one-thread-per-request model, inherits this queuing model and performs well on a multicore machine. On the other hand, Nginx and Memcached have tail latencies that are similar to a single-core setup. They both use a multiple-queue design and assign new TCP connections to one of these queues for its entire lifetime. Effectively, single instances of Nginx and Memcached on a multicore server behave as multiple instances of themselves on a single core machine, one on each core.

The fourth cause that the authors identified is the processing of interrupts, notably due to the management of network I/O operations. At high utilization, most server machines rely on the `irqbalance` daemon to balance the interrupts among the available cores. Similarly to background processes, this can cause delays in the processing of requests and slow down the processing due to cache pollution. In order to fix this issue, the authors configured the operating system to issue all interrupts on a core that was exclusively reserved for this purpose. Consequently latency-sensitive threads can be executed on the other cores without interruption.

The last two causes are related to NUMA (*Non Uniform Memory Access*) effects and power saving optimizations. In a NUMA architecture, accesses to remote parts of the memory can be significantly slower than accesses to the local memory. The authors show that for a multi-threaded server, a bad memory allocation strategy can cause many remote memory accesses and consequently degrade the latency. Power saving optimizations include *C-State* and the dynamic CPU frequency system of Linux. At low utilization, the different CPU C-States enable lower energy consumption by shutting down more components of the CPU chip at the cost of higher wake up times of these components and degradation of the tail latency. The dynamic CPU frequency system is able to reduce CPU frequency and thus power consumption, but it can slow down the execution of latency-sensitive tasks.

These studies shows that it is possible to substantially improve performance of Cloud workloads by carefully tuning the infrastructure. However, they validate their solutions in simple contexts (e.g., a simple Nginx or Memcached server) that are not representative of the complex nature of current Cloud infrastructures and more specifically FaaS infrastructures (a detailed description of a FaaS platform architecture is provided in Chapter 3).

3 OpenWhisk: a Case Study of FaaS Plaftorm Architectures

This chapter provides a description of the architecture of the OpenWhisk platform as a case study and preliminary step of our study of FaaS platform performance. This description focuses on the architectural components involved in the execution of Cloud functions as it is our main interest in this work. Although descriptions of the architecture of OpenWhisk are available [38, 39], we have found that they are sometimes no longer entirely accurate (OpenWhisk is a recent and quickly evolving project) and lack lower-level details which are important as they can have a great performance impact.

In our work, we chose to use OpenWhisk as our testing platform given that: (i) it is used in production (it has been in public beta for more than two years, since February 2016, and has been generally available since December 2016), (ii) it is increasingly being adopted by companies for their own platforms (public or private), (iii) it has a better documentation of its internals than other platforms and (iv) the development of the (academic) OpenLambda prototype seems to have been mostly stalled since its release (with less than 30 git commits in the last five months, while OpenWhisk has seen almost 300 commits in the same period).

3.1 Overall Architecture

OpenWhisk is a relatively complex piece of software implemented as a set of distributed components, as displayed in Figure 1: (i) the *API Gateway* (*OpenResty* [59] a distribution of *Nginx* [60] with support for *Lua* [61] extensions) that works as a user-facing reverse-proxy, (ii) an HTTP load balancer (another instance of *Nginx* [60]) that receives the end-user HTTP requests and forward them using the corresponding REST API call to the Controller (iii) an internal database (*Apache CouchDB* [62]) that stores the platform data (functions, execution results, and other informations), (iv) a message broker (*Apache Kafka* [63]), (v) the Controller, and (vi) the Invoker, the latter two being custom components written with the Akka actor framework [64] (in Scala, a programming language targeting the Java Virtual Machine). In a typical production setup, each component is deployed in multiple instances, both for scalability and fault tolerance purposes. However, we do not study these aspects in details in this work, as we focus on the inner working of individual Controller and Invoker machines. In the remainder of this section, we provide additional details on the main components: a Controller machine (§3.1), an Invoker machine (§3.1), and the containers used to execute the Cloud functions (§3.1).

Controller

The Controller is the component of the architecture that coordinates all the other components and has several roles. First, it implements the REST API, i.e. it is responsible for the creation, update and deletion of all the stored information of the system (e.g., the functions). More importantly, the Controller receives execution requests for all the functions and dispatches these requests towards the Invoker instances. The Controller balances the function executions assigned to each Invoker: in other words, the Controller keeps track of which Invokers are available[4] and the number of ongoing executions assigned to each Invoker. This number of ongoing executions is used as a measure of the load of the Invokers. When the Controller is duplicated in several instances (for redundancy or scalability reasons), each Controller maintains its own local counters of the number of functions assigned to each Invoker and these counts are synchronized using an eventually consistent algorithm (more precisely, a conflict-free replicated data type relying on the *Akka Distributed Data* feature [40]).

The Controller sends execution requests to Invokers through a dedicated Kafka topic (i.e., communication channel), one for each Invoker. As an example, if the Controller has decided to assign an execution request to the Invoker of id 3, it will produce a message containing the function identifier in the dedicated Kafka topic `invoker3`. The justification for the choice of Kafka is that [41]: *"[It] lifts the burden of buffering in memory, risking an OutOfMemoryException, off of both the Controller and the Invoker while also making sure that messages are not lost in case the system crashes."*

If the execution of a function is requested as non-blocking, the REST HTTP response is sent immediately after the execution has been sent to the Invoker instance and it contains the execution identifier that can be used to query the result of the execution later on. In the case of a blocking execution, the Controller gets the response from the Invoker through Kafka (via the `completed${controller_id}` Kafka topic) and returns it in the the REST HTTP response.

Invoker

The Invoker is designed as a machine-wide container pool. It creates new containers when necessary and implements the pooling strategy that reuses containers for consecutive executions of the same functions. Its role is to receive function executions, execute them inside a container, get the execution

[4]Invokers sends periodic *heartbeat* messages to notify the Controller that they are running.

results back and put these results in the CouchDB instance. As mentioned before, in the case of a blocking execution the Invoker will also send the result back to the Controller via Kafka.

The Invoker is implemented as an Akka [64] "actors system" running as a single process application in a Java Virtual Machine (JVM). Akka actors offer a programming abstraction different from the common multi-threaded model and closer to the event-oriented model. One of the most important characteristics of this model is that one actor instance cannot have two concurrent executing contexts. Each Akka actor is essentially an event-loop that processes incoming messages (i.e., events) from its mailbox (i.e., event queue). These messages can be received from external sources (e.g., network messages) or from other actors. This model promotes the development of an application targeted for multicore systems in a distributed fashion, i.e., it promotes an application design that splits the application in several actors that do not share state and only communicate by messages. This model reduces the need for synchronization mechanisms, which are a well-known cause of performance issues. In order to actually implement this actor model on existing operating systems that rely on threads to leverage the multiple cores of a machine, Akka uses its *Dispatcher* abstraction, which defines how actors are executed with threads [42]. A Dispatcher manages a pool of threads and executes actors with one of the available thread of the pool when they receive a new message.

In an event-oriented model, it is generally better to prefer non-blocking APIs than blocking APIs for I/O. Nonetheless, Akka provides several strategies to deal with blocking APIs when they are unavoidable (typically when a non-blocking library alternative is not available). For example, the Invoker uses blocking calls for everything related to the management of containers. The strategy used in the Invoker to handle blocking calls is to execute these calls in a *Scala future*. Given that futures are run on separate threads, the blocking call executed in the future does not block the actor (which would prevent it from processing other messages).

The Invoker is comprised of three main kinds of actors (represented in the Invoker process in Figure 2). The first one is the *MessageFeed*, its role is to poll the Kafka topic periodically for new execution requests received from the Controller. When it receives a request, it loads the metadata of the function to be executed from the database and forwards the request to the second actor, *ContainerPool*. The ContainerPool is used to manage the pool of containers, it receives execution requests from the previous actor, selects an already existing container instance or starts a new one when necessary and forwards the execution request to the relevant *ContainerProxy* actor. For each running container, a distinct instance of ContainerProxy actor is used. A ContainerProxy instance keeps track of the state of its associated container (starting, running, idle) and forwards the functions to be run to their container.

Invoker and Containers

While the Invoker manages the lifecycle of the containers, it relies on the Docker daemon to actually start, stop, pause and resume them. The Invoker sends commands to the Docker daemon via the `Docker` command line tool; the command line tool itself uses either a Unix domain socket or a TCP socket (depending on its configuration) to communicate with the Docker daemon. When the Docker daemon receives a command to start a container, it creates the new initial process of this container and issues the necessary system calls to configure the isolation settings (i.e., configuration of the namespace, cgroup, overlayFS and creation of virtual network interface). In order to stop a container, the Docker daemon simply sends a `SIGTERM` signal. For the two other operations (pause and resume) the Invoker bypasses the Docker daemon for efficiency. It does so thanks to the `runc` command line tool that is able to control running containers without going through the Docker daemon (thus avoiding the overhead of the communication over the socket). A container is paused via the *cgroup freezer* feature, which works similarly to the `SIGSTOP` signal in a standard process context.

For the language runtimes that are explicitly supported by OpenWhisk (e.g., JVM, Node.js, Python, Swift, ..., as opposed to custom black-box containers), the containers are started with pre-made container images, one image for each kind of these execution runtimes. By default (without any of the optimizations described below in this section), the execution of a function in a container can be separated in three main steps: 1) starting the container, 2) loading the function code in the container and 3) actually executing the function. When the execution has to go through these three steps, it is referred to as a *cold* start. In order to mitigate the overhead of the first two steps in the latency perceived by the end-user, the Invoker uses two techniques: pre-warm containers and reused containers.

The first technique allows the execution to skip the first step of starting the container. The Invoker can be configured to prepare a pool of ready containers for specific execution runtimes (e.g., Node.js) and memory limits. These *pre-warm* containers are already started, hence the execution skips the first step but they will still need to be loaded with the code (second step). Once a pre-warm container is used for the execution of a function, it is removed from the pool and a new container is started in parallel to replace it.

The second technique is to reuse containers: once a function has run, the Invoker will keep its container. If the same function is invoked again on the same Invoker the idle container will then be reused. This second execution, described as a *warm* start, can thus skip the first and second steps. An Invoker keeps the most recent containers after the execution of functions up to a configurable maximum number of running and warm containers. When the Invoker needs to run a function for which no warm container is available and this maximum number of containers has been reached, it will shutdown the oldest warm container. If all containers are executing a function, the execution will be queued. Warm containers are kept ready for 50ms and paused after that.

4 Experimental Methodology

This chapter describes our experimental process. Section 4.1 describes the deployment of the components of OpenWhisk on our testbed. Section 4.2 explains our choice of metrics.

Section 4.3 explains our experimental protocol and some of the changes we made to OpenWhisk to streamline the experimental process.

4.1 OpenWhisk Deployment

For our experiments, we use three different machines; the hardware and software specifications of these machines are summarized in Table 1, and the deployment of the OpenWhisk components and their configuration are summarized in Table 2. Given that we do not consider energy efficiency in our study (this aspect is left for future work), the machines are configured for maximum performance: we disabled the `cpufreq` daemon of Linux and configured the CPUs of the machine for maximum speed via the BIOS (notably we disabled change of C-State).

The *Load Injector machine* is used to simulate the user requests (directly through the Controller component, bypassing the REST API load balancer — see Figure 1). While the hardware configuration of this machine is somewhat less powerful than the two other machines, we have verified that it is not a limiting factor in practice. The *Controller machine* hosts the Controller process, the Kafka broker, and the CouchDB instance. Similarly to the Load Injector machine, we verified that this machine was not a limiting factor in the experiments (by checking that its utilization of hardware resources remains low[5]). The *Invoker machine* hosts the Invoker and runs the Cloud functions in the containers. This machine requires at least the version v4.13 of Linux: before this version, Linux was suffering from a performance issue with large number of cgroups in the system [43]. Additionally, we configured this machine using the guidelines provided by IBM for the performance tuning of Docker on large multicore machines [44]. All components of the FaaS platform are installed with the Ansible [65] configuration provided by OpenWhisk. In order to facilitate and automate the deployment, this configuration sets up every component (e.g., Controller, CouchDB, Kafka, Invoker) inside its own Docker container (not to be confused with the Docker containers used for the execution of Cloud function). Finally, all the machines are equipped with a 10Gb/s Ethernet Network Interface Controller (NIC) connected to the same switch and all network communications between the components across machines are configured to use these links.

4.2 Performance Metrics

We use two main metrics to evaluate the performance of our system. First, we use the *throughput* expressed as the number of function execution requests that have been completed (i.e, the results of the request has been received by the client) per second. Second, we also use *latency* as a performance metric. In order to assess the overall performance of the system we use the latency as perceived by the end-user, i.e., the elapsed time between the request of a blocking function execution and the reception of its results. We do not summarize the distribution of latencies using averages and standard deviations as they are inadequate. The latency distribution of a system such as OpenWhisk does not necessarily follow a normal distribution: they are usually multimodal (e.g., with measurements under regular system behavior and measurements under temporary slowdown such as a stop-the-world garbage collection), asymmetric, and can be significantly right-tailed which skews the mean towards the right. As a replacement we use the median as it provides a better estimation of the central tendency [23, Chapter 12.3] and percentiles as a description of the tail of the distribution. More specifically, we use 95% and 99% percentiles as they have become the standard in both the industry and academic world. In the industry, they are used to express *SLAs* (*Service-Level Agreements*), e.g., 99% of the requests must be serviced under 1 second. The use of high percentile values (95% and 99%) is motivated by the *"Tail at Scale" problem* (described in Section 2.2).

Throughput and latency are both important in their own right and their relation is not trivial: while an optimization can benefit both metrics, sometimes an improvement of the throughput can degrade the latency and vice-versa. Our main objective is to improve the resource usage efficiency of a FaaS platform. A primary means to achieve this objective is to improve the throughput as this improvement can directly be translated to a higher number of collocated functions on the same machine. However if this improvement in terms of throughput comes with a significant degradation of the latency, the Cloud provider might not be able to satisfy its SLAs. Given this complicated relation between throughput and latency, an important tool to assess the quality of a FaaS platform is the throughput versus latency plot. The points of the plot are obtained by measuring the throughput and latency with increasing levels of input load applied to the system under test and thus increasing levels of utilization of the system until it reaches saturation. This plot summarizes the values of throughput and latencies and how they relate to each other. Furthermore, it provides an insight into the performance characteristics of the system and more precisely in its ability to work at a high level of utilization.

4.3 Experimental Protocol

For our experiments we use a synthetic workload that consists in a single function written for the Node.js runtime that takes no argument, consumes 10 milliseconds worth of CPU cycles and returns with an empty response. We choose the duration of the function to be at the scale of the millisecond, in line with the focus on our study. More specifically, we choose 10 milliseconds, i.e., on the low end of the millisecond scale, to stress and reveal the infrastructure overheads.

One of the important objectives for the performance evaluation of a system via experiments is the adequacy of the experiments with real-world settings. In the context of a system that is used as a remote server by many simultaneous users, an accurate load injection [6] is of decisive importance. Although it is essential and has a significant impact on the validity of the measures, choosing an efficient and accurate load injector remains a non-trivial task [24, 25, 26]. One experiment

[5]Except for a garbage collection issue that we address in section 5.2.

[6]The term "load injection" describes the process of simulating a set of (concurrent) user requests, usually from another machine, in order to trigger the workload in the system under test and measure its performance characteristics.

consists in an injection of load lasting 10 minutes. The load injector creates blocking execution requests for the 10 millisecond function following the closed loop injection model described by Schroeder, Wierman, and Harchol-Balter [25]. We also provides results using correction for the coordinated omission problem as described by Tene [45].

In order to streamline the experimental process, we made some changes to OpenWhisk. First, our study focuses on the executions of functions in warm containers for simplicity and ease of reproducibility. In order to ensure that no request during the experiments will have to wait for the startup of a container, we modified OpenWhisk so that it does not pause warm containers and does not destroy them after a timeout. After a new deployment of OpenWhisk, we run a short load injection so that OpenWhisk fills its pool with a large number of containers (512).

Second, in its default configuration, once a the execution of a function is terminated, OpenWhisk collects its standard output using the Docker daemon and stores these logs alongside the function results in the CouchDB database. In practice, during our early tests, these steps of collecting the logs and storing the results in the database took substantially more time than the actual execution of the requests. Pending collection of logs and database operations would both continue to be executed long after all requests were processed. Moreover, Docker seems to suffer from a bug that makes containers crash when a large number of container logs are collected at the same time. These two problems made experiments impractical and unreliable, and fixing these two issues is not a trivial process. However, the container logs and the database are not essential for our performance analysis as they are not necessary for the execution of blocking functions. Therefore, we choose to disable the background tasks that perform these operations in order to focus our efforts on the intrinsic performance issues of a FaaS infrastructure.

5 Performance Issues

This chapter presents the performance issues that we have identified in our study of the OpenWhisk platform. For each issue, we describe the steps we followed and the tools and methodologies we relied on to understand its root cause. This chapter also highlights some of the challenges we faced in the process due to the mixture of language runtimes, concurrency models and the use of containers for isolation that are present in the OpenWhisk platform and discusses our approach to address them.

Section 5.1 presents our assessment of the importance of the Invoker in the performance of the platform. Section 5.2 details our study of the effect of the garbage collector of the JVM on the throughput and tail latency. In Section 5.3, we describe the bottleneck caused by the ContainerPool actor of the Invoker process and the performance isolation resulting from the use of containers. Section 5.4 provides an analysis of the performance overhead of the communication scheme used between the Invoker process and the containers running the functions.

5.1 Identification of the Bottleneck Component

Our first step in the process of improving the performance of the platform was to identify which component (Controller, Kafka, Invoker or the function runtime) of the whole system was acting as the main bottleneck. In order to do so, we followed a *latency analysis* approach [27, Chapter 2.5.12]. With this approach, the latency of an operation is divided into the substeps of the operation. This process can be reiterated as much as necessary by subdividing the steps which takes the most time. In our case, the operation is a Cloud function execution request and we were trying to pinpoint the component of the OpenWhisk architecture where the request spends the most time. OpenWhisk already provides a logging system that logs various timed events related to a request (e.g., when a request is received by the Controller, when the Invoker is sending the request to the container's HTTP server for execution). The log entries generated by the logging system contain (i) the unique identifier of the user request related to the log entry, (ii) a timestamp (from the nanosecond-scale system clock) in milliseconds relative to the start of the request, and (iii) additional information depending on the nature of the log entry. Table 3 provides a simplified trace gathered from these logs and filtered to include only the entries related to one request. This is the trace of a simple *hello-world* Cloud function that returns immediately. From this a trace we were able to assess that a significant part of the processing time of such a minimal request was due to the processing inside the Invoker machine – about 60% of the total processing. Therefore, in the remainder of this chapter, we focus our study on the Invoker machine — although we also discuss the Controller process in Section 5.2 as both the Controller and Invoker processes suffer from the same issue.

5.2 Garbage Collection

While we were applying the latency analysis approach to pinpoint the Invoker as the component to focus our effort on, we also observed substantially high tail latencies. These results led us to the first issue that we will discuss: the Garbage Collector (GC) of the Controller and the Invoker processes[7]. Figure 5 shows the tail latency of our system with and without the correction of `wrk`, i.e., with and without additional requests to account for the coordinated omission problem as described in Section 4.3. The difference between the two latencies observed with and without the correction is substantial. This result illustrates the importance of choosing the right injection model.

Evidence of the Issue

The main difference between the two tail latency distributions in Figure 5 is that the corrected values account for the large period of time when requests are significantly delayed because of a general slowdown of the system. During the experiment, we observe that the CPU utilization of the Invoker machine is showing both spikes (most CPUs are at 100% utilization, while a few others are idle at 0% utilization) and slowdowns (all CPUs are close to 0% utilization) for short

[7]Remember that each of these two processes corresponds to an Akka application running in a Java Virtual Machine.

periods of time (less than 1 second). The spike episodes are due to the garbage collections happening in the Invoker process, whereas the slowdown episodes are due to the lack of incoming requests for the Invoker because of the garbage collections happening in the Controller. The Controller machine displays the same type of behavior: high utilization episodes during its garbage collections and low utilization during the garbage collections in the Invoker machine (because of the closed-loop model, requests blocked by the stop-the-world GC in the Invoker prevent new requests from being created).

Figure 3 and 4 shows the evolution of the allocated heap memory (in orange) and the used memory (in blue) of the Controller and Invoker processes. In both cases, the used memory raises up to about one third of the allocated memory and then drops close to zero. The GCs used in the Controller and Invoker are *generational* [28]: they partition allocated objects into generations based on their lifetime. As an illustration, in the simplest version of a generational GC with only two generations, a newly allocated object is put inside the young generation and if it survives a certain amount of garbage collections, it will be promoted to the old generation. The default configuration of the JVM dedicates one third of the whole memory allocated to the JVM to the *Eden space* (the newest generation). When a new object is allocated while the Eden space is full, a garbage collection is triggered in order to free memory in the Eden space, by freeing objects that are no longer necessary or promoting the surviving objects to the older generations otherwise. The pattern shown in Figure 3 and 4 illustrates this behavior: the Eden space is quickly filled with objects until its maximum, at which point a garbage collection is triggered which brings the allocated memory close to zero as the large majority of allocated objects are ephemeral. Garbage collection is not an issue in itself. However, in our case, the stop-the-world step of the garbage collection was taking almost 1 second and delaying all the requests by the same amount of time.

Memory Allocation Flame Graph

In order to address a GC issue, it is important to understand what kind of objects are created and which parts of the code are creating them. Java Flight Recorder (JFR), a profiling tool (described in Section 5.3) bundled with the Oracle JVM, offers the ability to obtain a trace of the occurrences of two events related to memory allocations: *allocation in new TLAB* and *allocation outside TLAB*. The notion of Thread-Local Allocation Buffer (TLAB) corresponds to buffers, part of the Eden Space, that threads use for the allocation of new objects [28]. Each TLAB is exclusive to one thread, thus a thread can allocate an object inside its TLAB without synchronization. A synchronization is only necessary when the current TLAB of a thread is full and a new TLAB has to be allocated by the JVM for the thread. The allocation of a new TLAB by the JVM is an *allocation in new TLAB* event reported by JFR. The *allocation outside TLAB* event corresponds to (the rare) allocations of large objects. The designers of the Oracle JVM have made the allocation of a new TLAB available as a JFR event because it offers a low-overhead (i.e., less frequent than reporting each and every allocation), yet representative, sample of the allocations of objects. These two allocation events can be traced with JFR and report the type of object that is allocated and, like any JFR event, the Java application call stack that led to the event. In order to properly[8] analyze the results of these allocations, we used the `jfr-flame-graph` [66] tool that consumes the trace generated by JFR and generates a flame graph [29]. A *flame graph* is a type of visualization that displays call stacks using a diagram with an icicle layout. Each entry of the call stack is represented as a node of the diagram with its horizontal size being representative of the weight given to the entry. Flame graphs are notably used to analyze the time spent executing functions on the CPU, but are versatile. In our case, we use them to analyze memory allocations and the code paths that led to them and we weight the entries of the flame graph with the size in bytes of the allocation (see Figure 8 in appendix for an example).

Logging System

Our first *allocation flame graph* for both the Invoker and the Controller shows that most of the allocations are triggered by the logging system (the same logging system we used for our latency analysis in Section 5.1) and more precisely its internal buffers. We choose to disable this logging system to assess the performance impact of this bottleneck. Figure 6 shows the throughput vs. median latency curves using a closed-loop injection, before and after disabling the logs. As an illustration of the improvement, for a median latency of 20 milliseconds the throughput increases from roughly 2800 to 3400 requests per seconds: a 20% improvement. Figure 7 shows the same curves but for the 95% and 99% percentile latency with and without the correction of `wrk`. The tail latency is greatly reduced without the logging system. However the corrected version still shows a degradation of the 99% tail latency.

Although, we disabled the logging system of OpenWhisk as a quick fix to remove its overhead, a logging system is still desirable for a platform running in production. However, a new, more efficient logging system is necessary to achieve good performance for such an high rate of incoming requests. This is especially true regarding the latency of the requests as we have seen that the 99% percentile latency is substantially degraded because of the garbage collection pauses. One solution would be to replace these logs with creation of JFR events, effectively delegating the log processing to the JVM. Another possible alternative would be to integrate a logging system designed for microsecond-scale efficiency, like the very recent NanoLog research prototype [30, 47] developed in the context of the Granular Computing Initiative at Stanford University [67].

Communication Buffers

As explained previously, disabling the logs increases the throughput of our system. However, increasing the load to match this new maximum throughput reveals that the garbage

[8]The Java distribution offers a GUI interface to visualize the information of these traces [46] but it is impractical for large programs like the Invoker process with many code paths leading to allocations. Notably, this tool aggregates the call stacks from callee to caller whereas our solution aggregates from caller to callee. Both approaches have their advantages in different circumstances, yet, based on our experience, aggregating from caller to callee is often more relevant.

collectors of the Controller and Invoker are still degrading the tail latency. Figure 8 in appendix shows the allocation flame graph of the Invoker when the logs are disabled. This flame graph shows two prevalent sources of allocations: (i) the code paths starting with the function `HttpUtils#post` represent 36% of the memory allocations and are objects allocated for the HTTP communications between the Invoker and the containers, (ii) the code paths containing the package `spray.json` in their name account for more than 30% of the memory allocations. `Spray-json` is the Scala library used by the Controller and Invoker for the serialization of the messages that they exchanged through the Kafka message broker. The allocation flame graph of the Controller (not shown here due to a lack of space) shows that most of the allocations can be traced back to the creation of many `char[]` arrays used as buffers for the processing of HTTP requests and generation of HTTP responses by the `akka-http` framework [68].

One could argue that it would be sufficient to increase the allocated heap memory (and consequently the size of the Eden) to have less frequent GC phases but these GC phases would take a longer time as the memory space to reclaim would be bigger. We have also evaluated the `G1GC` algorithm (the other main garbage collection algorithm of the Oracle JVM, designed to minimize the length of garbage collections pauses to avoid creating large tail latencies as we have seen in the Section 5.2) but it does not improve the tail latency of our workload. Other alternative garbage collection algorithms with similar low latency objectives can be found in other JVM implementations, notably the Oracle JRockit Real Time JVM [69, 31] and the Azul Zing JVM [70, 32], but we lacked the time to evaluate them.

Ultimately, the root cause of these problems is that is that the diverse components that we have highlighted (HTTP server of the Controller, serialization of Kafka messages, HTTP client of the Invoker) are not designed for low tail latencies at microsecond-scale requests. Currently these components delegate all their memory management to the JVM, they allocate their buffers using the `new` JVM operation and let the garbage collector eventually deallocate them. If these components used their own memory management strategy (using techniques such a pooling and/or reference counting) they could release memory exactly at the moment when it is no longer necessary rather than eventually when the garbage collector decides that it should be reclaimed. For example, the `netty` Java HTTP library [71] uses reference counting to minimize the number of allocated buffers and a buffer pool to reuse the buffers instead of letting the GC reclaim them [48].

5.3 ContainerPool Bottleneck

The latency analysis of Section 5.1 shows that a significant part of the processing time of the request was due to the processing inside the Invoker machine. Besides, it shows more than that: 40% of the processing time of the request is spent just before the request is processed by the ContainerPool actor (see Section 3.1 for the explanation of its role). This actor acts as a bottleneck because of the complex interaction between two factors: its single-instance nature and the performance isolation mechanism introduced by containers (detailed in the following sections).

Evidence of the Issue

In Table 3, a significant part of the processing time of the request (39 milliseconds out of 98 milliseconds, i.e., about 40%) is spent in between two entries at timestamps 35 and 74. The first of these two entries shows when the Invoker has fetched the action source code (in this case from the cache) and is about to send the request (via an Akka actor message) to its ContainerPool actor. The second entry shows when the ContainerPool actor actually receives the request and is about to start processing it. The only thing happening in between these two steps is the queuing of the message. Consequently, the most probable reason explaining the large time spent between these two steps is that requests are delayed a long time in the ContainerPool actor message queue.

In order to verify this hypothesis, we implemented a new lightweight instrumentation facility to monitor the size of the actor message queues. Another instrumentation tool, the `kamon-akka` library [72] is available for this purpose and is integrated in OpenWhisk (although disabled by default) but using it is complex and requires setting up new distributed components that add complexity to the testbed and an additional overhead to the Invoker process: (i) a metric gatherer (e.g., *Statsd* [73]), (ii) a metric listener (e.g., *Graphite* [74]), and (iii) a visualization tool (e.g., *Grafana* [75]). In comparison, our solution is straightforward to setup, only relies on tools that are bundled with a standard Oracle Java distribution and is low-overhead as it gathers the profiling information directly in the JVM.The instrumentation of the message queues is inspired by the technique proposed in *Akka in Action* [33, Chapter 16.2.1] (replacing the implementation of the `MessageQueue` with a custom implementation) and exports message queue statistics via JFR [76] custom events. JFR is an in-JVM profiling tool; the JVM records occurrences of events via an efficient buffering scheme with minimal overhead and can dump the trace of these events in a file for post-mortem analysis. Although this process of collecting in-JVM and buffering makes this instrumentation less suitable for live monitoring, it enables fine-grained — and low-overhead — analyses. Furthermore the instrumentation is designed in such a way that it causes almost no overhead when not enabled. Our instrumentation of the message queues generates occurrences of custom JFR events [49] each time a message is queued and occurrences of another custom event each time a message is dequeued. It also reports additional information such as the size of the message queue at that moment for both events and the queuing delay of the message for the dequeue events.

With this instrumentation, we observed that the number of messages in the queue of the actor increases sporadically. A likely explanation for these sporadic increases of the number of messages in its queue is that the ContainerPool actor does not receive enough CPU time to process them, i.e., the increases correspond to periods where the ContainerPool is not scheduled, thus does not consume messages.

However, observing the CPU time (and more generally the behavior) of the ContainerPool actor to verify this hypothesis was not immediate. Usual profiling tools provide information associated to threads. The default configuration of the Invoker uses one single Dispatcher for all the actors of the

Invoker and, furthermore, all the actors and futures share the same thread pool. As explained in Section 3.1, the thread that executing the ContainerPool actor changes from one reception of a message to another[9]. We modified the configuration of the Invoker so that it uses a dedicated Dispatcher with its own thread pool — containing only one thread — for the ContainerPool actor. VisualVM (a Java monitoring tool [77]) shows that the single Java thread used by the ContainerPool actor is never in a JVM-level blocked state. In other words, this thread is always eligible to receive CPU time from the point of view of the JVM. However, the ContainerPool thread could still be in a situation where it does not receive enough CPU time from the OS scheduler. In order to verify this hypothesis, we run a new experiment with the ContainerPool thread configured to use the FIFO Linux scheduling policy (`SCHED_FIFO`). The FIFO policy has a higher priority than the standard (CFS) policy: threads configured as FIFO are scheduled before any other threads. This new setup improves the performance by 15%, confirming the hypothesis of a lack of CPU time at the OS level.

Cgroups Performance Isolation

In this section, we show that the lack of CPU time of the ContainerPool actor is explained by the performance isolation mechanism used by containers in Linux. The Linux CFS scheduler is a fair scheduler. In a standard server setting, this fairness is applied at the granularity of threads, i.e., the scheduler tries to give the same amount of CPU time to each thread. In the multi-tenant context of Cloud computing, fairness is applied at the granularity of containers (via cgroups) to ensure performance isolation. As an example, consider a server in a Cloud environment with two applications from different tenants running in containers: App_1 that uses only one thread and App_2 that uses one hundred threads. In this context, the single thread of App_1 receives one hundred times the average amount of CPU time that each thread of App_2 receives. Therefore, the two applications receive the same amount of CPU time. However, this strictly fair allocation of CPU time is only true if both applications make use of all their allotted CPU time because the Linux scheduler is first and foremost work conserving: if the single thread of App_1 blocks often and/or for long periods of time and requires less than half of the CPU time, the remaining available CPU time will be given to the threads of App_2.

In Section 4.1, we explained that the Invoker process is deployed inside a container. In our case, the single container that runs the Invoker process receives the same amount of CPU time as each of the containers running on the Invoker machine. In our experiments, when the load injection approaches the saturation point, the number of containers in a runnable state can be as high as 70. During these periods of high utilization, the performance isolation mechanism prevents the Invoker process and its ContainerPool from receiving enough CPU time. Consequently the number of pending requests in the ContainerPool message queue starts to increase.

[9]Or possibly from the reception of a batch of messages to another batch depending on the configuration of Akka.

We fixed this problem by increasing the `cpu-shares` parameter of the Invoker container. This parameter defines a relative weight for the allocation of the container CPU time. As an illustration, if the `cpu-shares` parameter of a container C_1 is twice the value of the `cpu-shares` parameter of container C_2: C_1 receives twice as much CPU time as C_2. We increase this parameter by a factor of one hundred[10] for the Invoker container and observe that the throughput of the platform increases from roughly 3400 requests per second to 3700 at a median latency of 20 milliseconds.

Increasing the `cpu-shares` parameter of the Invoker process ensures that it receives more CPU time during utilization spikes. However, this is done at the expense of the containers executing the Cloud functions. Given that these containers also process the functions and thus contribute to the throughput, the increase in CPU time allocated to the Invoker does not fully explain the increase of the throughput. To understand the effect entirely, it is necessary to consider the design of the actor system of the Invoker process. The ContainerPool actor is a single-instance actor that has to process every request assigned to the Invoker machine: it is a serial bottleneck. Giving more time to this actor ensures that it is able to assign functions to the containers in a timely manner and consequently that more functions are able to execute in parallel and to leverage the available cores.

In order to validate our understanding of the problem, we devised a new experiment. The hypothesis is that there is a mismatch between the single-instance nature of the container, which cannot scale, and the number of containers that scales accordingly to the capacity of the system. By changing the number of cores of the machine hosting the Invoker and adjusting the load injected into the system, we can verify this hypothesis. Figure 10 shows a set of experiments with the same Invoker machine booted with a different number of enabled cores (16 and 64 cores). The throughput is expressed as the throughput per core. As we can see in the bottom plot of Figure 10 with our modification of the `cpu-shares` parameter, the throughput per core does not change with 16 or 64 cores, the characteristic curves of the two configurations are the same. However, in the top plot, without the modification of the value of the `cpu-shares` of the Invoker, the curves are different: the 64-core machine has a lower throughput per core. By increasing the number of cores and the load injected into the system, we increase the number of containers that are necessary to sustain this throughput. In contrast, the ContainerPool actor remains a single instance actor and gets to the share the CPU with roughly four times the number of containers.

One can argue that this issue is simply a problem of a badly configured system, However, we think that there is a lesson to learn regarding the usage of Docker (and to some extend other container engines). While Docker has been designed with multi-tenant settings in mind, it is increasingly

[10]Increasing this value step by step shows that the throughput increase accordingly until it reaches a plateau around a factor of 37. Our understanding is that the CPU time given to the ContainerPool increases until it reaches the maximum CPU time that the actor can make use of. At this point the parameter has no more influence and the throughput stagnates.

being adopted as a DevOps tools, i.e., to ease the process of packaging and deploying applications. When the developers of OpenWhisk provide an Ansible configuration that deploys the components as Docker containers, they do it primarily to simplify the deployment, not to ensure performance isolation between the components and other containers running on the same machine. However, as we have seen above, the performance isolation mechanism can create undesirable degradation on the performance of the containerized application.

5.4 Overhead of the TCP/HTTP Communications in the Invoker

Once we had addressed the GC and ContainerPool issues, we wanted to understand which code paths of the Invoker machine were taking most of the CPU time. Such an analysis is achievable via `perf` [78], a versatile performance debugging tool integrated into Linux. `perf` can — among other features — sample the call stacks of running threads at regular intervals of time. This sample of call stacks can then be aggregated and visualized using a flame graph that shows the most called code paths.

However, using `perf` to obtain system-wide results is challenging with a platform such as OpenWhisk because of two reasons: container isolation [50] and language-level virtual machines (VMs) [51, 52]. Notably, `perf` relies on debugging symbols to retrieve the name of the functions appearing in a call stack. These symbols can be directly embedded into the executable or stored aside (usually for packages installed via the package manager of the Linux distribution). `perf` reads these files (executable or detached symbol files) from the filesystem. However, when the functions are part of programs running inside a container, given that `perf` is run from the host, the files may not be available because of the isolation of the file system. Furthermore, these symbols are not available for "JITed" (Just-in-time compiled) functions in language-level VMs such as the JVM and Node.js. For these functions, `perf` offers an alternative, the `perf-<pid>.map` files [53], which should be generated by the VM and contain the names of the JITed functions. If a language-level VM is run inside a container, this file should also be made available to `perf` running in the host. Additionally, `perf` expects the file name to contain the pid of the language-level VM, however, the pid of the VM as seen by `perf` from the host can be different from the pid inside the container because of the namespace isolation.

In order to obtain a trace of call stacks with `perf`, we had to devise various solutions to work around these limitations. For example, the Invoker uses the Oracle JVM. This JVM (i.e., the `libjvm.so` library file) embeds the debugging symbols of its internal functions. This library is part of the Invoker container filesystem, thus the library file is in a filesystem isolated from the host that is running `perf`. Consequently, when looking for the mapping between the function address and the function name, `perf` looks at the debug symbols inside the executable using the path that is reported by Linux, however this path is only valid from inside the container. Our quick fix for this specific issue was to create a copy of the Java distribution (and thus the `libjvm.so` file) in the host under the same path to trick `perf` into thinking it is reading the symbols of the running executable whereas it is only reading a copy. As another example, in order to allow `perf` to find the `perf-<pid>.map` file, we modified the parameters used to start the various containers of the Invoker machine to disable the namespace feature so that the pid seen from the JVM and Node.js processes inside the containers is the same as the pid seen by `perf`.

After addressing these issues we were able to generate the flame graph shown in Figure 11 in appendix. This flame graph shows the breakdown of CPU-time per code paths for the Invoker machine. With an analysis of the code paths in this flame graph that correspond to the TCP/HTTP communications between the Invoker and the containers, we can see that around 34% of the CPU time is spend on these communications. This large overhead could be mitigated by replacing TCP/HTTP communications with a faster communication scheme. Additionally, this could reduce the effects of the garbage collector in the Invoker on the tail latency if the communication scheme also minimizes its usage of buffers. For example, a very recent reverse-engineering analysis revealed that AWS Lambda containers most likely receive execution requests via a segment of shared memory [54]. A similar approach applied to OpenWhisk could be beneficial for the performance. However, this solution requires substantial changes for both the Invoker and the containers running the functions. Another less intrusive solution would be to replace the TCP socket by a UNIX domain socket and replace the HTTP protocol by a custom protocol on top of the UNIX domain socket. Given more time, we could have evaluated the benefits of replacing the TCP socket by a Unix domain using Slipstream [34]. Slipstream automatically detects and replaces the local TCP sockets with Unix-domain sockets to improve performance.

6 Conclusion & Perspectives

During our study of the performance bottlenecks of the OpenWhisk platform, we faced several technical challenges to observe and understand their root causes. These challenges stem from the complex mixture of language runtimes, concurrent execution models and OS-level virtualization of the OpenWhisk infrastructure. However, these challenges are not limited to OpenWhisk or FaaS platforms as this complexity of the software stacks is representative of modern Cloud infrastructures in general. For each of the profiling challenges we faced, we proposed a corresponding, although often ad-hoc, solution. We believe these challenges deserve to be addressed with more robust and perennial solutions that can be applied to production systems.

Our study uncovered several performance issues in the current design of the OpenWhisk platform, notably in the Invoker machine. These performance issues reduce the ability of the platform to achieve workload consolidation in the context of small-scale functions as they all significantly degrade the throughput and latency. We have highlighted : (i) the high rate of ephemeral memory allocations (resulting from the logging system and the unfit buffer management policy of the communications stacks), which, coupled to the reliance on delayed garbage collections for memory reclamation substantially degrades the tail latency of the requests, (ii) the ContainerPool bottleneck that prevents the Invoker process from being scaled up properly (because of the performance isolation of the Docker containers), (iii) the overhead of the TCP/HTTP communications. We also started investigating the potential impacts of general scheduling decisions on the performance; although this last part of the work is unsuccessful for now, we plan to continue investigating in this direction.

The solution applied to remove the ContainerPool and the garbage collection issues (by respectively adjusting the cgroup performance isolation and disabling the logging system) improved the performance of the OpenWhisk platform from 2800 requests per second to 3700 at a 20 millisecond median latency. We believe the performance could be even further improved with the application of our two improvement proposals related to the communications (to mitigate garbage collection effects on the tail latency and improve the speed of communication between the Invoker process and the functions containers).

Given that FaaS workloads have characteristics that differ from previous Cloud workloads, we think that the FaaS model can benefit from new scheduling strategies. As a simple example, Cloud functions, which can have very short running times, can benefit from a lower overhead of the scheduler. More generally, given that the unit of execution of a FaaS platform is the function and that functions are more likely to have predictable execution times, memory and I/O patterns (as opposed to traditional, long-running operations of previous Cloud workloads), a smart FaaS platform could try to identify the behavior of a function based on its name and arguments and implement a more optimized allocation of resources for this expected behavior. However, we think a first step towards devising these kinds of improvements of the scheduling strategy is to develop profiling tools to better understand the behavior of the kernel-level and runtime-level schedulers and how their strategies influence the execution of functions.

Our study has also highlighted some challenges related to the profiling and detection of performance issues on Cloud platforms. The main problems of these platforms stem from: (i) their distributed nature (i.e., they are designed with several cooperating and communicating components); (ii) the heterogeneous nature of their software stacks: diverse language runtimes (e.g., Java or Node.js) and diverse execution models (e.g., thread-based or actor-based concurrency model, event-driven models); (iii) the security and performance isolation implemented for multi-tenancy. In the continuation of our work, we believe an important objective is to work towards better support of profiling tools in the Cloud, in order (i) to provide platform operators with more insightful performance profiles that offer a consistent view of the bottlenecks and interactions throughout the system stack and (ii) to improve the detection and root-cause analysis of transient and sporadic inefficiencies, which are essential for mitigating issues like variability and high tail latencies.

Acknowledgement

This work has been partially supported by the LabEx PERSYVAL-Lab (ANR-11-LABX-0025-01) funded by the French program Investissement d'avenir A special thanks goes to Hugo Guiroux, for all his help, advices, quick insights and his willingness to dive deep into the idiosyncrasies of the Linux kernel with me. I would also like to express my gratitude to my supervisors Renaud Lachaize and Vivien Quéma for giving me the opportunity to work on such an interesting topic and for their dedication and continuous guidance throughout this Master thesis, both on the scientific and non-scientific levels.

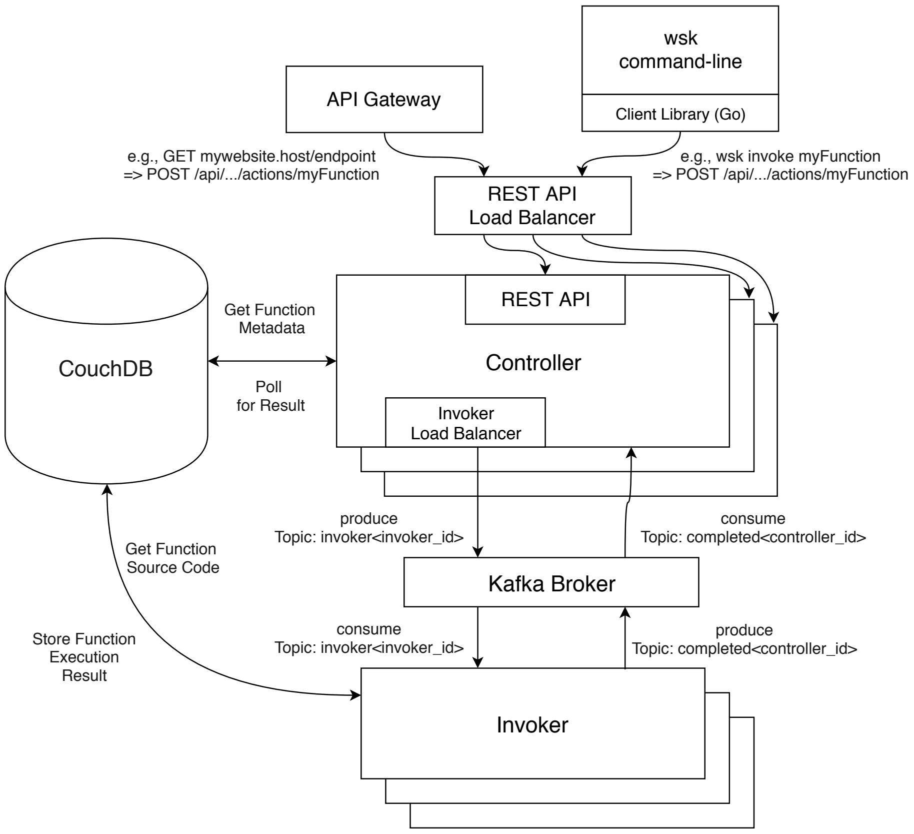

API Gateway
wsk
command-line
Client Library (Go)
e.g., GET mywebsite.host/endpoint
=> POST /api/.../actions/myFunction
e.g., wsk invoke myFunction
=> POST /api/.../actions/myFunction
REST API
Load Balancer
CouchDB
Get Function
Metadata
Poll
for Result
REST API
Controller
Invoker
Load Balancer
produce
Topic: invoker<invoker_id>
consume
Topic: completed<controller_id>
Get Function
Source Code
Kafka Broker
Store Function
Execution
Result
consume
Topic: invoker<invoker_id>
produce
Topic: completed<controller_id>
Invoker

Figure 1: Architectural diagram of all OpenWhisk components.

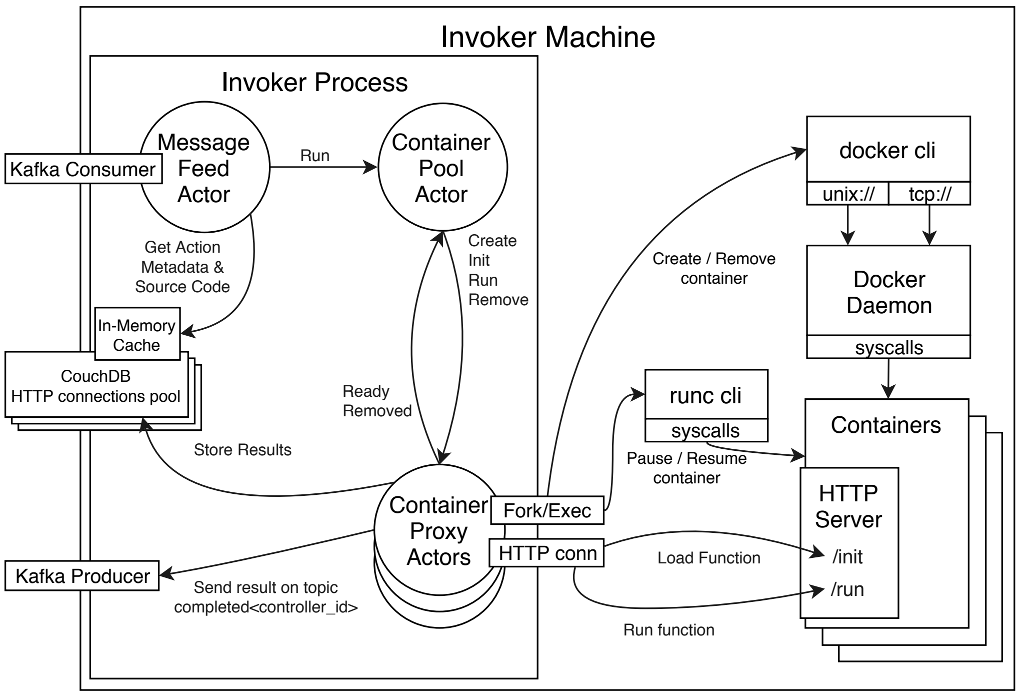


Figure 2: Architectural diagram of the OpenWhisk components on an Invoker machine.

Machine	Load Injector	Controller	Invoker
Model	DELL PowerEdge R905	DELL PowerEdge R815	DELL PowerEdge R815
Microarchitecture	-	Piledriver / Abu Dhabi	Bulldozer / Interlagos
Processor	AMD Opteron 8380	AMD Opteron 6344	AMD Opteron 6272
Frequency	2.5 GHz	2.6 GHz	2.1 GHz
Total # cores	16	48	64
Memory	32 GB	64 GB	128 GB
Type	DDR2	DDR3	DDR3
Frequency	667 MHz	1600 MHz	1600 MHz
# NUMA nodes	4	8	8
Linux kernel	v3.16.0	v4.14.0	v4.15.14
Docker	-	v1.11.2	v18.03.0-ce

Table 1: Hardware and software description of the machines used for the deployment of OpenWhisk.

Machine	Controller			Invoker
Component	Controller	Kafka	CouchDB	Invoker
Docker Image	(custom)	wurstmeister/kafka	apache/couchdb	(custom)
Version	-	0.11.0.1	2.1	-
JVM	Oracle	Oracle	-	Oracle
Version	8u141b15	8u144b01	-	8u141b15
GC	ParallelGC	G1GC	-	ParallelGC
Heap	16 GB	8 GB	-	64 GB

Table 2: Specifications of the software components.

Relative timestamp	Component involved	Event
0	Controller	Received execution request
0	Controller	Action metadata: Serving from cache
10	Controller	Load Balancer [start]
19	Controller	Sending function to Invoker0 through Kafka [start]
22	Controller	Sending function to Invoker0 through Kafka [end]
22	Controller	Load Balancer [end]
30	**Invoker**	**Received execution request from Kafka**
30	Invoker	Action source code: Serving from cache
35	Invoker	Sending request to ContainerPool actor
74	Invoker	ContainerPool actor starts processing the request
74	Invoker	Sending action to Container
80	Container	Function execution [start]
82	Container	Function execution [end]
85	Invoker	Received action results from Container
85	Invoker	Sending function to Invoker0 through Kafka [start]
91	**Invoker**	**Sending function to Invoker0 through Kafka [end]**
98	Controller	Sending response to client

Table 3: Simplified example of a trace gathered from OpenWhisk logs, showing consecutive events for the processing of an *hello-world* request.

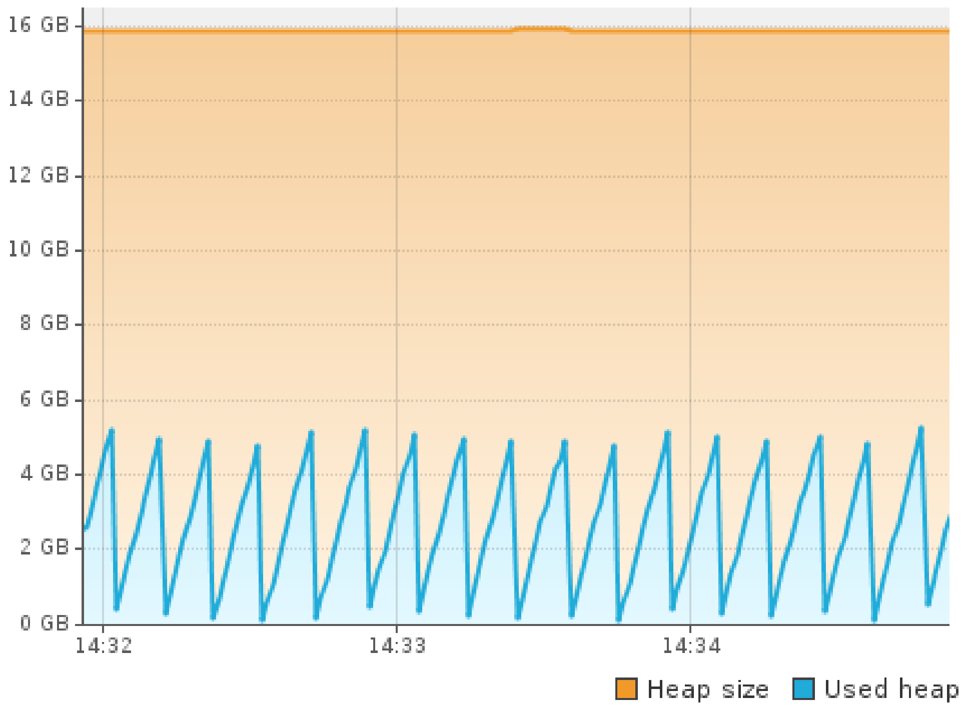


Figure 3: Evolution of used the Controller heap memory over time as displayed by VisualVM [77].

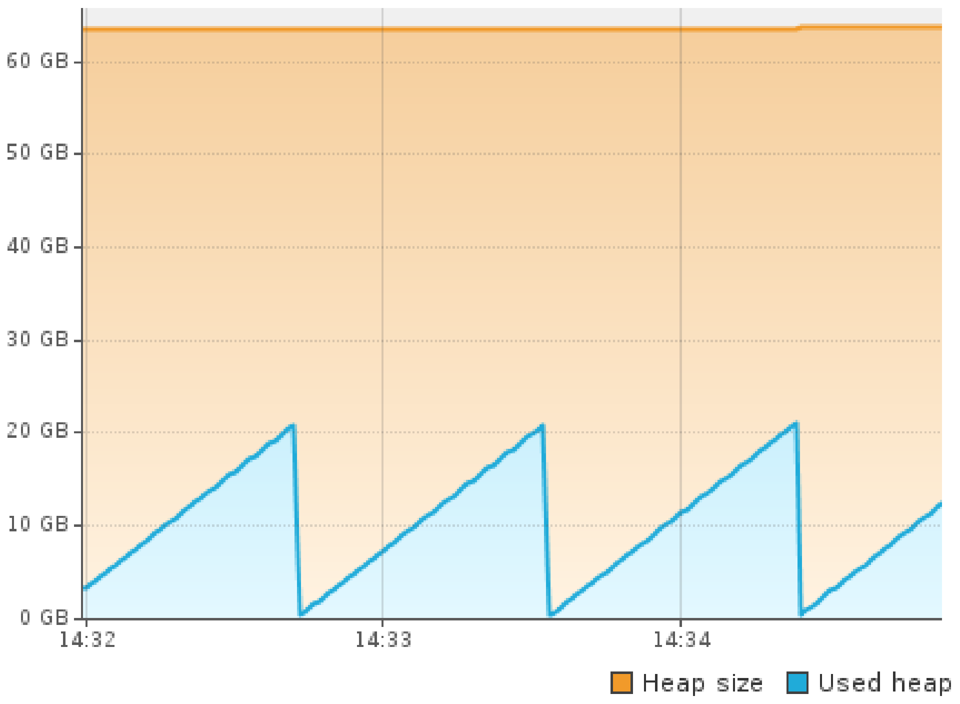


Figure 4: Evolution of used the Invoker heap memory over time as displayed by VisualVM [77].

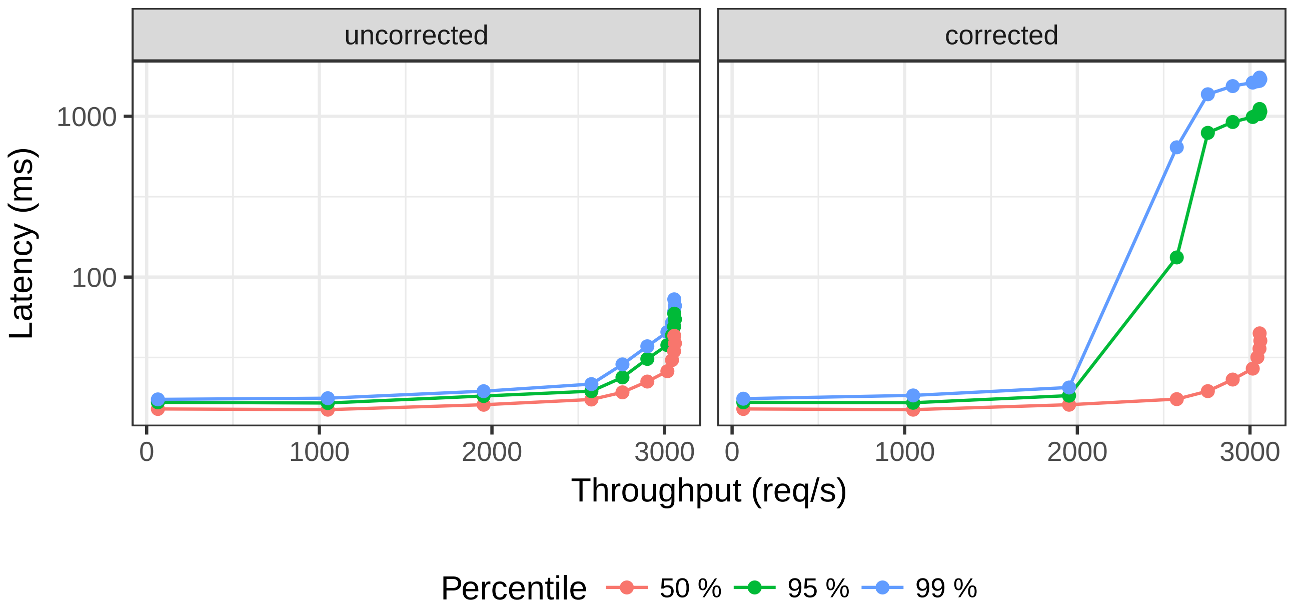


Figure 5: Throughput vs. Latency of the initial configuration of OpenWhisk. Note the logarithmic scale. These results were measured using the `wrk` closed-loop injector and illustrate the difference between the tail latency directly measured (uncorrected) and with the correction applied to account for the coordinated omission problem (corrected).

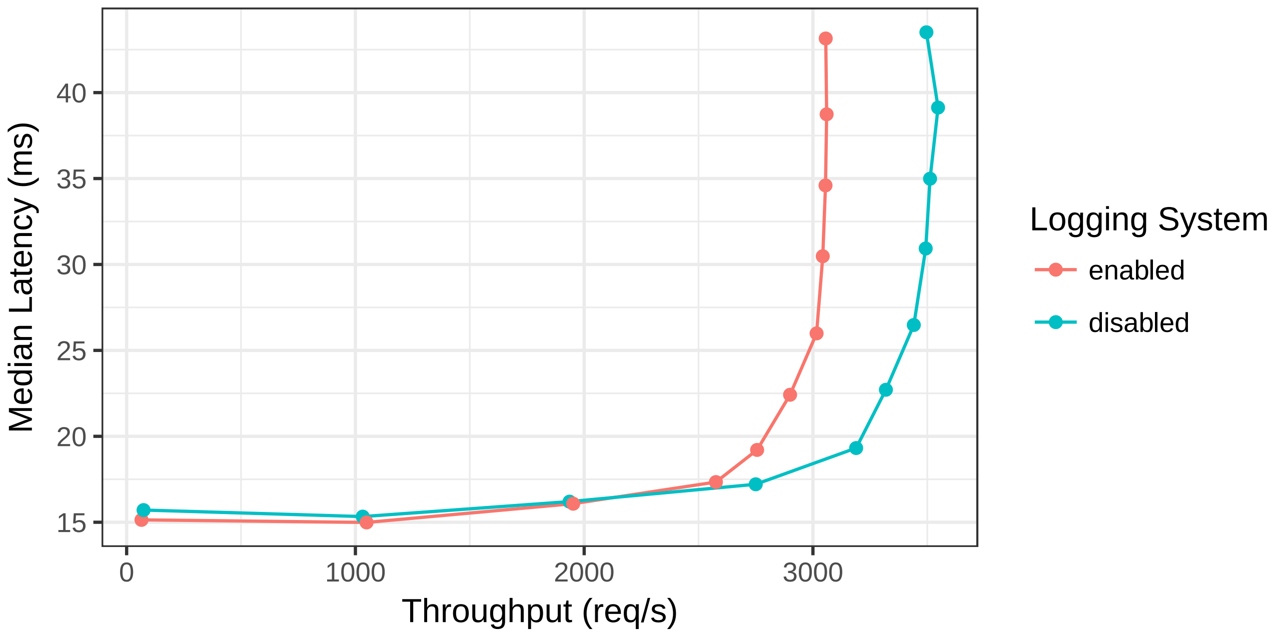


Figure 6: Comparison of the throughput vs. median latency before and after disabling the logging system of OpenWhisk.

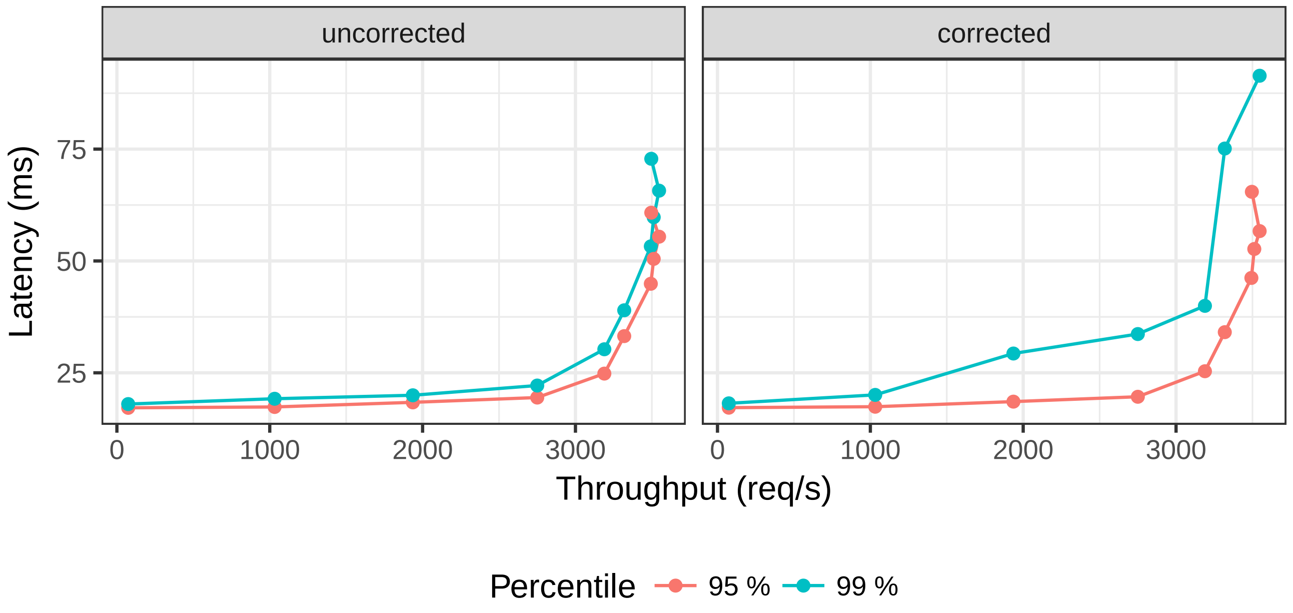


Figure 7: Comparison of the throughput vs. the tail latency with and without the correction of `wrk`. Note that the scale is not logarithmic (unlike in in Figure 5).

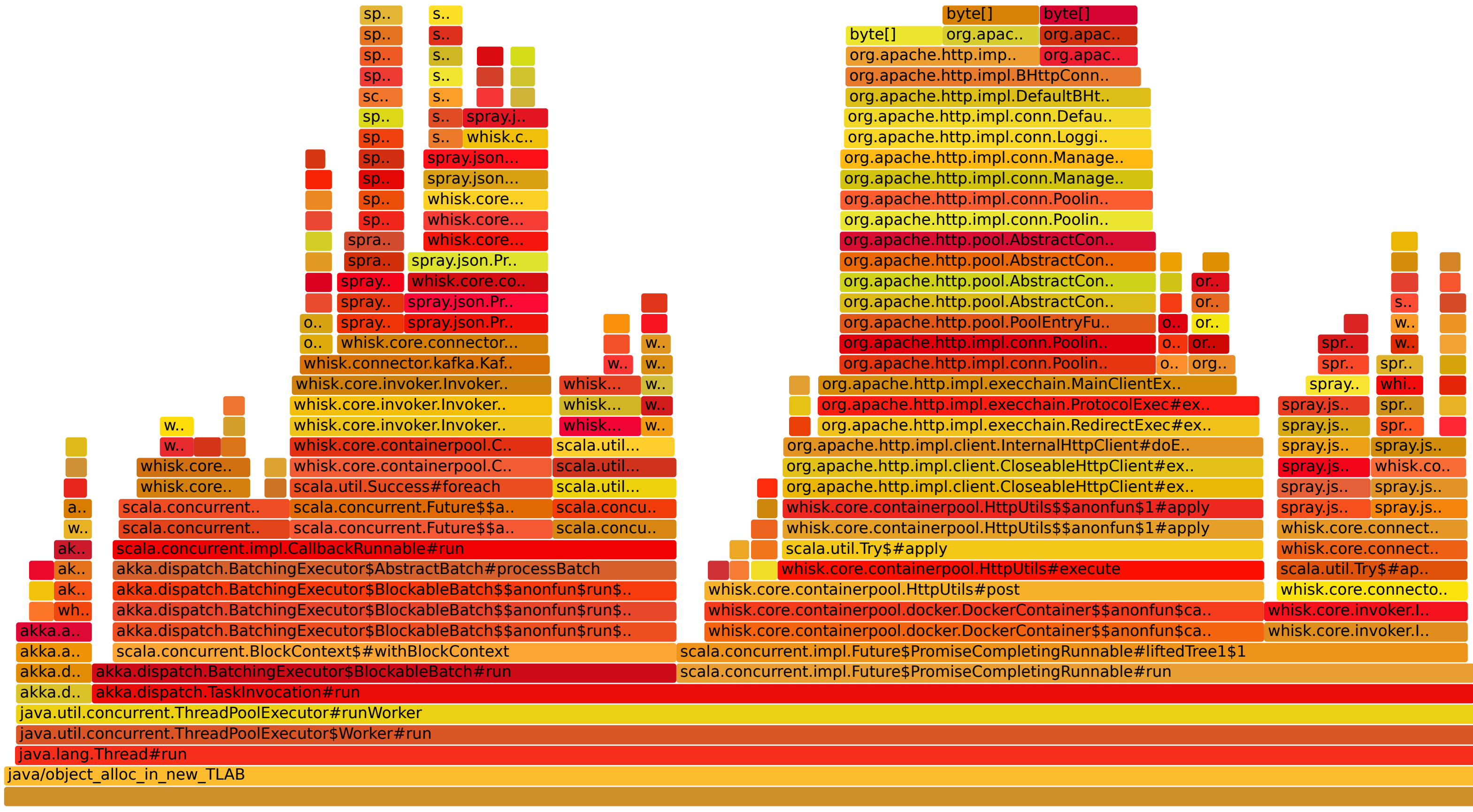


Figure 8: Allocation flame graph of the Invoker process. This flame graph only shows the *allocation in new TLAB* events as they represent 97% of the total of events. The colors are arbitrary. Some less relevant function calls (whose width is smaller than 5 pixels) are filtered out for clarity.

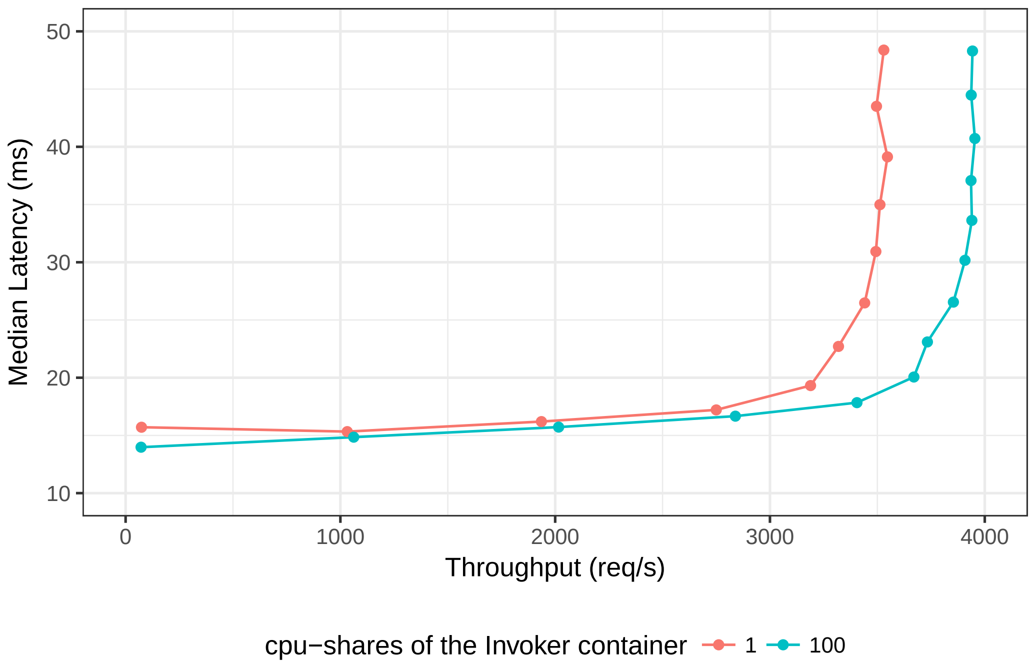


Figure 9: Comparison of the throughput vs. median latency with the initial value of `cpu-shares` for the Invoker process (1) and the value multiplied by one hundred (100).

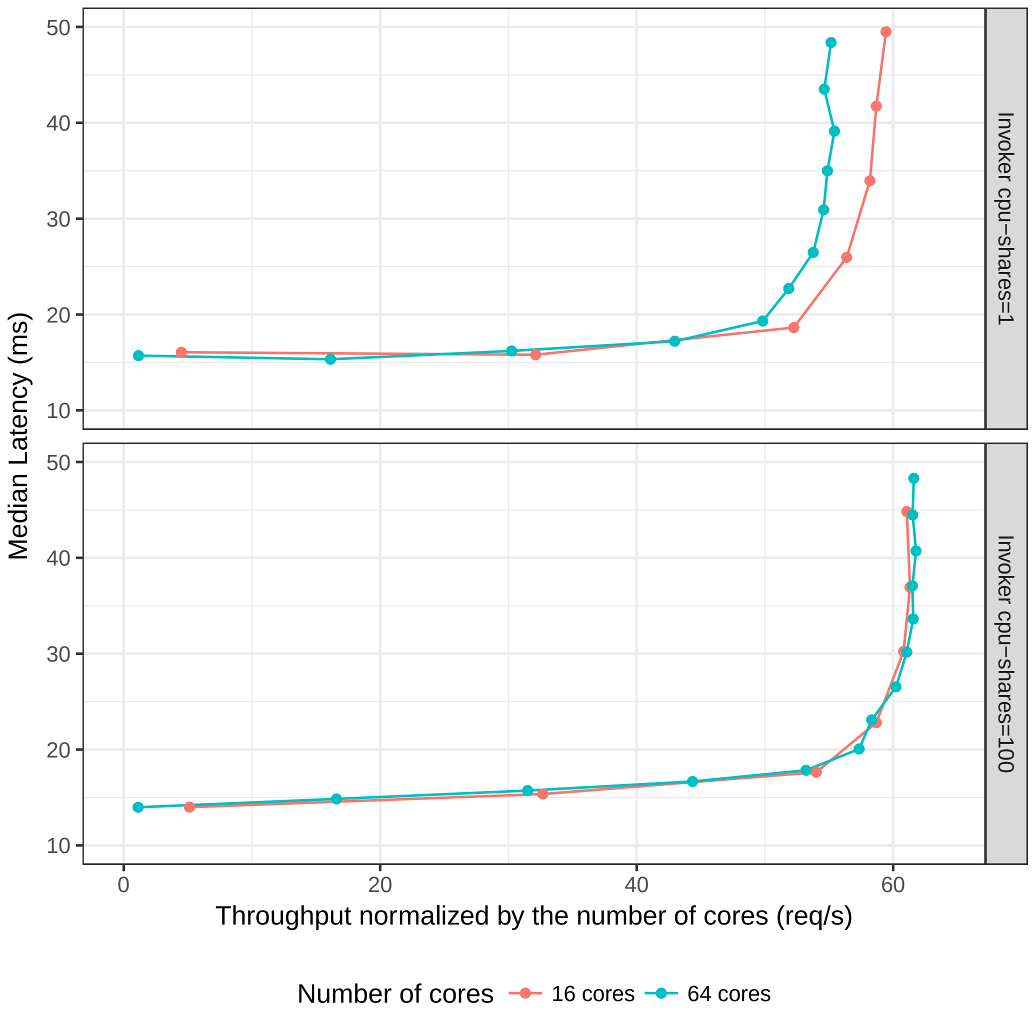


Figure 10: Comparison of the throughput per core vs. median latency, with a 16-core and a 64-core machine, with the initial value of `cpu-shares` for the Invoker process (1) and the value multiplied by one hundred (100).

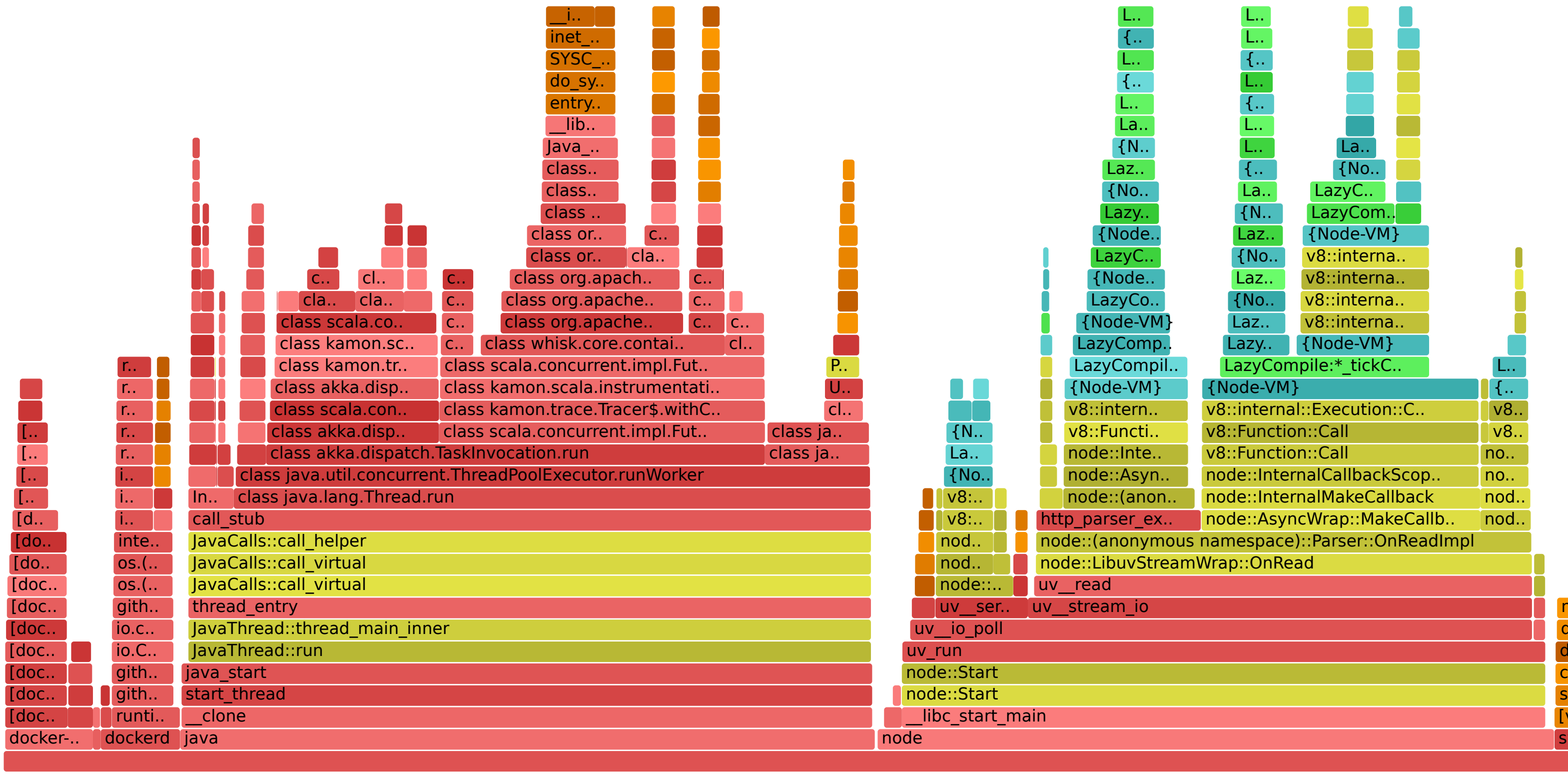


Figure 11: Flame graph of the most frequent code paths for the Invoker machine. The colors are arbitrary. Some less relevant function calls (whose width would be smaller than 2 pixels) are filtered out for clarity.

A Bibliography

Scientific Literature

[1] Ioana Baldini et al. “Serverless Computing: Current Trends and Open Problems”. In: *Research Advances in Cloud Computing*. Springer Singapore, 2017, pp. 1–20.

[2] Scott Hendrickson et al. “Serverless Computation with OpenLambda”. In: *Proceedings of the 8th USENIX Workshop on Hot Topics in Cloud Computing*. Hot-Cloud’16. Denver, CO, 2016.

[3] Neil Savage. “Going Serverless”. In: *Communications of the ACM* 61 (Jan. 2018), pp. 15–16.

[4] Erwin van Eyk et al. “A SPEC RG Cloud Group’s Vision on the Performance Challenges of FaaS Cloud Architectures”. In: *Companion of the 9th ACM/SPEC International Conference on Performance Engineering*. ICPE’18. Berlin, Germany, 2018.

[5] Edwin F. Boza. “Towards Improved Cloud Function Scheduling in Function-as-a-Service Platforms”. In: *Proceedings of the 12th EuroSys Doctoral Workshop*. EuroDW’18. Porto, Portugal, 2018.

[6] Simon Shillaker. “A Provider-Friendly Serverless Framework for Latency-Critical Applications”. In: *Proceedings of the 12th EuroSys Doctoral Workshop*. EuroDW’18. Porto, Portugal, 2018.

[7] Eric Jonas et al. “Occupy the Cloud: Distributed Computing for the 99%”. In: *Proceedings of the 9th Symposium on Cloud Computing*. SoCC’17. Santa Clara, California, 2017.

[8] Sadjad Fouladi et al. “Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads”. In: *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation*. NSDI’17. Boston, MA, 2017.

[9] Luiz André Barroso and Urs Hölzle. “The Case for Energy-Proportional Computing”. In: *Computer* 40 (Dec. 2007), pp. 33–37.

[10] Luiz André Barroso, Jimmy Clidaras, and Urs Hölzle. *The Datacenter As a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan & Claypool, 2013.

[11] Christina Delimitrou. “Improving Resource Efficiency in Cloud Computing”. PhD thesis. Stanford University, Aug. 2015.

[12] Ricardo Koller and Dan Williams. “Will Serverless End the Dominance of Linux in the Cloud?” In: *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*. HotOS’17. Whistler, BC, Canada, 2017.

[13] E. Oakes et al. “Pipsqueak: Lean Lambdas with Large Libraries”. In: *Proceedings of the 37th IEEE International Conference on Distributed Computing Systems Workshops*. ICDCSW’17. 2017.

[14] Filipe Manco et al. “My VM is Lighter (and Safer) Than Your Container”. In: *Proceedings of the 26th Symposium on Operating Systems Principles*. SOSP’17. Shanghai, China, 2017.

[15] Istemi Ekin Akkus et al. “SAND: Towards High-Performance Serverless Computing”. In: *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA: USENIX Association, 2018.

[16] Cristina L. Abad, Edwin F. Boza, and Erwin van Eyk. “Package-Aware Scheduling of FaaS Functions”. In: *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*. ICPE ’18. Berlin, Germany: ACM, 2018, pp. 101–106.

[17] Jeffrey Dean and Luiz André Barroso. “The Tail at Scale”. In: *Communications of the ACM* 56 (Feb. 2013), pp. 74–80.

[18] Jialin Li et al. “Tales of the Tail: Hardware, OS, and Application-level Sources of Tail Latency”. In: *Proceedings of the 5th ACM Symposium on Cloud Computing*. SoCC’14. Seattle, WA, USA, 2014.

[19] Jacob Leverich and Christos Kozyrakis. “Reconciling High Server Utilization and Sub-millisecond Quality-of-service”. In: *Proceedings of the 9th European Conference on Computer Systems*. EuroSys’14. Amsterdam, The Netherlands, 2014.

[20] Kenneth J. Duda and David R. Cheriton. “Borrowed-virtual-time (BVT) Scheduling: Supporting Latency-sensitive Threads in a General-purpose Scheduler”. In: *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles*. SOSP ’99. Charleston, South Carolina, USA: ACM, 1999, pp. 261–276.

[21] Esmail Asyabi et al. “CTS: An operating system CPU scheduler to mitigate tail latency for latency-sensitive multi-threaded applications”. In: *Journal of Parallel and Distributed Computing* (Apr. 2018).

[22] Jean-Pierre Lozi et al. “The Linux Scheduler: A Decade of Wasted Cores”. In: *Proceedings of the 11th European Conference on Computer Systems*. EuroSys’16. London, United Kingdom, 2016.

[23] Raj Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. Wiley, 1991.

[24] Gaurav Banga and Peter Druschel. “Measuring the Capacity of a Web Server Under Realistic Loads”. In: *World Wide Web* 2 (Jan. 1999), pp. 69–83.

[25] Bianca Schroeder, Adam Wierman, and Mor Harchol-Balter. “Open Versus Closed: A Cautionary Tale”. In: *Proceedings of the 3rd Conference on Networked Systems Design & Implementation*. NSDI’06. San Jose, CA, 2006.

[26] Marios Kogias, Christos Kozyrakis, and Edouard Bugnion. “Measuring Latency: Am I doing it right?” Presented at: 14th USENIX Symposium on Networked Systems Design and Implementation. NSDI’17. Boston, Massachusetts, USA. 2017.

[27] Brendan Gregg. *Systems Performance: Enterprise and the Cloud*. 1st ed. Prentice Hall Press, 2013.

[28] Sun Microsystems Inc. *Memory Management in the Java HotSpot(tm) Virtual Machine*. Tech. rep. Sun Microsystems Inc, 2006.

[29] Brendan Gregg. “The Flame Graph”. In: *Communications of the ACM* 59 (June 2016), pp. 48–57.

[30] Stephen Yang, Seo Jin Park, and John Ousterhout. “NanoLog: A Nanosecond Scale Logging System”. In: *Proceedings of the 2018 USENIX Annual Technical Conference*. ATC’18. Boston, MA, 2018.

[31] Oracle. *Deterministic Garbage Collection: Unleash the Power of Java with Oracle JRockit Real Time*. Tech. rep. Oracle, 2008.

[32] Gil Tene, Balaji Iyengar, and Michael Wolf. “C4: The Continuously Concurrent Compacting Collector”. In: *Proceedings of the International Symposium on Memory Management*. ISMM’11. San Jose, USA, 2011.

[33] Raymond Roestenburg, Rob Bakker, and Rob Williams. *Akka in Action*. Manning Publications, 2016.

[34] Will Dietz et al. “Slipstream: Automatic Interprocess Communication Optimization”. In: *Proceedings of the 201USENIX Annual Technical Conference*. USENIX ATC’15. Santa Clara, USA, 2015.

Internet Sources

[35] IBM. *How are EC2 instance-hours billed?* Oct. 26, 2017. URL: https://aws.amazon.com/premiumsupport/knowledge-center/ec2-instance-hour-billing/ (accessed: June 18, 2018).

[36] James Lewis and Martin Fowler. *Microservices*. Mar. 25, 2014. URL: https://martinfowler.com/articles/microservices.html (accessed: May 15, 2018).

[37] Bryan Cantrill. *Unikernels Are Unfit for Production*. Jan. 22, 2016. URL: http://dtrace.org/blogs/bmc/2016/01/22/unikernels-are-unfit-for-production/ (accessed: May 15, 2018).

[38] IBM. *OpenWhisk - Platform architecture*. May 18, 2018. URL: https://console.bluemix.net/docs/openwhisk/openwhisk_about.html#platform-architecture (accessed: June 3, 2018).

[39] Janakiram MSV. *An Architectural View of Apache OpenWhisk*. Feb. 3, 2017. URL: https://thenewstack.io/behind-scenes-apache-openwhisk-serverless-platform/ (accessed: June 3, 2018).

[40] Lightbend. *Akka - Distributed Data*. URL: https://doc.akka.io/docs/akka/current/distributed-data.html?language=scala (accessed: June 3, 2018).

[41] OpenWhisk Contributors. *OpenWhisk - about.md - Please form a line: Kafka*. May 27, 2018. URL: https://github.com/apache/incubator-openwhisk/blob/master/docs/about.md/#please-form-a-line-kafka (accessed: May 27, 2018).

[42] Lightbend. *Akka - Dispatchers*. URL: https://doc.akka.io/docs/akka/2.5/dispatchers.html (accessed: June 3, 2018).

[43] Tejun Heo. *LKDML.org - sched/fair: Fix O(# total cgroups) in load balance path*. Apr. 25, 2017. URL: https://lkml.org/lkml/2017/4/25/925 (accessed: June 4, 2018).

[44] Seetharami Seelam. *Docker at insane scale on IBM Power Systems*. Nov. 13, 2015. URL: https://www.ibm.com/blogs/bluemix/2015/11/docker-insane-scale-on-ibm-power-systems/ (accessed: June 8, 2018).

[45] Gil Tene. *How NOT to Measure Latency*. Jan. 24, 2014. URL: http://yowconference.com.au/slides/yow2014/Tene-HowNotToMeasureLatency.pdf (accessed: May 27, 2018).

[46] Marcus Hirt. *Allocation Profiling in Java Mission Control*. Sept. 12, 2013. URL: http://hirt.se/blog/?p=381 (accessed: May 27, 2018).

[47] Stephen Yeng. *NanoLog: A Nanosecond Scale Logging System*. Feb. 9, 2018. URL: https://platformlab.stanford.edu/Presentations/Yang__S.pdf (accessed: June 18, 2018).

[48] The Netty Project. *Reference counted objects - Netty project*. May 14, 2018. URL: http://netty.io/wiki/reference-counted-objects.html (accessed: June 15, 2018).

[49] Marcus Hirt. *Creating Custom JFR Events*. Nov. 10, 2013. URL: http://hirt.se/blog/?p=444 (accessed: May 27, 2018).

[50] Brendan Gregg. *Container Performance Analysis at DockerCon 2017*. Apr. 15, 2017. URL: http://www.brendangregg.com/blog/2017-05-15/container-performance-analysis-dockercon-2017.html (accessed: June 12, 2018).

[51] Brendan Gregg. *Java Flame Graphs*. June 12, 2014. URL: http://www.brendangregg.com/blog/2014-09-17/node-flame-graphs-on-linux.html (accessed: June 12, 2018).

[52] Brendan Gregg. *node.js Flame Graphs on Linux*. Sept. 17, 2017. URL: http://www.brendangregg.com/blog/2014-09-17/node-flame-graphs-on-linux.html (accessed: June 12, 2018).

[53] Andi Kleen. *Linux kernel - perf - jit-interface*. URL: https://github.com/torvalds/linux/blob/0fe7d7e9761ec7e23350b5543ddac470bb3cde1e/tools/perf/Documentation/jit-interface.txt (accessed: June 12, 2018).

[54] Anonymous. *Reverse engineering AWS Lambda*. May 31, 2018. URL: https://www.denialof.services/lambda/ (accessed: June 12, 2018).

Misc. Links

[55] Amazon. *AWS Lambda*. URL: https://aws.amazon.com/lambda/ (accessed: May 26, 2018).

[56] Microsoft. *Microsoft Azure Functions*. URL: https://azure.microsoft.com/en-us/services/functions/ (accessed: May 26, 2018).

[57] Google. *Google Cloud Functions (BETA)*. URL: https://cloud.google.com/functions/ (accessed: May 26, 2018).

[58] IBM. *IBM Cloud Functions*. URL: https://console.bluemix.net/openwhisk/ (accessed: May 26, 2018).

[59] OpenResty. *OpenResty*. URL: https://openresty.org/en/ (accessed: June 13, 2018).

[60] Nginx. *Nginx*. URL: http://nginx.org/ (accessed: June 13, 2018).

[61] Lua. *Lua*. URL: https://www.lua.org/ (accessed: June 13, 2018).

[62] Apache Software Foundation. *Apache CouchDB*. URL: http://couchdb.apache.org/ (accessed: June 13, 2018).

[63] Apache Software Foundation. *Apache Kafka*. URL: https://kafka.apache.org/ (accessed: June 13, 2018).

[64] Lightbend. *Akka*. URL: https://akka.io (accessed: May 27, 2018).

[65] Red Hat. *Ansible*. URL: https://www.ansible.com/ (accessed: June 4, 2018).

[66] M. Isuru Tharanga Chrishantha Perera. *jfr-flame-graph - Github*. URL: https://github.com/chrishantha/jfr-flame-graph (accessed: June 13, 2018).

[67] Platform Lab. *Granular Computing*. URL: https://platformlab.stanford.edu/platform-granular-computing.php (accessed: June 18, 2018).

[68] Lightbend. *akka-http*. URL: https://doc.akka.io/docs/akka-http/current/ (accessed: June 13, 2018).

[69] Oracle. *Oracle JRockit Real Time*. URL: http://www.oracle.com/technetwork/middleware/jrockit/overview/index-086343.html (accessed: June 13, 2018).

[70] Azul Systems. *Zing*. URL: https://www.azul.com/products/zing/ (accessed: June 13, 2018).

[71] The Netty Project. *Netty project*. URL: http://netty.io/ (accessed: June 15, 2018).

[72] Kamon. *Kamon - kamon-akka*. URL: http://kamon.io/documentation/1.x/instrumentation/akka/ (accessed: May 29, 2018).

[73] Datadog. *Statsd*. URL: https://www.datadoghq.com/blog/statsd/ (accessed: June 18, 2018).

[74] Graphite Labs. *Graphite*. URL: https://graphiteapp.org/ (accessed: June 18, 2018).

[75] Grafana Labs. *Grafana*. URL: https://grafana.com/ (accessed: June 18, 2018).

[76] Oracle. *About Java Flight Recorder*. URL: https://docs.oracle.com/javacomponents/jmc-5-4/jfr-runtime-guide/about.htm#JFRUH170 (accessed: May 28, 2018).

[77] Oracle. *VisualVM*. URL: https://visualvm.github.io/ (accessed: May 28, 2018).

[78] Vincent M Weaver. *The Unofficial Linux Perf Events Web-Page*. URL: http://web.eece.maine.edu/~vweaver/projects/perf_events/ (accessed: June 8, 2017).

Deciding Multivariate Polynomials Inequalities by combining Factorization, Euclidian Division and Handelman's Theorem

A.Delise, B.Grenet, A.Maréchal and M.Périn
Univ. Grenoble Alpes, CNRS, Grenoble INP*, VERIMAG, 38000 Grenoble, France

Abstract

Both assertion proof and automatic deciding need efficient way to solve problem containing multivariate polynomial inequalities. We are proposing in this paper new methods to fasten their resolution in certain cases.

1 Program verification using polynomial invariants

A property is an *invariant of a program point* if it holds in any execution whenever the program reaches that program point. Invariant property are the key of program verification. The goal of *static analysis of program* is to discover invariant properties. The goal of *satisfiability solvers modulo theory* (SMT) is to prove their invariance or to disprove it by producing a counter-example which can be used as a bug finder. Hence, program verification and bug finding is often built as a combination of a static analyzer and a SMT solver.

Multivariate polynomials on $\mathbb{Q}[x_1, \ldots, x_n]$ are sums of products of scalars in $\mathbb{Q}$ and indeterminate variables x_i, eg. $Q(x_1, x_2, x_3) \triangleq 1 + x_1x_2 + x_2^2 + x_2x_3^2$. Such constraints appear in static analysis of ... systems to capture invariant relations between program variables. For instance, the following constraint are invariants of a example program from [dOBP16] which provides recent progress on the generation of polynomial relations on program variables.

In this report, we focus on the SMT solver and more precisely on algorithms for deciding the satisfiability or unsatisfiability of systems mixing polyhedral and polynomial constraints. In other words, we consider a system of polyhedral constraints $\mathscr{A}$ and a system of polynomial constraints $\mathscr{Q}$ and we try to decided the satisfiability of the system $\mathscr{A} \cup \mathscr{Q}$.

1.1 Preliminaries

Notation We use boldface to denote vectors $\boldsymbol{x} = (x_1, \ldots, x_n) \in \mathbb{Q}^n$. We consider multivariate polynomials over the rationals. Polynomials are denoted by Q and affine forms (*i.e.* polynomial of degree ≤ 1) are denoted by A, eg. $A(\boldsymbol{x}) \triangleq a_0 + a_1x_1 + \ldots a_nx_n$ with $a_i \in \mathbb{Q}$.

Polynomial and polyhedral positivity constraints Without loss of generality we can focus on positivity constraint $Q(\boldsymbol{x}) \rhd 0$ where $\rhd \in \{>, \geq\}$. Indeed, an equality $Q(\boldsymbol{x}) = Q'(\boldsymbol{x})$ can be encoded as a conjunction of inequalities $Q(\boldsymbol{x}) \leq Q'(\boldsymbol{x}) \wedge Q(\boldsymbol{x}) \geq Q'(\boldsymbol{x})$, and every constraints $Q(\boldsymbol{x}) \rhd Q'(\boldsymbol{x})$ can be written as a positivity constraint $Q(\boldsymbol{x}) - Q'(\boldsymbol{x}) \rhd 0$. The negation of an inequality can also be turned into a positivity constraints: $\neg\,(Q(\boldsymbol{x}) > 0) \equiv Q(\boldsymbol{x}) \leq 0 \equiv -Q(\boldsymbol{x}) \geq 0$. However, disequalities require a special treatement as they introduce disjunctions.

A polyhedron is a linear system of inequalities, *i.e.* a conjunction of affine constraints $\{A_1(\boldsymbol{x}) \rhd 0, \ldots, A_p(\boldsymbol{x}) \rhd 0\}$. We use $\mathscr{A}$ to denote a polyhedron, $\mathscr{Q}$ to denote a system of polynomial inequalites $\{Q_1(\boldsymbol{x}) \rhd 0, \ldots, Q_q(\boldsymbol{x}) \rhd 0\}$. Hence, $\mathscr{A} \cup \mathscr{Q}$ refers to a system mixing p polyhedral constraints and q polynomial constraints

The notation $[\![\mathscr{A} \cup \mathscr{Q}]\!]$ is convenient to refer to the set of points satisfying the constraints of $\mathscr{A}$ and $\mathscr{Q}$:

$$[\![\mathscr{A} \cup \mathscr{Q}]\!] = \left\{ \boldsymbol{x} \in \mathbb{Q}^n \mid \bigwedge_{i=1}^{p} A_i(\boldsymbol{x}) \rhd 0 \ \bigwedge_{i=1}^{q} Q_i(\boldsymbol{x}) \rhd 0 \right\}$$

We also use $\mathscr{A}(\boldsymbol{x})$ for $\boldsymbol{x} \in [\![\mathscr{A}]\!]$ and $\mathscr{Q}(\boldsymbol{x})$ for $\boldsymbol{x} \in [\![\mathscr{Q}]\!]$.

Satisfiability An valuation is a function associating a value, denoted by $\underline{x_i}$, to each free variable, denoted by x_i. Given a theory, a logical formula φ is *satisfiable*, denoted by $\text{SAT}(\varphi)$ if there exists a valuation of the free variables $[x_1 \leftarrow \underline{x_1}, \ldots, x_\ell \leftarrow \underline{x_\ell}]$ such that $\varphi[x_1 \leftarrow \underline{x_1}, \ldots, x_\ell \leftarrow \underline{x_\ell}]$ holds. Otherwise the formula is said to be *unsatisfiable* denoted by $\text{UNSAT}(\varphi)$. A formula is *valid* if it holds for all possible instantiations. The following equivalences relates satisfiability, unsatisfiability and validity:

$$\neg\text{SAT}(\varphi) \quad \equiv \quad \text{UNSAT}(\varphi) \quad \equiv \quad \text{VALID}(\neg\varphi)$$

The value of a formula $\varphi(\boldsymbol{x})$ under a valuation of the vector $\boldsymbol{x} = (x_1, \ldots, x_n)$ is written $\varphi(\underline{\boldsymbol{x}})$ as a shortcoming for $\varphi[x_1 \leftarrow \underline{x_1}, \ldots, x_n \leftarrow \underline{x_n}]$.

1.2 Reductions of mixed polyhedral and polynomial constraints

Given a system of polyhedral constraints $\mathscr{A} = \{A_i(\boldsymbol{x}) \rhd 0 \mid i = 1..p\}$ and a system of polynomial constraints $\mathscr{Q} = \{Q_i(\boldsymbol{x}) \triangleright 0 \mid i = 1..q\}$ SMT must decide the question $\text{SAT}(\mathscr{A} \cup \mathscr{Q})$ which is equivalent to $\exists \boldsymbol{x},\ \mathscr{A}(\boldsymbol{x}) \wedge \mathscr{Q}(\boldsymbol{x})$ which expands to

$$\exists \boldsymbol{x},\ \bigwedge_{i=1}^{p} A_i(\boldsymbol{x}) \triangleright 0 \ \bigwedge_{i=1}^{q} Q_i(\boldsymbol{x}) \triangleright 0 \tag{1}$$

The general way for tackling such a problem is to separate polyhedral constraints from polynomial ones. Polyhedral constraints are used as assumptions under which the solver tries to decide the satisfiability of $\mathscr{Q}$. This requires some explanation as this resolution technique only provides a sufficient condition. Indeed, (1) is implied by (but not equivalent to) the following conjunction

$$\exists \boldsymbol{x},\ \mathscr{A}(\boldsymbol{x}) \ \wedge\ \forall \boldsymbol{x},\ \mathscr{A}(\boldsymbol{x}) \Rightarrow \mathscr{Q}(\boldsymbol{x}) \tag{2}$$

Remark 1 *This equivalence holds if we consider polyhedra $\mathscr{A}'$ included in $\mathscr{A}$, denoted by $\mathscr{A}' \sqsubseteq \mathscr{A}$.*

$$\exists \boldsymbol{x},\ \mathscr{A}(\boldsymbol{x}) \wedge \mathscr{Q}(\boldsymbol{x}) \Leftrightarrow \exists \mathscr{A}',\ \mathscr{A}' \sqsubseteq \mathscr{A} \wedge \exists \boldsymbol{x},\ \mathscr{A}'(\boldsymbol{x}) \wedge \forall \boldsymbol{x},\ \mathscr{A}'(\boldsymbol{x}) \Rightarrow \mathscr{Q}(\boldsymbol{x})$$

Proof ($\Leftarrow$) is obvious. For ($\Rightarrow$), assume that $\underline{\boldsymbol{x}}$ satisfies $\mathscr{A}(\underline{\boldsymbol{x}}) \wedge \mathscr{Q}(\underline{\boldsymbol{x}})$ then we can defined $\mathscr{A}' = \{\underline{x_i} \leq x_i \leq \underline{x_i} \mid i = 1..n\}$ which encode the equality $\boldsymbol{x} = \underline{\boldsymbol{x}}$. □

For proving $\text{SAT}(\mathscr{A} \cup \mathscr{Q})$ it is sufficient to prove (2) which can be reformulated as $\text{SAT}(\mathscr{A}) \wedge \text{VALID}(\forall \boldsymbol{x},\ \mathscr{A}(\boldsymbol{x}) \Rightarrow \mathscr{Q}(\boldsymbol{x}))$ and, since satisfiability of polyhedron $\mathscr{A}$ is efficiently handled by solvers, we will focus on the validity of

$$\forall \boldsymbol{x},\ \mathscr{A}(\boldsymbol{x}) \Rightarrow \mathscr{Q}(\boldsymbol{x}). \tag{3}$$

Exploiting the fact that $\mathscr{Q}$ is a conjunction of positivity constraints $\bigwedge_{i=1}^{q} Q_i(\boldsymbol{x}) \triangleright 0$, we can prove (see §A) that (3) is equivalent to a conjunction of subproblems consisting in establishing the sign of a polynomial Q_i on a polyhedral space $\mathscr{A}$:

$$\bigwedge_{i=1}^{q} (\forall \boldsymbol{x},\ \mathscr{A}(\boldsymbol{x}) \Rightarrow Q_i(\boldsymbol{x}) \triangleright 0) \tag{4}$$

Summarizing: we reduced the initial satisfiability problem (1) to the validity of implication in the form of (4). Section §2 is dedicated to the algorithms used to provide evidence that the constraints of $\mathscr{A}$ entail the positivity of Q_i. Basically, we search for rewritings of Q_i in an obvious positive form using constraints of $\mathscr{A}$.

Unsatisfiability While trying to prove satisfiability, typical solvers also launch processes that search for a proof of unsatisfiability. All processes can be non-terminating and are afforded a limited exploration time. If an UNSAT process terminates with a proof, all others processes are killed and the solver returns UNSAT. If all SAT in the opposite case it returns UNSAT; and if both processes reach their timeout, the solver returns DONT KNOW.

The same separation of concerns applies in the UNSAT case. We first establish the equivalence – no valuable intuition in its proof, just logical equivalence (see §A)

$$\text{UNSAT}(\mathscr{A} \wedge \mathscr{Q}) \ \equiv\ \text{UNSAT}(\mathscr{A}) \vee (\forall \boldsymbol{x},\ \mathscr{A}(\boldsymbol{x}) \Rightarrow \neg\mathscr{Q}(\boldsymbol{x})) \tag{5}$$

from which we derive a sufficient condition (6) for proving $\forall \boldsymbol{x},\ \mathscr{A}(\boldsymbol{x}) \Rightarrow \neg\mathscr{Q}(\boldsymbol{x})$ where $\mathscr{Q} = \bigwedge_{i=1}^{q} \mathcal{Q}_i$ and $\mathcal{Q}_i(\boldsymbol{x})$ is just a shortcut for $Q_i(\boldsymbol{x}) \triangleright 0$.

$$\bigvee_{i=1}^{q} \left(\forall \boldsymbol{x},\ \left(\mathscr{A}(\boldsymbol{x}) \bigwedge_{j=1, j\neq i}^{q} \mathcal{Q}_j(\boldsymbol{x}) \right) \Rightarrow \neg \mathcal{Q}_i(\boldsymbol{x}) \right) \tag{6}$$

This reformulation requires some explanations (see §A) but let us first explain its benefit for proving unsatisfiability: Exploiting (5) the UNSAT process attempts to prove the unsatisfiability of $\mathscr{A}$. If that fails, the problem boils down to $\forall \boldsymbol{x},\ \mathscr{A}(\boldsymbol{x}) \Rightarrow \neg\mathscr{Q}(\boldsymbol{x})$. Thanks to (6), it is sufficient to succeed on one case of the disjunction to conclude to unsatifiability. Thus, the solver runs q independant processes adressing a subproblem of the form

$$\forall \boldsymbol{x},\ \mathscr{A}(\boldsymbol{x}) \wedge \mathscr{Q}'(\boldsymbol{x}) \Rightarrow \neg\left(Q_i(\boldsymbol{x}) \triangleright 0\right) \tag{7}$$

The problem has been simplified since the goal is to determine the sign of *one polynomial Q_i* under *a greater number assumptions* consisting in polyhedral constraints $\mathscr{A}$ and polynomials ones $\mathscr{Q}' \triangleq \bigwedge_{j=1, j\neq i}^{q} \mathcal{Q}_j$. Now the goal of the algorithm is to find a appropriate rewriting of Q_i as ($\pm$) a sum of product of positivity constraints of $\mathscr{A} \cup \mathscr{Q}'$ so that the sign of Q_i becomes obvious.

Finally, using simple tricks we reduced the SAT and UNSAT questions (1, resp. 5) to the validity of implications (4, resp. 7) of the form

$$\forall \boldsymbol{x},\ \mathscr{A}(\boldsymbol{x}) \wedge \mathscr{Q}'(\boldsymbol{x}) \Rightarrow Q(\boldsymbol{x}) \triangleright 0 \tag{8}$$

Now we focus on the next phase of our solver: finding an appropriate rewriting of Q in terms of constraints of $\mathscr{A} \cup \mathscr{Q}'$.

2 Interesting polynomial positive forms

We present reasonings on polynomial forms that help deciding the satisfiability of systems mixing polyhedral and polynomial constraints.

Our goal is to find decomposition of Q that ease the resolution. We investigated three decompositions

1. *Handelman form.* Given a system of positivity constraints $\mathscr{A} \cup \mathscr{Q} = \bigwedge_{i=1}^{p} \mathcal{A}_i \bigwedge_{i=1}^{q} \mathcal{Q}_i$, Handelman's theorem says that a polynomial Q is positive on $\mathscr{A} \cup \mathscr{Q}$ if it is a positive linear combination of products of positive constraints on $\mathscr{A} \cup \mathscr{Q}$. It ensures that there exist some rewritings of Q in the form of $\sum \lambda_{(\boldsymbol{k},\boldsymbol{\ell})} \Pi_{i=1}^{p} \mathcal{A}_i^{k_i} \Pi_{j=1}^{q} \mathcal{Q}_j^{\ell_j}$ from which it becomes obvious to prove its positivity as a sum of products of positive terms. Obviously it is possible to add any square polynomials as a member of the system $\mathscr{Q}$. This is done in an opportunistic way in our solver.
2. *Factorization.* If $Q = Q_1 \times Q_2$ then
$$Q \triangleright 0 \equiv (Q_1 \triangleright 0 \wedge Q_2 \triangleright 0) \vee (-Q_1 \triangleright 0 \wedge -Q_2 \triangleright 0)$$
3. *Decomposition over an Ideal $\langle \mathscr{A} \rangle$.* $Q \in \langle A_1, \ldots, A_p \rangle$ iff there exist polynomials $Q'_1, \ldots, Q'_p$ such that $Q = \sum_{i=1}^{p} Q'_i \times A_i$. Remind that A_i are positivity constraints of $\mathscr{A}$; thus, Q is positive if all the Q'_i are.

2.1 Factorization

Let $\mathscr{A}'$ be a set of affine forms $\{A'_i \mid i = 1..p'\}$ and assume that Q has a factorization: $Q = Q'{\times}\Pi\mathscr{A}' = Q'{\times}\Pi_{i=1}^{p'}A'_i$. Then, the validity problem $\forall \boldsymbol{x},\ \mathscr{A} \wedge \mathscr{Q} \Rightarrow Q \rhd 0$ can be rephrased

$$\forall \boldsymbol{x},\ \mathscr{A} \wedge \mathscr{Q} \Rightarrow \left(Q'{\times}\Pi\mathscr{A}'\right) \rhd 0. \tag{9}$$

Let us introduce some notations to expose the reformulation. Let $s_i \in \{-1, +1\}$ denote signs. The sign of $\Pi\mathscr{A}'$ is obviously positive on the polyhedron $\bigwedge_{i=1}^{p'}(s_i \times A_i) \rhd 0$ with $s_i = +1$ for all i. We can generalize the reasoning by splitting the sign study onto a disjunction of polyhedra: $\bigvee_{(s_1,\ldots,s_{p'})\in\{-1,1\}^{p'}} \mathscr{A}'_{(s_1,\ldots,s_{p'})}$ where $\mathscr{A}'_{(s_1,\ldots,s_{p'})} \triangleq \bigwedge_{i=1}^{p'}(s_i \times A'_i) \rhd 0$ is a polyhedron made of the constraints of $\mathscr{A}'$ with signs settings. The sign of $\Pi\mathscr{A}'$ is $\Pi_{i=1}^{p'}s_i$ on $\mathscr{A}'_{(s_1,\ldots,s_{p'})}$.

Proof Note that, by construction, $\Pi_{i=1}^{p'}(s_i \times A_i)$ is positive on $\mathscr{A}'_{(s_1,\ldots,s_{p'})}$. Then, on $\mathscr{A}'_{(s_1,\ldots,s_{p'})}$, $\Pi\mathscr{A}' = \Pi_{i=1}^{p'}A_i = \Pi_{i=1}^{p'}s_i^2 \times A_i = \left(\Pi_{i=1}^{p'}s_i\right) \times \left(\Pi_{i=1}^{p'}(s_i \times A_i)\right) = \Pi_{i=1}^{p'}s_i$ □

Some sign settings give empty polyhedra. For instance $s_1{\times}x \geq 0 \wedge s_2{\times}(1-x) \geq 0$ has solutions for $(s_1, s_2) \in \{(1,1),(1,-1),(-1,1)\}$, it respectively corresponds to $0 \leq x \leq 1$, $x \geq 1$, $x \leq 0$, but it is empty for $(s_1, s_2) = (-1,-1)$ which stands for $1 \leq x \leq 0$. Let $\mathcal{P}$ denotes the set of sign settings $\boldsymbol{s} \in \{-1,+1\}^{p'}$ such that $\mathscr{A}'_{\boldsymbol{s}}$ is non-empty and $\Pi_{i=1}^{p'}s_i = +1$. Similarly $\mathcal{N}$ denotes the sign settings of non-empty polyhedra with $\Pi_{i=1}^{p'}s_i = -1$. Then, $Q'{\times}\Pi\mathscr{A}'$ is positive iff Q' is positive on $\mathcal{P}$ and negative on $\mathcal{N}$. Formally,

$$Q'{\times}\Pi\mathscr{A}'{\rhd}0 \ \equiv\ \wedge \begin{array}{l} \forall \mathscr{A}'_{\boldsymbol{s}} \in \mathcal{P},\ \forall \boldsymbol{x},\ \boldsymbol{x} \in \mathscr{A}'_{\boldsymbol{s}} \Rightarrow Q'(\boldsymbol{x}) \rhd 0 \\ \forall \mathscr{A}'_{\boldsymbol{s}} \in \mathcal{N},\ \forall \boldsymbol{x},\ \boldsymbol{x} \in \mathscr{A}'_{\boldsymbol{s}} \Rightarrow -Q'(\boldsymbol{x}) \rhd 0 \end{array}$$

Therefore, the problem (9) $\forall \boldsymbol{x},\ \mathscr{A} \wedge \mathscr{Q} \Rightarrow Q'{\times}\Pi\mathscr{A}' \rhd 0$ can be split into $|\mathcal{P}| + |\mathcal{N}$ subproblems

$$\begin{array}{ll} \forall \boldsymbol{x},\ \mathscr{A}'_{\boldsymbol{s}} \wedge \mathscr{A} \wedge \mathscr{Q} \Rightarrow Q' \rhd 0 & \text{for } \mathscr{A}'_{\boldsymbol{s}} \in \mathcal{P} \\ \forall \boldsymbol{x},\ \mathscr{A}'_{\boldsymbol{s}} \wedge \mathscr{A} \wedge \mathscr{Q} \Rightarrow -Q' \rhd 0 & \text{for } \mathscr{A}'_{\boldsymbol{s}} \in \mathcal{N} \end{array}$$

3 Experimentations

We study the impact of factorization on solvers: is it easier for an off-the-shelf solver (like Z3) to answer the question $Q \rhd 0$ knowing that $Q = Q_1{\times}Q_2$? An experimental study will tell that this information is not exploited by the standard resolution method called CAD (for Cylindric Algrebraic Decomposition). Roughly speaking the CAD can be thought as a usual study of the variation of a function, which is complicated by the fact that the domain the multiviariate polynomial function has numerous dimensions. We implement the rule of signs learned at high school in the Z3 SMT solver and measures a gain of performance. We use the SAGE math library for factorization. We explain how the rule of sign can be elegantly encoded using exclusive-or. It is not suprising since exclusive-or as strong connection with parity problems. We discovered that parity problems are intrisically difficult to solve as [GK14] shown it and are still an active research domain like in this study: [LJN13; HJ12]. This can explain that factorization has not been considered in SMT solver. However, Gwynne etal identified classes of parity problems for which the resolution can be efficient.

3.1 Xor Form Correctness

According the sign rule, a non null factor product is negative if and only if an odd number of its factor is negative. So we have to treat apart the null case. In the non null case, we can rewrite the sign rule using the Xor operator.

Proof We want to prove by recursion the following assertion:
A_n :
$\neg\left(\bigoplus_{i=0}^{n}(f_i < 0))\right) \wedge \bigwedge_{i=0}^{n} f_i \neq 0$
$\equiv \prod_{i=0}^{n}(f_i) > 0$
Initialization:
$\neg((f_0 > 0) \oplus (f_1 > 0)) \wedge f_0 \neq 0 \wedge f_1 \neq 0$
$\equiv ((f_0 > 0) \wedge (f_1 > 0) \vee \overline{(f_0 > 0)} \wedge \overline{(f_1 > 0)}) \wedge f_0 \neq 0 \wedge f_1 \neq 0$
$\equiv ((f_0 > 0) \wedge (f_1 > 0) \vee (f_0 <= 0) \wedge (f_1 <= 0)) \wedge f_0 \neq 0 \wedge f_1 \neq 0$
$\equiv ((f_0 > 0) \wedge (f_1 > 0) \vee (f_0 < 0) \wedge (f_1 < 0)) \wedge f_0 \neq 0 \wedge f_1 \neq 0$
By the sign rule:
$\equiv \prod_{i=0}^{1}(f_i) > 0 \wedge f_0 \neq 0 \wedge f_1 \neq 0$
$\equiv \prod_{i=0}^{1}(f_i) > 0$
We had initialized the assertion A_0 so let see for any A_{n+1} supposing A_n is true.
Recursion:
$\neg\left(\bigoplus_{i=1}^{n+1}(f_i < 0))\right) \wedge \bigwedge_{i=0}^{n+1} f_i \neq 0$
$\equiv \neg\left(f_{n+1} < 0 \oplus \bigoplus_{i=1}^{n}(f_i < 0))\right) \wedge \bigwedge_{i=0}^{n+1} f_i \neq 0$
$\equiv \neg\left(f_{n+1} < 0 \oplus \prod_{i=0}^{n}(f_i) > 0\right) \wedge \bigwedge_{i=0}^{n+1} f_i \neq 0$
Let's set $F' = \prod_{i=0}^{n}(f_i)$ we recognize the same formula as in the initialization phase, so we conclude
$\equiv \neg(f_{n+1} > 0 \oplus F' > 0) \wedge \bigwedge_{i=0}^{n+1} f_i \neq 0$
$\equiv f_{n+1} * F' > 0$
$\equiv f_{n+1} * \prod_{i=0}^{n}(f_i) > 0$
$\equiv \prod_{i=0}^{n+1}(f_i) > 0$
So we have prove if A_n true, A_{n+1} is true, but A_0 is true so by recursion A_n is true for any n which mean: $\forall n \in \mathrm{N},\ \neg\left(\bigoplus_{i=0}^{n}(f_i < 0))\right) \wedge \bigwedge_{i=0}^{n} f_i \neq 0$
$\equiv \prod_{i=0}^{n}(f_i) > 0$
□

This mean of writing the rule sign show well the necessity of treating in dis-junction the null case. The Xor form has the advantage that the solveur can't ignore the rule sign and have to solve it in order to solve the problem.

3.2 Xor form performance

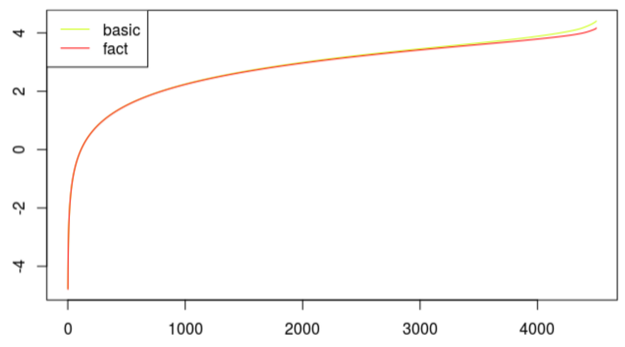
In order to test the efficiency of the Xor form we have to use some problems to test on them. We choose to not use usual workbench because all of them are designed for a particular characteristic but none of them were relevant in our case. Indeed most of them didn't have any constraint with polynomial of degree higher than 2. So we decide to generate our own problems to run this study. We designed an problem generator capable of generating some batch of problems in both classic and Xor form.

Definition 1 *Classic problem form*
$(x\_irr_1..x\_irr_a) \in \Re^a$
$(x_1..x_b) \in \Re^b$
$\forall n \in [\![1, c+d*a]\!], Aff_n = \sum_{i=0}^{b} const_{n,i} * x_i$
$\bigcap_{1\leq n\leq a} Pirr(x\_irr_n) * \prod_{i=1}^{d}(Aff_{i+c+1+d*(n-1)}) < 0$

Definition 2 *Xor problem form*
$(x\_irr_1..x\_irr_a) \in \Re^a$
$(x_1..x_b) \in \Re^b$
$\forall n \in [\![1, c+d*a]\!], Aff_n = \sum_{i=0}^{b} const_{n,i} * x_i$
$\bigcap_{1\leq n\leq a} Pirr(x\_irr_a) < 0 \oplus (\bigoplus_{i=c+1}^{c+d*a}(Aff_i < 0)))$
$\bigcap_{1\leq n\leq a} Pirr(x\_irr_n) \neq 0$
$\bigcap_{c+1\leq n\leq c+d*a} Aff_n \neq 0$

As it is proven, use factorized instead of developed polynomial in problems doesn't impact z3 efficiency we use the first one to fasten the generation. As a reminder Figure 1 is a plot of the efficiency comparison of both for z3.

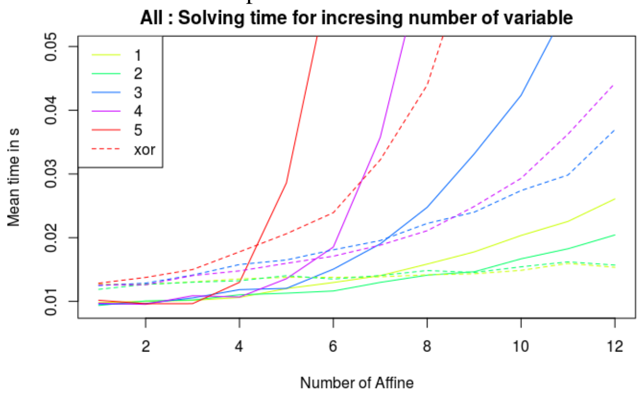
Figure 1: Cactus plot of z3 efficiency for factorized and developed formed



3.3 Xor problem efficiency

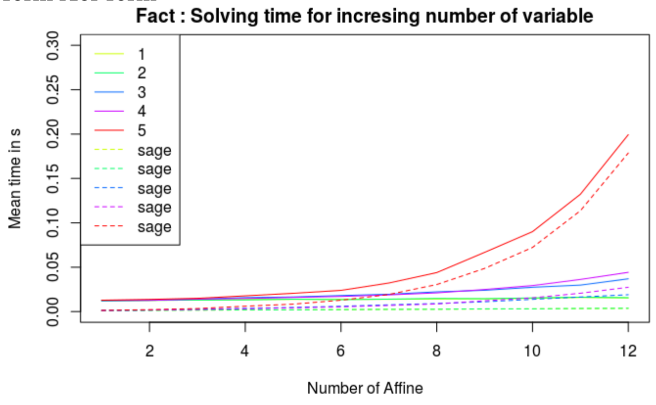
As advanced SMT solver are erratic by essence due to theory behaviour, the study test several combination of characteristic. The result is parameter dependant.
Figure 2 represent the time taken by z3 to solve problem in raw and Xor form. The curve color show the number variable of problems. The Xor form based on sage factorization algorithm start to be better than pure CAD between degree of 4 and 8 depending on the number of variable of the polynomial. The higher the number of variable is, more efficient the Xor form is, in comparison to the CAD. It can be explained by the fact, as previously said, the CAD doesn't manage very well multivariate polynomial. Multivariate constraint used in these study are pure linear constraint product, study including irreducible part were non conclusive due to both method seemed to struggle the same way resolving it which lead to same results.

Figure 2: Comparison between mean raw form resolution time and Xor form one plus factorization time



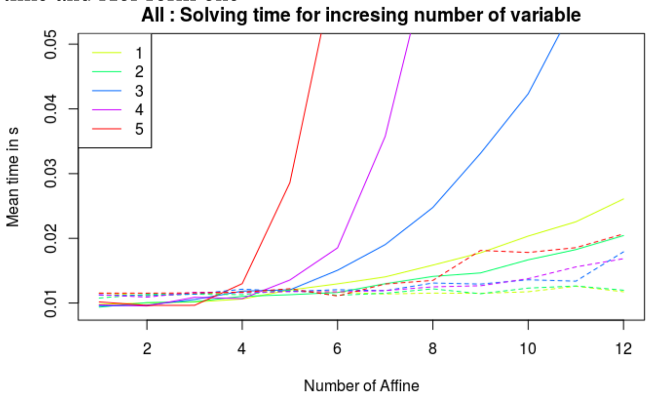
The current drawbacks of the method are it is worth only polynomial with many variable and an high degree, and its incompressible cost. With Figure 3 have the proportion of the sage factorization from resolution time. It appear half of the incompressible time and 70% or more of the time is taken by the factorization which mean many efficiency improvement can be done here.

Figure 3: Proportion of time taken by the factorization phase form Xor form



For instance Figure 4 is the same plot as Figure 2 despite the factorization time is ignored. There is a really substantial possible profit starting to degree of 3 which make interesting the research of different method of more efficient factorization.

Figure 4: Comparison between mean raw form resolution time and Xor form one



An ongoing work pursue by Bernard Grenet of LIRMM (IT laboratory of Montpellier) and Michäel Périn of Verimag show it exist a fast algorithm of linear factorization which would be an improvement for the Xor form, first because it could be directly implemented in the solveur reducing the incompressible cost, then because complete factorization algorithm are necessarily cost-full but the information gain is not worth in time consumed to compute. Focusing only on linear factor allow the algorithm to be really quick, more linear factor are worth for the other method discussed in this article too, as Handelman decomposition, the gain can be enhanced by side effect when combined with theses ones.

4 HSat

We developed a SMT solver to test more freely our methods, especially the method using Handelman theory. We used VPL to implement an polyhedral based theory for the solving and performed some test with it. We setup an experimentation process to study VPL behaviour and Handelman decomposition. The study showed up some heuristic VPL limitation and which need an update in order to redoing the study and conclude on Handelman decomposition efficiency.

4.1 Handelman SAT

The goal was to create a SMT solver based on the library of polyhedral calculus, the VPL, developed and maintained by the Verimag laboratory. As the VPL is written in Ocaml we decide to use a SAT solver in the same language for efficiency reason. We choose MSat, MSat is a modern Sat solver create by Guillaume Bury and derived from Alt-Ergo Zero from Microsoft. So HSat is the merging of MSat and the VPL library which make an important use of the Handelman mathematics theory giving the name to Hsat, Handelman SAT.

MSat wasn't fully compatible with theory addition and it needed many re-writing. The first important part was on the file parser, MSat using Dolmen a libriary developed by the same author which weren't able to parse correctly all mathematical operator. The biggest re-writing was on the typing and the theory handling. We redesigned the problem analyzing part of HSat to be theory oriented. We create a module interface for theory and added some algorythme in order to manage multi-theory capabilities. We made the choice to assign a theory to each symbol (as equation or boolean operator) during the analyze phase. The assignation is done via dedicated analyzer implemented in each theory. This behaviour is not optimal but it is an excellent compromise between efficiency, multi-theory capabilities and implementation time when no dynamic theory change are needed for the same expression.

4.2 Handelman decomposition

Definition 3 *Handelman Problem*
$(c_{11}..c_{NA}) \in \mathbb{Q}^{N*A}$
$(x_1..x_N) \in \mathbb{Q}^N$
$(p_1..p_A) \in \mathbb{N}^A$
$\forall j \in [1, A],\ Aff_j = \sum_{i=1}^{N} c_{ij} * x_i$
$\forall j \in [1, A],\ Aff_j > 0$
$\sum_{i=1}^{A}(Aff_i^{p_i}) > 0$

Definition 3 is an optimal form of problem for the Handelman decomposition. Studying efficiency of Handelman decomposition on it give us a mighty top bound of performance. We have generated some Handelman form problem test and their study showed up some limitation on the VPL heuristic. When confront with constraint generated by product of linear constraint of the problem, which should be the optimal form for the VPL, their heuristic are currently searching for an too complicated decomposition. That behaviour made the qualitative study to be delayed. Indeed the slowness induced by this behaviour made the result of studies too low in comparison of the expectation. That is the reason why we recommend to start over the study with the already done experimentation process as soon as VPL heuristic had their behaviour fixed.

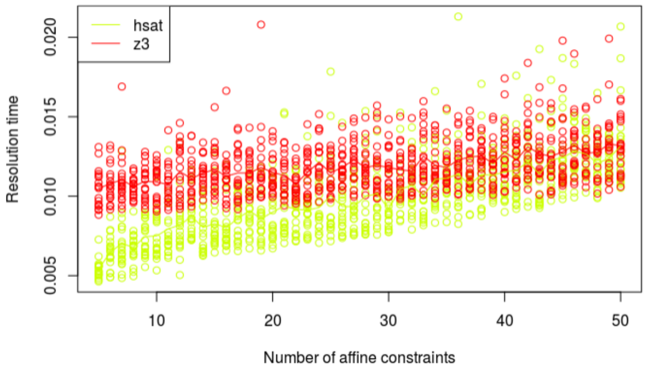
4.3 Xor issue

We tried to test the factorization with Xor form with HSat but it appear that MSat don't have a good implementaion of Xor operator handling. The incapacity of MSat handling imbricated Xor operation prevent us tu push farther the study on Xor form.

4.4 VPL experimentation

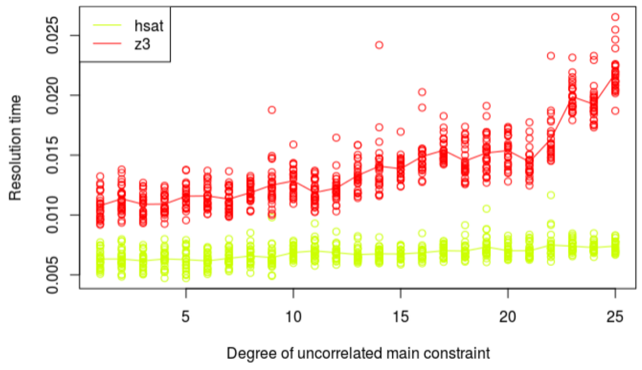
We studied the other main functionality of the VPL, the polyhedral calculus. To do so we designed some study focused on linear constraint analyze. Figure 5 is the time taken by z3 and Hsat (ie the VPL) to solve some linear unsat problem.

Figure 5: Comparison between mean raw form resolution time and Xor form one



Both solver had different behaviour, when HSat is quick to solve but its time taken grew up quickly too, z3 is slower but has an almost constant time consumption to solve the problem. We can conclude Hsat, thanks to the VPL polyhedral implementation, is more efficient to compute linear problem.

Figure 6: Comparison between mean raw form resolution time and Xor form one



The second interesting point we tested about the VPL is its ability to ignore irrelevant data. Indeed, du to its implementation the VPL has a more wide focus than the CAD theory which work separately on all constraint. To test this ability we generate unsat linear problem in which we include a difficult multilinear constraint with only variable not present in the linear constraints. The best strategy to treat such problem is to completely ignore the multilinear part and focus on the linear one, the multilinear constrain having absolutely no influence on the problem the optimal resolution time is the time taken to solve only the linear part. We can see in Figure 6 that HSat solve easily the problem ignoring the irrelevant part when z3 waste most of its time solving the multilinear constraint.

5 Tools

In order to test all the methods and solving tool we designed a tool chain of amortization and batch test managing. Figure [ref] is the global organization of the project as an layered architecture. Each layer had been designed for specific purpose and with an judged adapted technology. We decide the efficiency gain is higher benefit than the loss of maintainability due to heterogeneous technology.

5.1 Detailed layer

Solver

The most important part of this project is the solver, which is the program we are studying. It is in fact 2 different program, the first one is an external solver used as a reference, we choose z3. The second one is HSat which is the smt solver we developed during this project to test our diverses theory. HSat is developed in Ocaml because it allow the possibility of CoQ certification to prove the code and because it use mainly the VPL library which is written in Ocaml.

Problem Generator

As explained in part [] in order to test specifically some aspect of the behaviour of our theory, we designed some category of problem detailed in []. Then we developed an Ocaml program capable of generating a problem of these kinds according to a bunch of parameters. We choose Ocaml first because it's the technology use in the solver and it's the main technology of this research project which allow some code re-usability.

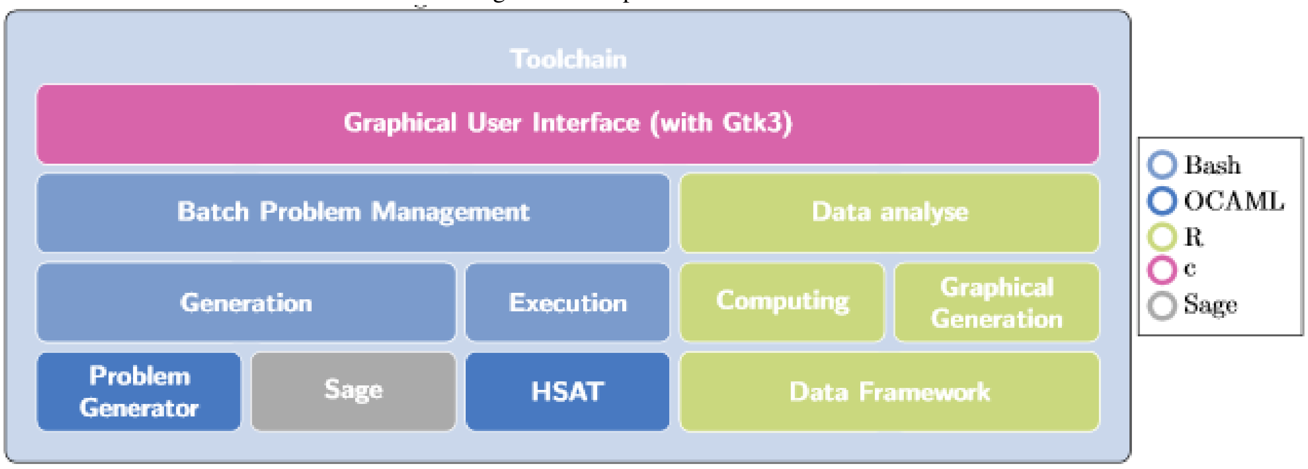
Batch Problem Manager

It would be extremely inefficient and painful to use directly the problem generator, indeed for a study user would use only one or two problems but a batch of problem. More he would make some variation between set of problem to see the difference. That's why we create this top layer script which allow to create easily some sets of problem with parameter varying gradually in range and more than one problem of each parameters to reduce the randomness effect.

DATA Framework

This work was intended to study several theory and many parameters for each one. That lead to extremely heterogeneous type of DATA. Indeed some are classical analyzing of the impact of parameters by an other, other was representing up to 5 degree of variation in the same picture, due to the frequency of "local behaviour" which could lead easily to wrong general conclusion. An example is, the solving time in function of the number of variable and the amount of linear constraint is considerably varying of behaviour depending of max degree, it can be approximately linear for degree of 2 then exponential for degree of 3 then faster for degree of 4. This behaviour can be explained by the theory strategy which are chosen by the solver sometime a characteristic change make the solver adopting an other strategy of solving which can be more efficient even if the problem is harder in theory.
In addition of the quantity of information that could be needed to be represented, many non "classical" graph and computation has to be done, for instance the cactus plot or even a graph which represent solving time for a solver in

Figure 7: Test platform architecture



function of solving time for another allowing to visually represent the difference for each problem/point specifically instead of watching for the global behaviour. For these reasons i decide to thing, first i would use R to work with my data, because it is convenient as R is intend and efficient for making statistics. The second is, writing algorithm in R for each plot and computation is really painful, even with copy and paste it's tedious and make the code thousand of line wide and opaque. So I decided to write a framework on top of R library to do computation and drawing special plot easily.
The framework is based on computation algorithm I already did for the preliminary test and is designed according 3 principles:

- Silent Data:
  All data has to be manipulated silently which mean the user do not manipulated variable or array of data but function already known of which data we are talking about allowed by the second point. That allow to reduce considerably the size of the code ignoring many code redundancy.
- Functional programming:
  I choose functional programming for 2 reason, the first one is because functional code is shorter (in character) than imperative one, the fact that long stack of function call is less readable should not be a weak point because it aim to be concise (one line wide) thank to it's effectiveness of writing.
  The second one is I would not the user writing code for making plot but only describing what he want as he do in natural language.
- Limited amount of function
  The framework must be 100% classic R compatible and has to be easily extended to make weird plot without really overstepping the "not code" principle, that point will be discuss after.

I achieve these goals by developing currification with optional parameter in R. That mean function of the framework can be called with less parameter than they are intended to have and it return a function which intended to have missing parameter, more some optional named parameters can be added which is particularly useful to limit the amount of function of the framework as parameter name for the function graf to add a title.
Even if data are silent they can be manipulated by function as "select" to filter row of data according value of a specific field, function "foreach" allowing to apply a function to each set of data with different value of a specific field, function "mean" which collapse a data set according mean function for each different value of a specific field, "col" which only return a column of data set or classical mathematical function as log, sum, cumsum...
We generate the Figure 8, the simplest instance of plot possible with the command:

$$graph(line(col("x"), col("y")))$$

Figure 8: A simple plot

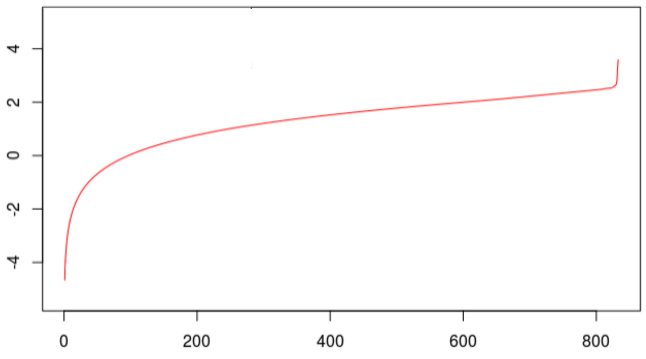
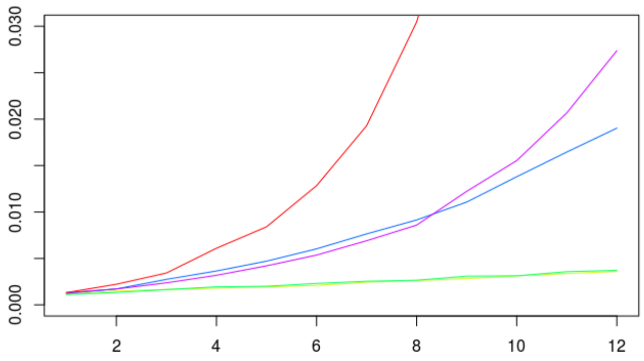


Figure 9 is a simple classical plot generated with the following command:

$$graph(auto\_color(foreach("solver",\\ line(col("x"), col("y")))))$$

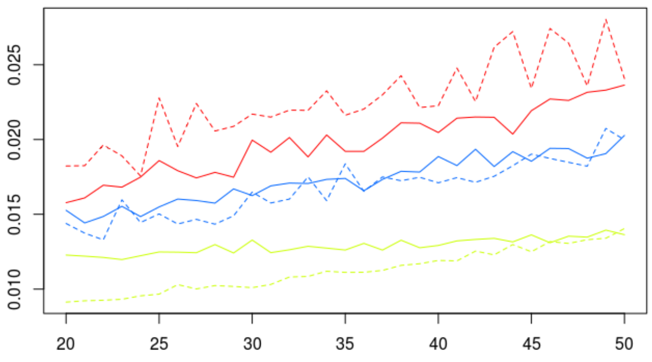
Figure 9: A classic plot



A more complex plot with 3 degree of variation: Use the function "then" which evaluate two function instead of one, "auto_color" which apply automatically color on data set in a determinate way and the function "ldoted" which draw a doted line.

$$graph(then(select("Type", "Raw", auto\_color(foreach(\\ "solver", line(col("x"), col("y"))))), select("Type", "\\ Factorized", auto\_color(foreach("solver", ldoted(col("x"),\\ col("y")))))))$$

Figure 10: Comparison between mean raw form resolution time and Xor form one



DATA Analyzer
To make a simpler use of the framework for external usage I wrote a simple R script that allow to execute a command as previously seen and produce a pdf of the generated graph.

Graphical User Interface
Finally, all these tools could be a bit complicated to use, especially the batch manager, because they need the user to remember the parameters which are a lot. That's the reason why I decide to produce a GUI showing visually every single feature described previously making they're use extremely simple.

Figure 11: Main screen of Batch Problem Manager

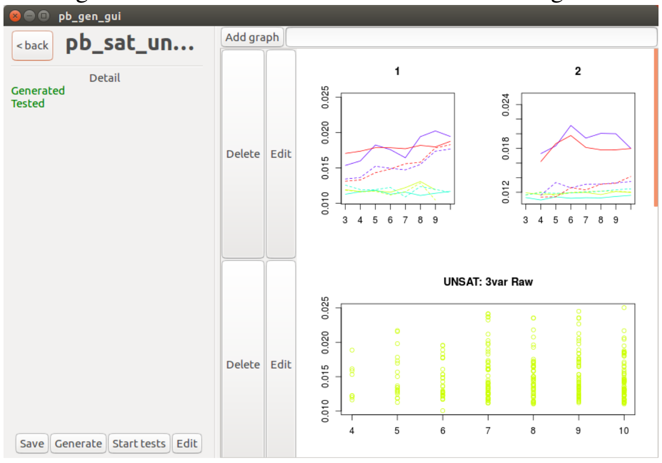
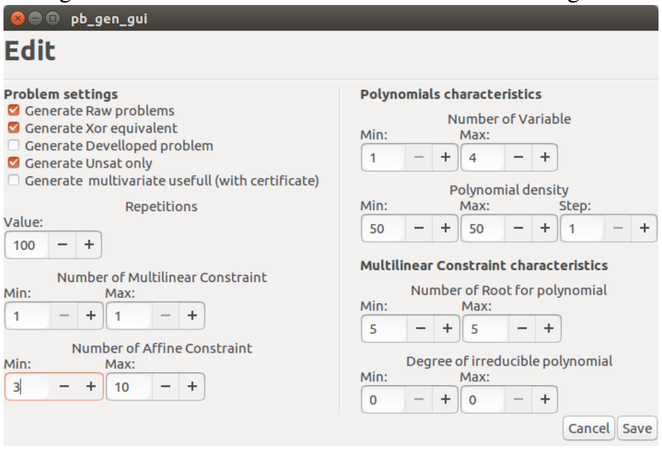


Figure 12: Detailed screen of Batch Problem Manager



pb_gen_gui

Edit

Problem settings
- [✓] Generate Raw problems
- [✓] Generate Xor equivalent
- [ ] Generate Develloped problem
- [✓] Generate Unsat only
- [ ] Generate multivariate usefull (with certificate)

Repetitions
Value: 100

Number of Multilinear Constraint
Min: 1 Max: 1

Number of Affine Constraint
Min: 3 Max: 10

Polynomials characteristics

Number of Variable
Min: 1 Max: 4

Polynomial density
Min: 50 Max: 50 Step: 1

Multilinear Constraint characteristics

Number of Root for polynomial
Min: 5 Max: 5

Degree of irreducible polynomial
Min: 0 Max: 0

For the framework, as it appear as a drawback managing parentheses in a long expression of the framework and the fact that the user would regularly use a copy-paste function (to create a doted line with only one parameter difference) which is sometimes a bit complicated due to the parentheses. I designed a small GUI which render the expression as a tree where node/sub-tree can be expanded or collapsed. That allow first to work only in with small expression negating the parentheses problem and allowing a simple copy/past of node or sub-tree. These two features make the framework really user friendly. It still remain a confident user could be faster writing directly the command manually, that is why I have let this possibility in the GUI. Finally in later work it could be interesting to add an integrated documentation and a code predication. An other main feature which still need to be added is the extension management, it is really simple like it would be only a way to add function before executing the command and making a simplified management and use in the GUI.

6 Conclusion & Future Work

6.1 Future Work

On the study
When VPL issues will be solved, il will be important to restart the experimentation to gather the results needed to conclude the study about Handelman theory methods. More, other related method had to be explored as ideal decomposition or euclidean division. When the VPL will permit it, it will be interesting to study the interacton between the VPL en the other methods.

On the Platform
To allow the test platform to lived outside its specific initial purpose it could be use-full to create an abstraction of the interfaces and layer. That would allow to use any and pilot automatically any software and gather its data to benefit of the features of the DATA Framework easily. Closer to its initial purpose, that would allow to easily adapt the platform to any other kind of testing, allowing to choose customized parameters and to provide your own test generator. Finnaly as the DATA Framework is the most important feature of the platform, improving it would be greatly beneficial for the platform. Or even enhancing the communication with the framework, like adding better functionalities to write framework command like auto-completion, integrated documentation or a better version of the current syntaxic tree viewer for more complicated command.

On the Framework
The Data Framework could be a really use-full tool, it could be shared with the community allowing people to use it but in order to do that a better and detailed documentation need to be written and exportation function must be improved to allow more flexibility. The top layer of the DATA Framework must be improved too to be usable more easily as an external tool from anybody.

6.2 Conclusion

In this article we have shown we could decompose polynomial constraint in different way to reduce problem complexity and resolution time. We point out non irreducibly is an information ignored by modern solver like z3. This information still use-full as it allow a performance gain when use an adapted form to force the solver knowing it. The gain reached in this article for the factorization method isn't optimal as we could improve it by reducing communication delay including directly algorithm in solver and using a more suitable and faster algorithm. We presented our work on the creation of HSat an SMT solveur based on the SAT solver MSat and the library of polyhedral computation the VPL and we explained how using polyhedron can be an improvement for SMT solvers. Finally we detailed the structure and functioning of the test platform we designed to conduct all theses studies.

A Proofs

Proof $\forall \boldsymbol{x},\ \psi(\boldsymbol{x}) \Rightarrow \varphi_1(\boldsymbol{x}) \wedge \varphi_2(\boldsymbol{x}) \equiv (\forall \boldsymbol{x},\ \psi(\boldsymbol{x}) \Rightarrow \varphi_1(\boldsymbol{x})) \wedge (\forall \boldsymbol{x},\ \psi(\boldsymbol{x}) \Rightarrow \varphi_2(\boldsymbol{x}))$
□

Proposition 1 (5 p.2) $\text{UNSAT}(\mathscr{A} \wedge \mathscr{Q}) \equiv \text{UNSAT}(\mathscr{A}) \vee (\forall \boldsymbol{x},\ \mathscr{A}(\boldsymbol{x}) \Rightarrow \neg\mathscr{Q}(\boldsymbol{x}))$

Proof
$$\begin{aligned} \mathsf{UNSAT}(\mathscr{A} \wedge \mathscr{Q}) &\equiv \neg\left(\mathsf{SAT}(\mathscr{A} \wedge \mathscr{Q})\right) \\ &\equiv \neg\left(\mathsf{SAT}(\mathscr{A}) \wedge \mathsf{SAT}(\mathscr{A} \wedge \mathscr{Q})\right) \\ &\equiv \neg\mathsf{SAT}(\mathscr{A}) \vee \neg\mathsf{SAT}(\mathscr{A} \wedge \mathscr{Q}) \\ &\equiv \mathsf{UNSAT}(\mathscr{A}) \vee \neg\left(\exists \boldsymbol{x},\ \mathscr{A}(\boldsymbol{x}) \wedge \mathscr{Q}(\boldsymbol{x})\right) \\ &\equiv \mathsf{UNSAT}(\mathscr{A}) \vee (\forall \boldsymbol{x},\ \neg\mathscr{A}(\boldsymbol{x}) \vee \neg\mathscr{Q}(\boldsymbol{x})) \\ &\equiv \mathsf{UNSAT}(\mathscr{A}) \vee (\forall \boldsymbol{x},\ \neg\mathscr{A}(\boldsymbol{x}) \vee \neg\mathscr{Q}(\boldsymbol{x})) \\ &\equiv \mathsf{UNSAT}(\mathscr{A}) \vee (\forall \boldsymbol{x},\ \mathscr{A}(\boldsymbol{x}) \Rightarrow \neg\mathscr{Q}(\boldsymbol{x})) \end{aligned}$$
□

Proposition 2 (6 p.2) *Given* $\mathscr{Q}(\boldsymbol{x}) = \bigwedge_{i=1}^{q} \mathcal{Q}_i(\boldsymbol{x})$

$$\forall \boldsymbol{x},\ \mathscr{A}(\boldsymbol{x}) \Rightarrow \neg\mathscr{Q}(\boldsymbol{x}) \equiv \forall \boldsymbol{x},\ \mathscr{A}(\boldsymbol{x})$$

$$\Rightarrow \bigvee_{i=1}^{q} \left(\bigwedge_{j=1, j\neq i}^{q} \mathcal{Q}_j(\boldsymbol{x}) \Rightarrow \neg\left(\mathcal{Q}_i(\boldsymbol{x})\right) \right)$$

Proof The proof relies on the general version of the following equivalences: $\neg\left(\varphi_1 \wedge \varphi_2 \wedge \varphi_3\right) \equiv \neg(\varphi_1 \wedge \varphi_2) \vee \neg\varphi_3 \equiv (\varphi_1 \wedge \varphi_2) \Rightarrow \neg\varphi_3$

$$\begin{aligned} \neg\mathscr{Q}(\boldsymbol{x}) &\equiv \neg\left(\textstyle\bigwedge_{j=1}^{q} \mathcal{Q}_j(\boldsymbol{x})\right) \\ &\equiv \textstyle\bigvee_{j=1}^{q} \neg\left(\mathcal{Q}_j(\boldsymbol{x})\right) \\ &\equiv \textstyle\bigvee_{i=1}^{q} \left(\bigvee_{j=1}^{q} \neg\left(\mathcal{Q}_j(\boldsymbol{x})\right)\right) \text{ since } A \equiv A \vee \ldots \vee A \\ &\equiv \textstyle\bigvee_{i=1}^{q} \left(\bigwedge_{j=1, j\neq i}^{q} \mathcal{Q}_j(\boldsymbol{x}) \Rightarrow \neg\left(\mathcal{Q}_i(\boldsymbol{x})\right)\right) \end{aligned}$$

□

References

[dOBP16] Steven de Oliveira, Saddek Bensalem, and Virgile Prevosto. Polynomial invariants by linear algebra. In *Automated Technology for Verification and Analysis - 14th International Symposium, ATVA 2016, Chiba, Japan, October 17-20, 2016, Proceedings*, pages 479–494, 2016.

[GK14] Matthew Gwynne and Oliver Kullmann. On sat representations of xor constraints. In *Language and Automata Theory and Applications (LATA)*, pages 409–420. Springer International Publishing, 2014.

[HJ12] Cheng-Shen Han and Jie-Hong Roland Jiang. When boolean satisfiability meets gaussian elimination in a simplex way. In *Proceedings of the 24th International Conference on Computer Aided Verification (CAV)*, CAV'12, pages 410–426. Springer-Verlag, 2012.

[LJN13] Tero Laitinen, Tommi Junttila, and Ilkka Niemelä. Simulating parity reasoning. In Ken McMillan, Aart Middeldorp, and Andrei Voronkov, editors, *Logic for Programming, Artificial intelligence, and Reasoning (LPAR)*, pages 568–583. Springer Berlin Heidelberg, 2013.

Automatic Grading based on Bisimulation

Alexandre Borthomieu
Grenoble, France
alexandre.borthomieu@etu.univ-grenoble-alpes.fr

Supervised by: Michael Perin

I understand what plagiarism entails and I declare that this report is my own, original work.
Name, date and signature:

Abstract

MOOC[1] and exams require lot of time to teachers and give no useful feedback to students. We believe that instant feedback on student's answer is a better way to practice and learn. In this paper, we present the first steps of an algorithm which will automatically grade students with a feedback as close as possible to their answer. This algorithm is based on Bisimulation [Sangiorgi, 2009] and can be used on trees, automatons and graphs.

1 Introduction

The first goal of this work is to grade students and to provide useful feedback on automaton. In order to achieve that, we use the structure of a colored labeled oriented graph (called CLOG). It is a simple generalization which allow a compatibility with graphs.

1.1 Grading

Before getting a grade, the teacher has to provide the solution, hidden from students, who have to provide their solution. Both solutions will be written in a certain syntax in order to be parsed and stored in a CLOG structure. The next step is to let the algorithm work.

1.2 Similar works

We can find similar ideas at Microsoft Research Center : [Wang *et al.*, 2018; Gulwani *et al.*, 2018] developed algorithms which aim to repair code's parts from students on MOOC.

2 Algorithm

The algorithm developed here need to compare both solutions from a student (1.a) and a teacher (1.d). To be able to compare those, both solutions need to be complete. Therefore, the first step of this algorithm is to add a new node which will gather all the edges. It is called a *black hole* (1.b).

[1]Massive Online Open Courses

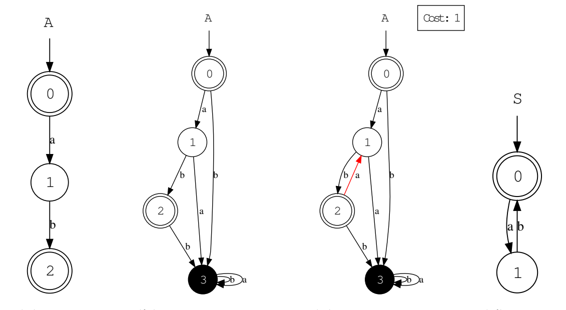


Figure 1: *(a)* Student's solution *(b)* complete Student's solution *(c)* Final solution and feedback example *(d)* Teacher's solution

2.1 Equivalence Classes and Reparation

Based on Bisimulation [Sangiorgi, 2009], the algorithm will create equivalence classes in which each node from both CLOG will be stored. If for each node of a teacher's solution, there is at least one node from the student's solution in an equivalence class and not any node of the student's solution is alone in a class, then these two CLOG can be called *equivalent*. Otherwise, a reparation will be required.

To be repaired and placed in a equivalence classes Eq, a node n needs to have the *same future* than the other nodes in Eq. In other word, n should be equivalent to every node of Eq. But, the algorithm do not have the information on which equivalence classes n should belong to. Therefore, it will explore all possible repairs (with pruning). It will try to put each wrong node in each equivalence classes. Each case will produce a new automaton.

Finally, we retain the automaton based on the first student's solution and with the less expensive and minimal repair(s) (1.c).

3 Future work

This algorithm can be upgraded by improving the exploration, in order of increasing cost for instance. But another algorithm may be developed without any enumeration of repair. The global cost of this algorithm should be better. And last, this work can be generalized to any larger domain if it contain a decidable equivalence's relation.

4 Conclusion

The aim of this work was to create a first algorithm which can provide the right answer with a minimal number of repair. We successfully achieved that goal. However, it is absolutely not an optimal solution, but we can now examine this algorithm and refine it to improve the global cost and the execution time.

References

[Gulwani *et al.*, 2018] Sumit Gulwani, Ivan Radiček, and Florian Zuleger. Automated clustering and program repair for introductory programming assignments. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, pages 465–480, New York, NY, USA, 2018. ACM.

[Sangiorgi, 2009] Davide Sangiorgi. On the origins of bisimulation and coinduction. *ACM Trans. Program. Lang. Syst.*, 31(4):15:1–15:41, May 2009.

[Wang *et al.*, 2018] Ke Wang, Rishabh Singh, and Zhendong Su. Search, align, and repair: Data-driven feedback generation for introductory programming exercises. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, pages 481–495, New York, NY, USA, 2018. ACM.

Automatic Grading based on Bisimulation

applications to automata, trees, graphs, all seen as colored labeled oriented graph

Exercise: Give an automaton A that recognises $\{(ab)^n \mid n \in \mathbb{N}\}$

Student's answer

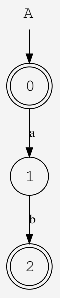
Teacher's hidden solution

Answer A

A := automaton {
-> ((0)) - a -> (1)
(1) - b -> ((2))
}

parsing ;

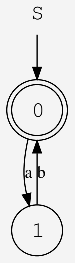
output



The question is

$\simeq$

?



parsing ;

output

for check

Solution S

```
S:= automaton{
  ->((0))-a->(1)-b->((0))
}
```

Algorithm

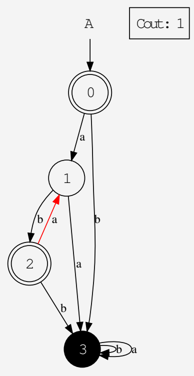
- First, complete both automaton with a "black hole" node
- Create equivalence classes for both (based on bisimulation[1])
- Find nodes of A that match with none of equivalence classes
- Create a new Automaton A^r for each repair possible (repairs listing) such that
$$A^r \simeq S$$
- Find less expansive and minimal repair(s)
- Produce a grade with the cost of the repair

Feedback

The purpose of this work is to give an appropriate feedback to students.

- That feedback correspond to the minimal repair to his answer completed with an additional node "black hole" (node 3) and can be display like this one

$\Rightarrow$

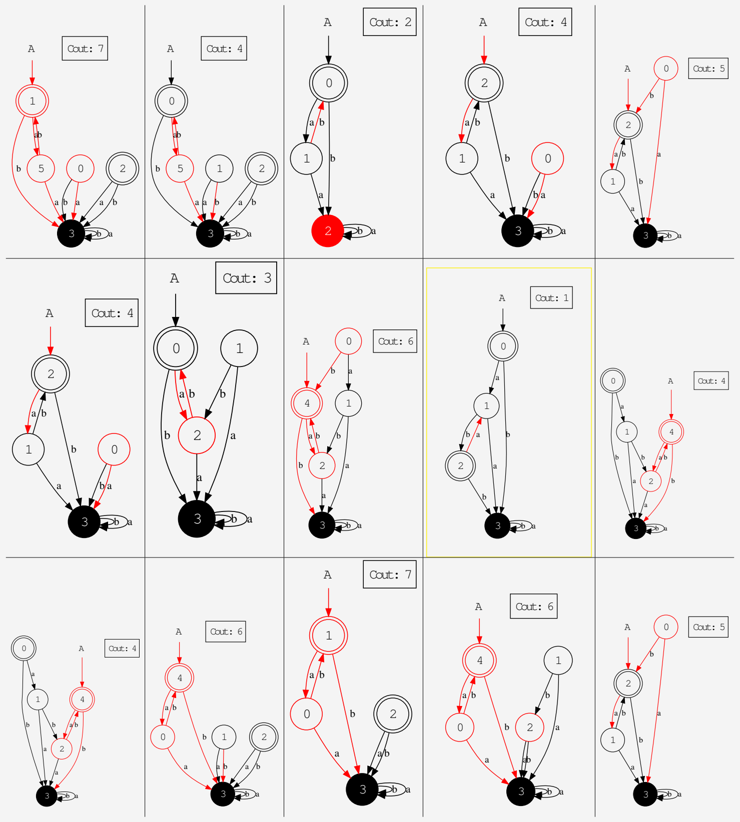


Final Solution

Limitations

- Exploration of all possible repairs with pruning

Solutions produced



Future Work

- Upgrade this algorithm by exploring repairs in order of increasing cost
- New algorithm not based on enumeration of repairs
- Generalization of this work to other domains with decidable equivalence

Related Work

- D. Sangiorgi, On the Origins of Bisimulation and Coinduction. TOPLAS'2009.
- Similar ideas are applied at Microsoft for assessing programming exercises on Massive Online Open Course to obtain developer certificate
  - K. Wang, R. Singh and Z. Su. Search, Align, and Repair: Data-Driven Feedback Generation for Introductory Programming Exercises. PLDI'2018
  - S. Gulwani, I. Radiček and F. Zuleger. Automated Clustering and Program Repair for Introductory Programming Assignements. PLDI'2018

UNIVERSITÉ Grenoble Alpes

Michael Perin, Alexandre Borthomieu

{firstname.name}@univ-grenoble-alpes.fr

Verimag

Large Scale Traces Analysis : Multi-Scale Patterns
Extended Abstract

Nils Defauw
POLARIS (LIG) and IM^{2}AG (UGA)
Grenoble, France
nils.defauw@etu.univ-grenoble-alpes.fr
Supervised by Jean-Marc Vincent

1 Introduction

In the context of large scale traces analysis, the quantity of information to compute is often too big for analysis without necessarily having this whole information relevant. A typical case is the analysis of irregularities in execution traces of high performance applications, counting millions of cores spread in several places in the world. In these cases, any irregularity arising in a single core, a machine or a whole cluster can be hard to notice. As a low-cost strategy, aggregation technique have been developped [Lamarche-Perrin, 2013] which consists of choosing the most qualitative partition in a set of acceptable partitions. But in a low-knowledge system, we can't always define the set of acceptable partitions and we thus aim to provide a way to aggregate without this knowledge. We explored Lempel-Ziv techniques for finding places where regularity is broken, such places could define partitions in the system for aggregation.

2 Aggregation based on acceptable partitions

The aggregation technique [Lamarche-Perrin, 2013] previously developed consists of reducing the complexity of a system without losing too much information. Actually, for each partition in the set of acceptable partitions, it aggregates the system with this partition replacing each part of the partition with a macroscopic object simpler than the part itself which represents this part. Then each aggregated system is interpreted, which means that each macroscopic object of the aggregation is redistributed on the microscopic objects that compose it and this interpretation is compared with the original system, the divergence between the two is called the information loss between the original system and the aggregated one. The information loss is one of the two criteria defining the quality measure, the other being the reduction of complexity of the aggregation, which is the number of macroscopic objects (or parts) in the aggregation, the lower parts an aggregation has, the simpler it is. Each acceptable partition is compared using this measure and the best partition is used for the aggregation. This leads to aggregations with a balance between the reduction of complexity and the loss of information, because aggregating homogeneous parts in a single macroscopic object leads to less information loss than aggregating in a single object an heterogeneous part of the same size. But the problem with this approach is that it requires knowledge on the system we're analyzing because defining the set of acceptable partitions needs for the user to know the topology of the system.

3 Lempel-Ziv techniques for partitioning

We studied Lempel-Ziv techniques and particularly the Lempel-Ziv complexity [Lempel and Ziv, 1976] as its usefulness in finding discontinuities in the regularity of a sequence for providing a way for the aggregation algorithm to partition itself the system in homogeneous components without the need of a set of acceptable partitions. We found that when inserting a small random pattern at random positions in a large regular sequence we could clearly identify discontinuities at the positions where the patterns where introduced in the graph of the eigenvalue [Lempel and Ziv, 1976] of the prefixes of the sequence. We also remarked a few other discontinuities that we can't explain yet in the sequence and that need to be studied in future work. We also started to study the eigenvalue of a sequence itself and found an elegant way to represent it with the construction of a tree of suffixes which would potentially lead to better understanding of this indicator.

4 Conclusion

We studied Lempel-Ziv eigenvalue applications in aggregation process with a few interesting and promising results but we left a lot of questions unanswered which leads us to think that a continuation of this work is necessary. In particular, how to explain the unexpected discontinuities in the graph of eigenvalue, what is the behaviour of an homogeneous sequence not perfectly regular unlike in our experiment, and can we use statistical tools to study the eigenvalue of a sequence compared to the filling of the tree of suffixes that represents it.

References

[Lamarche-Perrin, 2013] Robin Lamarche-Perrin. *Analyse macroscopique des grands systèmes*. PhD thesis, Laboratoire d'Informatique de Grenoble, oct 2013.

[Lempel and Ziv, 1976] Abraham Lempel and Jacob Ziv. On the complexity of finite sequences. *IEEE Transactions on Information Theory*, 22(1):75–81, jan 1976.

Large Scale Traces Analysis : Multi-Scale Patterns

Nils Defauw supervised by Jean-Marc Vincent

Team POLARIS - Laboratoire d'Informatique de Grenoble - Université Grenoble Alpes

Objectives

Using Lempel-Ziv techniques [1] for partitioning and aggregating large scale systems

- No need for an extra-knowledge of the system (no set of acceptable partitions)
- Possible discovery of the inner nature of the system (its topology)

First step :

- Understanding the link between Lempel-Ziv theory [1] and algorithms [2, 3] and the regularity of a sequence
- Testing Lempel-Ziv indicators on multiple specially forged sequences

Context

Execution traces of high performance applications

- Huge amount of data, difficult to summarize without losing too much information
- Multi-scale patterns in the data require multiple levels of aggregation

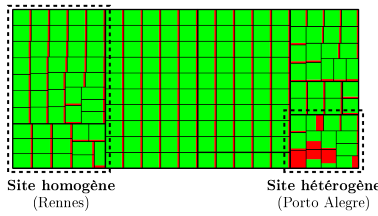


Figure 1:Execution trace on the GRID'5000 platform

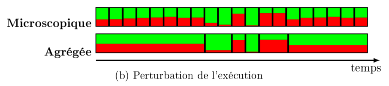


Figure 2:One dimension execution trace, function of the time

Based on aggregation technique

Extension of the efficient and low cost aggregation algorithm [4]

Pros

- Assurance to have the most qualitative partition in the set of acceptable partitions
- Assurance to have an aggregation reliable with the topology of the system

Cons

- Require a previous knowledge of the system
- Can miss a yet unknown link between microscopic objects

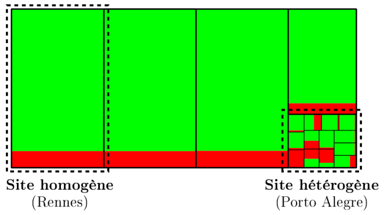


Figure 3:Aggregated execution trace on the GRID'5000 platform

Use Lempel-Ziv techniques

Lempel-Ziv techniques [1, 2, 3] as a notion of regularity in the sequence

- Usage of the Lempel-Ziv complexity [1] and more particularly the *eigenvalue* used by it
- Sub-additivity property useful for a measure of complexity (a low-complexity sequence is said regular, same concept)

$$C(S \cdot Q) \leq C(S) + C(Q)$$

- Can be summarized as the growth of vocabulary (list of sub-chains inside the sequence), easily represented by a tree

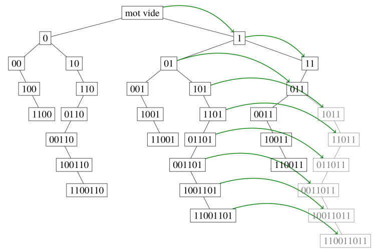


Figure 4:Vocabulary of 110011011 with the last nodes added in gray

Important Result

After generating a regular sequence 0101 0101 0101 . . . of length 1024 and adding a 32 bit long specific pattern at positions 86, 258 and 622, we remark discontinuities in the graph of the eigenvalue of the sequence at positions ≈ 90, ≈ 260, ≈ 430, ≈ 620 and ≈ 980.

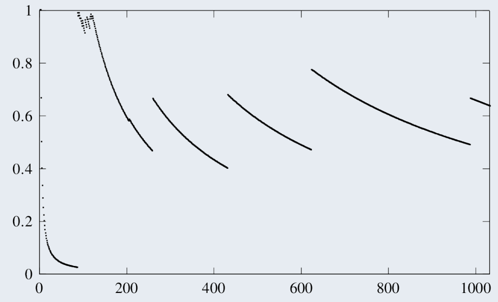


Figure 5:Eigenvalue of S, $y = k(S(1, x))$

Conclusion

This research has brought some interesting results like the discontinuities in the graph of the eigenvalue of a sequence and the vocabulary representation in a tree form. This is a promising lead for future work for which we hope some concrete applications will appear. A lot still needs to be understood like the two other discontinuities in the graph of the figure 5 and a lot of new experiments needs to be done in order to fully understand the measures involved and be able to use this knowledge to aggregate a large scale system without more information on the structure of the system.

References

[1] Abraham Lempel and Jacob Ziv.
On the complexity of finite sequences.
IEEE Transactions on Information Theory, 22(1):75–81, jan 1976.

[2] Jacob Ziv and Abraham Lempel.
A universal algorithm for sequential data compression.
IEEE Transactions on Information Theory, 23(3):337–343, may 1977.

[3] Jacob Ziv and Abraham Lempel.
Compression of individual sequences via variable-rate coding.
IEEE Transactions on Information Theory, 24(5):530–536, sep 1978.

[4] Robin Lamarche-Perrin.
Analyse macroscopique des grands systèmes.
PhD thesis, Laboratoire d'Informatique de Grenoble, oct 2013.

Knowledge base mining using compressed data structures

Adelina PROKHOROVA
adelina.prokhorova@etu.univ-grenoble-alpes.fr
SLIDE - LIG, Université Grenoble Alpes

1 Introduction

Knowledge bases (KBs) provide information about a great variety of entities such as people, places, organisations, countries, cities, films... Today's KBs contain millions of entities and hundreds of millions of facts, however, these KBs are limited to human knowledge and capacity, so they are still far from complete. Due to new approaches it is possible to find patterns(regularities) to infer new knowledge, to evaluate the sureness of each rule to construct a complete KB, to identify potential errors, to reason the rules, and to understand the data better.

2 Related work

Existing approaches of knowledge rule mining are : AMIE+ [1], Ontological Pathfinding [3]. The first version of AMIE [2] shows that Horn rule mining corresponds to association rule mining on a database, which are defined as a list of transactions (set of items). The problem of association rule mining is that the standard measurements for support and confidence do not produce good results for some applications.
Ontological Pathfinding and AMIE+ used association rule mining. A rule consists of a head : r(x,y) and a body : $\{B_1,...,B_n\}$, where the head is a single atom and the body is a set of atoms:

$$B_1 \wedge B_2 \wedge ... \wedge B_n \Rightarrow r(x,y) \Longleftrightarrow \overrightarrow{B} \Rightarrow r(x,y)$$

The used definition of support is explained as the number of distinct pairs of subjects and objects in the head of all instantiations that appear in the KB.

$$supp\ (\overrightarrow{B} \Rightarrow r(x,y)) := \#(x,y) : \exists z_1,...,z_m : \overrightarrow{B} \wedge r(x,y)$$

There are two definitions for confidence, the first one, called standard confidence, counts all unrepresented facts as negative evidence, and the second one calls PCA confidence and operates only with facts that we know to be true together with the facts that we assume to be false.

$$conf\ (\overrightarrow{B} \Rightarrow r(x,y)) := \frac{supp\ (\overrightarrow{B} \Rightarrow r(x,y))}{\#(x,y) : \exists z_1,...,z_m : \overrightarrow{B}}$$

$$conf_{pca}\ (\overrightarrow{B} \Rightarrow r(x,y)) := \frac{supp\ (\overrightarrow{B} \Rightarrow r(x,y))}{\#(x,y) : \exists z_1,...,z_m, y' : \overrightarrow{B} \wedge r(x,y')}$$

All the definitions were taken from article describing AMIE+. The problem of those two approaches is that they took a lot of time to execute the program on the big datasets, due to evaluating each rule from the queue of rules, initially the size of all rules is 1, later, deciding that the rule is sure, it would be added in the queue, and the execution would be continued, but those algorithms are limited by rule size equals to 3 and it they are using also head coverage, some facts could be loosed. Another one approach that could resolve this problem is SAMi, developed at LIG, using Frequent Subgraph Mining (FSM). SAMi, uses a compressed representation of the embeddings of a pattern based on automata. SAMi generates patterns recursively, by applying primitive operations on parent patterns of smaller size. This approach is more efficient, avoids duplicates and his complexity depends on the size of the automata instead of the number of embeddings.

3 Contribution

The aim of my internship was to find a possibility to implement the indicated definitions of support and confidence to evaluate SAMi results. My method to calculate support using automata is looking for distinct pairs of variables at indicated levels of automata. For exemple, we have a rule:
A isMarriedTo B ∧ B hasChild C ⇒ A hasChild C, so the goal is to find all pairs of (A, C) which satisfy this rule. To realise this aim, the algorithm must return a set of all values from level 1 in automata and all corresponding values from level 3. To compute standard confidence, "result" was computed in the same way, just automata was corresponding to body of rule :
A isMarriedTo B ∧ B hasChild C, but the idea is the same - find all pairs of (A, C), and the value of standard confidence is the number of distinct pairs in support divided by the number of distinct pairs in "result". The calculation of PCA confidence is a bit more complicated, but it is proceeding the same idea: computed pairs of (A, C) for a body must be controlled by A hasChild C', so A should have at least one child to be added to the "final" set of values to calculate PCA confidence. To get a value of this confidence, the number of distinct pairs in support must be divided by the number of distinct pairs in "final" set.

4 Conclusion

During this internship a goal to implement an algorithm which allows evaluate rule's confidence is achieved.
As the internship duration was limited, my algorithm works perfectly with these types of rules : ab ∧ bc ⇒ ac, ab ∧ ab ⇒ ab, ab ⇒ ab. The results of executing these types are the same as AMIE+ results on YAGO2 KB. Other types of rules could be implemented in the future.

References

[1] L. Galárraga, C. Teflioudi, K. Hose, and F. M.Suchanek. *Fast rule mining in ontological knowledge bases with amie+*. The VLDB Journal, 24(6):707–730, Dec. 2015.

[2] L. Galárraga, C. Teflioudi, K. Hose, and F. M.Suchanek. *AMIE: association rule mining under incomplete evidence in ontological knowledge bases*. WWW, 2013.

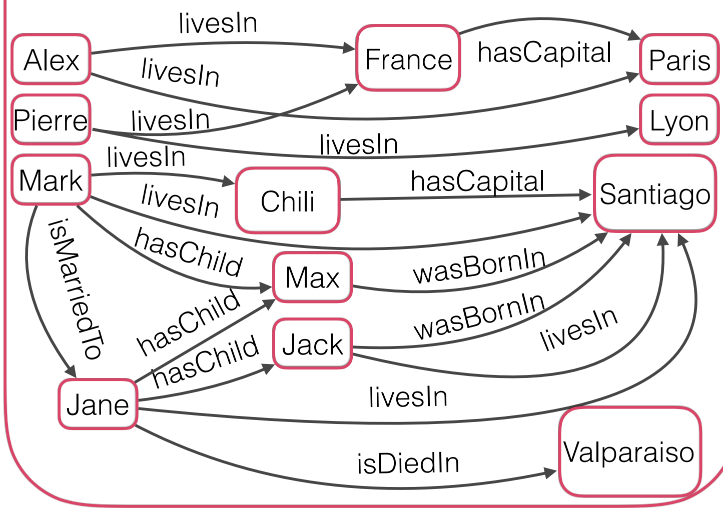
[3] Y. Chen, S. Goldberg, D. Z. Wang, and S. S. Johri. *Ontological pathfinding*. In Proceedings of the International Conference on Management of Data (SIGMOD), pages 835–846. ACM, 2016.

Knowledge base mining using compressed data structures

Adelina PROKHOROVA,
adelina.prokhorova@etu.univ-grenoble-alpes.fr,
SLIDE - LIG, Université Grenoble Alpes

Knowledge bases (KBs) provide information about a great variety of entities such as people, places, organisations, countries… It is possible to find patterns to infer new knowledge, to evaluate the sureness of each rule to construct a complete KB.

KB	Facts	Subjects	Relations
YAGO2 core	948K	470K	32
DBpedia 3.8	11.02M	2.20M	650
Wikidata	8.4M	4.00M	431



A rule consists of a head: r(x, y) and a body: $\{B_1, ..., B_n\}$, where the head is a single atom and the body is a set of atoms:

$$B_1 \wedge B_2 \wedge ... \wedge B_n \Rightarrow r(x,y) \iff \overrightarrow{B} \Rightarrow r(x,y)$$

Support is the number of distinct pairs of subjects and objects in the head of all instantiations that appear in the KB.

$$supp\,(\overrightarrow{B} \Rightarrow r(x,y)) := \#(x,y) : \exists z_1, ..., z_m : \overrightarrow{B} \wedge r(x,y)$$

Standard confidence counts all unrepresented facts as negative evidence.

PCA confidence operates only with facts that we know to be true together with the facts that we assume to be false.

$$conf\,(\overrightarrow{B} \Rightarrow r(x,y)) := \frac{supp\,(\overrightarrow{B} \Rightarrow r(x,y))}{\#(x,y) : \exists z_1, ..., z_m : \overrightarrow{B}}$$

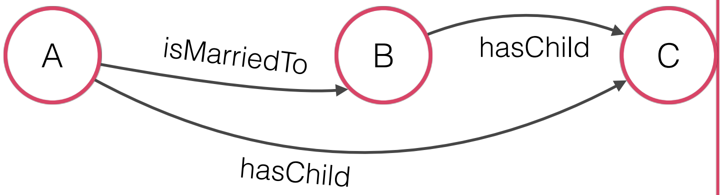
$$conf_{pca}\,(\overrightarrow{B} \Rightarrow r(x,y)) := \frac{supp\,(\overrightarrow{B} \Rightarrow r(x,y))}{\#(x,y) : \exists z_1, ..., z_m, y' : \overrightarrow{B} \wedge r(x,y')}$$

During this internship a goal to implement an algorithm which allows evaluate rule's confidence is achieved.

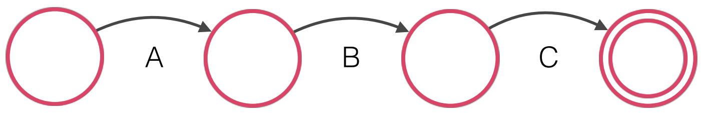
The aim of my internship was to find a possibility to implement the indicated definitions of support and confidence to evaluate SAMi* results.

For exemple, we have a rule:

$$A\, isMarriedTo\, B \wedge B\, hasChild\, C \Rightarrow A\, hasChild\, C$$



Automata returned by SAMi* for this example:

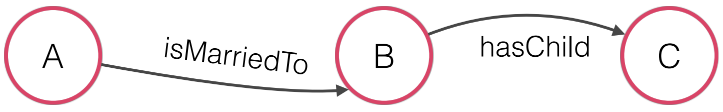


The support is a number of distinct pairs of (A, C), which satisfy this rule.

To compute standard confidence automata corresponds to body of rule :

$$A\, isMarriedTo\, B \wedge B\, hasChild\, C$$

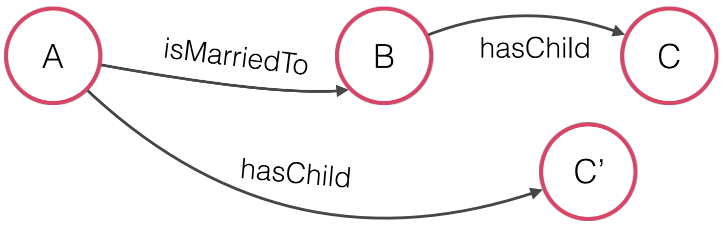
but the idea is the same - find all pairs of (A, C), and the result is support divided by the number of distinct pairs in body.



To find PCA confidence, computed pairs of (A, C) for a body must be controlled by a rule

$$A\, hasChild\, C',$$

so A should have at least one child to be accepted. To get a value of this confidence, support must be divided by the number of distinct pairs satisfying body and announced rule.



*SAMi approach, developed at LIG, which uses a compressed representation of the embeddings of a pattern based on automata.

Natural language generation : comparison of language models

Fabien Lefebvre
SIGMA (LIG)
Saint-Martin-d'Hères, France
Fabien.Lefebvre@etu.univ-grenoble-alpes.fr

Supervised by Cyril Labbé

1 Introduction

1.1 Natural language generation

Natural language generation is the art of generating understandable sentences. Text generation can be used in predictive keyboards. The model can also be used for other purposes than text generation. For example, you can train the model on differents author's book, and then guess who wrote a random book by comparing the perplexity by sentence of each model.

1.2 Existing language models

The most known language models are the N-grams, which consists of predicting the next word based on the N-1 previous words, by counting the occurences of this N-gram in the training corpus. The result is a fast trained but unoriginal text, due to N-grams only learning sequences of words.

2 Language model using RNN

Recurrent Neural Networks are Neural Networks with the capability process sequences of data with a state memory. This property makes RNN interesting in text generation for the reason that we need to memorize information from previous words to predict the next word.

2.1 The model

The model we will use to generate text is a modified version of the Tensorflow official LSTM tutorial.
The model take in input 20 words, and will try to predict the 2^{nd} to 21^{st} words of the sentence, using the precedent word and a state memory. The 21^{st} word is the one we are trying to predict.

2.2 Word embedding

Word embedding is a method to map words to real numbers vectors, improving natural language processing tasks. Similar words tends to have a similar vector, which gives more originality to word prediction. N-gram model may learn "I love pizza" but not "I love eating pasta", but words embedding will see the similarity between "pizza" and "pasta" and will give them almost the same probability, without learning the phrase "I love pasta".

2.3 LSTM

The LSTM is the part of the model which will memorize the state of the sentence.

2.4 Logits

Logits are value representing the likeliness of being the word predicted. They can be transformed into probability using a linear function or a softmax function. They are computed by an addition of the word bias matrix and the product of the predicted word embedding matrix and the reverse embedding matrix.

2.5 Backpropagation

When learning, the model need to modify its tensors to give better resutls. The model will change the value of tensors starting from the result (logits), to the logit computation (reverse embedding and word bias), to the word prediction, to the word embedding of the input.

3 Results

The evaluation of the model is done by computing the perplexity and accuracy on a corpus, and comparing it to the perplexity and accuracy of N-grams on the same corpus.

3.1 Perplexity

The RNN manage to reach a perplexity of 105, while bigram's only reach 364. I don't include trigram's result because it seems irrealistic, probably because of a bug.

3.2 Accuracy

The RNN predict right 22% of the words in the corpus, which is much higher than bigram's 10.7% and trigram's 7.8%

4 Conclusion

RNN seems to perform better than N-grams on perplexity and accuracy, but it must be noted that the corpus is relatively small, which favorise RNN over N-grams.

References

[Dan Jurafsky *et al*., 2017] Dan Jurafsky, James H. Martin. *Speech and Language Processing (3rd edition)*. 2017.

Tensorflow documentation, Google. 2018.

[Zaremba *et al*., 2015] Wojciech Zaremba, Illya Sutskever, Oriol Vinyals

Recurrent Neural Network Regularization. 2015.

[Mikolov *et al*.,2014] Yoav Goldberg, Omer Levy

Word2vec explained: Deriving Mikolov et al's Negative-Sampling word embedding method. 2017.

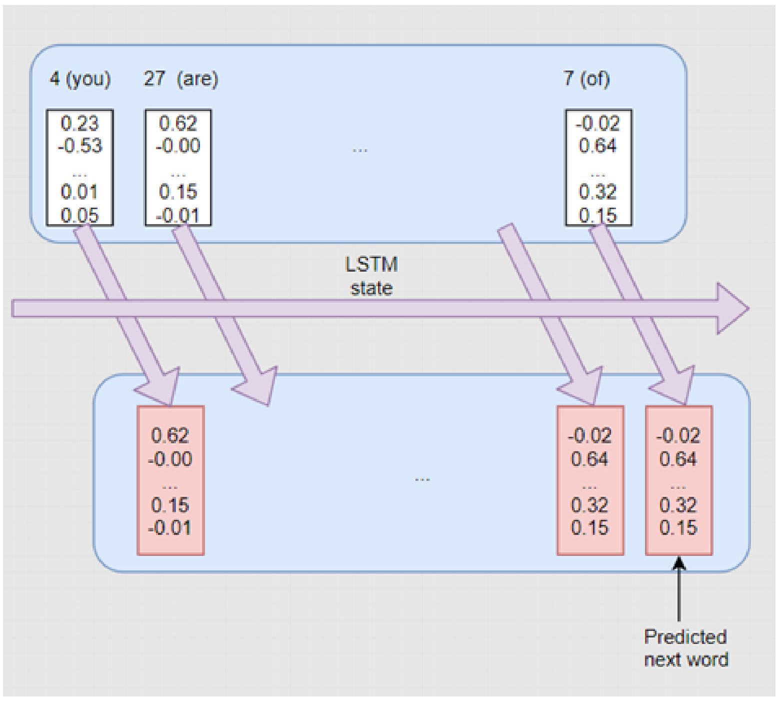
LIG - SIGMA

NATURAL LANGUAGE GENERATION : COMPARISON OF LANGUAGE MODELS

BY FABIEN LEFEBVRE

INTRODUCTION

The aim of this internship is to evaluate the use of Recurrent Neural Networks (RNN) as a language model to generate natural language.



N-GRAMS

N-grams use the N-1 previous word to predict the next word. With a trigram model, we try to predict a word with the 2 (3-1) previous words.
In the sentence "the cake is a lie", we try to predict "is" knowing "the cake".

RNN

Recurrent Neural Networks are Neural Networks with the capability process sequences of data with a state memory. This property makes RNN interesting in text generation for the reason that we need to memorize in-formation from previous words to predict the next word.
This is the model I worked on.

WORDS EMBEDDING

Word embedding is a method to map words to real num-bers vectors, improving natural language processing tasks. Similar words tends to have a similar vector, which gives more originality to word prediction. N-gram model may learn "I love pizza" but not "I love eating pasta", but words embedding will see the similarity between "pizza" and "pasta" and will give them almost the same probabil-ity, without learning the phrase "I love pasta".

Certifying Answers of Boolean SAT-Solvers with COQ and OCAML

Vandendorpe Thomas
Supervised by: Sylvain Boulmé.

I understand what plagiarism entails and I declare that this report is my own, original work.
Name, date and signature:

Abstract

This paper presents SATANS-CERT an OCAML program, certified in COQ, that wraps state-of-the-art Boolean SAT-solvers in order to check their answers. Hence, this tool increases trust in answers of these highly optimized and still evolving programs (themselves written in C/C++).

1 Introduction

Boolean Satisfiability solvers are now used in critical system industry. Ensuring their correctness thus seems important. But certifying efficient SAT-solvers written in C/C++ is difficult. It is easier to only certify a checker of their results.

Below, Section 2 briefly recalls fundamentals about Boolean SAT-problem. Section 3 sketches the state-of-the-art checking of SAT-solver answers. Section 4 presents our checker. Finally, Section 5 concludes from our benchmarks.

2 Preliminaries

A literal is either a variable or its negation, a clause is a disjunction of literals and a conjunctive normal form (CNF) is a conjunction of clause. A model for a formula F is an assignment of variables satisfying F. The resolution rule where c_1 and c_2 are disjunctions of literals, l and $\neg l$ are literals:

$$\frac{l \vee c_1 \qquad \neg l \vee c_2}{c_1 \vee c_2}\ \text{resolution}$$

Resolution rule ensures that any common model to the two input clause is also a model of the resolvant.

DPLL is an algorithm used by SAT solver that consists to recursively: choose an unassigned variable, assign a value to the variable, simplify the CNF. If the simplified CNF is trivally true, the current assignment is a model of the input CNF. If it is trivially false, there is a conflict which requires to backtrack with another assignment. CDCL [Silva *et al.*, 2009] is a refinement of DPLL: on each conflict, some new clauses are *learned* and added to the CNF (actually, they are implied by the input CNF and will avoid many “similar” conflicts). Learning the empty clause means that the CNF is unsatisfiable.

3 Checking UNSAT answers: a state of the art

While verifying a SAT answer is very easy (only needing to evaluate the formula on the assignment), verifying UNSAT answers is more complex. In theory, it reduces to check resolution proofs. But modifying a given SAT solver to produce such proofs is difficult and very inefficient because of their huge size. Thus, state-of-the-art solvers produce shorter proof traces, which require more work from the checker. Currently, there are several formats of proof traces. The most standard one is RUP (Reverse Unit Propagation), simply listing learned clauses from CDCL. The RAT format (Resolution Asymetric Tautology [Heule *et al.*, 2013]) is an extension of the RUP format, allowing to introduce new boolean variables during the proof.

The standard RUP/RAT checker is DRAT-TRIM [Wetzler *et al.*, 2014]. This tool is itself not certified: it is a C program using optimized data-structures. This tools is also able to optimize the proof trace in input, and to output a LRAT proof, adding informations to the RAT proof in order to check it with resolutions.

4 Contribution

We have developed SATANS-CERT to certify SAT-solvers answers. It includes a LRAT proof checker which is certified by using a new type of interaction between COQ and OCAML [Boulmé, 2018] – allowing to delegate a part of the COQ certification to the OCAML typechecker. The Trusted Computing Base (TCB) of our proof thus contains the Coq proof assistant, the OCaml compiler, and our process to link our OCaml oracles to our Coq code through Coq extraction. It also contains our OCAML parser of the input CNF in DIMACS format: only the computations from the CNF in abstract syntax are certified. In brief, SATANS-CERT first parses the formula, and then runs the SAT solver (given in parameter):

- If its anwer is SAT, the model is simply checked by a coq-certified checker.
- If the answer is UNSAT, DRAT-TRIM is executed to produce an optimized LRAT proof. The OCAML part of the LRAT checker learns new clauses (using COQ certified resolution functions) from existing clauses in order to produce the empty clause.

5 Conclusion

Our benchmarks shows that the running time of our checker is lower than the one of DRAT-TRIM. This implies that our checker is much faster than the previous LRAT checker written in coq (this latter is much slower than DRAT-TRIM) [Cruz-Filipe *et al.*, 2017]. But it remains a bit slower than other checker written in other languages as ACL2 [Cruz-Filipe *et al.*, 2017] or using an alternative format to LRAT (like GRAT [Lammich, 2017]). Actually, DRAT-TRIM is the bottleneck of our tool chain. In conclusion, some improvements remain to be done on our checker: using experimental machine integers of Coq (instead of the less efficient but standard binary integers of Coq); take into consideration other recent proof formats like GRAT; certify inputs/outputs.

Acknowledgments

Work supported by grant ERC Stator of David Monniaux.

References

[Boulmé, 2018] Sylvain Boulmé. What is the Foreign Function Interface of the Coq Programming Language? Coq Workshop 2018, 2018.

[Cruz-Filipe *et al.*, 2017] Luís Cruz-Filipe, Marijn J.H. Heule, Warren A. Hunt Jr., Matt Kaufmann, and Peter Schneider-Kamp. Efficient certified rat verification. *Automated Deduction - CADE-26*, pages 220–236, 2017.

[Heule *et al.*, 2013] Marijn J. H. Heule, Warren A. Hunt Jr., and Nathan Wetzler. Verifying refutations with extended resolution. *Automated Deduction - CADE-24*, pages 345–359, 2013.

[Lammich, 2017] Peter Lammich. The grat tool chain: Efficient (un)sat certificate checking with formal correctness. *Proc. of SAT 2017*, 2017.

[Silva *et al.*, 2009] João P. Marques Silva, Inês Lynce, and Sharad Malik. Conflict-driven clause learning SAT solvers. In *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 131–153. IOS Press, 2009.

[Wetzler *et al.*, 2014] N. Wetzler, M.J.H. Heule, and W.A. Hunt, Jr. Drat-trim: Efficient checking and trimming using expressive clausal proofs. *Theory and Applications of Satisfiability Testing – SAT 2014*, pages 422–429, 2014.

Certifying Answers of Boolean SAT-Solvers with Coq

github.com/boulme/satans-cert

Sylvain Boulmé
Sylvain.Boulme@univ-grenoble-alpes.fr

Thomas Vandendorpe
Thomas.Vandendorpe@etu.univ-grenoble-alpes.fr

State-of-the-art in Boolean SAT-Solving

Features

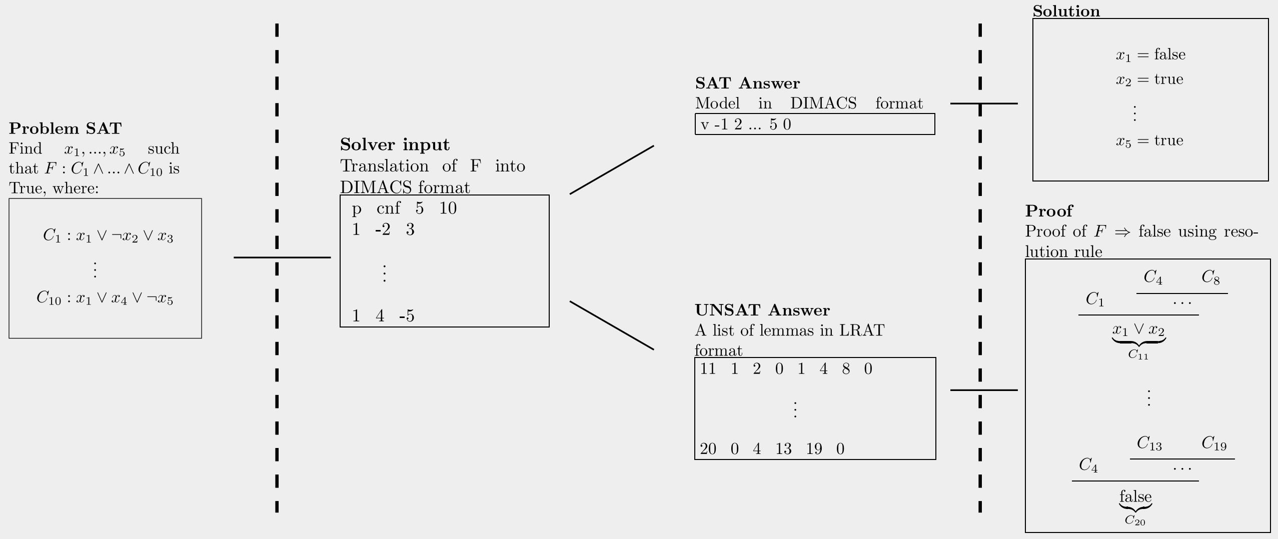
- Continuous efficiency increase of C/C++ Solvers through "SAT competitions"
- Decide satisfiability of **Large** Boolean Formulas : thousands of variables and millions of boolean constraints
- Output **certificates**, ie **checkable** answers

Applications in Electronic Design Automation

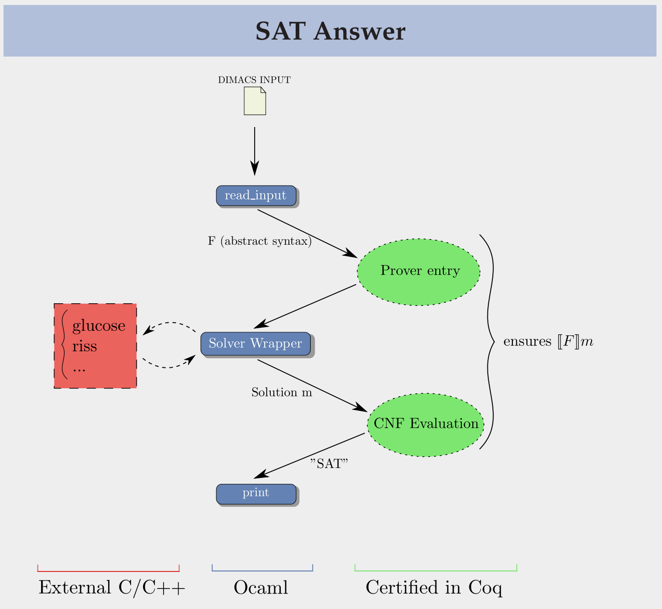
- Formal equivalence checking
- Model-checking
- Routing of FPGA
- Automatic test pattern generation
- Automated planning and scheduling
- ...

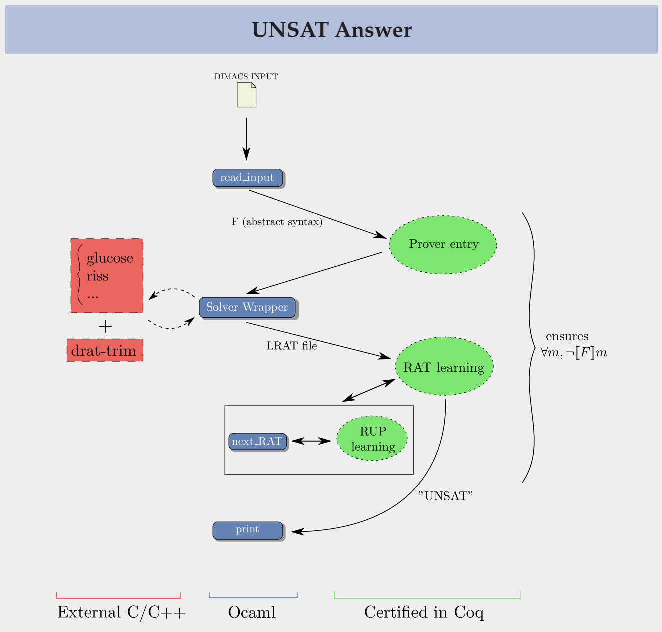
Others: artificial intelligence, theorem proving, software dependencies, ...

Checking answers of modern SAT-Solvers


Problem SAT
Find $x_1, ..., x_5$ such that $F : C_1 \wedge ... \wedge C_{10}$ is True, where:
$C_1 : x_1 \vee \neg x_2 \vee x_3$
$\vdots$
$C_{10} : x_1 \vee x_4 \vee \neg x_5$
Solver input
Translation of F into DIMACS format
p cnf 5 10
1 -2 3
$\vdots$
1 4 -5
SAT Answer
Model in DIMACS format
v -1 2 ... 5 0
UNSAT Answer
A list of lemmas in LRAT format
11 1 2 0 1 4 8 0
$\vdots$
20 0 4 13 19 0
Solution
$x_1 = \text{false}$
$x_2 = \text{true}$
$\vdots$
$x_5 = \text{true}$
Proof
Proof of $F \Rightarrow$ false using resolution rule
C_4 C_8 C_1 $\dots$
$\underbrace{x_1 \vee x_2}_{C_{11}}$
$\vdots$
C_{13} C_{19} C_4 $\dots$
$\underbrace{\text{false}}_{C_{20}}$

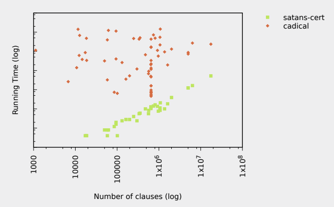
Architecture of satans-cert

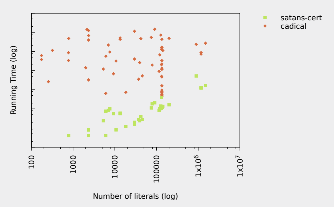




Benchmarks

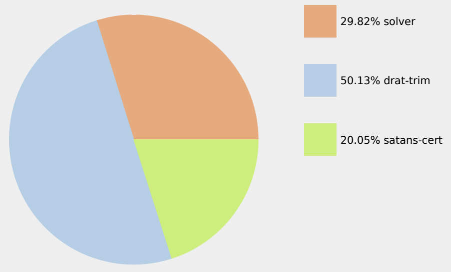
Mean running times (SAT)





Running with the Cadical SAT-solver on the 120 instances of the SAT competition 2018 benchmarks.

Mean running times (UNSAT)



Running with both Cadical and Cryptominisat SAT-solvers on 306 instances from the SAT competition 2015,2016,2018 benchmarks.