Magistère d'informatique 2017

Grand Amphithéatre de l'IMAG

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

Mercredi 30 Août

Enzo Brignon - 9:00 AM
Bare metal solutions to implement online monitoring for embedded software

Maxime Calka - 9:30 AM
Developing 3D nonrigid image registration techniques within CamiTK

Narek Davtyan - 10:15 AM
Test Architecture for Network Experimentation Platform WalT

Florian Marco - 10:50 AM
Modular Intruders and Stateful Protocols in ProVerif

Christopher Ferreira - 1:30 PM
Understanding the Thread Placement Strategy of the Linux Scheduler

Claude Goubet - 2:15 PM
Scalable Image Reconstruction Methods for Large Data. Application to Synchrotron CT of Biological Samples

Thomas Lavocat - 3:10 PM
Execution Management for Exascale Machines and InSitu Applications

Jeudi 31 Août

Steve Roques - 9:00 AM
Formal verification of a new cache coherence protocol

Lenaic Terrier - 9:40 AM
A Language for the Smart Home

Luc Libralesso - 10:35 AM
Steps toward designing brains on a chip

CONFERENCES OUVERTES AU PUBLIC

UNIVERSITÉ Grenoble Alpes

Magistère d'Informatique de Grenoble

Proceedings of MIG'2017

Organized by Michaël Périn and Cyril Labbé

August, 30-31, 2017

Program

Bare metal solutions to implement online monitoring for embedded software

Enzo Brignon
Magistère L3 Informatique, 2016/2017
Supervised by : Laurence Pierre
Laboratory : TIMA, Team : AMfoRS

1 Context

The context of this work is the online verification of temporal properties for embedded software. As an alternative to solutions like the ones of [Havelund, 2008], [Drusinsky, 2000], [Navabpour *et al.*, 2013], the AMfoRS team has developed the OSIRIS tool[Chabot *et al.*, 2015] that automatically generates assertion checkers from temporal properties and instruments C programs with these monitoring components, together with an observation mechanism that enables their event-driven activation during the program execution. These properties are formalized in PSL (Property Specification Language), IEEE Standard 1850)[PSL, 2005], for example:

`always (y#SET() -> y > 0)`

means that every time the variable y is set, it has to be greater than zero.

2 Existing solution

As explained in [Chabot *et al.*, 2015], OSIRIS takes as input the binary file of the application and produces an infrastructure that enables surveillance during execution. This infrastructure assumes a POSIX-compliant OS. It enables to run *two processes concurrently*:

- one that executes the original application.
- and one (called the Tracer) that observes its execution and activates the checkers when needed.

To that goal, the "ptrace" system call is used by the Tracer to put breakpoints on observed events in the C application (see for example [How debuggers work, 2011]) and to read the values of its variables. Those breakpoints correspond to statements on which one of the properties has to be re-evaluated. For example the property of section 1 must be re-evaluated at each statement that modifies y.

While runing, everytime the Tracer receives a signal because the tracee has stopped on a breakpoint, it notifies the corresponding assertion checkers. Finally the Tracer orders the tracee to resume its execution. To place a breakpoint on an assembly instruction I at address A in the original binary file, the Tracer has in fact replaced this instruction I by a specific instruction INT 3 (on x86 processors). So, before resuming the tracee execution, it must also order it to execute the replaced instruction I.

This solution has several disadvantages : it requires an operating system, creates two processes and uses ptrace for process synchronization and memory inspection. Moreover due to the *temporary* remplacement of instructions by INT 3, it has a huge CPU time overhead.

The goal of the magistere internship is to propose a less restrictive (that does not require an embedded OS) and more efficient solution.

3 Contribution

3.1 Principle

The software interrupts[SWI, 2001] are signals that are sent to the processor to make it suspend its current activity, save the state and execute an interrupt handler. In the proposed solution each assembly instruction (call it I) on which one of the properties has to be re-evaluated will be *definitively* replaced by a software interrupt instruction (SWI), such that the processor will enter the interrupt handler. Then the handler executes the replaced instruction I and calls the assertion checker to re-evaluate the corresponding property. Entering the handler function implies a change of context (addresses, position in the stack, etc...). Therefore the code generated by OSIRIS to execute instruction I must first restore an appropriate context.

3.2 Interrupt handling

We specified the interrupt handler that will be generated by OSIRIS, which is written in the C language and has two tasks to perform : execute the replaced instruction and call the assertion checkers to evaluate the corresponding properties. The handler (which is called `C_SWI_handler`) receives two parameters :

- The SWI number, used to identify the SWI position.
- The address of the top of the stack.

The `C_SWI_handler` uses the SWI number to identify on which interrupt the program has been suspended hence to know which instruction it has to execute and which assertion checkers it has to wake up (it uses a classical switch/case conditional structure).

3.3 Adaptations to ARM

OSIRIS was first designed for x86 processors. We have also adapted its observation context to ARM processors. To that goal, DWARF debugging information is used to compute the addresses of the observed instructions[DWA, 2013].

3.4 Experiments

This new instrumentation model has been tested on STM32F4 boards (equipped with an ARM Cortex M4 processor) on which instrumentation of small applications works fine but not on bigger ones due to the lack of memory. The configuration of the interrupt handler has been added to the startup file of the board[M4m, 2010], [Sim, 2010].

Other experiments have been made on Raspberry Pi 2 board (that has a ARM Cortex A7 processor). A "minimal kernel" has been adapted to make the C and C++ runtime usable on this board[Tut, 2014].

References

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

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

[DWA, 2013] Exploring the DWARF debug format information. https://www.ibm.com/developerworks/aix/library/au-dwarf-debug-format/index.html, 2013.

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

[How debuggers work, 2011] How debuggers work. http://eli.thegreenplace.net/2011/01/27/how-debuggers-work-part-2-breakpoints, 2011.

[M4m, 2010] ARM Cortex M4 exception model. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0553a/Babefdjc.html, 2010.

[Navabpour *et al.*, 2013] 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.

[PSL, 2005] *IEEE Std 1850-2005, Standard for Property Specification Language (PSL)*. 2005.

[Sim, 2010] Simplest bare metal program for ARM. https://balau82.wordpress.com/2010/02/14/simplest-bare-metal-program-for-arm/, 2010.

[SWI, 2001] SWI handlers in C and assembly language. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0056d/ch05s04s02.html, 2001.

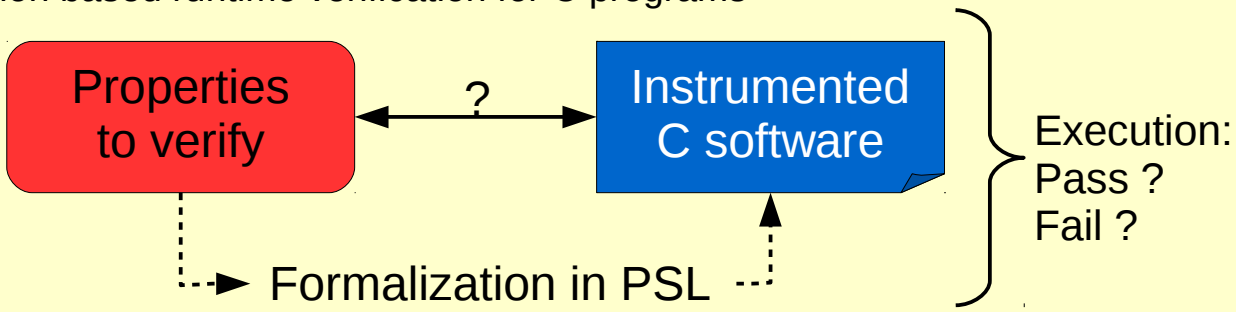
[Tut, 2014] Tutorial to do bare metal on RPI2. http://www.valvers.com/open-software/raspberry-pi/step01-bare-metal-programming-in-cpt1/, 2014.

Bare metal solutions to implement online monitoring for embedded software

Enzo Brignon
Supervisor : Laurence Pierre

Context

Assertion based runtime verification for C programs

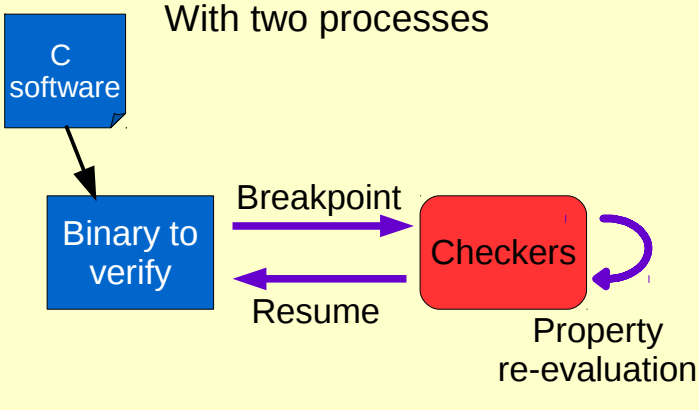


Example : The variable v becomes greater than 100 before the function f is called :

```
(v#SET() && v > 100) before! f#START()
```

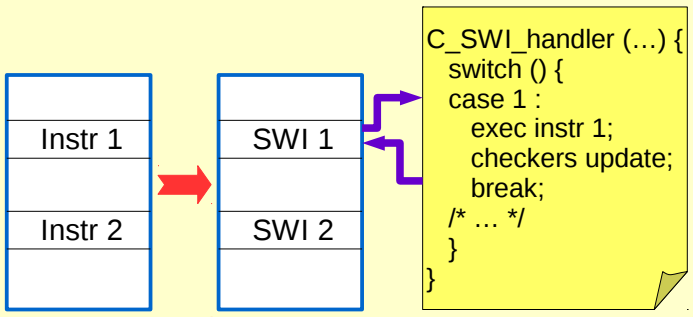
Existing solution

Ptrace-based solution
With two processes

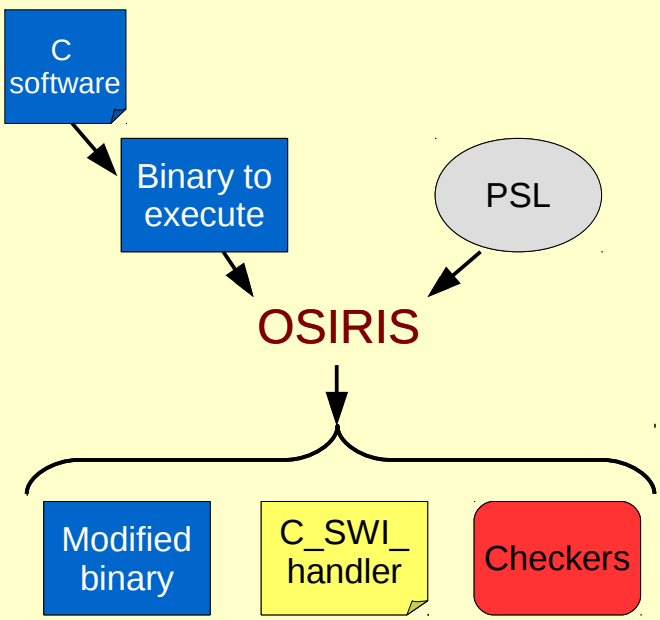


New solution

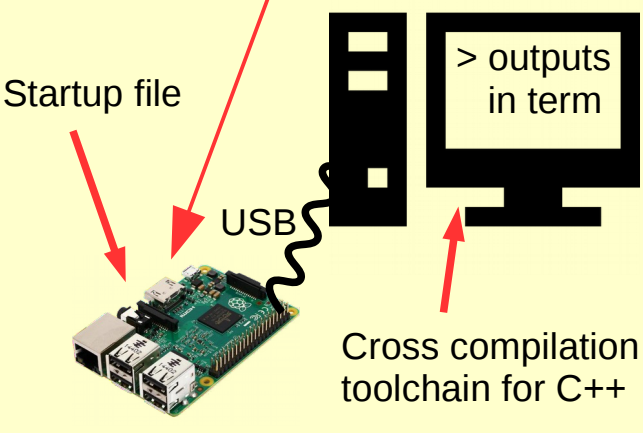
Software interrupt-based solution



New version of the tool



Implementation



STM32F4 (ARM Cortex M4)
Raspberry Pi 2 (ARM Cortex A7)

Developing 3D nonrigid image registration techniques within CamiTK

CALKA Maxime
Maxime.Calka@etu.univ-grenoble-alpes.fr

TIMC - GMCAO, Grenoble, France

Abstract

The aim of this internship was to implement 3D image registration methods in the Open Source platform CamiTK developed by the CAMI Team. To implement this functionalities, we have used the library Elastix developed by Stafan Kelin and Marius Staring.[3] This library is a toolbox permitting the rigid and non-rigid image registration. Finally, we have succeed to include the library in CamiTK and to create an "Action" operating the B-spline Free-Form Deformation (FFD) based registration.
Keywords: Image Registration, CamiTK, Elastix

1 Introduction

1.1 CamiTK

CamiTK is a computer assisted medical interventions Tool Kit developed by the GMCAO team. This software has been created in the aim to gather the knowledges from several research fields : perception (visualization, interaction, processing and analysis), reasoning (3D geometries, interaction, biomechanics), and action (tracking, navigation, robot control).

this software provides fast and stable tools for prototyping medical applications for clinicians and the medical industry.

1.2 Nonrigid registration

The image registration is the process of transforming different sets of data into one coordinate system. On this matter, our Group has previously developed and employed nonrigid image registration techniques on the basis of discrete optimization.[1]

2 Aim

The aim of my internship was to embed the FFDs registration framework into the CamiTK.

3 Methodology

The principle objective of this internship was to embed FFD registrations into CamiTK. Our group has previously employed discrete optimization techniques for optimization of nonrigid transformation. This method was primarily implemented in Matlab and is now being developed within C++ language. Although, this implementation is not finished yet, I have created a CamiTK Action that can bring this library to the CamiTK in the future. On the side, a third party library so-called Elastix that employs conventional continues optimization techniques for estimation of non-rigid transformation was chosen.

I have install Elastix 4.8 using CMake 3.1 and ITK 4.9. In addition, I have link Elastix with CamiTK 4.0. I have create an example program using the Elastix library who realise the registration of 2 images.

I have created a CamiTK Action with the CamiTK wizard. Then, I have implement the image registration program into the CamiTK Action. According to the registration process being employed (i.e., similarity measure, interpolation method, optimization technique, etc), some parameters must be configured. I have developed an IHM with QT permitting the parametrization of the image registration. This interface has been linked at my CamiTK Action.

4 Results & discussion

Finally, a CamiTK Action realising the BSpline registration has been created with the possibility parametrize each part of the nonrigid registration.

I have not tested the quality of the result because I don't know how make an efficient parametrization, but I have seen that the quality of the result and the execution time are in relation with the parametrization and the type of optimization.

The next step will be to compare the Elastix Optimization (continuous optimization) and the Optimization developed by our group (discrete optimization) in order to evaluate their performance and simulation speed.

It is worth to mention that my implemented CamiTK Action can be used to employ other image registration techniques provided by Elastix. However, the most difficult and time consuming step will be creation of an interface that parametrizes each registration method.

5 Conclusions

The program could be used to apply the transformation on finite elements meshes and to develop an another application

realising the automatic generation of subject-specific finite element meshes.

References

[1] A. Bijar, P.-Y. Rohan, P. Perrier, and Y. Payan. Atlas-based automatic generation of subject-specific finite element tongue meshes. *Annals of biomedical engineering*, 44(1):16–34, 2016.

[2] J. Shackelford, N. Kandasamy, and G. Sharp. Chapter 47 - deformable volumetric registration using b-splines. In W.-m. W. Hwu, editor, {*GPU*} *Computing Gems Emerald Edition*, Applications of GPU Computing Series, pages 751 – 770. Morgan Kaufmann, Boston, 2011.

[3] Stefan Klein and Marius Staring and Keelin Murphy and Max A. Viergever and Josien P.W. Pluim. elastix: a toolbox for intensity-based medical image registration. *IEEE Transactions on Medical Imaging*, 29(1):196 – 205, January 2010.

Developing 3D nonrigid image registration techniques within CamiTK

CALKA Maxime, L3 Informatics
TIMC – GMCAO, Université Grenoble Alpes

INTRODUCTION

1. CamiTK
   - Open Source Platform
   - Knowledge gathering
   - Prototype medical application
   - Cross platform
   - Free and open source
   - Easy integration of algorithms

2. Nonrigid image registration
   - Consist in the image matching
   - Use for medical images, to register a patient's data to an anatomical atlas
   - Our Group has developed nonrigid image registration techniques on the basis of discrete optimization.

OBJECTIVE

Embed the FFDs registration framework into the CamiTK

METHODS

- Elastix library to implements the image registration methods
- Embed image registration methods in CamiTK with CamiTK wizard/IMP
- QT Creator to add an IHM to parameterize the registration
- Link the interface to the CamiTK extension

RESULTS

- Creation of a CamiTK Action realising the Bspline registration with the possibility parametrize each part of the nonrigid registration.
- The quality of the result has not been tested but we have seen that the quality and the execution time are in relation with the parametrization and the type of optimization.
- The next step : compare the both Optimization to evaluate their performance and simulation speed. (continous optimization and discrete optimization)

RESULTS

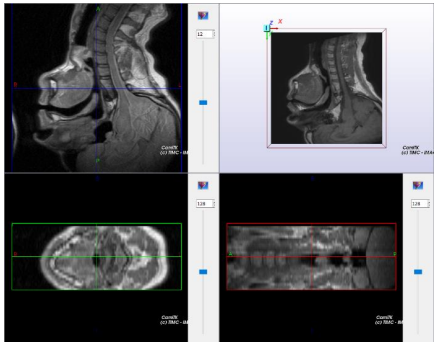


Figure 1: *Source image (moving)*

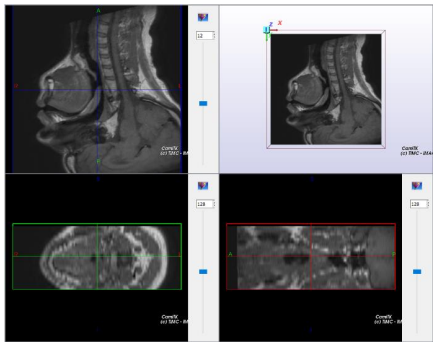


Figure 2: *Target image (fixed)*

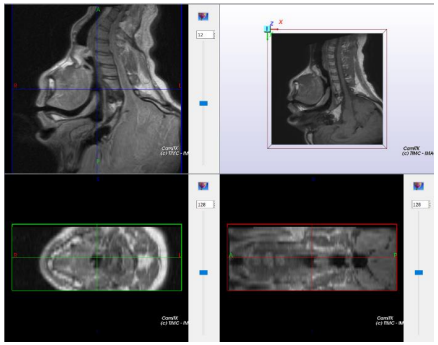


Figure 3: *Result image*

CONCLUSIONS

- The program could be using for several applications because the image registration is very important in medical imaging
- Possibility to develop a program for the automatic generation of subject-specific finite element meshes thanks to the transformation obtained with the FFD program.

Test Architecture for Network Experimentation Platform WalT
Extended Abstract

Narek Davtyan
DRAKKAR (LIG) and IM^{2}AG (UGA)
Grenoble, France
Narek.Davtyan@univ-grenoble-alpes.fr

Supervised by: Etienne Dublé

1 Introduction

The WalT platform allows to experiment network research. It ensures repeatability and reproducibility of experiments.

WalT nodes (clients) are computers on which an OS (file-system and kernel) packaged as a *docker* image can be deployed for customization and sharing. WalT server is a pc taking charge of communication between users and nodes.

With free software and standard components WalT platform can be easily reproduced to validate results in various real-world conditions. The platform can be set up on a desk for debugging or transported for mobile demos. The remote control over nodes is performed using a command line client which sends requests to the server. It allows actions such as node rebooting, shell sessions, experiment log management, manipulations of OS images (deploying, modifying, cloning from and pushing to docker hub). Due to the versatility of the platform as well as its complete control over WalT nodes, diverse experiment scenarios can be set up ranging from Wi-Fi handover measurements to evaluating routing protocols in wireless sensor networks.

Due to new developments the need of systematic testing of diverse features and functionalities of WalT platform becomes necessary. Automating the testing of a distributed system is a difficult task. In order to carry it out, rather than using a real physical platform, virtual nodes running as vms are to be designed and implemented. In some cases, virtual nodes might also be useful in some research experiments.

Hence the objective of this internship is to introduce virtual nodes into WalT server as well as implementing the test module itself.

1.1 Physical Nodes and Boot Procedure

At the time the internship started, WalT physical nodes were raspberry pi boards. They use a network boot procedure: first, a network boot-loader stored locally on the node (SD-card) is started; then, an OS (so-called "WalT image") stored on the server and chosen by the user is started.

2 Problems and Solutions

It is easier to automate the testing of the platform using virtual nodes, than real ones. Since the testing only makes sense if virtual nodes function like physical ones, a virtual machine is used: kvm.

Having several virtual machines running on the server may cause performance issues. To minimize them, we decided to design minimalist WalT images. Moreover, to avoid any CPU emulation, we decided to implement virtual nodes using the same architecture as the server (host machine), thus PC-type virtual nodes.

2.1 Support of PC-type nodes

A first step was to add support for the PC-architecture to regular (i.e. physical) WalT nodes. This implied to implement a network boot procedure for PC-type nodes, similar to the one that already existed for rapsberry pi boards.

The creation of such boot procedure resulted in a bootable USB device that allows turning any PC into a WalT node. This USB device works by loading two bootloaders in turn: grub and then ipxe.

2.2 Design of Minimalist Images

As explained above, we also designed minimalist WalT images based on buildroot in order to minimize the performance impact of virtual nodes. Virtual nodes, being virtual machines, require large amount of computing resources on the server, therefore it's wise to limit the demands of the OS.

It was the first task of the internship, because it allowed to understand various topics and internals related to the WalT platform. At this time, PC-architecture support was not available, thus we started by designing minimalist images for the raspberry pi boards. Later in the project, minimalist images for the PC architecture were also implemented.

2.3 Support of Virtual Nodes

The use of kvm allowed almost immediately to turn physical PC nodes into virtual nodes, by using the USB bootloader image handing over to the minimalist WalT image.

The integration of this node type into the platform included the launch of such virtual machines and running them on the background.

2.4 Design of Test Module

The test module is designed in a way that regularly tests diverse features of WalT. It consists of sufficient quantity of scenarios that would be otherwise performed by a user.

Bibliography ?

Test Architecture for Network Experimentation Platform WalT

Narek Davtyan - L3 Info UGA
Supervised by Etienne Dublé

Description and objectives

- The network experimentation platform WalT allows to experiment network research. It ensures repeatability and reproducibility of such experiments
- **WalT nodes** — computers on which an OS (filesystem and kernel) packaged as a docker image can be deployed for customization and sharing
- **WalT server** — a pc taking charge of communication between users and nodes
- The test architecture
  - Due to new developments the need of testing of diverse features and functionalities of the WalT platform becomes necessary
  - Automating the testing of WalT consists of designing the test module and virtual nodes
- The choice of virtual nodes
  - Virtual nodes are easier to manage in the context of an automated module, and less subject to failure

Support of PC-type Nodes

- USB boot image contains only a few files
  - The customized grub2 bootloader has no menu and starts ipxe network
- Boot procedure resembles greatly the one created for raspberry pi boards
  - Locally-stored grub2 bootloader starts ipxe with boot instructions as arguments
  - Ipxe network bootloader gets its IP (DHCP) from WalT server and downloads (TFTP) a 2nd stage script stored on the server
  - This scripts downloads the kernel and boots it
  - The kernel mounts the OS stored on the server (NFS)

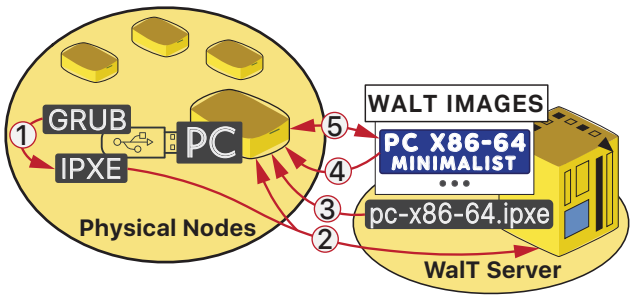


Figure 1: The boot procedure for pc architecture is shown: 1 - grub2 launches ipxe network bootloader, 2 - ipxe sends dhcp request with its identifier and receives an ip address and network configuration, 3 - the node downloads a second stage script, 4 - the script downloads (via TFTP) the appropriate kernel, 5 - the kernel is started and boots (via NFS) the OS located in the chosen image.

Design of Minimalist Images

- Performance issues can arise, since virtual nodes are in fact virtual machines running on the server, hence they need to consume the least possible
  - Minimalist images are used to limit the demands of the OS
  - CPU emulation provided by qemu is avoided by choosing the same architecture on both the server and the nodes, which is pc x86-64, thus the corresponding images (pc type) are used

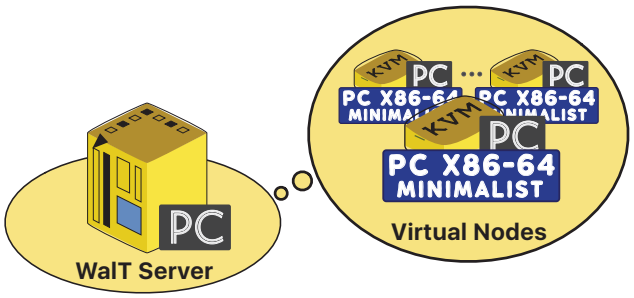


Figure 2: The indefinite amount of virtual nodes is shown. Both the nodes and the server have the same architecture (pc x86-64). Virtual nodes run minimalist images.

Support of Virtual Nodes

- Virtual nodes work like the physical ones
  - kvm is used to instantiate nodes
  - The same USB boot image is used to start the node
  - The same minimalist images are deployed on the virtual nodes
  - The same boot procedure is used to load the kernel to the vm
  - Virtual machines run on the background and have the same behaviour as physical ones

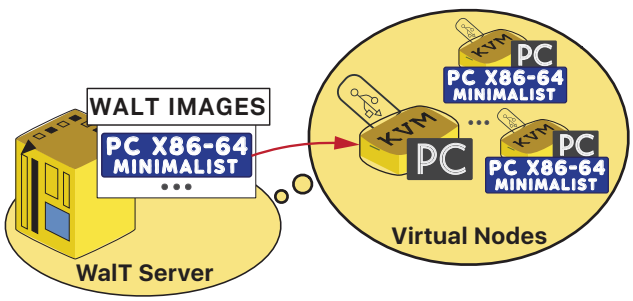


Figure 3: The same boot procedure (simplified), which results in obtaining the image, is shown with the same USB boot image and the same minimalist walt images.

Design of Test Module

- Test module verifies the correct behaviour of the platform
  - It deploys minimalist images and tries various predefined scenarios on them, which results in verification of specified WalT features

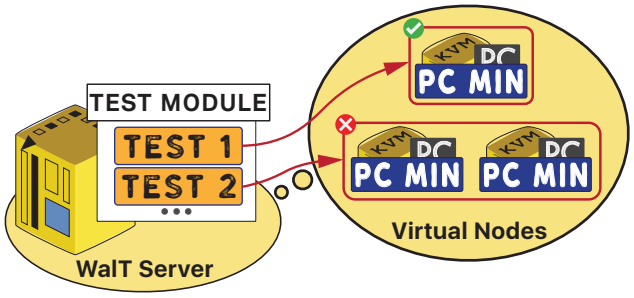


Figure 4: Two tests applied to three virtual nodes are shown: the first one (running on the node above) verified the correct behaviour of certain features using the specified scenario, the second one (running on the two nodes below) found an incorrect result for a certain test. Arrows represent the application of a certain test to a certain virtual node or group of virtual nodes running minimalist pc x86-64 images.

Modular Intruders and Stateful Protocols in ProVerif

Florian Marco - VERIMAG - Univ. Grenoble Alpes / Grenoble-INP, France

1 Introduction

In the context of industrial systems (or SCADA), devices can be used for more than 20 years and are hard to patch in case of vulnerability. Due to their central place, an attack could be disastrous such as the Stuxnet worm against nuclear facilities in Iran [Langner, 2011]. Industrial systems are well analyzed in terms of safety but such properties are usually not considered in presence of attackers. Thus we aim to check the safety properties of industrial systems in presence of an attacker.

1.1 Formal security proof and ProVerif

By creating a model of a cryptoraphic protocol, we're able to explicit the security properties that we want to be respected. The attacker is modelized as an omniscient intruder, that got all possible powers except breaking cryptography. This intruder named Dolev-Yao [Dolev and Yao, 1981] was used to highlight "Man In The Middle" attacks. We choose to use ProVerif [Blanchet, 2001], an automatic cryptographic protocol verifier. It is using an extended form of Pi-Calculus, a process algebra that allows concurrent computations and replication and is really powerful [Lafourcade and Puys, 2015]. Agents are communicating with each others using channels. During this analysis it will try every possible situation in order to determine if the security properties are violated or not .

1.2 Modular Intruder

According to its knowledge, the Dolev-Yao intruder can listen, forge, replay or modify messages, and play multiple sessions of a protocol. When we assess the security of an industrial system, we will find most of the time possible attacks due to the weak protocols used (like Modbus). In order to test our system against a realistic intruder, we propose to control the capacities of our modular intruder.

2 Industrial System Modelling

Cryptographic protocols are a finite and fixed sequences of messages aiming to guarantee some property at the end of the protocol, like secrecy, that targets the global state of the protocol. In industrial systems, we are studying potentially infinite sequences of commands. We need to check logical predicates on the variables of the system, which change depending on the commands exchanged. In order to do that, we need to model a global state of the system.

2.1 Limitation of ProVerif

We found that ProVerif is performing coarse abstractions that prevents us from easily modelling our system. They are necessary in order to verify cryptographic protocols. The Pi-Calculus models are translated into Horn clauses where the set of facts grows monotonically. Thus an old state cannot simply be replaced by a new one. Moreover, no loop or recursions are possible in ProVerif. Only replication is possible, but it creates a new copy of the process in parallel of the older one.

2.2 Construction encompassing those limitations

First, we encode memory as a chained list of variables and values. A memory accessing function will look recursively in order to retrieve the current value of a specific variable. it also allows an observer to check logical predicates on the variables. Secondly, in order to avoid mixed states and messages when using replication, we always communicate through fresh channels that we created for each replicated process. Finally, we need to maintain an order in the execution of all copies for which we use counters modelled as Peano numbers. Thus, all messages exchanged are tagged with a counter in order to only be received by the intended process. The association of these three modelling tricks allows us to model a sound execution of the system.

3 Modular Intruder Model in ProVerif

In the appendix we added a representation of our system architecture and the general topology of an industrial network.

3.1 Intruder definition

The Dolev-Yao intruder is generally assumed to control a single public channel used by all agents when analyzing cryptographic protocols [Cervesato, 2001]. In order to control its behaviour all agents must communicate over private channels. Accordingly, the intruder cannot do anything. Thus we need to provide him "oracles" that will work as proxies and let him access those privates channels.

3.2 Oracle implemantation

The oracles are communicating with the Dolev-Yao intruder using public channels. We modelled each oracle to perform a specific action, e.g. a wiretape oracle will broadcast all the messages to the intruder in order to increase its knowledge. a Modify Oracle will modify (parts of) messages exchanged, etc. We faced limitations when trying to implement oracle that act on their own (e.g. replaying or forging) and which will not require a message sent by the client. We supposed that the possibly infinite behaviour these oracles requires a dedicated abstraction in ProVerifs algorithm.

4 Conclusion

We successfully used ProVerif to analyze the safety of industrial systems. However the modelling tricks we needed to use still do not scale to real examples. One of the future possibility is to use another tool such as Tamarin.

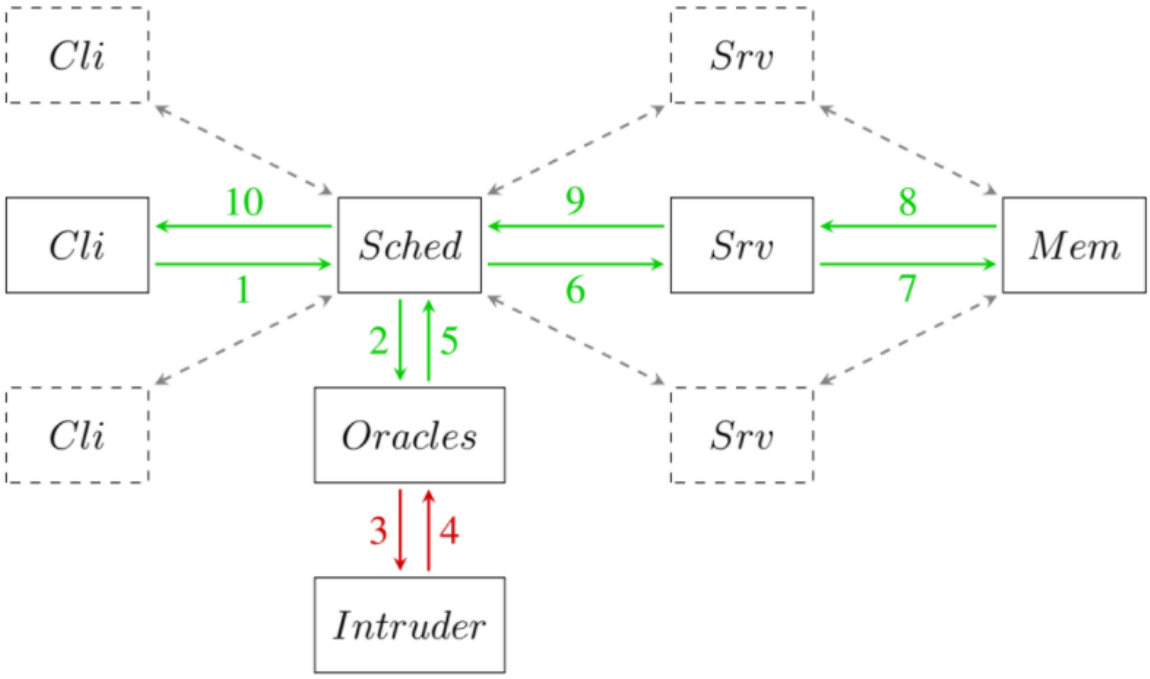


Figure 1 : System Architecture

References

[Blanchet, 2001] Bruno Blanchet. An efficient cryptographic protocol verifier based on Prolog rules. In *CSFW'01*, 2001.

[Cervesato, 2001] Iliano Cervesato. The dolev-yao intruder is the most powerful attacker. In *16th Annual Symposium on Logic in Computer ScienceLICS*, volume 1, 2001.

[Dolev and Yao, 1981] D. Dolev and Andrew C. Yao. On the security of public key protocols. *Information Theory, IEEE Transactions on*, 29(2):198–208, March 1981.

[Lafourcade and Puys, 2015] Pascal Lafourcade and Maxime Puys. Performance evaluations of cryptographic protocols verification tools dealing with algebraic properties. In *FPS'2015*, 2015.

[Langner, 2011] Ralph Langner. Stuxnet: Dissecting a cyberwarfare weapon. *Security & Privacy, IEEE*, 9(3):49–51, 2011.

Understanding the Thread Placement Strategy of the Linux Scheduler
Magistère M1 Report

Christopher Ferreira
LIG, ERODS Team

Supervised by: **Renaud Lachaize** and **Vivien Quéma**
With the help of: **Hugo Guiroux**
<firstname>.<lastname>@univ-grenoble-alpes.fr

I understand what plagiarism entails and I declare that this report is my own, original work. Christopher Ferreira

Abstract

Scheduling threads on multicore machines remains a challenging problem. Empirical evidence shows that the thread placement strategy implemented in the Linux scheduler is sometimes significantly less efficient than simple static schemes. However, such performance problems are currently difficult to explain because of two things. First, over the past ten years, the lack of a documented overall design and the frequent changes have turned the source code of this strategy into a patchwork full of intricacies and unclear heuristics that is hard to comprehend and maintain. Second, given the complexity of this thread placement strategy, new tracing facilities are necessary to obtain detailed insight about the thread placement decisions.

The main contribution of this work is an attempt to address these two shortcomings regarding thread placement decisions in Linux. We first propose an exhaustive documentation of the overall thread placement strategy implemented in the Linux scheduler. We then describe our own work-in-progress tool created to gather runtime data about the successive thread placement decisions that are made during the execution of an application. We believe this contribution is an important first step towards the understanding of these inefficiencies which could eventually help improving the Linux scheduler.

1 Introduction

General purpose schedulers were supposed to be a relatively simple matter until the early 2000s: before the advent of multicore processors, the problem of scheduling threads was simply a matter of time sharing a single CPU between all the running threads. On a multicore machine, while time slicing remains necessary, a scheduler also has the responsibility to choose the placement of threads on the available CPUs. Devising an efficient thread placement strategy is a complicated issue because a scheduler must attempt to maximize the utilization of all the available CPUs (by trying not to leave any CPU idle and keeping the load balanced among the different CPUs), while taking into account *memory affinity* constraints, i.e., differences in memory access costs depending on the CPU. These memory affinity constraints stem from the complex cache hierarchies and *NUMA* (Non Uniform Memory Access) effects[1] of modern hardware architectures [Zhuravlev *et al.*, 2012; Diener *et al.*, 2016]. In particular, migrating a thread from a given CPU to another may hurt performance because the data accessed by the thread may not be found in the cache of the new CPU and may have to be fetched from a remote cache, from main memory or on a remote memory node. On their own, these concerns (using all the CPUs, load balance, memory affinities) are already difficult to meet, but their combination raises even more complexity. For one thing, the implementation becomes much more intricate. More importantly, they are sometimes contradicting: for one instance of a thread placement decision it is difficult to assess beforehand which concern will have the most impact on performance. As an illustration, moving a thread farther away from its NUMA memory node in order to improve the overall load balance might consequently improve the performance. However this gain of performance can also be negated by the overhead of memory accesses from a remote NUMA node.

Our study focuses on Linux version 4.9 (released on 2016-12-11), and more precisely on its general purpose scheduler *CFS* ("Completely Fair Scheduler") [Molnar *et al.*, 2017; Volker, 2013] which is its default and most commonly used scheduler. The CFS scheduler[2], first introduced into Linux in 2007, was designed with the following goals: ensuring fairness, maximizing performance and being a one-size-fits-all solution. Since its inception, CFS has undergone various changes to cope with user requirements (e.g., responsiveness, virtualization) and hardware evolution. As an illustration, the number of lines of code dedicated to the scheduler has risen twofold from 2008 to 2016 (from 9,163 to 18,424 lines

[1] In these architectures the main memory is split into several *NUMA nodes* with uneven access times from a given CPU.

[2] Even though it is not the only scheduler implemented in Linux, further mentions of "the scheduler" or "the Linux scheduler" in this document refer to CFS.

of code[3]). Some of these changes are either directly related to, or have an influence on thread placements; among them: scheduling domains, load tracking, cgroups and NUMA balancing. (i) *Scheduling domains* is the solution devised by the Linux developers to take on the problem of uneven memory affinities [Corbet, 2004]. (ii) The *load tracking system* which keeps track of the load of each CPU and thus defines the way threads have to be placed for the load to be balanced has been changed several times. (iii) *Cgroups*, which are active by default, change the unit of fairness from threads to group of threads. This change of unit directly affects the load tracking system and the notion of load balance. (iv) *NUMA balancing* moves threads between CPUs to try to fix the remaining issues with thread and memory co-location on NUMA architectures [Van Riel, 2014].

While the initial principle of CFS is simple and all these accommodations make sense on their own, problems appear when everything is combined. First, the documentation of the overall behavior of the scheduler is not on par with its complexity. Second, the decision process leading to thread placement choices has become opaque, riddled with obscure heuristics. Third, this complexity leads to bugs, as described by Lozi *et al.* [Lozi *et al.*, 2016]. These problems are further aggravated due to the development model of Linux. Indeed, the source code of the Linux kernel is complicated to understand and evolves at a high pace. Besides, Linux lacks a central documentation [Landley, 2008], and worse, the documentation that is available through different means (in the source code tree [Kernel, 2017], in Git commit messages, on dedicated websites [LWN, 2017; KernelNewbies, 2017], in books [Mauerer, 2008; Love, 2010]) is not always exhaustive nor up-to-date.

Because of these issues, the application developers and users have become somewhat skeptical regarding the efficiency of the Linux CFS scheduler. Thus, the recent years have seen an increased usage of the so-called *thread pinning* primitives (e.g., `taskset`, `sched_setaffinity`). These primitives let a developer/user manually enforce the mapping of threads to CPUs: in other words, one can dismiss any kind of thread placement decision made by the scheduler. Even though pinning is found to improve the performance of one application for a given architecture and workload, there is no guarantee that this will remain true for all architectures and workloads. Ultimately, pinning is just a static ad-hoc thread placement strategy – in the general case, it cannot replace an efficient, dynamic thread placement strategy implemented at the operating system level, which has a global view and full control on the machine.

The main contribution of the work presented in this document is the study of the general strategy and the different criteria used by Linux in its thread placement decisions. The results of this study is organized as a taxonomy of thread placement decisions. We also present our work-in-progress tool, which gathers runtime information about the actual thread placement decisions taken by CFS relying on our taxonomy to provide meaningful contextual insight.

[3]Measured using David A. Wheeler's 'SLOCCount' tool `https://www.dwheeler.com/sloccount/`.

This document is structured as follows. §2 highlights the empirical evidence driving our investigation of CFS. In §3 we discuss related work. §4 describes the main principles guiding the design of the Linux scheduler. Our taxonomy of thread migration decisions is presented in §5. The tool we developed is described in §6. §7 gives the first results of our investigation and discusses perspectives for future work. Finally, §8 conclude the paper.

2 Empirical evidence

In this section, we present the empirical evidence that motivates our work. §2.1 describes our testbed and §2.2 presents our results.

2.1 Testbed

The pinning problem (i.e., the fact that some applications perform better when their threads are placed in a simple static fashion in comparison to the placement of that results from the Linux scheduler decisions) is observed with real production setups and workloads. In order to simplify our investigation, we settled on simpler setups that exhibit the same problem. Our testbed is composed of 11 multi-threaded applications from well-known benchmarks (Splash2 [Woo *et al.*, 1995], Parsec [Bienia, 2011] and Phoenix test programs [Yoo *et al.*, 2009]) with execution wall-clock times ranging from $0.5s$ to $60s$. Each of these applications is run alone on a dedicated 64-core machine (a Dell PowerEdge R815 server with four 16-core AMD Bulldozer 6272 processors, and four 64GB NUMA memory banks) running a Linux 4.9 kernel. Each application is configured to use as many threads as CPUs. Given that our study is focused on maximizing performance (i.e., lowest wall clock execution time), we choose to disable CPU frequency scaling. Moreover, in order to simplify our setup and inverstigations, we chose to disable NUMA balancing , as this feature did not seem to have a significant impact on performance during our early experiments. Apart from these two options, all configuration options are left to their default and prescribed values (with notably the autogroup feature, which is enabled by default in mainstream Linux).

For our experiments, the pinning configuration relies on the `PinThreads` program [Lepers, 2017]. PinThreads'own thread placement strategy is based on the following two policies: (i) Each new thread of the application is assigned to a CPU in a round-robin fashion:

$$initial\_cpu\_of(i^{th}\ thread) = CPU_{i \text{ modulo } nb(CPU)}$$

(ii) The application threads are prevented from being migrated away from their initial CPU.

2.2 Empirical results

All the details of our experiments are available online [Ferreira, 2017]. *Figure 1* and *Table 1* in the appendix display the average execution time of the applications of our testbed each ran 30 times. The light green color displays this average when the threads are placed by PinThreads, the darker red shows the average when CFS chooses the placement of the threads. As illustrated in this figure, PinThreads improves the

execution time substantially (with a maximum improvement of 271% for volrend). . The standard deviation bars also highlight that in most cases, PinThreads reduces the variability of the execution time to almost zero.

3 Related work

Our work revolves around two aspects: thread placement issues and performance analysis. The state of the art regarding the issue of thread placement is discussed in §3.1. §3.2 gives details about the existing ecosystem of tooling for performance analysis regarding these issues.

3.1 Thread placement problem

The importance of thread placement on performance has been abundantly studied as illustrated by the existence of two recent surveys [Diener *et al.*, 2016; Zhuravlev *et al.*, 2012]. The survey of Diener *et al.* deals with the problem of thread and data placement in terms of memory affinity. It offers a broad overview and classification of the techniques devised to improve these placement problems, and encompasses techniques that operate at different levels, ranging from source code analysis to the operating system. In contrast, our work is focused on all kinds of thread placement problems, considering not only memory affinity but also load balancing. Another difference is that our study focuses on the role of the operating system scheduler in the definition of this placement because we believe it is the best suited to do so (given that it has a global view and full control of the machine). Zhuravlev *et al.* survey the scheduling techniques proposed to address thread placement issues not only according to memory affinity issues but also to memory contention issues. The survey covers both propositions from the research community and the solutions implemented in the Linux and Solaris schedulers. However, the study of the Linux scheduler targets an old version of the code (Linux 2.6.32 released on 2009-12-03) and is furthermore very brief. Unlike our study, it does not attempt to extensively describe the thread placement strategy of CFS and only provides a quick overview of load balancing.

The work of Lozi *et al.* [Lozi *et al.*, 2016] is the closest to ours. However, they differ in their goal and approach. The authors highlight that, due to bugs, the scheduler fails at one of its main objectives: being work-conserving (there must be no idle CPU if some thread is ready to run). To explain the root causes of the observed problems, they present a study of the recent version of CFS (Linux 4.3 released on 2015-09-02). However, unlike ours, their description is mostly focused on load imbalance concerns (due to scheduler bugs), while we aim at studying all kinds of performance problems related to thread placement including inefficient heuristics. For example, we consider initial thread placement decisions in addition to thread migrations and we also discuss the (cache affinity driven) placement criteria used upon thread wake-up.

3.2 Performance analysis and tooling

Our work is motivated by the will to understand why the thread placement strategy of the Linux CFS scheduler can degrade performance compared to manual/static thread placement approaches. To that end, Linux exports some statistics about its scheduler via a file interface (`/proc/schedstats`) [Fields *et al.*, 2017]. Among others, this file contains several counters about different kinds of thread placement decisions. We tried to rely on this tool to understand the performance issues in our case studies but found out that the provided statistics were insufficient for this purpose: the breakdown of thread placement decisions is very coarse grained and hides too many details regarding the heuristics used within the scheduler. Besides the documentation of the schedstats file assumes the reader already has a thorough understanding of the overall behavior of CFS and we have also found that this documentation is sometimes misleading.

Apart from schedstats, a plethora of tools exists for performance analysis/debugging purposes on Linux [Gregg, 2014]. Among them, the main and most generic tool is *perf* [Weaver, 2013]. perf offers a consistent way to both access hardware performance counters and to trace software events in the kernel (static, predefined events with tracepoints and dynamic events with kprobes). Among these static events, one of them is triggered by thread migrations but it lacks important contextual details: it provides the source and destination CPUs of the migration but offers no means to understand the circumstances and motivation that triggered the migration. Nonetheless, perf can be used to understand the general performance characteristics of an application on a machine (e.g., CPU hotspots, memory bottlenecks or synchronization events). For this purpose, perf can produce a trace of performance events associated with the call stacks of the corresponding thread upon the occurrence of the event. These traces can then be summarized and visualized using techniques such as *flame graphs* [Gregg, 2016]. This kind of tools is useful for our goal. We want to understand why the scheduler can degrade the performance of certain applications and, to do so, we also need the insight that those tools can give us regarding the application behavior to complement our understanding of the scheduler behavior.

Other tools are focused on the visualization of scheduling issues: the `perf sched` command of perf, the visualization tool created by Lozi *et al.* [Lozi *et al.*, 2016], ViTE [ViTE, 2010; Trahay *et al.*, 2011], Paraver [Pillet *et al.*, 1995] and KernelShark [Rostedt, 2017]. All these tools summarize scheduler events with a timeline visualization. The tool of Lozi *et al.* can display the number of threads assigned to each CPU or their loads while the other tools rely on a color chart to display which threads were running on each CPU. This way of presenting data facilitates the detection of some kinds of scheduling problems: for example, it is easy to notice if one CPU remains idle for too long. However, even though these tools can highlight some problems with the thread placement strategy, they are not sufficient on their own and require an additional study of the scheduler behavior because they do not give any insight into the scheduler decisions to migrate threads.

4 The Linux scheduler

The Linux CFS scheduler is a scheduler tailored for fairness and aiming at maximizing performance on a wide range of machines: from simpler machines with one or a few CPUs

up to machines with hundreds of CPUs. In order to support machines with such a high number of CPUs, CFS (like most schedulers implemented in recent operating systems) is based on a two-level design [Zhuravlev *et al.*, 2012]. The first level, at the scale of each individual CPU, takes care of time sharing the CPU between all the runnable threads placed on this CPU. The second level takes care of space sharing, i.e., placing the threads on the different CPUs, to ensure that all the CPU resources of the machine are used. This section gives some insights into the overall design of these two levels in §4.1 and §4.2. Then, some CFS details important for the understanding of the taxonomy proposed in §5, are described in §4.3, §4.4 and §4.5.

4.1 Time sharing

The first level works as a regular time-slicing scheduler. For each CPU, there is one *runqueue* that contains the list of threads assigned to this CPU that are ready to run. At this level, the responsibility of the scheduler is to choose: (i) when to switch from a thread to another, and, (ii) which thread from the runqueue should be run next.

The solution devised for the first question relies on well-known and documented techniques (e.g., preemption on interrupts, blocking system calls). In contrast, the solution used by CFS to address the second question is rather original: Linux schedules the thread that has spent the least time executing on a CPU. In doing so, CFS approximates an "ideal, precise multi-tasking CPU" [Molnar *et al.*, 2017], that is, a processor that can run any number of threads in a truly parallel fashion. Thanks to this scheme, CFS provides near-optimal fairness at the scale of each CPU.

4.2 Space sharing

The second level, space-sharing, takes care of thread placement (i.e., assigning threads to the CPU runqueues). This section gives an overview of this level and §5 provides more advanced details on this topic. Space sharing introduces two main duties for the scheduler, in order to keep the amount of work assigned to each CPU balanced: (i) the scheduler must decide on which CPU a thread will run when the thread is created; (ii) the scheduler must decide when, which and where threads should be moved [Volker, 2013; Lozi *et al.*, 2016]. Due to the dynamic nature of the workloads, imbalance between the runqueues may arise: threads might start, die or block at any time and thus some CPUs can become busier than others. For example, consider a scenario with two CPUs. The runqueue of CPU_1 contains four threads while the runqueue of CPU_2 contains only one. This would defeat the fairness objective of CFS: the last thread will have 4 times more CPU time than the other 4 threads. This imbalance can also be problematic for performance: suppose that the thread on CPU_2 is I/O-bound (i.e., it spends most of its time blocked, waiting for I/O events). Then, CPU_2 will spend most of its time idle, waiting for the I/O-bound thread to become ready to run again. In the meantime, CPU_1 will be busy slicing its time between its 4 threads. To address these issues, the space-sharing level of CFS includes a *load balancing* logic, which tries to keep the runqueues balanced and to maximize the utilization of all the CPUs.

4.3 Scheduling domains

Modern multicore architectures have varying memory access times due to cache hierarchies and NUMA effects. These caches and NUMA memory nodes are resources that are shared among groups of CPUs, as depicted in *Figure 2* in the appendix. As an example, CPU_1 and CPU_2 share the same first-level cache but CPU_3 uses another first-level cache. An efficient thread placement strategy has to be aware of these memory affinities when placing threads. With our example figure, if a thread were to be moved off from CPU_1, CPU_2 would be a better candidate to receive it than CPU_3 (assuming they are the two best targets to improve the runqueues balance), because on CPU_2 the thread would stay close to its first-level cache. Keeping a thread close to its caches and NUMA node tends to decrease memory access times and thus to improve the overall performance. This is the reason scheduling domains were introduced. §5 explain the role of these scheduling domains.

Structure

Scheduling domains model the actual topology in terms of memory affinity. From the point of view of one CPU, one scheduling domain stands for the group of all the other CPUs that share a topological property (e.g., direct access to the same CPU cache or NUMA memory node or all the CPUs are at the same NUMA distance from the NUMA node of the reference CPU). These scheduling domains are stored per-CPU (to minimize the overhead of concurrent accesses), as a hierarchy from the most memory affinity to the least. *Figure 3* in the appendix gives an example of the scheduling domains constructed for a machine with both cache and NUMA memory.

Construction

The construction of scheduling domains relies on the shared data structure `sched_domain_topology_level`. Except for NUMA levels, these levels are predefined in the source code. NUMA levels are defined dynamically using the matrix of memory node distances that is derived from hardware-provided information. On our test machine, each CPU has two NUMA scheduling domains: one for the CPUs of the nodes directly connected to this CPU node (one hop), and one for the CPUs of the nodes that have to communicate through another intermediary node (two hops). When the scheduler is initialized at boot time, for each CPU, one scheduling domain is then created from each of these levels.

4.4 Load metric

The work assigned to the CPUs is balanced according to a metric called *load* which aims at reflecting how busy a CPU is. In this section, we first give the intuition behind this metric computation. We then describe the two metrics that Linux actually uses. Finally, we explain the mechanisms that Linux relies on to be "conservative" regarding load balancing.

Intuition

A naive version of the load metric could simply use the number of threads of the CPU runqueue. However the number of threads does not account for threads that tend to block often. Consider the following example: a first CPU with two

CPU-bound threads and a second CPU with two I/O-bound threads. The two CPUs have the same number of threads and appear balanced. However, the first CPU is actually spending most of its time active while the second CPU is most often idle, waiting for blocked threads. In order to account for I/O-bound threads, the scheduler introduces a metric called CPU utilization, which it uses to compute the load metric. This *CPU utilization* of a thread is computed as the ratio between the actual time spent running and the time allotted for the thread. A thread that tends to block often will have a lower CPU utilization than a CPU-bound thread.

First metric

The first load metric is a sum of the averages of the CPU utilization of each thread. To compute these averages, Linux samples the CPU utilization over periods of $1024\mu s$. We denote u_i the sample of CPU utilization of the thread over the period i, with $i = 0$ being the earliest period, $i = \theta$ the period the thread was created and $i = c$ the current period. The initial value $threadlocal_\theta$ is initialized at a high arbitrary value. Then the load of a thread $threadload_t$ after the period t is computed as:

$$threadload_t = u_t + y \times threadload_{t-1}$$

with $0 < y < 1$ being a constant value chosen so that $y^{32} \approx 0.5$. An any point in time, the current value $threadload_c$ for this metric is:

$$threadload_c = \sum_{i=\theta}^{c} u_i \times y^{c-i}$$

In other words, the load of a thread is an average of its CPU utilization samples with the most recent samples having greater weights.

Then the overall load metric $load1_t$ of a CPU is expressed as the sums of the load of the threads of its runqueue.

Second metric

The second metric is an exponential smoothing of the first metric. The scheduler actually computes multiple versions (denoted by index j whose value ranges from 0 up to 4) of this second metric:

$$load2_0^j = load1_0$$
$$load2_t^j = \frac{1}{2^j} \times load1_t + \left(1 - \frac{1}{2^j}\right) \times load2_{t-1}^j$$

For example:

$$load2_c^1 = \frac{1}{2} \times load1_c + \frac{1}{2} \times load2_{c-1}^1$$
$$load2_c^4 = \frac{1}{16} \times load1_c + \frac{15}{16} \times load2_{c-1}^4$$

As illustrated in these examples, greater values of j give more weight to older values.

Conservative load balancing

The load balancing performed by CFS is described as being conservative : when the imbalance is too small, the scheduler favors keeping threads on their current CPUs to avoid thread migrations whose cost (increased memory access time due to the migrations moving threads away from their memory) would outweigh the benefit. This also avoids a ping-pong effect, where a thread could be moved back and forth in between two CPUs. This is implemented with two mechanisms.

The first mechanism relies on the two load metrics. When comparing the load of two CPUs (one being the current CPU of the task to be moved, the second being the considered destination), CFS will: (i) consider the first load value $load1$ (ii) consider the jth value of the $load2$ metric. The value of j is configured in the lowest scheduling domain that contains the two CPUs[4] (iii) either takes the minimum (for the current CPU) or the maximum (for the destination CPU) between these two load metrics. The two values reflect the same load metric but with varying importance given to the past. By taking the minimum value for the current CPU, CFS might underestimate the load. On the other hand, the maximum for the destination will tend to overestimate the destination load. This reduces the chances for the destination CPU to be considered less loaded than the current CPU and thus reduces the chance for load balancing migrations to happen.

The second mechanism relies on an imbalance threshold. When considering the imbalance, i.e., the difference of loads, the scheduler only considers it significant if:

$$load_{curr} - load_{dest} > p \times load_{curr}$$

Where p is computed based on the `imbalance_pct` configuration option of the enclosing scheduling domain:

$$p = (imbalance\_pct - 100)/2$$

The value of the imbalance usually ranges from 110 (for smaller scheduling domain) to 125 (for larger domain), thus p ranges from 5% to 12.5%.

4.5 Cgroup & autogroup

Control groups (abbreviated as "cgroups") are a pervasive feature of Linux which is not limited to the scheduler. At its core, the cgroups feature allows the user to group processes together and to apply various policies to all members of a cgroup. Regarding CFS, cgroups change the unit of fairness. When cgroups are in use, CFS tries to be fair to cgroups instead of being fair to threads. Consider the following example: the operating system is running four threads. $Thread_1$ is in a cgroup $cgroup_1$ of its own, the other three threads are in the same cgroup $cgroup_2$. For CFS to be fair in this setup, it will have to give the same amount of CPU time to $cgroup_1$ and $cgroup_2$ which means $thread_1$ will get to have three times more CPU time than the threads of $cgroup_2$. Similarly, the load metrics described earlier should account for these groups to ensure a load balance that provides fairness in terms of cgroups. To do so, CFS divides the load of each thread by the number of threads of the corresponding cgroups.

cgroups can be setup arbitrarily by the user, but they can also be setup automatically with the autogroup feature. In

[4] With the default configuration of Linux, higher scheduling domains will have higher value for j

order to understand the motivation behind the autogroup feature, it is important to understand why fairness at the scale of threads can be a problem: consider a simple setup with a mono-threaded process p_1 and a process p_2 with 8 threads. If the unit of fairness is the thread, p_1 will receive significantly less CPU time than p_2. Now assumes that these are run on a regular user desktop system and that p_1 is a GUI application while p_2 is a source code compilation. Because it is starved of CPU time, the GUI application might appear significantly less responsive than usual.

The autogroup feature is meant to fix this issue. When enabled, each new process is placed in the cgroup of its parent. When a process calls the function `setsid` (notably during the initialization of a terminal emulator process), it is removed from this cgroup and placed in its own new cgroup. With this mechanism, in our example, p_1 and p_2 will end up in different cgroups, and thus p_1 will not be starved by the large number of threads from p_2.

5 A taxonomy of thread placement decisions

Thread placement decisions are made at multiple occasions in Linux. These occasions can be grouped in three main categories: (i) per thread placement, for example when the thread is created or when it wakes up; (ii) global load balancing, which can move multiple threads to achieve load balance; (iii) NUMA balancing, when the scheduler detects too many remote memory accesses. As explained earlier, enabling or disabling NUMA balancing did not have any significant influence on the observed performance difference, thus thread placement decisions triggered by NUMA balancing are not covered in this document.

5.1 Per-thread placement

Multiple occasions can trigger per-thread placement decisions: (i) the thread just got created (e.g., process fork, pthread creation, kernel thread creation); (ii) the thread calls one of the system calls from the exec family; (iii) the thread is about to wake up. The scheduler applies the same general scheme for these three triggers, but thread placement on wake up might use a different scheme called *wake affine* if some conditions are met. The general scheme used in the three occasions is explained first. The wake affine case and how it is triggered is explained in a second time.

General scheme

When choosing the destination CPU for a thread, the scheduler considers its affinity with the previous CPU (in case of a thread creation, the previous CPU is the CPU of the thread that created it). The scheduler defines three flags, one for each occasion (`SD_BALANCE_FORK`, `SD_BALANCE_EXEC`, `SD_BALANCE_WAKE`) that can be set independently in the configuration of each scheduling domain. To choose the destination of the thread, the scheduler first finds the largest scheduling domain that contains the previous CPU and has its relevant flag set (`SD_BALANCE_FORK`, `SD_BALANCE_EXEC` or `SD_BALANCE_WAKE` depending on the occasion that triggered the decision). Once the scheduling domain has been chosen, the scheduler then selects the target CPU among this scheduling domain. It first looks for the idle CPU with the smallest wake up time[5]. If there is no idle CPU, the scheduler falls back to picking the least-loaded CPU of the scheduling domain. The main motivation of this general scheme is to keep the load balanced on each CPU to ensure fairness.

Wake affine

One occasion does not always follow the above-described scheme: threads about to wake up. Wake affine is a special case in the logic of the thread placement strategy of waking threads. The motivation is to try to account for cases of waker/wakee threads that have a certain degree of cache affinity (e.g., a scenario with 1 producer thread and 1 consumer thread). For the wake affine case to be considered the scheduling domain has to be configured with the `SD_WAKE_AFFINE` option. If this scheduling domain option is set, then in order to decide if it should aim for load balance or for cache affinity, the scheduler checks the following two criteria: (i) the waker does not wake many threads (the threshold is tailored to the capacity[6] of the LLC scheduling domain of the waker thread), and (ii) the remaining capacity of the waker CPU is enough to host the wakee (based on its CPU utilization). If one of these criteria does not hold, the scheduler falls back to the general scheme described earlier. If these two criteria are met, then the scheduler places the waking thread with cache affinity in mind. To achieve this, the scheduler first selects the least loaded CPU among the previous CPU and the waker CPU. It then places the thread on the least-loaded CPU in the last level cache scheduling domain of this selected CPU.

5.2 Global load balancing

Global load balancing is run by the scheduler to account for load imbalance among all the CPUs of the machine resulting from the evolution of the load. In order to do so, the load balancing algorithm moves threads from the busy CPUs to the least-loaded CPUs. This section describes: (i) how this load balancing is triggered, (ii) what is the general scheme of the algorithm, (iii) the considerations about cache affinity

Triggers

Two triggers exist for the global load balancing algorithm: periodically and newly idle. (i) Load balancing is triggered periodically for each scheduling domain, on the least-loaded CPU. The period value is a configuration option specific to each scheduling domain (with the default configuration, larger domains have a larger period value). (ii) Load balancing can be triggered by a CPU about to become idle when it tries to schedule a new task but its runqueue is empty. Because this idle load balancing is run in a critical section (with interrupts disabled), it is limited to only one thread migration.

[5] Some processor hardware supports multiple idle states for CPUs with varying power consumption and wake up time characteristics.

[6] The capacity is a metric used to represent the computational power of a scheduling domain. In the case of fully independent CPUs, the capacity is equal to the number of CPUs in the domain, in the case of shared microarchitectural resources (Simultaneous MultiThreading), the capacity is less than that.

General scheme
The scheduler attempts to balance the load inside scheduling domains starting from the smallest (in our example the first-level cache domain) up to the largest (the domain that spans all the CPUs). In doing so, the scheduler minimizes the number of thread migrations across largest domains, hence it keeps threads close to their memory as much as possible. The load balancing algorithm is designed in such a way that its main part, deciding which threads to move and where, is always executed by the least-loaded CPU of each scheduling domain. Once the least-loaded CPU has been elected to run load balancing on behalf of the scheduling domain, it then executes *Algorithm 1*.

```
begin
    dst ← current_cpu ;
    while load is still imbalanced
    and not all busy CPUs have been considered do
        src ← pick next busiest CPU inside the busiest
          scheduling domain ;
        while load is still imbalanced do
            thread ← select one thread from src ;
            if cancelled > cache_nice_tries then
                proceed with migration ;
            else if task has cache affinity then
                cancelled++ ;
            else
                proceed with migration ;
            end
        end
    end
end
```

Algorithm 1: Load balancing general scheme

Cache affinity
The scheduler considers the following cache affinity criteria. If the thread considered for the migration is the last thread that has run, then the migration is cancelled. Also, if the duration between the last time the thread has run and now is less than the `sched_migration_cost_ns` configuration value, then the thread is not migrated. In both cases, the motivation is that the first level of cache is likely to contain some of the data of this thread hence it may be better for performance to keep it on its current CPU.

6 Tool

In order to understand the actual behavior of Linux thread placement strategy and the reasons that can explain why it is sometimes worse than pinning we devised new tracing facilities. §6.1 describes their implementation, §6.2 and §6.3 presents the information that they can provide.

6.1 Implementation

Our tool is provided as a Linux kernel patch that implements two tracepoints. Tracepoints are a tracing infrastructure implemented directly in Linux. More precisely, tracepoints are static hooks placed in the source code, which are optimized so that they have almost no overhead if they are not in use. Typically, a tracing tool will register itself as a consumer of a tracepoint and will be called each time the tracepoint is reached. Additional information relative to the specific tracepoint context can also be provided to the consumer. As an example the default Linux kernel provides a tracepoint named `sched_switch` which is triggered just before a CPU change the process that is currently being run. It provides the current and the next thread as additional information.

We chose to rely on tracepoints despite the fact that they are static and requires the compilation of a custom kernel (in contrast to less intrusive techniques such as kprobes [Kenniston *et al.*, 2017]) for two reasons. First, the fact that they are static means that the kernel is able to optimize them in order to minimize their overhead. Second, we think that it would be a good idea to directly integrate these tracepoints in the mainstream kernel source code given the performance problem described in §2.

Several methods exist to monitor these tracepoints. The easiest and probably one of the most efficient is to use the perf tool to obtain a trace of all the occurrences of the tracepoints being reached in the code.

6.2 Thread placement

The first tracepoint implemented by our tool is triggered each time a thread is either placed for the first time or moved from one CPU to another CPU. This tracepoint `sched_thread_placement` offers the following additional informations: (i) the `name` and `PID` of the thread being placed (ii) the `origin CPU` (iii) the `destination CPU` (iv) the `executing CPU` (v) the `trigger` and motivation which lead to the thread placement decision (among those described in section 5) (vi) the minimum `level` in the scheduling domains hierarchy which contains both the origin and destination CPU.

This new tracepoint has some advantages over the existing `sched_migrate_task` tracepoint. First and foremost, it adds a new field `trigger`, which gives insight about the circumstances and motivations that lead to the reported migration. Thanks to this additional information, we were able to observe that, for most of the applications we have found to be sensitive to pinning, the two prevailing kinds of thread placement decisions were 1. load balancing triggered when a CPU becomes idle and 2. the wake affine thread placement strategy. Another advantage is that our tracepoint is not limited to thread migrations and also reports initial thread placement decisions. This is important given that we have found that about half of the applications of our testbed are sensitive to the initial placement of threads. The `level` field is also important, because it serves as a first indication of applications that could suffer from degradation of memory affinities.

6.3 Load update

It is important to be able to monitor the evolution of the load metrics because of the influence of the overall load balance on the performance of the system and the importance of these load metrics in the Linux thread placement strategy. To that end, we created a tracepoint `sched_load_update` that is triggered each time one of the load metrics is updated.

Thanks to this tracepoint, it is possible to get a trace with timestamps of all the successive values that these metrics take. With this trace, it is possible to create a temporal heatmap visualization of the evolution of the load of each CPU similar to the one proposed by Lozi *et al.* but with more insight. More precisely, this tracepoint can be used in conjunction with the `sched_thread_placement` tracepoint to obtain one unified trace of these two tracepoints. With this unified trace, visual hints that would depict the thread placement decisions could be added to the visualization with colors used to represent what triggered the migrations. With such a visualization, it would be much easier to understand how the migrations affect the load of the runqueues and pinpoint occurrences of migrations which were harmful for the overall load balance. For example, with the wake affine strategy, whose first concern is not load balance, we could discover that, for some applications, this strategy is too aggressive.

7 Results and perspectives

The first thing that we discovered is that the autogroup feature has the most influence on the performance difference. When it is disabled, the performance difference drops significantly (see *Table 2* in the appendix). At this point, we are still unsure of the reasons that lead to this influence, but the fact that this feature only interferes with the load metric calculation indicates that the problem likely resides in this calculation. One of the bugs described by Lozi *et al.* is related to this feature and, as of today, has not been addressed in mainstream Linux. Due to lack of time and technical issues we have not yet ported and tested the tools and fix provided by Lozi *et al.* and cannot assess if the cause behind this influence of the autogroup feature is either partly or entirely explained by this bug. Our second finding is that the initial thread placement that results from the default Linux strategy is quite different from the strategy implemented by PinThreads. It seems that Linux fails to effectively spread the newly created threads on the available CPUs as it intends to. Our first investigation in this direction hints at a problem in the initialization of the load metric of newly created threads which ends up underestimating the load contribution of these new threads. Ultimately, this results in an initial thread placement with a few overpopulated CPUs and many underused CPUs. Because this affects the initial placement of threads, it is important to try to evaluate if this initial placement has influence on the overall execution or if conversely, its influence is limited to the start of the application and thus a longer execution time of the same application would have mitigated this influence. We plan to update our testbed with varying execution times in order to study this aspect.

The underlying issues behind the observed inefficiencies remain unclear. In order to understand the causes of these issues, we plan to continue the development of our tool. One issue we will probably have to face is the computation of the load metric. We have already shown that this computation is complicated, but from our investigation of its numerous changes during the past years of Linux development , we can tell that it is a major source of issues (which is confirmed by the fact that the two findings described above are probably problems related to this computation) and that it is most likely to change again in the near future. While being able to monitor these load metrics remain important, our tool should not rely solely on them to provide data about the load balance of the system because if the load metrics are inaccurate, the tool will also provide inaccurate and potentially misleading data. The traces generated using our tracepoint are a good start but the large amount of data they contain makes them hard to use on their own. One possible solution to address this could be to devise some meaningful metrics that aggregate these data. For example, we could compute at any point in time the instantaneous balance of the system by computing the standard deviation of the load metrics of all the CPUs. We could then derive the overall balance of an application execution by computing a temporal mean of these instantaneous balance metrics. Another solution could be to leverage these traces to produce a visualization that would simplify their analysis. We already discussed one example of this with the temporal heatmap visualization but many other kinds of visualization exists and could be interesting [Heer *et al.*, 2010].

We have not fully evaluated the overhead of our tool yet. Our first experiments indicates that the execution time overhead is relatively low for the `sched_thread_placement` tracepoint (about +17% for an application with a large number of migrations) but high for the `sched_load_update` tracepoint (almost +100%). More importantly, we plan to devise a method to assess that the tool does not significantly alter the behavior of the application (notably in terms of migrations, but also by checking the number of context switches and the memory usage) which would make the trace irrelevant.

8 Conclusion

The long-term goal of our work is to explain some of the inefficiencies of CFS related to thread placement issues. As we found out, explaining these inefficiencies was difficult because of two main shortcomings : the absence of an up-to-date comprehensive documentation and the lack of proper tooling. We presented our attempt to address this two shortcomings.

First, we proposed our own overall documentation of Linux thread placement strategy. In order to create this documentation we relied on the existing but scattered sources of documentation and checked which parts were correct and up-to-date with the latest versions of Linux. We also thoroughly studied the Linux source code to fill in the gaps and get a better understanding of the overall design of this strategy.

Then, we described our first steps towards the implementation of a tracing tool to address the second problem. We then highlighted the first results we obtained thanks to this tool and finally discussed perspectives for further improvements.

References

[Bienia, 2011] Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, jan 2011.

[Corbet, 2004] Jonathan Corbet. Scheduling domains, 2004. *URL:* https://lwn.net/Articles/80911 (*accessed* 2017-06-09).

[Diener *et al.*, 2016] Matthias Diener, Eduardo H. M. Cruz, Marco A. Z. Alves, Philippe O. A. Navaux, and Israel Koren. Affinity-Based Thread and Data Mapping in Shared Memory Systems. *ACM Comput. Surv.*, 49(4):64:1–64:38, 2016.

[Ferreira, 2017] Christopher Ferreira. Magistère results, 2017. *URL:* https://gitlab.com/aumgn/erods-bench/tree/results/exp/results/06-MIG (*accessed* 2017-08-20).

[Fields *et al.*, 2017] J. Bruce Fields, Rakib Mullick, and Linux Kernel Contributors. schedstats - Kernel Documentation, 2017. *URL:* https://www.kernel.org/doc/Documentation/scheduler/sched-stats.txt (*accessed* 2017-06-10).

[Gregg, 2014] Brendan Gregg. Linux Performance Analysis: New Tools and Old Secrets. In *LISA'14*, 2014.

[Gregg, 2016] Brendan Gregg. The flame graph. *Communications of the ACM*, 59(6):48–57, 2016.

[Heer *et al.*, 2010] Jeffrey Heer, Michael Bostock, and Vadim Ogievetsky. A tour through the visualization zoo. *Commun. ACM*, 53(6):59–67, June 2010.

[Keniston *et al.*, 2017] Jim Keniston, Prasanna Panchamukhi, and Masami Hiramatsu. Kprobes - Kernel Documentation, 2017. *URL:* https://www.kernel.org/doc/Documentation/kprobes.txt (*accessed* 2017-08-17).

[Kernel, 2017] Kernel. Kernel documentation, 2017. *URL:* https://www.kernel.org/doc/Documentation (*accessed* 2017-08-01).

[KernelNewbies, 2017] KernelNewbies. Linux Kernel Newbies, 2017. *URL:* https://kernelnewbies.org/ (*accessed* 2017-08-01).

[Landley, 2008] Rob Landley. Where Linux Kernel Documentation Hides. In *Proceedings of the Linux Symposium*, pages 7–18, Ottawa, 2008.

[Lepers, 2017] Baptiste Lepers. PinThreads - Github repository, 2017. *URL:* https://github.com/BLepers/PinThreads (*accessed* 2017-08-20).

[Love, 2010] Robert Love. *Linux Kernel Development*. Addison-Wesley Professional, 3rd edition, 2010.

[Lozi *et al.*, 2016] Jean-Pierre Lozi, Baptiste Lepers, Justin Funston, Fabien Gaud, Vivien Quéma, and Alexandra Fedorova. The Linux Scheduler: A Decade of Wasted Cores. In *Proceedings of the Eleventh European Conference on Computer Systems*, London, United Kingdom, 2016. ACM.

[LWN, 2017] LWN. Linux weekly News, 2017. *URL:* https://lwn.net/doc/Documentation (*accessed* 2017-08-01).

[Mauerer, 2008] Wolfgang Mauerer. *Professional Linux Kernel Architecture*. Wrox Press Ltd., Birmingham, UK, 2008.

[Molnar *et al.*, 2017] Ingo Molnar, Claudio Scordino, and Linux Kernel Contributors. CFS Scheduler - Kernel Documentation, 2017. *URL:* https://www.kernel.org/doc/Documentation/scheduler/sched-design-CFS.txt (*accessed* 2017-06-08).

[Pillet *et al.*, 1995] Vincent Pillet, Jesús Labarta, Toni Cortes, and Sergi Girona. PARAVER: A Tool to Visualize and Analyze Parallel Code. *Proceedings of WoTUG-18: Transputer and occam Developments*, 1995.

[Rostedt, 2017] Steven Rostedt. KernelShark, 2017. *URL:* http://rostedt.homelinux.com/kernelshark/ (*accessed* 2017-06-08).

[Trahay *et al.*, 2011] François Trahay, François Rue, Mathieu Faverge, Yutaka Ishikawa, Raymond Namyst, and Jack Dongarra. EZTrace: a generic framework for performance analysis. *International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 618–619, 2011.

[Van Riel, 2014] Rik Van Riel. Automatic NUMA Balancing, 2014. *URL:* https://events.linuxfoundation.org/sites/events/files/slides/kvmplumbers2014_riel_automatic_numa_balancing_1.pdf (*accessed* 2017-05-06).

[ViTE, 2010] ViTE. ViTE - Visual Trace Explorer, 2010. *URL:* http://vite.gforge.inria.fr/ (*accessed* 2017-06-08).

[Volker, 2013] Seeker Volker. Process Scheduling in Linux, 2013. *URL:* http://www.volkerseeker.com/docs/projects/linux_scheduler_v3_1.pdf (*accessed* 2017-05-06).

[Weaver, 2013] Vincent M. Weaver. Linux perf_event Features and Overhead. *The 2nd hInternational Workshop on Performance Analysis of Workload Optimized Systems, FastPath*, 2013.

[Woo *et al.*, 1995] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22Nd Annual International Symposium on Computer Architecture*, ISCA '95, pages 24–36, New York, NY, USA, 1995. ACM.

[Yoo *et al.*, 2009] Richard M Yoo, Anthony Romano, and Christos Kozyrakis. Phoenix Rebirth: Scalable MapReduce on a Large-scale Shared-memory System. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, IISWC '09, pages 198–207, Washington, DC, USA, 2009. IEEE Computer Society.

[Zhuravlev *et al.*, 2012] Sergey Zhuravlev, Juan Carlos Saez, Sergey Blagodurov, Alexandra Fedorova, and Manuel Prieto. Survey of scheduling techniques for addressing shared resources in multicore processors. *ACM Computing Surveys*, 45(1):1–28, 2012.

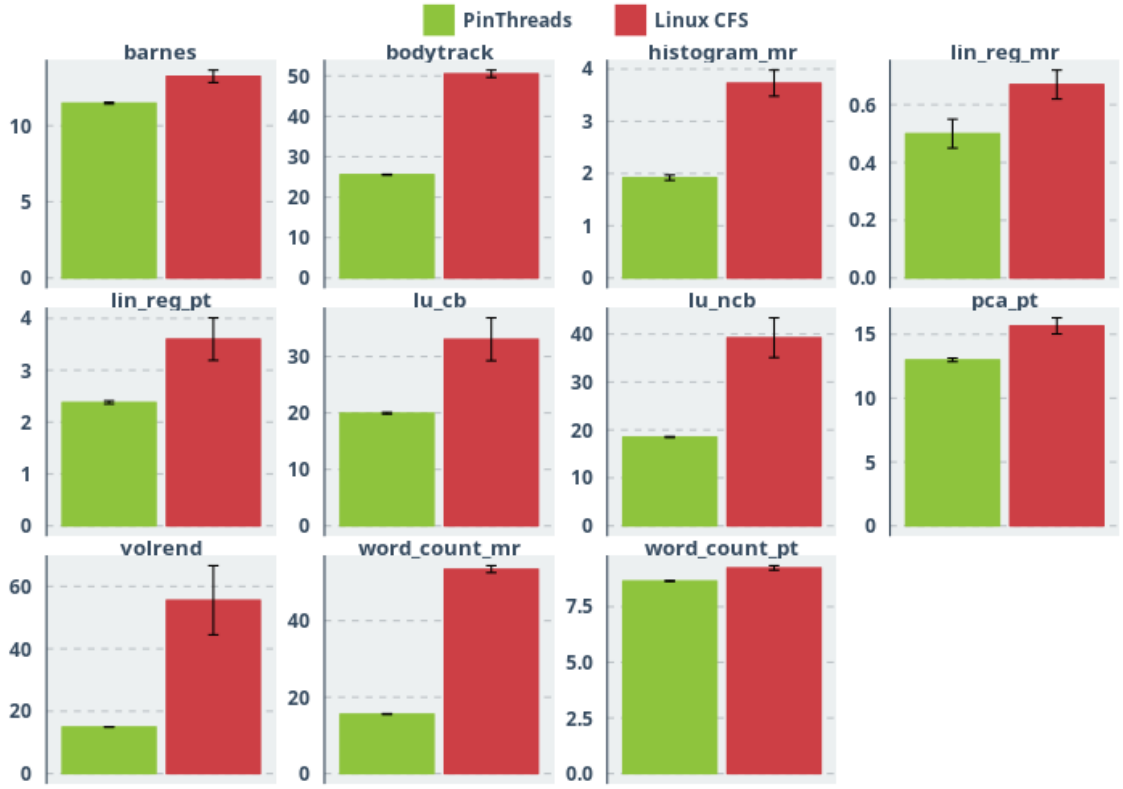


Figure 1: Average wall-clock time in seconds of the applications (lower is better). The bars depict the standard deviations.

	PinThreads		Linux CFS		Difference
word_count_pt	8.65 s	± 0.02	9.24 s	± 0.10	+ 6.82 %
barnes	11.48 s	± 0.06	13.24 s	± 0.41	+ 15.33 %
pca_pt	12.99 s	± 0.14	15.65 s	± 0.63	+ 20.48 %
lin_reg_mr	0.50 s	± 0.05	0.67 s	± 0.05	+ 34.00 %
lin_reg_pt	2.38 s	± 0.03	3.60 s	± 0.41	+ 51.26 %
lu_cb	19.95 s	± 0.15	33.04 s	± 3.79	+ 65.61 %
histogram_mr	1.92 s	± 0.05	3.73 s	± 0.25	+ 94.27 %
bodytrack	25.57 s	± 0.09	50.54 s	± 0.91	+ 97.65 %
lu_ncb	18.52 s	± 0.11	39.24 s	± 4.14	+ 111.88 %
word_count_mr	15.57 s	± 0.10	53.48 s	± 0.92	+ 243.48 %
volrend	14.95 s	± 0.06	55.57 s	± 11.07	+ 271.71 %

Table 1: Average wall-clock time in seconds and standard deviations of the applications for both the PinThreads and Linux CFS configuration. The last column is computed as : $\frac{Average_{LinuxCFS} - Average_{PinThreads}}{Average_{PinThreads}} \times 100$

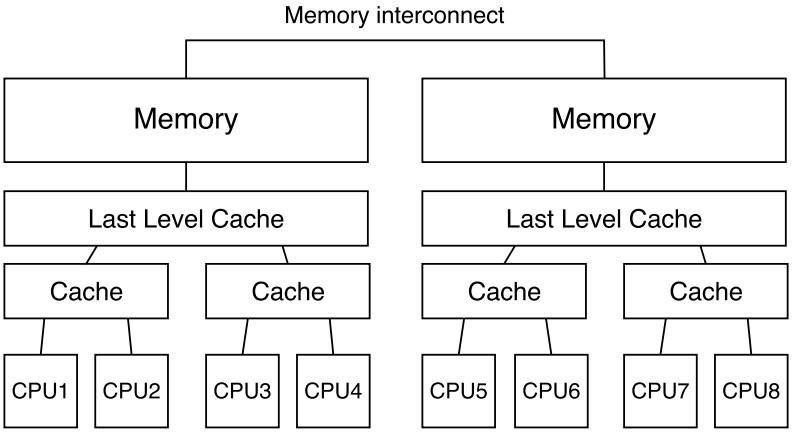


Figure 2: A simplified architecture with two cache levels and two NUMA nodes

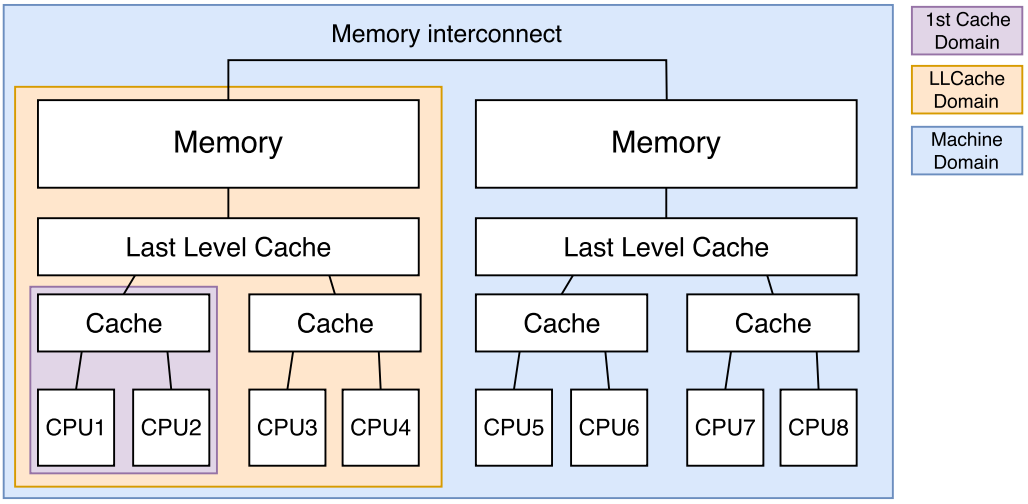


Figure 3: The scheduling domains derived from the architecture (from the point of view of CPU1)

	With Autogroup		Without Autogroup		Difference
lin_reg_pt	3.60 s	± 0.41	3.94 s	± 0.33	- 8.63 %
word_count_pt	9.24 s	± 0.10	8.66 s	± 0.08	+ 6.70 %
barnes	13.24 s	± 0.41	11.88 s	± 0.28	+ 11.45 %
pca_pt	15.65 s	± 0.63	13.95 s	± 0.82	+ 12.19 %
lin_reg_mr	0.67 s	± 0.05	0.57 s	± 0.02	+ 17.54 %
lu_cb	33.04 s	± 3.79	20.50 s	± 0.62	+ 61.17 %
histogram_mr	3.73 s	± 0.25	2.28 s	± 0.07	+ 63.60 %
lu_ncb	39.24 s	± 4.14	23.92 s	± 0.78	+ 64.05 %
bodytrack	50.54 s	± 0.91	23.57 s	± 0.20	+ 114.43 %
word_count_mr	53.48 s	± 0.92	23.01 s	± 0.36	+ 132.42 %
volrend	55.57 s	±11.07	15.92 s	± 0.33	+ 249.06 %

Table 2: Average wall-clock time in seconds and standard deviations of the applications with and without the Autogroup feature. The last column is computed as : $\frac{Average_{With\ Autogroup} - Average_{Without\ Autogroup}}{Average_{Without\ Autogroup}} \times 100$

Scalable Image Reconstruction Methods for Large Data: Application to Synchrotron CT of Biological Samples

Claude Goubet

Supervised by: Juan F. P. J. Abascal, Max Langer, Françoise Peyrin

I understand what plagiarism entails and I declare that this report is my own, original work.
Name, date and signature: Claude Goubet, 22/08/17

Abstract

Synchrotron radiation computational tomography allows to perform 3D medical imaging with high spatial resolution in the nano scale. Although this imaging technique raises many challenges as huge amount of data is required which can affect the sample during the acquisition and increase the acquisition time. This research proposes to use a compressed sensing reconstruction using split Bregman in order to reduce the number of projections required to reach high resolution. Quantitative performances of this approach on an ex-vivo bone sample Synchrotron imaging are presented, expressing the efficiency of the proposed reconstruction in terms of data reduction and reconstruction time.

1 Introduction

Understanding the origins and development of some diseases is still challenging today. It is the case for osteoporosis. In order to understand and identify the causes of this illness, studies on the bone structure must be done in the nano scale. Nano synchrotron radiation computational tomography can provide the necessary data. Research for higher resolution imaging techniques is hence ongoing.
The project in which this research is involved focuses on the development of a microscopic imaging technique for 3D representations of bones samples. In order to reach higher resolution reconstructed images, Synchrotron Radiation (SR) phase contrast Computational Tomography (CT) imaging is a good candidate as it allows to observe the phase shift induced by the refractive index of the matter, which is in the order of magnitude three times more sensitive than commonly used absorption in standard CT. Hence, More information can be acquired to optimize the reconstruction resolution.

The objective is to reconstruct images with a spatial resolution close to the current theoretical limit of $10nm$. However, for hardware as well as software issues this resolution is not yet possible. The European Synchrotron Radiation Facility (ESRF) upgraded it's nano beam-line in 2016 to a state-of-the-art technology and is currently able to reach a spatial resolution of $60nm$ [Martinez-Criado *et al.*, 2016].
Reaching higher resolution requires to acquire huge amount of data with an acquisition time reaching multiple hours, during which the sample will be subjected to high radiation doses as high as $8 \times 10^7 Gy$ [Langer *et al.*, 2012]. In the context of CT acquisitions, the data is composed of volume projection acquired using the ESRF nano-beam-line. 4×3000 projections of the volume are necessary to have a reconstruction of decent quality. This reconstruction is performed using the Filtered Back Projection (FBP) transforming data in the projection space to the image space. Reaching this spatial resolution is particularly interesting for studies requiring to understand the bone structure such as osteoporosis study. Indeed, we do not know yet how this disease progresses and where it comes from. Studying nano scale images of bones at different stages of the disease would allow to understand how the bone structure evolves.

Acquiring images at the ESRF allows to reach a high definition reconstruction but is invasive to the sample. Indeed, the exposition to high radiation for multiple hours has an effect on the composition of the sample. After few projections the sample can be subjected to motion, shrink, and change of colors. At this level of resolution each little change on the sample limits the quality of the reconstruction. It is then important to reduce the radiation dose necessary for the acquisitions. Another problem is the acquisition time. Access to the ESRF is limited in time, as there is a high demand for using this facility. The number of experiments possible to do is then limited by the acquisition time. Along with reducing radiation dose, reducing the acquisition time is also necessary.
These two objectives lead to two axes of research in SR CT imaging, *phase retrieval*, affecting the exposition time during the acquisition, and *tomography*, affecting the amount of data required for the reconstruction. Phase retrieval will not be discussed here as the data used for the experiments were provided using the method described by Langer in [Langer, 2008], with an exposition time already close to the theoretical limit. Although this research will focus on the

tomographic part, with an objective of designing a scalable algorithm allowing to reconstruct high resolution images of good quality.
In micro SR CT, Compressed Sensing (CS) [Candes *et al.*, 2006] has been used to reduce the number of projections needed for the image reconstruction [Gaass *et al.*, 2013; Li and Luo, 2011; Melli *et al.*, 2016a; 2016b; Yang *et al.*, 2015; Zhao *et al.*, 2012]. Among these researches, it is worth to underline iterative algorithms using Total variation (TV) minimization. These algorithms allowed to use only 40% of the data and still preserve the higher contrast edges in the images, resulting in a visually blurry or noisy reconstructed image [Melli *et al.*, 2016a; 2016b]. Research for CS on nano SR CT has mostly been assessed in the phase retrieval [Liu *et al.*, 2013]. To our knowledge, no research assessed CS algorithms in nano-CT reconstruction on bone samples. Moreover, neither micro nor nano-CT solutions have assessed their scalability.

There are two main contributions in this project: 1) contribution on the low dose reconstruction, 2) contribution on the scalability assessment on the proposed algorithm.
We will propose in this paper to approach the CS reconstruction problem using the minimization of the total variation. To address to this problem we aim at using the Split-Bregman (SB) iterative algorithm [Bregman, 1967], allowing an optimal approximation of TV minimization [Burger and Osher, 2013].
This algorithm has been used for multiple medical imaging techniques [Abascal *et al.*, 2011; 2016; **?**] and we wish here to apply it to SR nano CT. We will call along this thesis the proposed algorithm SB-TV-2D for Split-Bregman with 2D Total Variation minimization.
We assessed the feasibility of low dose SR nano CT by reducing the number of projections. We measured the effect of data reduction on the image reconstruction quality on bone images.

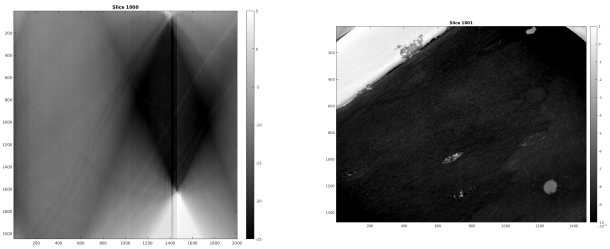
We target a scalable solution by using the PyHST2 functions. Moreover, we foreseen a higher level of scalability by splitting the bone sample into sub-volumes that will each be reconstructed in parallel on different nodes of ESRF computational cluster. With this approach, we aim to reach a reconstruction time of less than 24 hours.

2 Material and methods

2.1 Data acquisition

The dataset used in the experiments are composed of projections sinograms of a bone sample acquired in the beam-line ID19 of the ESRF. The ID19 beam-line is devoted to 3D imaging in the micro scale. With an energy between 10 and 250 KeV, the beam size ranges between 0.1×0.1 and 60×15 mm^2.
The sinograms used are the result of a post-processing of phase retrieval described in [Langer, 2008]. Indeed, in order to perform phase contrast imaging, multiple acquisition of the sample must be done at different distances, and a post-processing allows to perform phase retrieval and build phase maps.
The retrieved phase maps of each projection constitute our data, which corresponds to the Radon Transform of the refractive index that we wish to reconstruct.
The full sample is composed of 2000 projections of size 2048×2048 with a vertical and horizontal pixel size of 0.12 microns. The acquisition was performed in the range of 180 degrees with an angle between projections of 0.09 degree and an energy of $33.6KeV$. The rotation axis is set at pixel 1043.5×1043.5. The memory size of this dataset is in the order of $60GB$.

The sinogram data of one slice is represented in sub-figure 1a and the slice reconstructed with this data, of size 2048×2048 is represented in sub-figure 1b. The fully reconstructed 3D dataset is of size $2048 \times 2048 \times 2048$.



(a) One slice sinogram (b) One slice reconstruction

Figure 1: One slice of the 3D Micro SR-CT data used for the experiment

2.2 Compressed sensing formulation

Compressed sensing is a signal processing tool allowing to efficiently reconstruct a signal. It was first introduced in [Candes *et al.*, 2006] in 2006 and ever since has revolutionized image acquisitions. Thanks to compressive sampling images can be reconstructed from fewer data.
This method can be used under two conditions: the *sparsity* of the image under some transformation and *incoherence*. The image is then reconstructed as follows:

$$\min_f ||Wf||_1 \text{ such that } Ff = p \tag{1}$$

where f is the image, W is a transformation of f into a domain in which it is sparse. The parameter p is the acquired compressed data and F the transform from acquisition space to image space. The l_1 minimization $||Wf||_1$ promotes the sparsity and the constraint $Ff = p$ enforces the data consistency. In CT, the use of CS in $||Wf||_n$ with $n \leq 1$ allows to preserve borders, as $||Wf||_2^2$ cannot preserve borders [Burger and Osher, 2013].
As first proposed in [Rudin *et al.*, 1992], we defined $W = \nabla$ which lead to compute the total variation (TV):

$$TV = ||Wf||_1 = ||\nabla f||_1 = \sum_{i=1}^{N} |\nabla f_i| \tag{2}$$

where N is the number of voxels in the image.
The incoherence assumption holds as previously shown

[Candes *et al.*, 2006], and an image is generally sparse in the gradient domain. The two requirements are hence matched. Although, (1) is a constrained problem and is hence not straight forward to solve.

2.3 Split-Bregman iterative reconstruction

Scalable algorithms were presented by Presquet et al. for compress sensing solving using Split-Bregman iterations in [Combettes and Pesquet, 2011]. This algorithm was used by Abascal et al. in micro CT [Abascal *et al.*, 2016]. We wish to describe here the split Bregman algorithm [Goldstein and Osher, 2009] which gives a solution to an L1 constrained problem in an efficient way and will be used in our algorithm to minimize the L1 norm.

Split Bregman iteration

We wish to solve the constrained reconstruction optimization problem described in the section 2.2

$$\min_f ||\nabla f||_1 \text{ such that } Ff = p \tag{3}$$

The constrained problem expressed by the equation (3) is difficult to solve directly so it is usually approximated by an unconstrained problem as:

$$\min_f ||\nabla f||_1 + \frac{\mu}{2}||Ff - p||_2^2 \tag{4}$$

The Bregman iteration allows us to solve 3 as a sequence of unconstrained problems. These constrained problem can be resolved iteratively as follows:

$$\begin{aligned} f^{k+1} &= \min_f ||\nabla f||_1 + \frac{\mu}{2}||Ff - p^k||_2^2 \\ p^{k+1} &= p^k + p - Ff^{k+1} \end{aligned} \tag{5}$$

L1 regularization problem

Our compressed sensing reconstruction method is based on minimizing the total variation. It is hence difficult to minimize using standard techniques as TV is not differentiable. A splitting technique can be formulated to solve this issue, and we will see how this can be done iteratively with split-Bregman.
The idea is to "de-couple" the L1 and L2 parts of our original problem. We wish to minimize the Total Variation $||\nabla f||_1$ of the image and a weight function $H()$. Splitting can be done by introducing new variable and defining a new constraint:

$$\min_{f,d} ||d||_1 + H(f) \text{ such that } d = \nabla f \tag{6}$$

Using Bregman iteration we approximate the constraint iterativly:

$$\begin{aligned} (f^{k+1}, d^{k+1}) &= \min_{f,d} ||d||_1 + H(f) + \frac{\lambda}{2}||d - \nabla f - b^k||_2^2 \\ b^{k+1} &= b^k + \nabla f^{k+1} - d^{k+1} \end{aligned} \tag{7}$$

SB-TV-2D reconstruction

The isotropic TV reconstruction problem is given as follows:

$$\min_f ||\nabla f||_1 \text{ such that } ||Ff - p||_2^2 < \sigma^2 \tag{8}$$

where $\nabla f = (\nabla_x, \nabla_y) f$, p represents the projection space, F the projection operator, f the image domain and σ represents the variance of the signal noise.

$$\begin{aligned} f^{k+1} &= \min_f ||\nabla f||_1 + \frac{\mu}{2}||Ff - p^k||_2^2 \\ p^{k+1} &= p^k + p - Ff^{k+1} \end{aligned} \tag{9}$$

We fall here into an unconstrained problem which is not straight forward to solve. In order to get a constrained problem we will insert a variable such that $d = \nabla f$.
We can now use the Split Bregman iteration in order to solve our new problem:

$$\begin{aligned} f^{k+1} &= \min_{f,d} ||d||_1 + \frac{\lambda}{2}||Ff - p^k||_2^2 \text{ such that } d = \nabla f \\ p^{k+1} &= p^k + p - Ff^k \end{aligned} \tag{10}$$

And get to a solution where L1 and L2 elements of our original problem are split into two equations:

$$\begin{aligned} f^{k+1} &= \min_f \frac{\mu}{2}||Ff - p^k||_2^2 + \frac{\lambda}{2}||d^k - \nabla f - b^k||_2^2 \\ d^{k+1} &= \min_f ||d||_1 + \frac{\lambda}{2}||d - \nabla f - b^k||_2^2 \\ b^{k+1} &= b^k + \nabla f^{k+1} - d^{k+1} \\ p^{k+1} &= p^k + p - Ff^k \end{aligned} \tag{11}$$

Now it is left to solve the minimization on the f^{k+1} and d^{k+1} operations.

Solution for f

The solution for f^{k+1} is given by a quadratic problem. We can hence get the minimum by differenciating and equating to zero. We then get:

$$\begin{aligned} (\mu F^T F + \lambda \nabla^T \nabla) f^{k+1} &= \mu F^T p^k + \lambda \nabla^T (d_x^k - b_x^k) \\ K f^{k+1} &= rhs^k \end{aligned} \tag{12}$$

This can be efficiently solved by using a Krylov linear solver [Hestenes and Stiefel, 1952; Paige and Saunders, 1975].

Solution for d

The expression of d^{k+1} in 11 is given analytically. Hence the solution will be computed thanks to the shrinkage thresholding function:

$$shrink(x, \gamma) = \frac{x}{|x|} \times max(|x| - \gamma, 0) \tag{13}$$

so that:

$$d^{k+1} = shrink(\nabla f^{k+1} + b^k, \alpha/\lambda) \tag{14}$$

where α/λ represents the threshold parameter.

```
for k = 1:nbIterations
    % update f_k
    rhs_k= mu*FT(p_k)+lambda*Dt(d-b);

    f_k  = krylov(rhs_k = Kf); % solves (4.10)

    d    = D(f_k);  % gradient of f_k

    % update x and y
    d_k  = shrink(d+b_k, alpha/lambda);

    % update bregman parameters
    b_k  = b_k+d-d_k;
    pForw= F(f_k);
    p_k  = p_k + p0-pForw;
end
```

Figure 2: Pseudo code in Matlab: Iteration of the SB-TV-2D

Algorithm
The implementation of FB-TV-2D iteration in Matlab is presented in Figure 2.

2.4 Scalability

The second objective of this algorithm is scalability. This scalability will be assessed using tools provided by the ESRF. A software for tomographic reconstruction was developed by the ESRF and a Cluster is also available. We will see along this section how to take advantage of these tools.
The objective is to integrate SB-TV-2D to the ESRF reconstruction methods accessible by using the software PyHST2 and to make it scalable using parallelization on the ESRF cluster. We will first describe the PyHST2 software and then present the ESRF cluster capacities.

2.5 Evaluation metrics

Assessment of image quality
The evaluation of the performances was done by executing different scenarios of reconstruction with different number of iterations. We defined three metrics to evaluate and compare the reconstructed images. The aim of these metrics is to compare each reconstructed image to a target. The targets will be described section 3.1. The first metric is the Root Mean Squared Error (RMSE) defined as

$$RMSE = \frac{||f - \hat{f}||_2}{||\hat{f}||_2} \tag{15}$$

Where f is the target image and $\hat{f}$ is the reconstructed image. It is used to evaluate the global error of the image, and gives an idea of the image noise. But this metric is not sufficient. Indeed, the SB-TV-2D reconstruction method tends to create patches in the image, these patches are not detected by RMSE. Two other methods were then used, Strict Artifact Measure (SAM) and Peak Signal To Noise Ratio (PSNR). The SAM is defined as

$$SAM = TV(f, \hat{f}) = ||\nabla(f - \hat{f})||_2 \tag{16}$$

The PSNR is defined as

$$PSNR = 10 \log_{10} \frac{L^2}{MSE} \tag{17}$$

Where L is the range of the values of the image pixels and MSE is defined as follows:

$$MSE = \frac{1}{N} \sum_{i=0}^{N-1} \sum_{j=0}^{M-1} \Big[f(i,j) - \hat{f}(i,j) \Big] \tag{18}$$

For RMSE and SAM, lower values mean better quality, when for PSNR, low values mean poor reconstruction quality.

Assessment of the scalability
In order to assess the scalability of the algorithm the computation time for a full data reconstruction will be estimated. The scalability of the reconstruction is already provided by the integration of PyHST2 scalable reconstruction calls. But here we wish to make sure that the reconstruction of a bone sample can be completed within 24 hours.
To do so, we will estimate the number of PyHST2 calls necessary for an iteration, make different scenarios of parallelization and estimate how long each iteration would take. The metric used for the evaluation will then be the execution time with an objective of reconstruction within 24 hours.

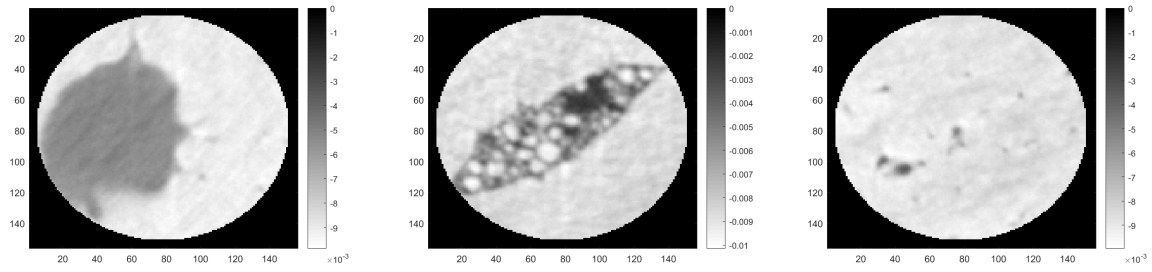
3 Experimental results and discussion

In order to answer our problematic expressed earlier, we made two hypothesis:
1) It is possible to use compressed sensing in order to preserve bone image quality with less data.
2) We can assume that we can make an algorithm which can be implemented at the ESRF so that using the ESRF software PyHST2, this algorithm can be scalable and executed within 24 hours.

3.1 Image reconstruction quality assessment

Experiment setting
The experiments were performed in Matlab. We chose to use Matlab for the implementation of our experiment as the ESRF implementation has to be done in Octave in order to be submitted to the Cluster. Running the reconstruction of a $2048 \times 2048 \times 2048$ volume on Matlab is not feasible in practice. In order to assess the algorithm in simulated data we had to work with smaller images. We then decided to extract three targets from our data as displayed in Figure 3.
The first target (sub-figure 3a) represents an osteocyte lacuna, the second target (sub-figure 3b) represents an osteocyte lacuna with calcium balls (healing process and the third target (sub-figure 3c) represents an individual canaliculis.
These target images were chosen because they represent actual details we wish to preserve with our reconstruction.

Experiment scenarios
For the experiments, scenarios of Low dose on different targets were defined. The number of projections ranges from $1/2$ to $1/10$ of the number required for fully projected reconstruction. An acquisition is considered fully projected when the number of projections generated is equal to $\pi/2$ times the



(a) Target 1 (b) Target 2 (c) Target 3

Figure 3: Targets used for the Matlab experiments. Add description

image size.
In our case we performed reconstruction with $1/2$, $1/4$, $1/7$ and $1/10$ of the projections. Knowing that our target images were of size 156×156 we used 125, 63, 36, 25 projections as displayed in Figure
These reconstructions were performed using FBP and SB-TV-2D reconstruction methods.

FBP and SB-TV-2D algorithms will be evaluated in terms of errors measured between the image target and the reconstruction and by visual inspection. We will also compare the preservation of edges and feature points with different number of projections.
The errors are expressed in terms of mean squared error (MSE), peak signal to noise ration (PSNR) and total variation (TV), computed between the target image and the reconstructed image.
The edge preservation will be evaluated in terms of total variation between the canny edge detection between the reference Target image and the reconstructed image.

First we will evaluate the classical FBP algorithm which will be used as a reference and then assess the improvements that can be brought by our proposed algorithm, SB-TV-2D.

FBP

The expectation of this experiment is that the classical reconstruction algorithm (FBP) is very sensitive to the variation of the number of projections.
The errors for the Target 1 are displayed in Table 1. The number of projections ranged between a half of a fully projected sample (125 projections) with a 6% error to the tenth (25 projections) with an error of 11%. The peak signal to noise ratio raged from 68 to 62. We may note that an image is considered as good quality with a PSNR ratio above $70dB$. The SAM ranged from 6 to 18. This means that with $1/2$ of the projections FBP is already considered noisy and ends up very noisy with $1/10$ of the projections.
These conclusion can also be verified visually looking at reconstructed images Figure 6 as well as post processing edge detection Figure 7. Without looking at the external part of the reconstruction (the black part outside of the circle) we can notice that the reconstruction seems noisier the less projection we have. And comparing the target we can notice that even with half of the projections, the edges are less differentiated, details are lost. This affects the post processing where we can notice that many edges are not recovered.

number of projections	1/2	1/4	1/7	1/10
RMSE	6%	7%	9%	11%
PSNR	67.90	66.92	64.17	62.21
SAM	5.70	8.44	14.17	17.95

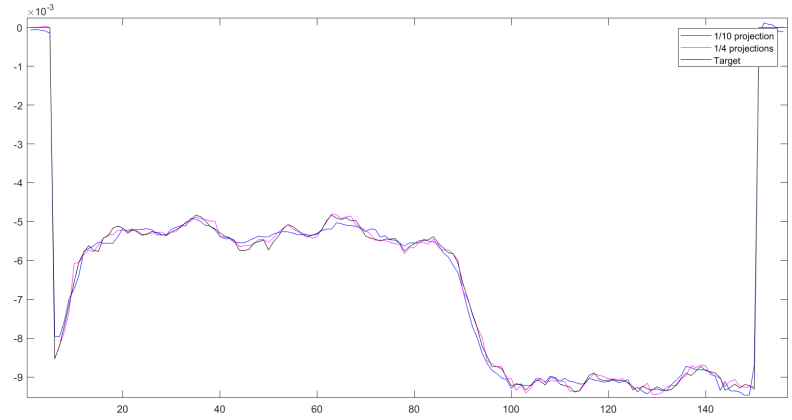
Table 1: FBP Errors on Target 1

These results allowed us to confirm that FBP reconstruction is highly sensitive to the number of projections. Alternative reconstruction methods are then necessary. We will evaluate the errors using our proposed algorithm.

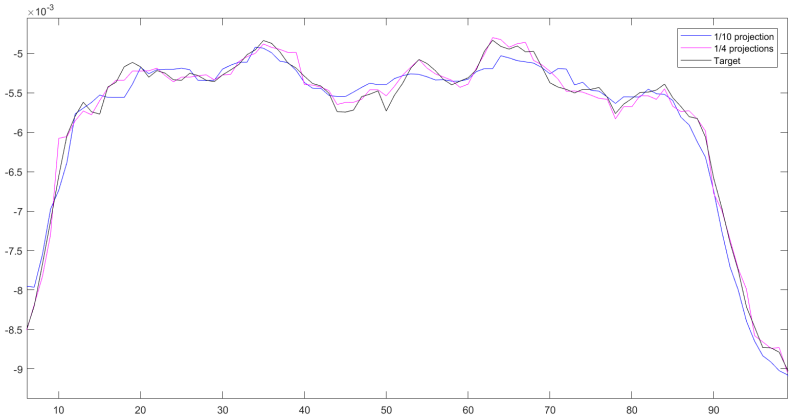
SB-TV-2D

Figure 5 displays the evolution of the MSE, PSNR and SAM of the reconstructed image and the target image at each iteration. The evaluation results are displayed in the Table 2.
Let us first focus on the MSE, Figure 5a. It is clear in this plot that each number of projection converges between 150 and 250 iterations. The RMSE ranges from 1% for one half of the projections to 2% for one tenth of the projections. The RMSE is also least affected by the number of projections, as showed in table 3, its variance is about 3×10^3 smaller with SB-TV-2D than with FBP.
The maximum PSNR values ranges from 76.75dB for half of the projection until 83.19dB. These values are globally higher than the reconstruction using FBP which were the maximum PSNR used was 67.90dB. The PSNR variance is also slightly less affected by the number of projection, with a variance of 6.73% with FBP and 6.01 with PSNR
The minimum SAM ranges from 1.39 to 2.47 the efficiency ranges between 24% better with the lowest number of projection, to 14% better with one half of the projections. With a variance of 0.16 against a variance of 30.57 for FBP, SAM is about 3×10^3 less affected by the number of projections.

Visually, looking at 6 and 7 we can see that until $1/4$ of the projections most of the important details are preserved. And then, as the number of projection decreases, the recon-



(a) Profile of the complete line



(b) Constrast on the osteocyte lacunae

Figure 4: Profile of the mid-line of the images

number of projections	1/2	1/4	1/7	1/10
RMSE	1%	1%	2%	2%
PSNR	83.19	81.56	79.11	76.75
SAM	1.39	1.69	2.07	2.47

Table 2: SB-TV-2D best error evaluations on Target 1

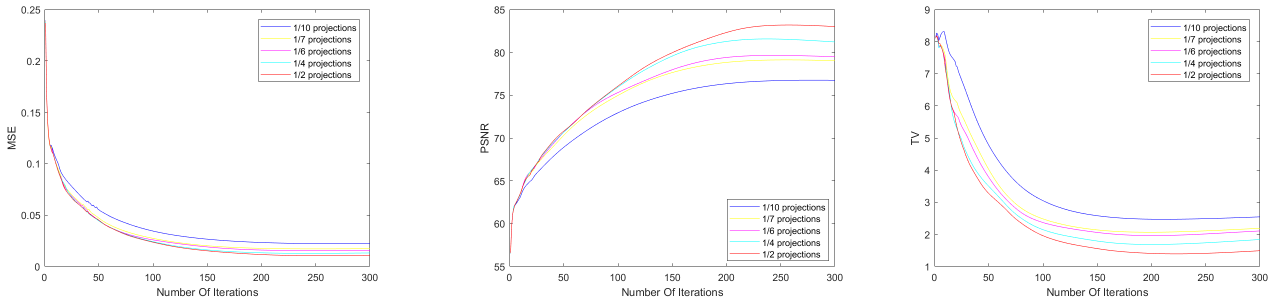
Reconstruction method	RMSE	PSNR	SAM
FBP	4.92%	6.73	30.57
SB-TV-2D	1.97×10^{-3}%	6.01	0.16

Table 3: Errors varience

structed image become smoother, and real edges are lost and false edges appear.
Looking at the horizontal mid-line of the target 1, we can tell the difference of accuracy between $1/4$ and $1/10$ of the projections. With $1/4$ of the projections local peaks are preserved with $1/3$ of the projections the reconstruction is subject to smoothing.

RMSE

The RMSE measures the average error. The results showed that the SB-TV-2D algorithm, with values ranging between 1% and 2% was less affected by the number of projection than the FBP reconstruction which RMSE ranges between 6% and 11%. In [Melli *et al.*, 2016a] and [Melli *et al.*, 2016b] Douglas-Rachford Splitting was used with TV, which has been shown to be equivalent to Split-Bregman TV [Setzer, 2009]. They evaluated the Relative Error (RE), equivalent to the RMSE on canine prostate soft tissues. In [Melli *et al.*, 2016b] The best reconstruction of their algorithm allowed to reconstruct the images with a RE of approximation 6% with 50% of the projections. Their worst reconstruction RE was 8% for 20% of the projections. On a femoral bone sample, in [Melli *et al.*, 2016a], their algorithm was assessed with a number of projection ranging only between 5% and 15%. With 10% of the projections they reached a RE of 15%.
Overall the SB-TV-2D has a very satisfying performance in



(a) RMSE (b) PSNR (c) SAM

Figure 5: SB-TV-2D errors on Target 1

terms of RMSE. Our proposed algorithm showed to be up to 5 times more efficient the the classical FBP reconstruction. Our results were better than those presented in [Melli *et al.*, 2016a] and [Melli *et al.*, 2016b] but we must note that our scenarios are based on removing projections with low noise (Gaussian noise of 1% of the maximum of the image). Our results are simulated from real images and further assessment is needed on real data.

PSNR

The PSNR measures the noise of the reconstruction compared to the target image. We may note that this metric can measure the smoothness of the reconstruction. Our algorithm reconstruction ranged between 83.19 and 76.75 dB. Such results are expected as we use TV minimization within the algorithm. We may note that this metric is subjected to the highest variance of 6.01. This variance results in the fact that with low number of projections, the reconstruction is smoother, as displayed in the reconstructions profiles in Figure 4. However, 76.75 dB is already a good signal to noise ratio and largely greater that the PSNR of the FBP reconstruction which never reached $68dB$ in our experiments. Our results are close to those presented in [Melli *et al.*, 2016b] for the Shepp-Logan phantom with a PSNR of $74.78dB$ for 10% of projections, $82.60dB$ for 20% and $84.83dB$ with 40%, on the other hand, for experimental data they reached noisier reconstructions with a PSNR ranging between 28.49 and 31.98 for 20 to 50% of projections. On bone sample in [Melli *et al.*, 2016a] a PSNR of $30.74dB$ was reached.
The SB-TV-2D algorithm comes with a very satisfying signal to noise ratio. These results are far more promising than those brought by FBP. The results also seem highly superior to those found in [Melli *et al.*, 2016a] and [Melli *et al.*, 2016b] for experimental data. However this metrics is affected by the smoothness of the reconstruction, in this context we may recall that by comparing the image profiles in Figures 4a, 4b, and 4 that our target sample is considerably smoother than theirs, which can explain the difference of ratio. Nonetheless, our results are similar to those found on phantom data presented in [Melli *et al.*, 2016b]. We can then imagine that on noisier data we would have similar results than for the experimental data in [Melli *et al.*, 2016a] and [Melli *et al.*, 2016b]. Although the low values of PSNR in [Melli *et al.*, 2016a] and [3] highlights the smoothness induced by the TV minimization.

SAM

The SAM measures the presence of artifacts in the reconstruction. Our experiments allowed showed that SB-TV-2D does not produce many artifacts. The value of the SAM ranges between 1% and 2%. The FBP reconstruction showed SAM values ranging from 6% to 18%. In the end, using SB-TV-2D allowed to divide the SAM by 6 for 50% of the projections and by 9 for 10% of the projections. We can then state that this method reduces significantly the presence of artifacts for low dose reconstruction.

Visually

Visually in Figure 6 we notice that we have a perfect reconstruction by using SB-TV-2D with 25% of the projections, and have a smooth reconstruction with the stronger details still present with 10% of the projections. FBP reconstruction using 50% of the projections looks blurry and from 25% of the projection becomes noisy, and with 10% of the projections lost most of the details.
Looking at the edge preservation on figure 7, we can notice that most of the edges of the image are preserved by using SB-TV-2D until 25% of the projections, but that the canaliculis present to the right of the osteocyte lacunae are still present.
With FBP most of the texture edges are already lost with 50% of the projections and with 14% of the projections many false edges appear, and finally with 10% of the projections only the osteocyte lacunae and one canaliculis contours remain.
Taking a visual comparison, SB-TV-2D allows a perfect reconstruction with 25% of the projections and has results of significantly better quality than the classical FBP method. The proposed algorithm leads to low dose reconstruction of better quality and preserves more details.

With two of our metrics we showed thanks to these experiments that 2 of our metrics were about 3×10^3 less affected by the number of projections. And with scores of RMSE equal to 1%, of PSNR above 80, of SAM bellow 2, and based on visual inspection, we can consider that it is possible with SB-TV-2D to reduce the number of projections by 4 and still get an acceptable reconstruction for the first target.

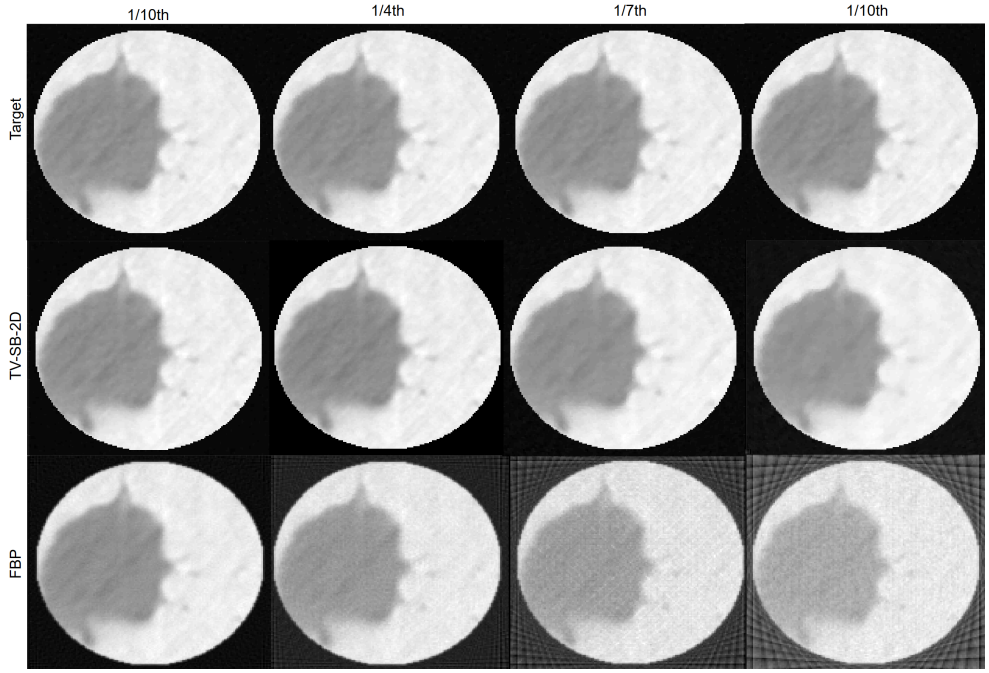


Figure 6: Comparision of SB

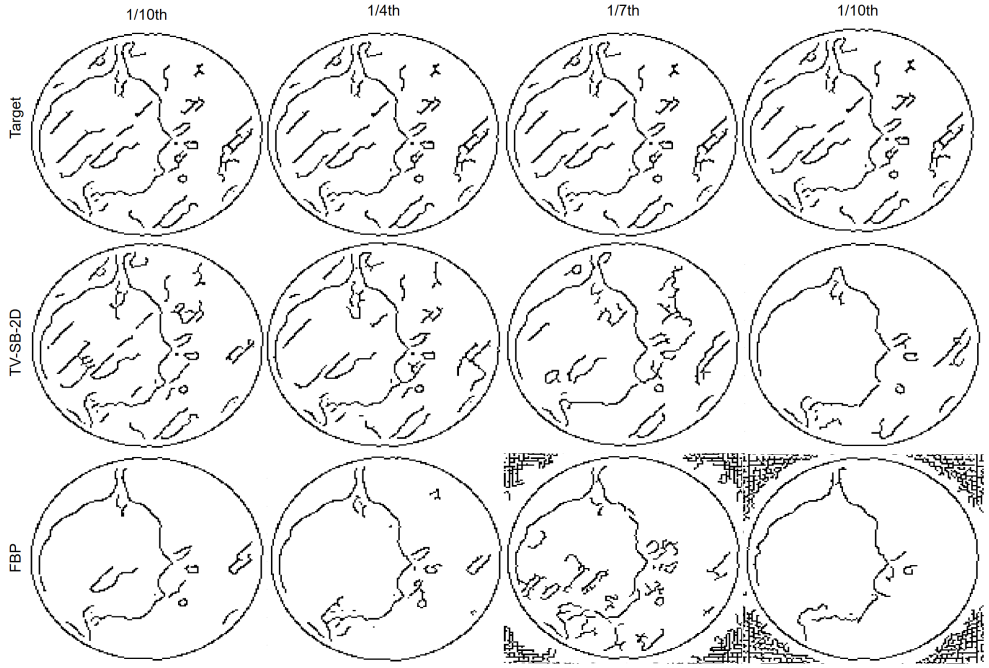


Figure 7: Canny edge detection

We then fixed as objectives on the evaluations for the two other targets with $1/4$ of the projections, a reconstruction with a RMSE of 1%, a PSNR above 80 and a SAM below 2. The results are displayed in table 4.
Visually, looking at Figure 8 we can compare SB-TV-2D and FBP reconstruction performances with $1/4$th of the projections. Almost no difference can be noticed between SB-TV-2D and the fully projected FBP, when FBP with $1/4$th of the projections misses some details and has an image which seems noisy.

Metric	Targer 1	Target 2	Target 3
RMSE	1%	1%	1%
PSNR	83.19	80.02	81.91
SAM	1.39	2.01	1.60

Table 4: SB-TV-2D evaluation for $1/4$ of the projections

3.2 Scalability results

We will now verify our second hypothesis regarding the scalability of the algorithm. We may recall that the previous experiments were performed in Matlab.

Experiments setting

PyHST2 is a software for tomographic reconstruction. In order to make our reconstruction more efficient, we need to change our reconstructions and projection calls. With respect to the reconstruction described in section 2.3, multiple forward and backward projection calls will be done per iterations. The objective of this experiment is to estimate the execution time of SB-TV-2D if it was integrated as a function of PyHST2 software. We will then first estimate the number of function calls necessary for a reconstruction and then assess the execution time of the algorithm on the ESRF computational cluster. In the previous section we state that using the proposed algorithm with $1/4th$ of the projection leads to a loss less reconstruction. As it is our objective, we will assess the execution time for a low dose reconstruction using $1/4th$ of the projections.

Estimation of the number of function calls

In order to estimate the execution time of the SB-TV-2D algorithm we had to estimate the number of PyHST2 call during the reconstruction of a experimental sample.
We decided to reduce the dose by 25% by using $1/4th$ of the projections. Looking at the Figure 5 we can see that with this specific number of projections SB-TV-2D converges within 200 iterations.
Yet, as presented in the pseudo code in Figure 2 for each iteration one call of F and FT is done. This means that for each iteration two calls of PyHST2 will be done. So far we get to 400 function calls. But most of the PyHST2 calls are done during the execution of the Krylov solver. Indeed this solver is iterative and each of its iterations required a call to F and FT. In average, for 200 iterations of TV-SB-2D, the Kyrlov call requires 10 iterations. We then reach a number of PyHST2 function calls of 4400 for a low dose reconstruction as each function calls F and FT are called 2200 times.
The dataset used to evaluate the speed of the algorithm is the complete bone volume described in section 2.1. This will allow us to evaluate the reconstruction of an actual experience done at the ESRF.

Parallelization results

The speed of the reconstruction with different parallelization scenarios are presented in Table 5. These result show that scalability of the algorithm is of high importance to reduce the computation time. By parallelizing the reconstruction on 10 different sub-volumes, the execution time can be about 7 times faster. The minimum reconstruction time was of $32.39h$. We then reach a reconstruction within 1.35 days, which is close to our objective of one reconstruction within one day.

Table 5: Execution time of F and FT PyHST2 calls with different parallelization factors. A parallelization factor of 2 will mean that the volume is split into two and the function calls are executed in two parallel nodes on the ESRF computational cluster

Function call	Parallelization factor	One call (minutes)	All reconstruction calls (hours)
5*F	1	1.16	42.53
	2	0.65	23.83
	4	0.4	14.67
	8	0.28	10.27
	10	0.23	10.27
5*FT	1	4.62	169.40
	2	2.58	94.60
	4	1.53	56.10
	8	0.77	28.23
	10	0.65	23.83
5*F + FT	1	5.78	211.93
	2	3.23	118.43
	4	1.93	70.77
	8	1.05	38.50
	10	0.88	32.39

We assess the scalability of the algorithm aiming to reach a reconstruction that would take less than a day. To do so we used PyHST2 software and parallelized the execution of the algorithm on the ESRF cluster. The results are displayed in the Table 5.
We showed that by parallelizing on the ESRF computational cluster it was possible to reduce the computation time by 7. Yet, even so we reached a best computation of $32h$. Even though this is above the $24h$ objective it is still a very good result as it is in the order of one day.
In order to reach the $24h$ reconstruction we can parallelize further the reconstruction, and optimize the algorithm. This computation time is long because too many projection (F and FT) calls are required per iteration of SB-TV-2D. In average for a 200 iteration SB-TV-2D reconstruction, the Krylov solver used to solve the equation (12) requires 10 inner iterations. This means that in order to solve (12) we have to make 10 times more projection.
The algorithm could be optimized in two different ways. The Krylov solver can be modified. It is currently based on linear conjugate gradient. Preconditioned conjugate gradient could be used and the threshold of the iterative solver can be optimized. The second way is to take only few steps with steepest descent and leave the rest for the next Bregman iteration.
The SB-TV-2D algorithm allows to perform low dose nano CT reconstructions of higher quality. The number of projections can be divided by 4 and still get a perfect reconstruction. With lower number of projection the reconstruction is still not affected by noise and artifacts and most of the important features of the image are preserved. With FBP the low dose reconstruction are quickly becoming noisy and most of details are lost. In all of the metrics used this reconstruction had significantly better results than the FBP and allowed bet-

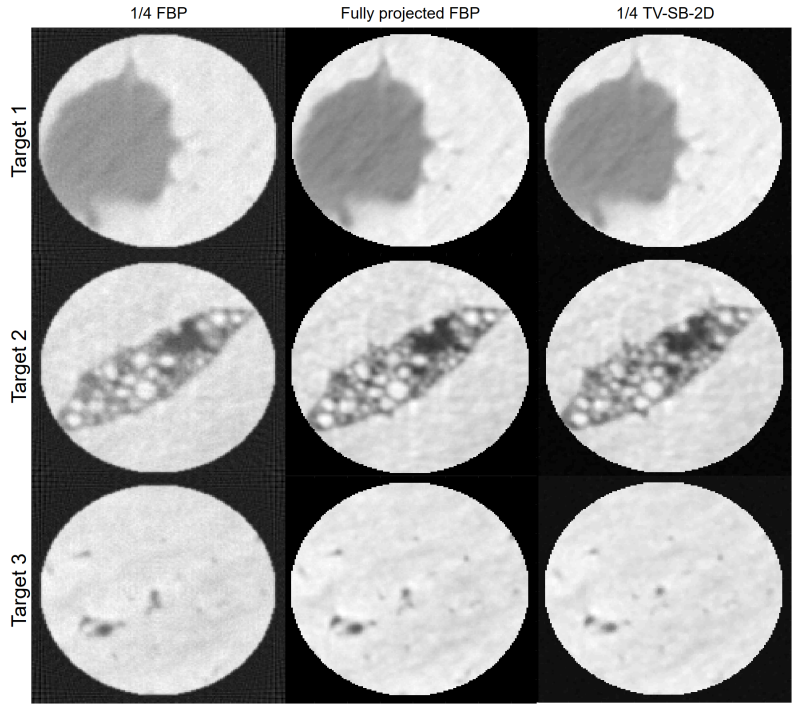


Figure 8: Reconstruction of all targets with $1/4$th of the projections

ter reconstruction quality than the state of the art methods.
As the dose is decreased below 4 times, the reconstructed image tend to be smoother than the target images. This limitation affects the ESRF data less as the reconstructed data tend be smoother than those used in the state of the art. Other methods in terms of regularization functional can be used based on the SB method one can easily change TV. Other works proposed to use dual complex wavelet shrinkage, non-local TV has been solved using Bregminized approach [Zhang *et al.*, 2010].

4 Conclusion

This project proposes SB-TV-2D algorithm for low dose SR nano CT reconstruction. This low dose reconstruction needs to be able to reduce the number of projection without affecting the image quality. Reducing the number of projections is important in order to reduce the invasivity of SR CT acquisition on the bone sample. This could have a direct effect on the image quality by reducing the artifacts that appear due to long acquisition time and high dose used during the experiment.
This research was motivated by two main objectives: 1) to present a low dose CT reconstruction of bone sample which does not impact the reconstructed image quality, 2) to make an algorithm that can be implemented at the ESRF to make the reconstruction scalable to reach the order of execution time should be one day.
We proposed as solution for low dose compressed sensing reconstruction based on SB-TV-2D algorithm. The CS problem is expressed using a constrained total variation minimization problem which is difficult to solve. The Bregman iteration is used to give an approximate of this solution iteratively, splitting L1 and L2 elements of the problem.
The experimentation showed that truncated under-sampled data could be reconstructed accurately using up to $1/4th$ of the projections and the higher contrasts could still be preserved with $1/10th$. The SB-RV-2D reconstruction showed to be less affected by the number of projection than the traditional FBP reconstruction and had better results than the state of the art solutions.
An implementation using PyHST2 software for tomographic reconstruction provided by the ESRF made this algorithm scalable and compatible with ESRF experiments data. We also note that using the ESRF computational cluster it was possible to make this algorithm more scalable by parallelizing the reconstruction over multiple sub-volumes. This way, the computation could be done within $32h$ by parallelizing the execution on 10 different sub-volumes. This is in the order of one day and is hence close to our objective of a reconstruction within a day.
Overall the results were promising. Yet some further validation is necessary. The reconstruction quality was assessed on truncated image for sake of feasibility on Matlab. We still need to reconstruct real complete data from the ESRF and verify if we get similar results. The speed of the reconstruction could also be improved by using alternatives to the conjugate gradient Krylov solver or even using different approach such as Bregmanized reconstruction.

References

[Abascal *et al.*, 2011] JFP-J Abascal, Judit Chamorro-Servent, Juan Aguirre, Simon Arridge, Teresa Correia, Jorge Ripoll, Juan José Vaquero, and Manuel Desco. Fluorescence diffuse optical tomography using the split bregman method. *Medical physics*, 38(11):6275–6284, 2011.

[Abascal *et al.*, 2016] Juan FPJ Abascal, Monica Abella, Eugenio Marinetto, Javier Pascau, and Manuel Desco. A novel prior-and motion-based compressed sensing method for small-animal respiratory gated ct. *PloS one*, 11(3):e0149841, 2016.

[Bregman, 1967] Lev M Bregman. The relaxation method of finding the common point of convex sets and its application to the solution of problems in convex programming. *USSR computational mathematics and mathematical physics*, 7(3):200–217, 1967.

[Burger and Osher, 2013] Martin Burger and Stanley Osher. A guide to the tv zoo. In *Level Set and PDE Based Reconstruction Methods in Imaging*, pages 1–70. Springer, 2013.

[Candes *et al.*, 2006] Emmanuel J Candes, Justin K Romberg, and Terence Tao. Stable signal recovery from incomplete and inaccurate measurements. *Communications on pure and applied mathematics*, 59(8):1207–1223, 2006.

[Combettes and Pesquet, 2011] Patrick L Combettes and Jean-Christophe Pesquet. Proximal splitting methods in signal processing. In *Fixed-point algorithms for inverse problems in science and engineering*, pages 185–212. Springer, 2011.

[Gaass *et al.*, 2013] T Gaass, G Potdevin, M Bech, PB Noël, M Willner, A Tapfer, F Pfeiffer, and A Haase. Iterative reconstruction for few-view grating-based phase-contrast ct: An in vitro mouse model. *EPL (Europhysics Letters)*, 102(4):48001, 2013.

[Goldstein and Osher, 2009] Tom Goldstein and Stanley Osher. The split bregman method for l1-regularized problems. *SIAM journal on imaging sciences*, 2(2):323–343, 2009.

[Hestenes and Stiefel, 1952] Magnus Rudolph Hestenes and Eduard Stiefel. *Methods of conjugate gradients for solving linear systems*, volume 49. NBS, 1952.

[Langer *et al.*, 2012] Max Langer, Alexandra Pacureanu, Heikki Suhonen, Quentin Grimal, Peter Cloetens, and Françoise Peyrin. X-ray phase nanotomography resolves the 3d human bone ultrastructure. *PloS one*, 7(8):e35691, 2012.

[Langer, 2008] Max Langer. *Phase retrieval in the Fresnel region for hard X-ray tomography*. PhD thesis, Villeurbanne, INSA, 2008.

[Li and Luo, 2011] Xueli Li and Shuqian Luo. A compressed sensing-based iterative algorithm for ct reconstruction and its possible application to phase contrast imaging. *Biomedical engineering online*, 10(1):73, 2011.

[Liu *et al.*, 2013] Y Liu, J Nelson, C Holzner, JC Andrews, and P Pianetta. Recent advances in synchrotron-based hard x-ray phase contrast imaging. *Journal of Physics D: Applied Physics*, 46(49):494001, 2013.

[Martinez-Criado *et al.*, 2016] Gema Martinez-Criado, Julie Villanova, Rémi Tucoulou, Damien Salomon, J-P Suuronen, Sylvain Labouré, Cyril Guilloud, Valentin Valls, Raymond Barrett, Eric Gagliardini, et al. Id16b: a hard x-ray nanoprobe beamline at the esrf for nano-analysis. *Journal of synchrotron radiation*, 23(1):344–352, 2016.

[Melli *et al.*, 2016a] S Ali Melli, Khan A Wahid, Paul Babyn, David ML Cooper, and Varun P Gopi. A sparsity-based iterative algorithm for reconstruction of micro-ct images from highly undersampled projection datasets obtained with a synchrotron x-ray source. *Review of Scientific Instruments*, 87(12):123701, 2016.

[Melli *et al.*, 2016b] Seyed Ali Melli, Khan A Wahid, Paul Babyn, James Montgomery, Elisabeth Snead, Ali El-Gayed, Murray Pettitt, Bailey Wolkowski, and Michal Wesolowski. A compressed sensing based reconstruction algorithm for synchrotron source propagation-based x-ray phase contrast computed tomography. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 806:307–317, 2016.

[Paige and Saunders, 1975] Christopher C Paige and Michael A Saunders. Solution of sparse indefinite systems of linear equations. *SIAM journal on numerical analysis*, 12(4):617–629, 1975.

[Rudin *et al.*, 1992] Leonid I Rudin, Stanley Osher, and Emad Fatemi. Nonlinear total variation based noise removal algorithms. *Physica D: Nonlinear Phenomena*, 60(1-4):259–268, 1992.

[Setzer, 2009] Simon Setzer. Split bregman algorithm, douglas-rachford splitting and frame shrinkage. *Scale space and variational methods in computer vision*, pages 464–476, 2009.

[Yang *et al.*, 2015] Xiaoli Yang, Ralf Hofmann, Robin Dapp, Thomas Van de Kamp, Tomy dos Santos Rolo, Xianghui Xiao, Julian Moosmann, Jubin Kashef, and Rainer Stotzka. Tv-based conjugate gradient method and discrete l-curve for few-view ct reconstruction of x-ray in vivo data. *Optics express*, 23(5):5368–5387, 2015.

[Zhang *et al.*, 2010] Xiaoqun Zhang, Martin Burger, Xavier Bresson, and Stanley Osher. Bregmanized nonlocal regularization for deconvolution and sparse reconstruction. *SIAM Journal on Imaging Sciences*, 3(3):253–276, 2010.

[Zhao *et al.*, 2012] Yunzhe Zhao, Emmanuel Brun, Paola Coan, Zhifeng Huang, Aniko Sztrókay, Paul Claude Diemoz, Susanne Liebhardt, Alberto Mittone, Sergei Gasilov, Jianwei Miao, et al. High-resolution, low-dose phase contrast x-ray tomography for 3d diagnosis of human breast cancers. *Proceedings of the National Academy of Sciences*, 109(45):18290–18294, 2012.

Execution Management for Exascale Machines and InSitu Applications

Thomas Lavocat
Supervised by: Olivier Richard, Guillaume Huard

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

Abstract

High performance computing (HPC) is moving from the Petascale platforms to the Exascale ones, meaning an increasing amount of computers are involved in high performance centers. Moving to the Exascale is a needed evolution to perform more complex simulations but it brings many challenges. Two of those challenges are the increased amount of computer failures and a computational power far beyond the I/O capacities. Workflow managers are large pieces of software that couple steps together in order to perform simulations. Usually this coupling is done by using disks I/O. Nowadays, simulations I/O are saturating HPC centers. To allow the next platform generation to be fully used by workflow managers, new configurations and organisations need to be applied. At their core, such configurations require the ability to partition the resources inside a job allocation, execute commands in those partitions, build communication links between possibly distinct programs, handle reconfiguration and failure events reliably, and let the user centralise this information to take decisions. To solve these issues, we are developing a scalable execution manager. The core idea of our implementation is to use aggregates of nodes as an execution unit. Linking those aggregates together in a hierarchy allows us to have a better scalability and to offer fine grained feedback over executions. Our solution uses groups to organise execution and to provide control plane communications between remotely executed programs. Control plane communications bring a new way for InSitu applications to plug their different parts together. Our solution also aims at providing sufficient functionalities to be used in a wide range of domains – already using remote execution tools. Integrating our solution into an InSitu application. This integration confirms that our approach is relevant, providing to it ways for its different sub parts to discover each others on an execution plan.

I would like to express my sincere gratitude to Olivier Richard, Guillaume Huard and Swann Perarnau for accompanying me during this internship and for their help and comments in reviewing this report ; to Théophile Terraz for his help understanding Melissa's code ; and to Françoise and John Barber for their precious corrections.

Contents

*

1 Introduction and motivation

As platforms approaching Exascale, systems are expected to provide hundred of thousands of computational units, each of them running thousands of execution threads. Having such a number of computation units leads to several issues.

On one hand, large-scale simulations are producing an ever-growing amount of data which is already saturating the disks bandwidth of current HPC centers. With the next generation platforms, researchers estimate than 1% of the data produced by the simulation will be stored on stable storage [Dreher and Peterka, 2016]. Post processing data analysis will no longer be an option. InSitu processing is a solution that, instead of using disks as a intermediary between different programs, transfers data when it is still in memory. Solutions like Melissa [Terraz et al., 2016] can consume experiment results on the fly in 1hr 30min, where it would need more than 10hrs just to read data from a conventional disk. Melissa is a non-common way to perform processing on HPC computers. It requires client/server communications, daemons, and a lot of small different jobs, all connected to each other. This system can be qualified as highly dynamic and has special needs. Those needs are not properly covered by the common message passing interface (MPI) and Melissa brings its own solutions to couple clients and servers. Even if it is an already working project, the way clients are discovering the server execution node needs to be improved. Researchers from PETS-C[1] have troubles in launching and monitoring machine learning and big data simulations. Their applications focus on wide parameter exploration of machine learning applications. In those configurations, depending on the parameters, applications can never converge or simply crash. Their applications need to have a constant feedback over the running code to detect failures and also to have a way to change at run-time parameters to correct potential problems. To change parameters at run-time an execution must be contactable. To our knowledge no current platform exploitation system is providing such communications facilities.

On the other hand, tools that configure, validate states, and exploit those systems, need to be highly scalable and failure resistant. On current High Performance Computing (HPC) platforms, systems are experiencing failures every few days : in the next generation, it will be several times an hour. To couple simulations, most of the current applications are using MPI. MPI is known not to be failure resistant. A single node failing takes down the entire execution. To avoid restarting from scratch on every problem, MPI applications can periodically checkpoint their states on persistent storage. If the application fails during its execution, it will be restarted using the last valid checkpoint as a starting point. Check-pointing and restarting is still possible, even with a MPI application exploiting an entire Petascale platform. Because of the failing ratio it will not be possible on an Exascale. Restarting many times an hour the application would lead to an ever-ending loop of checkpoint recovery without doing any proper computations. Using something else than MPI to couple applications is a solution to enable exploiting an entire Exascale platform, for instance, by having a global simulation made of several steps independent of other's failures. To our knowledge, such a solution is not available, and to be built, it will require a daemon oriented approach to control computers. Daemons are not commonly used because they tend to impact running simulation with some noise, but as platforms are gaining in complexity with more cores, new configuration are available to reduce the impact on simulations. The Argo project[2] is currently developing an operating system to exploit those Exascale platforms. This OS exposes light-weight containers to be attached to some cores allowing control programs to run without too much impact on the simulation. Such a solution aims at providing dynamic reconfiguration of computational units to host simulation code. Each computer of the simulation need to receive the same container configuration and at Argonne's researchers have not found a proper remote execution tool to apply dynamically their solution.

To address the next generation issues, we propose a solution which integrate a notion of resources partitioning and aggregation. Computational resources are aggregated in groups that can be defined on the fly as opposed to more static solutions. Partitioning in groups allows us to apply the same configuration on every resource within a group, making a group a container for configuration and execution. Our groups are hierarchically organised and can exchange messages. As a group has a constant contact over its computation nodes, and as a group can receive messages, we can apply at run-time reconfiguration over running simulations. By providing communication capabilities between remotely executed programs our solution can provide code-coupling capabilities. Groups also provide units for failure resilience by gathering every information of their resource. A group handles failures inside its resources set and thus can forward information to higher levels in the group hierarchy. The information gathering provides a constant feedback to the upper layer and enables a user to take apply failure resistant strategies. Our solution uses a remote execution tool as a basic layer. Initially designed to administrate platforms, remote execution tools are a handy way to contact computation nodes and are used in many contexts. Thus they are used in a wide variety of tools to exploit and administrate computers.

Preliminary results shows that our solution has a linear cost and a constant overhead regarding the remote execution tool our solution embeds. We present in our experiments an analysis of the cost to launch groups which also has a linear cost.

The rest of this report focus on the building of our tool and is organised as follows. We will first go through the state of the art, then we describe our design and implementation. We will evaluate our solution on Melissa, then we will discuss issues, have a word on further work, and then finally conclude.

[1] http://www.mcs.anl.gov/petsc/

[2] http://www.mcs.anl.gov/project/argo-exascale-operating-system

2 State-of-the-Art

2.1 Context

Many domains are intensively using remote execution engines : Workflow managers, tasks and jobs schedulers, DevOps, and reproducible research. Figure 1 shows a graphical representation of how those fields are related and table 1 summary the extracted functionalities we want to be able to provide.

2.2 Remote execution tools

The core activity of a remote execution tool is to make distant nodes execute programs. Those programs can be configuration tools in order to prepare a platform for a specific execution, simulation/programs to be run or clean-up programs to clean the computation units after executions. The most simple way to create a remote execution tool is to use a script looping over an ssh command, making any computer in the list execute a specific command. As this processing can be heavily parallelised, one can use a sliding window in order to work on x connections at the same time. A great majority of tools are doing this, but it as been proved inefficient and a distributed mechanism is more profitable. A distributed mechanism is one that uses particular nodes on the network in order to increase the parallel window. Usually those nodes are also in the bucket of those used for the remote execution. In the remote execution field, those specific nodes hold a running daemon running the same code as the root one and there are different strategies to start the daemon on the remote nodes. The daemon can be either always running on some key nodes and contacted when a remote execution is needed or can be started on demand. For those that are started on demand, two strategies are applied. Or the remote execution tools send its code on the distant node as an extra step or the code is already installed on the remote computer. The self-deploying mechanism provides strong independence regarding the remote computer. Lesser configuration is needed on the remote, and, better the tool is portable and provide flexibility.

Many improvements have been made in the literature and two main tools have been built. Clustershell [Thiell et al., 2012] and Taktuk [Claudel et al., 2009] are made to offer good properties such as scalability, reliability, efficient remote command execution, results gathering and other. Thus we will not present tools like pdsh [Claudel et al., 2009] that are less efficient.

2.3 Tasks/Jobs Managers

Centralised or distributed resources and tasks managers are tools used to manage large platforms. They come with scheduling capabilities and their purpose is to exploit computation nodes, making scheduling, execution, and monitoring. Those tools eventually use execution engines to contact and make computers execute jobs.

2.4 Workflow Managers

A workflow is the automation of a process [Liu et al., 2015]. There are two kinds of workflows : business and scientific ones. Business workflows are task-oriented and Scientific workflows are data-oriented. Business workflows are a set of activities linked together in order to optimise and orchestrate execution. Scientific workflows are used to model and run experiments on computational resources. The most general representation of a workflow is the Direct Acyclic Graph (DAG) which is a graphical representation of the tasks to execute. DAGs are translated into tasks and later scheduled by an external scheduler. Workflow managers are large pieces of software made of several layers : presentation, user services, workflow execution plan generation, and execution [Liu et al., 2015]. The three firsts layers are used to prepare the execution plan that will be and compiled optimised before the execution layer. The execution layer takes care in the general case of the scheduling, tasks execution and the fault tolerance. Fault tolerance can be classified in two kinds, proactive and reactive [Liu et al., 2015]. Reactive failure tolerance is about relaunching failed simulations. Proactive failure tolerance is about redundancy to handle possible failures. As computation becomes more and more complex on supercomputers, workflows have to deal with more and more steps and need underlying layers able to deal with this increasing amount of work. As examples of workflows we can find Pegasus[3], Taverna[4], Kepler[5], DIET[6] and others [Deelman et al., 2015].

2.5 DevOps

DevOps is a practice involving fast and flexible development and platform administration [Zhu et al., 2016]. A DevOps has to efficiently integrate development, delivery, and operations with fluidity between those separate competence fields [Ebert et al., 2016][Zhu et al., 2016]. They are using a lot of tools in order to achieve those tasks, from automatic tests bench, to automatic deployment and reconfiguration to achieve very short development cycles. In this field, cookbooks are used to pilot orchestration tools like Ansible[7], Puppet[8], Chef[9] among others [Liu et al., 2016]. Those tools are remote execution programs by themselves with flavours or policies dedicated to them. An interesting idea from the DevOps field is the philosophy deifying infrastructure as data. The physical layer is abstracted. And we can use infrastructure the same way that we use software [Johann, 2017]. Important points of the DevOps field are traceability and auditability. [Johann, 2017]

2.6 Reproducible Search and Tools

In reproducible research, the key idea is to have a tool able to understand a procedure and relaunch it every time it is needed, tools like Expoi[10], Plush/Glush[11], Execo[12], XPFlow[13] among others [Buchert et al., 2015]. All of those tools are execution engines plus some policies that make them

[3] https://pegasus.isi.edu/
[4] http://www.taverna.org.uk/
[5] https://kepler-project.org/
[6] http://graal.ens-lyon.fr/diet/?page_id=551
[7] https://www.ansible.com/
[8] https://puppet.com/fr
[9] https://www.chef.io/chef/
[10] http://expo.gforge.inria.fr/
[11] https://www.planet-lab.org/
[12] http://execo.gforge.inria.fr/doc/latest-stable/
[13] http://xpflow.gforge.inria.fr/

attached to specific kinds of platforms. Unfortunately, for some authors, those tools suffer from coupling problems. They are too attached to some grid philosophies to be used in a more general way. And some researchers tend to use their own scripts for remote execution[Stanisic et al., 2015].

2.7 Common Requirements

From workflow managers, DevOps, tasks/jobs mangers, reproducible research, and InSitu applications, we can extract a list of specific needed functionalities. We will briefly present those here and summarise them in the table 1.

High Level Interpretation
High level interpretation mean to offer the final user "user friendly" interfaces to setup its execution plan. A high level interpretation language can either be a graphical view of an execution as a Direct Acyclic Graph (DAG) or a description using a Domain Specific Language (DSL). The **DAG Interpretation** consists of extracting a list of steps to execute, to validate their coherency, and to give them to an underlying layer able to schedule the steps as tasks/jobs. Having user need validation. **Input validation** means for instance to compile user entry to ensure termination of its program or its correctness.

Remote Execution and Platform Control
Controlling a platform is knowing every characteristic of a range of computers, to have control of them, such as, reboot, making configurations, to be able to exploit them with some job scheduling. Or to expose API for a tool to discover the platform in order to ask resources reservations. **Job and Task scheduling** is having a list of jobs/tasks to process, to choose the right place at the right time to execute them. Complex algorithms are used to optimise user waiting time and platform utilisation. Jobs schedulers use specific **platforms support** to be able to make node reservation or job monitoring. Platform managers can expose **Resources discovery** APIs to help other programs to make reservations and to schedule tasks.

Execution Management and Fault Tolerance
At a point, executions need to be managed. **Results Analysis** allows a user to visualise using tools its execution results. **Interactive Execution** helps the user to debug his applications. **Provenance tracking** allows one to gather information about every configuration and hardware details on his execution. It is used by computer scientists to understand important impacting parameters. For programs to be executed, specific environments sometimes must be setup. Often used by tasks/jobs managers in order to prepare nodes before execution, **verification and configuration** is checking the state of a computational resource and configuring it.

Platforms and executed codes may suffer from failures such as instabilities or bugs. Running large code on HPC centers may take several days of computation and cannot be restarted from scratch at every problem. Solutions are made to provide fault tolerance. **Checkpointing** is a solution that periodically saves the state of the simulation on disk. Upon crashing, the simulation will be restarted from the last valid checkpoint. **Redundancy** is another way to be fault tolerant. Redundance supposes that an execution is made several times, then, if one fall other can still be used.

Code Coupling
Due to the increasing complexity of computations and to reduce disk usage, people are focusing nowadays on InSitu computing. InSitu means "in place", which in the simulation context means to keep the data close to its production source and not to use a slow intermediary like disk storage for instance. Those techniques consist in coupling existing codes in order to transit information when they are still in memory. Bredala [Dreher and Peterka, 2016] is a InSitu middleware enabling simulations coupling with N to M patterns. Bredala is using MPI as its communication layer to multiplex simulations. MPI is not dynamic, and simulations cannot be added on the fly to an existing coupling. This problem is highlighted by Bredala's author. Solutions like Melissa bring new ways with client/server patterns to multiplex simulations.

2.8 Related Work

We will focus this related work using the required functionalities highlighted in the previous section.

2.9 High Level Interpretation

High level interpretation helps users to translate their wishes to script a remote execution engine. For instance, workflow managers translate DAGs to execution engines. The paper [Deelman et al., 2009] gives a good overview of the execution model for workflow managers. Pegasus maps workflows on different targets like, PBS, LSF, Condor, and individual machines. Pegasus uses the DAGMan workflow engine which interfaces to Condor scheduling queues. From the DevOps world, Ansible, Puppet and Chef [Liu et al., 2016][Ebert et al., 2016] are configuration management tools, all working with recipes. Those recipes are sometimes written in DSL languages, or sometimes directly in a programming language (ruby for Chef). Due to their file-based configuration files, they are static tools made for administration purposes. Thus, if the group notion is embedded in those DSL it is only made to serve general administration purposes. For instance, a bunch of computers will be dedicated to run database software, another bunch will be dedicated to run web servers. Computers are not intended to move from a group to another quite often. For reproducible research, domain specific languages (DSL) are often used as an interface for the user. This is the case of EXPO, OMF, or XPflow per instance. Plush/Glush, requires a XML configuration file, which is not far from a DSL. Those tools are made to expose a research oriented API. They provide facilities to interact with tasks and resources managers, making them platform dependent. For the rest of the interpretation, the remote execution script, is mainly made of command line like Taktuk and clustershell. As for Ansible, Chef and Puppet Clustershell which embed a notion of group but need to be statically declared in configuration files. TakTuk is more in the Unix philosophy, a Swiss knife understanding complex command lines or interpreting its standard input to receive commands.

Command line tools present the advantage to be more simple to interface with. Tools asking for static configuration files to run may impact the dynamism of the tool we want to build. We would prefer a tool interpreting commands at run-

time on its standard input. Expo shows that it is possible to interpret DSL and to translate it over command line tools.

2.10 Remote Execution and Platform Control

OAR [Capit et al., 2005], developed at LIG, uses Taktuk [Claudel et al., 2009] to remotely contact nodes and provide a strong independency between OAR and the computation nodes. Taktuk is used to deploy images on nodes and to make deep configuration of the software stack on each computation node. Slurm [Yoo et al., 2003] another task manager uses its own daemon-based architecture to control computational node. Slurm is more administration dependent then OAR as it needs to be installed on each computational node. Flux [Ahn et al., 2014] asks the question of another way to manage HPC centers with a scalable distributed approach. Flux offers job scheduling and execution and a way for launched programs to access easily a distributed data base, that may be used as a way for them to communicate. Flux uses a tree-based topology to communicate between daemons using Zmq[14]. In order to be scalable, Flux's distributed approach seems natural. For Slurm – and for clustershell –, having a installed daemon on each node increase the administration cost of the platform. For Clustershell, not every node needs to be installed in advance, only group leaders. We still tend to prefer the OAR control on nodes using TakTuk as it is mostly an administration less tool. Ansible also only requires an ssh connexion to configure nodes. Like Flux, from a scalability point of view Puppet is an interesting tool, because it seems to be able to use a tree-based communication layer.

2.11 Execution Management and Fault Tolerance

In order to manage execution, remote execution tools need to gather information about executions. It is done depending on the needs of the upper layers. DAGMan used by Pegasus does not interact with jobs independently but reads the logs of the remote execution engine to keep track the status of the jobs. Ansible, Puppet, and Chef suffer from a lack of control of the execution cycle. And they do not collect execution results [Buchert et al., 2015]. They only offer information about the success or the failure of a step. For reproducible research, Expo, XPFlow, and Execo three related tools that use Taktuk as a base layer to contact nodes and execute commands, thus making those able to have a fine grain control over executions. Fine grain control enables message sending, standard input control, and life-cycle control of a remote process. More closely, TakTuk and Clustershell, are two highly parallel remote execution engines able to execute commands on remote computers and gather execution results. Taktuk gather every command results and node status to the its root. This feedback gathering is important to take decisions on a remote execution. Current workflow systems' fault tolerance, exception handling and recovery are ad hoc tasks undertaken by individual workflow designers rather than being part of the systems themselves. Triana passively warn the user when an error occurs and lets him debug the workflow. Using a smart re-run feature it can avoid unnecessary computation. Like Pegasus, Askalon supports fallback, task-level recovery, checkpointing, and workflow-level redundancy. Plush/Glush has the capacity to manage failures, through a constant stream of information from every node involved. Taktuk detects nodes failure and keeps a constant feedback about executed programs. Flux uses Zmq [Hintjens, 2011] as a communication layer between its nodes. Zmq is a high level communication library that provides some glue around TCP to make it more reliable. It also provides high level communication patterns that ease the development of distributed applications.

To apply different fault tolerance strategies, one needs at least to detect faults. Having a constant feedback at a basic layer allows decision taking for fault tolerance. It brings a failure detector by analysing timeouts on communications between nodes and by analysing remote execution return code.

2.12 Code coupling

Code coupling is a strategy to make InSitu programming. It avoid useless writings and disk use to transfer data directly from a program to another. Bredala [Dreher and Peterka, 2016] is an in situ middleware offering split and merge capabilities for data structures. It relies on MPI to connect different programs and this makes its weakness regarding flexibility. FlowVr is a virtual reality middleware with interesting design. Each node embeds a daemon to which sub-processes will register in order to communicate with others through an event-based programming model. FlowVr relies on a separate launcher to deploy its daemons and is not dynamic, no applications can join the runtime after lunch. Melissa is an InSitu library running as a server for parametric studies. Melissa uses Zmq for the communication layer between client and server application. Melissa's main weakness is using the file system for the client to discover the server. Melissa is dynamic, many clients can bootstrap and connect to the server to send data without knowing when and where they will be executed. To multiplex simulations and/or visualisation tools, different approaches are used, a common one is to build a meta application using MPI to connect different MPI codes into one big application, or less common is to use an external communication library to link programs together. Both solutions bring their challenges. From the MPI point of vue, it is a complex task to associate existing codes into a large fully functional one. Codes need to be coupling ready, for instance, comWorld is forbidden. This solution also suffers from a lack of flexibility at runtime, everything needs to be prepared in advance and the running configuration needs to be well setup. As MPI suffers also from a lack of failure tolerance, a single sick node in the runtime takes down the entire application and it may be very hard to build a correct checkpoint mechanism for complex workflows. With Exascale platforms the number of nodes involved in simulation will continue to grow, increasing the probability of failures per time unit, and the need of a more reliable way to connect MPI codes will quickly emerge.

Using MPI to couple simulations together is not a solution for the next platform generation. Coupled simulation's part need to be able to fall without restarting all the application. TakTuk enables remotely executed programs to communicate with each other on a common communication channel. Each TakTuk's launched program has an address on a special com-

[14] http://zeromq.org/

munication channel. Currently, for a remote program, there is no way to discover the address of another one on the communication channel, thus, it does not allow two programs having no knowledge about the run-time to exchange messages. Adding some glue around may enable one to develop those capabilities.

2.13 Summary

As a reminder, we want to be scalable in order to fill the Exascale gap, to manage aggregates of computation units as groups dedicated to executions, to bring control plane communication facilities to enable easy implementation of InSitu programs, to provide a centralised control point to have a constant feedback over the whole execution plan, and to be highly dynamic in order to add or remove aggregates at run time allowing dynamic client server applications.

Tools interpreting DSL or DAG are too much dependent on static configurations and our groups need to be dynamic. Thus we cannot tend to use tools not providing strong independence regarding configuration. TakTuk and ClusterShell, two command lines tools are preferred choices because they are able to contact ranges of computers with some parameters on a command line and to keep connexions open in order to be exploited on run time.

To be scalable we need to focus on a strategy like Flux, Puppet, ClusterShell and TakTuk does. Having a communication tree between daemons should be as much administrationless as possible in order to manage stock operating system. This platform independence enables a future user to develop specific platform support. Furthermore, TakTuk shows that having a tool with few dependencies with auto-propagation features can be efficient.

Regarding fault tolerance, Plush/Glush furnish a constant feedback that allow one to develop every fault tolerance mechanism over it. We should avoid the non-existent information collection of DevOps or workflow managers tools. TakTuk also provides a way to have interactive executions on remote nodes. Our remote execution tool should do the same.

TakTuk possesses a way for remotely executed programs to discuss on an external communication layer. Flux seems to offer a way for remotely executed programs to have access to a distributed database. Both solution may enable to provide code coupling.

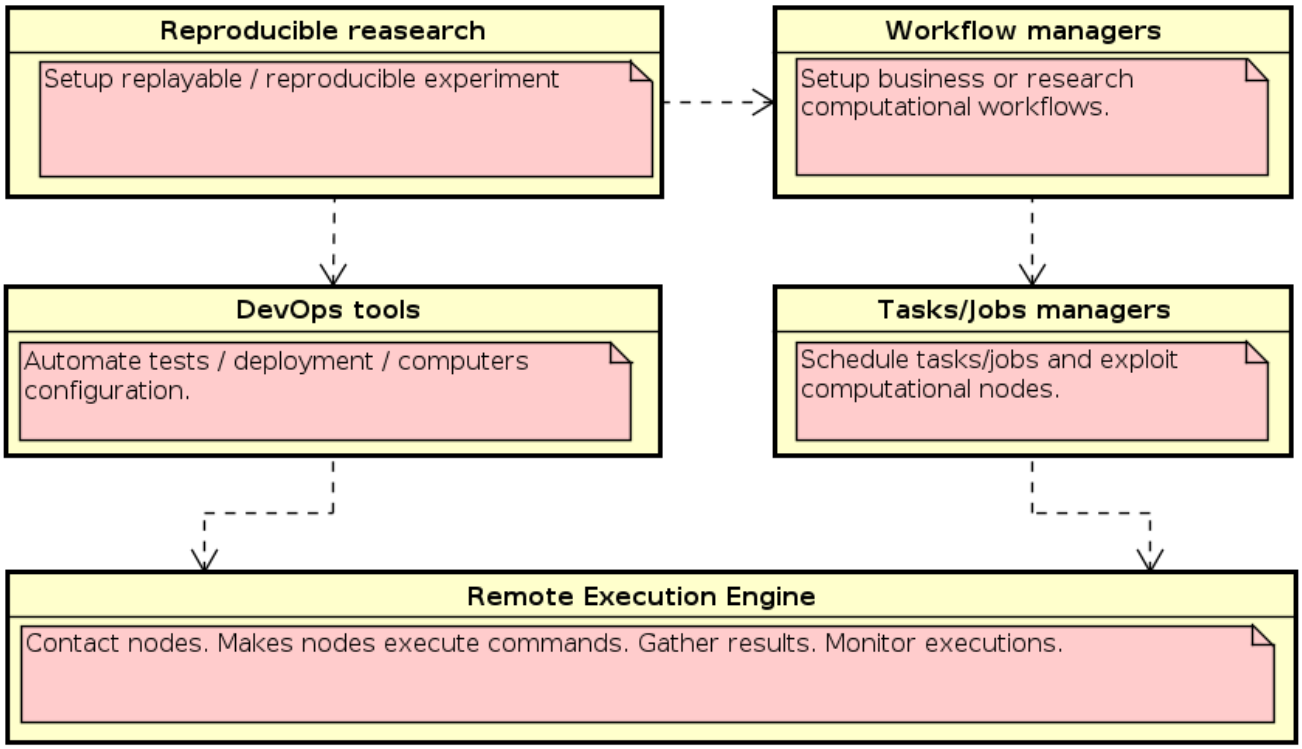


Figure 1 – Dependences between domains using remote execution tools.

functionality	Reproducible research	DevOps	Workflow managers	Tasks/Jobs managers	InSitu
DAG Interpretation			X		
Job/Task Scheduling			X	X	
Remote execution	X	X	X	X	
Results Analysis	X	X	X		
Input Validation	X	X	X		
Platforms support				X	
Ressources discovery		X		X	
Provenance tracking	X				
Checkpointing	X		X		
Verification and configuration		X		X	
Interactive Execution	X	X	X		
Code Coupling					X

Table 1 – Functionalities repartition over the different fields

3 Scalable Execution Management on Hierarchic Group Organisation

3.1 Design

Our design isolates applications in **groups** of computation units. Those groups are interconnected via **routers** which allow messages to transit between them. Having a messaging protocol between groups enables a `group leader` to receive orders from a remote controller, to send **feedback** about executed applications, and to provide a **control plane** mechanism.

3.2 Scalable Groups Hierarchy

A group is an isolated application, an encapsulation of a remote execution tool. As groups are connected together, it allow us to use several times a different remote execution tool instances. If the underlying remote execution tool is able to handle a hundred thousands nodes, then with this nested design, our tool will be able to handle as many times this amount of computers. A group is a high level representation of a set of computers. Usually remote execution tools are able to detect node failures, thus, enabling for a group, the detection of falling nodes. The information about a failure can be forwarded to a centralised place where a decision can be taken. As groups are linked together they are able to communicate with each other. The gateway for the communication is done by the group leader, the root of its group. Building a group is about two steps, to contact nodes in order to establish a stable communication tree between them and to use this tree to execute commands. Those two steps are given by the underlying remote execution tool. The group leader is in charge of controlling the underlying remote execution tool. A group hierarchy is constructed by making groups deploy groups inside themselves. A group does not necessary inherit from its parent resources. To be interconnected, nodes possess addresses. Those addresses are relative to the group. A node, being group leader for a group, can be follower for another one as shown on figure 3.

3.3 Control Plane Communications

The capability to exchange messages between two groups aims at providing a control plane to remotely executed programs. Each of those is able to send short messages to others. Control plane is a terminology from the networking community, as opposed to data plane. In the networking community, the data plane is the actual data exchange while the control plane is the setup of this exchange. It means learning the best routes to take to transfer data. Our capability let remotely executed programs contact each other without knowing the IP address of their peers, this ability to contact another program without knowing its actual execution machines. A program can send its localisation information in order to be contacted using a better communication layer to exchange data. Our solution is not suited to exchange intensive data streams nor large data chunks but gives the bare minimum to two foreign processes to open high-speed communication pipes.

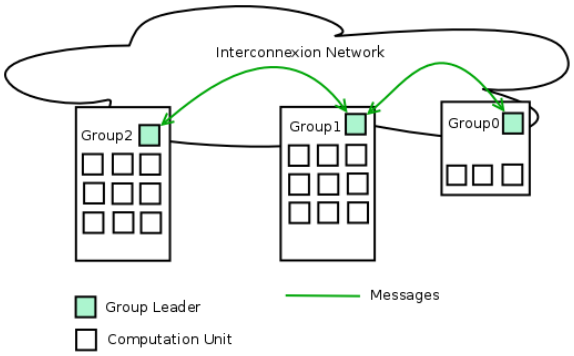


Figure 2 – We want to connect groups together on a meta network. Each node within a group is able to communicate with others. Our design allow out-of-group communications.

3.4 Propagation of Standard System Feedback

Because all nodes are linked through an SSH session, a group is easily aware of nodes failures. Informations about executions is naturally gathered by remote ssh execution. Thanks to the tree topology of the hierarchic group organisation, groups can easily send feedback information to their parents. Thus the root node can naturally gather information and take decisions about life cycles. Having a centralised control point in a distributed organisation enables adding/removing groups on the fly making this solution highly dynamic.

3.5 Chosen technologies

We present in this section the two major choices made to implement our design.

TakTuk

To achieve the best scalability, TakTuk offers the best choice, with its work-stealing algorithm. Furthermore, it only requires a Perl interpreter and a ssh daemon on the remote node. Beside node connections, monitoring and remote execution, TakTuk has a communication capability allowing remotely executed processes to send messages to each other. This capability is achieved by opening on each node, for each executed command, file descriptors. A child process only needs to read and write on those files to communicate with its local TakTuk's daemon, TakTuk's ship with a C library allowing one to easily use this functionality and build MPI like programs using only TakTuk as a communication layer. As the communication is done over SSH, it is not made for high performance data exchange and one should prefer RDMA[15] links over Infiniband. But despite this performance issue, its characteristics offer an easy way to exchange information between distinct programs by writing and reading through files descriptors.

The Distem benchmark [Buchert et al., 2014] shows how to emulate computational units using folding and advanced configuration. In their evaluation they perform a performance comparisons between TakTuk and Clustershell over forty thousand folded nodes. In this configuration it takes 100

[15]https://en.wikipedia.org/wiki/Remote_direct_memory_access

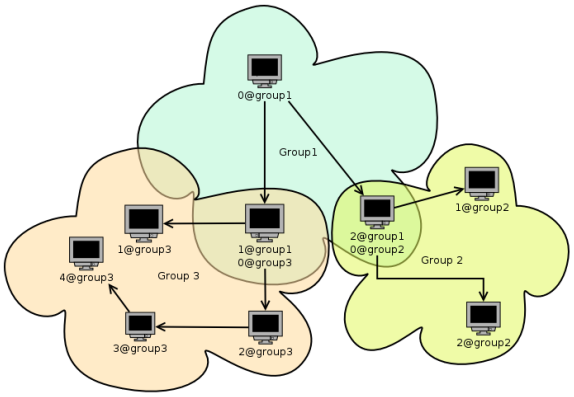


Figure 3 – Multiple groups connected through a tree topology. Each group is a group instance. Groups are nested and due to their access to TakTuk's communication layer, they are able to exchange messages between them.

seconds for TakTuk to complete the task and 600 for Clustershell.

Python

TakTuk is written in Perl, and a natural choice would have been to extend TakTuk in its language. But instead we used Python for three reasons : Python is widely used in the scientific community ; the context is more IO intensive than computing intensive, and thus, an interpreted language is usable ; and a modular approach allow to use another remote execution too than TakTuk if needed.

3.6 Implementation

This section highlights key concepts behind the implementation. Our solution is made of Groups and Routers as shown in figure 4. The basic Python encapsulation of TakTuk will be discussed first. This encapsulation will be referred as group or network depending on the context. The router layer enables message routing between different groups, and thus, the ability to construct group hierarchy, the task processing library abstract complex operations on the tree into an object-oriented application programming interface.

Experimentation setup

All through this document, we will evaluate performances of our solution on the Grid'5000[16] testbed. As we do not have access to many computation units for out tests we are emulating a larger number of nodes by folding them over real ones. In our configuration, to fold means to contact several time the same computation units as if it is a different machine each time.

The computers used for experiments come from Grenoble's clusters Genepi and Edel. A Genepi node contains two Intel Xeon E5420 QC working a 2.5GHz, has 8 GB of memory and gigabyte ethernet network. An Edel node contains two Intel Xeon E5520 working at 2.27 Ghz, has 24GB of memory and a gigabyte ethenet network. Those two clusters

[16]https://www.grid5000.fr/mediawiki/index.php/Grid5000:Home

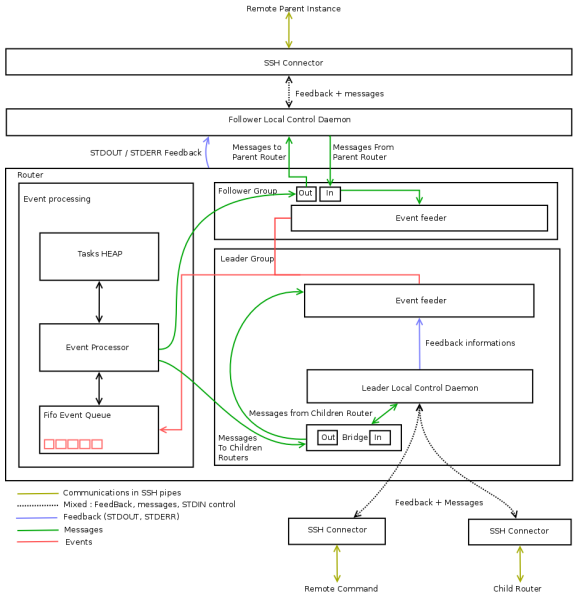


Figure 4 – Internal components organisation.

are also able to work on Infiny Band but we are not using this capability.

Group

At its core Groups use the Python Pexpect library to launch and control TakTuk's life cycle. TakTuk is able to connect nodes and number them. Numbering allows messages to be selectively sent to a node. Messages can be commands for the node to execute or messages to deliver to a running program on the remote computer. To enable receiving TakTuk's communicator messages, Groups make TakTuk launch an extra process called here a communication bridge. This process has access to TakTuk's communicator's file descriptors. By communicating with a group on a Unix socket, a bridge allows it to receive messages from TakTuk's. As shown on figure 4 Groups are able to parse TakTuk's feedback to extract meaningful information. This information is routed to different semantic callback functions. Groups give as an external API a set of registration functions allowing to get notified about : success or failures about operations, sub-process activity, or messages received on TakTuk's communication network.

Python has a global interpreter lock that forbids it to execute several threads at the same time, even on multi-cores computers. It is an advantage in the sense that it allows us to limit the footprint on computation nodes but it impact on the coding style. Because of the Pexpect library used to manage TakTuk, we already are in a multi-threading scheme. The library uses at least one thread to interpret TakTuk's I/O. We allow ourselves the use of a second thread in order to fetch information from the Unix socket. Because information about the tree manipulations can come from two independent sources, there are potential race conditions. Because of the global interpreter lock, only a thread can work at a time, thus, there is no performance gain using multi-thread to handle events from different sources. Furthermore, work-

ing with locks may lead to deadlocks or performance issues. We use an event-based programming model. When a thread deals with an incoming I/O, it adds an event in a synchronised queue, which contains for information, what to execute on which data. This way, if the Python scheduler interrupts an event execution to handle an incoming I/O it will only add a new event inside the queue and never compromise the internal data structures. This scheme provides us with an easy way to maintain codes by removing a lot of uncertainties due to race conditions, which are well known to be hard to debug especially on widely distributed software.

Groups are nested in a hierarchic tree topology. For a nested group, its group leader belongs to its parent's group. Thus, group instances can have two roles, or to be leader of a group or to be follower inside a group. Only group leaders can give orders, non-group leader group instances can only exchange messages.

Performance Evaluation A group is only a Python encapsulation of TakTuk. We need to set up an experiment validating this implementation's overhead. In this section we provide a performance analysis for the group's functionalities through a simple setup. The experiment takes place on 10 computation nodes from the Genepi cluster, a Grid'5000 cluster from Grenoble. Figure 5 shows an increasing workload from 1 to 2000 nodes for TakTuk and a group. At each iteration, the same work is performed, which is, starting, connecting nodes, numbering them, making them execute the uptime command, and performing a shutdown on them. As we do not have enough available computers to make a real 2000 nodes experiment, we fold them over ten real ones. The folding implies that we consider several times the same node and contact it several times.

The experiment shows that the group overlay has only a constant overhead compared to TakTuk. And without outliers to the 95 percentile we are able to compute linear regression 6 for both of them. The announced prediction for a forty-thousand execution like this one is about 96 seconds for TakTuk and 102 for groups. Distem [Buchert et al., 2014] papers who emulated forty thousand nodes on Grid'5000 with approximately the same folding ratio (around twenty instances per node) reached similar numbers for a TakTuk deployment, which let us claim that a group will be as highly scalable as TakTuk.

Meta-Network and Routing

We now want to interconnect them together in a way that they can exchange messages and achieve one of our primary goals : having an execution engine allowing control plane messages exchanged between launched programs. Group instances are connected to each other inside router instances. Routers take care of messaging transfer between groups. It is used to forward feedback information to the root instance and to send commands to leaf instances. Figure 4 shows the internal organisation of a router. Groups gather information and register events in a common synchronized queue. The router's job is to process those messages in a FIFO order. Figure 3 shows a configuration where three groups are connected to each other.

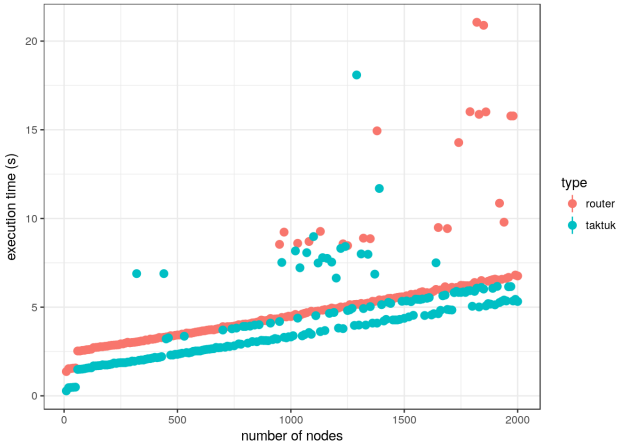
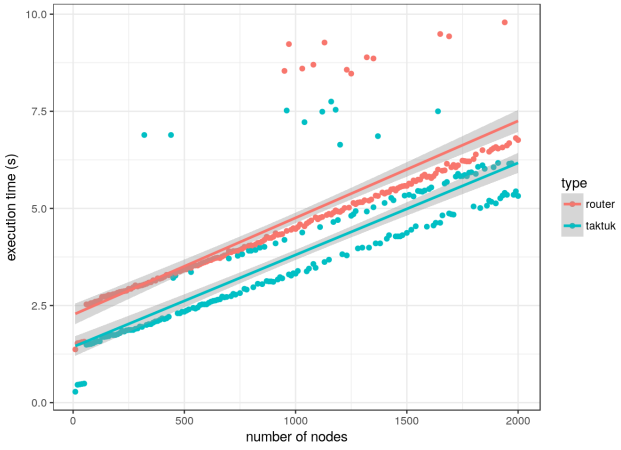


Figure 5 – Execution time comparison between TakTuk and the router with one measure per point. The experiment is done using 10 nodes to emulate 2000 nodes. This experiment shows the growing execution time as a function of the number of nodes and shows the overhead of the router. In this configuration the router only contains a group and is equivalent as an encapsulation of TakTuk. TakTuk and the router perform the same work. Figure 7 shows the needed code – without the task API – to generate this trace.

Figure 6 – Linear regressions based on data from figure 5 experiment.



As for groups, the router works in two modes. Root and non-root modes. The root mode is reserved for the only instance that is not launched by TakTuk. This instance has no father among the communication tree.

Life Cycle The router is able to replicates itself on computers. To achieve the configuration shown in the figure 3 the root instance, which in this case is `0@group1`, boot and start a group named `group1` with two computation nodes inside. The root instance then replicate itself on the remote computers. `1@group1` is now a child of `0@group1`. Fin-

ishing its replication, the newly created instance will send a message to its parent (with possible forwarding, until the root instance) advertising that a new network is available. Properly configured, this notification will trigger some work to do on the root node and `0@group1` will send some orders to `0@group3`, in this case, asking it to connect some nodes. The same is done for the second group. A router instance needs to periodically check its parent health. This is done through a heart-beat mechanism. In case of a potential failure of the parent node, the sub tree is terminated. In the hierarchical group tree, higher groups have a higher responsibility, if they fall, everything under them goes down. This is due to the top-down approach we use to launch nodes. A workaround would be, when a replication is done, to launch it as a daemon, make it independent of the ssh session that hosts it, and use another communication layer to connect router instances together, like Zmq per instance. This way, router instances may be able to be reconnected on a recovery mode.

Management Routers are hierarchically organized, the root instance having the ability to remotely control the other ones and is done through the use of trans-network messages. Those messages takes the form of `Makes node@group do action`. Still on the example figure 3, it can be used, for `0@group1` to make `1@group3` execute the command uptime and to be notified of its completion. A router support some high-level operations to manage groups and exploit them. Here is a non-exhaustive list of what is available :

- Self replication
- Execute a broadcast command on a managed network
- Execute a command on one node on a managed network
- Ask a remote instance to execute a registered callback
- Spawn/delete a new group
- Expand, shrink, or re-number a group
- Deliver or route messages to a node on a managed network

```
class Grid5000Test(FrameworkControler):

    def start(self, networkId):
        # After connexion
        def network_root_up(error_nodes):
            # After numbering
            def network_root_update(data):
                # After Broadcast is done
                def done(data) :
                    print("done {}".format(data))
                    self.erebor.terminate()
                    self.close()
                # Make each node in the group execute the
                # uptime command.
                # Get notified on the termination on done
                self.broadcast_exec_on("0",
                                       "uptime",
                                       consts.TRUE,
                                       "0",
                                       "root",
                                       "0",
                                       "root",
                                       done)
            # Number each node inside the root group
            self.network_update_on(consts.TRUE, "0", "root", "0", "root",
                                   network_root_update)
        # Makes the root group connect node_list nodes.
        # Get notified at the end of operation on network_root_up
        self.spawn_on("0",self.node_list, consts.FALSE, "0", "root",
                      "0", "root", network_root_up)
```

Figure 7 – Excerpt of the Router capabilities. This sample attach `self.node_list` nodes to the `root` group. After nodes bootstrap a numbering is performed. Upon the success of the numbering, a each node will execute the uptime command. When all the node have sent their results it is print on the standard out.

Communications A router instance is made to transit messages between instances. As written in example figure 3, each node on the graph has relative addresses. An address is relative because it belongs to a network. By default, inside a network, sending a message from a node to another is done by simply using the TakTuk communication layer. Trans-network routing is different, it means, getting a message out of its network and trying to find the right direction to send it. The knowledge of which node is below is easily gathered. When a child appears, it notifies its parent and this information crosses to the root node. This way, every router instance on the route is aware of the child's existence and the gateway to reach it. Taking a decision on where to send the message is done by simply looking into the list of instances. If the destination network is there, then send the message to the appropriate gateway, that will execute the same algorithm. If the destination is not in this list, then simply send the message to the father, that will apply the same algorithm. Only the root node can decide to drop a message or not. Of course, every node can send trans-network messages. But to reach their destination a message has to reach the local root first. This can be inefficient in some situations. Having a default gateway to the address 0 enable building a system where parts can communicate without a global view.

Performance Evaluation The figure 8 shows an experiment that makes the router engine replicate itself on an increasing workload from 1 to 2000 nodes by folding on 20 real ones. A replication means, bootstrapping, connecting the right amount of nodes, numbering them, starting a router instance over them, making them bootstrap an empty group, wait for the new groups to be started, and then print "Replication Finished". One measure per point is shown. Due to some perturbations, TakTuk may take more time on certain measure, those cases are outliers. Each colour represent a type of event, and a line is printed between the min and max values. When outliers appear, it seems because a node at the TakTuk layer is really slow to answer, and thus, delays the rest of the execution. The labels show different moments in the execution life cycle ; the label "Group ready" refers to the moment where TakTuk has launched and numbered the entire network ; the label "Replication Made" stands for each time a new replication is done ; the label "Replication Finished" stands for the final replication. We do not have comparison points to know if the time to replicate is good or bad. We can surely infer that the folding effect does not help us for this execution. A slave router instance managing one network is about 4 processes : the TakTuk process who launched the router child ; the router child process ; the TakTuk instance for the mastered network ; and the bridge. As we are folding, at worst, 90 times on a node, we have 360 processes dedicated to the execution. Even if they are not intensive for computations, those instances have messages to exchange and waiting for the scheduler to handle a waiting message slow down the all execution.

The label "Group ready" standing for the time where TakTuk has connected and numbered every nodes represent the same state as the one shown on figure 5. But, the value in this figure is quite less (for 1500) than for the first one. We are indeed going from 10 real nodes to 20 ones, and by reducing the folding we improve our performances, going from 5 seconds for 1500 nodes to approximately 4. This effect is not well pronounced and does not degrades the linear slope of our experiment.

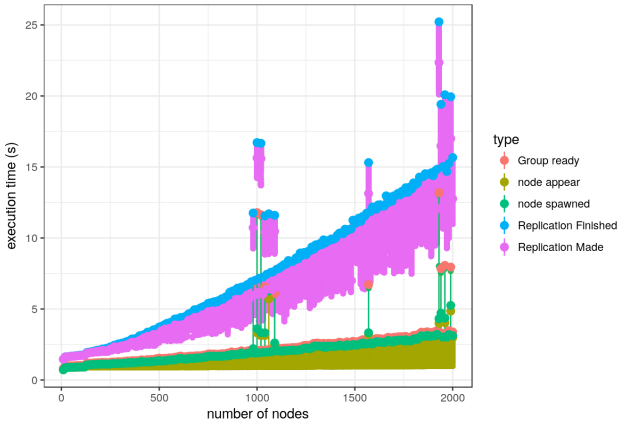


Figure 8 – Analysis of the time needed to launch a router, connect n nodes, and replicate the router on each of those nodes. This graph is showing an increasing workload from 1 to 2000 nodes folded on 20 real ones with one measure per point. The label "Replication Finished" correspond to the time measure where every replication is done.

Control Plane Communications

A router is able to route messages between networks and deploy complex topologies, at least, deep tree topologies. With the communication capabilities, any program launched by TakTuk has access to a communication layer allowing in-group and out-of-group communications. In-group are not a real challenge even if they are not to be used for sending too much information. Out-of-group communications are not used to exchange a large data volume. As messages travel through an SSH pipe and in the general case without a direct route between two nodes, we cannot expect a good throughput. Due to the user-land nature of the routing and several transfers through different independent processes, but still, providing the ability for different remotely executed programs to communicate, is the main improvement of our solution. It lets different programs to discover, organise, contact themselves, and is sufficient. For most setups once remotely executed programs along with isolation in groups, are able to establish communications, they can easily open high throughput communication channels.

MPI Use Case TakTuk provides a communication layer for each program it launches. This communicator is based on private file descriptors between TakTuk and its children processes. Those file descriptors act like a Unix socket will do. One is reserved for receiving data and the other one to send data.

This mechanism makes only a directly launched program able to send and receive messages on the TakTuk layer. What

if this program is a launcher itself ? To launch a MPI program, one uses the `mpirun` command associated with a computer list and a program to run. The launcher contacts each computer on the list and associates, for each core of each computer, a MPI process. MPI processes are pieces of codes identified with a unique ID and aiming at computing things and at exchanging information between them among different communication layers.

In this configuration, only the `mpirun` command has access to the TakTuk's file descriptors. And none of the MPI's launched process has knowledge about TakTuk's communication layer. Moreover, MPI has its own numbering mechanism. It means it is impossible for a router to predict which MPI rank will be hosted on which TakTuk's rank.

Our objective is to provide a mechanism for a MPI process from a MPI run to be able to exchange messages with another MPI process from another MPI run. Our solution provides what we call a "control plane container" to set this mechanism in place.

Control Plane Container Let's assume that we use the graph on figure 3. We are now in a configuration where `group2` and `group3` are dedicated for two distinct simulations. Each computer is an 8 core computer, which means that each of them will host 8 MPI processes. `group2` is a parallel server which gathers results from simulations. `group3` is a parallel simulation which sends its results to a known server. To make this scenario work, we define the control plane container (CPContainer) mechanism which comes in two parts : a communication daemon on each node and a C library.

The idea between the CPContainer is to provide a gateway for processes to enable them to send and receive messages. This gateway needs to be easy to find and attached to the TakTuk's network. As this gateway will allow trans-network communications, thus, it needs to embed a lot of code already written in the router code. To make the gateway easy to find, we have two solutions, having a centralised daemon for a simulation or decentralised daemon on each node. We tend to prefer the second version in order to be more scalable. For a process to communicate with the daemon, it needs an address and a port. The address is easy to recover as it is localhost and the port is known in advance. For the port, we can use Inter Process Communications (IPC) between the process and the daemon. IPC allows ports to be files and communication to be efficient. Using files to exchange data lets us generalise the file name on all the computers and avoid useless port lookup. By giving the file name relative to the execution, we can avoid conflict over folding. We are using Zmq as our communication link between the daemon and the lookup.

The daemon is simply an Erobor instance with CPContainer enabled. A daemon is setup on each node of the simulation. After bootstrap, the daemon waits for processes to notify their bootstrap sequences. Upon a registration, the daemon builds a local map between an ID and a socket to contact the associate process. Then it sends a message to its group master to notify it is in charge of this particular rank. Upon a sending request, the daemon will build a router trans-network message using the group and the ID of the destination and send it over the network. Upon a message to deliver, the group master will send the trans-network messages to the previously registered instance. The daemon on this instance will then look into its local map and route the message to the correct process. In order to setup a CPContainer, we need to contact each node in a group and replicate over them. Once it is done, he needs to start the component on each node. This operation can be time consuming, according to the evaluation performance on figure 8. We introduce a recycling functionality that flushes the routing tables of a group so they can be reused for new simulations. It provides for large case scenario a reactive way to use several times the same group.

This mechanism introduces extra routing needs. As discussed in the previous paragraph, a group leader router instance is keeping a track of where to find specific IDs. We introduced in the code the cpc-trans-network, which is an improved version of the trans-network message. Because it is always the group leader who decides where to route a trans-network message, this new routing cost seems to be negligible comparing to the previous one.

To ease the integration with existing codes we are providing a C library with a simple set of operations available, **register**, **send**, **receive**, and **close**. As we are using Zmq, it enables an advanced user to take care of the advantages of the flexibility of this library to receive messages and use advanced asynchronous functions to avoid blocking operations in the middle of the simulation code.

Performance Evaluation We want to evaluate how long a setup of a CPContainer over groups takes. The figure 8 shows the performance analysis of making a replication on each node of a group which is equivalent to setup a CPContainer.

Task processing

The router provides some functionalities and exposes itself through a Framework that eases its usage. The Framework class is a Python abstract class containing every high-level operation one can expect. Even if it is a high-level abstraction of what is really going on, its usage is cumbersome. One using it will develop again and again the same software parts to deploy computers, to number them, to make them execute programs, to gather results, and handle errors.

We would like a tool usable in DevOps use cases, for instance by allowing the easy creation of recipes, usable by workflow managers by, per instance, allowing native DAG interpretation, or by researchers for their experiments. All those fields need high-level error detection and handling. Our API should provide these functionalities. The chosen way to describe such an API is to setup a task processing library. In this library the user defines tasks that will be deployed on a router. Tasks may have dependencies upon startup and dependencies upon cancellation.

Task A Task defines a global abstraction level representing something to do. A task holds a current state which is presented on figure 9. A task can go from the initialisation state to the running or canceled one upon boolean conditions.

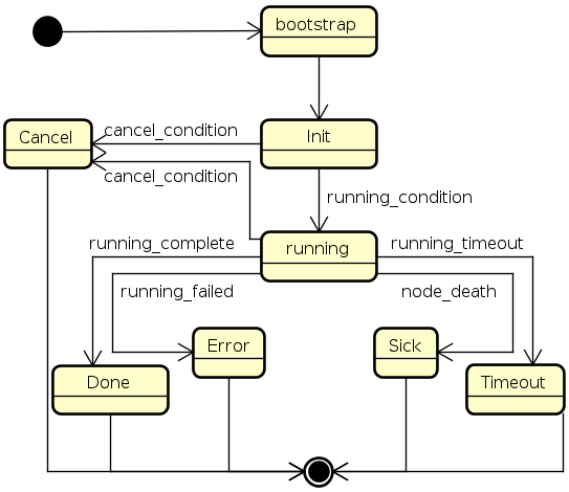


Figure 9 – Task life cycle represented as a state machine.

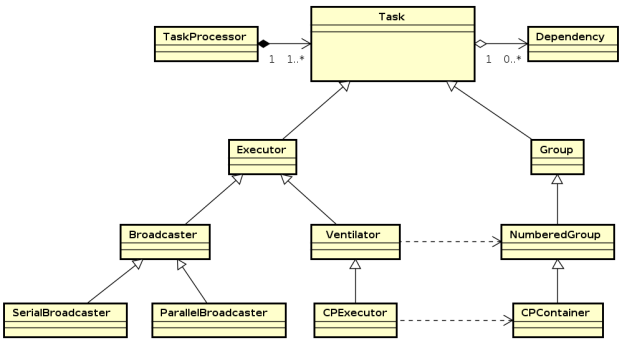


Figure 10 – UML synthesised representation of the user API.

Tasks are state machines and can be linked together to build more complex state machines. For now, the API does not take into account code validation, in the sense that, the user code may never finish. A user can write impossible states combinations, forget needed ones in case of errors, and end up with a never-ending code. Nevertheless, we can express DAG with this tasks engine. And thus be able to build, or take an existing DAG tool, to adapt over it in order to have a robust engine.

Tasks are processed by the task processor which is based on router's event loop. Tasks are registered in the initialisation state and starts to emit orders. Once the initialisation has finished, the main loop takes effect and processes incoming events. There are still no data race in the tasks processor due to the event oriented model of the router. Thus there is no ambiguity on tasks states.

The task processing library provides several task types. They are separated in two categories, tasks that will create groups and tasks which will use groups to execute something. Those types are shown in figure 10.

Group tasks are the kind of tasks launching a new group on a – newly created or not – router instance. They take a node list as parameter, start TakTuk, and contact nodes. The NumberedGroup is a Group where all nodes have a TakTuk rank.

Executor tasks exploit Group tasks in order to execute things on them. Ventilator tasks take a list of tasks to process and assign them with a first come, first served policy. Broadcasters have two flavors, one for which tasks are all executed in parallel and another in which they are executed in a sequential way.

In terms of fault tolerance, every task can fail and the programmer get notified of those failures through callbacks. He can add some code sequences to handle a failure such as relaunching the task on a recovery mode per instance by registering functions to event callbacks. In the paper, A Performance and Energy Comparison of Fault Tolerance Techniques for Exascale Computing Systems [Dauwe et al., 2016], authors introduce definitions for fault tolerances strategies :

- Rollback recovery : Periodically save the executing state and roll back in case of problems to a known state. This technique uses the concept of check-pointing.
- Redundancy : duplicate the computation.

Those two mechanisms can be implemented over our system. For the redundancy it's fast forward, one simply needs to create several groups and make them execute the same thing. For the state check-pointing and restart, the launched application needs to be developed to do it, but, if it is the case, then it's easy to attach a callback to the execution's failure to start a new task taking over from where the previous crashed.

API analysis The API does not provide an impact the performances of the underlying code. Thus its evaluation is on its usability as a high-level library. Figure 11 shows a code sample using the API. As opposed to the code sample figure 7, this does not stack callback functions and express concisely the operation made over the tree. For a second example figure 12 shows a code sample launching two programs inside CPContainers.

```
class Sample(TaskProcessor):

    def register_tasks(self) :
        # Start a new group, with computers A and B. Group leader
        # attached to the root.
        g1 = Group("g1", "g1", "A,B", "0", "root")
        g2 = NumberedGroup("g2", "g2", "A,A,A,A", "0", "root")

        # Make a simple broadcast on first group
        # b1 termination does not implies g1 termination.
        b1 = Broadcaster("b1", "uptime", g1)
        # the broadcast will start when the group g1 is in the running
        # state
        b1.add_dependency({g1:consts.RUNING}, True)

        # serial broadcast after b1 on g1, will close g1 after
        # termination
        # b2 termination implies g1 termination.
        b2 = SerialBroadcaster("b2", ["uptime", "pwd"], g1, True)
        # the broadcast will start when the broadcast b1 is DONE
        b2.add_dependency({b1:consts.DONE}, True)

        # Make a simple broadcast with a timeout of 30 seconds
        b3 = Broadcaster("b3", "sleep 1000", g2, True, 30)
        # b3 will start when g1 is done and g2 is running
        b3.add_dependency({g1:consts.DONE,
                                    g2:consts.RUNING}, True)

        # register all the generated tasks
        self.tasks.append(g1)
        self.tasks.append(g2)
        self.tasks.append(b1)
        self.tasks.append(b2)
        self.tasks.append(b3)
```

Figure 11 – Excerpt of the task library API.

```
# Compute client and server location
path        = os.environ["PWD"]
server_path = "{}/../c_wrapper/server -C {}".format(path, nb_clients)
client_path = "{}/../c_wrapper/client -R 1 -S {}".format(path, "server")

class Sample(TaskProcessor):

    def register_tasks(self) :
        nb_clients = 10

        # Create a CPContainer for the server, on a computer list
        sc = CPContainer("jail_server", 'server', 'A,B,C', '0', 'root')

        #Launch server
        server = CPExecutor(
                'sl',
                server_path,
                sc,
                True)

        # start when the cpcontainer is running
        server.add_dependency({sc:consts.RUNING}, True)
        self.tasks.append(sc)
        self.tasks.append(server)

        # Launch nb_clients Clients
        for i in range(0, nb_clients) :
            # Create a CPContainer for the client
            cc = CPContainer(
                    "jail_client{}".format(i),
                    "client{}".format(i),
                    'D,E,F',
                    '0',
                    'root')

            # Launch the client
            client = CPExecutor(
                    "mpi_executor{}".format(i),
                    client_path,
                    cc,
                    True)

            # Only start the client when the server is running and when the
            # cpcontainer is running
            client.add_dependency(
                    {cc:consts.RUNING, server:consts.RUNING}, True)
            self.tasks.append(cc)
            self.tasks.append(client)
```

Figure 12 – Setup of a CPContainer to enable two codes to exchange messages to each other. This client/server pattern will be reused on the integration of the Melissa case.

4 Integration on Melissa

4.1 Melissa

Melissa is an InSitu middleware able to create N to M communications between a range of simulations and a server. The server is gathering information about the simulations and produce statistics. Melissa is already integrated on Code_Saturne which is a complex workflow engine for simulation developed by Électricité De France.

4.2 Melissa's Communication

Melissa uses the Zmq request/reply pattern. In this pattern, a request socket is a finite state machine which cannot receive a message until it has sent one and the reply is the opposite. To open a communication between two entities and regardless of their types, one socket has to bind itself on an endpoint and the other socket has to connect to this endpoint. Notice that the binding on the network from one side is not needed for the other side to connect and a process can start to send messages to another one on an incomplete communication channel.

In the case of Melissa, the server job receives information from the clients jobs. At bootstrap, the server writes its endpoint on a file. Assuming that every computation nodes share the same file system, when a client appears, it retrieves the server endpoint with the file and establishes the communication.

4.3 Solving the Connexion Issue

As illustrated on figure 13, using a shared file system to exchange zmq's endpoint may lead to lacks of usability of Melissa. Shared file systems are not always available. This information exchange belongs to control plane between clients and the server. As our solution is providing control plane capabilities, we can use it to make Melissa work without using the file system. Melissa needs to be patched in order to be used over our solution. In this case, the patch is really small, around 30 lines of code for the client and the server. The patch works as follows. To avoid using the file system, when a client bootstrap, it sends a message to the server using router's control plane facilities. The server sends back its endpoint and the rest of the code takes place untouched.

Server Patch

The server is a MPI program with N process running the statistical analysis. The program consists of two phases. The first one is to initialise the analysis and the second one is to loop over Zmq events. MPI processes do not communicate with each other after the initialisation phase.

Our patch consists of, initialising the connection with the router in the startup phase, adding a socket to poll events list and, upon event arrival in this list, replies by providing the endpoint to the sender.

Client Patch

The client is a complex simulation code, different each time and may be really complex to modify. Luckily, Melissa provide a library for client codes and it is where the patch takes place.

As Melissa looks for a file to find the endpoint, we integrate the patch here. Our patch does the following, if the file is not found, initialise the connection to the router, send a message to the ID 0 of the server group and wait for the endpoint to come as an answer. Then continue the former code untouched.

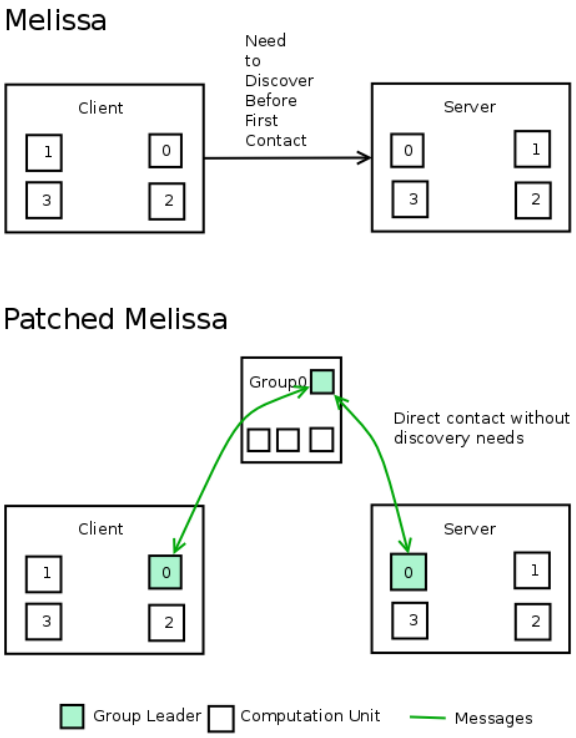


Figure 13 – melissa

5 Discussion and Further work

Some design choices can be discussed regarding their performance impact on simulations or their practicability.

5.1 Simulation Perturbations

Our solution execute some daemons on each remote node. A non-root router needs 2 processes to run plus 2 processes per mastered network. Within a network, each computer embeds at least one TakTuk daemon. In the best-case scenario, with a mostly silent simulation – on stdout, stderr – and a only a simple control plane usage through our communication library, those processes will be idle most of the time and will have a small footprint on running simulations. If the simulation outputs a lot, as each line is gathered to the router instance, it will suffer from CPU and memory usage and thus may result in a certain slow down. Further exploration regarding the impact needs to be settled in a future experimentation plan.

5.2 Linux Oriented

As we use Python, SSH, and Perl we aim at taking control of a computation node, our solution cannot be deployed on super computers like The Blue Gene machine. As we are using epoll, a state of the art event oriented IO notification system call, this solution only runs on Linux kernels. As Linux has become the default choice for HPC centers, this decision can be defended. Also work can be done to make a FreeBsd ready implementation using the kqueues.

5.3 Auto Propagation

Because of the only dependency on Perl and SSH, TakTuk is able to pipe itself on a remote Perl interpreter. This makes it a very portable tool. Our solution uses Python3 and requires some libraries to run. There is an ongoing work about the self-replication of a router instance. The question is less simple than for TakTuk. The naïve way is to copy every needed file in the temporary folder and then to execute. This solution may be a bit time consuming and we would like to find a way to pipe a router instance in the remote Python interpreter.

5.4 Reliability

Because a router uses TakTuk to spread itself on the computers, the lifetime of a router instance is dependent of its parent lifetime. This may be a problem for scalability. We are choosing this distributed approach in order to scale better and partition resources, but with Exascale platforms, failure will become more and more present. Our strategy may fail easily if a node failure on the top of the tree mean a half-deployment shutdown.

A workaround to this problem is to deploy non-root routers as autonomous daemons and to isolate those from their TakTuk source, by making them discuss over another communication layer, Zmq for instance. This way a parent failure will not impact a child lifetime anymore. This may open a way for dynamic re-routing inside the tree.

This said, as there are fewer nodes at the top of the tree than at the bottom, there is a small probability of failure and retaking over failure may be very complex to setup.

5.5 Communication Bottleneck

Our solution deploys a hierarchic tree with message routing capabilities between sub-trees. Routing is done by a router nodes among the tree and it is done at the application layer of the OSI model. To be routed, a message goes from ssh to the TakTuk daemon, is given to the Python bridge that makes an IPC to the router instance, then the router instance takes the routing decision and forwards the message to another bridge that will ask another TakTuk instance to send the message over ssh.

The more a message has gateways to pass before a destination, the more time-consuming it will be for the whole tree. Developers using the control plane capabilities should try to minimise the distance between a source and a destination. An extreme case where all applications from the left side of a tree communicating with applications on the right side of the tree will bring a terrible overhead to the root node. Thus, more detailed exploration of those overhead will be a future work.

5.6 Extra Parameters

We need to test our Melissa patch on large scale platforms. An ongoing work is the implementation over and the implementation should be fast forward. But in fact, as we need extra parameters, we have issues with Code_Saturne and we need to find a way to overcome this problem.

Those parameters allow a simulation to recover its router's endpoint. The real problem and the reason why parameters are given to simulation is to be folding resistant. In real world executions, we can expect only one simulation type per computing node. On folding cases, many router daemons are deployed on a same computer. To find its own, a simulation needs an address to talk to. We are currently working on this solution to have something folding resistant and with the least less possible impact.

5.7 Performance Analysis

This work only reports basic performance analysis. This shows that in its minimal configuration a router instance is almost as fast as TakTuk with a constant overhead. Experiments needs to be settled to study the impact on the startup of a simulation and this on a wide variety of machines.

6 Conclusion

With the Exascale approaching, because of the increasing complexity and failure ration of such platform, there are needs for scalable execution managers offering strong feedback over their execution plan, also Insitu applications wanting to exploit those platform may need failure resistant ways for its subparts to connect to each others.

In this report we have presented the construction of a scalable execution manager. A tool able to manage large sets of computational units, to provide resources isolation, and control plane capabilities between groups of computation units. Using TakTuk as the underlying remote execution control tool, each group can possess hundreds of thousands of computations units. Experiment shows that the overhead of our solution is constant over TakTuk and we were able to predict with linear regression its execution time. Those predictions actually are matching experiments made on the Distem benchmark. Having a scalable group unit enable stacking groups, thus, building hierarchy to a great scalability. Having a group hierarchy helps failure resistance, by having a constant feedback among the tree. One having fine-grained control on the execution and to apply failure resistant strategies such as check-pointing or redundancy. And thanks to the permanent node control, we can gather information and keep track of the state of every computation unit. A high-level API allow DevOps or workflow managers to use our tasks execution model to plug our solution under jobs/tasks schedulers, providing a way to implement DAG or DSL interpretation. As our solution is platform independent, specific platform support can be developed on top of it. OAR already use TakTuk to manage its platform and we could be able to uses advances made on Execo to provide plugins to our solution in order to control OAR or Slurm.

Finally, we are able to provide to the InSitu computation world with a way to implement control plane communications, opening a road to enable for two remotely executed programs to discover themselves and open high-speed communication pipes. To validate our approach we've patched the Melissa library, an InSitu framework for parametric studies analysis. Even without a complete run due to late adaptation on Melissa's code, we are able to launch simulations and to bring parts together in a more reliable and cleaner way than Melissa does. Table 2 provides a view of functionalities listed in the table 1, highlighting those that are embedded in our solution and those that can be built on top of our solution.

In collaboration with the Argonne National Laboratory, we will continue this work with integration on their internal project. And also this solution as been presented to the Joint Laboratories For ExaScale Computing (JLESC) that tooks place in July 2017 University of Illinois at Urbana-Champaign USA. This was an opportunity to find research collaborations and especially diverse uses case where our solution may help.

This work has been finished at the Argonne National Laboratory for some weeks in July 2017. It was an opportunity to meet PETS-C developers and to discuss with them possible adaptations of their code around our solution.

functionality	Can be build on top	Embded
DSL Support	X	
DAG Interpretation	X	
Job/Task Scheduling	X	
Results Analysis	X	
Input Validation	X	
Platforms support	X	
Ressources discovery	X	
Provenance tracking	X	
Checkpointing	X	
Verification and configuration	X	
Interactive Execution		X
Remote execution		X
Ressource isolation (Groups)		X
InSitu via control plane		X

Table 2 – Functionalities ready or buildable over ou solution.

References

[Ahn et al., 2014] Ahn, D. H., Garlick, J., Grondona, M., Lipari, D., Springmeyer, B., and Schulz, M. (2014). Flux: A next-generation resource management framework for large hpc centers. In *Parallel Processing Workshops (ICCPW), 2014 43rd International Conference on*, pages 9–17. IEEE.

[Buchert et al., 2014] Buchert, T., Jeanvoine, E., and Nussbaum, L. (2014). Emulation at very large scale with distem. In *Cluster, Cloud and Grid Computing (CCGrid), 2014 14th IEEE/ACM International Symposium on*, pages 933–936. IEEE.

[Buchert et al., 2015] Buchert, T., Ruiz, C., Nussbaum, L., and Richard, O. (2015). A survey of general-purpose experiment management tools for distributed systems. *Future Generation Computer Systems*, 45:1–12.

[Capit et al., 2005] Capit, N., Da Costa, G., Georgiou, Y., Huard, G., Martin, C., Mounié, G., Neyron, P., and Richard, O. (2005). A batch scheduler with high level components. In *Cluster Computing and the Grid, 2005. CCGrid 2005. IEEE International Symposium on*, volume 2, pages 776–783. IEEE.

[Claudel et al., 2009] Claudel, B., Huard, G., and Richard, O. (2009). Taktuk, adaptive deployment of remote executions. In *Proceedings of the 18th ACM international symposium on High performance distributed computing*, pages 91–100. ACM.

[Dauwe et al., 2016] Dauwe, D., Pasricha, S., Maciejewski, A. A., and Siegel, H. J. (2016). A performance and energy comparison of fault tolerance techniques for exascale computing systems. pages 436–443.

[Deelman et al., 2009] Deelman, E., Gannon, D., Shields, M., and Taylor, I. (2009). Workflows and e-science: An overview of workflow system features and capabilities. *Future Generation Computer Systems*, 25(5):528–540.

[Deelman et al., 2015] Deelman, E., Vahi, K., Juve, G., Rynge, M., Callaghan, S., Maechling, P. J., Mayani, R., Chen, W., da Silva, R. F., Livny, M., et al. (2015). Pegasus, a workflow management system for science automation. *Future Generation Computer Systems*, 46:17–35.

[Dreher and Peterka, 2016] Dreher, M. and Peterka, T. (2016). Bredala: Semantic data redistribution for in situ applications. In *Cluster Computing (CLUSTER), 2016 IEEE International Conference on*, pages 279–288. IEEE.

[Ebert et al., 2016] Ebert, C., Gallardo, G., Hernantes, J., and Serrano, N. (2016). Devops. *IEEE Software*, 33(3):94–100.

[Hintjens, 2011] Hintjens, P. (2011). Ømq-the guide. *Online: http://zguide. zeromq. org/page: all, Accessed on*, 23.

[Johann, 2017] Johann, S. (2017). Kief morris on infrastructure as code. *IEEE Software*, 34(1):117–120.

[Liu et al., 2016] Liu, F. C., Shen, F., Chau, D. H., Bright, N., and Belgin, M. (2016). Building a research data science platform from industrial machines. In *Big Data (Big Data), 2016 IEEE International Conference on*, pages 2270–2275. IEEE.

[Liu et al., 2015] Liu, J., Pacitti, E., Valduriez, P., and Mattoso, M. (2015). A survey of data-intensive scientific workflow management. *Journal of Grid Computing*, 13(4):457–493.

[Stanisic et al., 2015] Stanisic, L., Legrand, A., and Danjean, V. (2015). An effective git and org-mode based workflow for reproducible research. *ACM SIGOPS Operating Systems Review*, 49(1):61–70.

[Terraz et al., 2016] Terraz, T., Raffin, B., Ribes, A., and Fournier, Y. (2016). In situ statistical analysis for parametric studies. In *Proceedings of the 2nd Workshop on In Situ Infrastructures for Enabling Extreme-scale Analysis and Visualization*, pages 35–39. IEEE Press.

[Thiell et al., 2012] Thiell, S., Degrémont, A., Doreau, H., and Cedeyn, A. (2012). Clustershell, a scalable execution framework for parallel tasks. In *Linux Symposium*, page 77. Citeseer.

[Yoo et al., 2003] Yoo, A. B., Jette, M. A., and Grondona, M. (2003). Slurm: Simple linux utility for resource management. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 44–60. Springer.

[Zhu et al., 2016] Zhu, L., Bass, L., and Champlin-Scharff, G. (2016). Devops and its practices. *IEEE Software*, 33(3):32–34.

Formal verification of a new cache coherence protocol

Steve Roques Supervised by: Frédéric Pétrot
steve.roques@univ-grenoble-alpes.fr, frederic.petrot@univ-grenoble-alpes.fr

Abstract

The aim of this work is to perform formal verification of cache coherence protocols for multiprocessor machines. We want to verify certain properties of these protocols, such as the absence of deadlocks or the exclusivity of certain data. To formally carry out this verification, we use automatic theorem proving by formalizing protocols and properties to check. We apply this technique to two protocols, namely the MESI and H-NUCA protocols. The latter is a new protocol based on the MESI protocol, but differs from it by a diffusion phase which can lead to deadlocks. We verify these two protocols by exploiting their similarities and checking that the differences do not violate the properties to guarantee consistency. We limit ourselves to the level of abstraction representing the protocols state machine to remain independent of the architectural details. This choice does not allow to verify all the implications of certain properties and obliges us to define hypotheses which will have to be verified at the level of the implantation. It takes a lot of meticulousness and thoroughness to verify that a property is correct. Not only does the formalization of the property require a good knowledge of the protocol and the domain of existence of the property but also the generation of hypotheses to be verified at implantation level requires particular attention. Indeed, it is not necessary to generate hypotheses that are false or not verifiable at low level without which the property can not be verified correctly. Under this condition, we achieved the verification of the two protocols.

1 Introduction and presentation

Nowadays, the hardware processing devices follow two trends. On one hand, we have thousands of heavily synchronized single-instruction multiple-data (SIMD) cores, representative of the GPU type of processing, that are well suited to applications in which data is essentially private to each core. On the other hand, we have tens to hundreds of general-purpose processing cores, sharing a single memory, appropriate for the ”usual” posix-like programming model. In this work, we are interested by the latter type of architectures, in which the access to the memory is shared and coherent. To hide memory access latencies, using a hierarchy of caches, i.e. relatively small hardware controlled memories that contain copies of the data in the main memory, is mandatory. We take as example the Tilera-GX architecture [1], which embeds 72 processor cores, each having a L1 cache and sharing several L2. Figure 1 shows the typical organization of such a manycore architecture. Although necessary to limit latency,

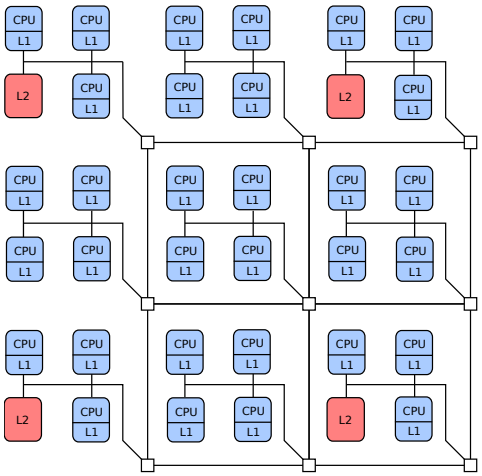


Figure 1: A typical manycore architecture

having multiple copies of a given data leads to incoherence when the write of a new value occurs, unless a *cache coherence protocol* is used. Our focus in this work is to formally specify and verify properties of cache coherence protocols.

Our motivation comes from a study on optimized use of memory resources in integrated multi-core processors, in which we have been led to define a new cache coherence protocol which avoids duplication of data in upper cache levels at the price of a higher latency when searching for data. This protocol basically exploits the MESI protocol (the cache coherence protocol used in most systems today), but requires special phases of search in all upper level caches by diffusion, which is a new approach. The protocol is completely defined

and implemented as a state machine in the SLICC language, which is an ad hoc specialized language used to describe protocols [2]. It has been validated by testing (boot of Linux SMP on 16 cores, and applications of the SPLASH2 benchmark), but absence of problems (races between accesses leading to dead locks or live locks or to erroneous results) is simply hoped for. The objective of this work is to formally validate the protocol. We have several techniques at our disposal to do this. The first one is Model checking that allows to do static analysis of the protocol, second one is Temporal logic which is also a static analysis technique, but based on traces of a given execution and the last one is Formal verification by theorem proving using a proof assistant.

Gerard J. Holzmann [3] describes processes for design and especially validation of protocols. This textbook mainly addresses communication protocols because at that time they were being widely deployed in all fields of computer science. Nevertheless, since then the general approaches to protocol validation in general did not change much: model checking, theorem proving and test suite are advocated. New combinations of these techniques were proposed to reinforce confidence in the validation and also to corroborate results of different techniques. Indeed, it is possible to process parts of some protocol with an automatic technique and another with a more formal technique. Gerard J. Holzmann [3] begins by describing conformance testing this way (section 9):

- "*A conformance test is used to check that external behavior of a given implementation of a protocol is equivalent to its formal specification.*"
- "*A validation is used to check that formal specification itself is logically consistent.*"

Holzmann separates functional testing from structural testing which allows to differentiate tests which makes it possible to verify behavior of a protocol which then makes it possible to verify its structure. In section 11, it deals with validation of protocol by manual and automated methods. Holzmann separates verification by formalization and verification by model checking. Although these two techniques still exist and are always differentiated, it is possible to find in literature more and more confusions between the two as formalization comes back to create a mathematical model. So it is normal that formal verification encompasses both these techniques. In addition, there are numerous techniques that allow to analyze a program or a protocol by combining several different techniques such as testing, formal verification and temporal analysis. Several of these papers are presented section 4 of this report.

In a multiprocessor system, having multiple copies in different places makes it challenging to maintain a consistent state of the values as seen by the program threads. The programmer expects (somewhat naïvely), that any read on any processor at a given address will return the last value written by any processor at this same address. Due to race conditions on the exact instant at which writes or read occur, the problem is usually simplified so that the read value will *eventually* contain the value written.

Cache coherence protocols address this issue, and propose different solutions to ensure coherence between the values of the copies. The solutions vary in latency, required bandwidth, number of messages exchanged, number of bits necessary to maintain per block status, and implementation complexity.

Validation and verification of cache coherence protocol is the purpose of this work. To achieve this several techniques are possible like model checking, runtime verification or theorem proving. It is also necessary to decide on a level of abstraction to be studied. Do we want to validate low-level implementation of a protocol which is highly dependent on used architecture or keep a higher level of abstraction in order to verify its general behavior?

We focus on the level of abstraction corresponding to the implementation of protocols in SLICC. Indeed this language, which will be described later, is a cache coherence protocol description language and this allows validating the behavior of a protocol without modifications by the compiler or even low level implementation dependent on the architecture. It is obviously important to validate implementation of these protocols at low level in order to verify that the final implementation corresponds to description of the protocol in SLICC. However, at SLICC level some properties are fixed and allow to set up the general behavior of protocols as well as state machines that will be executed on each cache line according to the cache level (L1, L2 or L3). Thus SLICC descriptions contain FSMs, request and response messages that are exchanged between components and a first level of implementation of the general architecture. It is what will be analyzed and checked in this work. For the current research, we study a hierarchy with two cache levels and a memory that can be seen as the L3 cache. We will focus on the cache coherence protocols dealing with the two levels L1 and L2.

Then in section 2 we will describe SLICC language used by two protocols namely MESI two level provided in simulator gem5 and H-NUCA based on MESI. Section 3 will deal with our contributions, namely mechanisms put in place to realize proofs of these protocols. We provide a state of the art on program and protocol verification in section 4 and finally we will conclude and enumerate possible future work in section 5.

2 Description of two cache coherence protocols

In this section, we detail the SLICC language in which the existing MESI and the new protocol named Hybrid Non Uniform Cache Access (H-NUCA) are written. Also, we describe each protocol and outline the main differences between them.

2.1 SLICC language

SLICC[2] is a domain specific language for specifying cache coherence protocols. SLICC is syntactically similar to C or C++, but it is intentionally limited to the specification

of hardware-like structures[4]. It is based on specifying individual controller state machines that represent system components such as cache controllers and directory controllers. Each controller is conceptually a per-memory-block state machine, which includes:

- States: set of possible states for each cache block,
- Events: conditions that trigger state transition, such as message arrivals,
- Transitions: cross-product of states and events (based on state and event, a transition performs an atomic sequence of actions and changes a block to a new state),
- Action: specific operations performed during a transition.

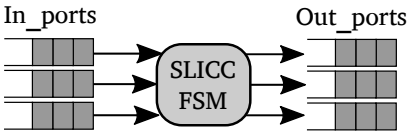


Figure 2: SLICC FSM

Each controller specified in SLICC consists of protocol-independent components, such as cache memories and directories[4]. MESI and H-NUCA protocols are written in SLICC. So we use this language as a point of departure for all our tools. At this level of abstraction, we skip the compilation phase necessary to produce a model executable by the Gem5 simulator and we concentrate on protocol specifications.

SLICC language contains, among others, the keywords necessary to specify the different parts of a finite state machine (FSM). All FSMs receive request-response messages from the network in_ports and send their messages on out_ports. Each message is pushed on a stack and the FSM uses actions described in transitions to process messages and to produce again, if necessary, messages to output ports to communicate with the rest of network. This language produces C++ that can be compiled and linked to libraries to run on the Gem5 simulator. We use only description written in SLICC because we want to verify some properties at this abstract level. Indeed, we want to evaluate certain properties, deadlock freedom or state uniqueness for example, directly at level of abstraction available in SLICC.

2.2 MESI Two Level protocol

The concept of MESI protocol, see Fig. 3, was introduced in 1984 by Papamarcos and Patel [5]. Its detailed modeling in SLICC, for the level 1 and level 2 caches is due to Binkert et al. [6].

This cache coherence protocol contains 4 states described below:

- Modified M : cache block has been modified, its value is different from the one in main memory and it is the only cached copy.
- Exclusive E : cache block is identical to what is in main memory and it is the only cached copy.

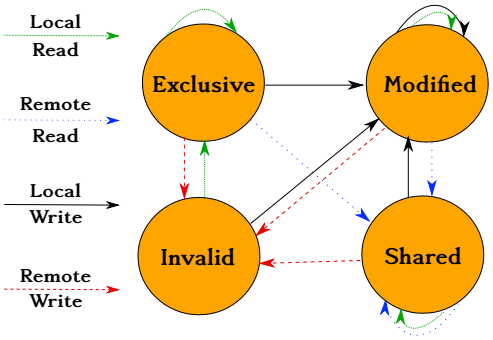


Figure 3: Basic MESI protocol

- Shared S : same value as main memory but copies may exist in other caches.
- Invalid I : cache block data is not valid.

The variant that we use is called "two levels", so 4 stable states of MESI are present in the automaton of the L1 cache and we have a second automaton for cache L2. Fig. 4 is a simplified view of the L1 cache FSM: the prefetch actions have been removed, and transitions between states are merged so as not to overload visualization.

In this FSM, NP state means that data is not present, this state is weakly equivalent to state I and it is used for initialization of cache block when Gem5 instantiates caches written in SLICC. All other states present in this FSM are transient states between the stable states of MESI. These transient states allow to indicate that the cached datum is in a certain state and is in the process of being changed of state. Thus, if the datum is accessed by the processor, the validity of this datum is indeed decidable. If this datum is in a stable state of MESI, we can take a decision of validity, but if it is in a transient state, we have to wait until the block states reaches a stable state to take a decision.

2.3 H-NUCA protocol based on MESI Two Level

Hybrid Non Uniform Cache Access (H-NUCA) is a new protocol based on MESI Two Level and created by Hela Belhadj Amor in the context of her thesis. The L1 cache of the H-NUCA protocol is similar to MESI Two levels. On other hand, the L2 cache is reworked and restructured. Indeed, in case of MESI, the memory is partitioned in segments, and each L2 cache is assigned a segment and only one, and caches a subset of that segment. So there is no data redundancy at this level. If two L1 caches request the same data, then it will be same cache L2 which will deliver it. In case of H-NUCA we have a network of L2 caches, a data can be in only one cache, but it can be in any cache. This requires dialogs between the L2s and exchange of information. See [7] for more details on this protocol.

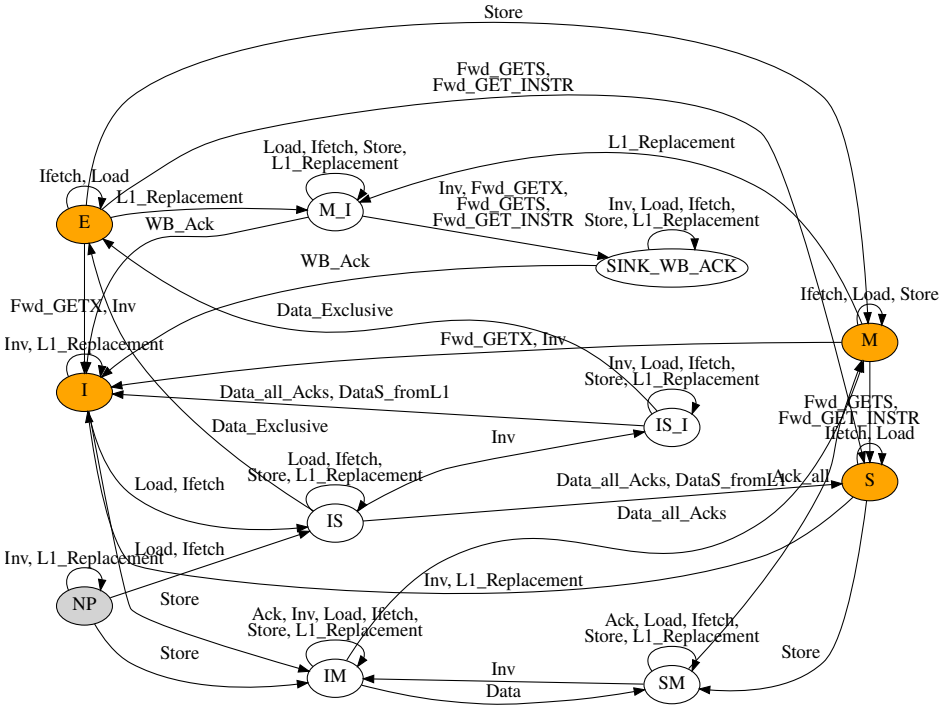


Figure 4: MESI protocol SLICC simplified representation

Non Uniform Cache Access (NUCA) has been described by Kim et al. [8]. These designs embed a network into the cache itself, allowing data to migrate within cache, clustering the working set in cache region nearest to the processor that makes use of it. Usually, there are two NUCA variants. S-NUCA for static NUCA, a conservative approach that reduces use of network and energy at expense of performance. D-NUCA for dynamic NUCA, which negotiates use of network and energy consumption for higher performances. H-NUCA is a new form where H stands for hybrid. It seeks to achieve two goals that are firstly to exploit efficiently the amount of internal memory and secondly to ensure low latency for these accesses.

H-NUCA is based on a point-to-point interconnection network and has a memory hierarchy at several levels, just like MESI Two Levels. Usually for systems based on this kind of network there are two approaches to maintaining cache coherence. *Source snoop* where a source diffuses a request to all caches of the system. Second is *home-based directory snoop* associated with the memory bank containing the cache block. H-NUCA is based on an hybrid approach. It consists of making a broadcast to the other L2s if the consulted *home* cache does not cache the data.

These visualizations make it possible to observe differences at each level with MESI protocol but also the similarities. Indeed, the automata corresponding to each level have same structures but that of H-NUCA protocol contains again eventually as Ack_H or even Ack_all_up. These events are therefore specific to the H-NUCA protocol, which will necessitate the generation of properties corresponding to them and specific to this protocol.

3 Proving the two protocols

3.1 Methodology and tools

We are therefore seeking the verification of the protocols described in SLICC by Theorem Proving. The protocol verification consists of listing and formalizing the properties we wish to verify on the protocol. If a property is not verified, it means that either the property is not true, or the protocol does not verify the property, or the formalization of one of them or both is not correct or is incomplete. Among the properties that we wish to check, the absence of deadlock, i.e. the protocol does not contain any possibility of execution leading to a deadlock, is a major one. We take it as an example. To verify it, we formalize what is a deadlock and we formalize the protocol using the same formalism: COQ in our case. Then it is enough to formalize that there is no presence of deadlock in our protocol and finally we must verify that this property is correct. We use the COQ proof assistant to verify this proof. We do the same for all properties that we wish to verify then we can make our verdicts. If all the properties are correct this means that the protocols are correct on the set of properties stated.

The formalization of the protocol itself consists in translating the representation of the protocol, here expressed in the SLICC language, towards a formalism that allows to realize

the formal proof of the properties. We use COQ because it allows some automation of the evidence by the tactics it provides. Thus we use the same formalism between that of the protocol and that of the properties to check and we can therefore realize the proofs easily using these formalizations. As far as the protocols are concerned, the translation can be automated. So we first parse the SLICC implementation into the form of an abstract syntax tree (AST). Then we translate the AST to a representation accessible in COQ.

Figure 5 shows the approach we propose, that includes the **slicc2coq** compiler, the tool that performs the compilation of the SLICC files in COQ. In addition to the automatically generated files, we can see that other, external, files written in COQ are added. These files contain the formalized descriptions of the properties we want to verify, and are written by an expert. A second library that allows to manipulate the nodes of the ASTs was created to facilitate the proofs of properties. Finally, we take the COQ files generated and the files containing the properties and proofs and use the COQ analyzer to provide a verdict. If the proofs are correct this means that the protocols verifies the properties and therefore that it is correct with respect to these properties. Otherwise, the COQ proof wizard shows which properties are not checked. This means either that the property is not respected by the protocol or that the property is badly formulated and does not match what happens in the protocol. In both cases, attention must be paid to these properties and the formalization must be refined if necessary.

All the tools have been written in the **Objective Caml** functional language. Compilation is composed of two parts, a SLICC parser as front end to parse the protocol description and create the corresponding AST, and a back end which uses the AST to generate a desired implementation in a new language, in our case COQ and graphViz.

A SLICC program consists of several modules and in order to maintain a certain readability, we have to keep the hierarchy of modules. So if a SLICC program contains n separate modules in a file, we will generate n separate ASTs. The generated code in COQ or graphViz respects the hierarchy of elements, and we therefore have a direct mapping between original sources files and generated files. It should be noted that in the graphViz back ends, which produces graphs to visualize finite state machines (FSM), only modules that contain FSM are generated. So if the original sources consists of several files but only a small part of them describe state machines, then only these files will be translated by the compiler and the other ones will be ignored.

The listing 1 contains a part of the grammar used for declaring types like Machines, Actions and Transitions. The complete grammar is available in $slicc\_parser.mly$ file.

The COQ back end of **slicc2coq** generates the AST in COQ syntax. For each file, we create a COQ implementation with an AST written on COQ. Then the proof file is created with properties that are to be verified. Finally we compile the COQ module and the compilation verifies if all properties have been proved correctly.

A second tool **slicc2gv**, translates the SLICC implementation to a Graphviz representation for the FSM of given protocol. Graphviz is an open source graph visualization software. Slicc2gv traverses the AST and translates all states as nodes and all transitions as edges of a directed graph. This tool is useful to represent FSMs and visualize the behavior of a protocol.

```
1  MACHINE LPAREN enumeration pairs RPAREN
       COLON obj_decls
2    LBRACE decls RBRACE
3    {MachineAST($3, $4, $7, $9)}
4  | MACHINE LPAREN ident pairs RPAREN COLON
       obj_decls
5    LBRACE decls RBRACE
6    {MachineAST((("MachineType", snd $3),
         $3), $4, $7, $9)}
7  | ACTION LPAREN ident pairs RPAREN
     statements
8    {ActionAST($3, $4, $6)}
9  | IN_PORT LPAREN ident COMMA itype COMMA
     var pairs RPAREN
10   statements
11   {In_portAST($3, $5, $7, $8, $10)}
12 | OUT_PORT LPAREN ident COMMA itype,
     COMMA var pairs RPAREN SEMI
13   {Out_portAST($3, $5, $7, $8)}
14 | TRANS LPAREN idents COMMA idents COMMA
     ident_or_star RPAREN
15   idents
16   {TransitionAST($3, $5, Some($7), $9)}
17 | TRANS LPAREN idents COMMA idents RPAREN
       idents
18   {TransitionAST($3, $5, None, $7)}
19 | TRANS LPAREN idents COMMA idents COMMA
     ident_or_star RPAREN
20   idents idents
21   {TransitionAST($3, $5, Some($7), $10)
       }
22 | TRANS LPAREN idents COMMA idents RPAREN
       idents idents
23   {TransitionAST($3, $5, None, $8)}
24 | EXTERN_TYPE LPAREN itype pairs RPAREN
     SEMI
25   {ExternTypeAST($3, $4, [])}
26 | GLOBAL LPAREN itype pairs RPAREN LBRACE
       type_members RBRACE
27   {TypeAST($3, $4, $7)}
28 | STRUCT LPAREN itype pairs RPAREN LBRACE
       type_members RBRACE
29   {TypeAST($3, $4, $7)}
30 | ENUM LPAREN itype pairs RPAREN LBRACE
     type_enums RBRACE
31   {EnumAST($3, $4, $7)}
32 | STATE_DECL LPAREN itype pairs RPAREN
     LBRACE type_states RBRACE
33   {StateAST($3, $4, $7)}
34 | type_member {TypeMemberAST($1)}
```

Listing 1: SLICC grammar declaration types

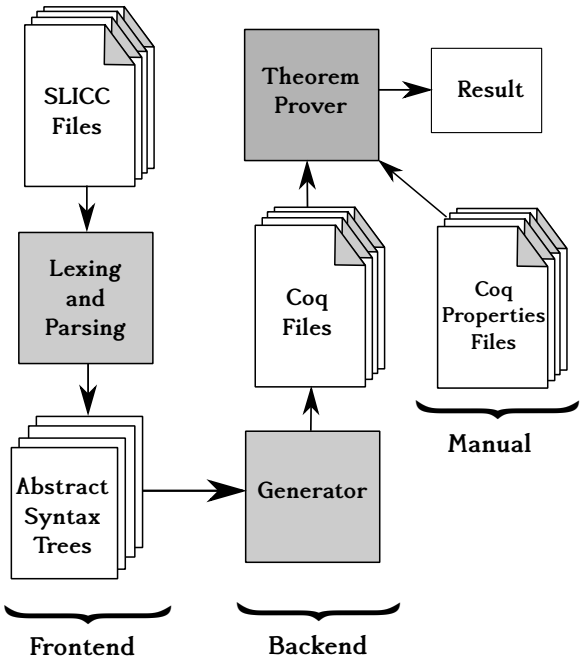


Figure 5: Slicc2Coq Compiler

3.2 Formal verification with the COQ proof assistant

Slicc2coq generates a COQ implementation of an AST. All nodes types are inductive types, which allows to easily manipulate AST and extract information quickly. For instance, a transition will have as type "$Transition\ of\ list\ ident * list\ ident * option\ ident * list\ ident$" where as we had previously presented, a transition contains a list of input states, a list of possible events, an optional arrival state and a list of actions.

As we said in the introduction we wish to remain at the level of abstraction of SLICC without memory addresses representation. Thus the ASTs written in COQ and generated by slicc2coq tool contains only the corresponding to SLICC representation without memory addresses mecanism. This leads to focus strongly on constrains properties that we wish to prove without the overhead of implementation. Indeed, if a high level property needs a property available at low level of implementation it will either have to admit the latter one by hypothesis or try to construct an abstraction mechanism that reflects the model low level behavior. At first we will admit that necessary low level properties are available and true, and this is our first strong hypothesis. Indeed, SLICC uses lower abstractions that are assumed to "behave as expected", and this can be verified only with a low-level execution check. In order not to repeat this assertion for each hypothesis generated, it is necessary to assume these hypothesis and consider at the SLICC level that the properties are correct if and only if the hypothesis are verified in low level implementation.

To begin with, we verify determinism of FSMs, which facilitates understanding the behavior of protocols and ensures that only transitions described have an effect on the execution of the program and that any path in the automaton is reachable in a distinct and unambiguous way. To verify determinism of a FSM we use following theorem.

Property 1 *An FSM is deterministic if there exists no two transitions that comes out of the same state with the same input words.*

```
1 Require Import Get_Set_Slicc
      MESI_Two_Level_L1cache_sm.
2 Definition transitionListL1 :=
3   getTransitionList
        MESI_Two_Level_L1cache_sm.body.
4 Lemma L1isDeterminist : isDeterminist
      transitionListL1.
5   simpl. (* simplify the call of
        isDeterminist *)
6   intuition. (* solve the SAT clauses *)
7 Qed.
```

Listing 2: Cache L1 is deterministic proof example

It is enough to use machine description, or more precisely its corresponding AST in COQ, to check if this theorem is true for our protocol. We will go through the list of transitions of the protocol and verify that determinism is ensured. Concerning completeness, we assume that missing transitions are so-called inert transitions. That is to say that we remain in same state (i.e. there is an implicit self-loop arc on which all missing conditions are or-ed) and also that the actions have no effect on environment. We can convince ourselves that this hypothesis is verified by observing the code generated by the gem5 compiler. Indeed transitions that are not described in SLICC are not generated and therefore the program will do nothing on these missing transitions.

Listing 3 contains all the necessary functions to check if a FSM is deterministic. The functions are implemented in COQ and are used as follows. The "isDeterminist" function takes a transition list present in the FSM. This function verifies the orthogonality of these transitions, i.e. in a given state there are no two distinct transitions having the same input actions. The "isDeterminist" function uses the "itexist" function to check that a transition is not present in a list. To verify that the L1 cache is deterministic we will retrieve the corresponding list of transitions and then define the lemma L1isDeterminist as in Listing 2. This lemma calls the "isDeterminist" function with the L1 cache transition list. The tactic *simpl* makes it possible to simplify this call and thus to obtain the conjunction of the clauses to be satisfied. The tactic *intuition* allows here to solve these clauses. If the clauses are satisfied, as here, this means that the list of the transitions of the L1 cache is deterministic and therefore that the FSM of the L1 cache is also deterministic.

Property 2 *The execution of the set of FSMs must not lead to deadlocks or livelocks, that is to say all cache line FSMs must be lock free.*

To check for deadlock during the execution of the protocol, it is necessary to analyze all the accesses in critical subsection and the dependencies between the processes with respect to the taking of locks. The only nodes of the ASTs that contain such a possibility are the In_ports and Out_ports, to indeed add a message of request (or response) in these files requires a request of atomicity so that the message is neither disturbed nor forgotten by any interuption. Now at the level of the SLICC there is no lock instruction to guarantee that these accesses are atomic. This therefore creates the hypothesis that at the level of the SLICC the input and output ports have atomic access and that the mechanism of atomicity prevents deadlock and livelock. This time again we must check this property at the implementation level because at the level of the SLICC it is impossible to verify it.

3.3 Verification of MESI two level

Following are specific propreties of MESI two levels. These properties concern only the analysis of the MESI and not of H-NUCA or another protocol. This shows that there remains a part of the properties that is not generalizable because they are specific to the protocol. Some of these properties may be a concrete property of the system or a specific part of a general property. Thus the properties which deals with the coherence of the data in case of the MESI can be the property that verifies the uniqueness for a given data to be in the exclusive state E. This property is specific to the MESI protocol but makes it possible to guarantee a more general property that the data in the caches are coherente and warrants share the cache coherence protocol. Let us therefore begin this part by the properties of the MESI.

Property 3 *For all data in a L1 cache line and in the E state, no other copy of this data exists in other caches in lines with a stable state (M, E or S).*

In this case, stable state is one of the 3 stable states of the MESI (M, E or S) for the L1 cache. Invalid state I is particular because if the data is present and is in the invalid state then the property holds because the copy is considered irrelevant in this state. Indeed the exclusive state E implies that only the cache line being in this state contains the data and that any other copy of this data that would still be present in the caches is not clean or not terminal. To verify this, it is enough to traverse the list of possible states present in the FSM of the L1 cache. Then for any intermediate state it is sufficient to verify if this state returns in the Invalid state in which case the property holds. Otherwise it must be ensured that the execution of the protocol will not lead to another stable state. To verify that no state leads to a stable state we verify for all the states of the graph whether it is possible to arrive in one of these states without going through the state of invalidation or not. As we have just said, state I is the state of invalidation of the given data considered as a state of start of the unfolding of the automaton. So any passage through this state allows to complete a cycle and start a new one. We therefore consider that if a path passes through I then the property uttered is true. In the case where a state allows the protocol to proceed with one of the stable states other than I then by constructing the protocol we assume that the property is true by hypothesis and that the path is not actually reachable. Indeed, at the level of the SLICC, we do not have enough information on the results of the requests and the answers made during the execution of the actions present in the transitions.

```
1  subsection determinist.
2  (* return true if src1 and src2 has a
       common subset *)
3  Fixpoint hasinterIdent(src1 : list ident)
4    (src2 : list ident) := match src1 with
5    | [] => false
6    | e::rest => if(containIdent src2 e)
         then true else
7      hasinterIdent rest src2
8    end.
9  (* return true if there exist same src on
       t1 and t2 *)
10 Definition existsameSrc (t1: transition)
       (t2: transition) := hasinterIdent
11 (getSrc t1)(getSrc t2).
12 (* return true if there exist same event
       on t1 and t2 *)
13 Definition existsameEvent(t1: transition)
        (t2: transition) := hasinterIdent
14 (getEvents t1)(getEvents t2).
15 (* return true if there exist same src
       and event on transitions *)
16 Definition sameSrcandEvent(t1: transition
       )(t2: transition):=
17   if(existsameSrc t1 t2) then
18     if(existsameEvent t1 t2) then true
19     else false
20   else false.
21 (* return true if there exist a
       transition on l equivalent as t*)
22 Fixpoint itexist (l: list transition) (t:
        transition) := match l with
23   | [] => False
24   | e::rest => if (sameSrcandEvent e t)
         then True
25     else itexist rest t
26   end.
27 (* isDeterminist return true if l is
       determinist, thus
28  * the FSM is determinist too if FSM is
       complete *)
29 Fixpoint isDeterminist(l: list transition
       ) := match l with
30   | [] => True
31   | e::rest => ~(itexist rest e) /\ (
         isDeterminist rest)
32   end.
33 End determinist.
```

Listing 3: isDeterministic functions in *Get_Set_Slicc.v* library

There exists an identical property of uniqueness for the state modify M or it is enough to replace the state E by M in the previous property 3. Here the property is similar to the previous property because a given data being in the modified state M must be unique as if it were exclusive. Modify state

means that this data is the most up-to-date version of the system and what has not yet been copied back into memory. On the other hand, the proof and the conclusions of this property are the same as before, this property is true if hypothesis are verified on low level implementation.

The memory is distributed on the L2 caches with the following rule:

Property 4 *The union of all the L2 caches address space is equal to the memory address space and the intersubsection between the L2 caches address spaces is empty.*

This means that each L2 cache maps a given partition of the memory address space and that there is no duplication of data at this level. In order to verify this property and thus to ensure the uniqueness of the data at the level of L2 caches, it is necessary to verify the way the distribution of the data is done in the implementation of the machine RubySlicc present in gem5. It is this machine that is used to distribute the data in the module $RubySlicc\_ComponentMapping.sm$ are the functions that allow to map the addresses to the corresponding block. At the level of the SLICC the implementation of these functions is not present because they are low level functions. So the partition property of the memory must be assumed at the SLICC level and checked again at low level. We have nevertheless verified this at the low level and the corresponding functions to distribute the data realizes many blocks of distinct memory and associates different L2 cache without repetition or cross-checking of the data. So the property is true in the case of the MESI.

3.4 Verification of H-NUCA

Now let's see properties specific to H-NUCA protocol. What properties need to be checked in this protocol? As its structure is based on the MESI protocol, this protocol has similarities with the latter. Certain properties of the MESI can be directly reused for H-NUCA, for example property 3. Indeed, at the level of the L1 cache H-NUCA contains the two exclusive E and modify M states and thus property 3 which deals with the uniqueness of these states, remains true in this protocol. Thus we must apply the same proof as MESI that has been verified, hence the properties remains true. However, as well as in the MESI case, some properties are specific to H-NUCA and can not be generalized.

Property 5 *Each data in L2 caches is unique.*

Although this property appears to be very close to property 4 of the MESI, the verification of this property in H-NUCA is quite different. The distribution of the memory is not carried out in the same way between MESI and H-NUCA. To ensure the veracity of this property, we will consider the following cases. If a L1 cache searches for a data that is not present in that cache then it requests the data from the target L2 cache. There are three possibilities :

- The L2 cache has the data then it transmits it to the L1 cache.
- Otherwise, if the L2 cache does not have the data then it will broadcast a request and query the other L2 caches. If one of them has the data then the L1 cache, which requests the data is notified that other L2 cache has the data.
- If no L2 cache has the data then the request falls through to memory.

It is easy to see that at a given instant there is at most one L2 cache which possesses the data and in the worst case no cache has the data. Thus we have uniqueness of the data at the level of the L2 caches and the property is checked.

Some of the properties at the SLICC level can be proved but sometimes it is also necessary to look for the low level implementation in order to validate some assumptions necessary for the properties to be true.

4 Related work

4.1 Automatic Verification

Stern and Dill [9] describes an ongoing effort to verify cache coherence protocol of IEEE/ANSI Standard for Scalable Coherent Interface (SCI) using the Murφ verification system. Murφ automatically checks if all reachable states in the model satisfy given specifications. Stern and Dill [9] constructs the model in three steps.

- Abstraction : extract details of SCI that are important for cache coherence protocol.
- Simplification : make model construction possible in a "finite" amount of time. Strong ordering constraints were assumed, not modeling of DMA reads and writes and not modeling full set cache coherence protocol.
- Implementation : keep some parameters, like the number of processors, the number of lines in each cache, the number of memories and the numbers of different data values.

SCI C code includes many assert statements to detect errors while running simulations with an in-house built execution environment. C code contains several statements for detection of memory and cache inconsistencies. Stern and Dill [9] adds some invariants to Murφ model to specify more accurately cache coherence. They conclude by saying that the abstraction done in modeling was relatively simple and straightforward. Verification should be viewed as a debugging tool. The task of modeling in a formal model should be reducible because it takes a lot of time.

4.2 Correctness for Tardis Cache Coherence Protocol

Yu et al. [10] prove correctness of Tardis, a cache coherence protocol, by developing an intuitive invariants system. Each memory operation in Tardis has a corresponding time-stamp which indicates its global memory order. Every cache line in Tardis has a read time-stamp (trs) and write time-stamp (wts). wts is the time-stamp of last store and rts is the time-stamp of last potential load to cacheline. Yu et al. [10] model a two-level memory subsystem. They prove correctness of

Tardis protocol specified by proving that it strictly follows sequential consistency and then they add deadlock freedom and live lock freedom to analyze Tardis in parallel composition. A parallel program can be seen as an sequentially consistent if result of any execution is same for all processors. Thus they create some properties to verify if the system satisfies this consistency model. This approach is based on Lamport's classical time-stamp model. Then they prove deadlock and live lock freedom on Tardis.

For Deadlock they define this Theorem :

Theorem 4.1 *After any sequence of transitions, if there is a pending request from any processor, then at least one transition rule can fire.*

And for Livelock they define this Theorem :

Theorem 4.2 *After any sequence of transitions, if there exists a pending request from any processor, then within a finite number of transitions, some request at some processor will dequeue.*

These two theorems have been proved in their article and thus they provide rigorous proofs of correctness for Tardis cache coherence protocol.

4.3 Verified compilation

It is also possible to check programs directly in the compilation chain. This is presented in the works of Leroy [11], Blazy et al. [12] and Boldo et al. [13]. All these works aim to improve the verification and validation of program during the compilation of these program by adding some semantic rules. Each compilation step contains verifiers which allows to master the compilation chain and thus ensure that translations remain correct at all levels. Thus, a program compiled in this way is verified and the guarantee of this verification is ensured by the compiler and not by an external tool. The main interest is therefore to simplify the task of the developer who will have to make a minimum effort to ensure that the program follows the specification associated with it. Blazy et al. [12] defines a first syntax to abstract the program C to be analyzed. This syntax is of the language Clight that is used to realize the first stage. The dynamic semantics of Clight is also described in their article. After this step it starts translating from the Clight to a new abstraction named C#minor. And so on. Step by step it translates the program by adding verification mechanism either directly in the semantic or in the translation tools.

Boldo et al. [13] describes the verification of floating point computations in compilers. They ensure in the compiler that the computations remain correct after modification. This verification remains in the same spirit as the other compilation phase, it tests to ensure the validation and verification of the program at compilation time.

All these works made it possible to realize the **CompCert** compiler which is a C compiler that aims at checking the given programs and generating a verified compiled version of them. This compiler is available at the following address: `http://compcert.inria.fr`.

4.4 Formal Verification of Hardware Designs with COQ

Braibant and Chlipala [14] present an implementation of certified compiler for a high-level hardware description language (HDL). And Vijayaraghavan et al. [15] presents a new framework for modular verification of hardware designs. Both uses Coq proof language to model and verify their systems. They developed a modular proof structure for distributed shared-memory hardware systems. They use labeled transition systems (LTS), consisting of a set of states and a set of transitions between those states. An LTS[16] is a tuple $(S, A, \rightarrow, s_0)$ where :

- S is a (finite) set of states;
- A is a set of actions;
- $\rightarrow \subseteq S \times A \times S$ is a transition relation;
- $s_0 \in S$ is initial state.

An LTS describes the behavior of some system or protocol: in any state of the system, a number of actions can be performed, each of which leads to a new state. For example, store atomicity can be described as an LTS that receives load and store requests. And all other parts of their evidences will use a description of LTS. Vijayaraghavan et al. [15] define sequential consistency (SC) property and they prove partial correctness in Hoare logic of this consistency. Sequential consistency [17] is one of many consistency models used in concurrent systems.

"*The result of any execution is same as if (read and write) operations by all processes on data store were executed in some sequential order and operations of each individual process appear in this sequence in order specified by its program.*"

Andrew S. Tanenbaum and Maarten van Steen [18]

In addition to this mechanism Vijayaraghavan et al. [15] use the COQ proof assistant. They have modeled all necessary structures in their COQ library and then declared and proved all properties in order to carry out verification of the system.

Bidmeshki and Makris [19] describe a Verilog to Coq converter to convert a hardware description language (HDL) to a formal representation. Verilog (IEEE 1364) is a HDL used to model electronic systems. It is used in design and verification of digital circuits at register-transfer level (RTL). They convert Verilog representation into an equivalent in COQ with inductive type value corresponding to low (lo), high (hi) and x (undefined) and all description of bus and elements in Verilog. VeriCoq creates axioms for modules and source code represent module instantiation.

4.5 Verification by temporal logic model checking

Clarke et al. [20] describe the formalization and verification of the cache coherence protocol included in the IEEE Futurebus+ standard draft. They construct a precise model of the protocol in a hardware description language and then use temporal logic model checking to verify if the model satisfies a formal specification. They use symbolic model verifier (SMV), a temporal logic model checker based on binary decision diagrams (BDDs). SMV accepts specifications expressed in the computation tree logic (CTL) temporal logic. Clarke et al. [20] conclude that all formal verification involves making a model of system under consideration. Model checking is not limited to finite-state models arising from hardware. Formalization and analysis of other type of systems should also be possible using SMV.

Hendriks et al. [21] use temporal logic and model checking for system analysis. They use Metric Temporal Logic (MTL) on execution traces. Checking validity of an MTL formula is a fundamental problem in model checking and runtime verification. They use MTL for two reasons, first to check if the system follows some properties and when some formulas are not respected, and second to analyze performance and scalability of the system. One can evaluate time complexity of an execution trace with temporal logic. Hendriks et al. [21] concluded that MTL provides a flexible mechanism to specify quantitative properties of execution traces.

5 Conclusion and future work

Program verification is a difficult task, it is dependent on the technique used and the level of abstraction chosen. In this work we target the verification of complete cache coherence protocols described using the SLICC language, a *de facto* standard for this task. The SLICC descriptions are at a fairly high level of abstraction, as they rely on an API to perform the actual implementation related actions. Working at the SLICC level causes some problems. Indeed, properties that are needing data or addresses associated with these data generates hypotheses that can not be verified at this level of abstraction. Nevertheless, the verification of protocol can be realized directly by verifying their description. We can know if the states that are described in the FSMs are reachable or if the FSMs are deterministic. The formal verification by Theorem Proving also allows us to quickly find bugs in the description of protocols and thus modify this protocol directly at this level of abstraction.

We are interested in the complexity concerning the comparison between theorem proving and model checking. What emerges from this work is that the amount of work the verification engineer has to produce is almost the same between the two techniques. Indeed, there is a (potentially large) gap between specification and implementation that has to be bridged by model translation for model checking. And we have the same discrepancy in the formalization of specification and implementation in theorem proving. On the other hand what the literature shows is that one can use the model checking as first pass in order to check the top level of property and for that which would not be checked, one could formalize the system and these properties in order to be verified again with theorem proving. The combination of the two approaches makes it possible to have two different views of the problem, therefore to reinforce the verification of the model.

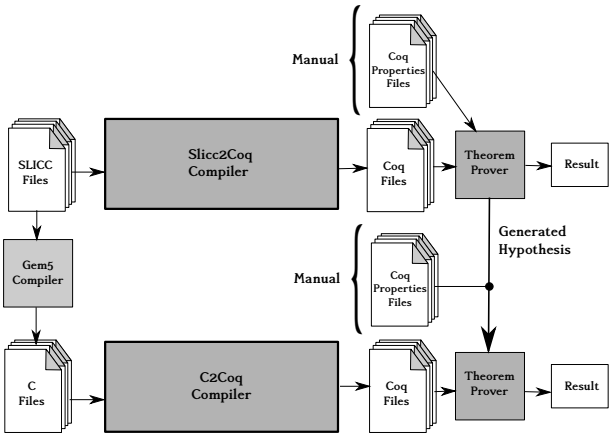


Figure 6: Two level verifier

Another possible approach would be to analyze as we are currently doing the SLICC level and then retrieve the hypotheses verified at the level of the implementation as shown on Figure 6. Then create a second tool that would do the same work at low level with the previous hypotheses as low level properties to check. Thus, using this two-phase technique we could verify the protocols both at the level of the SLICC and at the level of the implementation and thus reinforce the validation of the protocols. This technique would be similar to the one that was realized in the compilation checked by Leroy [11] for the **CompCert** compiler. Theoretical study of this new tool could thus be carried out as a future work.

Finally, we could modify the **slicc2coq** compiler so that it automatically generates the necessary assumptions to prove from a SLICC protocol and a list of properties checked. Thus we would have the list of hypotheses to prove at the level of implementation, and then check them manually because the automatic verifier does manage to reach a conclusive result. Thereby we would automate the verification part as much as possible to concentrate on the sensitive points of the protocols.

All the tools that we have realized and developed are available from the forge of the System Level Synthesis team of the TIMA laboratory. The tools are written in Ocaml and the available libraries are written in COQ without modification of the languages and without other dependency.

References

[1] Tilera. Tile-Gx72 Processor, May 2017. http://www.mellanox.com/page/products_dyn?product_family=238&mtag=tile_gx72.

[2] Gem5. SLICC, August 2017. http://www.gem5.org/SLICC.

[3] Gerard J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1991. ISBN 0-13-539925-4.

[4] Milo M. K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood. Multifacet's general execution-driven multiprocessor simulator (gems) toolset. *SIGARCH Comput. Archit. News*, 33(4):92–99, November 2005. ISSN 0163-5964. doi: 10.1145/1105734.1105747. URL http://doi.acm.org/10.1145/1105734.1105747.

[5] Mark S Papamarcos and Janak H Patel. A low-overhead coherence solution for multiprocessors with private cache memories. *ACM SIGARCH Computer Architecture News*, 12(3):348–354, 1984.

[6] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, August 2011. ISSN 0163-5964. doi: 10.1145/2024716.2024718. URL http://doi.acm.org/10.1145/2024716.2024718.

[7] Hela Belhadj Amor, Hamed Sheibanyrad, and Frédéric Pétrot. A meta-routing method to create multiple virtual logical networks on a single hardware noc. In *IEEE Computer Society Annual Symposium on VLSI*, pages 200–205. IEEE, 2017.

[8] Changkyu Kim, Doug Burger, and Stephen W. Keckler. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. *SIGARCH Comput. Archit. News*, 30(5):211–222, October 2002. ISSN 0163-5964. doi: 10.1145/635506.605420. URL http://doi.acm.org/10.1145/635506.605420.

[9] Ulrich Stern and David L Dill. Automatic verification of the sci cache coherence protocol. In *Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pages 21–34. Springer, 1995.

[10] Xiangyao Yu, Muralidaran Vijayaraghavan, and Srinivas Devadas. A proof of correctness for the tardis cache coherence protocol. *arXiv preprint arXiv:1505.06459*, 2015.

[11] Xavier Leroy. Mechanized semantics for compiler verification. In Ranjit Jhala and Atsushi Igarashi, editors, *Programming Languages and Systems, 10th Asian Symposium, APLAS 2012*, volume 7705 of *Lecture Notes in Computer Science*, pages 386–388. Springer, 2012. URL http://gallium.inria.fr/~xleroy/publi/mechanized-semantics-aplas-cpp-2012.pdf. Abstract of invited talk.

[12] Sandrine Blazy, Zaynah Dargaye, and Xavier Leroy. Formal verification of a C compiler front-end. In *FM 2006: Int. Symp. on Formal Methods*, volume 4085 of *Lecture Notes in Computer Science*, pages 460–475. Springer, 2006. URL http://gallium.inria.fr/~xleroy/publi/cfront.pdf.

[13] Sylvie Boldo, Jacques-Henri Jourdan, Xavier Leroy, and Guillaume Melquiond. A Formally-Verified C Compiler Supporting Floating-Point Arithmetic. In Alberto Nannarelli, Peter-Michael Seidel, and Ping Tak Peter Tang, editors, *Arith - 21st IEEE Symposium on Computer Arithmetic*, pages 107–115, Austin, United States, April 2013. IEEE. URL https://hal.inria.fr/hal-00743090.

[14] Thomas Braibant and Adam Chlipala. Formal verification of hardware synthesis. In *International Conference on Computer Aided Verification*, pages 213–228. Springer, 2013.

[15] Muralidaran Vijayaraghavan, Adam Chlipala, Nirav Dave, et al. Modular deductive verification of multiprocessor hardware designs. In *International Conference on Computer Aided Verification*, pages 109–127. Springer, 2015.

[16] MCRL2. Labelled transition systems, August 2017. http://www.mcrl2.org/web/user_manual/articles/lts.html.

[17] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, pages 690–691, 1979.

[18] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems: Principles and Paradigms (2Nd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2006. ISBN 0132392275.

[19] M. M. Bidmeshki and Y. Makris. Vericoq: A verilog-to-coq converter for proof-carrying hardware automation. In *2015 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 29–32, May 2015. doi: 10.1109/ISCAS.2015.7168562.

[20] Edmund M Clarke, Orna Grumberg, Hiromi Hiraishi, Somesh Jha, David E Long, Kenneth L McMillan, and Linda A Ness. Verification of the futurebus+ cache coherence protocol. In *CHDL*, volume 93, pages 15–30, 1993.

[21] M. Hendriks, M. Geilen, A. R. B. Behrouzian, T. Basten, H. Alizadeh, and D. Goswami. Checking metric temporal logic with trace. In *2016 16th International Conference on Application of Concurrency to System Design (ACSD)*, pages 19–24, June 2016. doi: 10.1109/ACSD.2016.13.

A Language for the Smart Home

Lénaïc Terrier

Abstract

This report presents CCBL (Cascading Contexts Based Language), an end-user programming language dedicated to Smart Home. We designed CCBL to avoid the problems encountered by end-users programming with ECA (Event Conditions Actions), which is the dominant approach in the field. We present the results of a user-based experiment where we asked 21 adults (11 experimented programmers and 10 non-programmers) to express four increasingly complex behaviors using both CCBL and ECA. We show that significantly less errors were made using CCBL than using ECA. From this experiment, we also propose some categorization and explanation of the errors made when using ECA and explain why users avoid these errors when programming with CCBL. Finally, we explore error reporting for CCBL by identifying two specific errors and by developing a solution base on Heptagon and ReaX to detect them in CCBL programs.

1 Introduction

[Mennicken *et al.*, 2014] define **Smart home** as "*a home that either increases the comfort of their inhabitants in things they already do or enables functionalities that were not possible before through the use of computing technologies*". As said in [Holloway and Julien, 2010 ; Crowley and Coutaz, 2015] increasing comfort cover a wide range of tasks : saving energy, increasing security, helping with every day chores or providing distant control and awareness to users.

In this work we consider that inhabitants of smart homes must have the control over the system. This means that inhabitants should be able to override the system behavior. More fundamentally, we consider inhabitants not as mere consumers of services provided by the home, but on the contrary, in the same vein than [Fontaine, 2012 ; Dautriche *et al.*, 2013 ; Coutaz and Crowley, 2016], we consider that inhabitants are also builders/makers. In order to support this underlying philosophy, we aim at proposing a whole **End-User Development** system in the long term. However, this work focus on one aspect of such a system, namely the programming language. In the first part of the report, we study which programming language is the most appropriated to enable inhabitants to program behaviors of their smart home. Based on that, in the second part of the report, we study possible static verification that can be checked when using such a language.

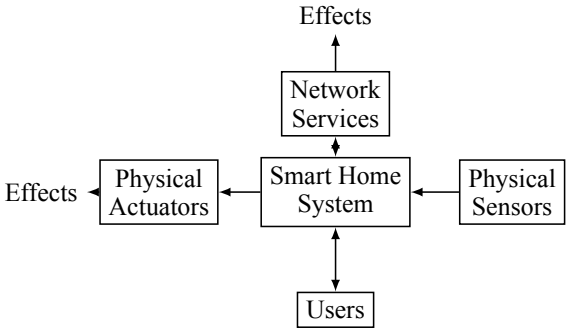


Figure 1 – Smart Home Summary Layout

In the rest of this section, we first present smart homes from a technical perspective, then from the perspective of its inhabitants. Last, we discuss how inhabitants can communicate their needs to their smart home.

1.1 Smart Homes : a technical perspective

From a technical point of view **Smart Home** services are provided by a **Home Automation System** which is based on the conjunction of three types of elements : **sensors** (thermometer, lightmeter, switch, clock…), **actuators** (lights, shutters, sound system…), **network services** (weather, traffic, synced calendar…). This view is summarized in the figure 1.

As observed in [Demeure *et al.*, 2015 ; Brush *et al.*, 2011], in the industry there is two types of home automation systems. What we could call *heavy installations* are installed when the house or apartment is built. They rely on wired hardware fixed in walls and ceilings such as **KNX** technologies. In recent years, another type of home automation systems has emerged, what we could call *light installations* are based on wireless communicating technologies such as **Z-Wave** [1] or **EnOcean** [2].

Variety of technologies is a both a recurring theme and a

1. `http://www.z-wave.com/`
2. `https://www.enocean.com/`

major issue in the home automation domain. Companies provide each their own technology with their own standard. Many standards means high incompatibilities and users are currently forced to rely on systems that aggregate some of those standards.

Again, there is two solutions to this standard problem. On one hand, solutions like **eeDomus** [3], **Zipabox** [4] or **HomeSeer** [5] that aggregate as much as standards as possible so they can be compatible with most devices. On the other hand, solutions like **Apple HomeKit** [6] and **Microsoft HomeOS** [7] that implement one standard. If a device constructor wants its products to be compatible, the device must implement the proper standard. Only companies powerful enough to dictate their conditions to the market can only provide this solution.

1.2 Smart Home from the perspective of its users

The users of a smart home are its inhabitants and that is one of the reason it is a unique and complex environment. A household is usually home to several inhabitants organized with specific sociological relations. Each inhabitant have needs, desires and habits that can be similar, different or even in contradiction with others. The smart home will not be the solution to solve these contradictions but should at least consider them. As observed in [Brush *et al.*, 2011 ; Demeure *et al.*, 2015], inhabitants have different roles regarding the smart home system. Usually, one inhabitant (the guru) is the only one in the household to maintain and modify the home automation system. The others inhabitants merely provide feedbacks to the guru. Although the guru is technically the administrator of the system, observation shows that often another inhabitant holds the final decision power. In a family, the guru is usually the *husband* and must receive validation of the *wife* to confirm a behavior of the system or the adoption of a new device. In the literature, this is called the **Wife Acceptance Factor** [see Demeure *et al.*, 2015].

That being said, the guru is not always the husband. For instance, in [Coutaz and Crowley, 2016] the author experimented home automation system on her own household and, although being the wife, she rapidly emerged as the guru. In the paper she explains that after a certain amount of time, the system reaches a stability meaning the users will not edit its behavior much. After they are used to the installed behaviors, they start playing with them. In the paper, she gives the example of a behavior the family designed to reduce energy consumption : when the fridge is open for a certain amount of time the system sends an email to alert the family members. Inhabitants turned this simple rule into a communicating tool. The author explains that they are using it as a "*Hurry to get home, I am cooking dinner*" notification. This kind of misuse of a tool to achieve a new goal can only emerge when the family knows well its environment, including the automation part.

In [Davidoff *et al.*, 2006], the author highlights the difference between a family **routine** and a computer **procedure**. A routine is flexible and subject to improvisation and changes whereas a procedure is not. Inhabitants do not really notice the system when the routine flows normally. However, when a routine breaks, if the systems does not acknowledge it and continues as if the routine was continuing it becomes another issue in the crisis. Being useful means that the system should help the users solve the crisis.

1.3 Interaction between inhabitants and their smart home

The simplest way of interacting with a smart home is probably the **direct control** (vocal or via graphical user interfaces). [Demeure *et al.*, 2015] observe that, even in 2015, it was frequent for inhabitant of Smart Homes to be equipped with vocal recognition system that could control actuators (e.g. "Close the shutters", or "Turns the TV on"). This trend continue nowadays with the apparition of assistants such as Google Home [8] or Amazon Echo's Alexa [9].

Automatic learning represents another strategy to interact with smart home. In this approach, the system learns for the users habits and behave accordingly. Several systems already use this technology. For instance, **hot water recirculation systems** are useful for saving energy. They take the unused hot water from pipes to put it in the boiler again. When they are coupled with automatic learning, the energy saving are even greater because the hot water is produced by the boiler just in time. Moreover, the system can place hot water in the pipes just before the users need it so that they will immediately get hot water without waiting [Frye *et al.*, 2013]. Although automatic learning works well for precise tasks with few variables it does not cover all the use cases. For instance, it is almost impossible for an end user to teach an automatic learning system to change the color of a lamp depending on the status of the train he uses to commute. ?

The approach for interacting with the system that we use in this work is based on **End User Development**. It relies on the idea that the smart home will enable inhabitants to program the behavior they want. End User Development (EUD) is defined in [Lieberman *et al.*, 2006] as follows : "*a set of methods, techniques and tools that allow users of software systems, who are acting as non-professional software developers, at some point to create, modify, or extend a software artifact*". Because the user needs to code the behavior he wants from the system, this solution is more demanding than automatic learning. Programming is a process that consist in translating mental representation of behaviors into a set of instructions executable by a system. It has to be noted that the fundamental motivation for using a smart home is not to learn how to program but rather configure the system behavior to better support inhabitants (in terms of comfort, energy saving, security, etc.). Developing is an even broader process, which include finding interesting behaviors to program but also debugging or evolving existing behaviors. Of course, end-user development, automatic learning and direct control can be used together to better assist the inhabitants. We choose to focus on end-user development for two reasons : first, from a philosophical

3. `http:/www.eedomus.com/`
4. `http://zipabox.domadoo.com/`
5. `http://www.homeseer.com/`
6. `https://www.apple.com/fr/ios/home/`
7. `https://www.microsoft.com/en-us/research/project/homeos-enabling-smarter-homes-for-everyone/`
8. `https://madeby.google.com/home/`
9. `https://www.amazon.com/Amazon-SK705DI-Echo/dp/B00X4WHP5E`

perspective, it considers the inhabitants as builders more than mere consumers ; second, this approach is more mature with respect to automatic learning while being still challenging in terms of research [Crowley and Coutaz, 2015]. Of the many aspects of end-user development for smart home, we choose to focus on the programming language and on static verifications of programs.

2 Event Condition Action (ECA) and Trigger Action Programming (TAP)

In the domain of end-user development for smart home and internet of things (IoT), the highly dominant used type of language is ECA ; **Event Condition Action**. As observed in [García-Herranz *et al.*, 2010 ; Brush *et al.*, 2011 ; Demeure *et al.*, 2015 ; Cabitza *et al.*, 2015, 2016 ; Fogli *et al.*, 2016], ECA is used in most home automation systems on the market.

ECA programs are usually a set of rules in the form : "*WHEN event IF condition THEN action*". When the `event` occurs, if the system evaluates the boolean `condition` to true, then it executes the `action`. ECA programming languages are derived from practice ; there is no unique formal definition that we can refer to.

In this report, we will use the definition proposed by [Pane *et al.*, 2001]. The authors state that ECA programming consists of specifying a set of rules where each of them "*autonomously reacts to actively or passively detected simple or complex events by evaluating a condition or a set of conditions and by executing a reaction whenever the event happened and the condition is true*". This means that :

— A rule is activated only by events ;
— Its execution is independent of other rules in the system ;
— It implements a reaction to the incoming event ;
— It contains a guarding condition to execute such actions.

Trigger-action programming (TAP) is a variant of ECA. Recent works [Ur *et al.*, 2014 ; Huang and Cakmak, 2015] used this term although authors do no explain in which way it differs from ECA. [Häkkilä *et al.*, 2005] seems to be the first to refer to it. While [Dey *et al.*, 2006] do not use the term (they write about if-then rules), they propose a very similar behavior and are cited in [Ur *et al.*, 2014] as an example of system that support trigger-action programming.

While none of the authors of aforementioned works do provide formal definition of trigger-action programming, we propose the following one, inspired by [Häkkilä *et al.*, 2005] : Trigger action is a variant of ECA, where it is possible to express rules of the form "*IF conditions THEN actions*", where conditions do not make any reference to an event. The semantics of this form of rule, expressed in standard ECA, is "*WHEN conditions become true THEN do actions*".

2.1 ECA and TAP in existing tools

One of the most popular tool using TAP is also probably the simplest. **IFTTT** is a online service that use `IF ... THEN ...` rules, as suggested by its name : IFTTT means *If This Then That*. Rules are called "*applet*" and one event can be bound to several actions in one applet. The strength of IFTT relies in numbers. More than 400 services are connected to IFTTT which runs more than 1 billion applets executed every month [10]. In 2016, IFTTT users enabled (started using) more that 15 millions applets and was used over 102 countries [11]. Technically, IFTTT uses a degraded version of ECA with only one event and no condition. The programmable behaviours are in theory lower than its competitors (low ceiling) however, what it lacks with its language, IFTTT makes up with the services connected.

Systems like **eeDomus** and **HomeSeer** offer capabilities that are more advanced [Demeure *et al.*, 2015]. They enable end-user to combine multiples conditions and actions in a same rule. Although we do not have access to a full description of their system (these are industrial products), it is interesting to notice that manufacturers added functionalities such as expressing duration for observed state (the temperature has been lower than 10 degrees for 5 minutes). This is the result of a capitalization of years of experience in the domain of home automation, the eeDomus home automation system even refined ECA into ECAN (N standing for notifications), explicitly stating that notifications are semantically different from other actions (such as opening shutters). The programming language used in these tools is ECA, although some of them (Homeseer) also support TAP programming.

OpenHAB is an open source project for controlling and programming Smart Homes. OpenHAB is quite popular in the community of *DoItYourselfers*. OpenHAB enable users to define ECA rules composed of a set of events and a block of events. If the system triggers any event of the set of instruction, then the system executes the block of instructions. This block of instructions may contain conditions and branches. Consequently, OpenHAB only supports strict ECA programming, it does not support the expression of TAP rules of the form "IF conditions THEN actions". OpenHAB provides a high ceiling, it enables users to program with variables, but it also has a quite high threshold for beginner to start programming.

Tasker is a tool to program automation on Android smartphones. It can use almost every input of the phone as a trigger and almost every output as an action [Lucci and Paternò, 2014]. The user can make ECA-like and TAP-like programs. Although the possibilities offered by Tasker are great (high ceiling), the user needs time to understand the application and all its concept (high threshold). Tasker is oriented towards android hackers and power users. It has to be noted that Tasker implements a *non sequitur* for TAP rules : it proposes to automatically roll back the state of devices modified in the action part of the rule when the conditions become false. For instance, if we consider the program "*IF I am at home THEN turn on Wi-Fi*", Tasker will turn on the Wi-Fi when I enter home, but it will also turn it off when I exit home if it was off before I entered home. This semantic is not part of TAP programming ; however, it is useful and seems to have been added to Tasker as an empiric response to a concrete problem.

AppsGate [Coutaz *et al.*, 2014a,b ; Coutaz and Crowley, 2016] propose to structure rules into programs. Each program contains a stub composed of actions that are executes once

10. `https://ifttt.com/`
11. `https://ifttt.com/blog/2017/01/year-in-review`

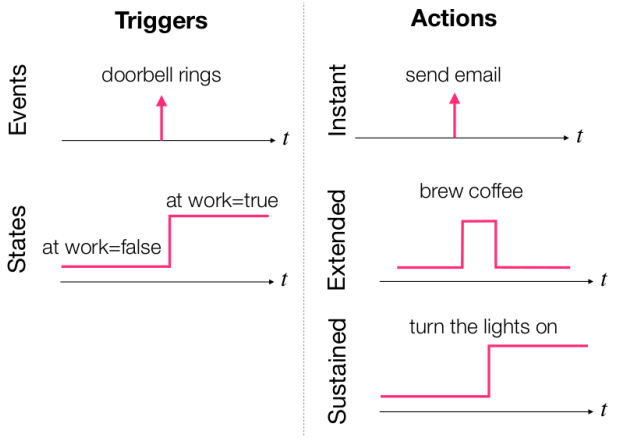


Figure 2 – Different types of triggers and actions (taken from [Huang and Cakmak, 2015])

the program is activated. The core of each program is a set of ECA rules that are active as long as the program is active. In addition to ECA, AppsGate introduce a new rule to take into account *non sequitur* reasoning. This new instruction is an empiric response to the problems regarding *non sequitur* reasoning observed in [Fontaine, 2012] in the same vein than what [Huang and Cakmak, 2015] proposed later.
"*While <state-condition> then keep <state> and as soon it is not true anymore then IMPERATIVE-SECTION*" For instance :
"*While TV is ON then keep Lights OFF and as soon as it is not true anymore then turn Lights ON.*"

In spite of the structure in programs, the programming language of AppsGate do not provide means for specifying priorities among rules nor among programs. Consequently, AppsGate propose a debugger with a "*graph of dependency*" that explicit which programs access to which devices.

2.2 Identified issues with ECA and TAP

Non sequitur **reasoning**

In [Huang and Cakmak, 2015] the author studied the different types of triggers and actions in TAP programming. They separated triggers in two categories : events and states, and they separated actions in three categories : instantaneous (sending an email), extended (brewing coffee) and sustained (turning on a light). These triggers and actions are illustrated in the figure 2. Based on that categorization, they observe that many users tend to do *non-sequitur* reasoning. When working with rules of the form "*WHEN state THEN DO sustained action*", they automatically associated this with the rule "*WHEN not state THEN DO reverse sustained action*". For instance, the rule "*WHEN I am at home, THEN lights are on*" is interpreted as implying that "*WHEN I am not at home, THEN lights are off*". This is probably what Tasker's authors also observed empirically. This was also noticed in [Fontaine, 2012].

Event enumeration

[Nandi and Ernst, 2016] studied 96 home automation rules written by end-users for the OpenHAB framework. The authors show that 77 of the 96 rules (80%) had a trigger part that contains fewer events that necessary. This mean that it is difficult for users to enumerate all the necessary events that are relevant to trigger a rule.

Coordination problems

While aforementioned researches study rules quite in isolation from each-others, [Cano *et al.*, 2014] study the coordination problems in ECA rules. They propose three categories for them :
1) Redundancy, when two rules will at least partially do the same actions when an event occurs ;
2) Inconsistency when two or more rules send contradictory actions to a same device ; and
3) Circularity when rules get activated continuously, without reaching a stable state.

2.3 Conclusion

ECA and TAP are dominant languages in the domain because they are easy to use to express simple behaviors. However, we identified several problems with ECA and TAP based on the literature. In order to overcome these problems, we propose to define a new language with the objective to keep ECA's low threshold while providing a higher ceiling for expressing behaviors that are more complex. The next section is dedicated to the presentation of this language.

3 Cascading Context Based Language

We have designed **Cascading context Based Language** (CCBL) a programming language for end-users in the smart home. This language is designed to avoid pitfalls observed in ECA and in TAP, and to provide an alternative to it. CCBL was design with two main goals in mind :

Low threshold What is simple to express in ECA should be at least as simple to express in CCBL. In ECA, event small programs with one or two rules can be a problem if the programmer uses *non sequitur* reasoning. We designed CCBL to be fully compatible with this ind of reasoning. Since the research shows that people tend to use it without noticing, we think it should not be a source of errors but a functionality.

High ceiling Complex or big programs should be writeable in CCBL. In ECA, more rules means less predictability : all the rules have the same level of priority and the visual aspect of ECA is a list of rules without any structure. Since we're doing EUD, the programmers are not professional developers, we think the programs should be quickly readable. We designed CCBL so that the structure of the program is not just cosmetic but plays a major role in the programming.

3.1 Principles

In the following sections, we illustrate CCBL concepts with graphical and textual notations to make it more clear for readers. However, it has to be said that our aim is not to propose a syntax nor a style for the language. We focus only on the logic and expressiveness of CCBL.

CCBL is defined in several layers of increasing complexity. For the sake of the explanation, this paper stops at level `CCBL-5` (see appendix).

3.2 Context

A context is made of two elements : a **selector** and a **code block**. The former determines when the context is active, the latter is a list of instructions to execute. The selector is an expression composed of states, events and operators which can be reduced to a boolean value. A context can be either a **state context** or an **event context** depending on the nature of its selector.

State context

A state context is a active while the **state** described in its selector is true. The code block can only be composed of values to maintain. The code sample 1 illustrates a state context.

```
Martin.isHome {
  Lamp.isLit: true;
}
```

Code sample 1 – Concrete context example

Event context

An event context is triggered when the **last event** of its selector is triggered. Its code block can only be composed of punctual or extended actions. An event context is composed by adding the keyword **do** at the end of its selector. The code sample 2 illustrates event contexts. Note that in the code sample 2b the notification is sent at the end of the state `CoffeeMaker.isBrewing`. In CCBL a state is always composed of two events : the starting event and the ending event. Since an event context is triggered at the last event described in the selector, it's the ending event of the state that is used.

```
Button.onPush do {
  Sound.sendSMS(
    Phone.contacts.Paul,
    "I'm home."
  );
}
```

(a) The button sends an SMS to Paul when it is pushed

```
CoffeeMaker.isBrewing do {
  System.notify("Coffee is
      ready");
}
```

(b) The systems send a notification when the coffee is ready

Code sample 2 – Event context examples

3.3 Cascade and *non sequitur* reasoning

State contexts can hold in their code block another context : they can be nested. It's important to understand that only state context can have children contexts. Event contexts can be child of another context but can not have children.

When a context is inactive its children are not awake, they will not activate even if their selector becomes true. A context is sensitive only if its parent is active. The code sample 3 illustrates a nesting. Here for the context "*Martin.isHome*" to be active, the lamp have to be lit. We should note also that the nesting is strictly equivalent to a conjunction in term of selector. In this case, we could have only one context with the selector "*Lamp.isLit and Martin.isHome*". Nested contexts form a tree of contexts. We call this a cascade because of the similarity of concept with the W3C's Casacading StyleSheets. Each context is a subcase of its parent, and inherits of its selector. We call the **total selector**, the conjunction of the selector of a context and the conjunction of the selectors of the ancestors of said selector. The selector we build previously `Lamp.isLit and Martin.isHome` is the total selector of the child context in the code sample 3a.

In the code sample 3b, the parent and the child act upon the same channel : the color of the lamp. Since the child represent a subcase of its parent, its total selector is more precise, therefore the child have priority.

```
Lamp.isLit {
  Martin.isHome {
    Lamp.color = Lamp.colors.
        BLUE;
  }
}
```

(a) The Lamp is lit in blue while Martin is home

```
Alice.isHome {
  Lamp.color = Lamp.colors.
      ORANGE;

  Alice.isAVailable {
    Lamp.color = Lamp.colors.
        GREEN;
  }
}
```

(b) The lamp is orange when Alice is home but unavailable. The lamp is green when Alice is at home and available.

Code sample 3 – Nesting examples

At the root of the tree, there is a special context. This context is called **default context** or **root context** and has the special selector `*`. This context is a state context which selector is always true, meaning the context is always active. It is used to describe the default state of the system. The code sample 4 illustrates this context. In this example, by default, the lamp is off. While Martin is at home, the lamp is on. In CCBL, all states and channels have a default value defined by the system, the default context is location where the programmer can specify a default value to override the system. Because the default context is at the root of the tree, its instructions have the least priority in the tree, meaning any descendant of the root context can override these instructions.

```
* {
  Lamp.isLit = false;

  Martin.isHome {
    Lamp.isLit = true;
  }
}
```

Code sample 4 – Root context example : the lamp is lit while Martin is home

3.4 Priority and conflict resolution

Since several contexts might try to maintain one channel to different values, it is important to define an order of priority

to enforce predictability. We think it is important that this order is constant and total to provide consistency (meaning it is impossible for two context to have the same priority). In this report we do not provide the final set of rule to define priority, this matter is still under study. For now, we have implemented the following strategy :

Lineage A child has higher priority than all its ancestors. The reason behind this is, as we said earlier, because a child describe a subcase of its parent. A child adds precision to the context.

Order of declaration Because it is important that each context have a unique level of priority so that all conflict are solvable, we must provide a fallback absolute priority order. We choose to say that the latter declared a context the higher the priority. This means a context added at the end of a program would be the one with the highest priority. Also, in conjunction with **Lineage** this mean, if a context A has priority over a sibling context B because A was declared after B, then A has priority over all descendants of B.

These two rules add together describe a priority order that corresponds to a depth-first tree search. We think, other priority rules can be interesting, either to add to the current strategy or to give choice to the end-user.

Complexity Since a nesting can be view as a conjunction of selectors, this means the **Lineage** give higher priority to longer **total selectors**. In a selector, the higher the number of events, states and operators the more precise the situation describe. We think more precise context should have priority over less precise contexts.

Implication If a subcase have priority, this is also true for a boolean implication. This means that if a context A implicates a context B, then the context A have priority over B.

3.5 Expressive power of CCBL with respect to ECA

CCBL has the same expressive power as ECA. An ECA rule can be translated into structure composed of a state context having C for selector and an event sub-context having E for selector and A for actions. The way CCBL handles a context having an event for selector is different from a selector based on a state : When the event E occurs, actions A are performed. The code sample 5 illustrate an example of translation of one ECA rule.

4 User experiment

We performed a user experiment to test CCBL against ECA. The main goal of the experiment was to see if users were able to use CCBL to write programs without making errors. We specifically chose to make them write programs that associate states from devices to contexts since we know from [Huang and Cakmak, 2015] that some users may build wrong mental models of them.

```
1 TV.isOn { // C
2   EMailBox.newEMail do { // E
3     TV.notify("New email");
4   }
5 }
```

(a) CCBL program

```
1 when EMailBox.newEMail
2   if TV.isOn
3   then TV.notify("New email")
        ;
```

(b) ECA program

Code sample 5 – Use the TV to notify the user when an email is received

The experiment was set up in the winter 2016/2017, we took 21 participants living in the south-east of France, aged between 18 and 51 (average : 31.5 years old, standard deviation : 9.7 years). None of them were equipped with a smart home system. 11 are programmers, 10 have no programming knowledge. Participants were split info four groups according to their programming knowledge and the order in which they used the languages during the evaluation (ECA and CCBL).

Each evaluation session last nearly one hour. The participants and the evaluator are alone during the whole session time. It takes place either in the laboratory room or at the participant's home. The participant is placed in front of a computer, sit at a desk or a table. They are allowed to have a pen and a paper to draw or write at will.

Before the actual experiment, the participant plays around in the editor and explore the possibilities offered by the tool. The evaluator explains the key concept of the language the participant is about to test, the participant manipulate and can write programs. Before beginning, the evaluator asks questions to the participant to ensure he or she understands how the language works.

We provide participants with a homemade graphical editor to program with ECA and CCBL. The editor is in the form of a web application. The user interface of the editor is illustrated in the figure 3 A header indicates who the current participant is ; what the current step of the process is and it enables evaluator to go to the next step. The panel on the left lists the two sensors and two sensors/actuators of the smart home

- Alice sensor : It detects if Alice is at home, at Martin's home and if she is available. Her availability is independent of her location.
- Martin sensor : It detects if Martin is at home, at work, if he is phoning and if he sits in his sofa at home.
- Volume of the music : It sets the volume to off, low, normal or high. It can also be used as a sensor that indicates the state of the volume of the music.
- Lamp A : It sets the lamp to off, orange, green or white. It can also be used to indicate the lamp's state.

For each of them, the user interface lists related events (for ECA rules) conditions and actions (for both ECA and CCBL). The central part presents the program under edition. The footer part represent the exercise statements. As there is no standard way to represent ECA rules, we chose a representation that was consistent in terms of colors with the representation of CCBL programs. Figure 3 illustrates an ECA rule. Each rule must have one and only one event. It can have zero or several conditions and zero or several actions.

The four exercises consist of associating values of the state

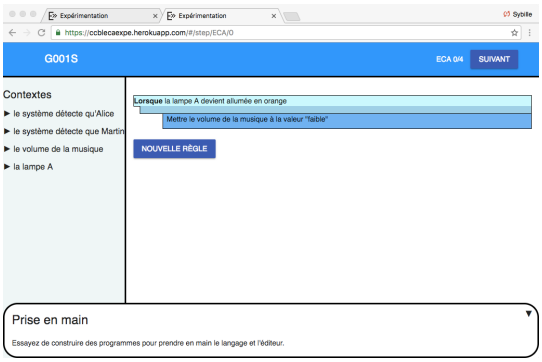


Figure 3 – User interface of the editor on ECA rules

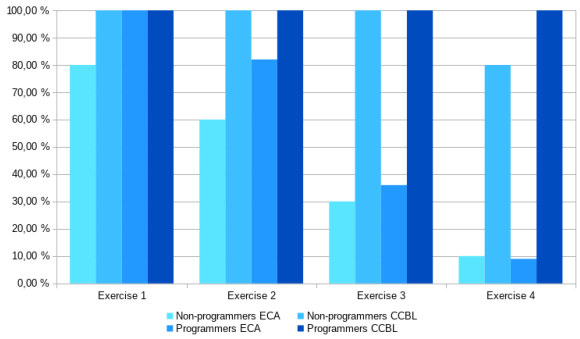


Figure 4 – Correctness of programs. We use ECA and TAP semantics to evaluate correctness of ECA programs.

of one device − either the color of the lamp or the volume of the music − to different contexts. We defined the four exercises to represent increasing levels of complexity. The complexity refers to the number of contexts taken into account as well as the number of possible transitions to switch from one context to another.

The exercises were the following :

1. **Exercise 1** : Martin wants the lamp to be lit in white if and only if he is at home (lamp is off otherwise).
2. **Exercise 2** : Martin wants the volume of the music to be low when he is at home, except when he sits in his sofa. In that later case, he wants the volume to be normal. When he is not at home, he wants the volume to be turned off.
3. **Exercise 3** : Martin wants the volume of the music to be normal when he is at home, except when he is on the phone, In that case, he wants the volume to be low. When he is not at home, he wants the volume to be turned off.
4. **Exercise 4** : Martin wants to use his lamp to be aware of Alice when he is at home. When Alice is at home and not available, he wants the lamp to be lighted in orange. When she is at home and available, he wants the lamp to be lighted in green. When Martin receives Alice, he wants the lamp to be lighted in white. In all other cases, the lamp should be off.

4.1 Experimentation results

First, we show that participants do not always respect the ECA semantics and tend to express rules with a TAP semantics. Second, we show that even when taking into account TAP semantics, participants make fewer errors when programming with CCBL than with ECA. Third, we analyze the errors made by participant when using ECA and ECA with TAP semantics. Finally, we analyze the structures of CCBL programs and discuss the users' point of view.

From ECA to TAP semantic

We identified two programmers and two non-programmers who expressed at least one ECA rule in which they added a redundant condition that can only be true when the system triggers the event. For instance, we observed the following rule : "When Martin enters home and if Martin is at home then set the volume of the music to low". For these participants the semantics difference between an event and a condition is not clear. These participants used ECA before CCBL, thus CCBL cannot have influence their mental model.

Several participants said during the sessions that they had problems to distinguish between events and conditions when programming with ECA. Sometimes, when reading the ECA programs, they transformed the event into a condition. For instance, "*Martin enters his home*" was pronounced "*Martin is at home*".

As stated before, an ECA rule is triggered by one event. When the event occurs, then conditions are evaluated. If conditions are evaluated as true, then actions are executed. However, we cannot understand several of the proposed ECA programs when using this semantics. Instead, we observe that participant blurred the distinction between event and condition. Participant considered the event as a condition. They used the following semantic : "*If the conditions become true, then execute the actions*". This is actually the semantics used in TAP.

Comparing CCBL and ECA (including TAP semantics)

Figure 4 summarizes the correctness of the programs specified by participants per exercise, language and programming knowledge. It takes into account that an ECA program can be correct either with the ECA semantics or with the TAP semantics. Non-programmers who correctly programmed exercises 3 and 4 used the TAP semantics, which suggest that this semantics better support users in programming behaviors that associate devices states to contexts. The correctness percentage is calculated from the number of programs produced without error.

Errors made by using ECA

From related work, we know that several types of errors exists when programming with ECA : *non sequitur* reasoning, using fewer triggers that necessary, redundancy, inconsistency and circularity. We observe no errors of inconsistency nor circularity. We observe only one error of redundancy (by a programmer in exercise 4). As the most complex exercise necessitates eleven rules to behave correctly, we postulate that this is not sufficient to produce this kind of errors.

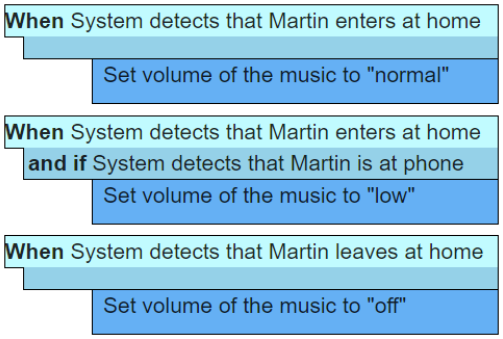


Figure 5 – Example of reasoning in terms of general case (rule at the top) and particular case (rule in the middle).

Despite the fact that the evaluator tried to ensure that participants understood ECA semantics well before starting exercises, we observe that two non-programmers used *non sequitur* reasoning for exercise 1. This suggest that this is a very instinctive way of reasoning. We observed other cases of *non sequitur* reasoning in exercise 2 : participant specified what to do when Martin sits down on the sofa but forgot to mention what to do when he leaves the sofa. We observed similar errors in exercise 3 : participants tend to forget to specify what to do when Martin hangs up the phone.

Many errors implied the use of fewer triggers than needed. This was especially the case for exercise 3 and 4. For instance, in exercise 4, the lamp lights in green when Martin is at home and Alice is at her home and available. This imply considering three events : "*Martin enters home*", "*Alice enters her home*" and "*Alice becomes available*". Most participants forgot about at least one of these events, which is consistent with what [Nandi and Ernst, 2016] observed.

Errors made by using ECA with TAP semantics

We observe two types of errors when programming with TAP : 1) Reasoning in terms of general and particular cases and 2) Forgetting to specify what happens when contexts are exited, which can be related to *non sequitur* reasoning. Figure 5 illustrates a participant using TAP semantics. Remember that every event has to be replaced with its related state in order to be interpreted correctly. The participant expresses a general case (Martin is at home) and associates it with the volume being **normal**. He then expresses a particular case (Martin is at home AND he is at phone) and associates it with the volume being **low**. However, this program is incorrect. Indeed, when Martin hangs up the phone, if he is still at home, no rule will trigger so the volume will stay **low** instead of being set to **normal**. We observe this error for $5/9$ non-programmers that used TAP semantics for exercise 3 (three programmed it correctly and one made another error). Programmers were less likely to use TAP semantics, only two of them did it for exercise 3, and one of them made the same error (the other programmed it correctly).

The other type of errors consists of forgetting to specify what happens when contexts are exited. We observe this type of error mainly for exercise 4. As illustrated in figure 6, participants were able to specify (with TAP semantics) when the lamp should light in orange, green or white. Every participant also mentioned to turn it off "*when Martin was not at home*" (it was explicitly stated in the exercise) but forgot to mention that it was also the case when Alice was not at her home nor at Martin's home. There are two possible explanations for this type of error : 1) Participants applied non-sequitur reasoning or 2) Participants were not aware of all possible contexts ("*Alice is not at her home nor at Martin's home*" fox example).

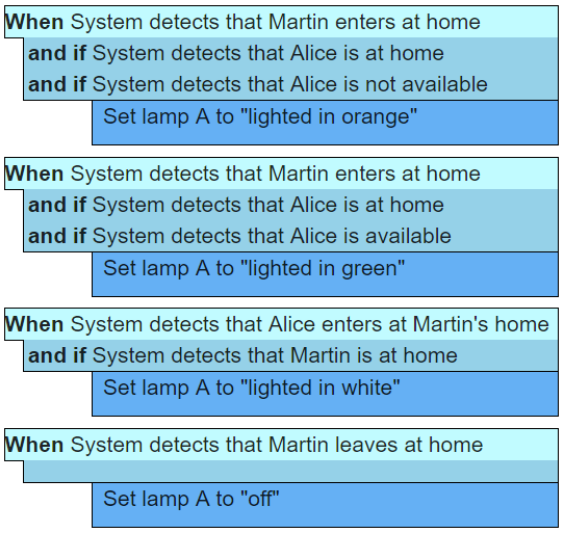


Figure 6 – Example of program where participant missed to express correctly when to turns light off (Miss the case when Alice is not at her home not at Martin's home).

4.2 Conclusion

The experimental results suggest that CCBL provides a threshold as low as that of ECA for simple problems. There was no significant difference observed between programming with ECA and with CCBL. This suggests that the CCBL cascade mechanism and the notion of sub-context do not introduce difficulties for the user that could lead to a higher threshold than for ECA.

Only two non-programmers made an error when using CCBL. Both did it at exercise 4. The first one forgot to mention that Martin had to be at home to light the lamp in orange, green or white. The second non- programmer expressed a very different behavior than what we asked him to do. We suspect that he was tired of the exercises and wanted to finish as soon as possible (it was his last exercise and he answered very quickly to the ending questions).

We analyze the structure of CCBL programs proposed by participant with respect to the errors made when they used ECA (with or without the TAP semantics). A source of errors when programming with ECA is *non sequitur*. In CCBL, there is always an active context, and the set of active contexts specifies the state of all devices. Consequently, users can always know what would be the state of a device when a context ends. Indeed, participants said that they preferred CCBL be-

cause "*it is not necessary to think about reverse actions as it is with ECA*".

5 Verification

End-user development means neophyte users will have to manipulate code. Since these programmers are not professionals developers, the editor should help them as much as possible. In this section we focus on the work we have done on providing means to check CCBL programs for errors. More precisely we focus on **unreachable code** and **dead code**.

It is a kind of error that is specific to the language and does not depend of the implementation or the presentation ; whether a section is reachable depends only of the logic. These errors are likely to appear as programs get longer and more complex.

5.1 Types of errors

Unreachable context

The first kind of unreachable code we identify is quite straightforward : unreachable context. It is when a context will never be active because its selector and the selector of at least one of his ancestors can not be true at the same time.

Inapplicable instructions

The second kind of error we want to address is what we call an inapplicable instruction. As said earlier, if the selector of a context A implicates the selector of a context B, we think A should have priority over B. But, if for some reason, B have priority over A, then inapplicable instructions can emerge. This is shown in the code sample 6. In this example, B has priority over A, but since the selector of A implicates the selector of B, whenever A is active, B is active. This means that when the instruction in A is applicable, in fact it is never applied because the instruction in B have priority. The only time when the instruction in A is applicable, there is always at least one instruction with higher priority, the instruction is never applied.

```
1  TV.isOn and Radio.isOn { // A
2    Radio.volume = 20; // Is never called
3  }
4
5  Radio.isOn { // B (have priority)
6    Radio.volume = 10;
7  }
```

Code sample 6 – Inapplicable instruction : B has priority over A and $A \implies B$. Radio volume will never be set to 20.

5.2 Detecting errors

Heptagon

Heptagon [12] is a synchronous dataflow language developed both at INRIA [13] by the Ctrl-A team [14] and at the ENS [15] by the PARKAS team [16]. Its grammar is based on **Lustre**

12. http://heptagon.gforge.inria.fr/
13. https://www.inria.fr/
14. https://team.inria.fr/ctrl-a/
15. http://ens.fr/
16. http://parkas.di.ens.fr/index.html

```
1 node main(inputEvent :bool) returns (outputValue :bool)
2   contract
3   reachable B
4 let
5   automaton
6     state A do outputValue = false until inputEvent then B
7     state B do outputValue = true until inputEvent  then A
8   end
9 tel
```

Code sample 7 – Simple Heptagon sample

which is presented in [Raymond, 2008]. As opposed to Lustre, Heptagon can handle states machines and include a very powerful behavioral contract system. In any Heptagon *node* we can specify a *contract*. It is composed of an optional **assume** part which describe behaviours of variables, a mandatory instruction and an optional **with** part who list editable variables. The instruction can either enforce a boolean equation stays true or a state of a state machine is reachable for example. After compilation, the resulting files can be used by other tools.

We use **ReaX** a controller synthesis tool developed at INRIA by the SUMO team [17]. It is based on **ReaVer** a tool developed by Peter Schrammel [18] [Schrammel and Jeannet, 2011]. ReaX can use the result of a Heptagon compiled program to produce a controller which uses the editable variables to ensure the contract is respected.

For our needs we are only interested in the *verification* part. Before producing a controller, ReaX performs verifications to ensure the contract can be respected, if not, the controller synthesis will fail. Without editable variables, ReaX must ensure the contract is respected *as is*. By using contract without variable, we can use ReaX as a verification tool.

The code sample 7 provide a simple example. At lines 5 to 7 is the description of a simple state machine. At lines 2 and 3 is the **contract**. It is part of what we called earlier the *assertion system*. In this example, the program is a state machine, with two states that switch at each input on the inputEvent channel. The program's return channel is true when the state is A and false when the state is B. To sum up, it is a basic switch.

Finding dead contexts

Checking CCBL dead or unreachable code, in CCBL, means finding specific relationships among contexts. For the *unreachable context* kind, we want to find contexts that can not be active at the same time and that are on the same branch. For the *inapplicable instruction* kind we want contexts with selectors that implicates one another with a reversed priority and that share at least one variable in the instructions of their code block.

The diagram in the appendix details the process describe in the next paragraphs and provide further details on the steps of the process. To proceed, we first developed a **translator** that converts CCBLjson programs into Heptagon state machines. CCBLjson is a standard we have developed for describing CCBL programs using json files. Then we developed **checkers** in python that performs verifications on the translated programs.

17. https://www.irisa.fr/sumo/
18. https://www.cs.ox.ac.uk/people/peter.schrammel/

```
* {
  TV.volume: 100;
  Radio.volume: 100;

  Phone.isInCall {
    TV.volume: 10;
    Radio.volume: 10;
  }

  TV.isOn and Radio.isOn {
    Radio.volume: 0;
  }
}
```

Code sample 8 – CCBL program before translation into code sample 9

The code samples 8 and 9a provide an example with parallel states machines. The heptagon code extracts come with a diagram of the state machine for better understanding.

Unreachable context Basically, the operation is to add a contract that checks the reachability of all positive states in the state machine. The checker, begins at the root of the tree and checks every context in the tree. The exploration of each branch stops at the first unreachable state as this means all its children are unreachable as well. For now, the checker has a minimal user interface, it only tells the user if there is dead code or not. In the future, we want it to be able to precisely tell the user which context is unreachable and which is not.

Inapplicable instruction For this operation, we needed to develop two others pieces of software : a program that which provide the list of contexts sorted by their respective priority level and a program that provides the list of all the ancestors of all contexts. The checker uses another contract to check boolean implication between selectors. For now, this program checks all the pairs of contexts possible in the tree but we already have another pieces of software that list common variables that will be soon plugged int that checker to optimize the search. As for the first checker, the user interface of this one is very limited and only returns the pairs of contexts with implication and reversed priority.

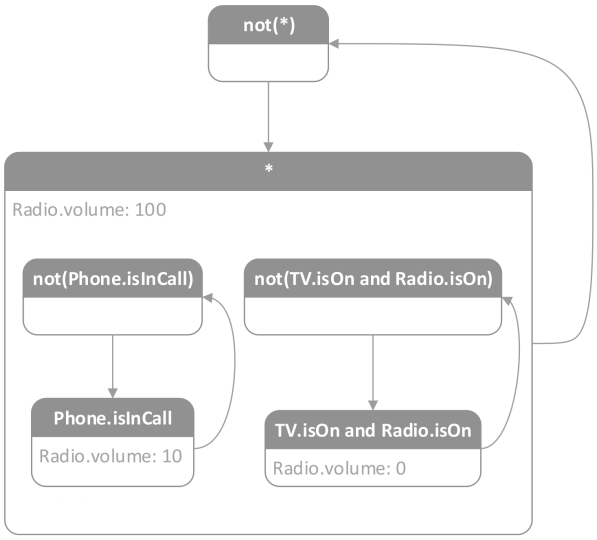
6 Conclusion and future works

In this work we identified flaws in end-user development languages for the smart home used in the industry. We designed a new language, CCBL, designed to overcome these flaws and provide a new approach to the field. This has lead to the publication [Terrier *et al.*, 2017b] in the 2017 edition of the conference ISEUD held in Eindhoven, Netherland (see appendix). We developed a simulator and a code editor. We performed a user based experiment that showed that non programmers do fewer errors with CCCBL than with ECA. This has lead to the publication [Terrier *et al.*, 2017a] in the 2017 edition of the conference EICS held in Lisbon, Portugal (see appendix). Finally, we designed and begin to develop tools to provide error correction support to users of CCBL.

For the future, our priority is to finish the verification tools and to plug them into our code editor in order to perform a user based experiment to attest of their usefulness. We think

```
node main(phoneisInCall:bool;tvisOn:bool;radioisOn:bool)
  returns(c1,c2,c3:bool;radiovolume:int)
  contract
  reachable c1
  var radiovolume1:int; radiovolumeD1:bool; radiovolume2:int;
        radiovolumeD2:bool; radiovolume3:int; radiovolumeD3:bool;
let
  radiovolume = radiovolume1;
  automaton
    state C1F
      do
        c1 = false;
        c2 = false;
        c3 = false;
        radiovolume1 = 0;
        radiovolumeD1 = false;
        radiovolume2 = 0;
        radiovolumeD2 = false;
        radiovolume3 = 0;
        radiovolumeD3 = false;
      until true then C1T
    state C1T
      do
        c1 = true;
        automaton
          state C2F
            do
              c2 = false;
              radiovolume2 = 0;
              radiovolumeD2 = false;
            until phoneisInCall then C2T
          state C2T
            do
              c2 = true;
              radiovolume2 = 10;
              radiovolumeD2 = true;
            until not phoneisInCall then C2F
        end;
        automaton
          state C3F
            do
              c3 = false;
              radiovolume3 = 0;
              radiovolumeD3 = false;
            until tvisOn & radioisOn then C3T
          state C3T
            do
              c3 = true;
              radiovolume3 = 0;
              radiovolumeD3 = true;
            until not (tvisOn & radioisOn) then C3F
        end;
        radiovolume1 =  if c2 & radiovolumeD2 then radiovolume2
             else if c3 & radiovolumeD3 then radiovolume3 else
             100;
        radiovolumeD1 = true;
      until not true then C1F
  end
tel
```

(a) Heptagon program after translation from code sample 8



(b) Hierarchical state machine of the code sample 9a

Code sample 9 – Processed state machine in Heptagon form and graphical form

CCBL still misses features, in particular in the event based edition of variables, we want to explore the possibilities in order to provide the possibility to, for example, increment or decrement variables in event contexts. We would also want to make an automatic translator of ECA to CCBL and if possible translate from CCBL to ECA. In the long term, we would like to explore the social and temporal aspects of End-User Development in the Smart Home.

Acknowledgements

I would like to express my thanks to my internship supervisors Dr. Alexandre Demeure and Dr. Gwenaël Delaval for giving me the opportunity to work with them and for their constant support. I would like to thank Jean-Jacques Parrain for his useful remarks and insights.

Références

James F Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11) :832–843, 1983.

AJ Brush, Bongshin Lee, Ratul Mahajan, Sharad Agarwal, Stefan Saroiu, and Colin Dixon. Home automation in the wild : challenges and opportunities. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 2115–2124. ACM, 2011.

Federico Cabitza, Daniela Fogli, Rosa Lanzilotti, and Antonio Piccinno. End-user development in ambient intelligence : a user study. In *Proceedings of the 11th Biannual Conference on Italian SIGCHI Chapter*, pages 146–153. ACM, 2015.

Federico Cabitza, Daniela Fogli, Rosa Lanzilotti, and Antonio Piccinno. Rule-based tools for the configuration of ambient intelligence systems : a comparative user study. *Multimedia Tools and Applications*, pages 1–21, 2016.

Julio Cano, Gwenaël Delaval, and Eric Rutten. Coordination of eca rules by verification and control. In *16th International Conference on Coordination Models and Languages*, page 16 p., Berlin, Germany, June 2014.

Joëlle Coutaz and James L Crowley. A first-person experience with end-user development for smart homes. *IEEE Pervasive Computing*, 15(2) :26–39, 2016.

Joëlle Coutaz, Alexandre Demeure, Sybille Caffiau, JR Courtois, T Flury, C Gérard, C Lenoir, K Petoukhov, P Reignier, C Roux, et al. Spok, an end-user development environment for smart homes. 2014.

Joëlle Coutaz, Alexandre Demeure, Sybille Caffiau, and James L Crowley. Early lessons from the development of spok, an end-user development environment for smart homes. In *Proceedings of the 2014 ACM International Joint Conference on Pervasive and Ubiquitous Computing : Adjunct Publication*, pages 895–902. ACM, 2014.

James L Crowley and Joelle Coutaz. An ecological view of smart home technologies. In *European Conference on Ambient Intelligence*, pages 1–16. Springer, 2015.

Rémy Dautriche, Camille Lenoir, Alexandre Demeure, Cédric Gérard, Joëlle Coutaz, and Patrick Reignier. End-user-development for smart homes : relevance and challenges. In *Proceedings of the Workshop" EUD for Supporting Sustainability in Maker Communities", 4th International Symposium on End-user Development (IS-EUD)*, page 6, 2013.

Scott Davidoff, Min Kyung Lee, Charles Yiu, John Zimmerman, and Anind K Dey. Principles of smart home control. In *UbiComp 2006 : Ubiquitous Computing*, pages 19–34. Springer, 2006.

Alexandre Demeure, Sybille Caffiau, Elena Elias, and Camille Roux. Building and using home automation systems : a field study. In *End-User Development*, pages 125–140. Springer, 2015.

Anind K Dey, Timothy Sohn, Sara Streng, and Justin Kodama. icap : Interactive prototyping of context-aware applications. In *International Conference on Pervasive Computing*, pages 254–271. Springer, 2006.

Daniela Fogli, Rosa Lanzilotti, and Antonio Piccinno. End-user development tools for the smart home : A systematic literature review. In *International Conference on Distributed, Ambient, and Pervasive Interactions*, pages 69–79. Springer, 2016.

Emeric Fontaine. *Programmation d'espace intelligent par l'utilisateur final*. PhD thesis, Université Grenoble Alpes, 2012.

Andrew Frye, Michel Goraczko, Jie Liu, Anindya Prodhan, and Kamin Whitehouse. Circulo : Saving energy with just-in-time hot water recirculation. In *Proceedings of the 5th ACM Workshop on Embedded Systems For Energy-Efficient Buildings*, pages 1–8. ACM, 2013.

Manuel García-Herranz, Pablo A Haya, and Xavier Alamán. Towards a ubiquitous end-user programming system for smart spaces. *J. UCS*, 16(12) :1633–1649, 2010.

Jonna Häkkilä, Panu Korpipää, Sami Ronkainen, and Urpo Tuomela. Interaction and end-user programming with a context-aware mobile application. *Human-Computer Interaction-INTERACT 2005*, pages 927–937, 2005.

Seth Holloway and Christine Julien. The case for end-user programming of ubiquitous computing environments. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*, pages 167–172. ACM, 2010.

Justin Huang and Maya Cakmak. Supporting mental model accuracy in trigger-action programming. In *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, pages 215–225. ACM, 2015.

Henry Lieberman, Fabio Paternò, Markus Klann, and Volker Wulf. *End-user development : An emerging paradigm*. Springer, 2006.

Gabriella Lucci and Fabio Paternò. Understanding end-user development of context-dependent applications in smartphones. In *International Conference on Human-Centred Software Engineering*, pages 182–198. Springer, 2014.

Sarah Mennicken, Jo Vermeulen, and Elaine M Huang. From today's augmented houses to tomorrow's smart homes : new directions for home automation research. In *Proceedings of the 2014 ACM International Joint Conference*

on Pervasive and Ubiquitous Computing, pages 105–115. ACM, 2014.

Chandrakana Nandi and Michael D Ernst. Automatic trigger generation for rule-based smart homes. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*, pages 97–102. ACM, 2016.

John F Pane, Brad A Myers, et al. Studying the language and structure in non-programmers' solutions to programming problems. *International Journal of Human-Computer Studies*, 54(2) :237–264, 2001.

Pascal Raymond. Synchronous program verification with lustre/lesar. In *Modeling and Verification of Real-Time Systems*, chapter 6. ISTE/Wiley, 2008.

Peter Schrammel and Bertrand Jeannet. Logico-numerical abstract acceleration and application to the verification of data-flow programs. In *Static Analysis Symposium, SAS'11*, volume 6887 of *Lecture Notes in Computer Science*, pages 233–248. Springer, 2011.

Lénaïc Terrier, Alexandre Demeure, and Sybille Caffiau. Ccbl : A language for better supporting context centered programming in the smart home. 2017.

Lénaïc Terrier, Alexandre Demeure, and Sybille Caffiau. Ccbl : A new language for end user development in the smart homes. Springer, 2017.

Blase Ur, Melwyn Pak Yong Ho, Stephen Brawner, Jiyun Lee, Sarah Mennicken, Noah Picard, Diane Schulze, and Michael L Littman. Trigger-action programming in the wild : An analysis of 200,000 ifttt recipes. 2012.

Blase Ur, Elyse McManus, Melwyn Pak Yong Ho, and Michael L Littman. Practical trigger-action programming in the smart home. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 803–812. ACM, 2014.