Automatically proving concurrent programs running on weak memory

08/27/13

Tristan CHARRIER

Rapport de stage de 2e année du Magistère informatique et télécommunications, ENS Cachan/Bretagne et Université Rennes 1

Stage effectué du 15 mai 2013 au 31 juillet 2013

Advisor : Jade ALGLAVE (UCL)

Abstract :

Keywords : weak memory, Hoare logic, concurrency.

Introduction

Multi-core processors are ubiquitous in computers nowadays: in fact, it is now impossible to enhance the frequency of processors because of heat problems, so it is mandatory to increase the number of cores for better performances. Yet, subtle behaviors occur in such processors, as showed in [1]. Internal optimizations within these processors make them shady to the programmer. Therefore, rigorous models are needed to treat these problems formally. Jade Alglave, in her papers [2] [**?**], has created a formal model using graphs to show these behaviors. This model was dealing with executions witnesses, defining whether they were correct or not by defining acyclicity constraints on the graph generated by the execution witness.

The next step in the research was to define a logic to make the checking more systematic. The PPLV team is currently developing one: the lace logic. This logic, strongly inspired from Hoare logic, consists on carrying more information on assertions to show subtle behaviors in weak memory programs .

To explain the contribution of the internship, we will explain in a first time what the general idea of the logic is, with some examples to show the hardness of the problem, then we will explain the logic itself, and finally we will describe the checker, explaining what problems were hard in its implementation, and showing some tests.

1 Weak memory: computational hardness

1.1 A small example

The way of programming usually known is sequential programming : the program is executed instruction by instruction, and two instructions shouldn't be switched. Here, we consider concurrent computing, where several programs, called threads, are running and can share accesses to memory slots (reads and writes), symbolized by variables. They have internal memory, called registers, which cannot be shared with other threads. In this report, the registers are shown with the letter 'r' followed by a number and variables are everything else. For instance, let us consider this small program :

```
1 initial state : x = y = 0
2 ---------------------------
3       Thread 0 |      Thread 1
4  (a) x := 1;   | (c) y := 1
5  (b) r1 := y   | (d) r2 := x
```

In sequential reasoning (it means here that (a) must be before (b) and (c) before (d)), there are three possibilities for $(r1, r2)$ at the end of the program :

- (1,1), by doing (a) $\rightarrow$ (b) $\rightarrow$ (c) $\rightarrow$ (d) for example

- (1,0), by doing (c) → (d) → (a) → (b)
- (0,1), by doing (a) → (b) → (c) → (d))

$(r1, r2) = (0, 0)$ shouldn't be possible, because it would mean that we have done (d) before (a), therefore, because we are in sequential reasoning, we should have done (c), so after executing (b), the couple $(r1, r2)$ shoud have been (0,1).

The major issue is that in weak memory, this statement is false : (0,0) is observed in this program. This can be interpreted by the fact that a write isn't instantaneous : when the program does $x := 1$, the value 1 isn't affected directly, it is buffered, so a read to this location can produce a 0 or a 1.

A very complex question arises from this example : how to program correctly if there are random effects like this one ? In fact, in Power-PC architecture, which is the one studied in the PPLV team, the programer can implement some tricks to deal with these issues : it is the mecanism of barriers. The barriers forbid some optimizations in the PPC (Power-PC) architecture, in this report, we will focus on two :

- lwsync barrier: it imposes current writes to be stored into memory and not just buffered.
- isync barrier: it forbid forward-computing in branching: for example, let us consider the command if $x = 0$ then C_1 else C_2. Without ths isync, the processor can begin to execute C_1 before evaluating the condition $x = 0$. It is an optimization to avoid loosing time. If $x \neq 0$, the processor gives up on evaluating C_1, and evaluates C_2. But there is a major issue if C_1 contains writes to variables : it can make the program false. The isync barrier is useful to avoid forward-looking of the processor in this case : we have to evaluate $x = 0$ before doing anything else with this barrier.

For instance, let us take the previous example with an lwsync :

```
1 initial state : x = y = 0
2 ---------------------------------
3       Thread 0 |      Thread 1
4  (a) x := 1;   | (d) y := 1
5  (b) lwsync    |
6  (c) r1 := y   | (e) r2 := x
```

Because of the lwsync, $(x, y) = (0, 0)$ becomes impossible: to do (c), x must be associated to 1 and if (e) was executed, y must be 1 too, therefore (0,0) is strictly impossible.

We can use too an other trick, called dependencies, to impose read orders: in fact, in a thread, the reads, $r := e$, can be reordered for optimization issue. The effect of a program can seem very random then, but we can command an order, by doing this :

```
(a) r1 := 0;
(b) r0 := r1 xor r1
(c) r2 := 1 + r0
```

Basically, at the end of this program, $r1 = r0 = 0$ and $r1 = 1$, but by doing this, we have made a dependency between $r2$ and $r0$, therefore (c) must be after (b). With same reasoning, (b) must be after (a). With the trick of (b), we have make the (c) line come after (a) in the execution. It will me summarized as a single command in the rest of the report:

```
(a) r1 := 0;
(c) r2 := 1 dep r1
```

To introduce what we do in the logic used in the internship, we have to introduce a reasoning about these programs : the rely/guarantee paradigm.

1.2 Rely/guarantee paradigm

It is very hard to make a reasoning with all the threads at once, the idea, explained in [3], is to consider interferences between threads while doing a proof thread by thread: it is the rely/guarantee paradigm. Here, the only memory shared by threads is variables, to the only type of commands that can generate interferences are writes : $v := e$.

Let us consider the previous example again :

```
initial state : x = y = 0
-------------------------------
     Thread 0 |      Thread 1
 (a) x := 1;  | (d) y := 1
 (b) lwsync   |
 (c) r1 := y  | (e) r2 := x
```

The only two commands that can generate interferences in that case are $x := 1$ and $y := 1$. To do a correct proof of this program, we need to consider this interference : it would be highly undesirable if an assertion in the proof of thread 0 becomes false because of the fact that we have $y := 1$ in thread 1.

To explain this concept of interference, we have to define two notions properly : the strongest precondition in sequential case of an assignment, and stability of a formula.

Strongest precondition in sequential case of an assignment *Let f be a formula, and $x := e$ an assignment. Then the strongest precondition after executing $x := e$ with precondition f is $sp(f, x := e) = \exists x' f[x \leftarrow x'] \wedge (x = e[x \leftarrow x']$ where x' is a fresh variable. The same relation holds with $r := e$.*

The interference of the assignment is then symbolized by the notion of stability : a formula is stable against the interference if after executing the assignment, the formula still holds. This is a rigorous definition.

Stability *A formula φ is stable against $(f, x := e)$ where f is the precondition of $x := e$ if and only if $sp(\varphi \wedge f, x := e) \Rightarrow \varphi$, which is the same than saying the Hoare triplet $\{\varphi \wedge f\}x := e\{\varphi\}$ is valid.*

The interference of the assignment $x := e$ is modeled here by the set of formulas which are stable or not against $(f, x := e)$. The way to define f will be refined later, but for now, we'll consider a simple case.

Let us take a small example : $(true, x := 1)$. The formula $x = 0$ is not stable against this interference because

$$\begin{aligned} sp(true \wedge x = 0, x := 1) &= sp(x = 0, x := 1) \\ &= \exists x'(x = 0)[x \leftarrow x'] \wedge x = 1[x \leftarrow x'] \\ &= \exists x' x' = 0 \wedge x = 1 \end{aligned}$$

which is equivalent to $x = 1$. But it is false to say that $x = 1 \Rightarrow x = 0$, so $x = 0$ is not stable against $(true, x := 1)$.

On the contrary, the formula $x = 0 \vee x = 1$ is stable against $(true, x := 1)$, because with the same reasoning, $sp(x = 0 \vee x = 1 \wedge true, x := 1) \equiv x = 1$ which implies $x = 0 \vee x = 1$.

In each thread, we define what we call a guarantee, which contains at least all the interferences of the thread. Each thread must be then stable at each step against guarantees of other threads. But it is not sufficient: interferences within the thread can occur, with other writes. These interferences are contained in the rely , which is intern in a thread. In fact, we will not consider just formulas in the proof, but formulas and their relies.

Formally, the guarantee and the rely are set of interference $(f, x := e)$, lists in the checker. Let us consider this example :

```
1 initial state : x = y = 0
2 ------------------------------
3       Thread 0 |      Thread 1
4  (a) x := 1;   | (d) y := 1
```

```
(b) z := 2    |
(c) r1 := y   | (e) r2 := x
```

A guarantee for thread 0 must contain $(f, x := 1)$ with f implied by the initial state here, so $x = y = 0 \Rightarrow f$. In thread 0, an assertion can have interferences because of $x := 1$ and $z := 2$, so we need to put relies that contain these interferences.

The last notion we need for the relies is the intersection:

Rely intersection *Let* $R_1 = \{(f_{x_1}, x := e)\}$ *and* $R_2 = \{(f_{y_2}, y := e')\}$ *two relies, then* $R_1 \cap R_2 = \{(f_{x_1} \wedge f_{x_2}, x := e)\}$ *with the convention* $f_{x_i} = false$ *if there* $x := e$ *does not appear in* R_i

2 Lace logic

As we have seen in the previous section, weak memory makes the programmer deal with heavy computational issues. A logic is in development in the PPLV team, let us explain what this logic consist on.

2.1 Definitions

To understand properly how this logic works, we need in a first step to define some notions. To begin, this is the syntax of programs :

Program syntax *A program is defined by*

C ::==		$x := r$	(write from a register)
	|	$x := r$ dep r	(write with an artificial dependency on a register)
	|	$r := x$	(read from a variable)
	|	$r := x$ adep r	(read with an artificial dependency on a register)
	|	$r := Op(r)$	(read from an operation on the registers)
	|	$C; C$	(sequence)
	|	if (b) $\{C\}$	(where b is a condition on the registers)
	|	if (b) $\{C\}$ *else* $\{C\}$	
	|	while (b) $\{C\}$	
	|	lwsync	(memory barrier)
	|	isync	(branching barrier)
	|	sync	(absolute barrier)

On this program, we want to check assertions on the form of Hoare triplets for each thread, $\{A\}C\{B\}$, where C is a program. let us define properly what is a Hoare triplet. To begin, we have to explain what A and B are here. As we have seen before, we have to carry

much information to cover all the behaviors we want to show. That's why A and B are not assertions, but what we call here CDMP components. In the next definition, the couples (f, R) are such that f is a formula and R a rely as seen before.

CDMP component *A CDMP component is a 4-uple where:*

- *C (Conditionnal) is a couple* (f, R)*. It stores all the information we have about the if or while previously done.*
- *D (Declarations) is a function* $r \mapsto (f, R)$ *where* r *is a register. It stores assertions about the registers in the program, and is symbolized by a list of couple* $(r, (f, R))$ *in the proof checker.*
- *M (Memory) is a function* $v \mapsto (f, R)$ *where* v *is a variable. It is similar to D, but for variables.*
- *P (Propagation) is a special line: it is a function* $v \mapsto (f, R)$ *where* v *is a variable, and stores information about the propagation of the variable in the program.*

It is these components we manipulate in the programs. But it not sufficient to implement the rely/guarantee paradigm, because we don't have any guarantee. Each thread contains two more information : initial information I (common to all threads), and a guarantee G. Here, we manipulate then proof of the form $(I, G, \{A\}C\{B\})$.

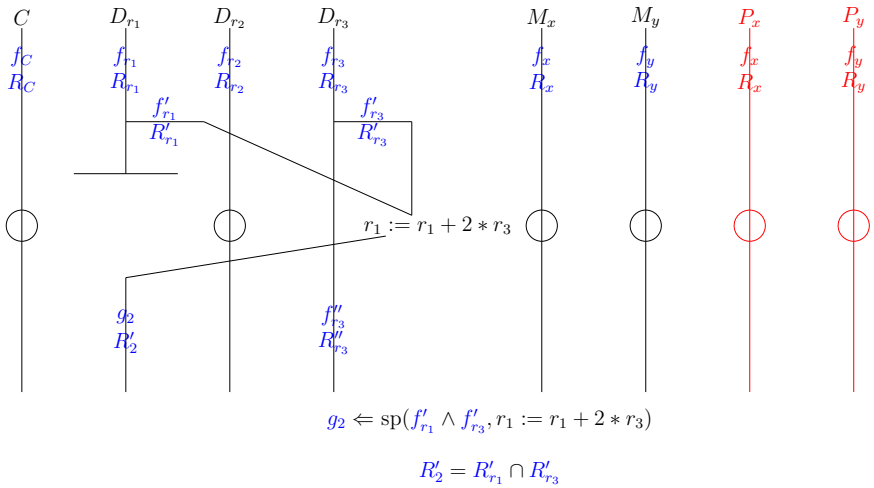
We have to define how to declare a Hoare triplet valid.

Hoare triplet and validity *A Hoare triplet* $(I, G, \{A\}C\{B\})$ *is valid if and only if :*

- $\forall f$ *which appears in* A, $I \Rightarrow f$
- $\forall$ *assignment* $x := e$ *appearing in the program, the interference of this assignment appears in the guarantee. (what the interference is will be explained after)*
- $sp(A, C) \Rightarrow B$*, it means that the CDMP component obtained as result of* C *with precondition* A *implies every component in* B

sp is the strongest postcondition of the program C, we will define it in the following subsections.

The formalism of this logic being quite heavy, so we will use drawings to explain it in. We have then to explain some conventions in the drawing. Here is an example:



In a CDMP component, C, D and M are in black and P in red because of his special role. The content of the component is in blue for C, D and M and in red for P.

In the C line, the circle means that we have to delete information about r_1 in f_C, which is done by replacing f_C by $\exists r_1 f_C$. We have to do the same for D_{r_2}, M_x, M_y, P_x, P_y.

The horizontal bar in D_{r_1} symbolize that we give up on this information, we don't need it anymore.

In D_{r_3}, we have a separation of the line in two, it is what we call a fork. The fork rule is :

Fork rule *If* (f, R) *is forked in* (f', R') *and* (f'', R'') *then the following statements must apply:*

- $f \Rightarrow f'$
- $f \Rightarrow f''$
- $R = R' \cap R''$

What is at the bottom of the drawing is additionnal constraints the proof need to verify in order to be correct. Now that everything is defined, we will explain the logic.

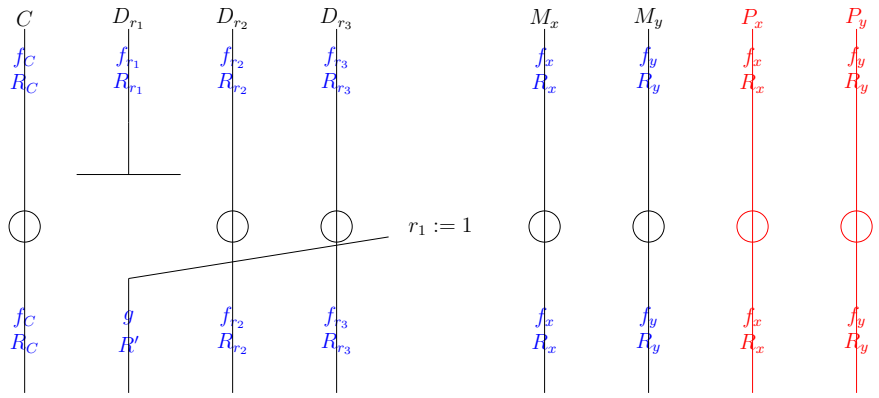
2.2 Reads and writes

To understand these rules, we take back the previous example :

```
1 initial state : x = y = 0
2 ---------------------------
```

```
     Thread 0  |     Thread 1
(a) x := 1;    | (c) y := 1
(b) r1 := y    | (d) r2 := x
```

There are two types of commands here : writes (a and c) and reads (b and d). Let us begin with reads.

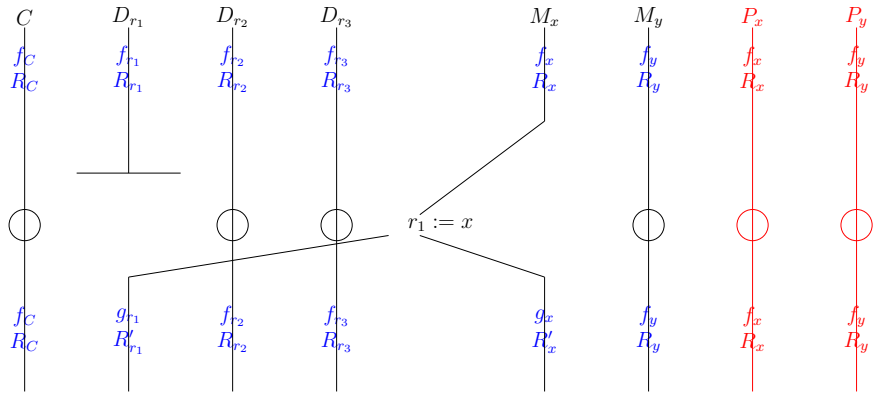


$$g \Leftarrow \mathrm{sp}(true, r_1 := 1)$$

$$R' = \varnothing$$

This case is quite simple: for $r_1 := 1$ (more generally, $r_1 := c$ with c a constant), we have to give up on the information for r_1, because it won't be good anymore, and replace it with the strongest postcondition of true with $r_1 := 1$, in the sense of section I.2. For all the other lines, the previous information of r_1 must be deleted because it doesn't hold anymore, so we have to do $\exists r_1 f$ for all f in other lines than D_{r_1}.

The second case is $r_1 := x$ with x a variable:



$$g_{r_1}, g_x \Leftarrow \mathrm{sp}(f_x, r_1 := x)$$

$$R_x = R'_{r_1} \cap R'_x$$

For this case, we have to add the information on M_x to the strongest postcondition, because it contains what we need to evaluate the condition.

2.3 Barriers and conditions

3 A checker for this logic

We have a checker for this logic !

3.1 Conventions

3.2 A simple test : message passing

3.3 A more complicated test

Conclusion

Acknowledgments

First of all, I am very grateful to Jade Alglave for welcoming me warmly in the research team, and being available for all my questions. It was a real pleasure to work in such a nice environment. I want to thank Richard Bornat and Matthew Parkinson too for their enlightenments about the subject, I don't think I would have managed to realize the internship properly without all the discussions I had with them. Finally, I wish to thank all the staff of the UCL, very welcoming and available too and patient.

References

[1] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O Myreen. x86-tso: a rigorous and usable programmer's model for x86 multiprocessors. *Communications of the ACM*, 53(7):89–97, 2010.

[2] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. Fences in weak memory models (extended version). *Formal Methods in System Design*, 40(2):170–205, 2012.

[3] Viktor Vafeiadis. Modular fine-grained concurrency verification. Technical Report UCAM-CL-TR-726, University of Cambridge, Computer Laboratory, July 2008.