Phd Computer Science Proposal

Akshay Gopalakrishnan, Clark Verbrugge

McGill University

October 5, 2021

1 Introduction

This research statement is for pursuing a PhD in Computer Science under the supervision of Dr. Mark Batty. The research topic is on memory consistency models. A memory consistency model specifies the possible outcomes a concurrent program using shared memory can have when executed. For the sake of performance, several hardware features were introduced, which sacrifice to a great extent reasoning with concurrent programming using interleaving semantics. Relaxed memory models were introduced to describe such semantics. These models, however have been victim to underspecified or prose specification, which has led to various problems such as compilation correctness, program transformations and a lack of understanding of the possible outcomes a program may have.

I propose two potential directions that can be taken to address the research problems in this domain specified by Mark Batty [1] in his research statement.

2 Concurrency - Bread and Butter for Performance

Almost every system we use for our day-to-day lives rely on concurrent computation. Right from our mobile phones to our personal desktop which can seemingly perform multiple tasks at the same time, concurrent computations have become part of our daily life. The amount of performance concurrent computation has given us is something that is near impossible to part with. But an important question remains, can we understand and utilize concurrency at its best for the various domain specific compuatations we intend to do?

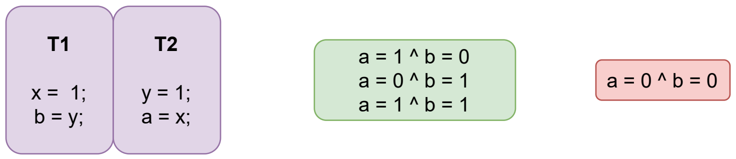

Figure 1: Program with its allowed (green box) and disallowed(red box) behaviors under SC

3 Memory Consistency Models - A way to tame Out-of-Order computation

Concurrent programs take advantage of *out-of-order* execution. Intuitively, this means that more than one unrelated computations can be done "simultaneously" without having any fixed order in which they should happen. This results in concurrent programs having multiple different outcomes. In terms of concurrenct compuations using shared memory, the possible outcomes are described by a *memory consistency model*.

Desipte the elaborate different ways in which we can think of writing programs that leverage the *out-of-order* notion, most programmers are accustommed to reasoning about compuations sequentially. Sequential Consistency(SC), which was a memory model first formulated by Lamport et al. [7], gives programmers this exact sequential reasoning for their programs running in a multiprocessor environment.

Though SC seems to be a very intuitive way to reason about programs using shared memory, it does not reflect how different processors do their computation. Hardwares in today's era have multiple features such as caches, read/write buffers, speculative execution, etc. which are designed to give us the performance we quite often take for granted these days. Usage of such features however, does not respect SC reasoning. For instance, consider the sample program in Fig 1 (in common literature known as WB or write buffering litmus test) and its possible outcomes under sequential consistency.

The disallowed outcome, however is something that can be seen for instance, in an x86 processor. It is possible due to usage of write buffers that delays the visibility of writes to x and y to other processors / threads.

For high level languages, SC also implies that basic program transformations that the compiler aggressively performs may be invalidated. For instance, the above disallowed outcome under SC can be justified by the compiler by just reordering independant reads and writes as shown in Fig 2.

The above concerns called for concise formal descriptions of the behaviors shown by hardwarwe. Using such a description, we then can, giving enough freedom for compilers, give a formal description for the behaviors we would want our high level languages give the users to leverage. As simple as the above objective sounds, it is still quite an open ended problem.

Hardware vendors are notoriously known to define their system using in-

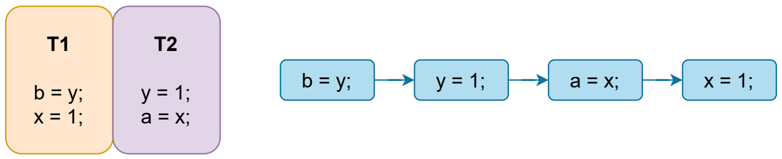

Figure 2: The above program with read-write reordered in the first thread, resulting in justifying the disallowed outcome under SC

formal prose specification. This has resulted in misunderstanding of what the hardware can do, which has resulted in many problems in the other layers[9]. For high level languages like Java, the initial versions of the memory model rendered even basic redundant elimination or introduction transformations invalid. This called for several revisions of the model [8]. To date, the specificaiton of the Java memory model is still not completely clear and formal[3]. The C11/C++ memory model also faced similar problems, in addition that the model gives no guarantees for programs with Data Races, unlike that of Java [2], [10]. This enables the compiler to perform any possible transformations, which may lead to values that appear "out-of-thin-air" (although the standard out of thin air problem is described as a subset of the one we refer to. Its definition is present in the works done on C11). This breaks the fundamental criteria based on which a programmer can reason with his/her programs, although its restriction would certainly disallow certain aggressive optimizations[11]. Quite recently, JavaScript have also come up with their own memory consistency model, with the catch being that it is based on mixed-size memory accesses. The validity of program transformations with such mixed-size accesses became a topic of investigation in my Masters program, part of which is being published [4].

A lack of clear consice descriptions of the models in both hardware and software layer brought forth the daunting problem of correct compilation. Having incorrect compilation breaks the holy contract of the program being able to do what the user intends. The difficulty with the introduction of relaxed memory behaviors is that in practice, unexpected outcomes of programs happen so infrequently that it is difficult to address the correctness of compilation. For instance, work done by Conrad et.al [12] showed that an earlier version of the JavaScript memory model, disallowed behaviors that were observable when JS programs were run in ARMv8. Lahav et.al [6] showed a counter example to emphasize that the compilation of C11 programs to POWER were not entirely correct, even though it was formally proven in this respect.

In the era where humans are working towards technological marvels such as self driving cars, autopilot aircrafts and spacecrafts to Mars, informal specifications itself is a problem. With the need for quick response of such systems, performance becomes an important factor and hence usage of realxed memory features becomes paramount. But their consequence due to above problems is in my view relatively more challenging as such relaxed behaviors that can occur in our programs is seen quite infrequently; one in a million or even billion runs.

Hence errors can be quite devastating which might take forver to idenitfy.

4 Focus

In the recent times, use of specialized hardware is growing exponentially. The seemingly effective usage of specific hardware for performing specific computations of our program has shown to be overall a great benefit for performance. The most common example of this would be that of computer games, wherein the rendering ahd the physics of the game are distributed between the central processing unit(CPU) and the graphical processing unit(GPU).

The focus of this proposal is on the correctness of heterogenous compilation of programs using relaxed memory; program whose differnt components are executed by different hardware having different memory consistency models. This is more difficult as we still do not have a prescribed way of comparing two differnt memory consistency models in addition to the established fact that specifications are most often ill defined. Quite concurrently, this brings the problem of compositional description of relaxed memory behaviors; something that has not been addressed.

In the above respect, I proposal two potential directions that can be taken. The first one is towards having a compositional description of memory models. The second one is towards have a transformational description of memory models.

4.1 Method 1: Compositional description of memory models

The compositionality problem defined in [1] questions whether it is possible to formally define a memory consistency model by composition of program components. Denotationally, the direction suggested is to define histories given a program component P and a context C. Histories in my understanding is the set of observable behaviors possible given a program and a memory model under which it is written. With the above information, I propose the following alternative problem to address compositionality with a possible approach.

Main Problem Can relaxed memory models be compositionally defined using program transformations.

Compositions using transformations To start investigation in this respect, I propose revisiting the role/validity of redundancy elimination/introduction w.r.t. observable behaviors of programs. The validity of the program transformations for any model is subject to the condition that the resultant program has observable behaviors to be a subset of the original.

Having this knowledge, we could for instance, consider a program component P_C and its other component that consists of exactly one program instruction e. Their composition in terms of observables can then be viewed as performing redundancy introduction of event e to program component P_C. Conversely, given a program P, we can divide it into components by performing redundnacy elimination, which would give us the exact behaviors that are captured by each component. Although here, we are viewing these transformations not in terms of compiler optimizations, rather in a perspective of what new observable behaviors introducing/elimination events can bring forth for a program given a memory model/

Doing this on a more fine grained level of can help us construct or divide a program from a perspective of set of observables. This could give us an a way to compositionally define a memory model.

The above approach may also help us prove the following: **It is possible to compositionally define relaxed memory models if and only if redundancy elimination / introduction is sound.**. If the answer to this is a firm **yes**, then we have a definite class of models that we know for sure we can define compositionally, by only validating two program transformations.

Shortcoming The above approach is not without its possible two main shortcomings as listed below:

- We would need to assume or prove that other transformations can be defined in terms of elimination and introduction. This may or may not be possible. For example, it may not be possible to define parallel composition.
- Program level constructs such as conditionals and loops may not be defined in terms of transformations of introduction and elimination.

Proving that the above shortcomings may not be so is another useful result in itself, and can be another direction of investigation.

4.2 Method 2: Deriving Transformational Specifications of Models

This approach could be considered an extension of a recent work done on Transoformational specification of memory models. The work done by Lahav et.al [5] show that TSO is equivalent to *write-read reordering* and *read-after-write elimination* over SC. They show how such a result helps in proving compilation correctness. Taking their results, the above problem can be addressed as followss:

For this, we take a relatively concrete example of heterogenous compilation from [1]. Consider a program P having two compoenets; one run by x86 (P_{x86}) and one by NVdia PTX (P_{PTX}). A correct compilation would require that the composition of these two program components having observable behaviors under their respective hardware is a subset of the original program under its memory model (eg: OpenCL).

$$[[P_{x86}]]_{x86} \bullet [[P_{PTX}]]_{PTX} \subseteq [[P_{OpenCL}]] \tag{1}$$

First, we need to find a transformational specification for $x86$ and PTX, which can be defined over a common stronger memory model SM. Having this, we would have the following two result from [5]:

$$[[P_{x86}]] \subseteq \bigcup \{[[P'_{x86}]]_{SM} \mid P'_{x86} \ s.t \ P_{x86} \mapsto P'_{x86}\} \tag{2}$$

$$[[P_{PTX}]] \subseteq \bigcup \{[[P'_{PTX}]]_{SM} \mid P'_{PTX} \ s.t \ P_{PTX} \mapsto P'_{PTX}\} \tag{3}$$

Here P'_{x86} and P'_{PTX} are programs that result by using valid program transformations over the stronger memory model. From [5], the requirements for compilation correctness, then reduces to showing the following three properties between SM and $OpenCL$:

- Compilation should be correct for the strong model, given the components run by $x86$ and PTX.

$$\forall P \ [[P_{x86}]]_{SM} \subseteq [[P]]_{OpenCL} \tag{4}$$

$$\forall P \ [[P_{TSO}]]_{SM} \subseteq [[P]]_{OpenCL} \tag{5}$$

- There should be a set of source program transformations valid under the $OpenCL$ memory model.

$$\forall P, P' \ .P \mapsto P' \Rightarrow [[P']]_{OpenCL} \subseteq [[P]]_{OpenCL} \tag{6}$$

- The above transformations should capture the set of all target program's transformations.

$$\forall P, P'_{x86}. \ P_{x86} \mapsto P''_{x86} \Rightarrow \exists P'.P'_{x86} = P''_{x86} \ \wedge \ P \mapsto P' \tag{7}$$

$$\forall P, P'_{PTX}. \ P_{PTX} \mapsto P''_{PTX} \Rightarrow \exists P'.P'_{PTX} = P''_{PTX} \ \wedge \ P \mapsto P' \tag{8}$$

This can help us reason with programs meant to be run on a heterogenous system non-compositionally.

Shortcomings This approach, however, would require us to:

- First find transformational specification of memory models, which may not be possible. This itself is an open problem, a subset of which is addressed by [5].
- Second, it may not even be possible to find a common stronger model over which two memory consistency models can be defined transformationally.

The above problems however, can be first investigated by keeping the stronger model as SC. I find this to be an appropriate choice because we tend to justify program transformations in example programs by finding a transformed program whose sequential interleaving can have the weak behavior.

5 Rough Roadmap

If the first direction of investigation is considered, here would be a tentative timeline:

- Start with a target memory model which is prefix closed and for which elimination and introduction transformaitons are sound. (eg: C11)
- Define compositionally using elimination and introduction its memory model.
- Investigate whether it is possible to define major class of program transformations in terms of elimination and introduction.
- Use the wisdom of the above two, to investigate whether we can assert the possibility of defining compositional memory model by only proving validity of elimination and introduction of the target model.

For the second direction of investigation, here would be a tentative timeline:

- Start with two basic target program memory models like TSO and $Coherence$. Investigate whether Coherence can be defined by a set of transformations over SC. (TSO we have from existing work.)
- Consider the parent program's memory model, from which we want to compile to target memory models we choose above.
- Prove that compilation correctness of composition of the program components under TSO and $Coherence$ reduces to proving Correctness over SC defined by conditions given in [5].
- Once the above pipeline is set, consdier instead of $Coherence$, the PTX memory model, investigating the same way as above.

References

[1] Mark Batty. "Compositional Relaxed Concurrency". In: *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences (2104)* 375 (2017).

[2] Mark Batty et al. "Mathematizing C++ concurrency". In: *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011.* 2011.

[3] John Bender and Jens Palsberg. "A formalization of Java's concurrent access modes". In: OOPSLA (2019).

[4] Akshay Gopalakrishnan and Clark Verbrugge. "Reordering under the ECMAScript Memory Consistency Model". In: *Workshop on Languages and Compilers for Parallel Computing (LCPC)* (In Press) (2020).

[5] Ori Lahav and Viktor Vafeiadis. "Explaining Relaxed Memory Models with Program Transformations". In: *FM 2016: Formal Methods - 21st International Symposium, Limassol, Cyprus, November 9-11, 2016, Proceedings*. 2016. URL: `https://doi.org/10.1007/978-3-319-48989-6%5C_29`.

[6] Ori Lahav et al. "Repairing sequential consistency in C/C++11". In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*. 2017.

[7] Leslie Lamport. "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs". In: *IEEE Trans. Computers* (1979).

[8] Jeremy Manson. "The design and verification of Java's memory model". In: *Companion of the 17th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, OOPSLA 2002, Seattle, Washington, USA, November 4-8, 2002*. 2002.

[9] Scott Owens, Susmit Sarkar, and Peter Sewell. "A Better x86 Memory Model: x86-TSO". In: *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*. 2009.

[10] Viktor Vafeiadis et al. "Common Compiler Optimisations are Invalid in the C11 Memory Model and what we can do about it". In: *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*. 2015.

[11] Clark Verbrugge, Allan Kielstra, and Yi Zhang. "There is nothing wrong with out-of-thin-air: compiler optimization and memory models". In: *Proceedings of the 2011 ACM SIGPLAN workshop on Memory Systems Performance and Correctness: held in conjunction with PLDI '11, San Jose, CA, USA, June 5, 2011*. 2011.

[12] Conrad Watt et al. "Repairing and mechanising the JavaScript relaxed memory model". In: *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*. 2020.