(Relative) Safety Properties for Relaxed Approximate Programs

Michael Carbin
MIT CSAIL
mcarbin@csail.mit.edu

Martin Rinard
MIT CSAIL
rinard@csail.mit.edu

ABSTRACT

Researchers have recently begun to explore a new class of program transformations called *approximate program transformations*. These program transformations take an existing program and produce a new, *relaxed approximate program* that trades accuracy of its results for increased performance.

In this paper, we explore how developers can use *relational* reasoning to verify relative properties of relaxed approximate programs in that if their original program satisfies a property (such as memory safety), then the relaxed program also satisfies the property. Such relational reasoning enables developers to transfer their reasoning about the original program over to verify the relaxed.

1. INTRODUCTION

Researchers have recently investigated a number of program transformations that trade the accuracy of a program's results for performance. *Approximate program transformations*, such as loop perforation [6, 9], approximate memories and data types [7, 13], approximate function memoization [3], dynamic knobs [5], and synchronization reduction [8, 14, 12, 10] take existing programs and produce new, relaxed approximate programs that approximate the behavior of the original and operate with higher performance (i.e., decreased execution time or power consumption).

A key aspect of the development of relaxed approximate programs through the application of approximate program transformations is safety. Specifically, when a developer first develops his or her initial program, he or she typically establishes several safety properties about the behavior of the program. For example, the developer may use program analysis or his or her own reasoning (code review) to establish that all pointer dereferences in the program are within bounds of an allocated object. However, unlike as in the case for traditional semantics-preserving compiler transformations, approximate program transformations specifically augment the semantics of the original program to produce a relaxed approximate version, which therefore jeopardizes the validity of the developer's reasoning on the final, approximate program.

In this paper we advocate that developers use relational reasoning — in the form of relational proofs that relate the behavior of the original and relaxed program — to establish that their desired safety properties still hold of the relaxed variant of their program. Relational reasoning enables us to define and provide a semantics for the *relative safety* of a relaxed program: if the original program satisfies a safety property, then the relaxed program also satisfies a safety property (given an appropriate relational proof). While this stands in contrast to the standard unconditioned definition of safety (i.e., all executions of the program satisfy a property), relative safety enables a developer to formally transfer their original reasoning about the program — which may be established through informal techniques, such as testing or code review — over to the relaxed program in cases where an approximate program transformation does not interfere with the validity of their reasoning.

Synchronization Reduction. In this paper we investigate relative safety in the context of *synchronization reduction* (SR) [8, 14, 10, 12]. Synchronization reduction (or opportunistic synchronization [10]) exploits the observation that some parallel programs are over-synchronized in that it is possible to increase their exploitable parallelism (and therefore performance) by reducing the number of synchronization operations in the program. While synchronization reduction may introduce data races into a previously race-free program, the effects of these data races on the quality of the program's output can be small and controlled if the transformation is applied in a directed and controlled manner.

Reasoning about Synchronization Reduction. In this paper we present a simple language for imperative concurrent programs and demonstrate how we can capture a semantics for synchronization reduction. We also present the basic definitions of relative safety for programs written in this language and illustrate how relative safety enables a developer to transfer properties about his or her original program to the relaxed version produced by synchronization reduction. We also discuss a program logic that enables developers to establish the relative safety of a relaxed program by describing and verifying relations between it and the original program.

2. EXPRESSING SR

Figure 1 presents a simple imperative language for concurrent programs. The language contains integer variables, arithmetic expressions, boolean expressions, conditional statements, while loops, and sequential composition. The language also includes `assume` e statements, which allow the programmer to note e as an assumption about the state of

$$
\begin{aligned}
E &::= \texttt{n} \mid \texttt{x} \mid E \;\texttt{iop}\; E \\
B &::= \texttt{true} \mid \texttt{false} \mid E \;\texttt{cmp}\; E \mid B \;\texttt{lop}\; B \mid \neg B \\
S &::= \texttt{skip} \mid \texttt{x = } E \mid S \;\texttt{;}\; S \mid S \;\|\; S \\
&\quad \mid \texttt{if}\;(B)\;\{S\}\;\texttt{else}\;\{S\} \mid \texttt{while}\;(B)\;\{S\} \\
&\quad \mid \texttt{assume}\; B \mid \texttt{assert}\; B \\
&\quad \mid \texttt{acquire l} \mid \texttt{release l}
\end{aligned}
$$

Figure 1: Base Language Syntax

the program at the point at which the statement occurs, and `assert` e statements, which allow the programmer to assert e as a property about the state of the program at the point at which the statement appears. The language also includes statements to enable and coordinate concurrent executions through the parallel composition statement $s_1 \;\|\; s_2$, which executes s_1 and s_2 in parallel, and the mutual exclusion statements `acquire` l and `release` l, which acquire and release named locks.

Semantics. The semantics for this language follow that of traditional presentations of imperative languages with parallel composition [11]. We define and denote the small-step operational semantics of the language by the evaluation relation $\langle s,\ \sigma,\ \phi\rangle \to \langle s',\ \sigma',\ \phi'\rangle$. This notation means that from a *configuration* consisting of a statement s, a *memory state* σ (a finite map from program variables to integer values), and a *lock state* ϕ (a finite map from lock names to boolean values indicating if the lock has been acquired), evaluation takes one step, yielding the configuration $\langle s',\ \sigma',\ \phi'\rangle$. So for example, if s is an assignment statement x = 2, then x is assigned the value 2 in the resulting memory state. Alternatively, if s is a lock acquisition statement `acquire l`, then the resulting lock state records that lock `l` has been acquired.

To introduce the ability to apply synchronization reduction, we extend the standard semantics with the concept of a *synchronization plan*. A synchronization plan is a function $\pi : \Sigma \times \mathbb{Z} \to \mathbb{B}$, that takes as input a state of the program and the line number of an `acquire` or `release` statement and returns a boolean indicating if the lock should be acquired. We extend the standard semantics by parameterizing it with a synchronization plan π, $\langle s,\ \sigma,\ \phi\rangle \xrightarrow{\pi} \langle s',\ \sigma',\ \phi'\rangle$. In this semantics, the inference rule for the lock acquisition statement `acquire l` first queries the synchronization plan to see if it should acquire the lock. If so, then the resulting lock state records that lock `l` has been acquired. Otherwise, evaluation does not acquire the lock and the statement behaves as a skip statement.

Synchronization plans provide a simple way to describe both the semantics of a fully synchronized program and its relaxed variant in that the fully synchronized version uses the synchronization plan π_t, which returns *true* for all program states and lock statements, whereas a relaxed variant (produced by synchronization reduction) uses an alternative synchronization plan. The task of a synchronization reduction transformation is therefore the task of producing an appropriate synchronization plan.

```
/* parallel section */
...
while (K < N) {
  if (RS[K] < gCUT2) {
    assume (K < len_FF);
    assert (K < len_FF);
    FF[K] = EXP(RS[K]); }
  K = K + 1;
}
```

Figure 2: Code snippet from Water

3. REASONING ABOUT SR

Hoare Logic is the standard way to describe and verify behavioral properties of sequential programs [4]. The Hoare Logic judgment $\vdash \{P\}\ s\ \{Q\}$ states that if the predicate P is true of the program state before the evaluation of s and the evaluation of s terminates, then Q is also true of the resulting state. A developer can therefore use Hoare Logic to verify the judgment $\vdash \{x = 1\}\ x = x + 1\ \{x = 2\}$.

The goal of our work is to enable developers to be able to provide a Hoare Logic proof that describes key safety properties of the original program and an additional small *relational* proof that, together, establish these key properties for the relaxed variant of the program. In our previous work on sequential relaxed approximate programs [2], we demonstrate that its possible to do just this for sequential programs by using a relational variant of the Hoare Logic.

Example. Figure 2 presents a code snippet from Water, a water molecule simulation application [1] for which researchers have previously explored synchronization reduction [8, 10]. In [8], the suggested synchronization plan eliminates lock operations that make updates to an array `RS` execute atomically. The resulting race conditions produce a parallel computation whose result may vary nondeterministically (because of CPU scheduling variations). The loop shown in the figure is a sequential loop that executes after the parallel portion of the computation; this loop compares `RS[K]` to a cutoff variable `gCUT2` and, if it is less than the cutoff, uses `RS[K]` to update an array `FF` (here `EXP(RS[K])` is an expression involving `RS`):

A key safety property that a developer may like to establish for this computation is that `K` is within the bounds of the array `FF`, which is stored in the variable `len_FF`.[1] The developer therefore specifies this property with the assertion statement `assert (K < len_FF)`. In this example, the developer establishes the validity of the `assert` statement by placing an assumption statement (`assume (K < len_FF)`) immediately prior to the assertion. As in most verification systems and unlike an `assert` statement, the `asssume` statement does not generate a obligation to verify that its condition is formally true of the program. Such statements therefore enable developers to establish properties about their program (such as the assert statement in the figure) through alternative means, such as testing, code review, or third-party program analysis tools.

[1]We note that `K` must also be within the bounds of `RS`.

Verification. Note again that the values in RS in the relaxed variant of the program may differ from that in the original program because synchronization reduction alters the semantics of the program by making updates to RS execute non-atomically. More importantly, this semantic difference may invalidate the developer's assumptions because the assumption is predicated by (and possibly dependent on) the condition `RS[K] < gCUT2`, which may now have a different value. Therefore, if a developer would like to re-establish the assertion for the relaxed program, he or she has two choices: 1) verify the assumption outright or 2) verify that the semantic changes do not *interfere* with the assumption. In the former case, the developer would need to repeat the reasoning process that they performed to establish their assumption about the original program, duplicating the effort that he or she has already done. Moreover, if the reasoning process for this assumption is informal, the developer may arrive at a different or incorrect conclusion that leads the developer to believe that synchronization reduction has introduced an error. In the latter case, the developer can provide a relational proof. Specifically, if the developer can prove that K and len_FF are the same in both the original and relaxed program then it is straightforward to establish relative safety: if `K < len_FF` in the original program, then `K < len_FF` in also the relaxed program.

4. RELATIVE SAFETY

Our example illustrates the power of relative safety, which relies on relational proofs to show that if some property is true of the original program then it is also true of the relaxed program. In the context of approximate program transformations, relative safety is an important addition to standard notions of safety because if the developer can establish an appropriate relation between the semantics of the original and relaxed variants of the program, then he or she can transfer proofs (both formal and informal) about the original program over to reason about the relaxed.

To make this more precise, we can formalize relative safety as follows:

Definition 4.1. *Relative Safety*

If $\langle s,\ \sigma,\ \phi\rangle \overset{\pi}{\to}^{*} \langle s',\ \sigma',\ \phi'\rangle$

and $assert(\langle s',\ \sigma',\ \phi'\rangle)$ ***and*** $blocked(\langle s',\ \sigma',\ \phi'\rangle)$

then exists s'', σ'', ϕ'' *such that* $\langle s,\ \sigma,\ \phi\rangle \overset{\pi_t}{\to}^{*} \langle s'',\ \sigma'',\ \phi''\rangle$

and $assert(\langle s'',\ \sigma'',\ \phi''\rangle)$ ***and*** $blocked(\langle s'',\ \sigma'',\ \phi''\rangle)$

In this definition the notation $\langle s,\ \sigma,\ \phi\rangle \overset{\pi}{\to}^{*} \langle s',\ \sigma',\ \phi'\rangle$ denotes the reflexive transitive closure of the small-step evaluation relation — i.e., evaluation from the configuration $\langle s,\ \sigma,\ \phi\rangle$ yields $\langle s',\ \sigma',\ \phi'\rangle$ in zero or more steps. The relation $assert(c) \subseteq S\times\Sigma\times\Phi$ defines configurations for which an `assert` e statement is the foremost statement of one of the threads of control. And the relation $blocked(c) \subseteq S \times \Sigma \times \Phi$ defines configurations for which it is not possible to take an additional step according to the evaluation relation.

In words, this definition states that if a relaxed variant of the program reaches an assertion statement in the program and evaluation becomes blocked (i.e., the assertion is false), then there exists an execution in the original program that reaches an assertion and also becomes blocked. The contrapositive of this statement is more difficult to express, but states the relative safety definition that we have described thus far: if the original program satisfies all assertions in the program, then the relaxed program also satisfies all assertions in the program.

In our previous work on verifying sequential, relaxed approximate programs [2] we demonstrated that its possible to define a program logic that enables developers to express relations between their original and relaxed programs such that the resulting relaxed programs exhibit the relative safety definition described here. This logic adapts the traditional Hoare Logic to include relational assertions between the original and relaxed program. So for example a developer can specify and prove a Hoare-style judgment such as

$$\vdash \{\mathtt{x}\langle\mathtt{o}\rangle = \mathtt{x}\langle\mathtt{r}\rangle\}\ \mathtt{x} = \mathtt{x} + 1\ \{\mathtt{x}\langle\mathtt{o}\rangle = \mathtt{x}\langle\mathtt{r}\rangle\}$$

This judgment states that if the value of x in both the original($\mathtt{x}\langle\mathtt{o}\rangle$) and relaxed ($\mathtt{x}\langle\mathtt{r}\rangle$) program are the same, then after executing the assignment statement $\mathtt{x} = \mathtt{x} + 1$, x is still the same in both the original and relaxed variant of the program. The design of this judgment captures both the original semantics of the program and the relaxed semantics of the program and, therefore, states that the approximate program transformation that produced the relaxed program did not *interfere* with the value of x. Non-interference properties such as this are the primary way that a developer can transfer properties from the original program. For example, consider any property that is solely a function of x (e.g., $\mathtt{x} > 0$). If a developer can establish that approximate program transformation does not interfere with x, then that property is still valid in the relaxed program.

Challenges. While we as a community have made great strides in verifying sequential programs, our progress with concurrent programs has been hampered by the inherent complexity of concurrency. The primary challenge that we face going forward with this work is developing an expressive (yet comprehensible) framework for *relationally* reasoning about concurrent programs, which requires reasoning about concurrency in two semantically different (though syntactically similar) programs.

5. CONCLUSION

Approximate program transformations present a new optimization tool that stands to provide developers with performance gains that are out of the realm of traditional semantics-preserving program optimizations. However, a key component to unlocking the power of approximate transformations is finding some technique to ensure that the resulting program is safe. Our position is that relational reasoning is the key to providing developers with a scalable reasoning mechanism wherein they provide small proofs relating the original and relaxed program that therefore enable them to transfer much of the reasoning they have already done for the original program over to verify the relaxed.

6. REFERENCES

[1] W. Blume and R. Eigenmann. Performance analysis of parallelizing compilers on the Perfect Benchmarks programs. *Transactions on Parallel and Distributed Systems*, 3(6), 1992.

[2] M. Carbin, D. Kim, S. Misailovic, and M. Rinard. Proving acceptability properties of relaxed nondeterministic approximate programs. PLDI, 2012.

[3] S. Chaudhuri, S. Gulwani, R. Lublinerman, and S. Navidpour. Proving Programs Robust. FSE, 2011.

[4] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10), October 1969.

[5] H. Hoffman, S. Sidiroglou, M. Carbin, S. Misailovic, A. Agarwal, and M. Rinard. Dynamic knobs for responsive power-aware computing. ASPLOS, 2011.

[6] H. Hoffmann, S. Misailovic, S. Sidiroglou, A. Agarwal, and M. Rinard. Using Code Perforation to Improve Performance, Reduce Energy Consumption, and Respond to Failures . Technical Report MIT-CSAIL-TR-2009-042, MIT, 2009.

[7] S. Liu, K. Pattabiraman, T. Moscibroda, and B. Zorn. Flikker: Saving dram refresh-power through critical data partitioning. ASPLOS, 2011.

[8] S. Misailovic, D. Kim, and M. Rinard. Parallelizing sequential programs with statistical accuracy tests. Technical Report MIT-CSAIL-TR-2010-038, MIT, 2010.

[9] S. Misailovic, S. Sidiroglou, H. Hoffmann, and M. Rinard. Quality of service profiling. ICSE, 2010.

[10] S. Misailovic, S. Sidiroglou, and M. Rinard. Dancing with uncertainty. In Submission to RACES, 2012.

[11] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6, 1976.

[12] M. Rinard. A lossy, synchronization-free, race-full, but still acceptably accurate parallel space-subdivision tree construction algorithm. Technical Report MIT-CSAIL-TR-2012-005, MIT, 2012.

[13] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman. Enerj: approximate data types for safe and general low-power computation. PLDI, 2011.

[14] A. Udupa, K. Rajan, and W. Thies. Alter: exploiting breakable dependences for parallelization. PLDI, 2011.