Verified Integrity Properties for Safe Approximate Program Transformations

Michael Carbin Deokhwan Kim Sasa Misailovic Martin C. Rinard
MIT CSAIL
{mcarbin, dkim, misailo, rinard}@csail.mit.edu

Abstract

Approximate computations (for example, video, audio, and image processing, machine learning, and many scientific computations) have the freedom to generate a range of acceptable results. Approximate program transformations (for example, task skipping and loop perforation) exploit this freedom to produce computations that can execute at a variety of points in an underlying accuracy versus performance trade-off space. One potential concern is that these transformations may change the semantics of the program and therefore cause the program to crash, perform an illegal operation, or otherwise violate its integrity.

We investigate how verifying *integrity properties* – key correctness properties that the transformed computation must respect – can enable the safe application of approximate program transformations. We present experimental results from a compiler that verifies integrity properties of perforated loops to enable the safe application of loop perforation.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; D.3.4 [*Programming Languages*]: Processors—Optimization; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Program Analysis

Keywords Approximate Computing, Integrity Properties, Loop Perforation, Relaxed Programs

1. Introduction

In recent years researchers have developed a range of approximate program transformations (e.g, task skipping [14, 15], loop perforation [11, 12, 17], function substitution [3, 4, 9, 18], approximate memoization [7], approximate memory regions [16], and racy-parallelization [10]). These transformations can significantly reduce the computational resources (e.g., time or energy) required to produce an acceptable result. But because they may change the semantics of the computation, they may, if inappropriately applied, cause the computation to crash, perform an out of bounds access, or otherwise violate its integrity. We therefore investigate an approach in which the program transformation system verifies that a potential transformation will not violate the integrity of the computation.

PEPM'13, January 21–22, 2013, Rome, Italy.
Copyright © 2013 ACM 978-1-4503-1842-6/13/01. . . $15.00

We define the integrity of a computation as its set of *integrity properties*, which we group into two categories:

- **Internal Integrity Properties:** Internal integrity properties characterize internal aspects of a transformed computation. The goal is to ensure that the transformed computation itself does not crash, perform an illegal operation such as an out of bounds array access or null pointer dereference, or violate the precondition of an invoked component.
- **External Integrity Properties:** A transformed computation typically produces a result that its client then uses. External integrity properties state constraints that the result must satisfy to preserve the integrity of the computation's client.

In this paper we present our experience building a compilation system that reasons about the effects of a single approximate transformation, loop perforation [8], on a single universal internal integrity property, pointer safety, and several external integrity properties describing the sign and/or range (upper and/or lower bounds) of the result that the transformed computation must produce for its client. Our compilation system statically verifies these integrity properties to enable the safe application of loop perforation.

Loop Perforation Strategies. Loop perforation increases performance by skipping iterations of time-consuming loops. In previous work, we have explored a variety of perforation transformations [11, 12, 17] — truncation perforation (which executes a prefix of the original loop iterations), interleaving perforation (which deterministically skips iterations at a given rate), and random perforation (which chooses a random subset of the iterations to execute). Our compilation system implements internal and external integrity verifiers that together determine how to use these transformations to produce one of the following perforation strategies:

- **Full:** If both the internal and external integrity verifiers succeed, then executing any subset of the original loop iterations preserves the full integrity of the computation. The system can therefore apply any perforation transformation.
- **Truncate:** If only the external integrity verifier succeeds, then the system can only apply truncation perforation. Truncation perforation naturally preserves internal integrity because all of the iterations in the perforated loop also execute in the original loop. Therefore, if the original loop satisfies its internal integrity properties, then so does the perforated loop.
- **Adapt:** The external integrity verifier succeeds, but only if it assumes that the external integrity property holds upon loop entry. In other words, the system can prove that an external integrity property is loop invariant provided that it holds initially, but is unable to prove that the property is established for all potential invocations of the loop.

In this case the system can apply *adaptive perforation* — the transformed loop initially executes without perforation. At each iteration, the loop dynamically checks if the external integrity property holds. Once the property holds, it can then perforate the loop using either full or truncation perforation (depending on the result of the internal integrity verifier).

This selection of perforation strategies maximizes the ability of the verification and transformation systems to work together to safely exploit a range of perforation opportunities.

Relative Internal Integrity. A key aspect of our analysis approach is the use of *relative internal integrity* to simplify the verification of internal integrity. Instead of attempting to verify that all executions of the perforated loop are safe, we verify that if the perforated loop violates an internal integrity property, then the original loop also violates that property [6]. Following this reasoning, if an integrity property holds in the original loop then it must also hold in the perforated loop.

Verifying relative internal integrity significantly simplifies the analysis by enabling the modular verification of computations that use pointers and offsets that are initialized outside the analyzed computation. It also makes it possible to successfully verify complex pointer or array index expressions whose absolute pointer safety may be difficult or impossible to verify statically.

Case Studies. We applied our compilation system to 25 loops in 7 applications drawn from the PARSEC Benchmark Suite [5], which contains emerging computational workloads such as machine learning, machine vision, financial applications, and video processing. In addition to pointer safety (i.e., internal integrity), we also identified and specified external integrity properties that constrain the sign and/or range of each loop's outputs and ensure that each loop's client can successfully use the result.

Our compilation system was able to find at least one safe perforation strategy for 16 of the 25 total loops. For 7 of the verified loops, the system's relative internal integrity analysis identified that it was possible to apply full perforation. For 6 of the verified loops, the system identified that truncation perforation is safe only if it is applied adaptively. For the remaining 3 loops the system verified the safety of truncation perforation.

Contributions. This paper makes the following contributions:

- **Conceptual Framework.** We define the integrity of a computation by its set of internal and external integrity properties. An approximate program transformation is therefore safe to apply to a computation if it preserves these properties.
- **Compilation System.** We present a compilation system that enables safe perforation. This system includes 1) a specification language that developers can use to identify a perforatable loop and specify its external integrity properties and 2) analyses to verify the integrity of a perforated loop.
- **Integrity Property Verification.** We present the analyses we have implemented to verify the integrity of a perforated loop. Our system uses relative internal integrity analysis and an external integrity analysis that verifies simple properties such as the non-negativity or positivity of a loop's results. Remarkably, these simple properties are enough to ensure the integrity of the majority of the loops in our case studies.
- **Experimental Results.** We present the results of applying our compilation system to 25 loops in 7 applications drawn from the PARSEC Benchmark Suite. In general, we find that loop perforation typically works well when loops are largely decoupled from their clients (as reflected in the simplicity of the external integrity properties). This decoupling gives approximate transformations substantial latitude in changing the result that the loop computes while preserving the integrity of the client.

```
/* perforate
     input nonnegative result;
     output nonnegative result;
 */
for (int i = 0; i < dim; i++) {
  float temp = p1.coord[i] - p2.coord[i];
  result += temp * temp;
}
```

Figure 1. Perforatable Distance Computation in Streamcluster

Approximate transformations can deliver significant performance, energy reduction, and adaptation benefits. Establishing the integrity of transformed computations can eliminate a key concern that would otherwise inhibit the widespread adoption of approximate transformations and deny users the significant benefits that they can deliver.

2. Example

Figure 1 presents a frequently-executed loop from the Streamcluster application in the PARSEC Benchmark Suite [5]. Streamcluster implements an online clustering algorithm that takes as input a stream of data points and computes the set of clusters observed in the stream thus far. This loop computes the distance between two points `p1` and `p2`. Each point has a field `coord` that corresponds to the vector of features of the data point. This particular implementation of the loop computes the squared Euclidean distance between two `dim`-dimensional vectors.

Internal Integrity. Pointer safety (i.e., that accesses to `p1.coord` and `p2.coord` remain within bounds) is the only property required to ensure the internal integrity of this loop. We use a novel relative pointer safety analysis (see Section 3) to verify that loop perforation preserves this property. Conceptually, the analysis verifies that the set of memory locations that the perforated loop accesses is a subset of the locations that the original loop accesses and, therefore, perforation does not introduce any new pointer safety violations. Because this relative approach enables the analysis to, in effect, leverage the assumed validity of references in the original loop, we have been able to develop an effective local analysis that analyzes only the transformed computation. Standard absolute pointer safety analyses, in contrast, must typically use complex global analyses to verify the validity of pointer expressions and reason about the size of accessed arrays.

External Integrity. The client is coded to work with any plausible (i.e., non-negative) distance as it clusters the points. The essential external integrity property is therefore that the computed result must be greater than or equal to zero.

Our compilation system provides a specification language that a developer uses to identify perforation targets, specify the preconditions that inputs to the perforated loop satisfy, and specify the external integrity properties that the loop's outputs must satisfy (and therefore the compilation system must verify). The `perforate` annotation in Figure 1 illustrates how a developer writes annotations in the specification language. The annotation itself indicates to the compilation system that the loop that follows should be considered as a potential target for perforation. The `input nonnegative result` annotation indicates that `result` is non-negative on entry into the loop. The annotation `output nonnegative result` indicates that the external integrity constraint is that the loop modifies only the `result` variable and that `result` must be non-negative upon loop exit.

Our external integrity verifier consists of several program analyses, including a custom sign analysis, a custom effect analysis, and an off-the-shelf verification condition generator that uses an

Application	Function	External Integrity Properties	Perforation Strategy
streamcluster	`dist`	non-negativity	full
x264	`x264_me_search_ref`	non-negativity, range checks	truncate
	`pixel_satd_wxh` (2)	non-negativity	full
	`x264_pixel_sad_wxh` (2)	non-negativity	full
bodytrack	`IntersectingCylinders` (2)	—	truncate
	`InsideError` (2)	positivity	truncate (adapt)
	`ImageErrorInside` (2)	positivity	truncate (adapt)
	`ImageErrorEdge` (2)	positivity	truncate (adapt)
canneal	`swap_cost` (2)	non-negativity	full

Table 1. Perforation Results

SMT solver to discharge the generated verification conditions. For this loop, the verifier uses the sign analysis to verify that if `result` is non-negative upon loop entry, then it is non-negative upon loop exit. It then uses the effect analysis to verify that the loop modifies only `result`.

3. Analysis Algorithms

Our approach to verifying internal integrity uses the concept of relative integrity: if the original program executes without error, then the transformed program executes without error. To make this concept more precise, let us consider a programming language that includes a labeled `assert` statement $\texttt{assert}_\ell\ e$. The relation $\rightarrow$ denotes the programming language's small-step execution semantics, which relates configurations of statements and program states $\langle s,\ \sigma\rangle$ to configurations $\langle s',\ \sigma'\rangle$ that result after one step of execution ($\rightarrow^*$ denotes the reflexive transitive closure of this relation). $T[s]$ denotes the transformed statement that results from applying an approximate program transformation T to a statement s. We define relative integrity for an arbitrary property e of the program state (as specified by an assertion statement $\texttt{assert}_\ell\ e$) as follows:

Definition 1 (Relative Internal Integrity)**.**

$$\begin{array}{rl} \textbf{\textit{If}} & \langle T[s],\ \sigma\rangle \rightarrow^* \langle \texttt{assert}_\ell\ e\ ;\ \cdot,\ \sigma'_t\rangle \\ \textbf{\textit{then}} & \exists \sigma'_o \cdot \langle s,\ \sigma\rangle \rightarrow^* \langle \texttt{assert}_\ell\ e\ ;\ \cdot,\ \sigma'_o\rangle \wedge \\ & \forall x \in \mathit{free}(e) \cdot \sigma'_t(x) = \sigma'_o(x) \end{array}$$

Relative integrity is a simulation relation between a transformed program and its original semantics that makes it possible to transfer a developer's reasoning about the original program to the transformed program. Specifically, if the transformed program executes an `assert` statement, then there must exist an execution in the original program that executes the same `assert` statement with the same value for the condition. Therefore, if the `assert` statement is valid in the original program (i.e., its condition always evaluates to true), then it must also be valid in the transformed program.

Our compilation system adapts this fully general relative integrity definition to verify pointer safety by checking the following properties of each pointer dereference in a loop:

Guaranteed Execution. If a perforated loop dereferences a pointer, then there must exist an execution of the dereference in the original loop. Our analysis verifies this by checking that there is no control flow that allows the loop to skip the dereference or exit early.

History Independence. If a perforated loop dereferences a pointer, then the value of that pointer must be *history independent* in that its value can be computed solely as a function of the current value of the loop's iteration variable and the state of the program before execution of the loop. A history independent pointer value has the same value in both the original and perforated versions of the loop at any given iteration because its value is independent of the computations performed in previous, potentially skipped iterations. Our system uses induction variable analysis and loop invariance analysis to verify that a pointer is history independent.

These two properties work together to establish relative integrity. Guaranteed execution ensures that the original loop executes each pointer dereference. History independence then ensures that the pointer value is the same as in the original loop.

In contrast to our internal integrity analysis, our external integrity analyses are fairly standard: our sign analysis is a forward dataflow analysis; for more complex constraints involving variable ranges we integrate with the Frama-C verification platform [1] through a source-to-source translation from our specification language to the annotations that Frama-C supports.

Implementation. Our compilation system supports programs written in C. We have implemented our system on top of the pycparser C AST generator [2] and the C Intermediate Language (CIL) program analysis infrastructure [13].

4. Case Studies

We next present a number of case studies in which we analyze the integrity of 25 perforated loops in 7 applications using our system.

We selected applications from the PARSEC Benchmark Suite [5] that have been explored in previous work on loop perforation [8, 11, 17]. Specifically, we selected the loops by using *quality-of-service profiling*, which speculatively applies loop perforation to the most time-consuming loops in an application and suggests as candidates for loop perforation those loops that effectively trade accuracy of their results for performance [17].

Experimental Results. Our system identified successful perforation strategies for 16 of the 25 analyzed loops. These 16 loops range in size between 4 and 131 lines of code. Perforating each individual loop produces full application speedups of between 7% and 42% while typically changing the output of the application by less than 10% (see previous publications [8, 11, 17] for more detailed performance and accuracy numbers).

Table 1 presents a summary of the perforation results for the 16 loops. Column 1 presents the application. Column 2 presents the function in which each perforatable loop resides. It also presents the number of perforatable loops in each function (if it is greater than one). Column 3 presents each loop's external integrity properties. We found that many of the loops in our benchmark suite shared a common set of remarkably simple external integrity properties, specifically non-negativity (applicable to 14 loops) and positivity (applicable to 6 loops). One loop (`x264_me_search_ref`) has a more complex integrity property involving range checks.

Column 4 presents which perforation strategies the compilation system verified can be applied and still preserve the internal and external integrity properties of the loop. If an entry contains the word "full", then the system verified that all perforation strategies preserve both the internal integrity of the loop (i.e, do not violate pointer safety) and its listed external integrity properties.

If an entry contains the word "truncate" this indicates that the compilation system was only able to verify that truncation perforation preserves both the internal and external integrity properties of the loop — i.e., other perforation strategies do not satisfy the relative integrity property. For example, fully perforating the loops in `IntersectingCylinders` does not satisfy the relative integrity property because the loops' control flow structures allow them to terminate before executing all potential iterations. Therefore, if perforated versions of these loops skip an iteration on which the original loops terminate, then the perforated versions can continue on to access an invalid pointer whereas the original loops would not.

If a perforation strategy contains an additional "adapt" in parentheses (i.e., six of the `bodytrack` loops), then the system verified that adaptively applying the strategy preserves the loop's external integrity whereas normal perforation may not. The `bodytrack` loops each have a positivity constraint on a variable that counts the number of samples that have been taken as they iterate over a set of objects. The loops increment the number of samples when inspecting an object only if an object satisfies a given property; this counter variable is then later used as the divisor in a division operation and therefore must be non-zero for the program to avoid a divide-by-zero error. Because the counter begins at zero and the object property is non-trivial, it is not straightforward to verify that perforation preserves the positivity constraint. However, adaptive perforation makes it possible to reason about these loops — our compilation system verifies that if we enable perforation only after the counter becomes positive, then the counter remains positive.

Remaining Loops. Verifying the integrity of the nine remaining loops is beyond the reach of our current system, typically because of complex external integrity properties that involve updates to data structures such as vectors and arrays. Our current system only supports external integrity properties that involve scalar variables.

This situation motivates the development of a more sophisticated reasoning system — in particular, we envision a system that can handle more complex data structures and use relative reasoning to verify complex external integrity properties. But it also highlights the remarkable simplicity of the reasoning required to safely perforate many loops — our current system, which focuses on a relatively simple set of integrity properties, is able to safely perforate 16 of the 25 perforatable loops in PARSEC (and verify the integrity of fully perforating 7 of these 16 loops).

5. Related Work

Our previous work on Relaxed Programs [6] provides a general programming methodology and verification framework for statically reasoning about transformations that change the semantics of a program. The framework allows a developer to specify arbitrary integrity properties along with more complex properties that relate the semantics of a relaxed program to its original semantics. The framework provides a program logic and Coq formalization with which a developer can manually prove the desired properties for programs written in a small imperative language. The work we present in this paper is complementary in that it provides a checker with which developers can automatically verify a set of target integrity properties for general C programs.

EnerJ's type system allows developers to separate approximate from non-approximate computation via a non-interference guarantee [16]. Our work differs in that our compiler can verify semantic integrity properties in cases where non-interference does not hold.

In addition to their essential integrity properties, approximately transformed programs also have essential accuracy properties, which characterize the acceptable differences between the outputs of the original program and the transformed. An accuracy property may for example state that a developer is willing to accept any output that differs by at most 10% from the original output according a specified metric. In previous work we have investigated how to specify, verify, and select transformations that satisfy accuracy properties using empirical methods (e.g., testing and profiling [8, 9, 11, 17], synthesis [18], and formal verification [6, 12]). These techniques are complementary to the approach we present in this paper in that approximately transformed computations must satisfy both their accuracy and integrity properties.

6. Conclusion

Approximate program transformations promise to deliver significant performance and resource consumption improvements at the cost of small accuracy losses. Most previous research in this area, however, has relied on purely empirical testing approaches to validate the integrity of the transformed approximate computations. Our results provide encouraging evidence that compilers can verify integrity properties to safely apply approximate transformations, specifically by verifying that the transformed computations respect the specified integrity properties.

Acknowledgments. This research was supported in part by the National Science Foundation (Grants CCF-0811397, CCF-0905244, CCF-1036241, and IIS-0835652), DARPA (Grants FA8650-11-C-7192 and FA8750-12-2-0110), and the United States Department of Energy (Grant DE-SC0005288).

References

[1] Frama-C. `http://frama-c.com/`.

[2] pycparser. `http://code.google.com/p/pycparser/`.

[3] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe. PetaBricks: A language and compiler for algorithmic choice. PLDI, 2009.

[4] W. Baek and T. M. Chilimbi. Green: A framework for supporting energy-conscious programming using controlled approximation. PLDI, 2010.

[5] C. Bienia, S. Kumar, J. Pal Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. PACT, 2008.

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

[7] S. Chaudhuri, S. Gulwani, R. Lublinerman, and S. Navidpour. Proving programs robust. FSE, 2011.

[8] 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.

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

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

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

[12] S. Misailovic, D. Roy, and M. Rinard. Probabilistically accurate program transformations. SAS, 2011.

[13] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. CC, 2002.

[14] M. Rinard. Probabilistic accuracy bounds for fault-tolerant computations that discard tasks. ICS, 2006.

[15] M. Rinard. Using early phase termination to eliminate load imbalances at barrier synchronization points. OOPSLA, 2007.

[16] 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.

[17] S. Sidiroglou, S. Misailovic, H. Hoffmann, and M. Rinard. Managing performance vs. accuracy trade-offs with loop perforation. FSE, 2011.

[18] Z. Zhu, S. Misailovic, J. Kelner, and M. Rinard. Randomized accuracy-aware program transformations for efficient approximate computations. POPL, 2012.