ApproxSymate: Path Sensitive Program Approximation using Symbolic Execution

Himeshi De Silva
School of Computing, National University of Singapore
Singapore
himeshi@comp.nus.edu.sg

Andrew E. Santosa
School of Computing, National University of Singapore
Singapore
santosa_1999@yahoo.com

Nhut-Minh Ho
School of Computing, National University of Singapore
Singapore
minhhn@comp.nus.edu.sg

Weng-Fai Wong
School of Computing, National University of Singapore
Singapore
wongwf@comp.nus.edu.sg

Abstract

Approximate computing, a technique that forgoes quantifiable output accuracy in favor of performance gains, is useful for improving the energy efficiency of error-resilient software, especially in the embedded setting. The identification of program components that can tolerate error plays a crucial role in balancing the energy vs. accuracy trade off in approximate computing. Manual analysis for approximability is not scalable and therefore automated tools which employ static or dynamic analysis have been proposed. However, static techniques are often coarse in their approximations while dynamic efforts incur high overhead. In this work we present *ApproxSymate*, a framework for automatically identifying program approximations using symbolic execution. ApproxSymate first statically computes symbolic error expressions for program components and then uses a dynamic sensitivity analysis to compute their approximability. A unique feature of this tool is that it explores the previously not considered dimension of *program path* for approximation which enables safer transformations. Our evaluation shows that ApproxSymate averages about 96% accuracy in identifying the same approximations found in manually annotated benchmarks, outperforming existing automated techniques.

CCS Concepts • **Computing methodologies** → **Hybrid symbolic-numeric methods**; • **Mathematics of computing** → *Approximation*; • **Computer systems organization** → *Embedded software*.

LCTES '19, June 23, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6724-0/19/06…$15.00
https://doi.org/10.1145/3316482.3326341

Keywords Approximate Computing, Symbolic Execution

ACM Reference Format:
Himeshi De Silva, Andrew E. Santosa, Nhut-Minh Ho, and Weng-Fai Wong. 2019. ApproxSymate: Path Sensitive Program Approximation using Symbolic Execution. In *Proceedings of the 20th ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '19), June 23, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3316482.3326341

1 Introduction

Energy efficiency of embedded software is a critical factor determining the successful operation of systems in power constrained environments. In this context, the paradigm of approximate computing, in which an application's output accuracy is carefully traded off in favor of energy savings, has become attractive for low power execution. Recognizing this phenomenon, hardware vendors are starting to provide more support for software approximations on their platforms. For example, Nvidia's latest GPGPUs now support half-precision floating point arithmetic natively [3]. However, to make use of these advances, program components that are suitable for approximation have to be identified, as not all of them are equally sensitive to error. One method of identification is to allow programmers to manually demarcate code that can tolerate error or be run on error-resilient hardware [8, 31, 32]. However placing such annotations require expert domain knowledge, and maintaining such annotations across versions, upgrades and code rewrites is costly. This has given rise to tools that enable automatic discovery of program approximability.

Precision scaling, loop perforation, skipping tasks and memory accesses have been shown to speed up programs through a graceful degradation of quality [15, 30, 33]. Search techniques are often used to discover code patterns or data for which these can be applied [10]. However, they can be time consuming and often rely on heuristics to produce reasonable results [1, 11]. Program analysis to examine the reach of input error and significance analysis to identify important

program components have also been studied [29, 34]. It is worthwhile to also mention that with these techniques, safe execution of the approximated program has also become a concern and methods to provide safety guarantees have been developed [6, 7, 26]. Recently, interest has swayed from applying singular transformations to more specialized ones such as phase-aware approximations [21]. Nonetheless, approximation based on different possible execution paths has not yet been explored. Given the nature of approximable programs to apply various sets of operations to the individual units of an image or audio file based on their initial input value, looking into *path based approximation* opens up new dimensions for energy savings. Moreover transformations that lead to drastic deviations from original paths, often result in unexpected increases in the output's error and therefore must be carefully investigated.

1.1 A Motivating Example

To demonstrate the impact of program paths on approximability, we conducted experiments in 2 major program paths in the `susan` edge detection kernel from MiBench [14] which share the same set of variables. The number of times each path is executed to process the sample image are 39,467 and 29,625 respectively. The program path taken is determined dynamically at run time by the input data. We injected a 50% relative error to all the variables in one of the two paths while keeping the variables error-free in the other. Figure 1 shows the differences compared to the reference output. From the figure, we can see that the same set of variables when executing in path 1 will have less impact on output error, and are therefore significantly approximable (Figure 1-c). However, in program path 2 approximating them will result in incorrect output for edge detection (Figure 1-d). Any path agnostic technique will either be too conservative (marking both paths as not approximable) or too aggressive (marking both paths as approximable) in this case.

In this work we explore path based program approximations using symbolic execution to automatically discover data and algorithmic program transformations. Our framework, ApproxSymate, extends the symbolic virtual machine KLEE [4] to produce symbolic expressions for any variable's error in terms of the input and possible error introduced into the input. It does this with a novel shadow symbolic representation that represents the relation between these quantities. Path conditions are also maintained by the shadow symbolic expressions allowing us to examine the effect of error on different program paths. The symbolic expressions generated are fed into a sensitivity analysis that identifies the approximability of these components. Our main contributions are:

- ApproxSymate - A framework that determines the error sensitivity of program components by generating symbolic error characterization expressions for the components;

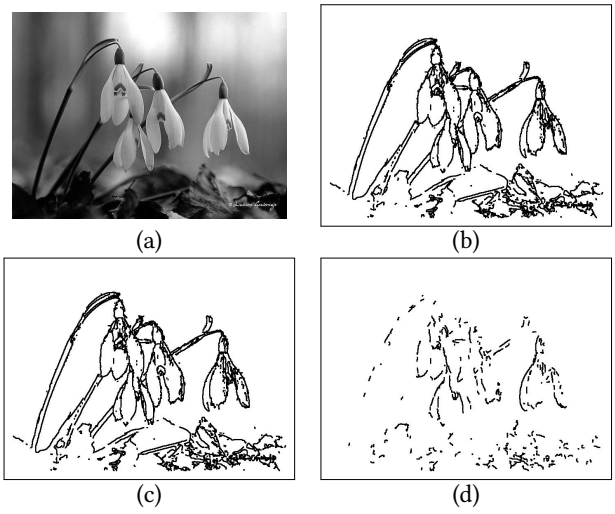

Figure 1. `susan` edge detection: (a) input taken from [23], (b) reference output, (c) approximation on path 1, (d) approximation on path 2.

- Incorporate program execution path in the approximation analysis - a factor that hitherto has not been studied.
- Demonstrate how ApproxSymate can be used for data and algorithmic approximations.

In Section 2 we present some background to our work. Section 3 presents our overall approach, focusing on the building of the symbolic formulas that represent the relationships between the program input and errors. In Section 4 we present the design of our implementation, and in Section 5 we present the results of experimenting with our implementation on a benchmark suite. In Section 6 we discuss related work, followed by a discussion in Section 7 and the conclusion in Section 8.

2 Background

2.1 Symbolic Execution

The main idea behind symbolic execution is to supply symbolic values instead of concrete data as input to a program. Program execution happens on these symbolic input values to produce symbolic expression for program variables. The symbolic execution engine maintains a program *state* along with a *path condition* which at the start of execution are empty and true. As the execution progresses, the state is updated with the symbolic expressions that are generated from the computations of the program. When a conditional statement is encountered, the execution engine invokes a constraint solver to check the satisfiability of the two possible branching conditions. If they are satisfiable, the program state is cloned to produce two states with the path condition

```
...
klee_make_symbolic(&a, sizeof(a), "a");
...
d = b * b - 4 * a * c;
w1 = 2 * a;
w2 = sqrt(fabs(d));
if(d > 0) {
       ...
       x1_r = (-b + w2) / w1;
       x1_i = 0;
       ...
} else if (d == 0) {
....
}  else {
       ...
       x1_r = -b / w1;
       x1_i = w2;
       ...
}
...
```

Figure 2. Example Code

of each updated with either the true or false branching condition. The execution continues for the pair of states and their updated path conditions with the next statement in the program until another conditional statement is reached, upon which these steps are repeated. At the end of the symbolic execution, for each path condition, the constraint solver will generate concrete values satisfying it for all the symbolic input. These concrete inputs can be useful for testing and debugging the program whenever an unexpected behavior occurs.

2.2 KLEE and LLVM

KLEE is a popular symbolic execution tool that has been widely used in software testing [2, 4]. KLEE takes as input a source, which has been annotated with the appropriate function calls to KLEE, in the LLVM bitcode format [17]. The basic instrumentation API of KLEE important to our purpose is `klee_make_symbolic` which is used to mark the input as symbolic, as shown in the example in Figure 2. On completion, KLEE will produce for each execution path, by default in a directory for that run, test cases in the form of concrete values for the symbolic values that were provided as input. Apart from this KLEE also provides some useful command line arguments such as `write-kqueries`, which prints out the path conditions in terms of the symbolic input for each execution path. We made use of such features in this work.

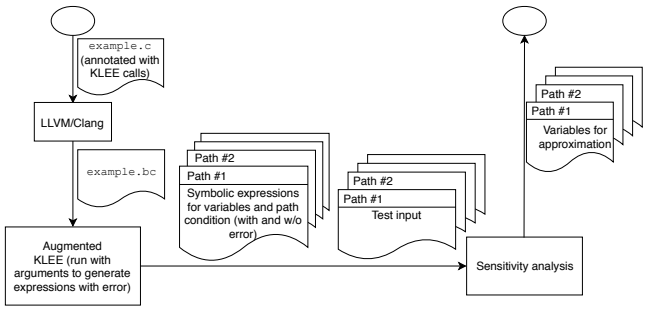

Figure 3. Overview of ApproxSymate

3 Approach

3.1 Overview

Figure 3 shows the overview of our framework. Initially, a source C program is annotated with the `klee_make_symbolic` function call to mark the input variables as 'symbolic' as well as the `klee_track_ error` call (explained later in Section 4) to indicate the input that may have uncertainty. If an input variable is marked as symbolic but not as permitting error, then it will be considered precise during the symbolic execution. When the LLVM bitcode (obtained from Clang) for a source annotated with these calls is executed with the augmented KLEE, the symbolic expression characterizing the relative error for each marked program variable as well as the path condition in terms of the symbolic input and input's relative error will be generated for all feasible program paths (in addition to the default output mentioned in Section 2.2). This is done through a *shadow state* that tracks input's error propagation. Whenever a computation occurs with the symbolic input in a state, the same computation is performed in the shadow state but with both the symbolic input as well as symbolic variables for the input's relative error. In this shadow state, the relative error of variables caused by the tainted input is accumulated and propagated according to established rules through the program. Note that assuming errors in the input is a means for symbolically analyzing the effects of input error. The actual input can be error-free.

The files generated by KLEE are then fed to a sensitivity analysis module that determines each program variable's, path's and input's sensitivity to error using the symbolic error expressions. For each program path, it will output a list of approximable and non-approximable variables as well as the path's sensitivity to error in the different inputs.

3.2 Generating Symbolic Error Expressions

For ease of our discussion, let's consider a simple programming language shown in Figure 4, a simplified form of the LLVM language.

To discuss symbolic manipulation of formulas during symbolic execution, we assume a domain *Expr*. One can understand *Expr* to be the domain of uninterpreted functions containing expressions including predicates in quoted form. We assume that logical operators ($\wedge$, $\vee$, etc.) and symbolic variables belong to *Expr* so that we can express both *atomic* and *composite constraints*. We assume two distinct sets of *symbolic* variables in *Expr*: Var_v, and Var_e, where $Var_v \subseteq Expr$ is the set of *value* variables and $Var_e \subseteq Expr$ is the set of *error* variables, which represent the relative error amount. Concretely, every program variable x has two representations: x_v, and x_e. x_v represents its ideal value when the operations executed introduces no error, and x_e represents the relative error introduced to x during the computation.

During symbolic execution, constraints are collected into the path condition whenever a conditional branch is executed and the condition cannot be determined to be either valid or unsatisfiable. More formally, a path condition is a conjunction of other constraints, where each constraint in the set is a branch condition along the execution path. Our approach augments KLEE with a shadow structure that computes extra information during symbolic execution. This shadow structure consists of two extra sorts of expressions computed during the symbolic execution and is representable by the elements of *Expr*: The *path conditions with error*, and the *error expressions*. A path condition with error is essentially a constraint on Var_v and Var_e. When a branch condition is encountered in the symbolic execution, the shadow state creates the same branch condition, but with the relative errors applied to the original values in the branch condition. These branch conditions are then collected as constraints in the path condition with error. We will revisit this towards the latter end of this section. An error expression, on the other hand, symbolically expresses the relative error amount of a value stored in a variable. The path conditions with error as well as the error expressions are the main outputs of the symbolic execution that are used as the input to the sensitivity analysis.

We define the symbolic execution semantics for the instructions in our language using structural operational semantics in Figure 5. The *big step* semantics of an executed path can be trivially built from this semantics and therefore is not shown. As is standard in the discussion of symbolic execution (see, e.g., [5]), our semantics considers the construction of path conditions. It is important to note that for simplicity, our description abstracts away the standard path conditions built by KLEE that are used to decide path satisfiability. Instead, we only show the path conditions with error and the error expressions, which are the outputs of our analysis. In Figure 5, we provide the semantics of each instruction as a state transition between *configurations* of the syntax $\langle l, \sigma, h, \pi_e \rangle$, whose components are:

1. a *label* $l \in Label$,
2. a *symbolic state* $\sigma \in \Sigma$, with $\Sigma \equiv \{v, e\} \times Register \rightarrow Expr$, and where:
   a. $\sigma_v \in Register \rightarrow Expr$, is the *value* state, where it is understood that the codomain expresses the real number values had the executed operations produced no errors,
   b. $\sigma_e \in Register \rightarrow Expr$ is an *error* state whose codomain contains expressions of the amount of error in the computation,
3. a *symbolic heap* $h \in \{v, e\} \times (\mathbb{N} \cup \{0\}) \rightarrow Expr$, where, similarly to state:
   a. $h_v \in (\mathbb{N} \cup \{0\}) \rightarrow Expr$ is the *value* heap where its codomain expresses the numeric values that have been computed using operations that produce no error,
   b. $h_e \in (\mathbb{N} \cup \{0\}) \rightarrow Expr$ is the *error* heap where its codomain expresses the amount of error in the computation,
4. a path condition with error $\pi_e \in \rightarrow Expr$. Note again that this path condition is not the path condition constructed by KLEE to decide path satisfiability, but one that expresses the constraints on the error-free values and the error amount.

It is important to note that a symbolic state, symbolic heap and path condition has *Expr* as their range, and not a value in a numerical domain. This is because our semantics describes how our symbolic expressions are manipulated during symbolic execution.

In Figure 5, the rule [ALLOC] is the semantics rule for a memory allocation. To express the memory operations done in KLEE for the rules [STORE] and [LOAD], we introduce the function *concretize* which assigns a constant to an address that may be represented as non-constant symbolic expression. This abstracts how KLEE behaves: KLEE tries to concretize a symbolic address using a constraint solver. In some cases when this fails, extra constraints are added into the path condition based on the memory region.

In Figure 5, for the rules [AP1] and [AP2], we assume a semantics function $[\![\cdot]\!]^1$ for unary operators *UnOp* where

$$[\![\cdot]\!]^1 : \{v, e\} \times UnOp \times Register \times \Sigma \rightarrow Expr.$$

and a semantics function $[\![\cdot]\!]^2$ for binary operators *BinOp* where:

$$[\![\cdot]\!]^2 : \{v, e\} \times BinOp \times Register \times Register \times \Sigma \rightarrow Expr.$$

We further define the semantics function $[\![\cdot]\!]^1$ using the semantics functions $[\![\cdot]\!]^1_v$ and $[\![\cdot]\!]^1_e$, respectively for providing the value and error expressions. The function $[\![\cdot]\!]^1_v$ follows the computation of symbolic expressions by KLEE during symbolic execution. Although this is the case, we assume a different interpretation of the expressions to KLEE, as we assume that the symbolic operators applied are mathematical operators that produce no error, whereas KLEE tries to be faithful to the actual LLVM semantics by interpreting the

Program	::=	*Block**
Block	::=	(*Label Instruction*)*
Instruction	::=	*Register* ← `alloc` *Immediate* |
		`store` *Register Register* |
		Register ← `load` *Register* |
		Register ← *BinOp Register Register* |
		Register ← *UnOp Register* |
		`br` *Label* |
		`br` *CmpInst Label Label*
CmpInst	::=	*Register CmpOp Register*
CmpOp	::=	= | < | > | ≤ | ≥

Figure 4. A simple programming language in BNF.

expressions in the bitvector domain. For $[\![\cdot]\!]^1_e$ function, for any $op \in UnOp$, we assume that $[\![op\ v]\!]^1_e\sigma = \sigma_e(v)$.

As with unary operations, we define the semantics function $[\![\cdot]\!]^2$ for binary operations using the semantics functions $[\![\cdot]\!]^2_v$ and $[\![\cdot]\!]^2_e$, respectively for the value and the error expressions. We also assume that the semantics function $[\![\cdot]\!]^2_v$ constructs the symbolic expressions in the same way as KLEE, hence we will not provide its detail. Our definition of $[\![\cdot]\!]^2_e$ for computing the error expressions is, however, the core of our approach and we present it in Figure 6. These rules are derived for relative error from standard absolute error propagation rules.

The rules [BR], [BTRUE], and [BFALSE] in Figure 5 describe our symbolic execution semantics for an unconditional branch, a conditional branch that jumps to its first (true) branch and another conditional branch that jumps to its second (false) branch, respectively. For the rules [BTRUE] and [BFALSE], we abstract away the branching decisions, which are made by KLEE. In the definition of both of these rules, we use the function *cond* which builds a constraint on the path condition with error. The purpose of constructing such condition is to further constrain the computed error expression, based on the branch condition being symbolically executed. Given $cmp \in CmpOp$, we define $cond(\sigma, r_1\ CmpOp\ r_2)$ as follows.

$$\sigma_v(r_1).(1 - \sigma_e(r_1))\ cmp\ \sigma_v(r_2).(1 - \sigma_e(r_2)).$$

The accumulation of each such constraint constitutes the construction of the path condition with error, which is one of the outputs of our symbolic execution.

3.2.1 An Example of Symbolic Expression Generation

Figure 7 shows the symbolic error expression generation with shadow structures for the computations along each branch for the example given in Figure 2, if we remove the `else-if` branch and start from the state shown in root node. π is KLEE's path condition with no error while π_e is the path condition with error as described above. σ_v is the program

$$\frac{\iota = r \leftarrow \texttt{alloc}\ i \qquad a = \texttt{malloc}(i) \qquad \sigma'_v = \sigma_v[r \mapsto a] \qquad \sigma'_e = \sigma_e[r \mapsto 0]}{(\iota, \langle l, \sigma, h, \pi_e\rangle) \longrightarrow \langle next(l), \sigma', h, \pi_e\rangle}\ [\text{ALLOC}]$$

$$\frac{\iota = r \leftarrow \texttt{store}\ r_1\ r_2 \qquad a = concretize(r_1) \qquad h'_v = h_v[a \mapsto \sigma_v(r_2)] \qquad h'_e = h_e[a \mapsto \sigma_e(r_2)]}{(\iota, \langle l, \sigma, h, \pi_e\rangle) \longrightarrow \langle next(l), \sigma, h', \pi_e\rangle}\ [\text{STORE}]$$

$$\frac{\iota = r \leftarrow \texttt{load}\ r_1 \qquad a = concretize(r_1) \qquad \sigma'_v = \sigma_v[r \mapsto h_v(a)] \qquad \sigma'_e = \sigma_e[r \mapsto h_e(a)]}{(\iota, \langle l, \sigma, h, \pi_e\rangle) \longrightarrow \langle next(l), \sigma', h, \pi_e\rangle}\ [\text{LOAD}]$$

$$\frac{\iota = r \leftarrow op\ r_1 \qquad op \in UnOp \qquad \sigma'_v = \sigma_v[r \mapsto [\![op\ r_1]\!]^1_v\sigma] \qquad \sigma'_e = \sigma_e[r \mapsto [\![op\ r_1]\!]^1_e\sigma]}{(\iota, \langle l, \sigma, h, \pi_e\rangle) \longrightarrow \langle next(l), \sigma', h, \pi_e\rangle}\ [\text{AP}_1]$$

$$\frac{\iota = r \leftarrow op\ r_1\ r_2 \qquad op \in BinOp \qquad \sigma'_v = \sigma_v[r \mapsto [\![op\ r_1\ r_2]\!]^2_v\sigma] \qquad \sigma'_e = \sigma_e[r \mapsto [\![op\ r_1\ r_2]\!]^2_e\sigma]}{(\iota, \langle l, \sigma, h, \pi_e\rangle) \longrightarrow \langle next(l), \sigma', h, \pi_e\rangle}\ [\text{AP}_2]$$

$$\frac{\iota = r \leftarrow \texttt{br}\ l_1}{(\iota, \langle l, \sigma, h, \pi_e\rangle) \longrightarrow \langle l_1, \sigma, h, \pi_e\rangle}\ [\text{BR}]$$

$$\frac{\iota = r \leftarrow \texttt{br}\ r_1\ l_1\ l_2 \qquad \pi'_e = \pi_e \wedge cond(\sigma, r_1)}{(\iota, \langle l, \sigma, h, \pi_e\rangle) \longrightarrow \langle l_1, \sigma, h, \pi'_e\rangle}\ [\text{BTRUE}]$$

$$\frac{\iota = r \leftarrow \texttt{br}\ r_1\ l_1\ l_2 \qquad \pi'_e = \pi_e \wedge \neg cond(\sigma, r_1)}{(\iota, \langle l, \sigma, h, \pi_e\rangle) \longrightarrow \langle l_2, \sigma, h, \pi'_e\rangle}\ [\text{BFALSE}]$$

Figure 5. Semantics of our simple programming language. The function *next*(l) returns the next label after l. `malloc` represents the actual UNIX system call as is done in KLEE. We also assume a function *concretize* which models KLEE's concretizing of symbolic addresses. The function *cond* computes the constraint to be added to the error path condition.

state containing symbolic values and expressions of the computations and σ_e holds their corresponding error expressions. As can be seen, the error expression for $x1\_r$ (the expression for $x1\_r$ in σ_e) is different in the two branches because the computation of this variable in the two branches are different. These error expressions are formed according to rules laid out in Figure 5 for the different operations involved in the computation.

4 Implementation

4.1 Extending KLEE

We modify KLEE (LLVM 3.4) such that during symbolic execution, a shadow structure builds the error expressions and path conditions with error (described in Section 3.2) [4]. The error expression is built using KLEE's `Expr` API. Our extension to KLEE is available as an open source software at https://github.com/ApproxSymate.

$$
\begin{aligned}
[\![\texttt{add}\ r_1\ r_2]\!]^2_e &= \lambda\sigma.\begin{cases} \dfrac{\sigma_v(r_1).\sigma_e(r_1)+\sigma_v(r_2).\sigma_e(r_2)}{d} & \text{if } d \neq 0 \\ 0 & \text{otherwise} \end{cases} \\
&\quad \text{where } d = [\![\texttt{add}\ r_1\ r_2]\!]^2_v\sigma \\
[\![\texttt{sub}\ r_1\ r_2]\!]^2_e &= \lambda\sigma.\begin{cases} \dfrac{\sigma_v(r_1).\sigma_e(r_1)+\sigma_v(r_2).\sigma_e(r_2)}{d} & \text{if } d \neq 0 \\ 0 & \text{otherwise} \end{cases} \\
&\quad \text{where } d = [\![\texttt{sub}\ r_1\ r_2]\!]^2_v\sigma \\
[\![\texttt{mul}\ r_1\ r_2]\!]^2_e &= \lambda\sigma.\sigma_e(r_1)+\sigma_e(r_2)-\sigma_e(r_1).\sigma_e(r_2) \\
[\![\texttt{div}\ r_1\ r_2]\!]^2_e &= \lambda\sigma.\begin{cases} \dfrac{\sigma_e(r_1)-\sigma_e(r_2)}{1-\sigma_e(r_2)} & \text{if } \vdash \sigma_e(r_2) \neq 1 \\ \sigma_e(r_1)-\sigma_e(r_2) & \text{otherwise} \end{cases} \\
[\![op\ r_1\ r_2]\!]^2_e &= \lambda\sigma.\sigma_e(r_1)+\sigma_e(r_2) \\
&\quad \text{when } op \notin \{\texttt{add}, \texttt{sub}, \texttt{mul}, \texttt{div}\}
\end{aligned}
$$

Figure 6. Semantics function $[\![\cdot]\!]^2_e$ for error expressions. Here we use $\vdash \varphi$ to denote that KLEE's bitvector solver was able to prove that φ holds.

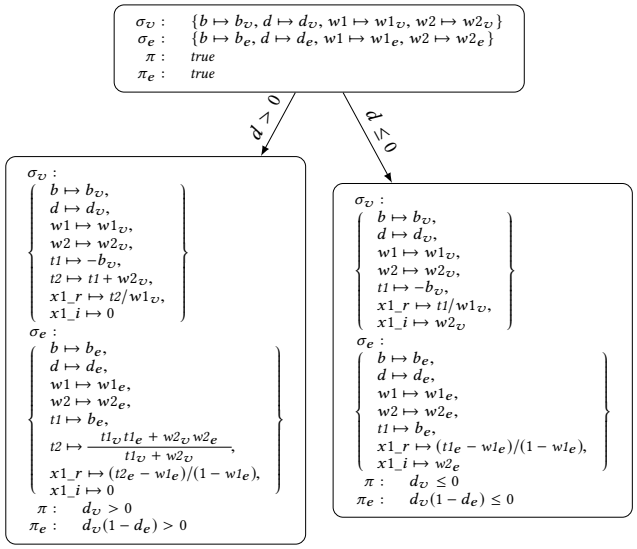

Figure 7. Example of symbolic error expression generation (assuming the first and last code blocks only). The root, left and right nodes represent the symbolic state before, and after the executions of the true and false branches of the program in Figure 2, respectively. π here is the path condition generated by KLEE, and tN are temporary registers, commonly found in LLVM.

KLEE provides a set of API for instrumenting the program under test. These instrumentations are useful to trigger the execution of special functionalities provided by KLEE. The `klee_make_symbolic` function is one such main API used to declare symbolic variables. Our extension adds the following function to the KLEE API:

- `void klee_track_error(void *addr, const char *name)`. This function is used to indicate the uncertainty of a symbolic input (declared by `klee_make_sym-bolic`). It triggers the generation of a symbolic value for that input's relative error. The input is contained in an object stored in memory at the address `addr`. `name` is a string used to identify the generated symbolic input error. If the input is an array, all array elements are deemed to have the same error.

To control the behavior of our error expression computation, we add the following command-line options to KLEE:

- `-approximate` - enables the symbolic error expression computation.
- `-debug-approximation` - outputs debug information related to the construction of the symbolic error expressions.

During its execution KLEE checks whether memory accesses are within the bounds of the memory allocated. Therefore, it does not allow memory addresses to be symbolic. This agrees with our (conservative) assumption that array indices and pointers should not be approximated since that would create potentially unpredictable error behavior in the program.

For each execution path `N` of the program, standard KLEE produces the following files:

- `testN.kquery`: Symbolic expression for path condition of path `N`.
- `testN.ktest`: Concrete values for the specified symbolic input satisfying the path condition of path `N`.

In addition, our extended KLEE produces the following files:

- `testN.expressions`: The symbolic error expressions for all program variables modified within path `N`. The error expression at the last modification is given since that will have the most likely impact on the program's output.
- `testN.kquery_error`: The path condition with error for the path `N`. For example, this is π_e in Figure 7.
- `testN.mathf`: Arguments and errors of `N`'s math function calls (see Section 4.2)
- `testN.prob`: Probability and length of execution tree of `N`. This is used mainly for comparison purposes (see Section 5)

4.2 Working with Floating Point

Many floating point applications are suitable for approximation. However, KLEE (LLVM 3.4 which is the stable version currently) does not provide support for symbolic execution of floating point programs (although it is planned). Therefore, we used the following workarounds to identify approximations in floating point code:

1. Floating point variables are first converted to integer ones. If there are floating point constants, wherever possible they will be replaced with integer constants. In cases where this is not possible, the variable will be replaced with a new symbolic value in KLEE. The actual value of the constant will then be substituted during the sensitivity analysis.
2. The `--scaling` command line argument is used to run programs that originally contain floating point division. This prevents floating point divisions from resulting in zero when the program is converted to use integers by multiplying the numerator by a new symbolic variable. This new variable will be substituted with 1.0 in the sensitivity analysis.
3. The `--math-calls` command line argument is used for programs that contain floating point math functions such as `sin, cos, fabs` etc. Again, new symbolic variables will be introduced to the execution and they will eventually be substituted by the original functions and arguments during sensitivity analysis.
4. Sensitivity analysis is then performed with floating point values as inputs.

Possible drawbacks of this approach are that some paths which are satisfiable in the integer domain maybe rendered unsatisfiable by KLEE's constraint solver in the floating-point domain and the new symbolic variables may also lead to unsatisfiable states. Nonetheless we believe that this improvisation is still useful, especially in enabling us to tackle a much wider range of applications.

4.3 Sensitivity Analysis

Once the extended KLEE generates the output files as described in Section 4.1, we run a sensitivity analysis on them. Our sensitivity analysis, implemented in Python, has the capability to calculate the sensitivity to input error of each variable on each program path. It can also determine whether introducing input error to a variable causes the program to deviate from the original execution path. It uses this information to then output the approximability of the variable, input and path.

Algorithm 1 shows the sensitivity analysis designed for this purpose. It starts by retrieving symbolic error expressions for its program variables and path condition (with and without error) for the current path p (Lines 7 - 10). Concrete values for inputs marked as symbolic are also obtained (Lines 11 - 12). For each program variable's symbolic error expression, it will then take each symbolic input at a time and generate a random value in a uniform distribution of (0,1) as its error. All other input's errors are set to zero (Lines 14 - 22). The input error values (random value for the selected input and zero for others) and input values are then applied to the symbolic expression for the path condition with error of p (Line 23). If the path condition with error is satisfied, then this means that even after applying error to the input, the program's execution path does not change. In this case, the analysis obtains the variable's relative error from its symbolic error expression by applying the input and input errors to it (Line 24). This is repeated for *REP* number of times to obtain *REP* expression output error values for the *REP* random input error values. These pairs are sorted by the input error values (Line 27). They are then divided into N equal groups and for each group the linear regression coefficient is obtained (Lines 28 - 31). Here $N < REP/2$. The gradient of 45° compared with the maximum linear regression determines approximability (Line 32). This means that the output error is never greater than the input error. This kind of demarcation is also done in other tools [28, 29]. For each program variable, this process is repeated with all the symbolic inputs marked with error. If a variable is approximable for at least one input, it is marked 'approximable' in path p.

The sensitivity analysis begins by assuming that all inputs that have been marked with `klee_track_error`, have uncertainty. However in reality, not all inputs may tolerate error. Therefore during the analysis, for each input we keep track of the total times the path condition is satisfied when error is applied to it (Line 26). This is output from the analysis as a fraction over the total attempts of applying error to it (Line 42). If introducing error to a particular input fails to satisfy the path condition more often, then that means that the input is less tolerant to error and vice-versa. If the aforementioned value is greater than at least 50% we maintain our assumption of approximability for that input. This is also useful when inferring the approximability of variables which are not *directly* computed from the input but instead whose values are determined by path conditions (in our evaluation we still mark such variables as non-approximable since they will not have an error expression). Using the variable and input approximability, the tolerance to error of a path can be determined by the user.

During evaluation, when obtaining concrete inputs for floating point programs, we use those that are provided by the benchmarks or generate them using a constraint solver. In the algorithm we assume that the variable's error function is Lipshitz continuous. Because we only allow for error when the original path condition is satisfied and from our empirical observation this assumption is justified. The piecewise linear regression allows the algorithm to better look at the actual error function while randomizing the input errors allow for identification of any abnormalities in the error function.

Algorithm 1 Algorithm for Sensitivity Analysis

```
1:  procedure SENSITIVITY(I, I_e, P, REP)
    ▷ I: Symbolic variables for inputs
    ▷ I_e: Symbolic variables for inputs marked with error
    ▷ P: List of all paths
    ▷ REP: Number of random tests, N: Number of subplots
2:      approximable_var ← []
3:      non_approximable_var ← []
4:      inp_sensitivity ← []
5:      Let ev : I → I_e                    ▷ Get corresponding error variable
6:      Let θ : (I ∪ I_e) → ℝ               ▷ θ gets the concrete value of
                                              a symbolic variable
7:      for each p ∈ P do                   ▷ For each path
8:          Let π^p be the path condition
9:          Let π_e^p be the path condition with error
10:         ξ ← get_symbolic_error_expressions(p)
11:         for each v ∈ I do
12:             θ(v) ← get_concrete_input(π^p, v)
13:             inp_sen[v] ← 0              ▷ Tracks input sensitivity
14:         for each ε ∈ ξ do               ▷ For each symbolic exp.
15:             is_approximable ← FALSE
16:             Plot ← []
17:             for each v ∈ I do           ▷ For each symbolic input
18:                 for each v_e ∈ I_e do
19:                     θ(v_e) ← 0          ▷ Reset all input errors
20:                 v_error ← ev(v)         ▷ Select error var. of v
21:                 for 1 ... REP do
22:                     θ(v_error) ← random(0...1)
23:                     if (θ ⊢ π_e^p) then ▷ Input with error
                                              still satisfy path condition
24:                         out_e = εθ      ▷ Apply input and error
                                              to current expression
25:                         Plot[θ(v_error)] ← out_e
26:                         inp_sens[v] ← inp_sen[v] + 1
27:                 Plot ← Sort(Plot)       ▷ Sort by input error
28:                 SubPlots ← Split(Plot, N)   ▷ Divide
                                              equally into N sub groups
29:                 C ← []
30:                 for each s ∈ SubPlots do
31:                     C ← linear_regression_coefficient(s)
32:                     if Max(C) ≤ 1 then
33:                         is_approximable = TRUE
34:                 x = get_variable_of_expression(ε)
35:                 if is_approximable == TRUE then
36:                     approximable_var.insert(x)
37:                 else
38:                     non_approximable_var.insert(x)
39:             Print(approximable_var)     ▷ Output results
40:             Print(non_approximable_var)
41:             for each v ∈ inp_sensitivity do
42:                 Print(inp_sen[v] ÷ (REP × sizeof(ξ)) × 100%)
```

5 Evaluation

5.1 Experimental Configuration

To the best of our knowledge, currently available automatic approximation tools do not provide path sensitive approximation information. Therefore, to evaluate the effectiveness of ApproxSymate against existing tools, we evaluated its ability to correctly classify program variables as approximable/non-approximable for the most likely program path [15, 28, 29]. The need to compare with multiple tools arises from the fact that there is no single "ground truth" for approximation defined for programs. *REP* was set to 10 by default but can be changed. We found that taking one linear regression coefficient was sufficient for comparison, so $N = 1$. We observed through empirical tests that these values were sufficient for our evaluation.

5.1.1 Path Probability and Depth

ApproxSymate does not rank execution paths but instead allows the user to decide which paths should be approximated. However, ApproxSymate produces a non-approximable/approximable classification for each program path while other tools do not. Therefore, for comparison purposes, we used path probability and depth to determine the most likely program path with the assumption that the results of other techniques are most likely related to this path. We observed that shorter paths often correspond to paths with error conditions while longer paths generate expressions for more program variables. Thus we selected the longest path with the greatest probability as the most likely execution path. Our path probability analysis uses LLVM's branch probability analysis. The probability for a path is obtained by multiplying edge probabilities of all the control-flow edges executed by a path. Other than the probability score, our system also outputs the length of the path, i.e., the depth of path in execution tree. While we used this method for comparison purposes, users are free to use other ways of selecting which path(s) to approximate.

5.2 Comparison with Approximation Tools Against Manually-Annotated Benchmarks

EnerJ is a framework which provides programmers with type qualifiers to specify approximate data in Java programs [32]. EnerJ evaluates benchmarks which are manually annotated with their `@approx` annotation to achieve energy savings of up to 38%. The benchmarks that were compared against EnerJ are given in Table 1. We used the C++ versions of their benchmarks, translating others into C to compare our classification against EnerJ's, assuming that their manual annotations are correct. Our evaluation criteria is based on true positive, true negative, false positive, and false negative counts. An agreement between ApproxSymate and EnerJ on the approximability of a particular variable is considered as a true positive. Agreement on the non-approximability

Table 1. Benchmarks compared with EnerJ

Application	Description	Error Metric	LoC
SOR	Scientific kernels from the SciMark2 benchmark	Mean entry diff.	42
SparseMatMult		Mean normalized diff.	42
MonteCarlo		Normalized diff.	59
LU		Mean entry diff.	87
FFT		Mean entry diff.	168
Raytracer	3D image renderer	Mean pixel diff.	184
ImageJ	Raster image manipulation	Mean pixel diff.	336

of a variable is considered as a true negative. When EnerJ considers a variable to be approximable contradicting the result of ApproxSymate, we count this as a false negative, while the opposite is a false positive. Based on these counts we consider the following evaluation criteria:

- **Accuracy** = $\frac{t_p+t_n}{t_p+t_n+f_p+f_n}$ where t_n, f_n, t_p, f_p are the numbers of true negative, false negative, true positive, and the false positive, respectively. This measure how much our classification matches with EnerJ's classification.
- **Precision** = $\frac{t_p}{t_p+f_p}$ measures the frequency with which ApproxSymate indicates a variable is approximable when it is not (according to EnerJ).
- **Recall** = $\frac{t_p}{t_p+f_n}$ is the frequency with which ApproxSymate indicates a variable is not approximable when it is (according to EnerJ).

Because our comparison is with a binary classification of approximability (which is path insensitive), considering ApproxSymate's classification for the most likely path for comparison also helps to overcome the problem of some variables not appearing along some program paths. We also assumed that the sensitivity of program variables when using smaller input sizes can be extrapolated to larger input sizes and loop iterations. Hence, we used reduced input sizes and loop trip counts in some of the benchmarks to manage KLEE's runtime when the same computations are applied to the input units. These assumptions are justified by the results and our goal to observe only the *behavior* of the program to error (for classification). To obtain the precise error amounts the framework should be run with the original parameters.

5.2.1 Results

Table 2 shows the results of our comparison with EnerJ as well as the approximation tools ASAC and PAC [28, 29]. ASAC is a dynamic testing tool which perturbs program variables during program execution to test their sensitivity to error. PAC is a static tool which relies on data flow analysis to propagate the error at the output to other program variables. Unfortunately, both these tools are not publicly available. Therefore, the numbers were obtained from the respective publications. As seen from the table, ApproxSymate outperforms both these tools in accurately classifying

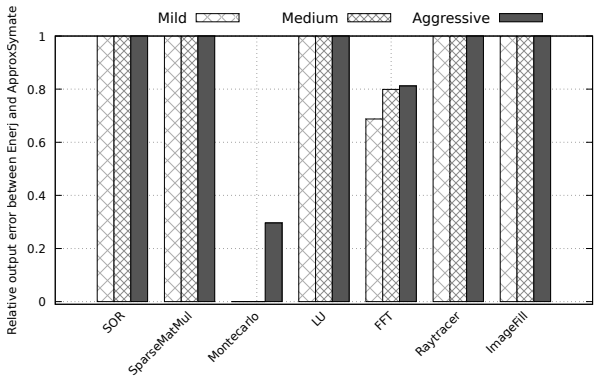

Figure 8. Relative output error between our result and EnerJ measured on different levels of approximation. The y-axis represent the ratio: *ApproxSymate_error ÷ EnerJ_error* using the corresponding error metric given in Table 1.

the approximability of program variables. PAC reports the same classification of variables as EnerJ for Monte-Carlo. The benchmark is small and the difference boiled down to a single variable. How PAC managed to classify this variable cannot be explained by the description of the tool in the paper. In any case, ApproxSymate performs well in precision - it does not incorrectly identify non-approximable variables as approximable. This is important for safe execution of the program. The recall numbers can be explained by the conservative approach taken by ApproxSymate in favor of safety - it does not allow error to cause a change in program path.

During our evaluation, apart from inputs, we also mark constants as being tolerable to error. EnerJ also uses *endorsements* to convert approximate data to precise. In our evaluation, we disregarded these points, since such a transition cannot be detected automatically as it requires specific domain knowledge. This is also the case with array indices and pointers which ApproxSymate strictly assumes cannot be approximated - approximating them can easily lead to undefined program behavior. Therefore variables that are used in calculating pointers or array indices are also deemed not approximable by ApproxSymate. However in EnerJ these can be approximable since *endorsements* can be used to transition them into precise variables.

Table 3 presents the timing information of ApproxSymate obtained on an Intel Xeon Gold 5118 CPU @ 2.30GHz with 32K L1i and L1d and 1024K L2 caches. ASAC reports timings of up to 30 minutes to run [28]. PAC, being a static technique implemented in GCC, reports much faster timing [29] of a few seconds. ApproxSymate produced better results in shorter or same amount of time. We are unable to compare the timings directly as we were unable to execute these tools on our (newer) platform.

Table 2. Comparison results using EnerJ benchmarks

Benchmark	ApproxSymate							ASAC [28]			PAC [29]		
	t_p	f_p	t_n	f_n	Accuracy	Precision	Recall	Accuracy	Precision	Recall	Accuracy	Precision	Recall
SOR	8	0	10	0	100.0%	100.0%	100.0%	88.0%	83.0%	100.0%	92.0%	100.0%	85.7%
SparseMatMult	4	0	13	0	100.0%	100.0%	100.0%	88.0%	50.0%	100.0%	100.0%	100.0%	100.0%
MonteCarlo	2	0	4	1	85.7%	100.0%	66.7%	80.0%	67.0%	100.0%	100.0%	100.0%	100.0%
LU	10	0	15	2	92.6%	100.0%	83.3%	86.0%	88.0%	88.0%	90.0%	100.0%	80.0%
FFT	15	0	25	2	95.2%	100.0%	88.2%	87.0%	88.0%	88.0%	77.0%	100.0%	56.3%
Raytracer	22	1	12	0	97.1%	95.7%	100.0%	-	-	-	-	-	-
ImageJ	7	0	7	0	100.0%	100.0%	100.0%	-	-	-	-	-	-
Average					95.8%	99.4%	91.2%	85.8%	75.2%	95.2%	91.8%	100.0%	84.4%

Table 3. Timing results

Benchmark	# of paths	Ext. KLEE time (s)	Sens. Ana. time (s)	Total time (s)
SOR	1	0.04	2.65	2.69
SparseMatMult	1	0.06	2.47	2.53
MonteCarlo	2	0.06	2.00	2.06
LU	27	1.24	3.03	4.27
FFT	1	0.07	4.59	4.66
Raytracer	7	10.52	8.40	18.92
ImageJ	20	0.26	5.22	5.48

5.2.2 Error Measurement

We use a similar approach as EnerJ to measure the influence of error injection on the set of approximable variables that were determined by ApproxSymate. The error injection for floating point arithmetic is the same as EnerJ's float type. We used the MPFR library to change the mantissa bitwidth of the approximable variables to 8 bits (aggressive), 12 bits (medium), 16 bits (mild) [12]. For integer type, we did not use the same mechanics as EnerJ's detailed simulation (DRAM refresh, arithmetic timing error, etc.). Instead, we use a simpler approach similar to ASAC [28]: randomly flipping a single bit when a value is read with a probability of 10^{-2} for aggressive, 10^{-3} for medium and 10^{-4} for mild approximation. We re-measured the EnerJ's result according to their annotation in the source code, and compare against our output error. The relative output error of program produced by our approach is then divided by the output error from EnerJ to show the relative error between the two approaches. Note that EnerJ's error is always scaled to 1.0 in each benchmark and each approximation level. This relative error is reflected in Figure 8. Although ApproxSymate has a different accuracy for LU and Raytracer, their error profiles are similar. For LU this is because while arrays can be approximated in Java, we cannot approximate pointer variables used for arrays in C as this may cause the program to crash. For Raytracer the difference in accuracy does not have a drastic effect on the final output error.

5.2.3 Path-specific Accuracy

There are four benchmarks in Table 3 that have multiple paths. Figure 9 shows the accuracy of these for all paths found by KLEE when compared to EnerJ's manual annotation. We notice that shorter paths tend to have less accuracy. Shorter paths also tend to correspond to error conditions. Therefore in practice, depending on the output error threshold, either shorter or longer paths or even a combination can be selected for approximation.

5.3 Comparison with Floating Point Precision Tuning

Floating point precision tuning tools exploit the many bits used to represent the mantissa in the floating point number format [15, 30]. Using dynamic search techniques they aim to find the minimum number of mantissa bits of the program variables required to satisfy an error threshold. We ran the tool described in [15] with a selected error threshold to generate the precision of program variables for benchmark programs. For comparison with ApproxSymate's classification we sort the variables by required precision bits and classify the variables as approximable and non-approximable based on it. The higher the precision required by a variable, the lower its ranking of being approximable. The approach used by ApproxSymate is the same as in Section 5.2.

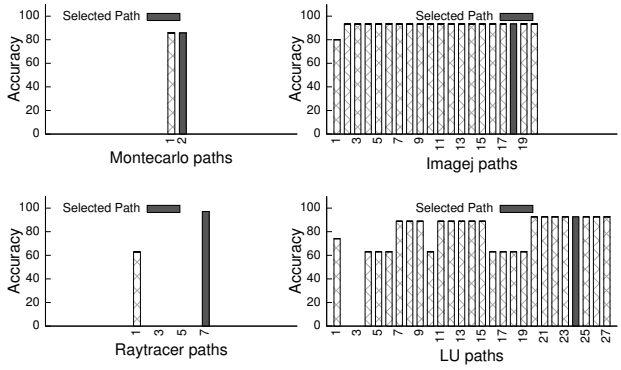

Figure 9. Path-specific accuracy measurement

Table 4. Comparison results with search-based floating point precision tuning of [15]

Program (Benchmark)	LoC	Acc.	Total time (s) (KLEE T. + Sens. T)	FPTuning Time (s)
dct (libjpeg)	343	100.0%	13 (4.17 + 8.36)	1
ep (NAS)	424	69.7%	194 (162.60 + 31.65)	29388
blackscholes (PARSEC)	517	79.6%	34 (14.20 + 19.45)	3
cg (NAS)	1087	76.5%	4 (0.27 + 3.57)	4614
lbm (PARSEC)	1324	81.5%	916 (914.30 + 2.15)	10156

Table 4 shows the information of the benchmarks that were used when comparing against floating point precision turning. It also gives the accuracy results using the same measure described in Section 5.2 (on the same platform). In general, ApproxSymate performs well in the comparison with precision tuning. The differences can be attributed to the varying technique of the two tools - ApproxSymate computes sensitivity to input error while precision tuning uses the actual output error to refine the variables bit width - and inputs. Table 4 also shows the time taken for these benchmarks with both tools (on the same platform as in Section 5.2). As can be seen, floating point precision tuning can be much more expensive in time when compared to ApproxSymate.

5.4 Approximating Program Components

Compared to actual execution, symbolic execution is slower since it maintains symbolic state information, and performs constraint solving on the symbolic information (path condition) to decide branch satisfiability. While techniques mentioned in Section 5.2 can be used to manage the runtime, another possible approach is to break up larger programs and approximate smaller units such as methods, code blocks etc. instead of analyzing the program in its entirety. In this case the variables at the input of these units (eg- method arguments) have to be marked as symbolic and tracked for error. To handle function calls an approach similar to handling math function calls is used (see Section 4.2).

We applied this approach to the `qurt` method of the qurt program from the SNU-RT benchmarks. A simplified version of the code is shown in Figure 2. The variables a, b and c were made symbolic and b was marked as having error. After obtaining the symbolic expressions, the sensitivity analysis was run on each path for the input sets provided in the benchmark. ApproxSymate detected the `if` block is very sensitive to error in b, taking the same path when error was introduced only 2% of the time. The `else-if` block was shown to be the most sensitive to error in b. This is understandable since error in b will very likely cause a path change in this case. The `else` block was the only path shown to be tolerant to error.

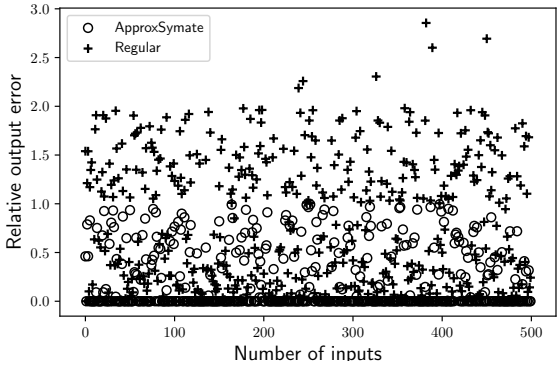

Figure 10. Impact of considering program paths in approximating the `qurt` benchmark.

Figure 10 shows two sets of error measurements for 500 experiments involving the `qurt` benchmark. In one set, errors were injected into the variable b in the `else` block of Figure 2 which was deemed to be approximable by ApproxSymate. In the other set of experiments, errors injected into b everywhere in the code of Figure 2 *without* any regard given to program path. Clearly, the latter produces more errors. In some rare cases when testing with more random inputs, not considering program paths can cause a drastic error of $> 700\%$ while ApproxSymate always gave output errors that are $\leq 100\%$ even in these extreme cases. This again highlights the importance of applying path information when performing program approximation.

6 Related Work

Techniques such as approximate circuitry and memories, voltage scaling etc. that achieve energy savings through approximation at the hardware level have been proposed [13, 16, 18, 22, 25]. Furthermore, there are software tools which allow programmers to specify error-resilient sections in their code [8, 32]. Attempts have also been made to try and reduce the burden of adding manual annotations on programmers [24]. However, here we focus on approaches that automatically identify such program components.

ASAC and ApproxIt are frameworks which employ dynamic testing methods to automatically extract approximation information from programs [28, 35]. Precision tuning is also a dynamic approach which has become popular specially for floating point programs [15, 19, 30]. As is the case with other dynamic approaches they cannot possibly cover all possible tests due to the computational complexity [9, 27]. ApproxSymate reduces this margin of error by statically producing symbolic error expressions for variables and path

condition for testing. PAC is a static technique for approximation roughly based on data flow analysis [29]. While it may be faster, the approximations produced by it are coarse grain. Given a program with a reliability specification and a hardware specification which provides the reliability and accuracy for approximate instructions and memory, Chisel selects which operations to run on the hardware to minimize energy consumption [20]. A significance analysis methodology using interval arithmetic and algorithmic differentiation to automatically rank the significance of variables to program output has been developed [34]. Interval arithmetic is prone to overestimation due to the dependency problem. However, to the best of our knowledge none of these frameworks take into consideration the effect of error on program path during approximation. Leveraging this information can possibly lead to a new dimension of program approximation.

7 Discussion and Future Work

While there is no widely accepted definition of a variable's approximability, or how classifications of approximability should be obtained, many existing tools make use of the relationship between a variable's error and the output's error to determine its approximability. ApproxSymate operates differently in that it primarily observes the relationship between a variable's error with its input's errors. To examine this difference in more detail, consider the following example;

```
// x - input variable, z - output variable
y = x^3;
z = y + 2;
```

Existing techniques determine x and y's approximability by their relationship to z. Then here, suppose y can be deemed approximable (since z is linear in y) and x, non-approximable. Now, if y is computed approximately (line 2), x too will be approximated (according to EnerJ's type system which we assumed) and will lead to unexpected errors in z, since x should not be approximated. This is an example of when y is sensitive to x (input) and thus should not be approximated even though z (output) is not sensitive to y. Similarly an example where the converse it true can also be constructed. Therefore, ideally, a variable's approximability should be determined based on its sensitivity to *both* the input and output. Still, empirically, our results indicate that the both directions of analysis generally yield similar results. Moreover our framework is robust in the following way: if the relative error at the output is known, or provided by the user, the output relative error expression can be analyzed first to determine bounds for the input relative errors. Then these bounds can be used when analyzing the relative error expressions of intermediate variables enabling a bi-directional analysis with little overhead. Furthermore, to observe the effect of error in a particular program variable on the overall output, all that needs to be done is to mark that variable as symbolic and erroneous, and execute the program with our framework.

We chose to use relative error in our symbolic calculation because we found that many approximate applications express error as thresholds, such as "5% error in variable x". Division by zero in relative error is handled empirically by the sensitivity analysis.

Unfortunately, scalability to larger applications is an issue that plagues any technique that uses symbolic execution. There exists a trade-off between execution time and the detail of the analysis. This is also true of other similar tools. To control the exponential escalation of analysis time due to large loop trip counts, we have also implemented a loop breaking mechanism which symbolically estimates the error for loops without executing it. However, it is not working as well as we had expected. We are currently working on improving it.

8 Conclusion

In this work we presented ApproxSymate - a framework which combines a static and dynamic approach to determine approximability in embedded software, safely and efficiently. As far as we know, our work is the first to make use of symbolic execution for program approximation. We augmented the symbolic virtual machine KLEE with a novel shadow symbolic expression mechanism that tracks the relationship between the output and input errors. It also produces the symbolic expressions for path conditions, thus providing the ability to study the effect of program path on approximation error - an aspect previously ignored. We showed that program paths can significantly affect the outcome of approximation. We introduced a sensitivity analysis that evaluates the symbolic expressions to ascertain whether variables, inputs and paths can be approximated. We also showed how floating point code can be dealt with. Used this way, our experiments show that it performs as well or better compared with other state-of-the-art tools for identifying approximable variables. ApproxSymate works well for small to medium size programs found in embedded computing where there are many use-cases for approximation. We also believe that it can be scaled for larger code through composition. Furthermore, we believe that the general approach of ApproxSymate opens up new possibilities for the use of symbolic execution as a technique for reasoning about and discovering opportunities in existing code for the purpose of approximate computing. It will be a useful addition to the set of tools for approximation.

A Artifact Appendix

A.1 Abstract

This artifact contains all the executables of the framework described in the paper "*ApproxSymate: Path Sensitive Program*

Approximation using Symbolic Execution". This primarily includes the augmented version of the symbolic execution engine Klee and the Sensitivity Analysis. It also contains benchmarks which have been modified to work with symbolic execution as well as shell scripts to run them in the framework. The artifact can support the accuracy, precision and recall of approximability classification results presented in the paper.

A.2 Artifact Check-list (Meta-information)

- **Algorithm: Sensitivity Analysis**
- **Program: Benchmarks used by tools Enerj and fptuning which have been modified to run with ApproxSymate is included. To compare ApproxSymate's classification results, the benchmarks with annotations/bitwidth results can be downloaded from their respective repositories.**
- **Compilation: Compiling the augmented Klee requires LLVM-3.4, clang-3.4, stp and klee-uclib. The Sensitiviy Analysis does not require compilation but requires cinpy which needs to compile tinycc. All of these are downloaded, compiled and included with the artifact.**
- **Binary: Binaries for the augmented Klee are included.**
- **Run-time environment: Ubuntu 16.04 with Python3**
- **Hardware: Timing results were obtained on Intel Xeon Gold 5118 CPU @ 2.30GHz with 32K L1i and L1d and 1024K L2**
- **Execution: Run the docker image with multiple cpus**
- **Metrics: Classification of approximability of program variables are reported which can be then used to obtain the accuracy, precision and recall against other Enerj and fptuning classifications.**
- **Output: The output is the classification of approximability of program variables. Result is the accuracy, precision and recall of the classification against Enerj and fptuning.**
- **Experiments: Pull the docker image and run it.**
- **How much disk space required (approximately)?: 3 - 4 GB**
- **How much time is needed to prepare workflow (approximately)?: 0.5 hours**
- **How much time is needed to complete experiments (approximately)?: 3 - 4 hours**
- **Publicly available?: Yes**
- **Workflow framework used?: No**

A.3 Description

A.3.1 How Delivered

1. Pull the docker image with:
   `docker pull himeship/approxsymate:v3`
2. Run the image with:
   `docker run -ti --cpus="<N>" himeship/approxsymate:v3`
   Replace <N> with the number of cpus for the docker image. Using more cpus will improve the speed of evaluation of the sensitivity analysis.

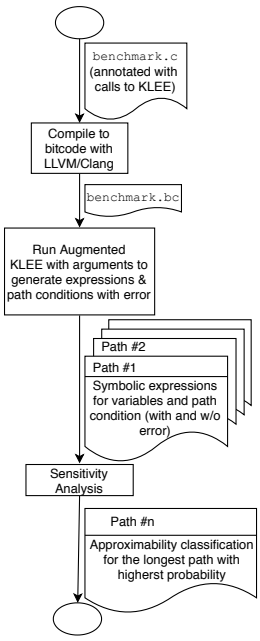

Figure 11. Shell script workflow

A.3.2 Hardware Dependencies

Hardware dependencies for Docker CE.

A.3.3 Software Dependencies

Requires Docker CE.

A.4 Installation

All software and dependencies required to run ApproxSymate have been built and installed into the docker image that can be obtained following the instructions in Section A.3.1. Therefore, only the installation of docker is required for this artifact. Instructions for installing docker can be found at https://docs.docker.com/install/.

Once the docker image is run, the installation of the augmented Klee, sensitivity analysis, and their dependencies can be found in the `/home/approxsymate` folder.

A.5 Experiment Workflow

The shell script for each benchmark will generally execute the workflow shown in Figure 11.

A.6 Evaluation and Expected Result

To evaluate a benchmark with ApproxSymate, after running the docker image as described in Section A.3.1, run the shell script located inside the folder for that benchmark. All benchmarks can be found in the `/home/approxsymate/benchmarks` folder.

For example, to run the Montecarlo benchmark, execute the following,

1. `cd /home/approxsymate/benchmarks/Enerj/montecarlo`
2. `./run_montecarlo.sh`

Running the shell script will produce the approximability classifications for the given benchmark. To obtain the accuracy, precision and recall metrics, this classification must be compared against the classifications of the manually annotated code of Enerj or the

bitwidths of the floating point precision tuning tool. These results are available at the following links.

1. Enerj: https://bitbucket.org/adrian/enerj-apps/get/tip.tar.bz2
2. fptuning: https://bitbucket.org/minhhn2910/c2mpfr/src/

For convenience, our evaluation of classifications against these tools can be found at: https://github.com/ApproxSymate/evaluation.

A.7 Notes

Issues related to the project can be submitted at our Github project - https://github.com/ApproxSymate.

References

[1] Woongki Baek and Trishul M Chilimbi. 2010. Green: a framework for supporting energy-conscious programming using controlled approximation. In *ACM Sigplan Notices*, Vol. 45. ACM, 198–209.

[2] Roberto Baldoni, Emilio Coppa, Daniele Cono DâĂŹelia, Camil Demetrescu, and Irene Finocchi. 2018. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)* 51, 3 (2018), 50.

[3] Ian Buck. 2015. Nvidia's next-gen Pascal GPU architecture to provide 10x speedup for deep learning apps. (2015).

[4] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs.. In *OSDI*, Vol. 8. 209–224.

[5] Cristian Cadar and Koushik Sen. 2013. Symbolic Execution for Software Testing: Three Decades Later. *Comm. ACM* 56, 2 (Feb. 2013), 82–90. https://doi.org/10.1145/2408776.2408795

[6] Michael Carbin, Deokhwan Kim, Sasa Misailovic, and Martin C Rinard. 2012. Proving acceptability properties of relaxed nondeterministic approximate programs. *ACM SIGPLAN Notices* 47, 6 (2012), 169–180.

[7] Michael Carbin, Deokhwan Kim, Sasa Misailovic, and Martin C Rinard. 2013. Verified integrity properties for safe approximate program transformations. In *Proceedings of the ACM SIGPLAN 2013 workshop on Partial evaluation and program manipulation*. ACM, 63–66.

[8] Michael Carbin, Sasa Misailovic, and Martin C Rinard. 2013. Verifying quantitative reliability for programs that execute on unreliable hardware. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 33–52.

[9] Michael Carbin and Martin C Rinard. 2010. Automatically identifying critical input regions and code in applications. In *Proceedings of the 19th international symposium on Software testing and analysis*. ACM, 37–48.

[10] Vinay K Chippa, Srimat T Chakradhar, Kaushik Roy, and Anand Raghunathan. 2013. Analysis and characterization of inherent application resilience for approximate computing. In *Proceedings of the 50th Annual Design Automation Conference*. ACM, 113.

[11] Vinay K Chippa, Debabrata Mohapatra, Anand Raghunathan, Kaushik Roy, and Srimat T Chakradhar. 2010. Scalable effort hardware design: Exploiting algorithmic resilience for energy efficiency. In *Proceedings of the 47th Design Automation Conference*. ACM, 555–560.

[12] Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, and Paul Zimmermann. 2007. MPFR: A Multiple-precision Binary Floating-point Library with Correct Rounding. *ACM Trans. Math. Softw.* 33, 2, Article 13 (June 2007). https://doi.org/10.1145/1236463.1236468

[13] Vaibhav Gupta, Debabrata Mohapatra, Sang Phill Park, Anand Raghunathan, and Kaushik Roy. 2011. IMPACT: imprecise adders for low-power approximate computing. In *Proceedings of the 17th IEEE/ACM international symposium on Low-power electronics and design*. IEEE Press, 409–414.

[14] Matthew R Guthaus, Jeffrey S Ringenberg, Dan Ernst, Todd M Austin, Trevor Mudge, and Richard B Brown. 2001. MiBench: A free, commercially representative embedded benchmark suite. In *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*. IEEE, 3–14.

[15] Nhut-Minh Ho, Elavarasi Manogaran, Weng-Fai Wong, and Asha Anoosheh. 2017. Efficient floating point precision tuning for approximate computing. In *Design Automation Conference (ASP-DAC), 2017 22nd Asia and South Pacific*. IEEE, 63–68.

[16] Andrew B Kahng and Seokhyeong Kang. 2012. Accuracy-configurable adder for approximate arithmetic designs. In *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*. IEEE, 820–825.

[17] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*. IEEE Computer Society, 75.

[18] Song Liu, Karthik Pattabiraman, Thomas Moscibroda, and Benjamin G Zorn. 2012. Flikker: saving DRAM refresh-power through critical data partitioning. *ACM SIGPLAN Notices* 47, 4 (2012), 213–224.

[19] Harshitha Menon, M Lam, D Kuffour, Markus Schordan, S Llyod, Kathryn Mohror, and Jeff Hittinger. 2018. *ADAPT: Algorithmic Differentiation for Floating-Point Precision Tuning*. Technical Report. Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States).

[20] Sasa Misailovic, Michael Carbin, Sara Achour, Zichao Qi, and Martin C Rinard. 2014. Chisel: Reliability-and accuracy-aware optimization of approximate computational kernels. In *ACM SIGPLAN Notices*, Vol. 49. ACM, 309–328.

[21] Subrata Mitra, Manish K Gupta, Sasa Misailovic, and Saurabh Bagchi. 2017. Phase-aware optimization in approximate computing. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization*. IEEE Press, 185–196.

[22] Sparsh Mittal and Jeffrey S Vetter. 2016. A survey of techniques for modeling and improving reliability of computing systems. *IEEE Transactions on Parallel and Distributed Systems* 27, 4 (2016), 1226–1238.

[23] M-E Nilsback and Andrew Zisserman. 2006. A visual vocabulary for flower classification. In *Computer Vision and Pattern Recognition, 2006 IEEE Computer Society Conference on*, Vol. 2. IEEE, 1447–1454.

[24] Jongse Park, Hadi Esmaeilzadeh, Xin Zhang, Mayur Naik, and William Harris. 2015. Flexjava: Language support for safe and modular approximate programming. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 745–757.

[25] Ashish Ranjan, Arnab Raha, Swagath Venkataramani, Kaushik Roy, and Anand Raghunathan. 2014. ASLAN: Synthesis of approximate sequential circuits. In *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*. IEEE, 1–6.

[26] Martin Rinard. 2006. Probabilistic accuracy bounds for fault-tolerant computations that discard tasks. In *Proceedings of the 20th annual international conference on Supercomputing*. ACM, 324–334.

[27] Michael Ringenburg, Adrian Sampson, Isaac Ackerman, Luis Ceze, and Dan Grossman. 2015. Monitoring and debugging the quality of results in approximate programs. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 399–411.

[28] Pooja Roy, Rajarshi Ray, Chundong Wang, and Weng Fai Wong. 2014. Asac: Automatic sensitivity analysis for approximate computing. In *ACM SIGPLAN Notices*, Vol. 49. ACM, 95–104.

[29] Pooja Roy, Jianxing Wang, and Weng Fai Wong. 2015. PAC: program analysis for approximation-aware compilation. In *Compilers, Architecture and Synthesis for Embedded Systems (CASES), 2015 International Conference on*. IEEE, 69–78.

[30] Cindy Rubio-González, Cuong Nguyen, Hong Diep Nguyen, James Demmel, William Kahan, Koushik Sen, David H Bailey, Costin Iancu, and David Hough. 2013. Precimonious: Tuning assistant for floating-point precision. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. ACM, 27.

[31] Mehrzad Samadi, Janghaeng Lee, D Anoushe Jamshidi, Amir Hormati, and Scott Mahlke. 2013. Sage: Self-tuning approximation for graphics engines. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 13–24.

[32] Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. 2011. EnerJ: Approximate data types for safe and general low-power computation. In *ACM SIGPLAN Notices*, Vol. 46. ACM, 164–174.

[33] Stelios Sidiroglou-Douskos, Sasa Misailovic, Henry Hoffmann, and Martin Rinard. 2011. Managing performance vs. accuracy trade-offs with loop perforation. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM, 124–134.

[34] Vassilis Vassiliadis, Jan Riehme, Jens Deussen, Konstantinos Parasyris, Christos D Antonopoulos, Nikolaos Bellas, Spyros Lalis, and Uwe Naumann. 2016. Towards automatic significance analysis for approximate computing. In *Code Generation and Optimization (CGO), 2016 IEEE/ACM International Symposium on*. IEEE, 182–193.

[35] Qian Zhang, Feng Yuan, Rong Ye, and Qiang Xu. 2014. Approxit: An approximate computing framework for iterative methods. In *Proceedings of the 51st Annual Design Automation Conference*. ACM, 1–6.