PAC : Program Analysis for Approximation-aware Compilation

Pooja Roy*
School of Computing
National University of Singapore
pooja.roy@amd.com

Jianxing Wang
School of Computing
National University of Singapore
wang1988@comp.nus.edu.sg

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

ABSTRACT

Approximate computing is a paradigm for trading off program accuracy to save energy in memory or computational resources. However, determining feasible program approximations is difficult to achieve. Popular solutions involve programmer in annotating instructions or data that can be approximated. Recently, program testing based techniques have also been explored. But these are computationally expensive and time consuming as they require running the applications many times over. In this paper, we propose *PAC - Program Analysis for Approximation aware Compilation*, a compiler framework to extract feasible approximation in a program. The state-of-the-art competitors only partition instructions or program data into accurate or approximable. In PAC, instructions and program data are assigned with a degree of accuracy required to maintain user specified Quality-of-Service (QoS) of an application. Such information allows the approximation to be fine-tuned in line with changes in the QoS requirements.

Categories and Subject Descriptors

[**ESS1.5**]: Compilation strategies, code transformation and parallelization techniques for embedded systems

Keywords

Approximate Computing, Code Generation, Program Analysis

1. INTRODUCTION

Approximate or inexact computing trades-off accuracy of applications to save memory or computational resources, and is especially attractive for power-constrained embedded devices. Low power approximate adders produce inexact sum of the inputs and introduce approximation in arithmetic operations [9, 27, 6, 20, 10].

*The author is currently working in AMD, India Pvt. Ltd.

Approximate memories operate at lower voltages saving substantial energy at the risk of possibly compromising the accuracy of the data stored [23, 13, 2]. Such approximate circuits and devices require collaboration from the software stack. Certain Instruction Set Architecture (ISA) extensions enable approximate hardware to switch between accurate and approximate computation during runtime [4].

However, identifying instructions or data of a program where approximation could be allowed without a loss of in the overall quality of service (QoS) of the application is a difficult task. The state-of-the-art methods rely on expressed type-classifiers and pragmas to indicate critical and approximable constructs in the source code [22, 1], thereby transferring the responsibility to the programmer. A few recent works involving profiling and iterative testing of applications are, unfortunately, computationally intensive [14, 21]. Augmenting such techniques with a static analysis is complementary and would reduce their runtime and overheads.

In this paper, we introduce *PAC - Program Analysis for Approximation aware Compilation*, a compiler framework to analyse and identify appropriate parts of a program where approximation may be applied with an acceptable loss in QoS. PAC computes a *degree of accuracy* (DoA) for each *program component*, that is required to attain the QoS of the program. The DoA is a metric for quantifying the approximation of a program component. A *program component* can be a variable, operation, instruction, function call, basic block or a procedure. Section 2 elaborates on the notion of DoA.

PAC outperforms the current state-of-the-art techniques in the following ways -

- PAC is a purely *static* framework, and, therefore, does not require computationally expensive runs to extract approximable program constructs. For instance, ASAC [21], Chisel [14] and ApproxIt [28] are techniques that explores a search-space by running the application repetitively to achieve an acceptable approximation regime. A program analyzed with PAC, can significantly reduce the search space and overheads of such dynamic testing methods. Moreover, being a compile-time technique, PAC is easy to use and easily complements other techniques.

- PAC takes an application and its QoS requirements (translated as the DoA of the outputs) as input and automatically computes the DoA of the program components in the application. These in turn can be used to automatically (or semi-automatically, keeping the programmer in the loop) generate type-classifiers like `@approx`, `@endorse` [22] and annotations [1] to facilitate approximation.

- PAC assigns a quantifiable measure of accuracy i.e. the DoA metric, for each program component that indicate their contribution to the overall QoS of the application. Such non-binary classification of data and instructions is more useful than the state-of-the-art binary (approximable or accurate) classification. For example, on reconfigurable devices, it is more useful to know *how many bits* of data or an operation can be approximated, than to merely know that it can be approximated.

1.1 Novel Contributions in PAC

The key idea is to propagate the expected accuracy of the output (QoS) to the entire program. Based on the definition and usage of variables and interdependence of the instructions, it computes the DoA for all program components. Firstly, we introduce the *Component Influence Graph (CIG)* that captures the relations between the various components. CIG is a novel graphical representation of data dependance in a polymorphic fashion. The CIG successfully captures the relationship, especially the interdependence, of program components. Using the CIG and dataflow equations, the analysis calculates the DoA.

Secondly, Lee et.al. [12] claimed that program variables affecting the control flow such as conditional statements, must always be accurate and approximation can only be introduced in multimedia data. However, PAC comprises a novel program transformation technique which allows certain conditional statements too, to be approximated.

Finally, PAC introduces the concept of a *Degree of Accuracy* of the program components that quantifies the extent to which a program component can be approximated, without user given QoS constraints.

1.2 Target Architectures

PAC is useful to architectures that supports approximate computing. Kahng et.al. [9] proposes an accuracy configurable adder which can adaptively adjust during runtime based on the required accuracy. As PAC provides the required accuracy of all addition or arithmetic operations, it is possible to exploit the adaptive nature of such adders. Thus, instead of only allowing an addition to be approximated, PAC can provide such adders with the DoA of that particular addition i.e. how many bits can be approximated. Memories that can control power supply at the bit level are widely explored [11, 5, 8]. For such memories, it is imperative to know how many bits of a variable is approximable. We believe that such information can be derived from PAC's DoA.

1.3 Approximations in PAC

In PAC, approximations are not applied indiscriminately to all data or operations. Rather, only data that does not impact control flow or memory accesses are approximated. In other words, pointers and memory addresses are never approximated as they are considered critical. In addition, other variables affecting control flow such as integer constants, integer loop bounds are also regarded as non- approximable. Only variables and instructions that do not affect the termination and behavior of a program are deemed as approximable. This is a common practice in approximate computing and PAC adheres to that.

1.4 Evaluation Summary

We compared PAC with the state-of-the-art techniques proposed in [22, 21, 12, 26, 3, 24]. Compared to current state-of-the-art techniques of approximate computing, PAC achieves a high accuracy of 92% (compared to [22]) and 85% (compared to [21]). In addition, runtime of PAC is $\approx 10^3\times$ less than ASAC [21]. When compared to [3], PAC achieves an accuracy of 91% on average. A detailed presentation on evaluation results are in Section 3.

2. PAC

The key idea of PAC is to propagate the accuracy (given as user defined QoS margins) required by the output to all the program components.

Formally, we define DoA it as follows -

Definition 2.1. *Degree of Accuracy*- For a variable v, DoA(v) is the accuracy required to maintain the QoS margins of the application. If DoA(v) = 1, it indicates that all the bits belonging to variable v must be correct in order to remain within the given QoS margins. Conversely, DoA(v) = 0 means that none of the bits of variable v matters to the program output, such variables can be removed by dead code elimination. In practice, the accuracy is usually $0 <$DoA(v) ≤ 1. For instance, for a 10-bit data, DoA can be easily translated as the number of bits that must not be incorrect (inverted).

The QoS of an application is required to be translated to the DoA of output variable(s). PAC assumes that output variable(s) and their DoA is available beforehand and the translation is done apriori. The DoAs are propagated using *influence relations* among the variables. The influence relations connect variables via the *def-use chains* (du-chain) such that an error in one variable impacts the other. A du-chain consists of the definition of a variable and all its uses. Formally we describe an *Influence Relation* as follows:

Definition 2.2. *Influence Relation*- Two variables u and v share an *influence relation* iff an error in u may result an error in v, or vice versa. We define two types of influence relation - *influenced_by* and *influences*. Variable u is influenced_by v if an error in v introduces error in u. We also say variable v influences u.

2.1 Component Influence Graph (CIG)

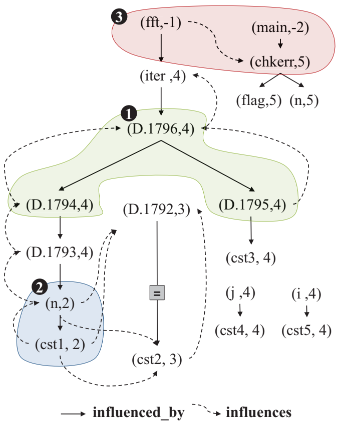
The component influence graph captures the influence relations of all the program variables. Each node in CIG is a tuple consisting of a variable and a basic block identifier. There are two types of edges in a CIG representing the two types of influence relations mentioned above. Figure 1 illustrates a sample kernel of the `FFT` benchmark from MiBench [7] and its corresponding CIG.

These two types of influence relations, though are corollary to each other, are necessary to maintain the correctness and termination of the dataflow analysis. This aspect will be elaborated later in greater details. An 'influenced_by' edge in CIG, connecting two nodes, also contains information about the operator that relates the variables of the nodes. For example, in Figure 1b the edge [(D.1792,3),(cst2,3)] denotes the operator '='. Any node together with its immediate child (or children) can be mapped to an instruction (eg.

```
<bb 2>:                                  fft1
  if (n <= 1)
    goto <bb 3>;
  else
    goto <bb 4>;
<bb 3>:
  D.1792 = 999;
  goto <bb 35>;
<bb 4>:
  D.1793 = (double) n;
  D.1794 = log (D.1793);
  D.1795 = log (2.0e+0);
  D.1796 = D.1794 / D.1795;
  iter = (int) D.1796;
  j = 1;
  i = 0;
  return iter;
```

```
<bb 5>:                                  main
  flag = 0;
  chkerr = fft1 (n, flag);
```

(a) Example Code



(b) Corresponding CIG

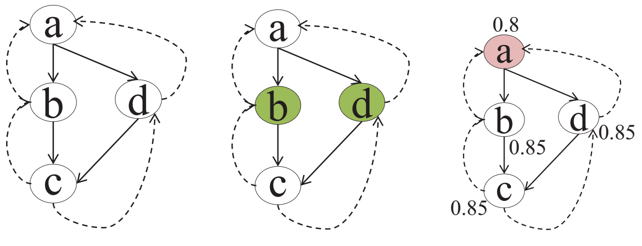
Figure 1: A kernel and corresponding CIG from fft.c (MiBench)

group 1 in Figure 1b). Moreover, a sub-graph of all nodes with the same basic block identifier captures the influence relation for the entire basic block (eg. group 2 in Figure 1b). Special nodes that are tuples consisting of a function name and a negative integer each represents a procedure. Such nodes are connected to the rest of the nodes in CIG via the return value and the parameter variables (eg. group 3 in Figure 1b). A node of CIG together with all its outgoing edges is equivalent to the variables' du-chains. Thus, the CIG is the union of du-chains of all variables of a program. In addition, CIG also contains interprocedural influence relations consisting of CIG nodes that are return variables and parameters of functions.

The CIG is constructed after the control-flow graph during compilation. The detailed explanation of CIG construction is given in Algorithm 1. For each assignment statement (line 6), a CIG node is created for the lhs of the assignment (line 8-10). It is assumed that assignment of a variable v in a basic block bb is a unique pair (v, bb), as in SSA form. Afterwards, n 'influenced_by' edges are created from this node to the existing nodes in the CIG representing n operands of the assignment statement (lines 12,14-17); $1 \leq n \leq 2$ due to SSA form. In addition, from the n operand nodes, one 'influences' edge is created, pointing back to the lhs node. For conditional statements (line 19), 'influences' edges are created from both operands of the condition to all the variables of the target basic blocks (lines 22-24). Note, that no 'influenced_by' edges are created as any event of error in the condition operands would result only in erroneous branching and not errors in other variables of the target basic blocks. Similarly, for function calls, edges are created between the parameters passed and the return value of the function (lines 27-30). These nodes are variables and function identifier (in negative integers) pairs instead of basic block (line 30).

2.2 Accuracy Equations

The CIG, together with a set of accuracy equations, is used to generate the DoA(v) for each variable v. As discussed before, PAC expects the user to provide the DoA of the output variable(s) using annotation. The accuracy equations are then applied to all other variables having an influence relation with the output variables. For example, if DoA(O) is the accuracy of a variable O, then the DoA of any variable V in an influence relation with O is derived from DoA(O) and other variables influencing O. From the runtime perspective, errors occurring in variables are non-trivially dependent events. CIG of a program can easily characterize this phenomena in the following way.



(a) (b) (c)

Figure 2: An example of a CIG showing the 'Error Independence' relations.

Definition 2.3. *Error Independence* - Two variables u and v share an *Error Independence* relation if (a) u does not appear in the sub-graph $G \in$ CIG, where G consists of v with all its children, and (b) v does not appear in the sub-graph $H \in$ CIG, where H consists of u and all its children. Such variables are said to be error independent.

For instance, in Figure 2(a), an error in b would not result in an error in d. However, an error in c would likely result in an error in b. Note that in the CIG, a child node's basic block occurs earlier than its parent in program order. So, Figure 2(b) shows that variables b and d are error independent. However, a and b (or b and c) are not, as they appear in each others' sub-graph in the CIG. We broadly classify instructions into three forms - copy statements, operation statements and branching statements, and define the accuracy equations for each of them.

1. Copy Statements of form $A = B$..

For simple copy statements of this form, the DoAs are calculated as -

$$\text{DoA}(A) = \text{DoA}(B) \tag{1}$$

The propagation of DoAs is a backward dataflow analysis (Section 2.3). So, DoA(B) is equal to the value of DoA(A), which is already known. Thereafter, for copy statements

Algorithm 1 CIG Construction

```
Require: Source code of program
Ensure: dug_head, a pointer to the first node of DUG
 1: for all function in CFG do
 2:     cfun←current function
 3:     for all basic block in cfun do
 4:         bb← current basic block;
 5:         for all instructions in bb do
 6:             stmt_i ← assignment statement i in bb;
 7:             if stmt_i is assignment then
 8:                 lhs=assigned variable;
 9:                 rhs1=first operand; rhs2=second operand;
10:                 (lhs,bb)=CREATE_NODE(lhs);
11:                 if rhs1 is a constant then
12:                     (lhs,bb)→child=FIND_NODE(cst,bb);
13:                 else
14:                     (lhs,bb)→left=(rhs1,bb);
15:                     (lhs,bb)→right=(rhs2,bb);
16:                     (rhs1,bb)→parent=(lhs,bb);
17:                     (rhs2,bb)→parent=(lhs,bb);
18:                 end if
19:             else if stmt_i is conditional statement then
20:                 lhs=first operand; rhs=second operand;
21:                 for each edge E_j from bb do
22:                     bb_j = E_j → dest → bb;
23:                     (lhs,bb)→parent= ∀var ∈ bb_j;
24:                     (rhs,bb)→parent= ∀var ∈ bb_j;
25:                 end for
26:             else if stmt_i is call statement then
27:                 call_return =first operand;
28:                 callee=second operand(func_name);
29:                 return=return value of callee;
30:                 (return,callee)→parent=(call_return,cfun);
31:             end if
32:         end for
33:     end for
34: end for
```

where B is the left hand side expression, the value of DoA(B) will be used to derive the DoA of the variable on the right hand side. In such copy statements, it is said that A has a direct error dependence on B.

2. Operation Statements of form $A = B$ *op* C..

These are standard assignment statements where the error dependencies between B and C are used to derive DoA(B) and DoA(C).

$$\text{DoA}(A) = \text{DoA}(B|C)\text{DoA}(C) + \text{DoA}(C|B)\text{DoA}(B) \quad (2)$$

where DoA($B|C$) is the DoA of B given a DoA of C.

CASE I: B and C are error independent. In other words, an error in B would not result in an error in C or vice versa. Then, DoA($B|C$) = DoA(B) and DoA($C|B$) = DoA(C). However, depending on the type of operator, the effect of the error is different. Assuming the source of error is unbiased, both B and C are equally likely to incur error. So,

$$\text{DoA}(B) = \text{DoA}(C) = \begin{cases} \sqrt{\text{DoA}(A)}, \text{when op} \in \{+,-\} \\ \sqrt{\text{DoA}(A)/2}, \text{when op} \in \{*,/\} \end{cases} \quad (3)$$

Taking the square root prevents the DoA of the operands from diminishing in a long *du* chain. Moreover, it preserves the notion of error accumulation. In other words, errors in both B and C, would result in higher deviation of A. Conversely, a given DoA of A (on the left hand side), would imply that the DoA of the operands (the right hand side) must be higher. The square root also achieves normalization, i.e., $0 \le \text{DoA}(A) \le 1$ always.

CASE II: B and C are not error independent. In this case, B or C must exist in each other's subtree in the CIG. Therefore, there must exist a chain of influence relations between B and C, such that $B \rightarrow X_i \rightarrow C$, where $0 \le i \le n$ for n nodes in the subtree. Also, because the DoAs are propagated backward, the event of error in a variable occurring in a statement is not dependent on an error event occurring *later* in program order. So, for instance, if B is defined at a program point earlier than C, then

$$\begin{aligned} \text{DoA}(C|B) = \text{DoA}(B, X_i) * \\ \text{DoA}(X_i|X_{i+1}) * ... * \text{DoA}(X_{i+n-1}|C) \end{aligned} \quad (4)$$

Note, that in certain sets of assignment statements, there may exist cycles in the CIG. However, during the computation of DoAs as part of the dataflow analysis, we maintain record of already computed DoAs, to avoid diminishing degrees of accuracy for such variables. Moreover, for our preliminary implementation, we considered only statically allocated variables. The analysis proposed however, can be extended to aliased variables with minor amendments.

3. Branching Statements.

Branching is a control flow decision. Every basic block containing a conditional or branching statement, has two successor basic block in the CFG. One of them is taken during execution, while the instructions in the path not taken remains unexecuted.

For example, in Figure 3, if the path taken is $1 \rightarrow 2 \rightarrow 4$ then the instruction `c=a+1;` is never executed. This implies that the instruction `a=10;` or the variable a in basic block 1 can be safely approximated. The branching probability of the edges from a basic block to its successors depicts the likelihood of the path being taken during runtime. This information is easily obtained from the compiler (for example, using the *-fguess-branch-prob* flag for GCC). The edge with less probability (for example, $1 \rightarrow 3$), leads to the basic block containing instructions that are less likely to be executed and thus, are more amenable to approximation. Therefore, for all variables whose reachability is found to be in either of the successor basic blocks and not in both, the DoA is lowered using the branch probabilities. In our experiments, only the branches with highly skewed probabilities of taken/not-taken (i.e., 0.6/0.4 or more) are considered for this. However, it is possible to include more branches for this approximation using additional branch profiling information.

Algorithm 2 elaborates on the method we apply, to handle branching statements. First, the probabilities of each branch edge is obtained (lines 3-4). Reachability of the variables are calculated by applying a standard reachability analysis. For every variable that reaches only one of the destination of the current branching (line 6), the branch probabilities are multiplied with the DoA already obtained using the accuracy equations mentioned earlier (line 9). Multiplication results in lowering of the DoAs of the variables according to whether the branch is taken.

Algorithm 2 Branching Statements' Accuracy Propagation

```
Require: List of basic blocks with edge probabilities
Ensure: Updated DoAs of affected variables
 1: for all branching statements do
 2:     bb←current basic block
 3:     dest_major← target branch with higher edge probability $e$
 4:     dest_minor← other branch with probability $1 - e$
 5:     for ∀ variables v∈ bb do
 6:         if (USED(v)∈dest_minor)∨(USED(v)∉dest_major) then
 7:             stmt← GET_USE(v);
 8:             lhs = GET_LHS(stmt);
 9:             DoA(lhs)=DoA(lhs)*$(1 - e)$;
10:         end if
11:     end for
12: end for
```

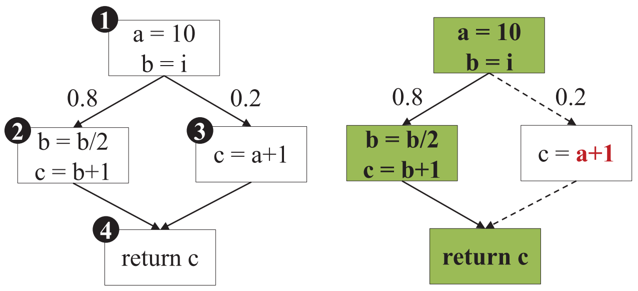


Figure 3: DoA propagation for branching statements in a CFG.

2.3 Analysis & Propagation

DoA propagation is modeled as a program analysis problem. The analysis is solved in an iterative manner where each iteration has two phases. Phase 1 is a backward flow analysis that considers the variables belonging to all statements except conditional statements. Phase 2 is a forward flow analysis for variables involved in conditional statements.

The flow of the analysis is represented in Equations 5 and 6. Phase 1 uses Equations 1 and 3 (Section 2.2). Phase 2 comprises of the technique described as 'form 3' in Section 2.2. Each iteration of the analysis partially fills Equation 4 with the DoAs that are calculated in previous iterations. The analysis attains a maximum fixed point (MFP) solution when the assigned DoAs do not change between successive iterations. This safe termination is ensured by keeping track of variables that have obtained a value other than `Init` in the lattice of the analysis. `Init` denotes the initial state of the variables, which is *Critical*, i.e., a DoA of 1. The analysis results in lowering of the DoAs. Variables with a DoA of 0, i.e., dead variable, will not be consider further in the analysis.

$$\mathrm{Out}(B) \tag{5}$$
$$= \begin{cases} \texttt{Init}, \text{ for } B = \texttt{Exit} \\ \prod_{P \in \mathrm{Succ}(B)} F_{1,2}(\mathrm{OUT}(P)), \ \forall \mathrm{var} \in B \wedge \mathrm{var} \notin \mathrm{COND} \end{cases}$$

$$\mathrm{IN}(B) = \begin{cases} \texttt{Init}, \text{ for } B = \texttt{Entry} \\ \prod_{P \in \mathrm{Pred}(B)} F_3(\mathrm{IN}(P)), \ \forall \mathrm{var} \in B \wedge \mathrm{COND} \end{cases} \tag{6}$$

Algorithm 3 DF Analysis (Partial Algorithm)

```
Require: Control Flow Graph
 1: $BB_e$ ← Entry basic block;
 2: equation[]←set of unsolved accuracy equations;
 3: for all basic blocks bb ∈ CFG do
 4:     if bb ∈ $BB_e$ then
 5:         dfin(bb) = Init;
 6:     else
 7:         dfin(bb) = ⊤;
 8:         worklist = variables v∈ bb;
 9:     end if
10: end for
11: for all basic blocks bb ∈ CFG do
12:     for ∀variables v∈ bb & v∈worklist do
13:         if matches form 1 or 2(I) then
14:             Calculate DoA(v) using equation 1 or 4;
15:             worklist -= v;
16:             FILL(equation,v);
17:         else
18:             if !SOLVE(equation[$bb_v$]) then
19:                 equation[$bb_v$]← partial equation 5;
20:             end if
21:         end if
22:     end for
23: end for
```

Algorithm 3 elaborates on the steps of the analysis as implemented in our framework. It follows the generic steps of a worklist based dataflow analysis with slight modifications. At the outset, the basic block dataflow information is initialized with the ⊤ of the lattice (lines 4-7), i.e. all variables are assumed critical. All variables are added to a *worklist* as they have not been assigned any DoA value at this step (line 8). Afterwards, traversing through the control flow graph (lines 11,12), each statement is matched against the forms discussed in Section 2.2 (line 13,17). If the corresponding accuracy equation can be solved, the variables are assigned the resulting DoA (line 14,18), and are removed from the worklist (line 15). Otherwise, from the program components found, equation 5 is partially filled (lines 16,19). When all the equation elements are available, the equation is solved and the variables are assigned with the DoA (line 18).

2.4 Approximating Comparisons

Comparison expressions are central to branching and loop termination and thus are considered as *critical* instructions [12]. From the perspective of approximate computing, variables in the comparison expression are often considered non- approximable. In our framework, we propose a simple program transformation that allows comparisons too to be approximated without any change in the program behaviour. Apart from the known benefits of approximation, allowing inexact comparison allows for the use of approximate comparators [18], thereby potentially resulting in a better power-performance.

Figure 4 shows a frequently occurring pattern in many program. Typically, there is a loop induction variable or some branching conditions comprising of relational operators, specifically $<, \leq, >, \geq$. For these operators, we propose a transformation technique that allows the particular comparison statement to be safely approximated. As a penalty,

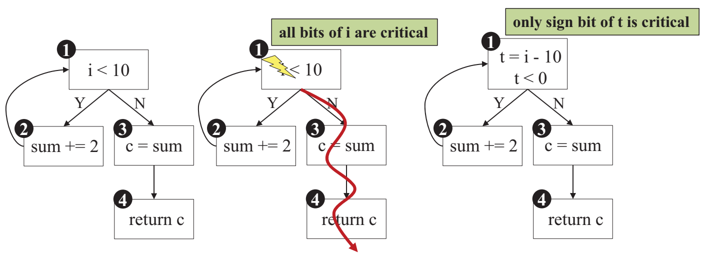


Figure 4: Transformation for approximate comparison.

for every comparison statement, a temporary variable is introduced. Such temporary variables do not pose a large overhead especially for the SSA form.

Definition 2.4. For any comparison statement of the form if(A op B){}, where op $\in \{<, \leq, >, \geq\}$, there exists a pair of statement:

$$\text{temp} = A - B;$$
$$\text{if (temp op 0)}\{\}$$

which is semantically equivalent and can replace the original comparison without any change of the program behavior.

Using the above, all the comparison statements are replaced with the appropriate pair of new statements. For example, in Figure 4, the statement `i<10;`, where `i` is a loop induction variable causing `i<10;` to be executed only in full precision. However, with the transformation `t = i - 10; t < 0;`, only the sign-bit of `t` remains critical and rest of the bits can tolerate errors without any change of the program's behavior. Approximation is thus introduced in the control flow statement with a penalty of one additional computation.

3. EVALUATION

We evaluated PAC in three ways. First, we compared it with the state-of-the-art methods for approximate computing. Next, we compared PAC with compile-time techniques designed for reliability against soft-errors. Such methods categorize program variables into critical and non-critical with the intention of 'hardening' critical data against soft-errors. We will show with our experiments that data identified as non-critical by these techniques are not always approximable. Finally, we evaluated PAC by injecting errors in the applications and thereby measuring the resulting QoS and overhead.

3.1 Comparison with approximation techniques

We compared PAC with two state-of-the-art methods, namely EnerJ and ASAC. EnerJ [22] uses type-classifiers such as `@approx` to annotate program variables meant for approximation. ASAC [21] ranks variables in terms of the output's sensitivity towards them, and allows approximation for less sensitive program variables.

Though PAC can produce DoA for each program component, for the comparison we only considered program variables. Furthermore, in order to perform the comparison, we assumed that variables with DoA less than 0.5 are approximable and rest are not. This is a conservatively assumed threshold and can be fine-tuned according to the demand of the application. We present the standard metrics of true

Benchmarks	GCC -O3 (seconds)	PAC (seconds)	ASAC (seconds)
SOR	0.147	0.168	1345.009
MonteCarlo	0.105	0.113	1929.476
SMM	0.104	0.127	1138.159
LU	0.164	0.186	1831.876
FFT_scimark2	0.135	0.219	1062.417
FFT_MiBench	0.56	0.83	53.069
adpcm	0.342	0.378	222.272
susan	1.2	1.45	30.014
JPEG	6.973	6.601	13.642 (only DCT kernel)

Table 2: Runtime of PAC as compared to standard `-O3` optimization flag in GCC and ASAC

positive (PAC classifies approximate data correctly), false positive (PAC mistakenly classifies critical data as approximable), true negative (PAC correctly identifies critical variables) and false negative (PAC marks a variable as critical where it can be approximated). PAC is a static method and hence it is conservative. In particular, false negatives are to be expected. However, false positives would be unsafe approximation of program variables that might lead to unacceptable QoS or unexpected termination of applications.

In our experiments, we used the Scimark2 [19] benchmarks, as `@approx` annotations are available only for this suite. We applied ASAC to the same benchmarks and present the results in Table 1. PAC achieved an accuracy of 92% when compared to EnerJ, and 74% when compared to ASAC, on average. ASAC is based on profiling of application and thus, has runtime information to analyse the sensitivity of the variables. However, for simple applications like `SOR` PAC is able to have an accuracy that is 85% that of ASAC. The key reason is that it has a simple CFG and the accuracy equations are mostly of forms 1 and 2(I).

The main advantage of PAC over ASAC is the runtime overhead of the analysis. Table 2 shows the different runtime of both methods. PAC is a compiler analysis pass, therefore, we also compare PAC's runtime with the compilation time of `-O3` of GCC. The runtime of ASAC depends on the total number of variables and the dynamic instruction count of the application. We tested comparatively small programs to measure the runtime of PAC and ASAC. Table 2 shows that ASAC is 3 orders of magnitude slower than PAC. As application becomes larger, the difference in runtime also increases. The standard `-O3` optimization in GCC, on the other hand, is 3% faster than PAC on an average (Table 2). In other words, PAC has minimum impact on compile time.

3.2 Comparison with software reliability techniques

To compare with state-of-the-art techniques for ensuring program level reliability, we use three applications, `adpcm`, `susan` and `jpeg` from MiBench [7] and three applications `464.h264ref`, `433.milc` and `482.sphinx3` from SPEC2006 benchmark suites (Table 3).

A. Bitwidth Analysis [Ste00] Bitwidth analysis determines and reduces the number of bits required for program variables [26]. This is often used to minimize the memory budget in silicon compilation. Intuitively, if the bitwidth analyzed by these techniques is shorter than the width of

	EnerJ [22]					ASAC [21]				
Benchmarks	**True Positive**	**False Positive**	**True Negative**	**False Negative**	**Accuracy**	**True Positive**	**False Positive**	**True Negative**	**False Negative**	**Accuracy**
SMM	4	0	4	0	**1**	3	0	3	2	**0.75**
MonteCarlo	2	0	4	0	**1**	2	0	2	2	**0.66**
LU	8	0	12	2	**0.9**	8	0	9	5	**0.77**
FFT	9	0	15	7	**0.77**	9	0	12	10	**0.68**
SOR	6	0	7	1	**0.92**	7	0	5	2	**0.85**
			Average		**0.92**			Average		**0.74**

Table 1: Comparison with EnerJ and ASAC to show PAC's accuracy.

Application	Lines Of Code	Description	Error Metric
adpcm	283	Adaptive differential pulse code modulation (variation of PCM)	SQNR(Signal to Quantization Noise Ratio)
susan	888	Image recognition(edge/corner detection)	Mean Pixel Difference
jpeg	10176	Image compression	SNR (Signal to Noise Ratio)
464.h264ref	18696	Video Compression	PSNR (Peak Signal to Noise Ratio)
433.milc	5401	Quantum Chromodynamics	Error per site (provided with benchmark)

Table 3: Description of the applications

the data type declared by programmer, the extra bits can be approximated safely. With this assumption, we compare PAC's analysis with a state-of-the-art bidwidth analysis [26]. Table 4 the number of variables in the benchmarks classified into three classes:

- CLASS I - these are the variables identified as 'approximable' by both PAC and bitwidth analysis. In particular, bitwidth analysis found that the variable in this case can have a shorter bitwidth than what was originally declared in the program.
- CLASS II - these are the variables identified as 'approximable' by PAC but where bitwidth analysis was not able to say for sure if less bits can be used.
- CLASS III - these are variables that both PAC and bitwidth analysis identified as non-approximable.

A variable with varying bitwidth will now have much approximation opportunity as there is little redundancy in the bits. In such scenario, it is desirable to mark them as non-approximable. Therefore, variables in CLASS III shows a strong correlation between PAC and bitwidth analysis.

However, CLASS II is a class of variables which contradict the usage of bitwidth analysis to characterize approximation in programs. Figure 5a shows that it is possible to approximate variables both from CLASS I and CLASS II without QoS loss. Therefore, we can safely conclude that bitwidth information alone is inadequate in identifying whether a variable is approximable or not, and motivates the need for PAC.

Lastly, Table 4 also shows the coverage of the two methods as a ratio of number of variables analysed by PAC to bitwidth analysis. PAC identifies 3× more variables, on average, that can be approximated, as the premise of approximate computing is to introduce as much as approximation possible to reduce energy consumption. This is due to the fact that PAC considers the interdependence of variables and also transforms conditional statements to more approximable equivalents. In addition, PAC has a better coverage of code, 40% more than bitwidth analysis, chiefly because of interprocedural influence relations.

B. Program Dependency Graph (PDG) Scheme [Cong11] The second scheme we compared PAC with is based on a weighted program dependence graph [3]. The

Application	CASE I	CASE II	CASE III	Coverage
adpcm	28	68	174	1.13
susan	147	435	3064	1.254
jpeg	134	531	1552	1.54
464.h264ref	165	231	46082	1.82
433.milc	152	452	35250	1.1
482.sphinx3	45	276	7348	1.65
Average	111.83	332.16	15578.3	1.41

Table 4: Comparison with bitwidth analysis with no. of variables for all cases (above paragraph) and ratio of code coverage.

Application	True Positive	False Positive	True Negative	False Negative	Accuracy
adpcm	35	6	198	31	0.86
susan	498	31	3034	83	0.96
jpeg	620	45	1470	82	0.94
464.h264ref	312	84	44447	1635	0.96
433.milc	515	89	33268	1982	0.94
482.sphinx3	279	42	6027	1321	0.82
Average	376.5	49.5	14740.66	855.66	0.91

Table 5: Comparison with PDG based scheme with no. of matches identified by both methods and PAC's accuracy.

authors proposed a technique to identify critical data based on the number of references to it in the whole program with the aim of protecting these data against soft errors. The technique classifies the data as *likely critical* (LC) or *likely not critical* (LNC).

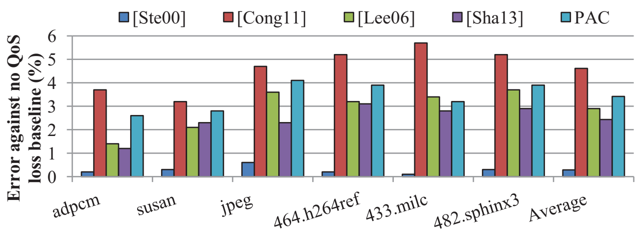
Table 5 shows the match between LC and LNC data with approximable and non-approximable data as characterized by PAC. The 'true positive' represents the number of variables with low (<0.5) DoA and also marked as LNC. Such variables can be safely approximated. 'False Positives' are variables that are marked as LC that, however, has a low DoA. This column suggests that approximability of program variables is not just the function of the total number of references to it.

Later in Section 3.3, we will show that injecting errors into this class of variables also does not result in the loss of QoS. 'True negatives' are variables that both schemes agree on. Lastly, 'false negatives' are cases where PAC characterized the variable as non-approximable, but was marked as LNC.

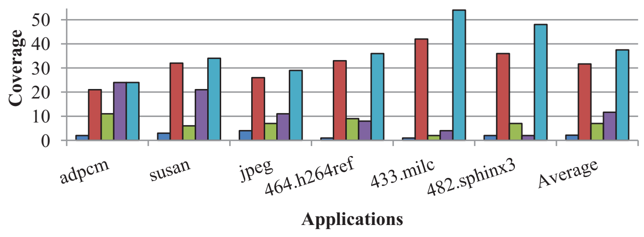
This shows that a variable which does not need extra protection from soft errors, may not tolerate errors aggressively due to deliberate approximation.

C. Multimedia Application Specific Data Partitioning [Lee06] In this method, the authors suggested selective protection of data in multimedia applications [12]. Any variable affecting termination of the application is characterized as critical and multimedia data (input or output) is deemed non-critical. We compare with this scheme in terms of error percentage obtained by running the applications under a synthetic error injection framework described in Section 3.3.

Figure 5b shows that the number of variables marked as approximable or non-critical by this scheme is 7% on average, which is much lesser than PAC's 37.5% on average. Thus, we can conclude that PAC performs better in terms of identifying possible approximation in a program. In other words, it shows that detecting critical components to protect them is not equivalent to identifying opportunities for approximation within the given acceptable QoS loss. In general, analysis for approximate computing can be significantly less conservative than reliability techniques.



(a) Error Percentage (error injected in approximable variables).



(b) Coverage (% of approximated variables in total no. of variables).

Figure 5: Impact of errors injection in approximable variables characterized by different methods.

D. Instruction Vulnerability based characterization [Sha13] The fourth technique is based on error masking and its effect on QoS of an application [24]. The scheme is based on the probability of masking of an error due to bitwise 'AND', shift or other similar operations. This technique, like the previous ones, suffers from poor coverage of source code and considers only specific cases. Figure 5b shows that it provides around 11% of coverage of a program. In applications where bitwise operators do not play a major role, this technique fails to identify possible approximations.

3.3 Impact of Errors

To evaluate the effectiveness of the data characterization, we present the quantitative QoS loss in terms of error percentage in Figure 5a. We used a synthetic error injection framework, which injects a random bitflip into variables that are identified as approximable. During execution, the error injector, randomly selects one or more variables at a uniform interval and injects the bitflip. For each application, the error percentage is calculated based on the correct (provided) value of the metric mentioned in Table 3 and the output obtained upon error injection. On average, PAC accounts for 3.4% of QoS loss. Though, schemes Ste00, Lee06 and Sha13 perform better and shows a QoS loss of 0.28, 2.9 and 2.4 % only, they do not provide a good coverage of approximation in the program. In other words, the total number of approximations allowed according to these schemes are much less than Scheme Cong11 and PAC. This phenomena is presented in the graph of Figure 5b. Poor coverage will lead to much less opportunities to reduce energy or computational resources. So while these schemes are as scalable as PAC, they provide lower quality information.

Applications	Total Conditional Statements	Transformed Conditional	Overhead (10^{-2})
adpcm	157	143	3.2
susan	149	84	2.8
jpeg	1454	1239	1.14
464.h264ref	3128	2743	1.06
433.milc	626	581	0.86
482.sphinx3	1263	912	1.11
Average	1129.5	950.33	1.695

Table 6: Overhead of conditional transformation

3.4 Impact of Approximating Conditions

In addition to the above results, we present the overhead introduced by our novel code transformation that allowed conditional statements to be approximated. For each conditional statement that is transformed, one assignment statement is added to the code (Section 2.4). Table 6 present the number of transformed conditional statements and the overhead in terms of percentage of additional instructions over the total static instruction count of the application.

To check the validity of this transformation, we applied it in the program components identified by PAC as approximable in all the benchmarks. In doing so, we observed that all the benchmarks terminated and produced the correct results. This proves the correctness of the transformation under normal execution environment. Further, approximation (in terms of random bitflip) was introduced to all bits except the sign bit, to aggressively stress test the transformation. We again observed correct results with no unacceptable QoS degradation.

4. RELATED WORKS

Many embedded applications do not require a strict QoS and can thus be approximated as long as they meet certain acceptable thresholds [12]. Approximation can be introduced in both the hardware and software stack. Gupta et.al. [6] introduced IMPACT, an approximate adder that produces inexact sum of two numbers. Many other designs for approximate adders have been proposed thereafter [9, 27, 20, 10]. Besides the operands, most of these adders also have an error tolerance as input. However, they are not fully exploited by the software stack as program analyses to extract approximable operations from programs [22, 21] do not provide an error tolerance for every addition operation. PAC is able to furnish with a fine-tuned error tolerance for each program instruction or data in terms of their DoA required to maintain the QoS. Other works explore various program transformations to allow for disciplined approximation [16, 17, 15, 25]. This includes statistical program testing, loop perforation, etc. However, these techniques introduce approximation only in the software stack. To achieve cross-layer approximations, Sampson et.al. [22] proposed a type-qualifier based programming paradigm to facilitate approximation of program data together with a customized ISA (instruction set architecture) extension [4]. To be effective, these type-classifiers need explicit programming and instrumentation of the source code. Recently, a dynamic testing based method has been proposed [21, 28] that automates the process of extracting approximable program data. Such analyses are computationally intensive with a long running time. PAC is a compile-time technique that does not suffer both drawbacks. However, being a static approach, PAC provides a more conservative outcome of possible approximation of program components than either of the above mentioned methods. Before the emergence of approximate computing, researchers presented similar characterization of critical and non-critical program components [3, 8, 12, 24, 26]. Such techniques were primarily concerns with correctly identifying critical data and protect them against soft-errors. One can relate this to approximate computing: critical data identified by the reliability techniques are non-approximable. However, a non-critical data or instruction is not necessarily approximable. The notion of approximation is stronger than non-criticality. Nonetheless, we compared PAC with various reliability measures and approximate computing techniques in this paper.

5. CONCLUSION

In this paper, we present PAC, a program analysis for approximation aware compilation. PAC computes degrees of accuracy for each program component required to maintain the quality of service of an application. Other than having the user specifying the QoS requirement, PAC is a completely automatic static analysis. Compared to the manual annotation of EnerJ, PAC attains a 92% accuracy. When compared to ASAC, a compute intensive search procedure, PAC attains 74% accuracy in characterizing variables that can be approximated while maintaining user given QoS constraints. However, PAC is $10^3\times$ faster than ASAC. Compared to software reliability methods, PAC achieved better coverage while maintaining the QoS under error injected execution of the applications. In summary, PAC offers something unique to the state of the art. Of the current techniques that computes the same information as PAC, none can scale to the large program that PAC can handle. Compared to similarly scalable software techniques designed for other purposes that may possibly be used to derive DoA, PAC computes higher quality results. We believe that this makes PAC an attractive complementary analysis to enhance other approximation approaches.

6. REFERENCES

[1] M. Carbin, S. Misailovic, and M. C. Rinard. Verifying quantitative reliability for programs that execute on unreliable hardware. OOPSLA '13.

[2] V. Chippa, A. Raghunathan, K. Roy, and S. Chakradhar. Dynamic effort scaling: Managing the quality-efficiency tradeoff. DAC '11.

[3] J. Cong and K. Gururaj. Assuring application-level correctness against soft errors. ICCAD '11.

[4] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger. Architecture support for disciplined approximate programming. ASPLOS XVII.

[5] K. Flautner, N. S. Kim, S. Martin, D. Blaauw, and T. Mudge. Drowsy caches: Simple techniques for reducing leakage power. ISCA '02.

[6] V. Gupta, D. Mohapatra, S. P. Park, A. Raghunathan, and K. Roy. Impact: Imprecise adders for low-power approximate computing. ISLPED '11.

[7] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. WWC '01, 2001.

[8] M. M. Islam and P. Stenstrom. Characterization and exploitation of narrow-width loads: The narrow-width cache approach. CASES '10.

[9] A. B. Kahng and S. Kang. Accuracy-configurable adder for approximate arithmetic designs. DAC '12.

[10] Y. Kim, Y. Zhang, and P. Li. An energy efficient approximate adder with carry skip for error resilient neuromorphic vlsi systems. ICCAD '13.

[11] J. Kong and S. W. Chung. Exploiting narrow-width values for process variation-tolerant 3-d microprocessors. DAC '12.

[12] K. Lee, A. Shrivastava, I. Issenin, N. Dutt, and N. Venkatasubramanian. Mitigating soft error failures for multimedia applications by selective data protection. CASES '06.

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

[14] S. Misailovic, M. Carbin, S. Achour, Z. Qi, and M. C. Rinard. Chisel: Reliability- and accuracy-aware optimization of approximate computational kernels. OOPSLA '14.

[15] S. Misailovic, D. Kim, and M. Rinard. Parallelizing sequential programs with statistical accuracy tests.

[16] S. Misailovic, D. M. Roy, and M. C. Rinard. Probabilistically accurate program transformations. In *Proceedings of the 18th International Conference on Static Analysis*, SAS'11, pages 316–333, Berlin, Heidelberg, 2011. Springer-Verlag.

[17] S. Misailovic, S. Sidiroglou, and M. C. Rinard. Dancing with uncertainty. In *Proceedings of the 2012 ACM Workshop on Relaxing Synchronization for Multicore and Manycore Scalability*, RACES '12, pages 51–60, New York, NY, USA, 2012. ACM.

[18] M. Nesenbergs and V. O. Mowery. Logic synthesis of some high-speed digital comparators. Bell System Technical Journal'13.

[19] R. Pozo and B. Miller. Scimark 2.0. www.math.nist.gov/scimark2/.

[20] A. Ranjan, A. Raha, S. Venkataramani, K. Roy, and A. Raghunathan. Aslan: Synthesis of approximate sequential circuits. DATE '14.

[21] P. Roy, R. Ray, C. Wang, and W. F. Wong. Asac:

Automatic sensitivity analysis for approximate computing. LCTES '14.

[22] 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 '11, 2011.

[23] A. Sampson, J. Nelson, K. Strauss, and L. Ceze. Approximate storage in solid-state memories. MICRO-46.

[24] M. Shafique, S. Rehman, P. V. Aceituno, and J. Henkel. Exploiting program-level masking and error propagation for constrained reliability optimization. DAC '13.

[25] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. Rinard. Managing performance vs. accuracy trade-offs with loop perforation. ESEC/FSE '11.

[26] M. Stephenson, J. Babb, and S. Amarasinghe. Bidwidth analysis with application to silicon compilation. PLDI '00.

[27] R. Ye, T. Wang, F. Yuan, R. Kumar, and Q. Xu. On reconfiguration-oriented approximate adder design and its application. ICCAD '13.

[28] Q. Zhang, F. Yuan, R. Ye, and Q. Xu. Approxit: An approximate computing framework for iterative methods. DAC '14.