Coalescing Executions for Fast Uncertainty Analysis

William N. Sumner Tao Bao Xiangyu Zhang Sunil Prabhakar
Department of Computer Science, Purdue University
{wsumer,tbao,xyzhang,sunil}@cs.purdue.edu

ABSTRACT

Uncertain data processing is critical in a wide range of applications such as scientific computation handling data with inevitable errors and financial decision making relying on human provided parameters. While increasingly studied in the area of databases, uncertain data processing is often carried out by software, and thus software based solutions are attractive. In particular, Monte Carlo (MC) methods execute software with many samples from the uncertain inputs and observe the statistical behavior of the output. In this paper, we propose a technique to improve the cost-effectiveness of MC methods. Assuming only part of the input is uncertain, the certain part of the input always leads to the same execution across multiple sample runs. We remove such redundancy by coalescing multiple sample runs in a single run. In the coalesced run, the program operates on a vector of values if uncertainty is present and a single value otherwise. We handle cases where control flow and pointers are uncertain. Our results show that we can speed up the execution time of 30 sample runs by an average factor of 2.3 without precision lost or by up to 3.4 with negligible precision lost.

Categories and Subject Descriptors

D.1.2 [**Programming Techniques**]: Automatic Programming; D.2.5 [**Software Engineering**]: Testing and Debugging; D.3.3 [**Programming Languages**]: Language Constructs and Features

General Terms

Languages, Experimentation, Performance

Keywords

uncertainty, sensitivity, monte carlo, coalescing

1. INTRODUCTION

Uncertain data processing is becoming more and more important. In scientific computation, data are collected through instruments or sensors that may be exposed to rough environmental conditions, leading to errors. Computational processing of these data may hence draw faulty conclusions. For example, it was shown that a protein was mistakenly classified as a cancer indicator by slightly altering a parameter of the program used to process experimental data [29]. These parameters are uncertain because they are provided by biologists based on their experience. Such mistakes could be highly costly because expensive follow-up wet-bench experiments could be further conducted based on the faulty protein. One of the most widely used sources for protein data is Uniprot [8], in which proteins are annotated with their functions. The annotations may come from real experiments (accurate) or computation based on protein similarity (uncertain). Software operating on these data must be aware of the uncertainty [16]. In modern combat, soldiers rely on data collected by sensors deployed on harsh battlefields. Processing unreliable data and making proper decisions is key to their survival. Software facilitating financial decision making is often required to model uncertainty [19].

Traditionally, uncertainty analysis is conducted on the underlying mathematical models [28]. However, modern data processing uses more complex models and relies on computers and programs, rendering mathematical analysis difficult. Realizing the importance of uncertain data processing, recently, researchers have proposed database techniques to store, maintain and query uncertain data [25, 13]. However, more sophisticated data processing is often performed outside a database. Addressing uncertainty from the software engineering and program analysis perspective becomes natural. Continuity analysis [6] is a static technique analyzing if a given program output varies in a continuous way as input changes. While the analysis has been shown to work on simple programs like sorting algorithms, applying it to programs with complex computation remains a challenge. Taint analysis [21, 7] tracks the set of inputs used to compute individual outputs through program dependence tracking. Users can thus focus their attention on the relevant inputs when analyzing uncertainty. However, it does not provide direct help as it cannot identify whether one input in the lineage set is more important than another.

Monte Carlo (MC) methods provide a simple and effective means of studying uncertainty [13, 5, 26]. They randomly select input values from predefined distributions and aggregate the computed outputs to yield statistical insights on the output space. They are increasingly studied by researchers in software engineering. For instance, MC methods are used to classify input values critical to the output [5] and test

```
1  float foo (float * A, float x) {
2    float t = 10.0 – x;
3    float s=0, y;
4    for (int i=0; i<10; i++) {
5        y=(t+i)*A[i];
6        s=s+y;
7    }
8    return s;
   }
```
(a) Original function

```
10 float foo (float * A) {
11   float s=0, y;
12   for (int i=0; i<10; i++)  {
13       y=(5.0+i)*A[i];
14       s=s+y;
15   }
16   return s;
   }
```
(b) After specialization on x=5.0

```
21 float foo (float * A) {
22   y=5.0*A[0];
23   s=y;
24   s=s+6.0;    //s=s+(5.0+1)*1.0
25   s=s+14.0;   //s=s+(5.0+2)*2.0
26   y=8.0*A[3]
27   s=s+y;
28   …
29 }
```
(c) Trace based specialization on A

Figure 1: Program Specialization. Assume `A[0]` and `A[3]` are uncertain; other `A[i]=i`.

stability of numerical implementations [26].

In this paper, we improve the efficiency of MC methods. MC methods require computing independent solutions, or *trials*, for many samples. In the context of uncertain data processing, often only part of the data are uncertain. Hence, the sample runs have a lot redundancy, dictated by the certain part of the input, i.e. the part that remains the same across sample runs.

Eliminating redundant execution caused by a certain part of the input is not a new challenge. Classic solutions include partial evaluation (program specialization) [15, 17, 24] and memoization. However, they fall short for our purpose. In particular, partial evaluation generates specialized versions of a program that can run faster. For example, suppose a function has multiple input parameters, and some of the parameters often take the same values. Specialized versions of the function can be generated by replacing those parameters and computation related to them with concrete values at compile or binding time. Consider the program in Fig. 1 (a). Suppose variable `x` often takes the value `5.0`. Specialization creates the version in (b). Specialization is harder where only part of an array is uncertain. Recent work specializes execution traces [24], working at the array element level, but such techniques unroll loops and generate linear code [14]. Supposing `A[0]` and `A[3]` are uncertain in our example, figure (c) shows the program after trace based specialization . Observe that statements 2, 3, 4, and part of 5 are concretized as their outcomes are certain. Trace based specialization is intended for use on functions, whereas we need to specialize whole programs. Section 6 shows that when only one input element is uncertain, up to 98.84% of the executed instructions for some programs operate on uncertain data and are hence not concretizable. This implies the specialized programs will have a huge static size.

Memoization [18, 22, 1] is a technique that caches function level output for frequently occurring function inputs so that redundant computation can be pruned. It works well when a function is called frequently in a single run and possible inputs to the function are few. However, our scenario is different: in one sample run, a program may call some functions a large number of times, but likely with different inputs each time; the same function inputs only occur across runs. To reduce such cross-run redundancy, we may have to cache for all the possible function inputs in a single run.

We propose a technique called *execution coalescing* that packs multiple MC trials into one execution. The approach allows users to mark program inputs that should receive samples from a random distribution. Using this information, our approach automatically finds work that is common across multiple trials and *coalesces* it so that the common work is only done once, but the *results* of that work can still be used independently in each trial. In particular, a variable

```
   Trace                           State
2  float t = 10.0 – x;             5.0
3  float s=0;                      0.0
4  for (int i=0; i<10; i++)        T
5      y=(t+i)*A[i]                {-5.0, 0.0, 5.0}
6      s=s+y;                      {-5.0, 0.0, 5.0}
4  for (int i=0; i<10; i++)        T
5      y=(t+i)*A[i]                6.0
6      s=s+y;                      {1.0, 6.0, 11.0}
   …
```

Figure 2: Coalesced execution. Assume three samples are taken for `A[0]` and they are {-1.0, 0.0, 1.0}. The state shows the left hand side values after each statement execution.

is associated with a vector if it is directly or indirectly computed from uncertain input. Assume its size is n. The ith element in the vector corresponds to the value of the variable in the ith sample run. Operations on uncertain values are carried out on individual elements in the vectors. One step forward in the packed execution is equivalent to stepping forward in the n sample runs simultaneously. Variables that are certain have only one value. In this case, stepping forward in the n sample runs is done by executing only one operation instead of executing n instructions simultaneously, which allows eliminating redundancy.

Consider the program in Fig. 1 (a). Assume the random values `-1.0`, `0.0` and `1.0` are chosen for `A[0]` in three separate trials. The coalesced execution of these trials together is shown in Fig. 2. These different values are shown as a vector at the first instance of line 5. Furthermore, the instances of line 6 also have vector values as they rely on the uncertain input and may yield different results for each trial. In contrast, because the values generated at other places are shared across all trials, the state of the variables is represented by individual values. Note that if the three trials are executed independently, the shared evaluations must be repeated three times. For example, line 2 needs to execute three times without coalescing but only once with it.

Recent advances in hardware and software enable efficiency in execution coalescing. Our technique tracks uncertain values that need to be associated with a vector through program dependence tracking, which is often the dominant overhead factor. Recent work [23] shows that such analysis can be implemented efficiently. Moreover, the intel Streaming SIMD Extensions (SSE) instruction set for x86 architectures allows operating on a vector of values in one cycle. Our technique is mostly *transparent* to users. They only need to indicate which input values are uncertain and their distributions. It has no specific requirements for the subject programs and hence can be adapted to other applications such as combinatorial testing. Our current implementation supports both C and FORTRAN programs.

Our contributions are summarized as follows.

- We present vector based program evaluation rules for packing multiple MC trials into one execution. It coalesces the common work between trials while producing unique and correct results for each individual trial. The semantics handles conditional statements on uncertain values. If necessary, it executes both branches, one after the other. It also handles uncertain pointers.
- We conduct formal analysis on the possible savings of our technique that allows the user to estimate benefits in different use cases.
- We devise optimizations to remove unnecessary uncertainty on the fly to further improve cost effectiveness. The optimization allows the user to exchange precision for efficiency. Our evaluation shows that high performance can be achieved with little loss of precision.
- We have implemented the approach and evaluated it to examine the impact and utility of different parameters that characterize efficacy in reducing the runtime costs of Monte Carlo techniques.

2. COALESCING EXECUTIONS

In this section, we formally define execution coalescing and discuss some important properties.

$$\begin{array}{rl}
P \in \mathcal{L} & ::= s \\
s \in \mathit{Stmt} & ::= s_1;\ s_2 \mid \texttt{skip} \mid x\ \leftarrow\ e \\
 & \mid \texttt{if}\ (x)\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{endif} \\
 & \mid \texttt{while}\ (e)\ s\ \texttt{endwhile} \\
e \in \mathit{Expr} & ::= x \mid c \mid x_1\ \texttt{binop}\ x_2 \mid x_1\ \texttt{binop}\ c \mid \mathit{input}() \mid ... \\
x \in \mathit{Var} & ::= \{\texttt{x}, \texttt{y}, \ldots\} \\
c \in \mathit{Const} & ::= \{\texttt{true}, \texttt{false}, 0, 1, 2, \ldots\} \\
\mathit{input}() & ::=\ \perp \rightarrow\ (c \mid \texttt{normal}(c, c) \mid \texttt{uniform}(c, c) \mid ...)
\end{array}$$

Figure 3: Simple kernel language $\mathcal{L}$ with Monte Carlo sampling.

Our system is built on top of *gcc* and hence supports multiple programming languages. For generality, our formal discussion is facilitated with a simple kernel language presented in Fig. 3. The kernel language includes no-ops, assignments, conditional execution via `if` statements, looping with `while` statements, binary operators, and both constant and variable values. It explicitly models program input through the *input*() function. The function returns a certain value or a distribution. The simplest form is a uniform distribution over a range. For the moment, the kernel language does not model functions, arrays and pointers. We will discuss how to support these features in Section 4.

$$\begin{array}{lll}
\mathit{Trial} & ::= & \{1, ..., N\} \\
\sigma \in \mathit{Store} & ::= & \mathit{Var} \rightarrow \mathit{Const} \\
\Gamma \in \mathit{SampleStore} & ::= & \mathit{Var} \rightarrow (\mathit{Trial} \rightarrow \mathit{Const}) \\
\mu \in \mathit{SampleMask} & ::= & \mathcal{P}(\mathit{Trial})
\end{array}$$

Figure 4: Definitions for evaluation.

2.1 With Certain Control Flow

We first introduce how the technique works assuming control flow is certain. In other words, we assume predicates do not operate on uncertain variables, and hence different trials follow the same control flow. For instance, in a matrix multiplication program, if only the matrix elements are uncertain, execution always follows the same path.

Fig. 4 presents the definitions relevant to program state. Symbol N represents the number of sample runs we want to coalesce. We assign a unique id for each sample run. Trial represents the set of ids. Symbol σ represents the regular store, which is a mapping from variables to constants. To allow coalescing, we introduce an extra store in the sample space, denoted as Γ. It maps a variable to a mapping from a sample run id to a value. Intuitively, for each variable, it stores the values of the variable for each sample run if the variable is uncertain. We call the mapping the *vector value* of the variable, whereas the value in the regular store is the *regular value*. Symbol μ is used in the presence of uncertain control flow, which will be discussed in the next section.

The evaluation rules are presented in Table 1. They specify the actions for evaluating a statement when the conditions are satisfied. The rule names are provided in the last column. We allow two types of input through the explicit *input*() method. In rule `Input-Certain`, if the statement reads a certain input value to variable x, x is associated with the value in the regular store and with $\perp$ in the sample store, meaning that x is undefined in the sample store. Note that x might have had an uncertain value before the assignment. Rule `Input-Uncertain` specifies that if the input is uncertain, i.e., the input method returns a distribution, N samples will be taken and stored to the sample space.

The remaining three rules specify evaluation of the assignment of a binary operation. Rule `Binop-Both-Uncertain` applies when both source operands x_1 and x_2 have uncertain values. According to the rule, the left-hand-side variable x maps to a vector storing the results of the binary operation on the corresponding elements in the two source vectors. If x_1 is uncertain and x_2 certain, rule `Binop-1st-Uncertain` applies and x has a vector value. Each vector element is computed from the corresponding vector element in x_1 and the value of x_2 in the regular store. If both source operands have certain values (rule `Binop-Both-Certain`), the resulting value is computed from the regular values and updated to the regular store, and the sample store of x is reset to undefined. Other rules are similarly derived and hence elided. Fig. 2 shows an example of such evaluation.

2.2 With Uncertain Control Flow

In real programs, there are often predicates operating on uncertain values. In such cases, it is uncertain which branch will be taken. To handle these cases, upon encountering an uncertain predicate, we split the coalescing into two subcoalescings: one with the sample runs following the true branch, called the *true coalescing*, and the other with those following the false branch, called the *false coalescing*. We evaluate them one after the other. At the merge point of the two branches, the two subcoalescings conjoin to the original coalescing. If uncertain predicates nest, a subcoalescing may further split into smaller subcoalescings[1]. We have to make sure the evaluations of the split subcoalescings of the same predicate are isolated, otherwise a definition in the true branch evaluation may reach a use in the following false branch evaluation of the same predicate, leading to errors.

Table 2 presents an important subset of evaluation rules.

[1]Such splittings are bounded because subcoalescings with only one trial cannot be split.

Table 1: Evaluation rules with certain control flow.

statement	condition	action	name
$x \leftarrow input()$	input() returns constant c	$\sigma(x) = c;\ \Gamma(x) = \bot$	Input-Certain
	input() returns distribution d	$\forall i \in Trial,\ \Gamma(x, i) = random(d)$	Input-Uncertain
$x \leftarrow x_1$ `binop` x_2	$\Gamma(x_1) \neq \bot \quad \Gamma(x_2) \neq \bot$	$\forall i \in Trial,\ \Gamma(x, i) = \Gamma(x_1, i)$ `binop` $\Gamma(x_2, i)$	Binop-Both-Uncertain
	$\Gamma(x_1) \neq \bot \quad \Gamma(x_2) = \bot$	$\forall i \in Trial,\ \Gamma(x, i) = \Gamma(x_1, i)$ `binop` $\sigma(x_2)$	Binop-1st-Uncertain
	$\Gamma(x_1) = \bot \quad \Gamma(x_2) = \bot$	$\sigma(x) = \sigma(x_1)$ `binop` $\sigma(x_2);\ \Gamma(x) = \bot$	Binop-Both-Certain
	...	...	...

Table 2: Evaluation rules with uncertain control flow. We use $[\![s]\!]$ to denote the action of evaluating a statement s. We use $\Gamma(x, \mu)$ as the shorthand for $\forall i \in \mu, \Gamma(x, i)$.

statement	condition	action	name
`if` (x) `then` s_1 `else` s_2	$\Gamma(x, \mu) =$ `true`	$[\![s_1]\!]$	If-UC-True
	$\exists i \in \mu,\ \Gamma(x, i) =$ `true` $\exists j \in \mu,\ \Gamma(x, j) =$ `false`	$\mu^T = \{i \in \mu,\ \Gamma(x, i) = \texttt{true}\};$ $\mu^F = \mu - \mu^T;$ `push`(μ); $\mu = \mu^T;\ [\![s_1]\!];$ $\mu = \mu^F;\ [\![s_2]\!];$ $\mu =$ `pop`$()$;	If-UC-Both
	...	...	...
$x \leftarrow input()$	input() returns constant c $\mu = Trial$	$\sigma(x) = c;\ \Gamma(x) = \bot$	Input-C
	input() returns constant c $\mu \subset Trial$	$\Gamma(x, \mu) = c$	Input-C-Mask
	input() returns distribution d	$\Gamma(x, \mu) = random(d)$	Input-UC
$x \leftarrow x_1$ `binop` x_2	$\Gamma(x_1) = \bot \quad \Gamma(x_2) = \bot \quad \mu = Trial$	$\sigma(x) = \sigma(x_1)$ `binop` $\sigma(x_2);\ \Gamma(x) = \bot$	Binop-Both-C
	$\Gamma(x_1, \mu) = \bot \quad \Gamma(x_2, \mu) = \bot$ $\mu \subset Trial$	$\Gamma(x, \mu) = \sigma(x_1)$ `binop` $\sigma(x_2)$	Binop-Both-C-Mask
	$\Gamma(x_1, \mu) \neq \bot \quad \Gamma(x_2, \mu) \neq \bot$	$\forall i \in \mu,\ \Gamma(x, i) = \Gamma(x_1, i)$ `binop` $\Gamma(x_2, i)$	Binop-Both-UC
	...	...	...
$x \leftarrow x_1$ `binop` c	$\Gamma(x_1, \mu) = \bot \quad \mu \subset Trial$	$\Gamma(x, \mu) = \sigma(x_1)$ `binop` c	Binop-C-Const-Mask
	...	...	...
`while` (x) s `endwhile`		$[\![$`if` (x) `then` s; `while` (x) s `endwhile` `else skip`$]\!]$	While

We enhance program state with a sample mask μ, whose definition is in Fig. 4. It identifies which trials are being evaluated along the current path. We only need to compute the values for these trials in the current path. Other trials require separate computation along different paths.

The first two rules describe the evaluation of an `if` statement predicating on uncertain values. The first rule `If-UC-True` specifies that even though the predicate operates on a vector value, if the vector elements are universally true, we only evaluate the true branch. Rule `If-UC-Both` specifies that if predicate x has both `true` and `false` values, we divide the current mask μ into two submasks μ^T and μ^F, each identifying those trials that follow the true and false branches, respectively. The true branch statement s_1 evaluates with μ^T and s_2 evaluates with μ^F. Note that although the stores are not explicitly specified in the rule, they update in evaluation order. In other words, the evaluation of s_2 operates on stores that have been updated in s_1's evaluation. The two submasks facilitate isolation, i.e. preventing the evaluation of s_2 from seeing values produced in s_1.

The next three rules describe input behavior. Rule `Input-C` specifies that we save the certain input to the regular store only when μ is the universal set $Trial$, meaning the current coalescing has not been split, i.e. the current path is certain. Rule `Input-C-Mask` specifies that although the input is certain, the assignment is performed on the sample store if μ is a subset. More intuitively, the rule dictates that if the evaluation is for a split coalescing, we cannot save the value to the regular store even though it is certain. Because it is certain only regarding the sample runs indicated by μ, it might have a different value in other sample runs.

The next three rules are for the assignment of a binary operation. Rule `Binop-Both-C` specifies that if both operands are certain and the sample mask is the universal set, we update the regular store of x and reset the sample store. In contrast, from `Binop-Both-C-Mask`, if the sample mask is only a subset, we update the sample store although both operands are certain. Note, we test if an operand is certain *regarding the current mask*, i.e. $\Gamma(x, \mu) = \bot$, instead of $\Gamma(x) = \bot$. The reason is that $\Gamma(x, Trial - \mu)$ might have been defined in the evaluation along the other split branch, which has no implications on whether x is certain along this branch. Rule `Binop-Both-UC` specifies that if both source operands are uncertain, we update the sample store, constrained by μ. Rule `Binop-C-Const-Mask` is similar to `Binop-Both-C-Mask`. Observe that any assignment along a split path ($\mu \subset Trial$) only updates the sample store but never the regular store, which ensures that uses of the regular store in rule `Binop-Both-C-Mask` in a split branch never see values defined in the other branch, but rather those before the split.

Rule `While` evaluates a `while` statement to an `if` statement so that the rules for the `if` statement can be used.

Example. Consider the program in Fig. 5. It computes a person's salary based on the rate and the hours she/he works. Table 3 presents a sample evaluation. The first column shows the control flow, the second and third columns show the regular store and the sample store, and the last column shows the rules applied. Here, we coalesce five sample runs. At the beginning, the sample mask is the universal set. The input at line 2 is uncertain, so we take 5 samples. At line 3, we apply rule `If-UC-Both`, and the mask is divided. The 1st, 3rd, and 5th sample runs evaluate in the

```
1  r  ← input();
2  h  ← input();
3  if ($p$ ← h $> 40$)
4      then r ← r + 2
5           s ← r × h
6           if ($q$ ← s $> 500$)
7               then s ← 500
8      else r ← r − 1
9           s ← r × h
10 endif;
11 output(s)
```

Figure 5: Example for uncertain control flow. The program computes salary s from rate r and work hours h. The rate is higher (line 4) if a person works for more than 40 hours. The salary has a cut-off value 500 (lines 6 and 7). If a person works for less than 40 hours, the rate is reduced (line 8).

true branch, and the 2nd and 4th runs in the false branch. At line 4, even though r holds a certain value, the sample store updates for the 1st, 3rd, and 5th runs. At line 6, the mask further divides into two submasks. Hence, the assignment at line 7 only updates the value for the 5th run to 500 as highlighted. After the true branch evaluates, the false branch evaluates. At line 8, rule `Binop-C-Const-Mask` applies so that the 2nd and 4th elements of r's vector update. Note that the values are computed from r's value in the regular store σ, and hence the definition to r at line 4 has no effect on line 8. At line 11, the submasks join. Observe, at the end the values in the sample store are identical to those acquired in the corresponding independent sample runs.

Safety. In the following, we present the safety claim of our technique. It is critical to show that the coalesced execution is equivalent to the N independent MC trials. Note that an independent MC trial is a regular program execution. Assume random sampling is deterministic for each uncertain input, i.e., given the same random seed and a distribution, the same N samples are generated. We further define the ith independent trial as the execution with each uncertain input taking the ith sample in the sequence of N. For instance, the first MC trial in Table 3 corresponds to the execution taking the inputs $r = 8$ and $h = 45$.

PROPERTY 1. *Execution coalescing is safe. In particular, at the end of program evaluation,*
(1) if a variable x has $\Gamma(x) \equiv \perp$, it must have the same value across the N independent MC trials and the value is identical to $\sigma(x)$;
(2) if $\Gamma(x) \neq \perp$, $\Gamma(x, i)$ must be identical to the value of x in the ith MC trial.

The property holds for our kernel language. The proof is elided. For real programs, it holds when we do not consider exceptions and interrupts. We leave it to our future work to extend the technique to handle such cases.

3. COST BENEFIT MODEL

The computational benefits of execution coalescing depend on a combination factors. In this section, we present a model for estimating the benefits.

Without coalescing, all MC trials are executed independently. Let's first consider the case that all trials execute along the same control flow. If there are N trials, with each trial executing I instructions, the total number of instructions $S(N)$ is given as follows.

$$S(N) = NI \tag{1}$$

In comparison, the cost of execution coalescing, in terms of executed instructions, is decided by the following factors.

- The number of trials that are coalesced. Let it be N.
- For each instruction, our technique must check whether it operates on uncertain variables. Let the slow down factor for such checking be K.
- We represent the percentage of instructions operating on uncertain values as T. Hence, in the coalesced execution, $I(1-T)$ instructions need to execute just once for the n trials. For each of the remaining IT instructions, the operation is performed on vectors, which is equivalent to executing the instruction N times.
- There is a constant bookkeeping overhead factor M when executing an instruction on a vector.

Combining all these components yields the expected cost presented in equation 2.

$$C(N, K, T, M) \;=\; KI + NMTI \;=\; (K + NMT)I \tag{2}$$

We further compute the benefit factor B in equation 3.

$$B = \frac{S(N)}{C(N, K, T, M)} = \frac{NI}{(K + NMT)I} = \frac{1}{K/N + MT} \tag{3}$$

Execution coalescing is beneficial when $B > 1$. A few observations can be made from equation 3.

- N needs to be larger than K and MT needs to be smaller than 1. In our implementation, K is often a constant in the range 4-10. Hence, we need to coalesce enough executions to make the technique beneficial.
- If $MT < 1$, B increases as N increases, reflecting that if more trials are coalesced, each executed instruction that is shared across trials has a greater reduction in overall work. It is bounded by $\frac{1}{MT}$.
- The benefit increases as the uncertain percentage T decreases. This reflects that as fewer instructions operate on uncertain values, more of the execution can be coalesced across trials. It is bounded by $\frac{N}{K}$.

The above analysis allows us to decide if execution coalescing is appropriate and tune the configuration of N (and T if possible). For instance, assume $M = 1.20$ and $T = 0.90$. Since $MT = 1.08 > 1$, it is guaranteed that coalescing provides no benefit. Moreover, assume $M = 1.20$, $T = 0.33$, and $K = 4.0$. If we want to achieve a speed up $B = 2$, we should use $N = 40$.

When uncertain control flow is considered, the benefit factor is computed as follows.

$$B = \frac{1 + U}{K/N + MT + KMU} \tag{4}$$

U is the ratio between the instructions that are unique in a trial over those common in all trials, describing the percentage of uncertain control flow. We assume U is a constant over all trials. The derivation of the inequality is omitted for space. From the equation, to satisfy $B > 1$, the following condition must hold.

$$U < \frac{1 - K/N - MT}{KM - 1} \tag{5}$$

Table 3: Sample execution for program in Fig. 5. $h \mapsto \{..., 52_5\}$ means h has value 52 in the 5th sample run.

control flow	σ	Γ	μ	rule
1 r $\leftarrow$ input ()	r $\mapsto$ 8		{1,2,3,4,5}	Input-C
2 h $\leftarrow$ input ()		h $\mapsto \{45_1, 30_2, 48_3, 25_4, 52_5\}$	{1,2,3,4,5}	Input-UC
3 `if` ($p \leftarrow$ h > 40)		p $\mapsto \{\mathtt{T}_1, \mathtt{F}_2, \mathtt{T}_3, \mathtt{F}_4, \mathtt{T}_5\}$	{1,2,3,4,5}	If-UC-Both
4 `then` r $\leftarrow$ r + 2		r $\mapsto \{10_1, 10_3, 10_5\}$	{1, 3, 5}	Binop-C-Const-Mask
5 s $\leftarrow$ r $\times$ h		s $\mapsto \{450_1, 480_3, 520_5\}$	{1, 3, 5}	Binop-Both-UC
6 `if` ($q \leftarrow$ s > 500)		q $\mapsto \{\mathtt{F}_1, \mathtt{F}_3, \mathtt{T}_5\}$	{1, 3, 5}	If-UC-Both
7 `then` s $\leftarrow$ 500		s $\mapsto \{450_1, 480_3, \mathbf{500}_5\}$	{ 5}	Assgn-Const-Mask
8 `else` r $\leftarrow$ r − 1		r $\mapsto \{10_1, \mathbf{7}_2, 10_3, \mathbf{7}_4, 10_5\}$	{2, 4}	Binop-C-Const-Mask
9 s $\leftarrow$ r $\times$ h		s $\mapsto \{450_1, \mathbf{210}_2, 480_3, \mathbf{175}_4, 500_5\}$	{2, 4}	Binop-Both-UC
11 output(s)			{1,2,3,4,5}	

Table 5: Example for uncertain array indices. Assume the sample mask is $\{1, 2\}$; $\sigma = \{A[0] \mapsto 3,\ A[1] \mapsto 9\}$; $\Gamma = \{i \mapsto \{0_1, 1_2\},\ j \mapsto \{1_1, 0_2\}\}$.

instruction	wrong	correct
1 $A[i] = 5$	$\sigma = \{A[0] \mapsto 5,$ $A[1] \mapsto 5\}$	$\Gamma = \{..., A[0] \mapsto \{5_1\},$ $A[1] \mapsto \{5_2\}\}$
2 $A[j] = 7$	$\sigma = \{A[0] \mapsto 7,$ $A[1] \mapsto 7\}$	$\Gamma = \{..., A[0] \mapsto \{5_1, 7_2\},$ $A[1] \mapsto \{7_1, 5_2\}\}$

z=__lib_foo(x,y)	1 for (each $i \in \mu$) 2 $\Gamma(z, i)$=__lib_foo($\Gamma(x, i)$, $\Gamma(y, i)$);

Figure 6: Calling a library function.

While we can configure N to reduce the effect of K/N in the condition, the technique must not be beneficial if $U > \frac{1-MT}{KM-1}$. Intuitively, our technique cannot be beneficial when the ratio between the divergent control flow and the common control flow is higher than a threshold.

4. HANDLING PRACTICAL ISSUES

In this section, we discuss how to support more complex features that are not modeled by our language.

Uncertain Array Indices. Extra effort is needed when uncertain values are used as array indices. We cannot simply perform the array operation on the regular store with the addresses specified by the vector value of the index, even though that seems straightforward. Instead, the operation must be performed on the sample store. Specifically, a vector v needs to be created for each address specified by the uncertain index. Only the ith element of the vector created for the ith address gets updated. The reason is that the ith address should only be used in the evaluation of the ith trial.

Consider the example in Table 5. Indices i and j are uncertain. Both can take values 0 and 1 but take different values within the same trial. The second column shows that if we follow the naïve approach, after evaluating line 1, $A[0]$ and $A[1]$ map to 5 in σ. After line 2, both map to 7. The state after line 2, however, is wrong and does not correspond to the coalescing of the two trials. Proper evaluation should yield that at line 1, both $A[0]$ and $A[1]$ map to vector values, and only the 1st trial in $A[0]$ and the 2nd trial in $A[1]$ are set to 5. Similarly, after line 2, only the 2nd trial in $A[0]$ and the 1st trial in $A[1]$ are set to 7. The state correctly represents the coalescing of the two independent trials.

Table 4 presents some of the array access rules. We currently do not support uncertain base addresses (A has to be certain). Uncertain pointers are supported in the same way because they are equivalent to uncertain array indices.

Functions. Handling user defined functions is direct. For library functions taking uncertain arguments, since we do not have their source code, we cannot transform the functions to operate on uncertain values. Our solution is to wrap the function call in a loop that iterates through each trial in the sample mask. Each iteration calls the function with the regular values in a trial (extracted from the sample store). The results are written to the corresponding elements in the result vector. Fig. 6 shows an example.

5. OPTIMIZATION BY REMOVING UNNECESSARY UNCERTAINTY

We observe that during coalesced execution, there are variables that are considered uncertain, having vector values, but the values in the vector are identical or have only negligible differences. We can reduce the vector to a single value so that the subsequent computation with these variables can be re-coalesced. Such cases can be caused by:

- An uncertain value going through an operator representing a many-to-one mapping. Multiple inputs to the operator can lead to the same output. Sample scenarios include: multiply by 0 (e.g. $y \leftarrow x \times 0$), modulo operation (e.g. $y \leftarrow x\%10$), bit operations (e.g. $y \leftarrow x\ \&\ 0xfff$), and comparisons (e.g. $p \leftarrow (x > 10)$). Note that our evaluation rules are suboptimal for these cases because y is uncertain as long as x is uncertain, disregarding the values in the vector of y.
- Floating point round-off, overflow, and underflow can also lead to identical values in vectors. For instance, consider the following statement.

  $$y \leftarrow x + 1.0e9$$

  And assume $\Gamma(x) = \{0.002_1, 0.005_2, 0.007_3\}$. Since the floating point representation can only hold a fixed number of the most significant digits (suppose it can hold 7 digits), the contribution from x is then rounded off, leading to $\Gamma(y) = \{1.0e9_1, 1.0e9_2, 1.0e9_3\}$.
- Recall that when we encounter an uncertain predicate, we first evaluate a subset of trials along the true branch and then the remaining trials along the false branch. Any assignments inside these branches are performed on the sample store. It is possible that a variable is defined with the same value in the two branches.

There are also cases where floating point values in a vector are highly similar although they are not completely identical. We hence develop a general solution to leverage the above observations. Given a *significance threshold* k, if the differences between the first value in a vector $\Gamma(x)$ and any other values in the vector are less than k, we re-coalesce the vector to a single value. The threshold based recoalescing

Table 4: Evaluation rules for uncertain array indices.

statement	condition	action	name
$A[x]=c$	$\Gamma(A)=\perp \quad \Gamma(x,\mu)\neq\perp$	$\forall i\in\mu,\ \Gamma(A[\Gamma(x,i)],i)=c$	Arr-Const-Wr-UC
$y=A[x]$	$\Gamma(A)=\perp \quad \Gamma(x,\mu)\neq\perp \quad \forall i\in\mu,\ \Gamma(A[\Gamma(x,i)],i)\neq\perp$	$\forall i\in\mu,\ \Gamma(y,i)=\Gamma(A[\Gamma(x,i)],i)$	Arr-Rd-UC

provides a means to trade a *configurable degree* of precision for increased efficiency by decreasing T in equation 3 (the uncertain ratio). Note that when $k = 0$, we only merge identical values, no precision is lost.

6. EMPIRICAL EVALUATION

Our system is built in `gcc`, based on the GIMPLE IR. Since our technique operates on vectors, we leverage SIMD (Single Instruction, Multiple Data) instructions for better performance. In particular, we use packed floating point instructions, such as addition, subtraction or computing square roots, from the SSE2 extension of SIMD instructions. Note that the benefit of SSE2 instructions is limited by the width of the SSE registers, which is 128-bit for our machine. In other words, we can only pack two double precision floating point values at a time.

Our experiments are performed on an Intel quad core Xeon 1.86GHZ machine with 4GB RAM. We use SPECFP-2000 as the benchmark set, which includes both C and Fortran programs. We excluded 2 programs. In particular, `189.lucas` is a program that identifies prime numbers and hence uncertainty analysis is not applicable. `177.mesa` is omitted because we have not supported direct assignments of aggregate types such as assignments of struct. We have total 12 programs (3 C and 9 Fortran).

Our first experiment evaluates the benefits of execution coalescing together with its space overhead but without the optimization removing unnecessary uncertainty. For each program, we vary the number of coalesced trials (factor N in eqn 3 in Section 3) among 1, 10, 20, and 30. We also vary the percentage of input marked as uncertain, which will affect the percentage of instructions operating on uncertain values (T in eqn 3), among 1 input, 1%, 5%, 10%, 15%, and 20% of the input. For each uncertain input, we select samples from 50%-100% of the original value following a uniform distribution. The reason that we select samples smaller than the original is that larger samples may fall out of the legal range.

Detailed results are presented in Table 6. Details for 30 samples are omitted for space. The *native* columns show the original time. The $T\%$ columns show the percentage of statements operating on uncertain values. The K columns show the normalized overhead when only one sample is taken, which corresponds to running the program on the original input (w/o uncertainty) with the overhead of checking and propagating uncertainty (the K factor in eqn 3). The B columns show the benefit factor (i.e. the speedup) with the subscripts representing the number of samples. For `172.mgrid` and `301.apsi`, the columns under *1% uncertain input* present data for marking all inputs as uncertain due to their small number of inputs. We also summarize the results in Fig. 7, showing the variation in the benefit over the number of samples, taking the average over the percentage of uncertain input. Note that we put in the data for 30 samples. Fig. 8 further shows the variation in benefit over uncertain input, taking the average over the number of samples. Fig. 9 summarizes the normalized space overhead (details omitted for space). From the table and the figures, we make the following observations.

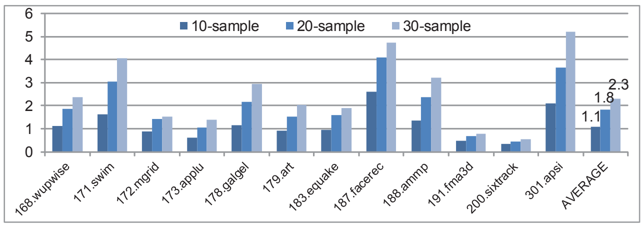

Figure 7: Change in speedup over the number of samples.

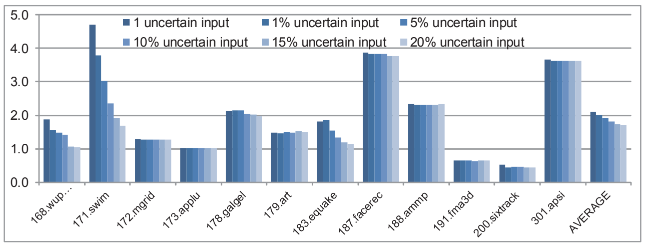

Figure 8: Change in speedup over uncertain input.

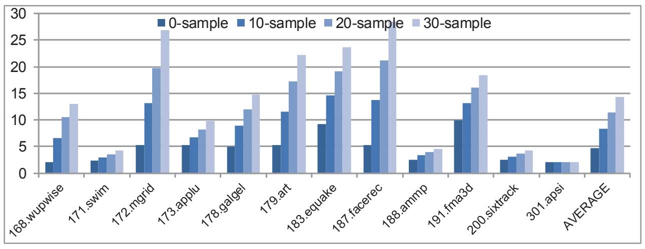

Figure 9: The normalized space overhead.

- Coalescing can speed up MC simulations by an average factor ranging from 0.6 to 5.2 with an average of 2.3 when 30 samples are coalesced. We expect the number to be high if memory is large enough to fit more samples. The average overhead of checking and propagating uncertainty (K) ranges from 2.1 to 17.0 with an average 7.9. Since the nature of the uncertainty propagation is similar to dynamic information flow tracking and the state of the art [23] reports an average 2 times speedup if their aggressive optimizations are applied, we speculate our overhead can be similarly reduced in the future. The large K implies that we have to coalesce sufficient trials to make it beneficial (see eqn 3).
- The speedup increases with the number of coalesced samples. It decreases as more inputs are marked uncertain. The decrease is not that substantial for some programs. When T is large, we hardly benefit with 10 samples.
- The space overhead ties closely with the number of coalesced samples. It could be high when coalescing a large number of samples. We use the standard shadow

Table 6: Performance with different configurations.

program	native	# of input	1 uncertain input				1% uncertain input				5% uncertain input			
			$T\%$	K	B_{10}	B_{20}	$T\%$	K	B_{10}	B_{20}	$T\%$	K	B_{10}	B_{20}
168.wupwise	4.39	4.6E6	29.27%	4.98	1.24	2.00	30.31%	5.14	1.23	1.90	33.02%	5.24	1.14	1.80
171.swim	0.40	2.6E5	0.64%	4.16	2.36	4.69	3.18%	4.38	2.00	3.94	6.72%	4.78	1.73	3.20
172.mgrid	34.09	10	98.84%	5.80	0.88	1.43	98.84%	5.80	0.88	1.43	N/A			
173.applu	0.41	65	66.42%	10.10	0.63	1.04	66.42%	10.10	0.63	1.04	67.87%	10.06	0.63	1.05
178.galgel	1.61	5.2E3	54.01%	8.06	1.17	2.21	54.05%	8.34	1.17	2.18	54.08%	7.87	1.17	2.22
179.art(C)	27.64	1.0E4	25.42%	7.78	0.89	1.48	25.42%	6.99	0.89	1.48	25.43%	6.56	0.92	1.57
183.equake(C)	0.48	7.3E3	20.88%	6.26	1.18	1.94	21.90%	6.35	1.17	1.93	26.94%	6.86	1.00	1.65
187.facerec	9.64	6.6E4	16.76%	2.08	2.66	4.15	16.95%	2.08	2.63	4.09	17.01%	2.08	2.63	4.07
188.ammp(C)	4.49	9.6E3	9.59%	5.94	1.38	2.39	9.59%	5.88	1.37	2.38	9.59%	5.91	1.36	2.42
191.fma3d	5.08	544	87.82%	13.57	0.47	0.67	87.95%	13.41	0.47	0.68	88.65%	13.52	0.46	0.67
200.sixtrack	7.19	7.9E4	65.17%	15.12	0.41	0.52	78.40%	16.84	0.35	0.44	78.40%	16.82	0.35	0.45
301.apsi	4.41	9	8.35%	4.15	2.11	3.67	8.35%	4.16	2.09	3.65	N/A			

program	10% uncertain input				15% uncertain input				20% uncertain input			
	$T\%$	K	B_{10}	B_{20}	$T\%$	K	B_{10}	B_{20}	$T\%$	K	B_{10}	B_{20}
168.wupwise	34.65%	5.44	1.10	1.74	36.84%	5.53	1.06	N/A	37.92%	5.59	1.04	N/A
171.swim	11.34%	5.39	1.44	2.53	17.38%	5.91	1.18	2.06	20.38%	6.36	1.09	1.85
172.mgrid	N/A				N/A				N/A			
173.applu	67.89%	10.20	0.63	1.05	67.91%	10.20	0.63	1.06	67.94%	10.16	0.63	1.07
178.galgel	58.74%	8.53	1.11	2.15	58.78%	8.31	1.12	2.13	58.90%	8.28	1.12	2.05
179.art	25.43%	6.93	0.90	1.57	25.44%	6.77	0.97	1.55	25.44%	6.77	0.91	1.56
183.equake	31.08%	7.35	0.84	1.41	33.50%	7.71	0.80	1.32	34.76%	7.88	0.79	1.27
187.facerec	17.03%	2.07	2.61	4.11	17.04%	2.07	2.55	4.09	17.05%	2.10	2.61	4.04
188.ammp	9.59%	5.89	1.36	2.39	9.59%	5.86	1.36	2.37	9.59%	5.88	1.38	2.38
191.fma3d	88.70%	13.51	0.47	0.66	88.82%	13.54	0.47	0.67	88.88%	13.80	0.47	0.67
200.sixtrack	78.40%	16.95	0.35	0.45	78.40%	16.96	0.35	0.45	78.40%	16.97	0.34	0.44
301.apsi	N/A				N/A				N/A			

memory allocation strategy [20] to allocate shadow space for a page when any address in that page is used by the original program. We expect a more sophisticated strategy would reduce the overhead.

The second experiment shows the effect of optimizing away the unnecessary uncertainty. Here, we randomly select one configuration: 10 samples and 1 single uncertain input. We vary the re-coalescing threshold k, i.e. if the variations of all elements in a vector are smaller than k, we re-coalesce the vector back to one value. Fig. 10 presents the speedup results. Table 7 presents the resulting output errors due to re-coalescing. We do not collect output errors for the programs that do not benefit from the optimization. Observe, the optimization is not always applicable, depending on program semantics (see Section 5). In such cases, we have to pay the overhead for testing variations, which explains the slight degradation for `183.equake` in Fig. 10. For some programs, it substantially improves the speedup factor, e.g. from 1.25 to 3.4 for `168.wupwise`, with little lost in precision. Note that a larger k leads to a better speedup. The precision lost also slightly increases. Our experience with the few other randomly selected configurations also shows similar results.

The third experiment observes the effectiveness of MC simulation. Here, we vary the uncertain inputs and observe the output variations. We randomly select `183.equake`. The program simulates the propagation of elastic waves in large basins. An unstructured mesh consists of nodes and linear tetrahedral elements is used to model the earthquake area. The program computes the displacements for each individual node. We vary the altitude of a node from 100% to 80% (of its original value) [2] and observe the outputs. Fig. 11 presents the outputs with the most substantial displacement

[2]We used the provided test input and varied node 977, which is randomly selected from those in the source of the quake.

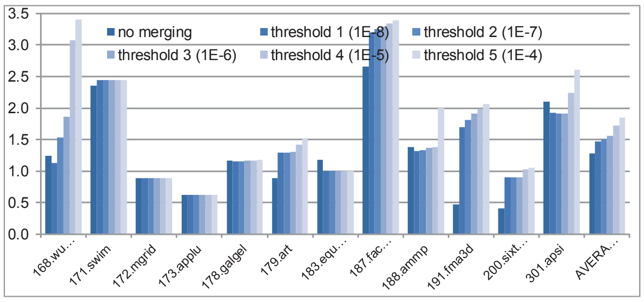

Figure 10: Speedup by re-coalescing with different thresholds.

values and their variations over the input changes. Each curve represents one output. We can observe that many of these outputs change irregularly and substantially. Traditional uncertainty analysis based on monotonicity, linearity, and continuity would fail in this case. Furthermore, we observe that a few percent change of the inputs may lead to substantial output changes. The variations could be a few times larger than the origianl values.

Case Study. In the following, we further study the case and explain the substantial and irregular output variations by connecting them to uncertain control flow and uncertain array indices.

For each node i, its position is defined by a tuple of (latitude, longitude, altitude) stored in ($c[i][0]$, $c[i][1]$, $c[i][2]$), where $c[i][2]$ is the altitude. The altitude of node 977 is -11.2. We look at the 10 samples from 95% to 77% of the original altitude value to explain the impact.

Initially, we have a vector for the altitude of node 977, $\sigma = \{c[977][2]\mapsto -11.2\}$; $\Gamma = \{\ c[977][2]\mapsto \{-10.64_1, -10.416_2, ..., -8.624_{10}\}\ \}$.

Table 7: The average and maximum output error after merging, for 10 samples and 1 uncertain input.

program	threshold 1 (1E-8)	threshold 2 (1E-7)	threshold 3 (1E-6)	threshold 4 (1E-5)	threshold 5 (1E-4)
	avg/max	avg/max	avg/max	avg/max	avg/max
168.wupwise	9.6E-10/1.6E-08	7.7E-09/5.5E-08	8.7E-09/1.6E-07	8.7E-09/9.1E-06	2.1E-08/9.6E-05
171.swim	3.0E-10/1.8E-05	3.0E-10/1.8E-05	3.0E-10/1.8E-05	2.8E-10/1.3E-05	2.8E-10/1.3E-05
179.art	6.4E-16/3.1E-13	6.4E-16/3.1E-13	6.4E-16/3.1E-13	6.4E-16/3.1E-13	6.4E-16/3.1E-13
187.facerec	5.4E-10/1.6E-09	3.3E-09/1.3E-08	3.3E-09/1.3E-08	3.3E-09/1.3E-08	3.3E-09/1.3E-08
188.ammp	1.3E-14/4.3E-12	3.1E-13/3.6E-12	3.1E-13/3.6E-12	3.1E-13/3.6E-12	4.7E-03/9.5E-02
191.fma3d	3.0E-09/2.9E-11	3.0E-09/2.9E-11	3.0E-09/2.9E-11	8.4E-09/2.0E-10	8.4E-09/2.0E-10
200.sixtrack	1.6E-09/4.6E-09	2.2E-09/9.3E-09	1.4E-09/6.2E-09	1.4E-09/6.2E-09	1.4E-09/6.2E-09
301.apsi	8.3E-06/1.0E-02	1.7E-04/7.7E-02	2.0E-03/9.0E-01	3.8E-05/7.9E-03	3.8E-05/8.0E-03

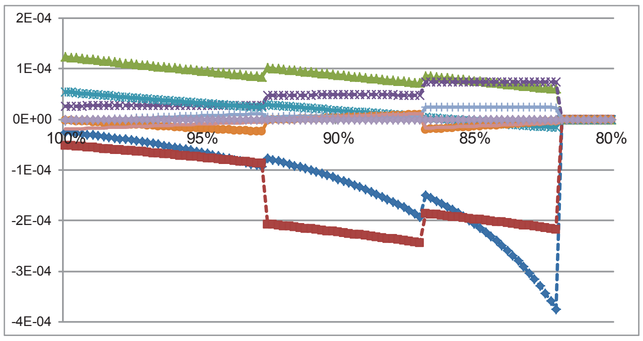

Figure 11: Output variations for `183.equake`.

```
     /* Search for the node closest to the point source */
244  bigdist1 = 1000000.0;
247  for (i = 0; i < ARCHnodes; i++) {
248    c0[0] = c[i][0];
249    c0[1] = c[i][1];
250    c0[2] = c[i][2];
251    d1 = distance(c0, Src.xyz);
254    if (d1 < bigdist1) {
255      bigdist1 = d1;
256      Src.sourcenode = i;
257    }
264  }
```

One step of the computation selects the closest node to the given earthquake source point in `Src.xyz` in the above code. All the nodes are traversed with a loop between line 247 and 264 (shown above). In the 979th iteration, at line 250, $c0[2]$ receives the vector $\{-10.64_1, -10.416_2, ..., -8.624\}$. Returning from the function `distance()`, `d1` at line 251 gets the vector of values $\{0.675_1$, 0.998_2, 1.422_3, 1.946_4, ..., 6.072_9, $7.198_{10}\}$. The variable `bigdist1` holds the value 4.657. Therefore, we encounter an uncertain predicate (`d1 < bigdist1`) at line 254. The sample mask μ is divided into two submasks $\mu^T = \{1, 2, ..., 7\}$ and $\mu^F = \{8, 9, 10\}$. According to the evaluation rules in Table 2, we compute the values of `bigdist1` and `Src.sourcenode` in both branches with μ^T and μ^F respectively. After line 257, the values in the sample stores are joined together, i.e. `bigdist1` $\mapsto \{0.675_1, 0.998_2, 1.422_3, 1.946_4, ..., 4.657_8, 4.657_9, 4.657_{10}\}$, `Src.sourcenode` $\mapsto \{977_1, 977_2, 977_3, ..., 973_8, 973_9, 973_{10}\}$.

After the loop, the node closest to the source point is selected and stored in variable `Src.sourcenode`. Now we have `Src.sourcenode` $\mapsto \{977_1, 977_2, 977_3, ..., 973_8, 973_9, 973_{10}\}$. Thus, for the first 7 trials, the computed source node is node 977 while for the last 3 trials it is node 973.

Another code snippet shows that uncertain array indices add to the irregularity. Variable `cor` holds a certain value at line 2. When it equals 977 or 973, the coalescing splits after evaluating the branch at line 293. For instance, when `cor` equals 973 at line 293, we get a submask of $\{8, 9, 10\}$, so only the 8th, 9th and 10th elements of vertices[j][k]'s vector get updated, according to the evaluation rules in Table 4. Similarly, the 1st to 7th elements of the vector get updated when `cor` equals 977.

```
289    for (i = 0; i < num_elems; i++) {
293      if (cor == Src.sourcenode) {
           ...
298            vertices[j][k] = c[cor][k];
306      }
307    }
```

In later phases, the sub-coalescings are further divided because of other predicates. This implies that we observe discontinuity or different trends between outputs belonging to different sub-coalescings because they go through different computation. In contrast, outputs belonging to the same sub-coalescing often have continuity or even monotonicity. These explain the curves in Fig. 11, where monotonic segments are observed but irregularity exists across segments.

7. RELATED WORK

Parallelized Monte Carlo. Parallelization has improved the efficiency of MC techniques in the context of specific applications [2, 4]. In contrast, our approach is fully automated and works on already developed programs. Additionally, we don't rely on parallelization to improve running time, instead we reduce the common, redundant work across Monte Carlo trials. Hence, it is orthogonal to parallelization.

Partial Evaluation & Memoization. Partial evaluation generates a new version of a program where additional assumptions about runtime behavior allow more aggressive optimization [15]. Memoization caches the results of a function for given input such that the results can be reused instead of reevaluating the function. As noted in section 1, these are useful in program optimization, but they cannot realistically handle the combining of multiple disparate executions.

Delta Execution. Delta execution [9] eliminates redundancy across state explorations in model checking. Each type in the program is wrapped in a container type that maintains a vector of the original type. At runtime, operations translate to operations on these container types. The technique closely ties into model checking. The savings largely come from places specific to model checking, such as state comparison to avoid repeated state exploration. They also handle control flow splitting differently. Leveraging state management in the model checker, they allow split branches to work on isolated stores. Branches only merge at the end of a method. In contrast, we cannot afford checkpointing and restoration; thus our split branches operate on the same stores and achieve isolation through careful evaluation rules. We also aggressively merge split branches at the join point.

A variant of delta execution also relates [27]. While our approach enables execution of one program on multiple inputs at once, the variant enables execution of multiple programs on one input at once, efficiently finding differences.

Uncertainty Analysis. Uncertainty and sensitivity analyses compose one client field of Monte Carlo techniques. Other efforts provide a means of partially automating these analyses efficiently. These approaches include model checking and theorem proving [10, 6], automated differentiation [3] and controlled perturbation [12]. They have difficulty handling data structures with heterogeneous data (certain and uncertain) or multiple uncertain variables, which is noted as one of the reasons to use sampling based approaches [11].

8. CONCLUSIONS

We propose a technique that can coalesce multiple Monte Carlo sample executions into one. We leverage the observation that these sample runs often share a lot of common execution and hence coalescing avoids repeating the common execution. Coalescing is achieved by allowing the program to operate on a single value if it is the same across all the coalesced runs, and on a vector otherwise. Our technique executes both the true and false branches of a predicate in an isolated fashion if its vector values can be both true and false. Pointers, array indices, and function calls on vector values are also handled safely such that the coalesced run produces the same results as independent runs. Our evaluation shows that we can speed up the runtime of 30 sample runs by an average factor of 2.3 without precision lost or by up to 3.4 with negligible precision lost.

9. ACKNOWLEDGMENTS

This research is supported, in part, by the National Science Foundation (NSF) under grants 0845870 and 0916874. Any opinions, findings, conclusions, or recommendations in this paper are those of the authors and do not necessarily reflect the views of NSF.

10. REFERENCES

[1] U. Acar, G. Blelloch, and R. Harper. Selective memoization. In *POPL*, 2003.

[2] V. Alexandrov, I. Dimov, A. Karaivanova, and C. Tan. Parallel monte carlo algorithms for information retrieval. *Math. Comput. Simul.*, 62(3-6), 2003.

[3] J. Barhen and D. Reister. Uncertainty analysis based on sensitivities generated using automatic differentiation. In *ICCSA*, 2003.

[4] I. Beichl, Y. Teng, and J. Blue. Parallel monte carlo simulation of mbe growth. In *IPPS*, 1995.

[5] M. Carbin and M. Rinard. Automatically identifying critical input regions and code in applications. In *ISSTA*, 2010.

[6] S. Chaudhuri, S. Gulwani, and R. Lublinerman. Continuity analysis of programs. In *POPL*, 2010.

[7] J. Clause, W. Li, and A. Orso. Dytan: a generic dynamic taint analysis framework. In *ISSTA*, 2007.

[8] U. Consortium. The universal protein resource (uniprot) in 2010. *Nucleic Acids Res*, 38, Jan 2010.

[9] M. d'Amorim, S. Lauterburg, and D. Marinov. Delta execution for efficient state-space exploration of object-oriented programs. In *ISSTA*, 2007.

[10] M. Heimdahl, Y. Choi, and M. Whalen. Deviation analysis through model checking. In *ASE*, 2002.

[11] J. Helton, J. Johnson, C. Sallaberry, and C. Storlie. Survey of sampling-based methods for uncertainty and sensitivity analysis. *Reliability Eng. & Sys. Safety*, 91(10-11), 2006.

[12] Y. Ho, M. Eyler, and T. Chien. A gradient technique for general buffer storage design in a production line. *Int. Jour. of Prod. Res.*, 17(6), 1979.

[13] R. Jampani, F. Xu, M. Wu, L. Perez, C. Jermaine, and P. Haas. Mcdb: a monte carlo approach to managing uncertain data. In *SIGMOD*, 2008.

[14] N. Jones, C. Gomard, and P. Sestoft. *Partial evaluation and automatic program generation.* Prentice-Hall, Inc., 1993.

[15] N. Jones, P. Sestoft, and H. Sondergaard. An experiment in partial evaluation: the generation of a compiler generator. In *RTA*, 1985.

[16] P. Karp. What we do not know about sequence analysis and sequence databases. *Bioinformatics*, 14(9), 1998.

[17] G. Keller, H. Chaffey-Millar, M. Chakravarty, D. Stewart, and C. Barner-Kowollik. Specialising simulator generators for high-performance monte-carlo methods. In *PADL*, 2008.

[18] Y. Liu and T. Teitelbaum. Caching intermediate results for program improvement. In *PEPM*, 1995.

[19] M. Morgan and M. Henrion. *Uncertainty: A Guide to Dealing with Uncertainty in Quantitative Risk and Policy Analysis.* Cambridge University Press, 1992.

[20] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI*, 2007.

[21] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *NDSS*, 2005.

[22] W. Pugh and T. Teitelbaum. Incremental computation via function caching. In *POPL*, 1989.

[23] O. Ruwase, S. Chen, P. Gibbons, and T. Mowry. Decoupled lifeguards: enabling path optimizations for dynamic correctness checking tools. In *PLDI*, 2010.

[24] A. Shankar, S. Sastry, R. Bodík, and J. Smith. Runtime specialization with optimistic heap analysis. In *OOPSLA*, 2005.

[25] S. Singh, C. Mayfield, R. Shah, S. Prabhakar, S. Hambrusch, J. Neville, and R. Cheng. Database support for probabilistic attributes and tuples. In *ICDE*, 2008.

[26] E. Tang, E. Barr, X. Li, and Z. Su. Perturbing numerical calculations for statistical analysis of floating-point program (in)stability. In *ISSTA*, 2010.

[27] J. Tucek, W. Xiong, and Y. Zhou. Efficient online validation with delta execution. In *ASPLOS*, 2009.

[28] B. Worley. Deterministic uncertainty analysis. Technical report, Oak Ridge National Lab. TN, 1987.

[29] M. Zhang, X. Zhang, X. Zhang, and S. Prabhakar. Tracing lineage beyond relational operators. In *VLDB*, 2007.