Verifying Quantitative Reliability of Programs That Execute on Unreliable Hardware (Appendix)

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

A. Additional Semantics Definitions

This section contains additional semantics definitions elided from the presentation in Section 3 in the paper.

A.1 Semantics of Arrays

Figure 11 presents the dynamic semantics of array operations.

Declarations. An array declaration allocates a new array in the heap. The boundaries of an array are given by a sequence of expressions, each of which can evaluate unreliably [E-ARRAY-DECL-R]. Given these boundaries, a new array is allocated using the function *new*, which returns the base address of an array that has been freshly allocated in the memory region m [E-ARRAY-DECL]. The function *new* executes reliably. To guard against ill-defined behavior, the given semantics also reliably checks that the length of each dimension is non-negative.

Loads. Executing an array load entails multiple steps. In the first step, the program reduces the expressions for each index in left-to-right order [E-ARRAY-LOAD-IDX]. Note that reduction of the index expressions may encounter faults, producing incorrect indices. While our dynamic semantics incorporates bounds checks to prevent incorrect indices from yielding ill-defined, out-of-bounds behaviors, we have left the choice of index expression reliability to the discretion of the developer.

Given the array reference and the reduced indices of each dimension, the program then checks to see if each index is within bounds of the allocated dimensions of the array and also calculates the offset of the element to be accessed [E-ARRAY-LOAD-C, E-ARRAY-LOAD-F]. In the final step, the array load proceeds by attempting to fetch the corresponding value from the given memory region. With probability $\psi(rd(m))$, this step executes correctly and returns the value from memory [E-ARRAY-LOAD-C]. With probability $1 - \psi(rd(m))$, the memory read fails, producing an alternative value n_f with probability $P(n_f)$ [E-ARRAY-LOAD-F].

Stores. The semantics of array stores are similar to that for loads except with the reliability of writes to the array's memory region ($\psi(wr(m))$) substituted for the reliability of reads. We have elided a full presentation of the semantics of array stores for brevity.

We note that the store operation may fail in two ways: 1) the index computation produces a wrong index (the rule is similar to [E-ARRAY-LOAD-IDX]) or 2) the write operation may fail, with probability $1 - \psi(wr(m))$. If the index computation fails and the computed index is outside of the loop bounds, then the computation skips the write operation. If the index computation fails and the computed index is within the bound, then the write will modify another location within the array. If the write operation fails, the semantics stores an alternative value n_f with probability $P(n_f)$.

A.2 Semantics and Analysis of Functions

A.2.1 Dynamic Semantics

Figure 12 presents the dynamic semantics of function calls and returns. A function call and return sequence executes via the following procedure:

Call Argument Evaluation. A function call first evaluates its arguments in left-to-right order [E-CALLX-ARGS]. Note that evaluation of the arguments may encounter faults. A parameter value v is either value that a numerical expression evaluates to, which is denoted as v^{num} or a memory location of the array, which is denoted as v^{ref}.

Call Body Unfolding. After the arguments to a call have been fully evaluated, control then transfers to the body of function [E-CALLX-UNFOLD]. The rule transfers control by fetching the code for the body of the function via the utility function *code*(f).

The rule also creates a new state σ' and pushes the old state σ onto the program stack δ. The rule initializes the new state with the appropriate values for its parameters by prepending a sequence of declarations s_{int_init} to allocate and initialize the numerical parameters of the function. The function *array_params* returns the set of names and position of the formal array parameters of the function f. The function *int_params* returns the set of names, memory locations, and positions of integer formal parameters of the function f.

Note that this rule executes correctly with probability 1; this implies that both control transfers and manipulations of the program stack are performed fully reliably. While

E-Array-Decl-R

$$\dfrac{\langle e_i, \sigma\rangle \xrightarrow{\theta,p}_{\psi} e'_i}{\langle \texttt{int}\ a\texttt{[}n_1,\dots,e_i,\dots,e_k\texttt{]}\ \texttt{in}\ m, \langle \sigma :: \delta, h\rangle\rangle \xrightarrow{\theta,p}_{\psi} \langle \texttt{int}\ a\texttt{[}n_1,\dots,e'_i,\dots,e_k\texttt{]}\ \texttt{in}\ m, \langle \sigma :: \delta, h\rangle\rangle}$$

E-Array-Decl

$$\dfrac{\forall i.\, 0 < n_i \qquad \langle n_b, h'\rangle = \mathit{new}(h, m, \langle n_1,\dots,n_k\rangle) \qquad \sigma' = \sigma[a \mapsto \langle n_b, \langle n_1,\dots,n_k\rangle, m\rangle]}{\langle \texttt{int}\ a\texttt{[}n_1,\dots,n_k\texttt{]}\ \texttt{in}\ m, \langle \sigma :: \delta, h\rangle\rangle \xrightarrow{\mathrm{C},1}_{\psi} \langle \texttt{skip}, \langle \sigma' :: \delta, h'\rangle\rangle}$$

E-Array-Load-Idx

$$\dfrac{\langle e_i, \sigma\rangle \xrightarrow{\theta,p}_{\psi} e'_i}{\langle x \texttt{ = } a\texttt{[}n_1,\dots,e_i,\dots,e_k\texttt{]}, \langle \sigma :: \delta, h\rangle\rangle \xrightarrow{\theta,p}_{\psi} \langle x \texttt{ = } a\texttt{[}n_1,\dots,e'_i,\dots,e_k\texttt{]}, \langle \sigma :: \delta, h\rangle\rangle}$$

E-Array-Load-C

$$\dfrac{\sigma(a) = \langle n_b, \langle l_1,\dots,l_k\rangle, m\rangle \qquad n_o = l_k + \sum_{i=0}^{k-1} n_i \cdot l_i \qquad n = h(n_b + n_o)}{\langle x \texttt{ = } a\texttt{[}n_1,\dots,n_k\texttt{]}, \langle \sigma :: \delta, h\rangle\rangle \xrightarrow{\mathrm{C},\, \psi(\mathit{rd}(m))}_{\psi} \langle x \texttt{ = } n, \langle \sigma :: \delta, h\rangle\rangle}$$

E-Array-Load-F

$$\dfrac{\sigma(a) = \langle n_b, \langle l_1,\dots,l_k\rangle, m\rangle \qquad n_o = l_k + \sum_{i=0}^{k-1} n_i \cdot l_i \qquad p = (1 - \psi(\mathit{rd}(m))) * P(n_f \mid \mathit{rd}(m))}{\langle x \texttt{ = } a\texttt{[}n_1,\dots,n_k\texttt{]}, \langle \sigma :: \delta, h\rangle\rangle \xrightarrow{\langle \mathrm{F}, n_f\rangle,\, p}_{\psi} \langle x \texttt{ = } n_f, \langle \sigma :: \delta, h\rangle\rangle}$$

Figure 11: Dynamic Semantics of Arrays

E-Callx-Expr-Arg

$$\dfrac{\langle e_i, \sigma\rangle \xrightarrow{\theta,p}_{\psi} e'_i}{\langle x \texttt{ = } f\texttt{(}v_1,\dots,e_i,\dots,e_k\texttt{)}, \langle \sigma :: \delta, h\rangle\rangle \xrightarrow{\theta,p}_{\psi} \langle x \texttt{ = } f\texttt{(}v_1,\dots,e'_i,\dots,e_k\texttt{)}, \langle \sigma :: \delta, h\rangle\rangle}$$

E-Callx-Int-Arg

$$\dfrac{\langle e_i, \sigma\rangle \xrightarrow{\theta,p}_{\psi} n \qquad v_i^{\mathit{num}} = n}{\langle x \texttt{ = } f\texttt{(}v_1,\dots,e_i,\dots,e_k\texttt{)}, \langle \sigma :: \delta, h\rangle\rangle \xrightarrow{\theta,p}_{\psi} \langle x \texttt{ = } f\texttt{(}v_1,\dots,v_i^{\mathit{num}} \dots,e_k\texttt{)}, \langle \sigma :: \delta, h\rangle\rangle}$$

E-Callx-Arr-Arg

$$\dfrac{v_i^{\mathit{ref}} = \sigma(a_i)}{\langle x \texttt{ = } f\texttt{(}v_1,\dots,a_i,\dots,e_k\texttt{)}, \langle \sigma :: \delta, h\rangle\rangle \xrightarrow{\mathrm{C},1}_{\psi} \langle x \texttt{ = } f\texttt{(}v_1,\dots,v_i^{\mathit{ref}},\dots,e_k\texttt{)}, \langle \sigma :: \delta, h\rangle\rangle}$$

E-Callx-Unfold

$$\dfrac{\begin{array}{c}\forall (a_i, i) \in \mathit{array\_params}(f)\ .\ \sigma'(a_i) = v_i^{\mathit{ref}} \\ s_{\mathit{int\_init}} \equiv \texttt{int}\ x_{i_1} \texttt{ = } v_{i_1}^{\mathit{num}}\ \texttt{in}\ m_{i_1}\ \texttt{;}\ \dots\ \texttt{;}\ \texttt{int}\ x_{i_k} \texttt{ = } v_{i_k}^{\mathit{num}}\ \texttt{in}\ m_{i_k}\ \text{where}\ (x_{i_j}, m_{i_j}, i_j) \in \mathit{int\_params}(f)\end{array}}{\langle x \texttt{ = } f\texttt{(}v_1,\dots,v_k\texttt{)}, \langle \delta, h\rangle\rangle \xrightarrow{\mathrm{C},1}_{\psi} \langle x \texttt{ = } f\texttt{(}v_1,\dots,v_k\texttt{)}\ s_{\mathit{int\_init}}\ \texttt{;}\ \mathit{code}(f)\ , \langle \sigma' :: \delta, h\rangle\rangle}$$

E-Callx-Body

$$\dfrac{\langle s, \langle \sigma :: \delta, h\rangle\rangle \xrightarrow{\theta,p}_{\psi_f} \langle s', \langle \sigma' :: \delta', h'\rangle\rangle}{\langle x \texttt{ = } f\texttt{(}v_1,\dots,v_k\texttt{)}\ s, \langle \sigma :: \delta, h\rangle\rangle \xrightarrow{\theta,p}_{\psi_c} \langle x \texttt{ = } f\texttt{(}v_1,\dots,v_k\texttt{)}\ \ s', \langle \sigma' :: \delta', h'\rangle\rangle}$$

E-Return-R

$$\dfrac{\langle e, \sigma\rangle \xrightarrow{\theta,p}_{\psi} \langle e', \sigma\rangle}{\langle \texttt{return}\ e, \langle \sigma :: \delta, h\rangle\rangle \xrightarrow{\theta,p}_{\psi} \langle \texttt{return}\ e', \langle \sigma :: \delta, h\rangle\rangle}$$

E-Callx-Return

$$\dfrac{}{\langle x \texttt{ = } f\texttt{(}v_1,\dots,v_k\texttt{)}\ \texttt{return}\ n, \langle \sigma :: \delta, h\rangle\rangle \xrightarrow{\mathrm{C},1}_{\psi} \langle x \texttt{ = } n, \langle \delta, h\rangle\rangle}$$

E-Seq-Return

$$\dfrac{s_1 \in \{\texttt{return}, \texttt{return}\ n\}}{\langle s_1\ \texttt{;}\ s_2, \langle \delta, h\rangle\rangle \xrightarrow{\mathrm{C},1}_{\psi} \langle s_1, \langle \delta, h\rangle\rangle}$$

Figure 12: Dynamic Semantics of Function Calls and Returns

$$\begin{array}{rcl} RP_\psi(x = f(v_1, \ldots, v_n), Q, C) & = & Q\,[\psi(wr(\Lambda(x)) \cdot \mathit{Relspec\_act}(f, 0, X)/\mathcal{R}(\{x\} \cup X)] \\ & & [\mathit{Relspec\_act}(f, i_1, X)/\mathcal{R}(\{a_{i_1}\} \cup X)] \ldots \\ & & [\mathit{Relspec\_act}(f, i_k, X)/\mathcal{R}(\{a_{i_k}\} \cup X)] \\ & & \\ \mathit{Relspec\_act}(f, i, X) & = & \mathit{Relspec\_form}(f, i) \\ & & [\rho_1(\mathit{act\_par}(f, 1)) \cdot \mathcal{R}(\rho_2(\mathit{act\_par}(f, 1)) \cup Y)/\mathcal{R}(\{\mathit{form\_par}(f, 1)\} \cup Y)] \ldots \\ & & [\rho_1(\mathit{act\_par}(f, j)) \cdot \mathcal{R}(\rho_2(\mathit{act\_par}(f, j)) \cup Y)/\mathcal{R}(\{\mathit{form\_par}(f, j)\} \cup Y)] \ldots \\ & & [\rho_1(\mathit{act\_par}(f, n-1)) \cdot \mathcal{R}(\rho_2(\mathit{act\_par}(f, n-1)) \cup Y)/\mathcal{R}(\{\mathit{form\_par}(f, n-1)\} \cup Y)] \\ & & [\rho_1(\mathit{act\_par}(f, n)) \cdot \mathcal{R}(\rho_2(\mathit{act\_par}(f, n)) \cup X)/\mathcal{R}(\{\mathit{form\_par}(f, n)\})] \end{array}$$

Figure 13: Constraint Generation for Function Calls

reliable control transfers are given by Rely's machine model, reliable program stack manipulations require that a compiler allocate the program stack in a reliable memory region.

Return Value Evaluation. A `return` e statement fully evaluates e under the hardware reliability model, yielding a value n [E-RETURN-E].

Call Return Execution. Once the body of the call executes and reaches a `return` statement, execution proceeds by restoring the old state σ' from the program stack, assigning the return value to the destination variable, and transferring control back to the caller [E-CALL-RETURN-E]. Note that as with [E-CALLX-UNFOLD], this step executes correctly with probability 1 and therefore both the control transfer and stack manipulation are fully reliable.

B. Constraint Generation for Functions

Figure 13 presents the constraint generation rule for function calls. Intuitively, for a function call $x = f(e_1, \ldots, e_n)$ the constraint generator performs two tasks. First, it substitutes the declared reliabilities of with the reliability expressions for the actual parameters of the function known at the call site. Second, to update the reliability expressions for the modified array variables, it constructs a constraint that abstracts the reliability of a function call as the reliability of multiple assignment statements – one statement represents the assignment of the final value of the function to the variable x, the remaining statements represent the assignment of each potentially modified array parameter of the function.

To substitute the return value, the rule in Figure 13 computes the reliability expression derived from the reliability specification of the function's return value. The helper function *Relspec_act* replaces substitutes the reliability of all actual parameters of the function's call for the declared parameter names within the declared reliability expression for the return value (when $i = 0$) or one of the function's parameters (when $i \geq 1$). The rule then identifies array variables (in the presentation there is total of $k \leq n$ array variables) that are passed as parameters to the function and updates their reliabilities analogously.

The helper function *Relspec_act* takes as input the function's specification, the index i of the array parameter or 0 for the return value, and the set of the variables from the context of the calling function. The function 1) obtains the reliability specification associated with the i-th parameter of the function, 2) the name of the i-th formal parameter using the *formal_arg* function, and 3) the reliability of the actual parameter expression using the *act_arg* function. The function *act_arg* returns the reliability of an expression if the actual parameter is a numerical value or the reliability pair $(1.0, \mathcal{R}(a_i))$ if the actual parameter is an array variable.

Then the function *Relspec_act* substitutes reliabilities of all function's formal parameters with the expressions of the corresponding actual parameters. The resulting expression does not contain the names of the formal parameters. We note that the rule for function calls requires that the names of the formal parameters are distinct from the names of variables used in the actual parameter reliability expressions. This can be enforced by additional renaming of formal parameters.

C. Properties of Reliability Factors

In this section we present proofs of several properties of reliability factors.

C.1 Discrete Distribution

This theorem ensures that the distribution of unreliable distributions is discrete, i.e., it can be represented by a probability mass function φ.

Lemma 1 (Discrete Distribution)**.** *The probability space of unreliable environments* (E, φ) *is discrete.*

Proof. A probability distribution is discrete if it is defined on a countable sample space. Therefore, we need to check that the set E is countable. In this proof we denote the cardinality of a set X as $|X|$ and as a special case the cardinality of a set of natural numbers, $|\mathbb{N}| = \aleph_0$.

The cardinality of the set of heaps is $|\mathrm{Loc} \times \mathrm{Int_M}|$. Since both $|\mathrm{Loc}|$ and $|\mathrm{Int_M}|$ are finite, the cardinality of their product is also finite. The cardinality of a single stack frame is $|\Sigma| = |\mathrm{Var} \times \mathrm{Ref}|$. The ardinality of Var is finite since a function can have only a finite number of variables. Cardinality $|\mathrm{Ref}| = |\mathrm{Loc} \times \mathrm{Loc}^k \times M|$ is finite, since Loc and M are finite, and the number of the array dimensions is a finite number (which follows from the definition of the program's syntax). The cardinality of a stack is then equal to $\aleph_0$ since as a finite or a countable length list of finite-length frames, it can be mapped onto $\mathbb{N}$. The cardinality of E is equal to the product of cardinalities of $|H|$ and $|\Sigma|$, and is equal to $\aleph_0$. Therefore, E is a countable set.

□

C.2 Reliability Factor Ordering

The ordering proposition states that the function $\mathcal{R}(X)$ is monotonically decreasing – for a larger set of variables X, the probability that the variables in that set have the same value decreases. This proposition also enables comparison of symbolic reliability predicates.

Proposition 1 (Ordering; restated). *For two sets of variables X and Y, if $X \subseteq Y$ then $\mathcal{R}(Y) \leq \mathcal{R}(X)$.*

Proof. First, we consider the case when all variables in X and Y are scalars. Let $U_Y = \mathcal{E}(Y, \varepsilon)$ be the set of alternative environments in which all variables in the set Y have the same value as in the original environment and let $U_X = \mathcal{E}(X, \varepsilon)$ be the set of alternative environments in which all variables in the set X have the same value as in the original environment. Then, if $X \subseteq Y$, the set $U_Y \subseteq U_X$, since the variables in $Y \backslash X$ provide additional restrictions on the states that are contained in U_Y. Finally, the theorem statement follows from the inequality $\sum_{v \in U_X} \varphi(v) \geq \sum_{v \in U_Y} \varphi(v)$.

If a is an array variable, then the function equiv adds a constraint for each element of a. Then, we can apply the same argument for each such obtained sets U_X and U_Y. □

As a side note, joint reliability of multiple variables can be obtained from the marginal reliabilities of these variables:

Proposition 1 (Union Bound). *Let $X = \{x_1, \ldots, x_n\}$ be a set of n program variables. Then, $\mathcal{R}(X) \geq \sum_{i=1}^{n} \mathcal{R}(\{x_i\}) - (n-1)$.*

Proof. The probability that the in the set of variables X, at least one variable has an incorrect value is $1 - \mathcal{R}(X)$. The probability that a variable x_i has an incorrect value is $1 - \mathcal{R}(\{x_i\})$.

The probability that any of variables in X has an incorrect value is smaller than the probability that every individual program variable has an incorrect value (from the classical probabilistic union bound). Then, the statement follows from $1 - \mathcal{R}(X) \leq \sum_{i=1}^{n} 1 - \mathcal{R}(\{x_i\})$. □

C.3 Redundant Constraints

As a consequence of ordering we can define the simplification procedure that removing redundant predicates from the constraints that the analysis produces.

Proposition 2 (Predicate Subsumption). *A reliability predicate $r_1 \cdot \mathcal{R}(X_1) \leq r_2 \cdot \mathcal{R}(X_2)$ subsumes (i.e., soundly replaces) a predicate $r_1' \cdot \mathcal{R}(X_1') \leq r_2' \cdot \mathcal{R}(X_2')$ if $r_1' \cdot \mathcal{R}(X_1') \leq r_1 \cdot \mathcal{R}(X_1)$ and $r_2 \cdot \mathcal{R}(X_2) \leq r_2' \cdot \mathcal{R}(X_2')$*

Proof. If $P = r_1 \cdot \mathcal{R}(X_1) \leq r_2 \cdot \mathcal{R}(X_2)$ and $P' = r_1' \cdot \mathcal{R}(X_1') \leq r_2' \cdot \mathcal{R}(X_2')$, we show that $P \wedge P_{rest} \implies P \wedge P' \wedge P_{rest}$, where P_{rest} are the remaining predicates in the constraint.

Let us first consider the case when P_{rest} is empty. Then, we need to show that $P \implies P \wedge P'$. This predicate follows from the inequality assumptions from the theorem statement. Specifically, if $r_1 \cdot \mathcal{R}(X_1) \leq r_2 \cdot \mathcal{R}(X_2)$ is true, by the ordering lemma and the widening of the interval, $r_1' \cdot \mathcal{R}(X_1') \leq r_2' \cdot \mathcal{R}(X_2')$ is also true.

If P_{rest} is not empty, one can replace the logical value true or false for P_{rest} and then use the same argument as in the previous derivation. □

```
#define tolerance 0.000001
#define maxsteps 40

float<0.9999*R(x)> F(float x in unrel);

float<0.9999*R(x)> dF(float x in unrel);

float <0.995*R(xa, xb)> secant
  (float xa in unrel, float xb in unrel) {
   float a in unrel;
   float b in unrel;
   float c in unrel;
   float fa in unrel;
   float fb in unrel;
   float fc in unrel;
   bool converged in unrel;

   a = xa;
   b = xb;

   fa = F(a);
   fb = F(b);

   while ((a -. b >=. tolerance) ||.
          (a -. b <=. 0.0-.tolerance))
         : maxsteps {
      c = (a +. b) /.2;
      fc = F(c);

      if ((fa >= 0) &&. (fc >= 0)) {
        a = c;
        fa = fc;
      } else {
        b = c;
        fb = fc;
      }
   }
   if (!.((a -. b <=. tolerance) &&.
      (a -. b >=. 0.0-.tolerance))) {
      x = INFTY;
   }
   return x;
}
```

Figure 14: Secant Method Implementation

```
//Array with sine polynomial coefficients
#define nsin 20
const csin (1) = { /*...*/ };

//Array with cosine polynomial coefficients
#define ncos 19
const ccos (1)  = { /*...*/ };

float<0.99999*R(x)> usin(float x in unrel) {
     float res in unrel;
     float t in unrel;

     int i; i = 1;

     res = csin[0];
     while (true) : nsin {
        t = csin[i];
        res = res *. x +. t;
        i = i + 1;
     }
     return res;
}

float<0.99999*R(x)> ucos(float x in unrel) {
     float res in unrel;
     float t in unrel;

     int i;
     i = 1;


     res = ccos[0];
     while (true) : ncos {
        t = ccos[i];
        res = res *. x +. t;
        i = i + 1;
     }
     return res;
}

void main
   (float r in unrel, float theta in unrel
    float<0.99995*R(r, theta, xy)> xy) {

    float x in unrel;
    float y in unrel;
    float t in unrel;

    t = ucos(theta);
    xy(1) = r *. t;
    t = usin(theta);
    xy(2) = r *. t;

}
```

Figure 15: Coordinate Conversion Implementation

```
const num matx = 64;
const num maty = 64;

void matvec (float mat(2) in urel,
             float v(1) in urel,
             num<.997*R(mat, vec, outvec)>
                  u(1) in urel)
{
  num t1 in urel;
  num t2 in urel;
  num t3 in urel;
  int i, j, k;

  i = 0;
  repeat matx  {
    j = 0;
    t3 = 0;
    repeat maty {
           t1 = mat[j, i];
           t2 = v[j];
           t3 = t3 +. t1 *. t2;
    }

    u[i] = t3;
    i = i + 1;
  }
}
```

Figure 16: Matrix-Vector multiplication Implementation

D. Benchmarks

D.1 Newton's Method

This is the computation we presented in Section 6.2.

D.2 Secant Method

This computation also searches for a root of function f. It takes as its input two points x_a and x_b for which the function has the opposite sign and returns the value x_0 within the interval $[x_a, x_b]$ for which the function f evaluates to zero. The result of the computation can be checked (as in Newton's method) by ensuring that $f(x_0)$ is close to zero.

This is the fixed point computation that at each step computes the middle point x_c and its function $f(x_c)$. Based on the sign of $f(x_c)$ the algorithm divides the search interval (using a conditional statement). The whole loop can execute unreliably. If the function f has the reliability specification `float<0.9999*R(x)> F(float x)`, then the analysis verifies that the reliability of the computation is at least `.995*R(xa, xb)`.

Figure 14 presents the implementation of this computation. Note that the reliability of this computation is higher than the reliability of Newton's method. This is because the Secant method makes a single call to the function f, whereas Newton's method makes calls to both f and f' in every iteration. Given multiple options (such as Newton's method and Secant), verified reliability specifications may help developers select the option that best satisfies their combined reliability and efficiency goals.

D.3 Coordinate Conversion

This computation converts polar coordinates of a point to Cartesian coordinates. Given a coordinate (R, θ), where R is a radius and θ is an angle, it computes the coordinates $x = R\cos(\theta)$ and $y = R\sin(\theta)$. This is also a checkable computation: the square of the diameter is equal to the sum of the squares of Cartesian coordinates.

Figure 15 presents the implementation of this computation. The function takes the inputs `r` and `theta` and stores the result in the output array `xy`. The body of the computation can execute unreliably. The analysis first verifies the implementation of the trigonometric functions (which are evaluations of appropriate Chebyshev interpolating polynomials). If the reliability of the input x is `R(x)`, then the the analysis verifies that the functions $\sin(x)$ and $\cos(x)$ have the reliability at least `0.99999*R(x)`. The analysis uses this result to verify that if the reliability of the parameters `r` and `theta` is `R(r,theta)` , then the reliability of the coordinate conversion computation is at least `0.99995*R(r,theta,xy)`.

D.4 Motion Estimation

This is the computation we presented in Section 2.

D.5 Matrix-vector Multiplication

This computation calculates the product of a matrix M and a vector v. The dimensions of the matrix are $w \times h$, the length of v is h, and the length of the resulting vector u is w. The computation takes `M`, `v`, and `u` as inputs, and computes the values of the elements of the vector `u`. This computation is an important kernel in numerical computations. For example, a linear operator (such as Fourier or Discrete Cosine transforms of arbitrary sizes) can be implemented as a matrix-vector multiplication.

Figure 16 presents the implementation of this computation. All operations on the input data can execute unreliably. The output of the function is the vector u. Assuming the maximum size of the square matrix to be 64x64 (as in some signal processing applications), the specified output reliability for the vector `u` is `0.997*R(M, v, u)`. The analysis result states that the reliability of *any* element of u is greater than this specified output reliability. Note that the specification needs to account for the possible unreliability of assignments to the vector u before entering the function, because it does not track the array indices to determine that every element of `u` is modified inside the function.

D.6 Hadamard Transform

This computation takes as input two blocks of 4x4 pixels and computes the sum of differences between the pixels in the frequency domain. This computation is used in digital signal processing applications (including motion estimation algorithms).

Figure 17 presents the implementation of this computation. The computation calculates the intermediate distances

```
int <0.9999995 * R(val)> abs
    (int val in unrel)
{
 int t in unrel = val;
 if (t <=. 0) {
   t = 0 -. t;
 }
 return t;
}

int<0.99995 * R(bA, bB, satdstart)>
hadamarddiff
   (int<R(bA)> bA(2) in unrel,
    int<R(bB)> bB(2) in unrel,
    int satdstart in unrel)
{
 int isatd in unrel;
 int tmp00 in unrel; int tmp01 in unrel;
 int tmp02 in unrel; int tmp03 in unrel;
 int tmp10 in unrel; int tmp11 in unrel;
 int tmp12 in unrel; int tmp13 in unrel;
 int tmp20 in unrel; int tmp21 in unrel;
 int tmp22 in unrel; int tmp23 in unrel;
 int tmp30 in unrel; int tmp31 in unrel;
 int tmp32 in unrel; int tmp33 in unrel;
 int t1 in unrel; int t2 in unrel;
 int t3 in unrel; int t4 in unrel;
 int tt in unrel;

 t1 = bA[0,0]; tt = bB[0,0]; t1 = t1 -. tt;
 t2 = bA[0,1]; tt = bB[0,1]; t2 = t2 -. tt;
 t3 = bA[0,2]; tt = bB[0,2]; t3 = t3 -. tt;
 t4 = bA[0,3]; tt = bB[0,3]; t4 = t4 -. tt;
 tmp00 =  t1 +. t2 +. t3 +. t4;
 tmp01 =  t1 +. t2 -. t3 -. t4;
 tmp02 =  t1 -. t2 -. t3 +. t4;
 tmp03 =  t1 -. t2 +. t3 -. t4;

 t1 = bA[1,0]; tt = bB[1,0]; t1 = t1 -. tt;
 t2 = bA[1,1]; tt = bB[1,1]; t2 = t2 -. tt;
 t3 = bA[1,2]; tt = bB[1,2]; t3 = t3 -. tt;
 t4 = bA[1,3]; tt = bB[1,3]; t4 = t4 -. tt;
 tmp10 =  t1 +. t2 +. t3 +. t4;
 tmp11 =  t1 +. t2 -. t3 -. t4;
 tmp12 =  t1 -. t2 -. t3 +. t4 ;
 tmp13 =  t1 -. t2 +. t3 -. t4 ;
 t1 = bA[2,0]; tt = bB[2,0]; t1 = t1 -. tt;
 t2 = bA[2,1]; tt = bB[2,1]; t2 = t2 -. tt;
 t3 = bA[2,2]; tt = bB[2,2]; t3 = t3 -. tt;
 t4 = bA[2,3]; tt = bB[2,3]; t4 = t4 -. tt;
 tmp20 =  t1 +. t2 +. t3  +. t4;
 tmp21 =  t1 +. t2 -. t3  -. t4;
 tmp22 =  t1 -. t2 -. t3  +. t4;
 tmp23 =  t1 -. t2 +. t3  -. t4;

 t1 = bA[3,0]; tt = bB[3,0]; t1 = t1 -. tt;
 t2 = bA[3,1]; tt = bB[3,1]; t2 = t2 -. tt;
 t3 = bA[3,2]; tt = bB[3,2]; t3 = t3 -. tt;
 t4 = bA[3,3]; tt = bB[3,3]; t4 = t4 -. tt;
 tmp30 =  t1 +. t2 +. t3 +. t4;
 tmp31 =  t1 +. t2 -. t3 -. t4;
 tmp32 =  t1 -. t2 -. t3 +. t4;
 tmp33 =  t1 -. t2 +. t3 -. t4;

 isatd = satdstart;

 t1 = abs(tmp00 +. tmp10 +. tmp20 +. tmp30);
 t2 = abs(tmp00 +. tmp10 -. tmp20 -. tmp30);
 t3 = abs(tmp00 -. tmp10 -. tmp20 +. tmp30);
 t4 = abs(tmp00 -. tmp10 +. tmp20 -. tmp30);
 isatd = isatd +. t1 +. t2 +. t3 +. t4;

 t1 = abs(tmp01 +. tmp11 +. tmp21 +. tmp31);
 t2 = abs(tmp01 +. tmp11 -. tmp21 -. tmp31);
 t3 = abs(tmp01 -. tmp11 -. tmp21 +. tmp31);
 t4 = abs(tmp01 -. tmp11 +. tmp21 -. tmp31);
 isatd = isatd +. t1 +. t2 +. t3 +. t4;

 t1 = abs(tmp02 +. tmp12 +. tmp22 +. tmp32);
 t2 = abs(tmp02 +. tmp12 -. tmp22 -. tmp32);
 t3 = abs(tmp02 -. tmp12 -. tmp22 +. tmp32);
 t4 = abs(tmp02 -. tmp12 +. tmp22 -. tmp32);
 isatd = isatd +. t1 +. t2 +. t3 +. t4;

 t1 = abs(tmp03 +. tmp13 +. tmp23 +. tmp33);
 t2 = abs(tmp03 +. tmp13 -. tmp23 -. tmp33);
 t3 = abs(tmp03 -. tmp13 -. tmp23 +. tmp33);
 t4 = abs(tmp03 -. tmp13 +. tmp23 -. tmp33);
 isatd = isatd +. t1 +. t2 +. t3 +. t4;

 return isatd;
}
```

Figure 17: Hadamard Transform Implementation

between the elements stored in the arrays and computes the sum of the absolute differences of combinations of these elements. The video can be stored in unreliable memory and the entire computation can execute unreliably. The analysis verifies that if the reliability of the two input blocks `bA` and `bB` is `R(bA, bB)`, then the reliability of the computation is greater than `0.99995*R(bA, bB)`.