Improved Methods for Modeling Uncertainty in RC Timing Analysis

Cheryl L. Harkness
Daniel P. Lopresti

Department of Computer Science
Brown University
Providence, Rhode Island 02912

Technical Report No. CS-89-43
November 1989

Improved Methods for Modeling Uncertainty in RC Timing Analysis †

Cheryl L. Harkness

Daniel P. Lopresti

Department of Computer Science

Brown University

Providence, RI 02912

Abstract

Uncontrollable variations in the fabrication process make it impossible to accurately predict the circuit parameters used in timing analysis. Instead, these values are known to fall within certain limits, resulting in a range of possible circuit behaviors. In this paper, we present two new interval-based techniques for computing best- and worst-case bounds on circuit delay. Both produce tighter bounds than an approach we described in an earlier paper. When compared to Monte Carlo simulation, all of the interval methods are more precise and significantly faster.

†This research was supported in part by the Defense Advanced Research Projects Agency under ONR contract N00014-83-K-0146. Daniel Lopresti is also supported by the National Science Foundation under grant MIP-87-10745.

1. Introduction

The purpose of RC timing analysis is to estimate the operating speed of a VLSI chip from the resistances and capacitances of its components. These resistances and capacitances are determined from the chip structure and the manufacturing process. Even subtle variations in the fabrication process change the values of the circuit parameters. Although expected values can be obtained for them, these elements actually have a range of possible values, typically varying between 1 and 20 percent from the mean. Currently, most timing analyzers use only expected values in their computations, thus returning only a single delay estimate. Using the full range of possible input values can yield a more complete characterization of potential performance. Our goal, therefore, is to determine the corresponding range of possible timing behaviors produced by these variations.

In mathematical terms, this problem can be expressed as follows: Let $F(x_1, x_2, ..., x_n)$ be any real-valued, arithmetic function of n independent variables such that $x_i \in \Re$ and $F : \Re^n \rightarrow \Re$. Now assume that the value of each variable, x_i, is not uniquely determined. Instead, we are given a range of possible values $\hat{x}_i$ for each x_i. The tightest possible bounds on F are found by evaluating the function on all possible combinations of these values and taking the union of the results. This is called the *United Extension of F* [Moor79, p.18] and is formally defined as follows:

$$\hat{F}(\hat{x}_1, \hat{x}_2, ..., \hat{x}_n) \equiv \bigcup_{x_i \in \hat{x}_i} F(x_1, x_2, ..., x_n) \tag{1}$$

For timing analysis, the function F is the delay equation and the variables x_i are the capacitances, resistances, and transistor sizes used to compute this delay. Because $x_i \in \Re$, each range $\hat{x}_i$ contains an infinite number of possible assignments for x_i; therefore, we cannot compute $\hat{F}$ directly and must find some other method to approximate it.

One common method for approximating $\hat{F}$ is the Monte Carlo method [Sobo75] or the "Method of Statistical Trials." With this technique, analysis is repeated for random combinations of values chosen from within the acceptable range of each parameter. The minimum and maximum values computed in these trials form the best- and worst-case bounds. The quality of these bounds increases with the number of trials performed. Unfortunately, accurately determining the behavior of a circuit requires a large number of experiments; thus, while Monte Carlo Simulation can be effective, it is also time-consuming.

In [Hark89], we presented a method based on interval algebra for modeling the effects of uncertainty in RC analysis. In particular, we represented uncertain parameters as intervals and used interval arithmetic to manipulate these ranges. Formally, an interval $[a, b]$ is defined as the set $\{x \mid x \in \Re, a \leq x \leq b\}$. A *degenerate* interval $[a, a]$ is an interval which contains only a single element, a. Let I denoted the set of all intervals over $\Re$. The basic interval arithmetic operations

$\oplus$ (addition), $\ominus$ (subtraction), $\odot$ (multiplication), and $\oslash$ (division) over the set I are defined as follows:

$$[a,b] \oplus [c,d] \equiv [a+c, b+d] \tag{2}$$

$$[a,b] \ominus [c,d] \equiv [a-d, b-c] \tag{3}$$

$$[a,b] \odot [c,d] \equiv [min(ac,ad,bc,bd), max(ac,ad,bc,bd)] \tag{4}$$

$$[a,b] \oslash [c,d] \equiv [a,b] \odot [1/c, 1/d], \quad 0 \notin [c,d] \tag{5}$$

Using these interval operations, we created an interval extension $\mathcal{F}$ of the function F which we used to approximate $\hat{F}$. We formed this interval extension by replacing the basic arithmetic operators $+$, $-$, $\cdot$, and $/$ in F by their respective interval counterparts $\oplus$, $\ominus$, $\odot$, and $\oslash$. We then proved a key theorem which states that the interval extension of any function generates bounds which contain its United Extension.

Theorem 1.1 $\hat{F}(\hat{x}_1, \hat{x}_2, ..., \hat{x}_n) \subset \mathcal{F}(\hat{x}_1, \hat{x}_2, ..., \hat{x}_n)$. *(from [Hark89])*

To illustrate this theorem, we implemented an interval version of an existing RC analysis algorithm, Crystal's PR-Slope model [Oust83, Oust84, Oust86], and compared the bounds that it generated to the theoretical bounds ($\hat{F}$) for several circuits. While the interval results ($\mathcal{F}$) always contain the theoretical bounds, they are often overly conservative. The reason for this is that the interval algebra ignores variable interdependencies. Each occurrence of a variable is considered independent of all other occurrences. For example, consider the function $F(x) = x + 1/x$ where $\hat{x} = [0.5, 2]$. The United Extension of this function is $[2, 2.5]$, but the interval solution is $[1, 4]$. The interval upper bound of four results from using the maximum value of x for the first occurrence of the variable and its minimum value for the second (i.e., $2 + 1/0.5 = 4$). If we wish to produce tighter delay bounds, we must develop alternative methods that account for variable interdependence.

In this paper, we explore two other interval-based techniques for approximating $\hat{F}$. The first of these methods employs centred forms and the second uses interval differential calculus, an interval extension of real differential calculus. We incorporate them into Crystal's PR-slope model to create two new algorithms for computing bounds on circuit delay. Although Crystal was chosen for these experiments, the same techniques can be applied to other timing verifiers.

In the next two sections, we describe the two new interval methods. Following this, we show how these techniques may be applied to RC circuit analysis for timing verification. Finally, we analyze the performance of these methods, comparing them to our original interval algorithm and with Monte Carlo simulation.

2. Krawczyk's Centred Form

Centred forms provide an alternative method for approximating $\hat{F}$. While there are many types of centred forms (see [Rats84]), all share a common framework and structure. We begin by choosing a particular set of values, $c_1, c_2, ..., c_n$, such that each $c_i \in \hat{x}_i$ and evaluating $F(c_1, c_2, ..., c_n)$. While any choice $c_i \in \hat{x}_i$ can be used, the midpoint of each interval is typically chosen. We now rewrite the function F to create an equivalent representation F_C which is centered around $F(c_1, c_2, ..., c_n)$. This new representation is known as the *General Centred-Form of F* and is defined as follows:

$$F(x_1, x_2, ..., x_n) = F(c_1, c_2, ..., c_n) + G(x_1 - c_1, x_2 - c_2, ..., x_n - c_n) \tag{6}$$

Having transformed F, we then create an interval version by replacing each variable x_i by its interval range $\hat{x}_i$ and each arithmetic operator by its interval counterpart. When evaluated, this new interval function produces symmetric bounds centered at $F(c_1, c_2, ..., c_n)$ containing $\hat{F}$. Although function F and its centred form F_C are equivalent, their interval extensions are not. Since the resulting interval versions contain different numbers and combinations of the arithmetic operations, the accumulation of error is different. In most cases, the centred forms produce tighter bounds.

Centred forms differ from one another in their choice of G. For instance, *standard centred forms* use a Taylor series expansion of order k to create G. Using this method, we would have to generate a different closed form function G for each function F. In RC analysis, the delay equation varies with the number of transistors in the stage (called the *stage length*); therefore, to use the standard centred form, we must precompute a closed centred expression for each possible stage length encountered. We require a more flexible approach.

Mean-value forms use derivatives to create G. In this case, $G \equiv F'(x) \cdot (x - c)$; therefore, the mean-value form of $F(x_1, x_2, ..., x_n)$ is

$$F(x_1, x_2, ..., x_n) = F(c_1, c_2, ..., c_n) + \sum_{i=1}^{n} \frac{\partial F(x_1, x_2, ..., x_n)}{\partial x_i} \cdot (x_i - c_i). \tag{7}$$

Ratschek and Rokne, however, claim that the mean-value form produces poorer bounds estimates than higher-order standard forms [Rats84, pp. 78-81].

Another centred form is *Krawczyk's centred form* [Rats84, pp. 59-62]. This form uses interval slopes to compute G. With this method, there is no general closed form for G; instead G is computed directly from F by construction. We begin by writing a step-by-step algorithm for computing F, called a *function procedure*. Let $B = \{b_1, b_2, ..., b_r\}$ be the set of real constants that appear in F, let $X = (x_1, x_2, ..., x_n)$ be the vector of independent variables in F, and let $U = \{u_1, u_2, ..., u_n\}$ be a finite set dependent variables used to compute F. A function procedure is a series of computational steps for calculating the value of a function and is defined as follows:

$u_i = x_i$	for $i = 1, ..., n$	/* load variables */
$u_i = b_{i-n}$	for $i = n+1, ..., n+r$	/* load constants */
$u_i = u_j *_i u_k$	for $i = n+r+1, ..., s$	/* calculate F */
	where $j, k < i$ and $*_i \in \{+, -, \cdot, /\}$	

For example, if $F(x) = x + 1/x$, then the following sequence of instructions constitutes one possible function procedure for computing F:

$$\begin{aligned} u_1 &= x \\ u_2 &= 1 \\ u_3 &= u_2/u_1 \\ u_4 &= u_1 + u_3 \end{aligned}$$

Given a function procedure for F, we employ it to compute the interval slope G. Define $E_1, E_2, ..., E_n \in I^n$ to be n-dimensional unit vectors such that $E_1 = ([1,1],[0,0],...,[0,0])$, $E_2 = ([0,0],[1,1],...,[0,0])$, ..., $E_n = ([0,0],...,[0,0],[1,1])$. Likewise, let $O \in I^n$ be an n-dimensional zero vector. The interval slope $G = (\hat{g}_1, \hat{g}_2, ..., \hat{g}_n)$ is defined recursively from the s-instruction function procedure as follows:

$G_i = E_i$	if $u_i = x_i$	/* load variables */
$G_i = O$	if $u_i = b_{i-n}$	/* load constants */
$G_i = G_j \oplus G_k$	if $u_i = u_j + u_k$	/* addition */
$G_i = G_j \ominus G_k$	if $u_i = u_j - u_k$	/* subtraction */
$G_i = G_j \odot \mathcal{F}_k(\hat{X}) \oplus G_k \odot F_j(C)$	if $u_i = u_j u_k$	/* multiplication */
$G_i = G_j \oslash \mathcal{F}_k(\hat{X}) \ominus G_k \odot F_i(C) \oslash \mathcal{F}_k(\hat{X})$	if $u_i = u_j/u_k$	/* division */

The recursion terminates after s steps and the interval slope $G = G_s$. For example, let $F(x) = x + 1/x$, $\hat{x} = [0.5, 2]$, and $c = 1.25$. Using the function procedure given above, we can derive the interval slope G for this function. The results of each computational step are shown in Table 1. The last row of the table displays the resulting interval slope.

Having found G, we can construct Krawczyk's centred form $\mathcal{F}_C$ for function F using the following formula:

$$\mathcal{F}_C(\hat{x}_1, \hat{x}_2, ..., \hat{x}_n) = F(c_1, c_2, ..., c_n) \oplus \sum_{i=1}^{n} \hat{g}_i \odot (\hat{x}_i \ominus c_i). \tag{8}$$

Using the results in Table 1, this equation yields $\mathcal{F}_C = [1.6, 2.5]$. This interval is centered at $F(c) = 2.05$ and represents the smallest such interval containing the theoretical bounds $[2, 2.5]$. Recall that the interval bounds $\mathcal{F}$ for the same function are $[1, 4]$. In this case, the centred form produces tighter bounds.

i	u_i	$F_i(c)$	$\mathcal{F}_i(\hat{x})$	G_i
1	x	1.25	[0.5,2]	[1,1]
2	1	1	[1,1]	[0,0]
3	u_2/u_1	0.8	[0.5,2]	[-1.6,-0.4]
4	$u_1 + u_3$	**2.05**	**[1,4]**	**[-0.6,0.6]**

Table 1: Computing the Interval Slope G

Unlike the standard centred forms, Krawczyk's centred forms do not have to be precomputed and can be easily programmed for any function. The algorithm has a time complexity of $O(sn)$ where s represents the number of instructions in the function procedure for computing F and n represents the number of independent variables. The flexibility of this format coupled with its low computational complexity make it an ideal candidate for the timing analysis problem.

Each of the centred forms presented above yield bounds which subsume the true bounds $\hat{F}$; thus, $\hat{F} \subset \mathcal{F}_C$. Proofs of convergence for all of these centred methods are found in [Rats84] and will not be repeated here.

3. Interval Differential Calculus

3.1. Rall's Algorithm

The second method for approximating $\hat{F}$ employs interval differential calculus, which was introduced by Rall in [Rall86]. Given a function $F(x)$ over $\Re$ and an interval $\hat{x} = [a, b]$, Rall uses differential calculus to compute $\hat{F}$. The derivative F' provides information about the monotonicity of the function which in turn can be used to obtain a better approximation to $\hat{F}$. For instance, if $F'(x) \geq 0$ for all $x \in \hat{x}$, then function F is monotone increasing over the interval $\hat{x}$ and

$$\hat{F}(\hat{x}) = [F(a), F(b)]. \tag{9}$$

Likewise, if $F'(x) \leq 0$ for all $x \in \hat{x}$, then the function is monotone decreasing over the interval $\hat{x}$ and

$$\hat{F}(\hat{x}) = [F(b), F(a)]. \tag{10}$$

Rall further demonstrates that $F'(x)$ for all $x \in \hat{x}$ can be approximated using interval differential arithmetic, a natural extension of real differential arithmetic.

The operands for real differential arithmetic are ordered pairs in $\Re^2$ such that the first element of the pair contains a variable or function and the second element its derivative. If $X = (x, x')$ and

$Y = (y, y')$ are two such operands, then the rules of real differential arithmetic are:

$$X + Y = (x, x') + (y, y') = (x + y, x' + y') \tag{11}$$

$$X - Y = (x, x') - (y, y') = (x - y, x' - y') \tag{12}$$

$$X \cdot Y = (x, x') \cdot (y, y') = (x \cdot y, x \cdot y' + y \cdot x') \tag{13}$$

$$X/Y = (x, x')/(y, y') = (x/y, (x' - (x/y) \cdot y')/y), \quad y \neq 0 \tag{14}$$

The interval differential algebra is similar except that all operands are interval pairs and the basic arithmetic operators $+$, $-$, $\cdot$, and $/$ have been replaced by their interval equivalents $\oplus$, $\ominus$, $\odot$, and $\oslash$. If $\hat{X} = (\hat{x}, \hat{x}')$ and $\hat{Y} = (\hat{y}, \hat{y}')$, then the interval differential operators are:

$$\hat{X} \oplus \hat{Y} = (\hat{x}, \hat{x}') \oplus (\hat{y}, \hat{y}') = (\hat{x} \oplus \hat{y}, \hat{x}' \oplus \hat{y}') \tag{15}$$

$$\hat{X} \ominus \hat{Y} = (\hat{x}, \hat{x}') \ominus (\hat{y}, \hat{y}') = (\hat{x} \ominus \hat{y}, \hat{x}' \ominus \hat{y}') \tag{16}$$

$$\hat{X} \odot \hat{Y} = (\hat{x}, \hat{x}') \odot (\hat{y}, \hat{y}') = (\hat{x} \odot \hat{y}, \hat{x} \odot \hat{y}' \oplus \hat{y} \odot \hat{x}') \tag{17}$$

$$\hat{X} \oslash \hat{Y} = (\hat{x}, \hat{x}') \oslash (\hat{y}, \hat{y}') = (\hat{x} \oslash \hat{y}, (\hat{x}' \ominus (\hat{x} \oslash \hat{y}) \odot \hat{y}') \oslash \hat{y}), \quad [0,0] \notin \hat{y} \tag{18}$$

We use these interval differential operators to compute the interval derivative of F, $\mathcal{F}'(\hat{x})$.

Having introduced interval differential calculus, Rall incorporates it into his algorithm to estimate $\hat{F}(\hat{x})$ as summarized in Figure 1. We begin by computing the interval derivative of function F. If this derivative indicates that the function is monotone over the interval $\hat{x}$, then we compute $\hat{F}(\hat{x})$ directly using Equation 9 or 10; otherwise, we partition the interval $\hat{x}$ into two halves and repeat the analysis on each half. The number of interval bisections performed by this algorithm is specified by the user. When this number is exceeded, the bounds must be estimated. In this case, Rall uses $\mathcal{F}$ to estimate subinterval bounds. The final bounds on the function are found by taking the union of all subinterval bounds. The number of estimated subinterval components indicates the quality of the final result. In particular, the fewer the number of non-monotone subintervals, the better the resulting bounds are likely to be.

To illustrate this approach, consider our previous example $F(x) = x + 1/x$ with $\hat{x} = [0.5, 2.0]$ and a bisection limit of 3. The results of each stage of this computation are displayed in Table 2. The last column in the table shows the subinterval bounds (if any) computed for each stage. Taking the union of these bounds yields the final result $\mathcal{F}_I(\hat{x}) = [1.82, 2.50]$. The theoretical bounds for this function are $[2, 2.5]$ and the interval bounds are $[1, 4]$. For this function, $\mathcal{F}_I$ produced much better results than $\mathcal{F}$.

```
Current bounds = F((a + b)/2)
Add interval x̂ = [a, b] to evaluation queue
While queue not empty
    Get next (sub)interval ŷ from queue
    Compute F'(ŷ)
    Check range of F'(ŷ)
    If monotone
        Compute exact (sub)interval bounds using Equation 9 or 10
        Update current bounds
    Else if possible, bisect ŷ into halves ŷ1 and ŷ2
        Add intervals ŷ1 and ŷ2 to evaluation queue
    Else (interval ŷ is too small to bisect)
        Approximate (sub)interval bounds using F(ŷ)
        Update current bounds
Return current bounds
```

Figure 1: Rall's Interval Differential Algorithm

$\hat{x}$	$\mathcal{F}(\hat{x})$	$\mathcal{F}'(\hat{x})$	Action	$\mathcal{F}_I(\hat{x})$
[0.50, 2.00]	[1.00,4.00]	[-3.00,0.75]	bisect	
[0.50, 1.25]	[1.30,3.25]	[-3.00,0.36]	bisect	
[1.25, 2.00]	[1.75,2.80]	[0.36,0.75]	compute bounds	[2.05,2.50]
[0.50, 0.88]	[1.64,2.88]	[-3.00,-0.31]	compute bounds	[2.02,2.50]
[0.88, 1.25]	[1.67,2.39]	[-0.31,0.36]	bisect	
[0.88, 1.06]	[1.82,2.21]	[-0.31,0.11]	estimate bounds	[1.82,2.21]
[1.06, 1.25]	[1.86,2.19]	[0.11,0.36]	compute bounds	[2.00,2.05]
TOTAL				**[1.82,2.50]**

Table 2: An Interval Differential Bounds Computation

3.2. Extensions to Rall's Algorithm

Rall's algorithm only works for single-variable functions, so we have generalized it to create one for multi-variable functions. Given a function $F(x_1, x_2, ..., x_n)$ over $\Re$ and the range of values for each variable $x_i \in \hat{x}_i = [min_i, max_i]$, we wish to approximate $\hat{F}$ using interval differential calculus. For multi-variable functions, we use *partial* derivatives. Let $L = (l_1, l_2, ..., l_n)$ and $U = (u_1, u_2, ..., u_n)$ be lower and upper bound vectors over $\Re^n$. If $\partial F/\partial x_i \geq 0$ for all legal value assignments, then F is monotone increasing with respect to x_i and

$$\begin{aligned} l_i &= min_i \\ u_i &= max_i. \end{aligned} \tag{19}$$

Likewise, if $\partial F/\partial x_i \leq 0$ for all legal value assignments, then F is monotone decreasing with respect to x_i and

$$\begin{aligned} l_i &= max_i \\ u_i &= min_i. \end{aligned} \tag{20}$$

If the function F is monotone over the ranges of all variables $x_1, x_2, ..., x_n$, then we can precisely compute $\hat{F}$ as follows:

$$\hat{F}(\hat{x}_1, \hat{x}_2, ..., \hat{x}_n) = [F(l_1, l_2, ..., l_n), F(u_1, u_2, ..., u_n)] \tag{21}$$

We can approximate each partial derivative $\partial F/\partial x_i$ evaluated at all possible assignments $x_j \in \hat{x}_j$ with an interval partial derivative, $\partial \mathcal{F}/\partial x_i$. The interval partial differential algebra needed for this computation is similar to the algebra presented in the previous section. As before, the operands are variable-derivative pairs, but in this case, the derivative is represented as a vector of n partial derivatives (e.g., $\hat{X} = (\hat{x}, (\hat{x}'_1, \hat{x}'_2, ..., \hat{x}'_n))$ where $\hat{x}'_i$ denotes the partial derivative of the i^{th} variable with respect to x). The arithmetic operators for this calculus are defined as follows:

$$\hat{X} \oplus \hat{Y} = (\hat{x} \oplus \hat{y}, (\hat{x}'_1 \oplus \hat{y}'_1, ..., \hat{x}'_n \oplus \hat{y}'_n)) \tag{22}$$

$$\hat{X} \ominus \hat{Y} = (\hat{x} \ominus \hat{y}, (\hat{x}'_1 \ominus \hat{y}'_1, ..., \hat{x}'_n \ominus \hat{y}'_n)) \tag{23}$$

$$\hat{X} \odot \hat{Y} = (\hat{x} \odot \hat{y}, (\hat{x} \odot \hat{y}'_1 \ \oplus \ \hat{y} \odot \hat{x}'_1, ..., \hat{x} \odot \hat{y}'_n \ \oplus \ \hat{y} \odot \hat{x}'_n)) \tag{24}$$

$$\hat{X} \oslash \hat{Y} = (\hat{x} \oslash \hat{y}, ((\hat{x}'_1 \ \ominus \ (\hat{x} \oslash \hat{y}) \odot \hat{y}'_1) \oslash \hat{y}, ..., (\hat{x}'_n \ \ominus \ (\hat{x} \oslash \hat{y}) \odot \hat{y}'_n) \oslash \hat{y})) \quad [0,0] \notin \hat{y} \tag{25}$$

Now when F is evaluated using these interval differential operations, the result is the operand $(\mathcal{F}, (\partial \mathcal{F}/\partial x_1, \partial \mathcal{F}/\partial x_2, ..., \partial \mathcal{F}/\partial x_n))$. The C language code for each of these operators in presented in the Appendix.

We contend that these interval partial derivatives may be used to determine the monotonicity of F. Recall that F is monotone increasing (decreasing) with respect to the variable x_i, if $\partial F/\partial x_i \geq 0$ (≤ 0) for all possible assignments $x_j \in \hat{x}_j$. By definition, $\partial F/\partial x_i$ evaluated at all possible variable values is just the United Extension of $\partial F/\partial x_i$. Furthermore, Theorem 1.1 states that the interval extension of a function produces bounds which contain its United Extension; therefore,

$$\hat{\partial F/\partial x_i}(\hat{x}_1, \hat{x}_2, ..., \hat{x}_n) \subset \partial \mathcal{F}/\partial x_i(\hat{x}_1, \hat{x}_2, ..., \hat{x}_n). \tag{26}$$

Hence, if $\partial \mathcal{F}/\partial x_i \geq 0$, then $\hat{\partial F/\partial x_i} \geq 0$ and F is monotone increasing with respect to x_i. We thus conclude that the interval partial derivatives computed by our algebra may be employed to determine the monotonicity of the function F.

Having defined the interval partial differential algebra, we are now ready to present our version of the bounds algorithm (See Figure 2). The algorithm begins by computing interval partial derivatives for each variable. It then checks the ranges of all partial derivatives for monotonicity, loading the lower L and upper U bounds arrays with the appropriate values. If the function is monotone with respect to all variables, then the true bounds on the function can be precisely found. If, however, the function is non-monotone over the range of some variable, we bisect the intervals and repeat the analysis on the pieces. As before, $\mathcal{F}$ is used to estimate the bounds when the subintervals become too small.

The partitioning method employed in this algorithm is crucial to both its efficiency and correctness. If we subdivide and carelessly group the intervals, we risk underestimating the true bounds. On the other hand, if we are too conservative (e.g., calculate all combinations of subintervals), the problem becomes computationally intractable. We, therefore, seek a balance between these two extremes. Our approach is to bisect only the non-monotone intervals and group the resulting subintervals such that all the values that are likely to produce upper bounds on the solution are collected together and all the values that are likely to produce lower bounds are similarly collected. More formally, if non-monotone interval $\hat{x}_i = [min_i, max_i]$, then let $\hat{x}_{i,1} = [min_i, mid_i]$ and $\hat{x}_{i,2} = [mid_i, max_i]$ where $mid_i = (min_i + max_i)/2$. Also let $\hat{L}$ represent the set of subintervals $\hat{x}_{i,j}$ that are likely to produce lower bounds and $\hat{U}$ represent the set of subintervals that are likely to produce upper bounds. The goal is to produce a partitioning $(\hat{L}, \hat{U})$ of the problem space. The difficulty here is that F is non-monotone with respect to variable x_i and we do not know which values of x_i minimize or maximize F. Here again, we use the partial derivatives to make an educated guess. Specifically, the magnitude of the derivative indicates the slope of the curve, while its sign denotes the direction (whether the function is increasing or decreasing). For non-monotone x_i, its upper and lower derivative bounds specify the maximum increasing and decreasing slopes respectively. A large upper bound implies that the function rapidly increases over some values of $\hat{x}_i$. Likewise, a large lower bound implies

```
Current bounds = F(c_1, ..., c_n) where c_i = (min_i + max_i)/2
Add interval vector X̂ = (x̂_1, x̂_2, ..., x̂_n) to evaluation queue
While queue not empty
    Get next entry Ŷ = (ŷ_1, ..., ŷ_n) from queue
    Compute ∂F/∂y_i for all i = 1, ..., n
    Check ranges of all partial derivatives, loading bound arrays L and U
    If F is monotone for all y_i
        Compute exact (sub)interval bounds using Equation 21
        Update current bounds
    Else if possible, bisect Ŷ into Ŷ_1 and Ŷ_2
        Add vectors Ŷ_1 and Ŷ_2 to evaluation queue
    Else (intervals are too small to bisect)
        Approximate (sub)interval bounds using F(Ŷ)
        Update current bounds
Return current bounds
```

Figure 2: Multi-Variable Interval Differential Algorithm

that the function rapidly decreases over some values of $\hat{x}_i$. Therefore, if the magnitude of the upper bound is greater than that of the lower, we know that the function increases faster than it decreases and assign $\hat{x}_{i,1}$ to $\hat{L}$ and $\hat{x}_{i,2}$ to $\hat{U}$. If the magnitude of the lower bound is the greater, then we assume that the function decreases faster than it increases and assign $\hat{x}_{i,2}$ to $\hat{L}$ and $\hat{x}_{i,1}$ to $\hat{L}$. In practice, this partitioning method works very well, efficiently yielding tight bounds that approximate the theoretical solution.

To illustrate this algorithm, consider $F(x_1, x_2, x_3) = x_1 + x_2/x_1 + x_3/x_2$ where $\hat{x}_1 = [0.5, 1.5]$, $\hat{x}_2 = [1, 2]$, $\hat{x}_3 = [4, 5]$, and the bisection limit is 3. The results of each stage of the computation are displayed in Table 3. As the table shows, the interval differential bounds $\mathcal{F}_I$ for this function are $[4.32, 7.64]$. The theoretical bounds $\hat{F}$ are $[4.83, 7.50]$ and the interval bounds $\mathcal{F}$ are $[3.17, 10.50]$. Clearly, this approach produced a significantly better approximation to $\hat{F}$ than $\mathcal{F}$ did.

Because we partition and group non-monotonic variables, the outer loop of our interval differential algorithm (see Figure 2) is executed a maximum of $2^{b+1} - 1$ times, where b represents the maximum number of bisections allowed. Therefore, the worst-case computational complexity of this algorithm is $O(2^{b+1}sn)$, where b is the bisection limit, s is the number of arithmetic operations in F, and n is the number of independent variables. Fortunately, the algorithm achieves fairly good results with small values of b. In this paper, all interval differential bounds were generated with $b = 3$.

$\hat{x}_1$	$\hat{x}_2$	$\hat{x}_3$	$\partial\mathcal{F}/\partial x_1$	$\partial\mathcal{F}/\partial x_2$	$\partial\mathcal{F}/\partial x_3$	Action	$\mathcal{F}_I$
[0.5, 1.5]	[1.0,2.0]	[4.0,5.0]	[-7.0,0.6]	[-4.3,1.0]	[0.5,1.0]	bisect	
[0.5, 1.0]	[1.0,1.5]	[4.0,5.0]	[-5.0,0.0]	[-4.0,0.2]	[0.7,1.0]	bisect	
[1.0, 1.5]	[1.5,2.0]	[4.0,5.0]	[-1.0,0.3]	[-1.6,0.0]	[0.5,0.7]	bisect	
[0.5, 1.0]	[1.0,1.25]	[4.0,5.0]	[-4.0,0.0]	[-4.0,-0.6]	[0.8,1.0]	compute	[5.45,7.50]
[0.5, 1.0]	[1.25,1.5]	[4.0,5.0]	[-5.0,-0.3]	[-2.2,0.2]	[0.7,0.8]	bisect	
[1.0, 1.25]	[1.5,2.0]	[4.0,5.0]	[-1.0,0.1]	[-1.4,0.0]	[0.5,0.7]	bisect	
[1.25, 1.5]	[1.5,2.0]	[4.0,5.0]	[-0.3,0.3]	[-1.6,-0.2]	[0.5,0.7]	bisect	
[0.5, 1.0]	[1.25,1.38]	[4.0,5.0]	[-4.5,-0.3]	[-2.2,-0.1]	[0.7,0.8]	compute	[5.28,7.00]
[0.5, 1.0]	[1.38,1.5]	[4.0,5.0]	[-5.0,-0.4]	[-1.6,0.2]	[0.7,0.7]	estimate	[4.54,7.64]
[1.0, 1.12]	[1.5,2.0]	[4.0,5.0]	[-1.0,-0.2]	[-1.3,0.0]	[0.5,0.7]	compute	[4.90,5.83]
[1.12, 1.25]	[1.5,2.0]	[4.0,5.0]	[-0.6,0.1]	[-1.4,-0.1]	[0.5,0.7]	estimate	[4.32,6.36]
[1.25, 1.38]	[1.5,2.0]	[4.0,5.0]	[-0.3,0.2]	[-1.5,-0.2]	[0.5,0.7]	estimate	[4.34,6.31]
[1.38, 1.5]	[1.5,2.0]	[4.0,5.0]	[-0.1,0.3]	[-1.6,-0.3]	[0.5,0.7]	estimate	[4.38,6.29]
TOTAL							**[4.32,7.64]**

Table 3: A Multi-Variable Interval Differential Bounds Computation

4. Improved Delay Estimates for Timing Analysis

Having described two general interval techniques for approximating the United Extension of a function, we now apply these methods to the timing analysis problem. For this experiment, we have chosen Crystal's PR-slope model [Oust83, Oust84, Oust86], but these interval techniques can be similarly applied to other timing analysis algorithms. We begin by briefly reviewing the PR-slope delay model.

4.1. The PR-slope Model

The purpose of the delay modeler in Crystal is to compute the signal delay through a chain of transistors (or *stage*). Each stage represents a path through the circuit originating at a strong signal *source* (Vdd or Gnd) and terminating at an output node or transistor gate (called the *target*). A stage is enabled by the last transistor in the path to switch on. This transistor is called the *trigger* for the stage. Given a stage description consisting of the sizes and types of the transistors in the stage, the parasitic resistances and capacitances of the nodes, the trigger transistor, and the waveform at the gate of the trigger transistor, the delay modeler generates the resulting waveform at the target of the stage.

To estimate delay, the PR-Slope model begins by calculating a resistance and capacitance for each node and transistor along the stage. In Crystal, a transistor is modeled as a perfect switch in series with a resistor. This resistance is fixed for non-trigger transistors and is determined by a table lookup on the transistor type and signal value. This factor is then multiplied by the length/width ratio of the transistor to generate an estimate of its effective resistance. Likewise, the effective capacitance of a transistor is computed by multiplying a type-dependent capacitance factor by the area of the transistor.

The effective resistance of the trigger transistor is more accurately modeled. This resistance is not a constant, but a function of the input waveform at the gate of the transistor. In particular, the timing analyzer combines the rise time at the transistor gate, the load being driven, and the transistor size into a single ratio (called the *rise time ratio*). It then consults a table indexed by these rise time ratios to find the resistance factor of the trigger transistor. This factor is multiplied by the aspect ratio of the transistor to determine its resistance. All transistor characterization tables used in these computations are generated by running SPICE simulations on sample circuits and compiling the results.

Having calculated the resistances and capacitances of each node and transistor in the path, Crystal uses these values to compute the total delay of the stage. The PR-Slope model employs the RC equations presented by Penfield and Rubinstein in [Rubi83]. Since a Crystal stage is just an RC line, these delay equations simplify to the following:

$$D = \sum_{t<i\leq T} \sum_{S<j\leq i} C_i R_j \tag{27}$$

where t represents the trigger transistor, T the target node, S the source node, R_j the resistance of element j, and C_i the capacitance at element i.

4.2. The Centred Version

First, we produce a Krawczyk's centred-form version of the PR-slope delay modeler. For this problem, the function $D(x_1, x_2, ..., x_n)$ is the delay function given in Equation 27 and the variables $x_1, x_2, ..., x_n$ are the resistance factors, capacitance factors, and transistor sizes needed to compute the delay. The transistor resistance and capacitance factors are predetermined by table lookup.[1]

[1] Because Crystal uses table lookup to determine the trigger transistor resistance factor, any dependency between this value and the other variables is hidden from the algorithm. Both the centred and interval differential methods model variable interdependence if all dependent values are explicitly expressed as functions of the independent variables. Because this is not the case in Crystal, we chose to precompute this factor and treat it as an independent variable for the delay computation.

Computing the centred form of this delay function is accomplished in two phases. First, we calculate the interval slope $\hat{g}_i$ for each variable x_i, then we use these interval slopes to find bounds on the delay. To calculate the interval slope, we require a function procedure for computing D. To create this function procedure, we encode the delay equation within a computer program and employ it to compute the interval slope vector G. Once we have determined G, we use Equation 8 to calculate the final delay bounds.

4.3. The Interval Differential Version

To compute interval differential delay bounds $\mathcal{D}_I$, we must first create an interval differential version of the delay function D. This is accomplished by replacing each arithmetic operator $+$, $-$, $\cdot$, and $/$ in D by its interval differential equivalent specified in Equations 22 through 25 and each variable x_i in D by an interval operand $\hat{X}_i$ indicating its value and derivative. When evaluated, this new function returns the interval partial derivative $\partial\mathcal{F}/\partial x_i$ of each x_i. Then we follow the algorithm given in Figure 2 to compute delay bounds.

5. Performance

We implemented both centred-form ($\mathcal{D}_C$) and interval differential ($\mathcal{D}_I$) versions of Crystal's PR-Slope model. To test these methods, we compared them against the original interval algorithm ($\mathcal{D}$), a Monte Carlo version (D with 10,000 trials), and the nominal delay (D using only the expected values of the variables). To select variable values for each Monte Carlo trial, we employed a random number generator with a uniform distribution. For the interval differential version, the maximum number of bisections was set to three. All implementations were written in the C programming language and run on a Sun Sparcstation. In this section, we present an analysis of the performance of each algorithm.

5.1. Quality of the Bounds

To determine the quality of the bounds, we conducted a number of experiments comparing the generated bounds to the theoretical bounds ($\hat{D}$) for a given circuit.[2] Figures 3 and 4 illustrate the relationship between delay and stage length (the number of transistors in the stage). These graphs display the percentage error for both the upper and lower bounds generated by each method. As

[2]For these experiments, the theoretical bounds were determined by fixing as many of the variables as possible and computing the United Extension. As with all of these methods, the trigger transistor resistance was precomputed and treated as an independent variable in the delay function.

these graphs demonstrate, the Monte Carlo and nominal value techniques always underestimate the bounds, while the interval approaches always overestimate them. Of all the methods, the interval differential version $\mathcal{D}_I$ produces the best bounds, coming within about 4% of the desired result. Note that the centred form $\mathcal{D}_C$ generates a skewed interval, producing a tight upper bound and a poor lower bound. Since centred-forms yield an interval centered around the nominal value, we conclude that the nominal delay does not fall in the middle of the true bounds, thus causing the skewed centred values that we observe. The original interval method produces bounds within 8% of the theoretical values and the Monte Carlo technique comes within 12%. The nominal delay produces the poorest estimate of best- and worst-case delay. For most of these methods, the error grows with the length of the stage. Fortunately, the stages encountered in most circuits are typically small, containing fewer than five transistors.

In addition to the above experiments, we also analyzed a portion of the Brown Systolic Array (B-SYS)[Hugh89], a 2-μ CMOS design. We extracted a 2,000 transistor piece of the circuit and computed its critical path. This path consists of nine stages, each ranging in length from one to three transistors. Table 4 displays the delay bounds generated by each method for the entire critical path and for each individual stage in the path. Table 5 gives the percentage error in the bounds for each method. As the tables shows, the interval differential version ($\mathcal{D}_I$) produces the tightest delay bounds, achieving the theoretical bounds for all stages. Because the function is monotone over the ranges of all variables, this interval technique can precisely calculate the bounds. As in the previous experiments, the centred form algorithm ($\mathcal{D}_C$) produces very tight upper bounds (matching the theoretical ones), but poorer lower bounds (only coming within 2.3% of the desired result). On average, the interval method comes within 0.3% of both upper and lower bounds. The Monte Carlo method produces an average error of about 4.3%, while the nominal delay has an average error of about 15%.

5.2. Speed of the Algorithm

The graph in Figure 5 depicts the relative speed of each of the implementations. As expected, the Monte Carlo method is the least efficient of the bounding techniques. Of the interval-based techniques, the original interval algorithm is the most efficient. As stated in [Hark89], this method runs roughly four times slower than the non-interval PR-slope model in Crystal and has a time complexity of $O(n)$, where n is the number of independent variables.

The centred-form method is the next fastest interval method. As previously stated, its operating speed is $0(sn)$, where s is the number of instructions in the function procedure for D. For Crystal's delay equation, s is proportional to the number of variables; therefore, the operating speed of the

Stage	Size	Theoretical		Centred		Differential		Interval		Monte Carlo		Nom
		min	max	min	max	min	max	min	max	min	max	
1	2	6.8	9.3	6.7	9.3	6.8	9.3	6.8	9.3	7.2	8.7	8.0
2	3	54.3	72.9	53.1	72.9	54.3	72.9	54.1	73.1	56.6	69.5	63.0
3	2	10.3	14.1	10.1	14.1	10.3	14.1	10.3	14.1	10.8	13.4	12.1
4	2	10.6	14.4	10.4	14.4	10.6	14.4	10.6	14.4	11.0	13.9	12.4
5	2	1.2	1.6	1.1	1.6	1.2	1.6	1.2	1.6	1.2	1.5	1.4
6	1	0.7	1.0	0.7	1.0	0.7	1.0	0.7	1.0	0.8	0.9	0.8
7	1	0.8	1.1	0.7	1.1	0.8	1.1	0.8	1.1	0.8	1.0	0.9
8	3	20.4	27.4	19.9	27.4	20.4	27.4	20.3	27.5	21.2	26.5	23.7
9	1	2.0	2.7	1.9	2.7	2.0	2.7	2.0	2.7	2.0	2.6	2.3
CPath		107.1	144.5	104.6	144.5	107.1	144.5	106.8	144.8	111.7	138.1	124.5

Table 4: A B-SYS Critical Path Delay Computation (ns)

Stage	Size	Centred		Differential		Interval		Monte Carlo		Nominal	
		min	max	min	max	min	max	min	max	min	max
1	2	2.3	-0.1	0	0	0.3	-0.3	-5.9	5.9	-16.7	14.0
2	3	2.3	0	0	0	0.3	-0.3	-4.2	4.7	-16.0	13.6
3	2	2.4	-0.1	0	0	0.1	-0.1	-4.8	5.3	-17.0	14.3
4	2	2.4	0	0	0	0	-0.1	-3.8	3.7	-16.8	14.1
5	2	3.5	0	0	0	0.9	-0.6	-5.2	5.7	-18.1	15.4
6	1	2.9	0	0	0	0	0	-7.1	5.1	-18.6	15.3
7	1	3.9	0	0	0	0	0	-3.9	4.6	-19.5	15.6
8	3	2.3	-0.1	0	0	0.4	-0.4	-4.1	3.1	-16.0	13.6
9	1	2.0	0	0	0	0	0	-1.5	1.5	-16.2	13.6
CPath		2.3	0	0	0	0.2	-0.3	-4.3	4.4	-16.3	13.8

Table 5: % Error in Delay Bounds for the Critical Path

centred-form delay algorithm ($\mathcal{D}_C$) is $O(n^2)$.

The interval differential method is the slowest of the interval algorithms. Since it computes derivatives for each of the n variables, this method requires $O(n^2)$ time for each function evaluation. If the function is monotone over the range of all variables, then the algorithm evaluates this function only once. If, however, the function is non-monotone, then we must bisect the intervals and evaluate the function over the subintervals. In the worst-case, the algorithm may evaluate the function $2^{b+1} - 1$ times, where b represents the maximum number of interval bisections allowed. Therefore, this algorithm has a worst-case operating speed of $O(2^{b+1}n^2)$.

6. Summary

In this paper, we presented two interval-based methods that improve on our previously-published technique for modeling uncertainty in RC timing analysis. The first of these uses Krawczyk's centred form, while the second employs an extension of Rall's interval differential calculus.

A comparison of the three interval-based approaches reveals a classic trade-off between efficiency and precision. Our experiments demonstrated that both the centred and interval differential algorithms produce tighter bounds than the original interval method. The primary reason for this is that the new methods take into account variable interdependence, while the original does not. The original, however, requires less time to compute its bounds.

The centred method is the second fastest interval approach, but generates bounds of inconsistent quality. In all of our experiments, this algorithm yielded a tight upper bound and a poor lower bound. These skewed bounds may be attributed to the fact that the nominal delay did not fall in the center of the true interval. Since the resulting centred interval is centered at this nominal value, the algorithm must overestimate the lower bound in order to encompass the theoretical upper bound.

In general, the interval differential approach is the slowest of the algorithms, but generates the tightest bounds. In our experiments, this algorithm produced bounds within about 4% of the true results using at most three bisections. It is also the only one of the three methods that provides a measure of the goodness of its estimate; the number of estimated subinterval bounds indicates the precision of the result.

All three interval-based approaches are much more efficient than Monte Carlo simulation. In experiments with 10,000 Monte Carlo trials, all three also generated significantly tighter bounds.

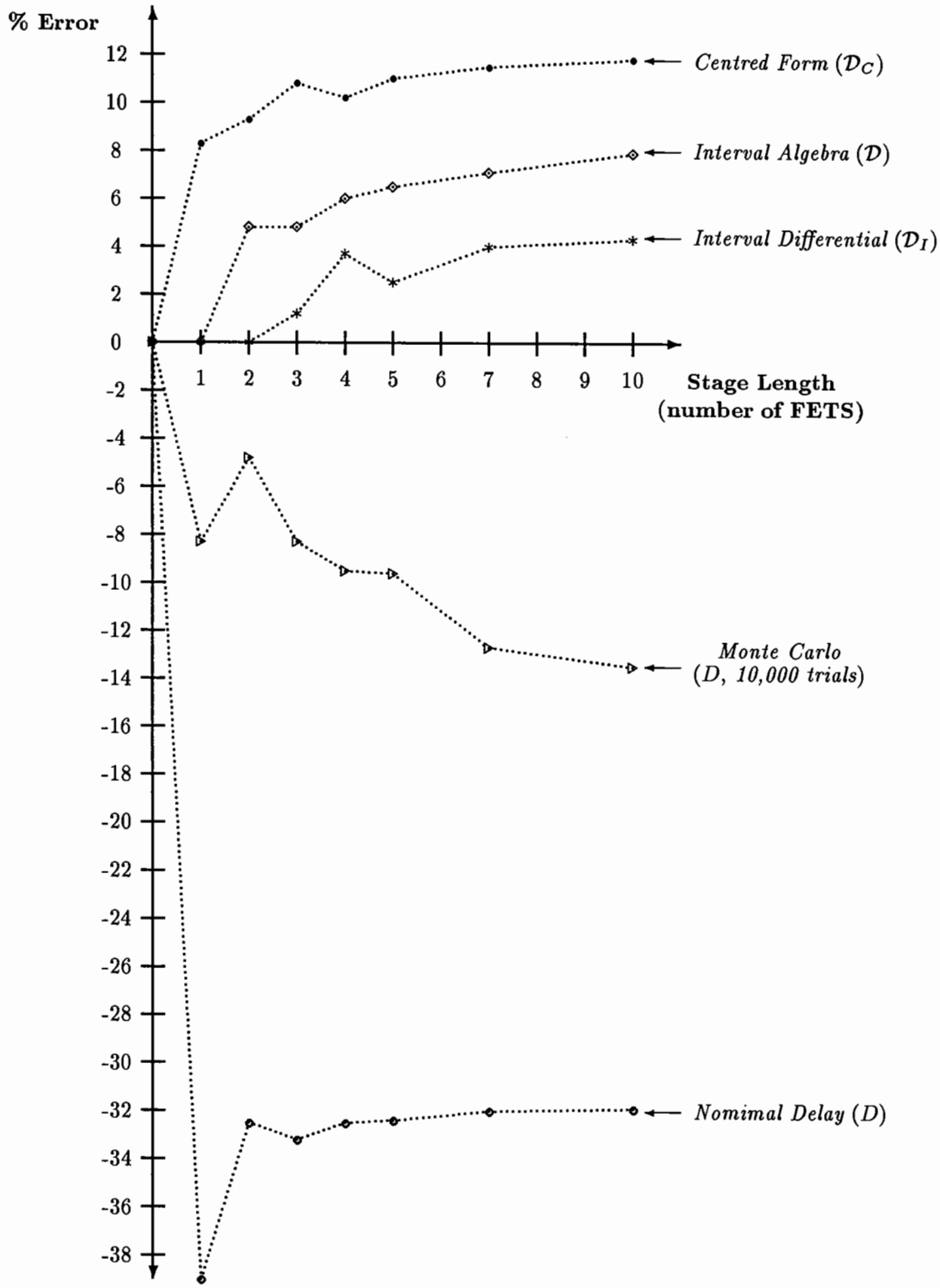

Figure 3: %Error vs. Stage Length (Lower Bounds)

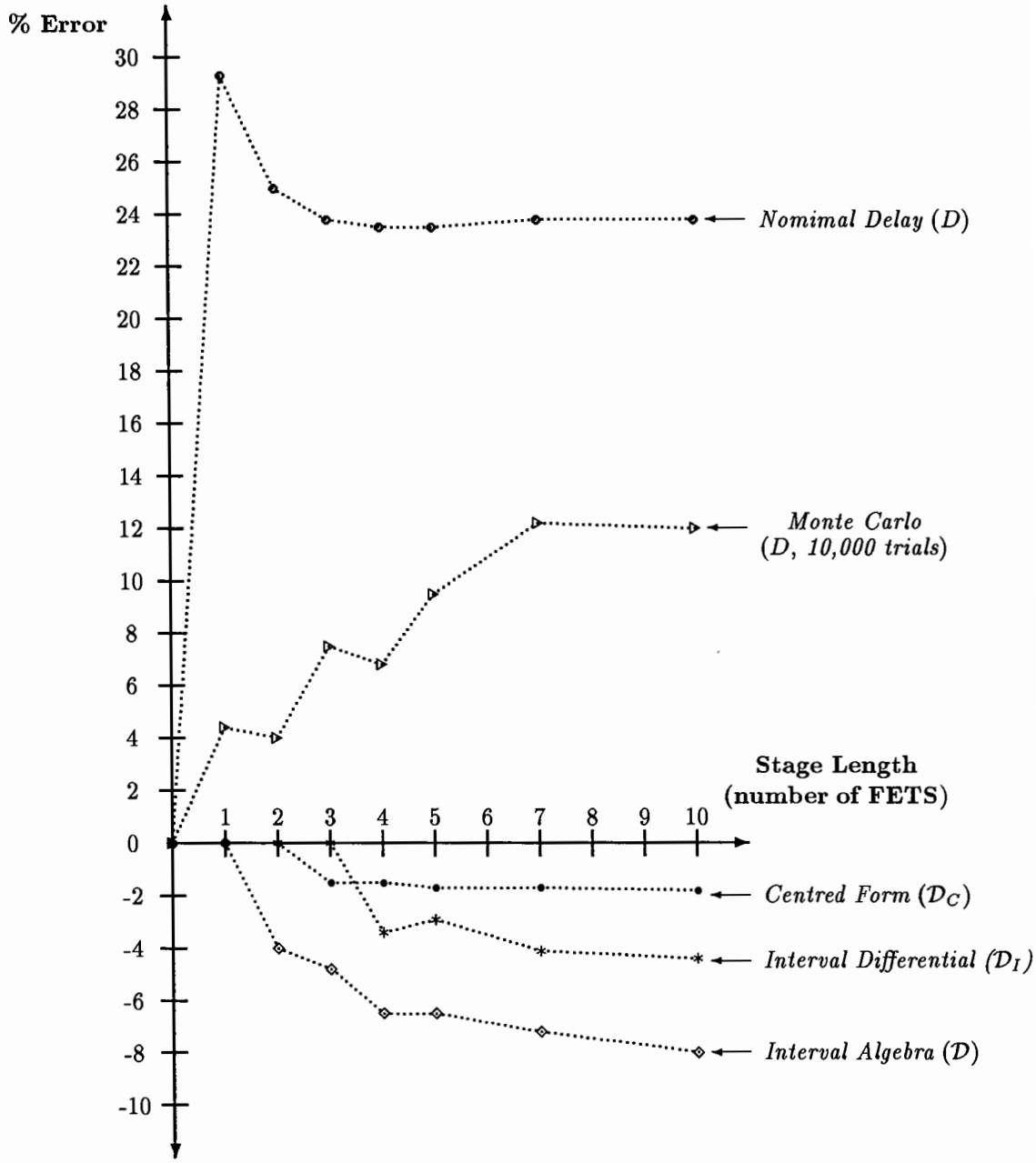

Figure 4: %Error vs. Stage Length (Upper Bounds)

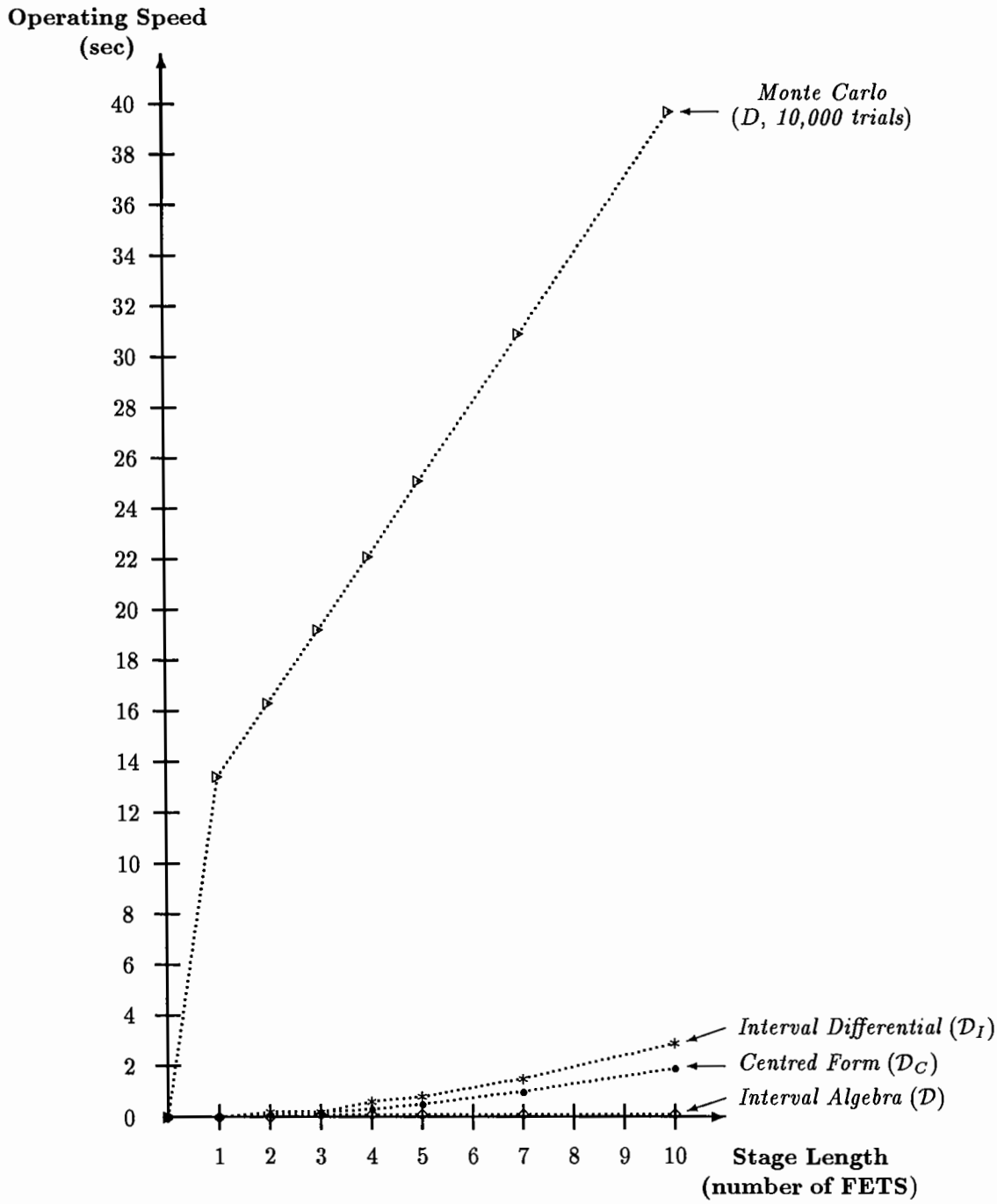

Figure 5: Operating Speed vs. Stage Length

Appendix A. The Interval Differential Operations

In this section, we present the C language code for the interval differential operations defined in Section 3.2. These operations use the interval algebraic operators ADD, SUB, MULT, and DIV to perform their calculations. They also employ the following definitions of an *interval* and an *interval operand*:

```
typedef struct
  {
  float min;          /* the lower bound of the range */
  float max;          /* the upper bound of the range */
  }  INTERVAL;


typedef struct
  {
  INTERVAL value;     /* the value of the operand */
  INTERVAL *deriv;    /* the vector of partial derivatives */
  }  IOPERAND;
```

A.1. Interval Differential Addition

Given two interval operands, this function computes and returns the interval sum of the operands and the corresponding interval partial derivatives for all variables.

```
IOPERAND DADD (o1, o2)
  IOPERAND o1;
  IOPERAND o2;
  {
  IOPERAND result;
  int i;
  INTERVAL ADD();

  result.value = ADD(o1.value, o2.value);       /* compute the interval sum */
  create_deriv_array(&(result.deriv));          /* create derivative array */
  for (i = 0; i < numvariables; ++i)            /* compute partial derivatives */
    result.deriv[i] = ADD(o1.deriv[i], o2.deriv[i]);
  return(result);                               /* return the result */
```

```
}
```

A.2. Interval Differential Subtraction

Given two interval operands, this function computes and returns the interval difference of the operands and the corresponding interval partial derivatives for all variables.

```
IOPERAND DSUB (o1, o2)
  IOPERAND o1;
  IOPERAND o2;
  {
  IOPERAND result;
  int i;
  INTERVAL SUB();

  result.value = SUB(o1.value, o2.value);      /* compute the interval difference */
  create_deriv_array(&(result.deriv));         /* create derivative array */
  for (i = 0; i < numvariables; ++i)           /* compute partial derivatives */
    result.deriv[i] = SUB(o1.deriv[i], o2.deriv[i]);
  return(result);                              /* return the result */
  }
```

A.3. Interval Differential Multiplication

Given two interval operands, this function computes and returns the interval product of the operands and the corresponding interval partial derivatives for all variables.

```
IOPERAND DMULT (o1, o2)
  IOPERAND o1;
  IOPERAND o2;
  {
  IOPERAND result;
  int i;
  INTERVAL ADD(), MULT();

  result.value = MULT(o1.value, o2.value);     /* compute the interval product */
  create_deriv_array(&(result.deriv));         /* create derivative array */
```

```
  for (i = 0; i < numvariables; ++i)          /* compute partial derivatives */
    result.deriv[i] = ADD(MULT(o1.value, o2.deriv[i]),
                          MULT(o2.value, o1.deriv[i]));
  return(result);                             /* return the result */
  }
```

A.4. Interval Differential Division

Given two interval operands, this function computes and returns the interval quotient of the operands and the corresponding interval partial derivatives for all variables.

```
IOPERAND DDIV (o1, o2)
  IOPERAND o1;
  IOPERAND o2;
  {
  IOPERAND result;
  int i;
  INTERVAL SUB(), MULT(), DIV();

  result.value = DIV(o1.value, o2.value);     /* compute the interval quotient */
  create_deriv_array(&(result.deriv));        /* create derivative array */
  for (i = 0; i < numvariables; ++i)          /* compute partial derivatives */
    result.deriv[i] = DIV(SUB(o1.deriv[i],
                              MULT(result.value, o2.deriv[i])),
                          o2.value);
  return(result);                             /* return the result */
  }
```

References

[Hark89] Harkness, Cheryl L. and Daniel P. Lopresti. "Modeling Uncertainty in RC Timing Analysis." *International Conference on Computer-Aided Design.* Santa Clara, California, November 1989, pp. 516-519.

[Hugh89] Hughey, Richard and Daniel Lopresti. "The B-SYS Programmable Systolic Array." Department of Computer Science, Brown University, Report No. CS-89-32, 1989.

[Moor79] Moore, Ramon E. *Methods and Application of Interval Analysis.* Philadelphia: SIAM, 1979.

[Oust83] Ousterhout, John K. "Crystal: A Timing Analyzer for nMOS VLSI Circuits." *Third Caltech Conference on Very Large Scale Integration.* Ed. Randal Bryant. Computer Science Press, 1983, pp. 57-70.

[Oust84] Ousterhout, John K. "Switch-Level Delay Models for Digital MOS VLSI." *21st IEEE Design Automation Conference.* 1984.

[Oust86] Ousterhout, John. "Using Crystal for Timing Analysis." *1986 VLSI Tools.* Report No. UCB/CSD 86/272. University of California at Berkeley. 1986.

[Rubi83] Rubinstein, Jorge, Paul Penfield Jr. and Mark A. Horowitz. "Signal Delay in RC Tree Networks." *IEEE Transactions on Computer-Aided Design.* Vol. CAD-2, No. 3, July 1983, pp. 202-211.

[Rall86] Rall, L.B. "Improved Interval Bounds for Ranges of Functions." *Interval Mathematics 1985.* Ed. K. Nickel. Berlin: Springer-Verlag, 1986, pp. 143-155.

[Rats84] Ratschek, H. and J. Rokne. *Computer Methods for the Range of Functions.* Chichester: Ellis Horwood Limited, 1984.

[Sobo75] Sobol, I.M. *The Monte Carlo Method.* Trans. by V.I. Kisin. Mir Publishers: Moscow, 1975.