Performance Analysis of Embedded Software Using Implicit Path Enumeration

Yau-Tsun Steven Li and Sharad Malik, *Member, IEEE*

***Abstract*— Embedded computer systems are characterized by the presence of a processor running application-specific dedicated software. A large number of these systems must satisfy real-time constraints. This paper examines the problem of determining the extreme (best and worst) case bounds on the running time of a given program on a given processor. This has several applications in the design of embedded systems with real-time constraints. An important aspect of this problem is determining which paths in the program are exercised in the extreme cases. The state-of-the-art solution here relies on an *explicit* enumeration of program paths. This solution runs out of steam rather quickly since the number of feasible program paths is typically exponential in the size of the program. We present a solution for this problem that does not require an explicit enumeration of program paths, i.e., the paths are considered *implicitly*. This solution is implemented in the program `cinderella` (in recognition of her hard real-time constraint—she had to be back home at the stroke of midnight), which currently targets a popular embedded processor—the Intel i960. The preliminary results of using this tool are also presented here.**

I. Introduction

A. Motivation

EMBEDDED computer systems are characterized by the presence of a processor running application-specific dedicated software. Recent years have seen a large growth of such systems. In particular, "system on a chip" is becoming an important implementation technology. These systems integrate an embedded processor, memory, peripherals, and a gate array application-specific integrated circuit on a single integrated circuit. An important factor leading to their growth is the migration from application-specific logic to application-specific code running on existing processors. This migration is driven by two distinct forces. The first is the increasing cost of setting up a fabrication line for semiconductor vendors. At over $1 billion for a new line, the only components that make this affordable are high-volume parts such as processors, memories, and possibly field programmable gate arrays. Application-specific logic is getting increasingly expensive to manufacture and is the solution only when speed constraints rule out programmable solutions. The second force comes from the application houses, which are facing increasing pressures to reduce the time to market as well as to have predictable schedules. Both of these challenges can be better met with software programmable solutions made possible by embedded systems. Thus, we are seeing a movement from the logic gate's being the basic unit of computation on silicon to an instruction's running on an embedded processor. This has motivated our research efforts directed toward examining analysis and optimization problems for embedded software.

This paper examines the problem of determining the extreme (best and worst) case bounds on the running time of a given program on a given processor. This has several applications in the design of embedded systems.

- In hard real-time systems, the response time of the system must be strictly bounded to ensure that it meets its deadlines.
- These bounds are required by schedulers in real-time operating systems. Numerous scheduling methods have been based on the *rate monotonic scheduling algorithm* first proposed by Liu and Layland [1]. These depend on a measure of the deterministic computation requirement of each task.
- The selection of the partition between hardware and software, as well as selection of the hardware components, is strongly driven by the performance analysis of software. For example, in a high-performance engine controller design, the designer may choose to use an expensive MC68040 if he cannot prove that a less expensive MC68030 can always compute the engine control parameters within a single rotation.

B. Problem Statement

A more precise statement of the problem addressed in this paper is as follows. We need to bound (lower and upper) the running time of a given program on a given processor assuming uninterrupted execution. The term "program" here refers to any sequence of code and does not have to include a logical beginning and an end. The term "processor" here includes the complete processor and memory system.

The running time of a program may vary according to different input data and initial machine state. Suppose that of all the possible running times, $T_{\min}$ and $T_{\max}$ are the minimum and maximum of these times, respectively. We define the *actual bound* of the program as the time interval $[T_{\min}, T_{\max}]$. Our objective is to find a correct estimate of this bound without introducing undue pessimism. The estimated time interval $[t_{\min}, t_{\max}]$, defined as the *estimated bound*, must enclose the actual bound. This is illustrated in Fig. 1.

Manuscript received June 7, 1995; revised October 17, 1996. This paper was recommended by Associate Editor R. Camposano.

Y.-T. S. Li is with Monterey Design Systems, Inc., Sunnyvale, CA 94089 USA.

S. Malik is with the Department of Electrical Engineering, Princeton University, Princeton, NJ 08544 USA.

Publisher Item Identifier S 0278-0070(97)09362-7.

0278–0070/97$10.00 © 1997 IEEE

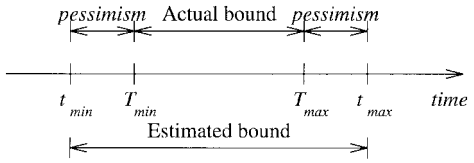


Fig. 1. The estimated bound $[t_{\min}, t_{\max}]$ of a program must always bound its actual bound $[T_{\min}, T_{\max}]$.

C. Subproblems

There are two components to the prediction of extreme case performance. To predict the performance of a given piece of software on a given processor, we must:

- determine what sequence of instructions will be executed in the extreme case—this is referred to as the *program path analysis problem;*
- compute how much time it will take the system to execute that sequence—this requires a modeling of the host processor system, which is referred to as *microarchitectural modeling*.

Both these aspects need to be studied well in order to provide a solution to this problem. In our research, we have attempted to isolate these aspects as far as possible in an attempt clearly to understand each problem. The focus of this paper is on the program path analysis problem.

II. Previous Work

A static analysis of the code is needed to see what the possible extreme case paths through the code are. Puschner and Koza [2] observed that this problem is undecidable in general and equivalent to the halting problem. In practice, however, designers select algorithms and data structures for embedded programs that they believe to be easily bounded. Since the programmer must prove to himself that a real-time program will terminate, a restricted programming style is typically used. The above researchers, as well as Kligerman and Stoyenko [3], have suggested restrictions on programs that make this problem decidable. These are absence of dynamic data structures, such as pointers and dynamic arrays; absence of recursion; and bounded loops. These restrictions may be imposed through either specific language constructs or programmer annotations on conventional programs. While specific language constructs, such as those provided in Real-Time Euclid [3], are useful inasmuch as they provide checks for the programs, they come with the usual costs associated with a new programming language. An entire new set of software development tools needs to be developed, and programmers need to be trained in the new language, both of which are costly propositions. In the absence of optimizing compilers for the new language, it is likely that the code quality will be inferior in comparison to that produced by the vast array of compilers that exist for the established programming languages. Thus, predictability, in this case, comes at the cost of performance. This tradeoff is not needed, since the restrictions can be enforced by a mechanism external to the language. Several researchers have suggested the use of annotations to existing programs that enforce these conditions. Mok *et al.* [4], Puschner and Koza [2], and Park and Shaw [5] all provide for annotations of programs that fix the bounds on loops. We believe that this approach is more practical since it involves only minimal additional programming tools.

There is some debate over where the analysis should be done—at the programming language level or the assembly language level. Providing for analysis at the level of the programming language makes it largely independent of the target processor. Initial research efforts by Shaw in this direction resulted in a "schema" wherein time bounds were constructed for each high-level language construct using time bounds on its constituent parts. Subsequently, these bounds were used to provide bounds for entire programs. Subsequent work by Shaw and his coworkers demonstrated the inadequacy of this approach, since it was difficult to predict the bounds for a high-level language construct independent of the context in which it appeared and independent of the compiler and target processor. They augmented their initial solution by providing limited interaction with assembly code to take into account the effects of program context, compilers, and the target processor. In contrast, Mok *et al.* [4] use the high-level program description to provide functional information about the program through annotations, which are then passed on to the assembly language program. Last, it is the assembly language program that is analyzed to determine the actual bounds on the entire program. We believe that this is the correct approach; the high-level language program is the right place to provide useful annotations, since that is what the programmer directly sees. However, the final analysis must be performed on the assembly language program in order to capture all the effects of the actual microarchitectural implementation. Furthermore, aggressive optimizing compilers transform programs so aggressively that high-level language constructs no longer maintain a direct correspondence with executed code.

The functionality of the program determines the actual paths taken during its execution. Any information regarding this functionality helps in deciding which program paths are feasible and which are not. While some of this information can be automatically inferred from the program, it is widely felt that this is a difficult task in general. In contrast, it is relatively easier for the programmer to provide such information since he is familiar with what the program is supposed to do.[1] Initial efforts to include this information were restricted to providing annotations about loop bounds and the maximum execution counts of a given statement within a given scope. These were provided through program annotations [2] or annotations in a separate timing analysis program [4]. Both of these mechanisms are equivalent in terms of the information that they provide for analysis. This information, albeit useful, is very limited in terms of describing what can and cannot happen during the actual program execution. In particular, it does not capture any information about the functional interactions between different parts of the program.

[1] This has an analog in the domain of digital circuits. There, designer annotations were commonly used to mark paths in the digital circuit that were never exercised [6]. These paths were then eliminated from consideration in the timing analysis of the circuit.

```
for (i=0; i<100; i++) {
  if (rand() > 0.5)
    j++;
  else
    k++;
}
```

Fig. 2. Exponential blowup of paths: an example.

Subsequent work by Park and Shaw [5] in this area attempts to overcome this limitation. Here, they recognize the fact that the set of all possible path sequences through the program can be expressed as a regular expression. They then propose techniques for representing this set of regular expressions using a shorthand notation. This is considered to be too difficult for use by programmers, and a language [Information Description Language (IDL)] is provided for the users by which they can specify most, though not all, path traces that can actually happen. This is then used to eliminate from consideration all possible path sequences that can never occur during the analysis stage. All the paths that are determined to be feasible by this analysis are then examined *explicitly* to determine the best and worst case paths.

There are several important contributions made by this research. The first is recognizing that it is important to provide information about how different parts of the program interact with each other. The second is to recognize that regular expressions are capable of describing all possible path sequences that can and cannot occur. However, there are some drawbacks that arise due to expressing feasible sequences as regular expressions. First, as the authors admit, these are not amenable for specification by programmers. The language interface provided to the programmer is an exercise in compromise, giving up full generality for ease of use and analysis. Next, the complexity of computations of negation and intersection for regular expressions may be prohibitive, resulting in the need for approximate solutions. Last, the need to explicitly examine all possible feasible paths very often results in an exponential blowup since the number of these paths is typically exponential in the size of the program. The small example shown in Fig. 2 illustrates this point. Here, the function `rand()` generates a random number in the range [0, 1]. The loop has 2^{100} possible paths depending on the which branch is taken in each iteration of the loop. In fact, if the timing cost to increment j is equal to the timing cost to increment k, all of these paths are worst case paths. Such scenarios are common in programs, and any analysis technique that explicitly enumerates paths tends to have limited application.[2]

The main contribution of this paper is to provide a method that *does not* explicitly enumerate program paths but rather *implicitly* considers them in its solution. This is accomplished by converting the problem of determining the bounds to one of solving a set of integer linear programming (ILP) problems. While each ILP problem can in the worst case take exponential time, in practice, exponential blowup never occurred in our experiments. We observed that the actual computation done by the ILP solver is solving a single linear program. The reasons for this will be examined in Section III-C, and practical data supporting this will be presented in Section VI.

[2]Again, in the domain of digital circuits, the initial timing analysis algorithms that considered the functionality of the circuit elements enumerated paths explicitly. More recent work there shows how analysis can be done by an implicit consideration of all circuit paths [7].

III. ILP Formulation

A. Objective Function

The following observation helps us avoid an explicit enumeration of paths to determine the best and worst case. Our objective is to determine the extreme case running times and not necessarily actually identify the extreme case paths. This observation led to the following formulation of the problem. For the rest of this section, the focus will be the worst case timing; the best case can be obtained analogously. Let x_i be a variable denoting the number of times basic block B_i is executed when the program is executed once. Clearly, the basic block variable x_i must be a positive integer. A basic block of code is a maximal sequence of instructions for which the only entry point is the first instruction and the only exit point is the last instruction. Let c_i be the running time (or cost) of this basic block in the worst case. For now let us assume that c_i is constant over all possible times this basic block is executed and also that there are practically feasible techniques for determining this. This issue will be examined in more detail in Section IV. Thus, for a program with N basic blocks, its worst case timing is given by the maximum value of the following linear expression:

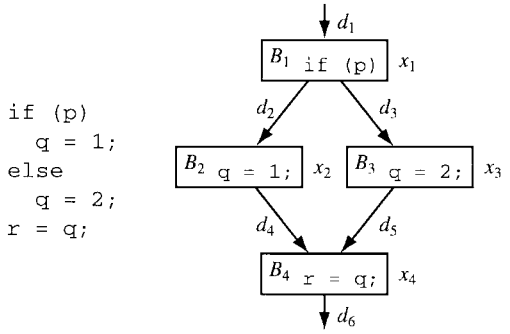
$$\sum_{i=1}^{N} c_i x_i. \tag{1}$$

Without any additional information, the maximum value of this expression is ∞ since any x_i can take on the value of ∞. This is precisely why this problem is undecidable in general. As pointed out in Section II, however, restrictions must be imposed on the programs if we hope to be able to bound the running times. One of the restrictions is that the bounds on the number of iterations of each loop must be specified. This directly provides an upper bound for each x_i that can be used in the above expression. The bound obtained by this method will typically be very loose, however, since in general the upper bound for each x_i is rarely achieved simultaneously.

There are two factors that may preclude the upper bounds from being achieved simultaneously. These are illustrated by the following example code fragment:

```
for (i = 0; i < k; i++) {
  if(OK)
    do_something();
  else {
    do_something_else();
    OK = false;
  }
}.
```

The first factor is that the program structure itself will impose conditions that make this impossible. In our example, the loop has an upper bound of k iterations. The loop body itself



(a) (b)

Fig. 3. An example of the `if-then-else` statement and its CFG.

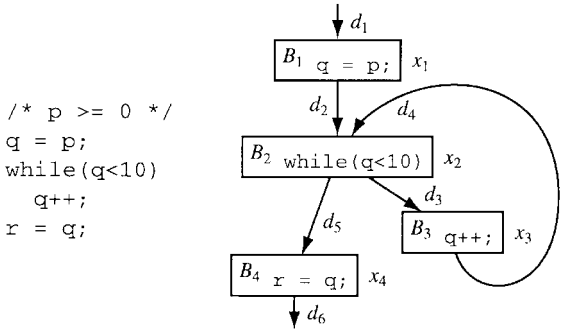


Fig. 4. An example of the `while`-loop statement and its CFG.

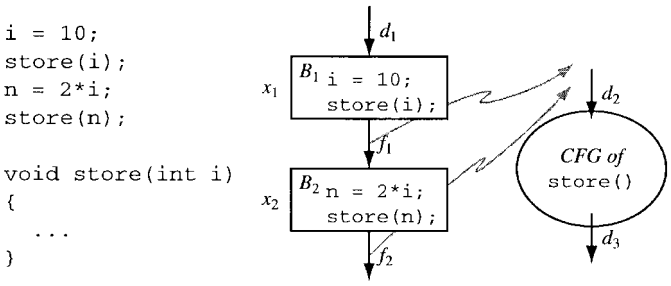


Fig. 5. An example showing how function calls are represented.

consists of a single `if-then-else` statement. If we ignore the functionality of the program, in the worst case the `then` part can have k iterations, as can the `else` part. However, the mutual exclusion of the `if-then-else` prevents this from happening simultaneously. Thus the *program structure* imposes conditions on what can and cannot happen in terms of execution counts of basic blocks. The second factor that comes into play is the *program functionality*. This deals with what the program is computing. In the above example, the execution of the `else` part sets a condition that precludes the `else` part from being executed again. Thus, the execution count of the `else` part is at most one. This type of information is not easily obtained from the program in most cases but is something that the writer of the program can provide with relatively less difficulty. So we need to maximize (1) while taking into account the restrictions imposed by the program structure and functionality.

B. Linear Constraints

Since (1) is a linear expression, it would be nice if we could state the program structure and functionality restrictions in the form of linear constraints. This will enable us to use ILP to determine the maximum value of the expression. In this section, we demonstrate how both the program structure and the program functionality restrictions can be specified in the form of sets of linear constraints.

1) Program Structural Constraints: Since these constraints arise from the program control flow graph (CFG) [8], they can be automatically extracted from the CFG. This is illustrated in Fig. 3 for a program fragment containing an `if-then-else` statement. We label the edges and the basic blocks by variables d_i and x_i, respectively [Fig. 3(b)]. These variables represent the number of times that the program control is passing through those edges and basic blocks when the program is executed. Here, for illustration purpose, the code in each node of the CFG is shown in the source level code. In actual implementation, the assembly code is analyzed to construct the CFG. Each node contains a sequence of assembly instructions.

The constraints can be deduced from the CFG as follows. At each node, the execution count of the basic block must be equal to both the sum of the control flow going into it and the sum of the control flow going out from it. Thus, from the graph, we have the following constraints:

$$x_1 = d_1 = d_2 + d_3 \tag{2}$$
$$x_2 = d_2 = d_4 \tag{3}$$
$$x_3 = d_3 = d_5 \tag{4}$$
$$x_4 = d_4 + d_5 = d_6. \tag{5}$$

Fig. 4 shows a `while`-loop statement and its CFG. The constraints extracted from this CFG are

$$x_1 = d_1 = d_2 \tag{6}$$
$$x_2 = d_2 + d_4 = d_3 + d_5 \tag{7}$$
$$x_3 = d_3 = d_4 \tag{8}$$
$$x_4 = d_5 = d_6. \tag{9}$$

Note that the above constraints do not contain any loop count information. This is because the loop count information depends on the values of the variables, which are not tracked in the CFG. However, the loops can be detected and marked. After all structural constraints are constructed, the user is asked to provide the loop bounds as part of specifying the program functionality constraints (see Section III-B2).

The function calls are represented by using the f-edges in the CFG as shown in Fig. 5. An f-variable is similar to a d-variable, except that its edge contains a pointer pointing to the CFG of the function being called.

The construction of the structural constraints in the caller function remains the same. In the example, the structural constraints of the caller function are

$$x_1 = d_1 = f_1 \tag{10}$$
$$x_2 = f_1 = f_2. \tag{11}$$

```
1:        check_data()
2:        {
3:          int i, morecheck, wrongone;

4:   x1     morecheck = 1; i = 0; wrongone = -1;

5:          while (morecheck) {
6:   x2       if (data[i] < 0) {
7:   x3         wrongone = i; morecheck = 0;
8:           }
9:           else
10:  x4         if (++i >= DATASIZE)
11:  x5           morecheck = 0;
12:  x6     }

13:  x7     if (wrongone >= 0)
14:  x8       return 0;
15:         else
16:  x9       return 1;
17:       }
```

Fig. 6. `check_data` example from Park's thesis. Basic block variables (x_i) are labeled alongside with the source code.

The number of times that the function is executed can be tracked by knowing the f-edges pointing to it. Here, this information is represented by

$$d_2 = f_1 + f_2 \tag{12}$$

where d_2 represents the count of the starting edge of the callee function `store()`'s CFG. For the main function, since it is only executed once, the count of its starting edge must be equal to one. Therefore, if variable d_1 represents the count of this edge, the following constraint is constructed:

$$d_1 = 1. \tag{13}$$

2) Program Functionality Constraints: As described previously, these constraints are used to denote loop bounds and other path information that depend on the functionality of the program. Currently, the user of `cinderella` (typically the author of the program being analyzed) is expected to provide these constraints. In the future, we envisage some of this being done automatically by using data flow analysis. We illustrate the use of these constraints to capture conditions on feasible program paths with the example (Fig. 6) taken from Park's thesis [5].

Function `check_data()` checks the values of the array `data[]`. If any of the values are less than zero, the function will stop checking and return zero immediately; otherwise, it will return one.

The loop count of the `while`-loop in the function is bounded by one and `DATASIZE`, i.e., the loop will be executed at least once and at most `DATASIZE` times. Suppose that `DATASIZE` is previously defined as a constant value ten. Then to specify this loop bound information, the following two constraints are used:

$$1x_1 \le x_2 \tag{14}$$

$$x_2 \le 10x_1. \tag{15}$$

```
          check_data()
          { ...
   x7       if (wrongone >= 0)
   x8         return 0;
            else
   x9         return 1;
          }

          task()
          { ...
   f10    status = check_data();
   x11    if (!status)
   f12      clear_data();
          ...
          }
```

Fig. 7. An example showing how the path relationship between the caller and callee functions can be specified.

Here, x_1 is the count for the basic block just before entering the loop and x_2 is the count for the first basic block of the loop body. These two variables can be determined automatically from the CFG. The user only needs to provide the values one and ten.

The minimum user information required to perform timing analysis is the loop bound information. After that, the user can provide additional information so as to tighten the estimated bound. For example, we see that inside the loop, lines 7 and 11 are mutually exclusive, and either of them is executed at most once. This information can be represented by the following user constraint:

$$(x_3 = 0 \,\&\, x_5 = 1) | (x_3 = 1 \,\&\, x_5 = 0). \tag{16}$$

The symbols "&" and "|" represent conjunction and disjunction, respectively. Note that this constraint is not a linear constraint by itself but a disjunction of linear constraint sets. This can be viewed as a set of constraint sets, where at least one constraint set member must be satisfied.

As another example, we also note that lines 7 and 14 are always executed together, and each of them will be executed at most once. This can be represented by

$$(x_3 = 0 \,\&\, x_8 = 0) | (x_3 = 1 \,\&\, x_8 = 1). \tag{17}$$

The path information is not limited to within a function. The user can also specify the path relationship between the caller and callee functions. This is illustrated in the example shown in Fig. 7.

We see that function `clear_data()` will only be executed if the return value from function `check_data()` is zero. More precisely, the number of times that function `clear_data()` is called at f_{12} must be equal to the execution count of basic block B_8 when function `check_data()` is called at the position represented by f_{10}. This information can be represented by the constraint

$$f_{12} = x_8.f_{10}. \tag{18}$$

Here, $x_8.f_{10}$ is a variable representing the execution count of basic block B_8 when function `check_data()` is called at the position represented by variable f_{10}. This variable differs

TABLE I
THE FUNCTIONALITY CONSTRAINTS FOR FUNCTION `check_data()` ARE EXPANDED INTO FOUR FUNCTIONALITY CONSTRAINT SETS. AT LEAST ONE OF THEM MUST BE SATISFIED. NOTE THAT SETS 2 AND 3 ARE NULL SETS BECAUSE OF THE CONTRADICTING VALUES ON VARIABLE x_3

Set 1	Set 2	Set 3	Set 4
$x_1 \le x_2$	$x_1 \le x_2$	$x_1 \le x_2$	$x_1 \le x_2$
$x_2 \le 10x_1$	$x_2 \le 10x_1$	$x_2 \le 10x_1$	$x_2 \le 10x_1$
$x_3 = 0$	$x_3 = 1$	$x_3 = 0$	$x_3 = 1$
$x_5 = 1$	$x_5 = 0$	$x_5 = 1$	$x_5 = 0$
$x_3 = 0$	$x_3 = 0$	$x_3 = 1$	$x_3 = 1$
$x_8 = 0$	$x_8 = 0$	$x_8 = 1$	$x_8 = 1$

from x_8, which represents the *total* execution count of basic block B_8 during the execution of the program. If function `check_data()` is only called at f_{10}, then

$$x_8 = x_8.f_{10}. \tag{19}$$

But suppose that the function is called again at some position f_j; then

$$x_8 = x_8.f_{10} + x_8.f_j. \tag{20}$$

This approach can be thought of as inlining an instance of the callee function at each function call location. And each instance of the callee function has its own set of basic block variables.

Since the functionality constraints are serving the same purpose as constructs in the IDL language provided by Park in his work [5], it is instructive to compare their relative expressive powers. We have been able to demonstrate that our mechanism for providing the functionality constraints is more powerful than the IDL language. A complete proof is shown in Appendix A.

C. Solving the Constraints

The program structural constraint set is a set of program structural constraints that are conjunctive, i.e., they must all be satisfied simultaneously. Due to the presence of the disjunction "|" operator, the program functionality constraints may, in general, be transformed into a disjunction of conjunctive constraint sets. There may be one or more functionality constraint sets. At least one of these program functionality constraint sets must be satisfied for any assignment to the x_i. For example, the functionality constraints of function `check_data()` [(14)–(17)] are expanded into four functionality constraint sets shown in Table I.

To solve the estimated bound, each of the functionality constraint sets is combined (the conjunction taken) with the set of structural constraints. This combined constraint set is passed to the ILP solver with (1) to be maximized. The ILP solver provides the maximum value of the expression and solves basic block variables that result in this maximum value. The above procedure is repeated for every functionality constraint set. The maximum over all of these running times is the maximum running time of the program. Note that a single value of the basic block counts for the worst case is provided in the solution even if there may be a large number of solutions, all of which result in the same worst case timing. The ILP solver in effect has implicitly considered all paths (different assignments to the x_i) in determining the worst case.

```
x1  for (i = 0; i < 11; i++)
x2    if (i mod 3 == 0)
x3      j *= j;
      else
x4      ++j;
```

Fig. 8. An example code fragment to illustrate how the functionality constraints will affect the performance of solving the ILP program.

Clearly, the solving time is directly proportional to the number of functionality constraint sets. This number is doubled every time a functionality constraint with disjunction operator "|" is added. While no theoretical bounds on this can be derived, our observations have been that in practice, this is not a problem. There are two ways to reduce the number of functionality constraint sets. The first is to use more sophisticated linear constraints to describe the same path information in a more concise way without using the "|" operator. For example, the functionality constraint (17) can be replaced by $x_3 + x_5 = 1$. And the second functionality constraint (17) can be replaced by $x_3 = x_8$. These two functionality constraints do not have the "|" operator. Therefore, there will be only one set of functionality constraints. Yet, the new functionality constraints bound the feasible values of x_3, x_5, and x_8 the same way as the original ones do.

The second way to reduce the number of functionality constraint sets is to detect the existence of null set—i.e., the set with no solutions—before calling the ILP solver. Although the ILP solver can detect the existence of a null set much faster than actually solving the solution, the detection of null functionality constraint sets is trivial in many cases (e.g., $x_3 = 1$ and $x_3 = 0$ in Table I) and can be checked when new functionality constraints are added. The functionality constraints within a set are maintained in a sorted order. When a simple functionality constraint of the form $x_a \ge b$ is added, it is checked with the constraints $x_a \le b'$ in the set and determines if a null set will result. Other, more complicated functionality constraints are added to the set unchanged.

The other computational issue is the complexity of solving each ILP problem. In general, this problem is known to be NP-complete. However, there are certain special cases that have polynomial time solutions. Since the structural constraints are derived from the CFG, which is a network flow graph, they exhibit very good integer property. We were able to show that when they are combined with the functionality constraints that correspond to the constructs in the IDL language, the resultant ILP problem has an optimum integral solution, i.e., the ILP problem collapses to an LP problem, which can be solved in polynomial time. The complete proof is shown in Appendix B. However, the full generality of the functionality constraints can result in its being a general ILP problem. In practice, however, this was never experienced. In general, we found that the more accurate the path information, the higher the likelihood that the ILP problem will have an integral optimum solution. This is illustrated in the code fragment shown in Fig. 8.

In this example, the loop body is iterated 11 times. Hence, we have $x_3 + x_4 = 11$. Suppose that $c_3 > c_4$ and the user pro-

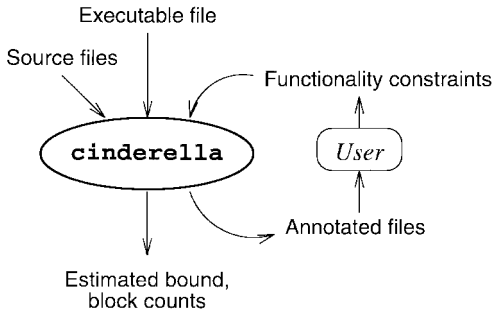


Fig. 9. Block diagram of `cinderella`.

vides a functionality constraint $x_3 \leq x_4$. Then the nonintegral optimum solution will give $x_3 = x_4 = 5.5$ and the integral solution will be $x_3 = 5$ and $x_4 = 6$. This constraint does not tightly bound that actual paths. We can further tighten the solution by noting that since the loop index is iterated from zero to ten, $x_3 = 4$. This functionality constraint will override the previous one and will also result in an integral optimum solution.

IV. Microarchitectural Modeling

As mentioned in Section I-C, the emphasis of this paper is on the path analysis problem; the issues related to microarchitectural modeling will be dealt with separately. Currently, we are using a simple hardware model to determine the bound of the running time of a basic block. For each assembly instruction in the basic block, we analyze its adjacent instructions within the basic block and determine its bound $[t_{i_{\min}}, t_{i_{\max}}]$ from the hardware manual. The bound of the complete basic block is obtained by summing up all the bounds of the instructions using Shaw's methodology [9]. The sum of two bounds $[t_{i_{\min}}, t_{i_{\max}}]$ and $[t_{j_{\min}}, t_{j_{\max}}]$ is equal to the bound $[t_{i_{\min}} + t_{j_{\min}}, t_{i_{\max}} + t_{j_{\max}}]$. This model can handle pipelining reasonably well. However, it is very simplistic in its approach to modeling cache memory. For the best case running time, we assume that the execution always has cache hits, whereas for the worst case running time, we assume that the execution will always result in cache misses. This is clearly a conservative approximation and needs to be tightened. We have extended our method to model the cache memory. The cache modeling is described in [10] and is beyond the scope of this paper.

V. Implementation

We have developed a tool called `cinderella`[3] that incorporates the ideas presented in this paper for timing analysis. It is written in C++ and contains approximately 20 000 lines of code. The formulated ILP problems are passed to a public-domain ILP solver `lp_solve`.[4] Fig. 9 shows the block diagram of `cinderella`.

`cinderella` first reads the executable code for the program. It then constructs the CFG's and derives the program structural constraints. Next, it reads the source files and outputs the annotated source files, where the basic block variables (x_i) are labeled beside the source code (Fig. 6). Then, for all loops in the program, it asks the user to provide the loop bounds. This is the minimum information that is needed to solve the timing bounds, and an initial estimate of these bounds can be obtained at this point. To tighten the estimated bound, the user can provide additional functionality constraints and reestimate the bounds. After each estimation, `cinderella` outputs the estimated bound and the basic block costs and counts.

Currently, `cinderella` is implemented to estimate the running time of programs running on the Intel i960KB processor, i.e., its built-in microarchitectural model is a simplistic one for the i960. The i960KB processor is a 32-bit reduced instruction set computer processor that is being used in many embedded systems (e.g., in laser printers). It contains an on-chip floating point unit and a 512-byte instruction cache [11]. The instruction cache is direct mapped and the line size is 16 bytes.

[3]Details of the tool can be obtained via the `cinderella` Web page: http://www.ee.princeton.edu/~yauli/cinderella.

[4]`lp_solve` was written by M. Berkelaar and can be retrieved by Anonymous ftp: ftp.es.ele.tue.nl Directory: /pub/lp_solve.

VI. Experimental Results

As described in Section III-A, the estimated running time of a program is given by the expression

$$\sum_{i=1}^{N} c_i x_i.$$

Our solution is not guaranteed to give the exact bounds, and in general, some pessimism will be introduced into the estimation. There are two sources for the pessimism: the pessimism in c_i and the pessimism in x_i. The former pessimism results from the inaccuracies in the microarchitectural modeling. It can be reduced by improving the modeling. The latter is due to insufficient information in the program path analysis, where some infeasible paths are considered to be feasible. This can hopefully be reduced by providing more functionality constraints.

Since our current work focuses on the path analysis problem, we would like to evaluate the efficacy of our methodology in determining the worst and best case paths. Experiment 1 described below is directed toward evaluating the pessimism in path analysis. In addition, we conducted Experiment 2, with the goal of measuring the inadequacies in our current microarchitectural modeling. Here, we compare the estimated running time given by `cinderella` and the actual running time obtained by measuring the execution time of the program on an evaluation board.

A. Experiment 1: Evaluating the Path Analysis Accuracy

Since there are no established benchmarks for this purpose, we collected a set of example programs from a variety of sources for this task. Some of them are from academic sources: from Arnold [12] and Park [5] on timing analysis of software and also from Gupta's thesis [13] on the hardware-software codesign of embedded systems. Others are from standard DSP applications, as well as software benchmarks used for evaluating optimizing compilers. There is also an application program called `djpeg`, which decompresses JPEG images. It

TABLE II
SET OF BENCHMARK EXAMPLES

Program	Description	Lines	Bytes
check_data	Example from Park's thesis	23	88
circle	Circle drawing routine (from [13])	100	1,588
des	Data Encryption Standard	192	1,852
dhry	Dhrystone benchmark	761	1,360
djpeg	Decompression of 128×96 color JPEG image	857	5,408
fdct	JPEG forward discrete cosine transform	300	996
fft	1024-point Fast Fourier Transform	57	500
line	Line drawing routine (from [13])	165	1,556
matcnt	Summation of 2 100×100 matrices, (from [12])	85	460
piksrt	Insertion Sort of 10 elements	19	104
sort	Bubble sort of 500 elements, (from [12])	41	152
stats	Calculate the sum, mean and variance of two 1,000-element arrays, (from [12])	100	656
whetstone	Whetstone benchmark	196	2,760

is the biggest and most complicated program being analyzed in this research area. A full description of these programs, their source-level line sizes, and their binary code sizes is shown in Table II.

We studied each program carefully and determined the loop bounds and other path information as much as possible. This information is fed into `cinderella` in the form of functionality constraints to obtain the estimated bound.

Our objective in this experiment is to validate the accuracy of the program path analysis. This implies comparing the values of x_i with the actual measured basic block execution counts. A program may have more than one extreme case execution path, however, and therefore there may exist more than one set of basic block execution counts that result in the same extreme case execution time. For this reason, we compared the estimated bound with the *calculated* bound, which is equal to

$$\left[\sum_{i=1}^{N} c_i^{\text{best}} \times \text{measured\_block\_count}_i^{\text{best\_data}},\ \sum_{i=1}^{N} c_i^{\text{worst}} \times \text{measured\_block\_count}_i^{\text{worst\_data}}\right].$$

Here, c_i^{best} and c_i^{worst} are the best (shortest) and the worst (longest) execution times of basic block B_i. They are identical to those used by `cinderella` in computing the estimated bound. By using the same costs in the estimated bounds and the calculated bounds, the pessimism due to the costs was not accounted for in this experiment.

To measure the extreme case basic block execution counts, we need to determine the best and worst case input data for each program. The input data affect the running time of a program in two ways. In the microarchitectural level, the execution times of some instructions, such as floating instructions, are data dependent, which may in turn depend on the input data. In the program path level, the input data may effect the branch decision taken at each compare-and-branch instruction. This will in turn affect the execution path and hence the execution time of the program. In this experiment, since basic block costs (c_i) are isolated, only the second effect is accounted for in the calculated bound.

Note that in order to determine the actual upper (lower) bound on the execution time, we would actually have to run the program for all possible inputs. This is clearly not feasible. Thus, we have replaced this step by trying to identify the best (worst) case data set by a careful study of the program. For simple programs like `check_data` and `sort`, this is trivial. But for complicated programs like `des` and `djpeg`, it becomes extremely difficult. In this case, we generated a series of data sets that we believed would result in execution times close to the extreme case execution times. We took those that resulted in the shortest and longest execution times. The basic block counts were measured by inserting counters into the basic blocks of the program and executing the program with the extreme case data sets.

The result of this evaluation is shown in Table III. The second column indicates the number of constraint sets being passed to the ILP solver. Of the eight constraints sets of function `dhry`, five of them are detected as null sets. They are eliminated, and the remaining three sets are passed to the solver. The pessimism in the evaluation is defined as the relative difference between the calculated bound and the estimated bound and is calculated as follows:

$$\text{lower} = \frac{\text{Calc. lower} - \text{Est. lower}}{\text{Calc. lower}}$$
$$\text{upper} = \frac{\text{Est. upper} - \text{Calc. upper}}{\text{Calc. upper}}.$$

All estimated bounds correctly bound their corresponding calculated bounds. We also see that when given enough information, the path analysis can be very accurate. The exception is the upper bound of the `djpeg` program. The main reason for this discrepancy is due to the Huffman decoding routine in the program. For computing the upper estimated bound, we assume that in the worst case, all 64 coefficients in every 8 × 8 block are random, and hence no compression of the coefficients is achieved. But for every random image we generated for the worst case measurements, some sort of compression is still achieved. Therefore, each Huffman symbol takes longer to fetch in the worst case estimation than in our

TABLE III
PESSIMISM IN PATH ANALYSIS. THE ESTIMATED BOUND AND THE CALCULATED BOUND ARE IN UNITS OF CLOCK CYCLES

Program	Constraint Sets	Estimated Bound		Calculated Bound		Pessimism	
		lower	upper	lower	upper	lower	upper
check_data	4 ⇒ 2	35	1,193	35	1,193	0.00	0.00
circle	1	431	15,958	431	15,726	0.00	0.01
des	2	73,912	672,298	75,033	667,127	0.01	0.01
dhry	8 ⇒ 3	314,266	1,326,475	314,266	1,326,475	0.00	0.00
djpeg	1	12,703,432	122,838,368	12,925,769	98,696,050	0.02	0.24
fdct	1	5,587	16,693	5,587	16,693	0.00	0.00
fft	1	1,589,026	3,974,624	1,593,122	3,974,601	0.00	0.00
line	1	380	9,148	380	9,148	0.00	0.00
matcnt	1	1,722,105	8,172,149	1,722,105	8,172,149	0.00	0.00
piksrt	1	236	5,862	236	5,862	0.00	0.00
sort	1	13,965	50,244,928	13,965	50,244,928	0.00	0.00
stats	1	1,007,815	2,951,746	1,007,815	2,951,746	0.00	0.00
whetstone	1	5,634,926	14,871,610	5,634,926	14,871,610	0.00	0.00

TABLE IV
DISCREPANCY BETWEEN THE ESTIMATED BOUND AND THE MEASURED BOUND. THE ESTIMATED BOUND AND THE MEASURED BOUND ARE IN UNITS OF CLOCK CYCLES

Program	Estimated Bound		Measured Bound		Pessimism	
	lower	upper	lower	upper	lower	upper
check_data	35	1,193	35	430	0.00	1.77
circle	431	15,958	585	14,483	0.26	0.10
des	73,912	672,298	111,468	243,676	0.34	1.76
dhry	314,266	1,326,475	575,492	575,622	0.45	1.30
djpeg	12,703,432	122,838,368	14,975,268	35,636,948	0.15	2.45
fdct	5,587	16,693	7,616	9,048	0.27	0.84
fft	1,589,026	3,974,624	1,719,813	2,204,472	0.08	0.80
line	380	9,148	929	4,836	0.59	0.89
matcnt	1,722,105	8,172,149	2,202,276	2,202,698	0.22	2.71
piksrt	236	5,862	337	1,705	0.30	2.44
sort	13,965	50,244,928	16,492	9,991,172	0.15	4.03
stats	1,007,815	2,951,746	1,158,142	1,158,469	0.13	1.55
whetstone	5,634,926	14,871,610	6,935,612	6,935,668	0.19	1.14

actual test runs. This benchmark program also illustrates the point that timing simulation has its limitation in determining the worst case execution time, as the worst case input data may not be determined.

As shown in the table, the number of constraint sets is very small, usually one. Also, the CPU times taken for this analysis were insignificant, less than 2 s on an SGI Indigo Workstation in each case. This is largely due to the fact that the branch-and-bound ILP solver finds that the solution of the very first linear program call it makes is integer valued.

B. Experiment 2: Comparison with Actual Running Times

In this experiment, we measured the actual running time of the program and compared it with the estimated bound. Each program was compiled by an Intel i960 C compiler on a PC and then downloaded onto an Intel QT960 board [14], which is a development board containing a 20-MHz i960KB processor, memory, and some other peripherals. To determine the upper measured bound, we flushed the cache memory first before executing the program with its worst case data set. The execution time was measured by a logic analyzer. For lower measured bound, we executed the program first with its best case data set, so that the cache was filled with the program's code. We then executed the program with its best case data set again and measured its execution time.

Table IV shows the results of this experiment. The estimated bound is the same as in Experiment 1. The measured values described above are shown in the measured bound column. The pessimism is calculated from the following formulas:

$$\text{lower} = \frac{\text{Mea. lower} - \text{Est. lower}}{\text{Mea. lower}}$$
$$\text{upper} = \frac{\text{Est. upper} - \text{Mea. upper}}{\text{Mea. upper}}.$$

We observe that while the estimated bound does enclose the measured bound, the pessimism in the estimation is rather high. This is mainly due to the fact that a simple hardware model is used. In particular, the lower pessimism is generally smaller than the upper pessimism. The main reason is that in computing lower estimated bound, we assumed that all instruction fetches would result in cache hits. This is close to the actual cache hit ratio, which is typically around 90%. But for computing the upper estimated bound, we conservatively assumed that all instruction fetches would result in cache misses. This pessimistic assumption results in a loose up-

TABLE V
TRANSFORMATION FROM IDL TO FUNCTIONALITY CONSTRAINTS

Description	IDL information clause	Functionality Constraint
Statement A is always executed.	**always**(A)	$x_A \geq 1$
Statements A and B are always executed together.	**samepath**(A, B)	$(x_A \geq 1 \ \& \ x_B \geq 1) \mid (x_A = 0 \ \& \ x_B = 0)$
No execution path passes through both statements A and B.	**nopath**(A, B)	$x_A = 0 \mid x_B = 0$
Statements A and B are mutually exclusive.	**exclusive**(A, B)	$(x_A \geq 1 \ \& \ x_B = 0) \mid (x_A = 0 \ \& \ x_B \geq 1)$
Statement A is executed between l and u times.	**execute** A [l, u] **times**	$l \leq x_A \ \& \ x_A \leq u$
If statement C is executed, then information clause I_C is valid.	C **imply** I_C	$x_C = 0 \mid$ (functionality constraint of I_C)
Loop L is iterated between l and u times.	**loop** L [l, u] **times**	$lx_L \leq x_B \ \& \ x_B \leq ux_L$, where B is the first statement in the loop body.

per estimated bound. A more sophisticated microarchitectural modeling, including cache modeling, will certainly improve the accuracy of the estimated bound.

VII. CONCLUSION AND FUTURE WORK

In this paper, we have presented an efficient method to estimate the bounds of the running time of a program on a given processor. The method uses integer linear programming techniques to perform the path analysis without explicit path enumeration. It can accept a wide range of information on the functionality of the program in the form of sets of linear constraints. A tool called `cinderella` has been developed to perform this timing analysis. Experimental results on a set of examples show the efficacy of this approach.

Future work includes improving the hardware model to take into account the effects of cache memory and other features of modern processors that tend to make the timing relatively nondeterministic. In addition, we would like also to explore the possibility of using data flow analysis techniques to derive automatically some of the functionality constraints that are currently being provided by the user. Last, we are working on porting `cinderella` to handle programs running on other hardware platforms. In this direction, in collaboration with AT&T, we have completed a port for the AT&T DSP3210 processor. This is intended for use in the VCOS operating system to bound the running times of processes for use in scheduling.

APPENDIX A
FUNCTIONALITY CONSTRAINTS VERSUS IDL

This appendix shows that the functionality constraints described in Section III-B2 are at least as powerful as the IDL devised by Park [5]. This is done by showing that every information clause in IDL can be represented by the functionality constraints.

In IDL, each statement is labeled by an upper case alphabet. The statement corresponds to a basic block in our case. In the following, if A is a statement label used in IDL, then x_A will be the basic block variable representing the execution count of the statement A. Table V shows the transformation.

The *loop* information clauses are used to specify loop bounds. Other information clauses are used to provide additional path information. Each of these is transformed into a set of linear constraints of the form $x_i \geq b$, where b is an integer. More complicated path information, such as that the execution count of a statement is no greater than that of the other, cannot be represented by IDL. The functionality constraints overcome this disadvantage. The above path information can be easily expressed as $x_A \leq x_B$.

APPENDIX B
INTEGER PROPERTY OF ILP FORMULATION

This appendix describes the conditions under which the optimum solution of the ILP problem is guaranteed to be integral values. We will show that for a structured program with functionality constraints derived from Park's IDL, the ILP problem collapses to an LP problem.

Proof:

Case 1: If the program contains no loops and all functionality constraints provided by the user are derived from Park's IDL, then the CFG becomes a simple network flow graph and the functionality constraints are of the form $x_i \geq b$, where b is an integer. These constraints essentially bound the flow at the nodes of the graph. As a result, the problem of determining the estimated bound is equivalent to a network flow program, which is known to have an integral optimum solution because of the total unimodular property [15].

Case 2: If there is a single loop in a structured program, and the loop bound is provided as $lx_L \leq x_B \leq ux_L$, where l and u are positive integers denoting the lower and upper loop bounds, basic block B_L is the basic block just before the loop is entered and B_B is the first basic block in the loop body. Here, the loop bound constraints are not in the form $x_i \geq b$ as in Case 1, as there are variables on both sides of the inequality. If we can show that x_L is always an integer in optimum solution, however, then the loop bound constraints will be transformed to the form $l' \leq x_B \leq u'$, and we can

apply the result in Case 1 to prove that all basic block variables are integral.

Because of the structured program, the CFG is reducible, and the whole loop can be reduced to a single node in the CFG [8], with its execution count equal to x_L. Since there is no loop in the reduced graph, the reduced graph is a simple network flow graph, and therefore x_L must be an integer no matter what the cost of the loop and the loop bound are. Consequently, x_B must also be an integer, as are the rest of the basic block variables inside the loop.

Case 3: If there are nested loops in the structured program, we can apply the proof from Case 2 to reduce the loop into a node repeatedly from the innermost level to the outermost level. Then, by using the result from Case 2 again, we can show that all levels of loop bounds can be transformed to the form $l' \leq x_B \leq u'$, and hence all basic block variables must be integral in the worst case.

REFERENCES

[1] A. M. van Tilborg and G. M. Koob, Eds., *Foundations of Real-Time Computing: Scheduling and Resource Management*. Norwell, MA: Kluwer, 1991.

[2] P. Puschner and C. Koza, "Calculating the maximum execution time of real-time programs," *J. Real-Time Syst.*, vol. 1, no. 2, pp. 160–176, Sept. 1989.

[3] E. Kligerman and A. D. Stoyenko, "Real-Time Euclid: A language for reliable real-time systems," *IEEE Trans. Software Eng.*, vol. SE-12, pp. 941–949, Sept. 1986.

[4] A. K. Mok, P. Amerasinghe, M. Chen, and K. Tantisirivat, "Evaluating right execution time bounds of programs by annotations," in *Proc. 6th IEEE Workshop Real-Time Operating Systems and Software*, May 1989, pp. 74–80.

[5] C. Y. Park, *Predicting Deterministic Execution Times of Real-Time Programs*, Ph.D. dissertation, University of Washington, Seattle, Aug. 1992.

[6] R. B. Hitchcock, "Timing verification and the timing analysis program," in *Proc. 19th Design Automation Conf.*, June 1982, pp. 594–604.

[7] S. Devadas, K. Keutzer, and S. Malik, "Computation of floating mode delay in combinational circuits: Theory and algorithms," *IEEE Trans. Computed-Aided Design*, vol. 12, pp. 1913–1923, Dec. 1993.

[8] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers Principles, Techniques, and Tools.* Reading, MA: Addison-Wesley, 1986.

[9] A. C. Shaw, "Reasoning about time in higher-level language software," *IEEE Trans. Software Eng.*, vol. 15, pp. 875–889, July 1989.

[10] Y.-T. S. Li, S. Malik, and A. Wolfe, "Performing estimation of embedded software within instruction cache modeling," in *Proc. IEEE/ACM Int. Conf. Computer-Aided Design*, Nov. 1995, pp. 380–337.

[11] Intel Corporation, *i960KA/KB Microprocessor Programmer's Reference Manual.* Santa Clara, CA: Intel Books, 1991.

[12] R. Arnold, F. Mueller, D. Whalley, and M. Harmon, "Bounding worst-case instruction cache performance," in *Proc. 15th IEEE Real-Time Systems Symp.*, Dec. 1994, pp. 172–181.

[13] R. K. Gupta, *Co-Synthesis of Hardware and Software for Digital Embedded Systems*, Ph.D. dissertation, Stanford University, Stanford, CA, Dec. 1993.

[14] Intel Corporation, "QT960 user manual," Santa Clara, CA, 1990.

[15] A. Sultan, *Linear Programming, An Introduction with Applications.* New York: Academic, 1993.

Yau-Tsun Steven Li received the B.A. (first-class honors) degree in engineering and computer science from Oxford University, U.K., in 1992 and the M.A. and Ph.D. degrees in electrical engineering from Princeton University, Princeton, NJ, in 1994 and 1997, respectively.

Currently, he is a Member of the Technical Staff with Monterey Design Systems, Inc. His research interests are in embedded system designs and electronic design automation tools.

Sharad Malik (S'88–M'90) received the B.Tech. degree in electrical engineering from the Indian Institute of Technology, New Delhi, in 1985 and the M.S. and Ph.D. degrees in computer science from the University of California, Berkeley, in 1987 and 1990, respectively.

Currently, he is an Associate Professor in the Department of Electrical Engineering, Princeton University, Princeton, NJ. His current research interests are design tools for embedded computer systems, synthesis, and verification of digital systems. He is on the editorial boards of the *Journal of VLSI Signal Processing* and *Design Automation for Embedded Systems*.

Dr. Malik received the President of India's Gold Medal for academic excellence (1985), IBM Faculty Development Award (1991), National Science Foundation (NSF) Research Initiation Award (1992), Princeton University Rheinstein Faculty Award (1994), NSF Young Investigator Award (1994), Best Paper Award at both the IEEE International Conference on Computer Design (ICCD) (1992) and the ACM/IEEE Design Automation Conference (DAC) (1996), Walter C. Johnson Prize for Teaching Excellence (1993), and Princeton University Engineering Council Excellence in Teaching Award (1993–1995). He has served on the program committees of the DAC, International Conference on Computer-Aided Design, and ICCD.