ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems, La Jolla, California, June 1995.

Performance Analysis of Embedded Software Using Implicit Path Enumeration

Yau-Tsun Steven Li and Sharad Malik
Department of Electrical Engineering,
Princeton University,
NJ 08544, USA.

May 14, 1995

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 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` [1] which currently targets a popular embedded processor — the Intel i960. The preliminary results of using this tool are also presented here.

1 Introduction

1.1 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 ASIC 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 a billion dollars for a new line, the only components that make this affordable are high volume parts such as processors, memories and possibly FPGAs. 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 have predictable schedules. Both of these can be better met with software programmable solutions made possible by embedded systems. Thus, we are seeing a movement from the logic gate being the basic unit of computation on silicon, to an instruction running on an embedded processor. This has motivated our research efforts directed towards 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/she cannot prove that a less expensive MC68030 can always compute the engine control parameters within a single rotation.

[1] In recognition of her hard real-time constraint — she had to be back home at the stroke of midnight!

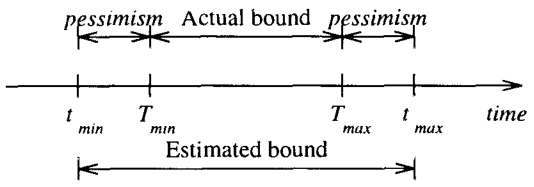


Figure 1: The estimated bound $[t_{min}, t_{max}]$ and the actual bound $[T_{min}, T_{max}]$ of a program's all possible execution times.

1.2 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. In general, information about interrupt latencies is not known at the application programming level, and any analysis that includes consideration of interrupts should be done at the process level. 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 out a correct estimate of this without introducing undue pessimism. Thus, the estimated time interval $[t_{min}, t_{max}]$, defined as the *estimated bound*, must enclose the actual bound. This is illustrated in Fig. 1.

1.3 Sub-Problems

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 execute in the extreme case – this is referred to as the **program path analysis problem**.
- compute how much time it will take to the system to execute that sequence. This requires a modeling of the host processor system – this aspect 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 to clearly understand each problem. The focus of this paper is on the program path analysis problem.

2 Previous Work

A static analysis of the code is needed to see what the possible extreme case paths through the code are. It is well accepted that this problem is undecidable in general and equivalent to the halting problem. However, in practice, designers select algorithms and data structures for embedded programs that they believe to be easily bounded. Since the programmer must prove to himself/herself that a real-time program will terminate, a restricted programming style is typically used. Several researchers have suggested restrictions on programs that make this problem decidable. These are: absence of dynamic data structures, such as pointers and dynamic arrays; the absence of recursion; and bounded loops. Kligerman and Stoyenko [2] as well as Puschner and Koza [3] are variously credited with these observations. These restrictions may be imposed either through specific language constructs, or programmer annotations on conventional programs. While specific language constructs, such as those provided in Real-Time Euclid [2], are useful in as much 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, as well as programmers 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 trade-off 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 and his co-workers [4], Puschner and Koza [3], 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 co-workers demonstrated the inadequacy of this approach, since it was difficult to predict the bounds for a high-level language construct independent of the context it appeared in, 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 and his co-workers [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. Finally, 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 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/she is familiar with that the program is supposed to do [2]. 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 [3] or annotations in a separate timing analysis program [4]. Both 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 what 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.

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 (called 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 the 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

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

Figure 2: Exponential Blowup of Paths: An Example

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. Finally, 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 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 which explicitly enumerates paths tends to have limited application [3].

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. In fact, in practice, we observed that the actual computation done by the ILP solver is solving a single linear program. The reasons for this will be briefly examined in Section 3.2, and practical data supporting this will be presented in Section 6.

3 ILP Formulation

3.1 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

[2]There is an analog of this 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.

[3]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].

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 the number of times basic block B_i is executed when the program takes the maximum time to complete. 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 4. Thus, the worst case timing for the program is given by the maximum value of the following expression:

$$\sum_{i=1}^{N} c_i x_i \qquad (1)$$

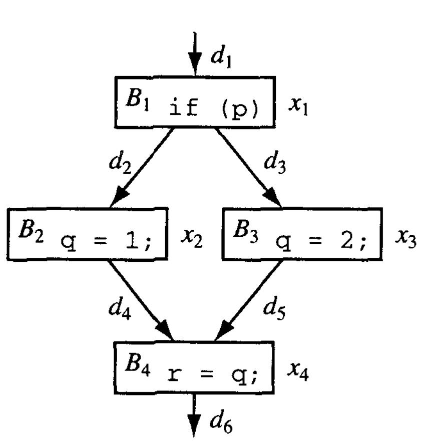
In the above expression, the x_i are positive integers and N is the total number of basic blocks. Clearly without any additional information, the maximum value of this expression is ∞ since any x_i can take on the value of ∞. This is precisely why this problem is undecidable in general. However, as pointed out in Section 2, 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 which can be used in the above expression. However, the bound obtained by this method will typically be very loose, since, in general, the upper bound for each x_i is rarely achieved simultaneously.

There are two factors which 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 = true;
  }
}
```

The first 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 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 what 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,

```
if (p)
  q = 1;
else
  q = 2;
r = q;
```



(i) Code (ii) CFG

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

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 1. 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 what we need to do is to maximize Expression 1 above while taking into account the restrictions imposed by the program structure and functionality.

3.2 Linear Constraints

Since Expression 1 is a linear expression, it would be nice if we can 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 as well as the program functionality restrictions can be specified in the form of sets of linear constraints.

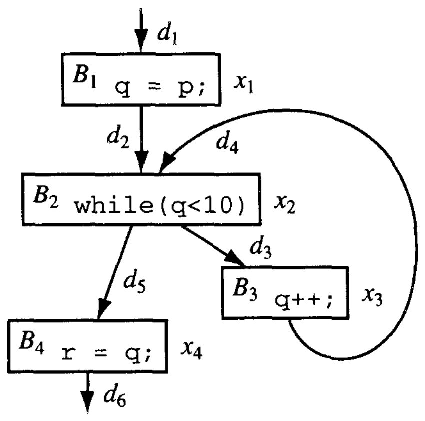
3.2.1 Program Structural Constraints

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's and x_i's respectively (Fig. 3(ii)). These variables represent the number of times that the program control is passing through those edges and basic blocks when the program is executed.

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:

```
/* p >= 0 */
q = p;
while(q<10)
  q++;
r = q;
```

(i) Code



(ii) CFG

Figure 4: An example of the while-loop statement and its CFG.

$$x_1 = d_1 = d_2 + d_3 \quad (2)$$
$$x_2 = d_2 = d_4 \quad (3)$$
$$x_3 = d_3 = d_5 \quad (4)$$
$$x_4 = d_4 + d_5 = d_6 \quad (5)$$

Fig. 4 shows a while-loop statement and its CFG. The constraints extracted from the CFG of this example are:

$$x_1 = d_1 = d_2 \quad (6)$$
$$x_2 = d_2 + d_4 = d_3 + d_5 \quad (7)$$
$$x_3 = d_3 = d_4 \quad (8)$$
$$x_4 = d_5 = d_6 \quad (9)$$

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 the structural constraints have been constructed, the user must provide the loop bound information as part of specifying the program functionality constraints (see Section 3.2.2).

The function calls are represented by using 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 structural constraints in the caller function remains the same. In the example, the structural constraints of the caller function are:

$$x_1 = d_1 = f_1 \quad (10)$$
$$x_2 = f_1 = f_2 \quad (11)$$

```
        check_data()
        {
          int i, morecheck, wrongone;

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

          while (morecheck) {
    x2      if (data[i] < 0) {
    x3        wrongone=i; morecheck=0;
            }
            else
    x4        if (++i >= DATASIZE)
    x5          morecheck = 0;
    x6    }

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

Figure 6: check_data example from Park's thesis.

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

$$d_2 = f_1 + f_2 \quad (12)$$

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 starting edge of the main function must be equal to 1. Therefore if variable d_1 represents the count of this edge, the following constraint is constructed.

$$d_1 = 1 \quad (13)$$

3.2.2 Program Functionality Constraints

As described previously, these constraints are used to denote loop bounds and other path information that depends 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 symbolic analysis techniques. We illustrate the use of these constraints to capture conditions on feasible program paths with the following example (Fig. 6) taken from Park's thesis [5].

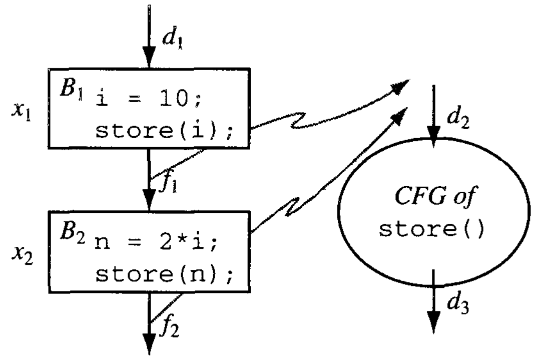
The function check_data() checks the values of data[] array. If any of them is less than zero, the function will stop checking and return 0 immediately, otherwise it will return 1.

The loop count of the while-loop in the function is bounded by 1 and DATASIZE, i.e., the loop will be executed at

```
i = 10;
store(i);
n = 2*i;
store(n);

void store(int i)
{
   ...
}
```

(i) Code



(ii) CFG

Figure 5: An example showing how function calls are represented.

least once and at most DATASIZE times. Suppose that DATASIZE is previously defined as a constant value 10, then to specify this loop bound information, we can use the following two constraints:

$$1x_1 \leq x_2 \quad (14)$$

$$x_2 \leq 10x_1 \quad (15)$$

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 inside the loop. Since all the loops are marked while determining program structural constraints, these two variables can be determined automatically. All the user has to provide are the values 1 and 10.

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

$$(x_3 = 0 \,\&\, x_5 = 1) \mid (x_3 = 1 \,\&\, x_5 = 0) \quad (16)$$

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 of this set must be satisfied.

As an another example, we also note that line 7 and line 14 are always executed together. This can be represented by:

$$x_3 = x_8 \quad (17)$$

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

We see that the function clear_data() will only be executed if the return value from the function check_data() is 0. This information can be represented by the constraint:

$$f_{12} = x_8.f_{10} \quad (18)$$

Here, the dot symbol '.' in $x_8.f_{10}$ means that the basic block represented by x_8 in function check_data() called at location f_{10}. If the function check_data() is called from some other place, the value of x_8 will not affect that of f_{12}. For purpose of analysis, a separate set of x_i variables is used for this instance of the call to function check_data().

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 that the IDL language. For every construct

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

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

Figure 7: An example showing how the path relationship between the caller and the callee function can be specified.

in IDL we can provide a disjunctive form constraint. In addition, we can provide disjunctive constraints for practically useful annotations that are beyond the capabilities of IDL. A complete proof of this claim is beyond the scope of this paper.

3.3 Solving the Constraints

The program structural constraint set is a set of constraints that are conjunctive, i.e., they must all be satisfied simultaneously. The program functionality constraints may, in general, be a disjunction of conjunctive constraint sets, giving us a set of constraint sets, at least one of which is satisfied for any assignment to the x_is. This is because of the disjunction '|' and conjunction '&' operators. From Eq. (14) through Eq. (16), there are two sets of program functionality constraints the for function `check_data()`. They are shown in Fig. 8.

To estimate the running time, each set 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 Expression 1 to be maximized. The ILP solver provides the maximum value of the expression, as well as basic block counts (x_i values) that result in this maximum value. The above procedure is repeated for every set of functionality constraint sets. The maximum over all 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 variables) in determining the worst case.

Clearly, the number of calls to the ILP solver is equal to the number of functionality constraints sets. This number is doubled every time a functionality constraint with disjunction operator '|' is introduced. It is, therefore, a good idea to minimize the number of sets before calling the ILP solver. We found that in many cases some of the constraint sets will become a null set. That is, the set encloses the null region in space. Usually, the detection of null sets is trivial (e.g., $x_1 \geq 1$ & $x_1 = 0$). So, after each functionality constraint is added, we check the list of constraints sets and remove some trivial null sets. Each functionality constraint set is also maintained in a sorted order for easier null set detection.

Set 1	Set 2
$x_1 - x_2 \leq 0$	$x_1 - x_2 \leq 0$
$10x_1 - x_2 \geq 0$	$10x_1 - x_2 \geq 0$
$x_3 = 0$	$x_3 = 1$
$x_5 = 1$	$x_5 = 0$
$x_3 - x_8 = 0$	$x_3 - x_8 = 0$

Figure 8: Two sets of program functionality constraints for the function `check_data()`.

There are two computational issues that merit further discussion here. The first is the issue of the total number of constraint sets. Since each distinct constraint set results in an ILP problem that needs be solved, we have to be careful that this number does not become too large. While no theoretical bounds on this can be derived, our observations have been that in practice this is not a problem. The number of constraint sets are small to begin with, and with the pruning of null sets, this number is further decreased. Specific data will be provided in Section 6. The second issue is the complexity of solving each ILP problem. In general this problem is known to be NP-hard. However, there are certain special cases that have polynomial time solutions. If the only constraints that we consider are the program structural constraints, then in fact our ILP problems correspond to a network flow problem with polynomial time solutions. Interestingly, we were also able to demonstrate that if we restrict our functionality constraints to those that correspond to the constructs in the IDL language, then the ILP problem retains the special form of a network flow problem. However, the full generality of the functionality constraints can result in it being a general ILP problem. In practice, however, this was never experienced, i.e., in the branch and bound solution to the ILP, the first call to the linear program package resulted in an integer valued solution.

4 Microarchitectural Modeling

As mentioned in Section 1.3, the emphasis of this paper is on the path analysis problem, the issues related to microarchitectural modeling will be dealt 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 reasonable well. However, it is very simplistic in its approach to modeling cache memory. For best case running time, we assume the execution always has cache-hits, whereas, for worst case running time, we assume that the execution will always result in cache-misses. This is clearly a conservative approximation and needs to tightened. We are currently working on accurately modeling the effect of cache misses. Several other researchers in the real-time community are also looking into this problem [10].

The one issue in microarchitectural modeling that we do need to address in this paper is the assumption that c_i (worst (best) case running time of a basic block) is the same for each iteration of the basic block. Clearly if we pick c_i to be such that it is the maximum (minimum) time over all iterations, then this is a correct, though possibly pessimistic se-

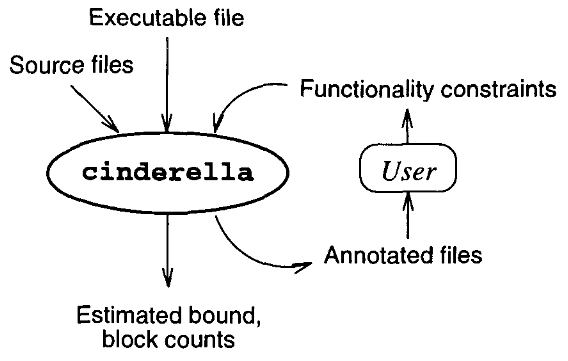


Figure 9: Block diagram of `cinderella`.

lection. In particular, it may happen that the first iteration of a loop results in cache misses, while the subsequent iterations find all the instructions in the cache. Assuming that all iterations result in all cache misses can be very pessimistic. However, this pessimism can easily be avoided in the path analysis stage by considering the first iteration of the loop as a separate basic block, distinct from the other iterations, with its own x and c variables. This duplication method is general enough to incorporate dealing with special execution cases of any basic block.

5 Implementation

We have developed a tool called `cinderella` that incorporates the ideas presented in this paper for timing analysis. It is written C++ and contains approximately 8,000 lines of code. `Cinderella`'s ILP solver module is obtained by modifying the I/O interface of the public domain program `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 and derives the program structural constraints. Next, it reads the source files and outputs the annotated source files, where all the variables denoting the basic block counts (x_i's) are labelled alongside with the source code (Fig. 6). Then, for all loops in the program it asks the user to provide the loop bounds. This is all the information that is mandatory to provide 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 re-estimate the bounds again. After each estimation, `cinderella` outputs the estimated bound, the basic block costs and counts.

Currently, `cinderella` is implemented to estimate the running time of programs running on Intel i960KB processor, i.e., its built in microarchitectural model is a simplistic one for the i960. The i960KB processor is a 32 bit RISC 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.

[4] `lp_solve` is written by Michel Berkelaar and can be retrieved by anonymous ftp from `ftp.es.ele.tue.nl` in directory `/pub/lp_solve`.

6 Experimental Results

As described in Section 3.1, the estimated running time of a program is given by the expression:

$$\sum_{i}^{n} c_i x_i$$

Our solution is not guaranteed to give the exact bounds, and in general some pessimism will be introduced in the estimation. There are two sources for the pessimism: the pessimism in c_i's and the pessimism in x_i's. The former pessimism results from the inaccuracies of the microarchitectural modeling. It can be reduced by improving the modeling. The latter is due to insufficient information in 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 towards evaluating the pessimism in path analysis. In addition to this, we also conducted Experiment 2, with the goal of measuring the inadequacies in our current microarchitectural modeling. Here we compare the estimated running time given by the `cinderella` and the actual running time obtained by measuring the execution time of the program on an evaluation board.

6.1 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 Park's thesis [5] on timing analysis of software and also from Gupta's thesis [12] on the hardware-software co-design of embedded systems. Others are from standard DSP applications, as well as software benchmarks used for evaluating optimizing compilers. These routines are shown in Table 1. In addition to a description of the source of the routine, its size in terms of the number of lines of C is also stated there. For each routine, we obtain the estimated bound by using `cinderella` and calculate the *calculated* bound, which is obtained by the following steps:

1. Insert a counter into each basic block of the routine.
2. Identify the initial data set that corresponds to the longest (shortest) running time of the routine.

Table 1: Set of Benchmark Examples

Function	Description	Lines
check_data	Example from Park's thesis	17
fft	Fast Fourier Transform	56
piksrt	Insertion Sort	15
des	Data Encryption Standard	185
line	Line drawing routine in Gupta's thesis	143
circle	Circle drawing routine in Gupta's thesis	88
longloop	A simple long loop for evaluating cache behavior	292
jpeg_fdct_islow	JPEG forward discrete cosine transform	150
jpeg_idct_islow	JPEG inverse discrete cosine transform	246
recon	MPEG2 decoder reconstruction routine	137
fullsearch	MPEG2 encoder frame search routine	204
whetstone	Whetstone benchmark	245
dhry	Dhrystone benchmark	480
matgen	Matrix generating routine in Linpack benchmark	50

3. Run the routine with that data set and record the values of all the counters.
4. Multiply each counter value with the slowest (fastest) running time for that basic block as provided by `cinderella`.
5. Add up all these products. This is the upper (lower) bound of the calculated bounds.

The calculated bound can be summarized by the following expression:

$$[\sum measured\_block\_count_{best\_data} \times block\_cost_{lower},$$
$$\sum measured\_block\_count_{worst\_data} \times block\_cost_{upper}]$$

Note that in order to know the actual upper (lower) bound on the execution time, we would actually have to run the routine for all possible inputs. This is clearly not feasible. Thus, we have replaced this step by actually trying to identify the best (worst) case data set by a careful study of the program. Clearly if we could rely on this identification of the best and the worst case, we do not need to do the analysis at all. However, as we have no other mechanism to evaluate the result, we have to use this method. We do know however, that if the analysis result agrees with our selection of the data set, then in fact it is indeed the worst case data set and also that our analysis is completely accurate. As the results of Experiment 1 show, the analysis provides results that are either in agreement with the selection of the data set, or very close to it.

The result of this evaluation is shown in Table 2. The second column indicates the number of constraints 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 actual bound and the estimated bound and is calculated as follows:

$$lower = \frac{Cal.\ lower - Est.\ lower}{Cal.\ lower}$$

$$upper = \frac{Est.\ upper - Cal.\ upper}{Cal.\ upper}$$

From these results, we see that when given enough information, the path analysis can be very accurate. As shown in the table, the number of constraint sets is very small, usually 1. Also, the CPU times taken for this analysis were insignificant, less than 2 seconds 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.

Although in some cases, it is difficult for us to trace the program, we think that the programmer who writes the program is familiar with the program and so he/she can provide the path information accurately and with relatively less effort.

6.2 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 is compiled by the Intel i960 C compiler on a PC and then downloaded to an Intel QT960 board [13], which is a development board containing a 20MHz i960KB processor, memory and some other peripherals. To measure the worst case (maximum) running time, we initialize the routine with its worst case data set and then run it in a loop several hundred times and measure the elapsed time. The cache memory is flushed before each function call. Since this value includes the time to do the loop iterations and cache flushing, we run an empty loop and measure its execution time again. The difference between these two values is the actual running time

Table 2: Pessimism in path analysis.

Function	Number of Constraints Sets	Estimated Bound		Calculated Bound		Pessimism	
		lower	upper	lower	upper	lower	upper
check_data	2	32	1,039	32	1,039	0.00	0.00
fft	1	966,652	3,346,000	976,328	3,312,544	0.01	0.01
piksrt	1	146	4,333	146	4,333	0.00	0.00
des	2	42,302	604,169	43,254	592,559	0.02	0.02
line	1	336	8,485	336	8,485	0.00	0.00
circle	1	502	16,652	502	16,301	0.00	0.02
longloop	1	12,254	14,104	12,254	14,104	0.00	0.00
jpeg_fdct_islow	1	4,583	16,291	4,583	16,291	0.00	0.00
jpeg_idct_islow	1	1,541	20,665	1,541	20,665	0.00	0.00
recon	1	1,824	9,319	1,824	9,319	0.00	0.00
fullsearch	1	43,082	244,305	43,085	244,025	0.00	0.00
whetstone	2	4,538,310	13,711,800	4,538,310	13,711,800	0.00	0.00
dhry	$8 \Rightarrow 3$	218,013	1,264,430	218,013	1,264,430	0.00	0.00
matgen	1	5,507	13,933	5,507	13,933	0.00	0.00

Table 3: Discrepancy between the estimated bound and the measured bound.

Function	Estimated Bound		Measured Bound		Pessimism	
	lower	upper	lower	upper	lower	upper
check_data	32	1,039	38	441	0.16	1.36
fft	966,652	3,346,000	1,928,477	2,048,959	0.50	0.63
piksrt	146	4,333	338	1,786	0.57	1.43
des	42,302	604,169	109,329	242,295	0.61	1.49
line	336	8,485	963	4,845	0.65	0.75
circle	502	16,652	641	14,506	0.22	0.15
longloop	12,254	14,104	13,556	13,557	0.10	0.04
jpeg_fdct_islow	4,583	16,291	7,809	10,062	0.41	0.62
jpeg_idct_islow	1,541	20,665	2,913	13,591	0.47	0.52
recon	1,824	9,319	4,566	4,614	0.60	1.02
fullsearch	43,082	244,305	62,463	62,468	0.31	2.91
whetstone	4,538,310	13,711,800	6,831,720	6,831,840	0.34	1.01
dhry	218,013	1,264,430	551,460	551,840	0.60	1.29
matgen	5,507	13,933	9,260	9280	0.41	0.50

of the routine. The best case (minimum) running time is obtained analogously.

Table 3 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 formulae:

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

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

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 pessimism is bigger when there are many small basic blocks in the function. This is because all the cost analysis is currently being done within the basic block. For basic blocks with only a few assembly instructions, the cache and pipeline analysis is not being modeled very accurately because they depend a lot on the surrounding instructions. Consequently, the costs of these basic blocks are loose and these contribute to the discrepancy between the estimated and the measured bounds. A more sophisticated microarchitectural modeling will certainly improve the accuracy of the estimated bound.

7 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.

The 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 non-deterministic. In addition, we would also like to explore the possibility of using symbolic analysis techniques to automatically derive some of the functionality constraints that are currently being provided by the user. Finally, 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.

References

[1] A. M. van Tilborg and Editors G. M. Koob, *Foundations of real-time computing: Scheduling and resource management*, Kluwer Academic Publishers, 1991.

[2] Eugene Kligerman and Alexander D. Stoyenko, "Real-time Euclid: A language for reliable real-time systems", *IEEE Transactions on Software Engineering*, vol. SE-12, no. 9, pp. 941–949, September 1986.

[3] P. Puschner and Ch. Koza, "Calculating the maximum execution time of real-time programs", *The Journal of Real-Time Systems*, vol. 1, no. 2, pp. 160–176, September 1989.

[4] Aloysius K. Mok, Prasanna Amerasinghe, Moyer Chen, and Kamtron Tantisirivat, "Evaluating tight execution time bounds of programs by annotations", in *Proceedings of the 6th IEEE Workshop on Real-Time Operating Systems and Software*, May 1989, pp. 74–80.

[5] Chang Yun Park, *Predicting Deterministic Execution Times of Real-Time Programs*, PhD thesis, University of Washington, Seattle 98195, August 1992.

[6] R. B. Hitchcock, "Timing Verification and the Timing Analysis Program", in *Proceedings of the* 19^{th} *Design Automation Conference*, June 1982, pp. 594–604.

[7] Srinivas Devadas, Kurt Keutzer, and Sharad Malik, "Computation of floating mode delay in combinational circuits: Theory and algorithms", *IEEE Transactions on Computed-Aided Design of Integrated Circuits and Systems*, vol. 12, no. 12, pp. 1913–1923, December 1993.

[8] Alfred V. Aho, Ravi Sethi, and Jeffery D. Ullman, *Compilers Principles, Techniques, and Tools*, Addison-Wesley, 1986, ISBN 0-201-10194-7.

[9] Alan C. Shaw, "Reasoning about time in higher-level language software", *IEEE Transactions on Software Engineering*, vol. 15, no. 7, pp. 875–889, July 1989.

[10] Byung-Do Rhee, Sang Lyul Min, Sung-Soo Lim, Heonshik Shin, Chong Sang Kim, and Chang Yun Park, "Issues of advanced architectural features in the design of a timing tool", in *Proceedings of the 11th IEEE Workshop on Real-Time Operating Systems and Software*. May 1994, pp. 59–62, IEEE Computer Soc. Press, ISBN 0-8186-5710-3.

[11] Intel Corporation, *i960KA/KB Microprocessor Programmers's Reference Manual*, 1991, ISBN 1-55512-137-3.

[12] Rajesh Kumar Gupta, *Co-Synthesis of Hardware and Software for Digital Embedded Systems*, PhD thesis, Stanford University, December 1993.

[13] Intel Corporation, *QT960 User Manual*, 1990, Order Number 270875-001.