Approximation of the Worst-Case Execution Time Using Structural Analysis

Matteo Corti
Department of Computer Science
ETH Zürich
Zürich, Switzerland

Thomas Gross
Department of Computer Science
ETH Zürich
Zürich, Switzerland

ABSTRACT

We present a technique to approximate the worst-case execution time that combines structural analysis with a loop-bounding algorithm based on local induction variable analysis. Structural analysis is an attractive foundation for several reasons: it delivers better bounds on the number of executions for each basic block than previous approaches, its complexity is well understood, and it allows the compiler to easily work on Java bytecode without requiring access to the original program source. There are two major steps. We first compute (min, max) bounds on the number of iterations for each loop. Then we use precise structural information to propagate these bounds to the whole control-flow graph and compute a bound for each basic block. Such a fine-grained result eases the identification of infeasible paths and improves the approximation of the worst-case execution time of a function or method. This analysis was successfully implemented in an ahead-of-time Java bytecode to native compiler and produces input for a worst-case execution time estimator. We describe the effectiveness in reducing the worst-case execution time for a number of programs from small kernels and soft-real-time applications.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*performance measures*; C.3 [**Special-Purpose and Application Based Systems**]: Real-time and embedded systems

General Terms

Algorithms, Experimentation, Measurement, Performance

Keywords

Worst-case execution time (WCET), real-time computing, structural analysis

1. INTRODUCTION

A central problem in the static approximation of the worst-case execution time (WCET) is the estimation of the maximal number of loop iterations. Then this number can be combined with information about the cost to execute instructions (i.e., the loop body) to obtain an estimate of the worst-case execution time. For many soft-real-time programs (e.g., an MP3 decoder), knowing the WCET allows better resource usage, but an underestimation of the execution time is not fatal. A tight upper bound on loop iterations can greatly reduce overestimations. Several techniques have been presented in the last years trying to extract enough semantic information from the object code or program source to understand the behavior of the induction variables of loops [6, 10].

The simplest idea is to let the user explicitly specify the minimal and the maximal number of loop iterations. This information must be supplied as annotations while writing the program. User annotations in the source code are easy to understand but have too many disadvantages to be useful in practice. The whole responsibility for the correctness of the bounding information is left to the programmer. Furthermore, the annotations must be updated every time the original program is modified; an omission may lead to errors in the WCET estimation, and annotations are difficult for complex kinds of information, e.g., infeasible paths.

One possible solution to this problem is to interpret the program in an abstract domain. The interpreter then keeps track of possible variable values on different execution paths [6, 22]. In many situations, it is then possible to determine when a loop finishes, or if a given path is feasible. Since it is impossible to analyze all possible paths in a program, usually heuristics are employed to reduce the complexity of the interpreter. Such heuristics may, however, overlook important information.

To avoid the twin problems of an explosion of the number of paths that are analyzed and the loss of precision caused by simplifications, other approaches have been presented. Healy et al. [10] propose to analyze loops one-at-a-time, looking at the behavior of induction variables only. This approach may gather less information than abstract interpretation but is guaranteed to finish in a reasonable time.

In this paper we present a technique that is able to precisely propagate the loop iteration bounds computed by this method to every basic block. Our technique is based on structural analysis, a specialized form of interval analysis used in optimizing compilers.

Several groups investigated the WCET analysis of real-time Java programs [2, 17] but since the algorithm presented here, although implemented in a Java compiler, is language independent, we will not address language-specific topics in this paper.

The paper is structured as follows: In Section 2, we present our compiler and instruction duration estimator. In Section 3, we outline the basic idea behind the structural analysis, and in Section 4 we explain how to derive precise bounding information for every basic block in the program's control-flow graph. In Section 5, we

EMSOFT'04, September 27–29, 2004, Pisa, Italy.

briefly describe how we estimate the duration of an instruction on a given architecture, and in Section 6 we present some experimental results. Section 7 contains the concluding remarks.

2. SYSTEM OVERVIEW

Our worst-case execution time approximator is integrated as a module in an ahead-of-time Java bytecode to native compiler. The static analyzer is able to bound the minimal and maximal number of iterations for various loops using the technique described in this paper (see Section 4) and is able to identify infeasible paths using abstract interpretation on linear code segments. The analyzer then includes the information about the program's behavior in the output assembler file in form of (text) comments, which are processed by the instruction duration estimator.

2.1 Partial abstract interpretation

Before starting any analysis on the semantics of loops we perform an abstract interpretation pass over linear code segments, trying to identify infeasible paths and trying to determine the set of possible values for as many program variables as possible.

Many techniques can be used to determine false paths in a program [1, 12], among them abstract interpretation [8, 7] is a common method in the field of worst-case execution time analysis since it can safely discover the run-time behavior of the analyzed program. The program is executed in an *abstract domain* keeping track of the possible variable values on each execution path. Due to an exponentially growing set of possible control flow paths in a program, some trade-off between precision and analysis time must be made. At control flow merge points (after an "if" or after a loop), information coming from different paths is merged. This step loses precision but, at the same time, reduces the problem complexity.

Since abstract interpretation keeps track of all the possible values of a variable on a certain path it is possible to derive important information such as loop bounds and *infeasible paths*, i.e., paths which are not executed under any combination of input values.

In our approach we limit the abstract interpretation to linear code segments, avoiding to iterate over loops. This simplification does not allow us to bound loops but still enables us to find out the range of possible values for many variables and to discover several infeasible paths. One obvious advantage is the significant reduction of the number of paths and the consequent decreased need to apply heuristic approximations. We limit the analysis to a subset of infeasible paths obeying the following rules:

- Infeasible paths cannot cross loop boundaries (but infeasible paths in loop bodies or infeasible paths containing loops are handled).
- Each variable determining an infeasible path cannot be modified at a loop nesting level greater then the one that contains the infeasible path itself.

Table 1 shows the results of the infeasible-path analysis alone, on a set of Java benchmarks: even without crossing loop boundaries, our tool is able to detect a number of infeasible paths on real applications.

Although we found some infeasible paths that are relevant to the WCET analysis since they correspond to the longest path of the method (see Table 2), the main purpose of the abstract interpretation pass is to detect the range of possible values for many variables allowing a better detection of the start or end values of a loop's induction variable.

A simple example is shown in Figure 1; here the range of possible values of `i` at the beginning of the method is unknown, but the possible values of `i` within the body of the "if" statement can easily be reduced to positive values allowing our tool to bound the loop.

Table 1: Infeasible paths in the SPECjvm98 benchmark suite.

Benchmark	Infeasible paths
`_201_compress`	2
`_202_jess`	3
`_205_raytrace`	7
`_209_db`	2
`_213_javac`	240
`_222_mpegaudio`	19
`_228_jack`	22

Table 2: Infeasible paths corresponding to the longest path.

Program	Infeasible longest paths
`javalayer`	2
`linpack`	2
`whetstone`	1

```
void foo(int i) {
  if (i > 0) {
    while (i < 100) {
      i++;
    }
  }
}
```

Figure 1: Example for bounded loop

2.2 Estimating loop bounds

Simulation or interpretation of a program in an abstract domain suffers from the exponential growth of the paths to analyze; therefore we choose to base the core of the loop iterations bounding module on a method developed by the Real-Time Systems Group at the Florida State University [10]. This technique analyzes loops by gathering information on when conditional jumps could change their direction (i.e., when there is a change in the result of the boolean expression that determines the direction of the jump). In this section, we briefly summarize this technique to introduce some concepts and terminology needed to understand our algorithm.

There are four main steps: First the program's control flow graph (CFG) is built and the loops and the nodes that could be responsible for loop termination are identified (*iteration branches*). These nodes are then linked using a precedence relation representing the order in which these nodes could be executed in a loop iteration. The resulting directed acyclic graph is called *precedence graph*. In a second step, for each of these nodes, it is computed when the conditional jump for each iteration branch could change its result based on the number of iterations. I.e., the algorithm computes, if possible, the iteration at which the control flow could change at such a node. In the next step, we determine the range of possible iterations when each of the outgoing edges is reached. In a fourth and final step, the maximum (and minimum) number of loop iterations is computed.

This powerful method is able to handle loops with multiple exits and a variable number of iteration, covering a great range of loops used in real-time applications. The method can also be extended to support equality operators in the iteration branches conditions and to support non-constant number of iterations such as in non-rectangular loops [11]. This extension requires data structures to handle non-contiguous integer ranges.

3. STRUCTURAL ANALYSIS

A limitation of the approach presented by Healy et al. is that although the minimal and maximal number of iterations for the loop header are a safe bound for every block in the loop, this approximation does not take into account different paths inside the loop body.

To propagate bounding information from a loop header to each program block we need precise knowledge of the control flow graph structure, which is normally not available in a standard compiler. In this section we shortly describe a technique due to Sharir [20, 16] that allows to derive semantic and structural information from a program's control-flow graph.

In Section 4 we show how this idea can be brought together with prior work on estimating loop executions to provide precise worst-case execution time estimations.

Structural analysis is an enhanced interval analysis, which is able to identify control-flow based on statically predefined structural patterns. The program's control-flow graph is decomposed into a hierarchical tree of subgraphs embedded into each other. These subgraphs, or regions, represent code patterns or semantic constructs and can be summarized as follows: blocks, if-then, if-then-else, while, natural-loop, repeat, proper and improper regions. Structural analysis handles complex structures and provides information in a practical way. This algorithm, in contrast to a normal analysis based on the dominator relation, can, e.g., easily recognize cyclic structures with more than one back-edge as a single loop (see Figure 2).

```
while (A) {
  if (B)
    C;
  else
    D;
}
```

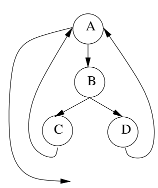


Figure 2: Loop with more than a back-edge.

Structural analysis proves particularly useful when the source code is not available as in a decompiler [3] or, as in our case, in an optimizing bytecode to native compiler [15].

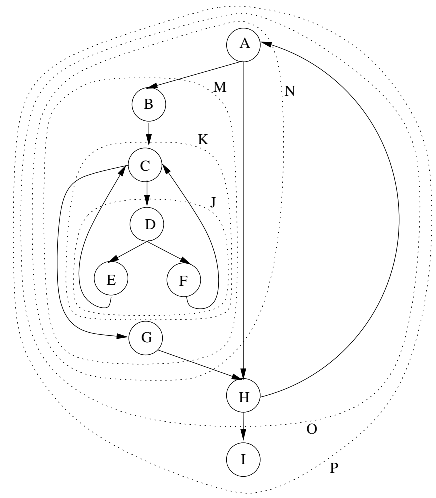
The structural analysis algorithm, as described by Sharir, builds a depth-first spanning tree of the control-flow graph and iteratively tries to recognize regions on the partially reduced graph. The nodes in the graph are compared to well-known predefined structural patterns, and if a match is found, the region is collapsed to form a new node. Figure 3 shows an example decomposition of a simple Java program.

3.1 Extension to complex boolean expressions

The comparison of small graph regions with a series of predefined patterns, as initially presented by Sharir, is not sufficient to handle every reducible control-flow graph, as some non-cyclic structures generated by complex boolean expressions in conditional statements do not follow a general scheme.

These complex boolean expressions are reducible regions that cannot be mapped to any know predefined pattern and are con-

```
do {
  if (A) {
    B;
    while (C) {
      if (D)
        E;
      else
        F;
    }
    G;
  }
} while (H);
I;
```




```
J:=If-then-else(D, E, F)   N:=If-then(A, M)
K:=While(C, J)             O:=Repeat(N, H)
M:=Block(B, K, G)          P:=Block(O, I)
```

Figure 3: Structural regions for a simple Java program.

nected to the rest of the control-flow graph by two *joint nodes*[1]. Since such a region has a unique entry point, every node is dominated by this region header. Figure 4 shows such a region (dashed line) for an if–then–else statement (the two joint nodes are shown in gray).

```
if (A || (B && C)) {
    D;
} else {
    E;
}
F;
```

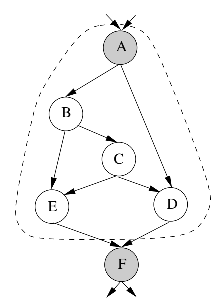


Figure 4: Region that cannot be represented by a predefined pattern.

To properly identify these commonly found regions, we developed the following graph coloring algorithm that is performed on the partially reduced graph: Starting from the node that is supposed to be the region header ($header$), after having tried to match it with a known pattern, we iteratively traverse the graph looking for the second joint node. We stop, and return an empty region, if we find a node that has a successor or a predecessor that is not dominated by the header. Otherwise we create a new region (*proper region*) containing the subgraph between the two joint nodes.

$W := successors(header)$
$R := \emptyset$
mark the $header$ as visited

[1]*Joint nodes* are a set of nodes, normally two, which connect a region of the graph to the rest of it, removing these *joint nodes*, this region remains disconnected.

```
while W ≠ ∅ do
  node := ◇W
  R := R ∪ node
  mark node as visited
  if a predecessor of node is not dominated by the header then
    (* this means that the header is not a joint node *)
    return ∅
  end if
  for all successors s of node do
    if s is not dominated by the header then
      (* we do not follow back edges *)
      return ∅
    else
      if s is not visited then
        W := W ∪ s
      end if
    end if
  end for
  return R
end while
```

Figure 5 shows an example with a snapshot of the traversal algorithm. Starting from node K (the possible header of the region) we traverse all the successors and mark them as visited (gray nodes). For each node we check if all its successors and predecessors are dominated by the header. In this case node L is part of the region while node M is not since node N or node O are not dominated by the header (K). This means that K is not a *proper region* header and we return the empty set.

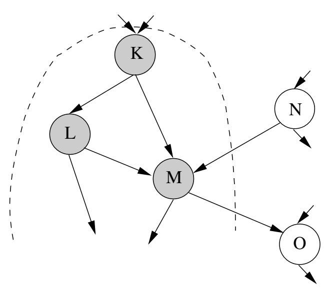


Figure 5: Snapshot of the traversing algorithm.

4. BASIC BLOCK ITERATIONS

In this section we show how bounds on loop iterations can be refined on a basic block level using structural analysis. Our idea is to propagate the loop minimal and maximal number of iterations to each basic block, adapting these values in accordance with the different number of iterations of the various paths inside the loop body. We compute a fine-grained per-block bounding information that allows the compiler to handle infrequent paths inside a loop body separately, as shown in Figure 6. The body of the "if" statement is executed only once before exiting the loop, and its duration should therefore not be included in each loop iteration.

The number of iterations for a block is not directly dependent on the block's predecessors but is, instead, determined by the type of the enclosing semantic construct or structural region, along with the dominator relation. Therefore the bounds cannot be propagated easily to every basic block. Figure 7 shows three simple examples of structural and semantic constructs that influence the number of iterations (shown between brackets) of a given basic block (striped node). The number of iterations of a block cannot be derived from its direct predecessors, but only from the header (gray node) of the biggest enclosing structural region (a loop in Figure 7.a and an "if–then–else" in Figures 7.b and 7.c [2]).

```
i = 0;
while (true) {
  if (i > 10) {
    // code executed once
    break;
  }
  // code executed 10 times
}
```

Figure 6: An example of a loop.

.

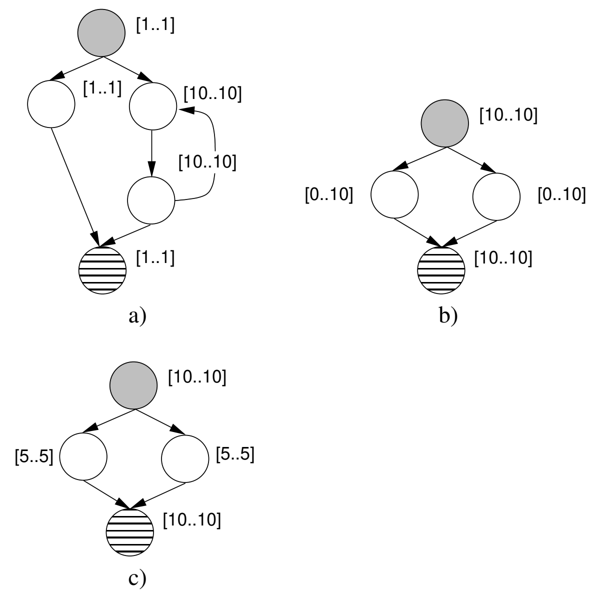


Figure 7: An example of block iterations.

.

To propagate the loop header bounding information to each block, we perform two main steps: In a first phase we propagate the minimal and maximal number of iterations along each loop's precedence graph (see Section 2.2). In a second phase we spread this information to every basic block in the control flow graph.

4.1 Execution counts for iteration branches

Loops are locally analyzed starting from the innermost one, allowing a simple handling of nested structures. Once the minimal and maximal iterations for the loop header ($[header_{min}..header_{max}]$) have been computed (e.g. with the method proposed by Healy et al. [10]), we propagate these values to the blocks with a conditional jump that could determine the number of iterations of the loop with a top-down traversal of the precedence graph (these node are usually called *iteration branches*; see Section 2.2).

For each iteration branch i we compute the smallest cyclic structural region scr that contains the node itself, detecting the presence

[2] In figure 7.b we assume that we have no information on the conditional branch of the gray node and we have to conservatively assume that both white nodes could be executed 10 times.

of inner loops. We then define for each node and each edge in the precedence graph in topological order a range $iter$ representing the minimal and maximal number of times the node or edge can be traversed.

If the smallest cyclic region scr corresponds to the loop l we are analyzing (this means that i is not part of an inner loop, we assign the range $iter$ to each outgoing edge e as follows:

1. If we know when the result of the conditional branch at the end of block i will change (in this case we say that i is *known*), we distinguish the following three situations:
   - If the maximum possible value of the range[3] of the edge e is smaller than minimum of $iter(i)$:
     $$iter(e) := [max(range(e))..max(range(e))].$$
     This means that the edge will always be taken for the allowed iterations, and that the minimal and maximal number of iterations correspond to the maximum of the range.
   - If the minimum of the range of the edge e is bigger than the maximum of $iter(i)$, the edge will never be taken since this node will never be executed enough times to allow the control flow to take the edge. Therefore we set:
     $$iter(e) := [0..0].$$
   - Otherwise we compute the range by considering the minimum and maximum of the range of the edge e and the minimum and maximum number of iterations $iter$ of the iteration branch i:
     $$\begin{aligned} iter(e) := [\ & max(min(iter(i)), min(rng(e))) - \\ & min(rng(e)) + 1.. \\ & min(max(iter(i)), max(rng(e))) \\ & - min(rng(e)) + 1\ ]. \end{aligned}$$
2. If i is *unknown* (i.e., we do not have any information about when the jump result will change) for each outgoing edge e we have to assume that it could be executed any number of times up to $max(iter(i))$:
   $$iter(e) := [0..max(iter(i))].$$

If the smallest cyclic region (scr) of i does not correspond to the loop l we are analyzing, i is in an inner loop. In this case we multiply the computed bounds by the minimal and maximal number of iterations of the inner loop. Note that while the edge ranges are dependent on loop iterations, as they indicate when the edge could be traversed, the iterations ($iter$) only indicate how many times a given edge will be traversed. The iterations for a given iteration branch i are then computed from its incoming edges using the following rules:

1. If the scr corresponds to the loop l we are analyzing, i.e., there are no inner loops, $iter$ corresponds to the sum of the iterations of each incoming edge if each predecessor is known:
   $$iter(i) := \sum_{p \in \text{predecessors}(i)} iter(edge(p, i)).$$
   Otherwise, as we do not have enough information on the previous nodes we safely approximate $iter$ with the $iter$ value of the smallest region header containing i.
2. If i is the header of an inner loop scr, we multiply the already computed iterations of i with the sum of the iterations of each incoming edge:
   $$iter(i) := \sum_{p \in \text{predecessors}(i)} iter(edge(p, i)) \cdot iter(i).$$
   In addition, we store in the node i a multiplication factor for the inner loop l defined by the region scr computed as follows:
   $$mul(l) := \sum_{p \in \text{predecessors}(i)} iter(edge(p, i)).$$
3. Otherwise, for inner loop nodes, we simply multiply the iterations of the inner loop node i with the region multiplication factor stored in the header of each outer loop l:
   $$iter(i) := iter(i) \cdot mul(l).$$

4.2 Basic blocks iterations

Now that the minimal and maximal number of iterations of each basic block that could influence the flow of control of the program is known, we have to propagate the bounding information to every block in the graph.

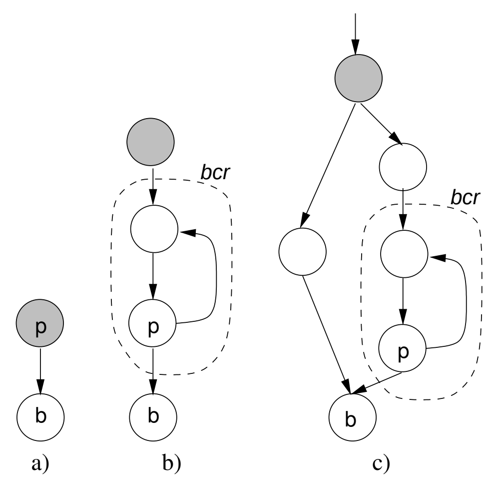


Figure 8: Biggest cyclic regions.

.

Once again we take advantage of the available structural information and we compute the number of iterations ($iter(b)$) for each basic block b. We treat the control-flow graph entry block differently, setting the minimal and maximal number of iterations to 1. For each node b we define $bcr(b)$ as the biggest cyclic region containing at least one predecessor p of node b and not containing the node b itself. If $bcr(b)$ exists, it means that the node b is the exit point of one or more loops and that $bcr(b)$ corresponds to the outermost one. If p is the single predecessor of b, and $bcr(b)$ is empty, we simply copy the range of iterations of the edge to the node (Figure 8.a). Otherwise, we have detected a loop breaking edge ($p \rightarrow b$), and the number of iterations of b is set equal to the number of iterations of the loop preheader[4] (Figure 8.b). If we have more than one predecessor, we look for the first common predecessor of all the predecessors of b (Figure 8.c). If the common

[3]The *range* of a node or an edge is defined as the set of loop iterations during which it is possible to execute this node or edge.

[4]The *preheader* is a special basic block artificially inserted by the compiler acting as a unique loop header predecessor.

predecessor is an inner loop header, we take the number of iterations of the corresponding preheader, since we detected multiple loop exits.

The following algorithm summarizes the computation of $iter(b)$ for nodes that are not iteration branches:

$P :=$ predecessors(b)
bcr$(b) :=$ biggest cyclic region containing $p \in P$ and not b
if $|P| = 1$ **then**
  $p := \diamond$predecessors(b)
  if p is not an iteration branch **then**
    $iter(b) := iter(p)$
  else
    if $bcr = \emptyset$ **then**
      $iter(b) := iter(p \rightarrow b)$
    else
      $iter(b) := iter(preheader(bcr))$
    end if
  end if
else
  $q :=$ first common predecessor of P
  if q is the header of a loop l **and** $b \notin l$ **then**
    $q := pheader(l)$
  end if
  $iter(b) := iter(q)$
end if

4.3 Complexity

One of the main advantages of this algorithm is that it is able to compute precise per block bounds in linear time over the number of iteration nodes while the time to perform the structural analysis is $O(B^2)$ where B is the number of basic blocks.

Healy and Whalley [12] presented a different approach using automatically detected value-dependent constraints. The compute per block bounds analyzing the possible outcome of conditional branches by looking at the induction variable changes on all the possible paths. The algorithm has a complexity of $O(P)$ where P is the number of paths in a loop.

4.4 Example

The following simple example shows how the iteration counts for basic blocks are computed (Figures 9 and 10).

```
for (i=0; i<100; i++) {
  if (i < 50) {
    if (i > 10 && somecond)
      break;
  } else {
    for (j=0; j<10;)
      j++;
  }
}
```

Figure 9: Example Java program.

First the bounds for these two loops are computing (see Section 2.2): $[12..101]$ for the loop identified by region `F` and $[11..11]$ for the loop identified by region `C`.

Following the rules described in Section 4.1, we compute the bounds for the iteration branches of both loops: 10, 2, 3, 4, 8 for loop F and 8 for loop C (gray nodes). At this stage we are already able to see that some areas of the graph have different iteration bounds depending on the information gathered for the iteration branches. The algorithm is able to reduce the number of iterations of the inner loop (region `C`) from $[12..1010]$ to $[11..550]$, reducing the inner loop duration by a factor of two.

In the last step we perform the last top–down traversal of the graph computing the bound for every remaining basic block as explained in Section 4.2.

5. INSTRUCTION EXECUTION TIME

Once the number of iterations of each basic block is bounded, we need to compute for each instruction the number of cycles it takes to execute. Since our algorithm works on the semantics of the program it is architecture independent and could be used with precise or approximated hardware analyzers.

There are several approaches trying to precisely compute or approximate the behavior of the different level of caches (instruction and data) and pipelines [9, 23, 14]. Other groups measure some of the paths using static analysis and code instrumentation [18].

On modern processors, the instruction duration depends on the status of the caches and pipelines. On systems supporting preemption and a dynamic set of processes it is therefore very difficult to compute an exact estimation of a process running time.

We are currently targeting soft-real-time applications on Linux running on the Intel IA-32 platform. For our application domain we do not need an exact result and we try to compute a good and conservative approximation of the worst-case execution time using some heuristics and approximations. We implemented a first prototype for the PowerPC platform which used statistic information to estimate the pipeline stalls and cache misses [4]. This approach was further refined replacing some of the empirical approximations with a partial and simplified simulation of the processor behavior [5].

It should be noted that we approximate the instruction duration only, and not the the number of times a given instruction (or block) will be executed. Since the instruction duration estimation is not the focus of this paper we will describe shortly the basic principle on which our estimator is based.

To precisely estimate the CPU state (pipeline and caches) before an instruction is executed and to exactly compute its duration, we should simulate all the paths leading to this instruction. Since a complete simulation of each possible execution trace is clearly not possible, we reduce the duration of the simulated code, relying on the locality of the effects that a given instruction can have on the hardware (pipeline and caches). We assume that the effects of an instruction on the pipeline and caches will be no more significant after a certain time.

For each basic block b we only simulate the last n instructions of each incoming path to approximate the pipeline and the content of the CPU's execution units before its first instruction. The value of n is specified as a parameter and normally lies between 50 and 100 instructions, such a value allows us to compute good approximations in a reasonable time. Experiments show that increasing n above this threshold, the WCET of an application does not change any more proving that our principle of locality holds.

During the simulation we use a simplified model of the CPU (an Intel Pentium III in our case) to approximate the execution of the pipeline updating the state of each processor unit. We can then analyze the duration of each instruction in the block b using the simulated initial CPU state.

In this way we can efficiently compute a realistic estimation of the duration of each instruction for each possible execution path. In a second step, each path is weighted according to its maximal number of possible executions (see Section 4). An instruction's duration is then computed as the weighted sum of all the durations for each possible incoming path. The sum of all the instruction of a block defines its duration which, together with its maximal number of iterations, is used to compute the longest path of the control flow graph.

At present we do not analyze the behavior of the processor's different levels of caches, although an instruction cache hit rate approximation could be integrated in the partial trace simulator. The hit rate is measured at run-time for each program with different traces, and the worst-case value is then used in the worst-case execution time computation.

6. RESULTS

Testing the soundness of a WCET predictor is a tricky issue since, for complex examples, the real maximum execution time is difficult or even impossible to measure. To compare the estimated values with the measured time we must force the execution of the longest path, which is not normally known. The easiest way to validate such a tool is therefore the comparison of the results for small known synthetic applications where the real longest path is known or computable by hand.

For this reason we first present results for a couple of small synthetic benchmarks that illustrate how our technique can help to reduce an eventual worst-case execution time overestimation due to infrequent paths inside a loop.

All the tests where performed on a 1-GHz Intel Pentium III-based PC. We compare our implementation of method proposed by Healy et al. [10] with and without the basic block refinement computed with structural information (see Section 4).

The instruction duration estimations are computed with the tool presented in Section 5 while the real measurements (to cross-check with the actual system) where done using the Pentium on-chip performance monitoring hardware.

In the first example (Figure 11.a) there is a simple loop that traverses an array and treats the first half of the elements differently. The WCET estimation with loop bounding (30.78 s) can be reduced to (24.38 s) using our analysis, a value that is close to the effective execution time used by the test program (24.467 s). The overestimation (26%) due to the conservative handling of the first "if" construct was successfully eliminated.

Note that since the analysis on the hardware level computes only a likely approximation of the instruction duration, the computed values can also be slightly smaller than the actual worst-case execution time.

In the second synthetic example (Figure 11.b) an inner loop is executed only in the last iteration of the main loop: In this case a precise handling of the different paths inside loop bodies avoids the need to assume that the inner loop is always executed. The WCET estimation for this example (0.08 s) is close to the measured value (0.08 s) while a simple bound on loop results in a huge overestimation (0.69 s) of the maximum running time. These small synthetic benchmarks show the soundness of our technique: We are able to propagate loop bounds to every basic block keeping track of different path frequencies.

Table 3 shows the results for some bigger applications. JavaLayer [13] is a pure Java library that decodes, converts and plays MP3 files (in our benchmark we decode some sample MP3s to raw audio data). SciMark [19] is a composite Java benchmark measuring the performance of numerical kernels occurring in scientific and engineering applications (FFT, SOR, sparse matrix multiply, Monte Carlo integration and dense LU matrix factorization). _201_compress[5] is part of the SPEC JVM98 benchmarks [21].

The first column represents the maximum execution time that we were able to observe (in cycles), while the second and the third are the estimated WCET using the approach of Healy et al. and our enhancement. Note that the maximum observed cycles do not necessarily correspond to the measured WCET since the input set causing the WCET is unknown.

All the tests present some improvement over the original algorithm. _201_compress shows a huge improvement since our algorithm is able to detect that a significant part of the compressor main loop is executed only every 10'000 iterations.

Although the actual instruction duration estimator leaves ample room for improvement, we can safely compare the two techniques: in both cases we get sound results bounding the maximum execution of the analyzed programs. Our enhancement allows us to detect infrequent paths inside loop bodies and therefore reduce the computed WCET of the program.

7. CONCLUDING REMARKS

This paper describes an approach to tighten the bounds on the number of iterations of a program's basic blocks using structural analysis.

The semantic structures, or language constructs, of the program are extracted from the Java bytecode binaries. This information, along with the precomputed loop iteration bounds, is then used to propagate the minimal and maximal iteration count to each basic block. As a result, more precise information is available in a fine-grained manner for each basic block.

Structural analysis is a valuable tool for worst-case execution time program analysis since this technique allows the compiler to reduce the maximal bounds of basic block iterations as shown in this paper. In addition, it allows the compiler to easily work with low-level representations that are similar to Java bytecode, e.g., native binaries. Structural analysis also provides valuable information that can help other analyses or optimizations. Good estimations of the WCET are important for many soft-real-time systems. Structural analysis is a technique that is used in many optimizing compilers, and as our system demonstrates, this technique can be employed also in the domain of soft real-time systems with good results.

8. ACKNOWLEDGMENTS

We thank Christoph von Praun for his contributions on the compiler, Fabiano Ghisla for the implementation of the infeasible path analysis, and Stephan Heimann for evaluation and tuning of the instruction duration estimator. We appreciate the thoughtful comments of the reviewers.

This work was funded, in part, by the NCCR "Mobile Information and Communication Systems", a research program of the Swiss National Science Foundation, and by a gift from Intel's Microprocessor Technology Laboratory.

9. REFERENCES

[1] P. Altenbernd. On the false path problem in hard real-time programs. In *Proc. 8th Euromicro Workshop on Real-Time Systems*, pages 102–107, L'Aquila, Italy, June 1996.

[2] I. Bate, G. Bernat, and P. Puschner. Java virtual-machine support for portable worst-case execution-time analysis. In *Proc. 5th IEEE Intl. Symp. on Object-Oriented Real-Time Distributed Computing*, pages 83–90, Washington, D.C., April 2002.

[3] C. Cifuentes. *Reverse Compilation Techniques*. PhD thesis, Queensland Univ. of Technology, School of Computing Science, 1993.

[5] We replaced the file input with a 4KB chunk of random bytes.

Table 3: WCET estimations (in cycles).

Benchmark	Maximum observed (cycles)	Base (cycles)		Enhanced (cycles)	
javalayer	$2.67{\cdot}10^{9}$	$1.49{\cdot}10^{10}$	(558%)	$1.30{\cdot}10^{10}$	(487%)
scimark2	$2.47{\cdot}10^{10}$	$2.12{\cdot}10^{11}$	(858%)	$1.42{\cdot}10^{11}$	(579%)
_201_compress	$9.45{\cdot}10^{9}$	$4.76{\cdot}10^{12}$	(50'370%)	$1.11{\cdot}10^{10}$	(117%)

[4] M. Corti, R. Brega, and T. Gross. Approximation of worst-case execution time for preemptive multitasking systems. In *Proc. ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, Vancouver, Canada, June 2000.

[5] M. Corti and T. Gross. Instruction duration estimation by partial trace evaluation. In *Proc. WIP Session 10th Real-Time and Embedded Technology and Applications Symposium*, Toronto, Canada, May 2004. IEEE.

[6] J. Gustafsson. *Analyzing Execution-Time of Object-Oriented Programs Using Abstract Interpretation*. PhD thesis, Department of Computer Systems, Information Technology, Uppsala Univ., 2000.

[7] J. Gustafsson. A prototype tool for flow analysis of object-oriented programs. In *Proc. 5th IEEE Intl. Symp. on Object-Oriented Real-Time Distributed Computing*, Crystal City, VA, April 2002.

[8] J. Gustafsson and A. Ermedahl. Automatic derivation of path and loop annotations in object-oriented real-time programs. *Journ. of Parallel and Distributed Computing Practices*, 1(2):61 – 74, June 1998.

[9] C. Healy, R. Arnold, F. Mueller, D. Whalley, and M. Harmon. Bounding pipeline and instruction cache performance. *IEEE Trans. on Computers*, 48(1):53–70, January 1999.

[10] C. Healy, M. Sjödin, V. Rustagi, and D. Whalley. Bounding loop iterations for timing analysis. In *Proc. 4th Real-Time Technology and Applications Symp.*, pages 12–21, Denver, CO, June 1998.

[11] C. Healy, R. van Engelen, and D. Whalley. A general approach for tight timing predictions on non-rectangular loops. In *WIP Proc. 5th IEEE Real-Time Technology and Applications Symp.*, pages 11–14, Vancouver, Canada, June 1999. IEEE.

[12] C. Healy and D. Whalley. Tighter timing predictions by automatic detection and exploitation of value-dependent constraints. In *Proc. 5th Real-Time Technology and Applications Symp.*, pages 79–88, Vancouver, Canada, June 1999.

[13] JavaZOOM. Javalayer. http://www.javazoom.net/javalayer/javalayer.html.

[14] T. Lundqvist. *A WCET Analysis Method for Pipelined Microprocessors with Cache Memories*. PhD thesis, Chalmers Univ. of Technology, Göteborg, Sweden, June 2000.

[15] J. Miecznikowski and L. Hendren. Decompiling Java using staged encapsulation. In *Proc. Working Conf. on Reverse Engineering*, pages 368–374, Stuttgart, Germany, October 2001. IEEE.

[16] S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997.

[17] P. Persson and G. Hendin. An interactive environment for real-time software develpment. In *Proc. 33rd Tech. of Object-Oriented Languages Intl. Conf.*, Saint-Malo, France, June 2000. IEEE.

[18] S. Petters. Bounding the execution time of real-time tasks on modern processors. In *Proc. 7th Intl. Conf. on Real-Time Computing Systems and Applications*, Cheju Island, South Korea, December 2000.

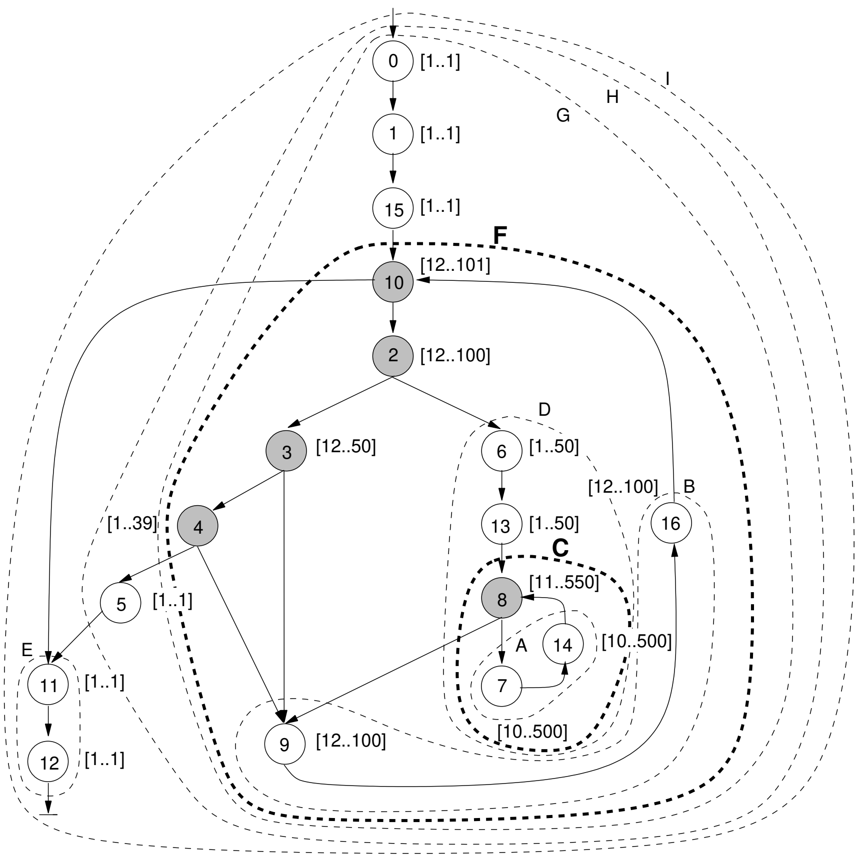
[19] B. Pozo, R. Miller. Scimark2. http://math.nist.gov/scimark2/.

[20] M. Sharir. Structural analysis: a new approach to flow analysis in optimizing compilers. *Computer Languages*, 5(3-4):141–153, 1980.

[21] The Standard Performance Evaluation Corporation. SPEC JVM98 Benchmarks. http://www.spec.org/osg/jvm98, 1996.

[22] H. Theiling and C. Ferdinand. Combining abstract interpretation and ILP for microarchitecture modelling and program path analysis. In *Proc. 19th IEEE Real-Time Systems Symp.*, pages 144–153, Madrid, Spain, December 1998. IEEE.

[23] H. Theiling, C. Ferdinand, and R. Wilhelm. Fast and precise WCET prediction by separated cache and path analyses. *Real-Time Systems*, 18(2/3):157–179, May 2000.




```
A := Block(14, 7)          B := Block(16, 9)
C := While(8, A)           D := Block(C, 13, 6)
E := Block(12, 11)         F := Natural-loop(10, 2, 3, 4, B, D)
G := Block(F, 15, 1, 0)    H := If-Then(G, 5)
I := Block(H,21)
```

Figure 10: Example control flow graph.

```
for (i = 0; i < 10000; i++) {
    if (i < 5000) {
        array[i] = -array[i];
    }
    if (array[i] > max) {
        max = array[i];
    }
}
```

a)

```
for(i = 0; i < 10; i++) {
    for (j = 0; j < 10; j++) {
        if (j < 9) {
            m[i][j] *= m[i][j];
        } else {
            for (k = 0; k < 9; k++) {
                m[i][j] += m[i][k];
            }
        }
    }
}
```

b)

Figure 11: Test programs.