Using Measurements to Derive the Worst-Case Execution Time

Markus Lindgren, Hans Hansson and Henrik Thane
Mälardalen Real-Time Research Centre, Mälardalen University, Sweden
www.mrtc.mdh.se

Abstract

Execution time analysis is used in the development of real-time and embedded systems to derive the timing estimates required for schedulability analysis. The execution time of the analyzed program is typically obtained by combining results from program flow analysis (such as number of iterations in loops) with low-level timing information.

This paper proposes a method for low-level timing analysis based on measurements of execution times of programs executing on the actual target architecture. The essence of the method is to derive a system of linear equations from a limited number of timing measurements of an instrumented version of the considered program. The solution to these equations give execution times for program fragments, from which execution time measures for the entire program can be derived. The main advantage with this approach is that architectural modeling is not needed, hence the risk of a discrepancy between model and real system is avoided. Also, compared to the non-exhaustive measurements performed in industry today, our approach is more structured and gives complete coverage in terms of the program paths considered.

We present our method in the context of a simple, but reasonably realistic, processor model and show how it can be extended to architectures with pipelines.

1. Introduction

Worst-case execution time (WCET) analysis or WCET estimation is crucial for successful analysis and implementation of hard real-time systems. The goal of WCET analysis is to generate a safe (i.e. no underestimation) and tight (i.e. small overestimation) of the worst case execution of a program. When performing WCET analysis it is assumed that the program execution is uninterrupted and that there are no interfering background activities, such as DMA or refresh of DRAM. Timing interference caused by such resource contention should be handled by subsequent analysis, typically by the schedulability analysis.

The process of analytically generating a WCET estimate from a program can be divided in high-level analysis (also referred to as program flow analysis), low-level analysis, and calculation. The task of the high-level analysis is to determine the possible paths through the program. The result is information on which functions get called, how many times loop iterate, if there are dependencies between different if-statements, etc. In the low-level analysis, the execution time of program fragments (typically basic blocks) on a particular target system is estimated. This analysis can be partitioned into global low-level analysis, in which the execution time effects of machine features that reach across the entire program are considered (e.g., cache effects), and local low-level analysis, which handles machine timing effects that depend on a single instruction and its immediate neighbors. Examples of such effects are pipeline overlap and memory access speed. Finally, in the calculation phase the WCET estimate is calculated, given the high-level and low-level analysis results. (For a more detailed and thorough overview of issues and work in the area of WCET analysis we refer to [15, 7].)

This paper is focused on the low-level estimation of WCET. Such analysis is becoming increasingly difficult due to new processor features, such as pipelines, caches, and branch prediction units. Modeling of processors’ timing behavior has been the way research has been approaching this problem, but this may become impossible in the future, since processor manufacturers are not always releasing information about the internal structure of processors, making it hard to create accurate models. Another difficulty, which afflicts all kinds of modeling, is how to determine if a model is correct or not. Using an incorrect model will, of course, invalidate any analysis based on it. The ever-increasing flora of processors is yet another problem, since each new processor family requires its own timing model.

In contrast, industry has relied on measurements of execution times when designing real-time software. The current practice in industry is however a trial and error process where no guarantees can be given since exhaustive testing/measuring seldom can be performed [14, 17].

The major problem facing measurements of WCET is
Consider for example a program with a single 32-bit integer as input. To be sure to find the WCET we would need to measure the execution time of the program for all possible inputs, i.e., make $2^{32}$ measurements (assuming that the program does not rely on internal state). Unless the execution times of programs are extremely short this process would take years to perform. The number of required measurements depend both on the program being measured and the hardware itself; for some architectures it is sufficient to test all paths through the program (although this is also time-wise intractable for nontrivial programs). The question is then if the number of measurements can be reduced, while still being able to produce safe timing estimates.

In for example [16], it has been proposed that the program should be divided into smaller parts, e.g., basic blocks, which are then measured in isolation. This is a reasonable approach for some processor architectures, but not for all, since the timing of many processor features depend on the history of execution, i.e., the path and variable bindings prior to reaching an instruction. The history is especially important in analyzing pipelines, locality for caches, and variable bindings for certain types of instructions, e.g., division. Hence, timing behavior is often context dependent. Hence, care has to be taken when measuring execution time on isolated parts of software.

Fortunately enough from a WCET analysis perspective, the majority of processors used in embedded applications are simple 4-, 8-, and 16-bit processors for which WCET analysis is a simple matter of adding up execution cycles for individual instructions. This corresponds well to the sum of basic blocks execution time model, used in e.g. [10]. This model assumes that the execution time of a sequence of basic blocks is the sum of the individual execution times of the included basic blocks. Furthermore, the more advanced processors used in embedded and safety-critical applications, such as the popular ARM micro-controller family [1], have a single simple pipeline, and very few have caches. In fact, for critical applications on-chip caches are usually traded for ROM/RAM to which critical code and data can be statically allocated. Hence, methods for WCET estimations of relatively simple processors has a potential of satisfying the execution time analysis needs for a majority of embedded and safety-critical applications.

The main contribution of this paper is that it presents a novel solution to the low-level analysis problem for relatively simple architectures, i.e., those with no global low-level effects and (possibly) with a simple pipeline.

The proposed method derives the execution time of program fragments by measuring the execution time of the whole program. Faced with intractable quantities of test cases for exhaustive execution time measurements, the method presented in this paper only requires a limited number of measurements of the entire program in order to derive the execution times of the program fragments.

By executing and measuring the execution time of the entire program, instead of just measuring program fragments in isolation as done by [16], we avoid the context sensitivity of certain instructions. The method provides timing information of program fragments to be combined with high-level WCET analysis providing information about loop iterations and feasible paths. Hence, the method assumes a context with high-level analysis and calculation methods, e.g., as proposed in [7].

The outline of the paper is as follows. In Section 2 we give an overview of our method. Section 3 describes which timing information can be derived from programs, and how this can be performed. In Section 4 we show how the method can be extended to handle pipelines and other effects related the sequencing of basic blocks. Section 5 illustrates the method by a simple example. Section 6 presents related work and in Section 7 we conclude and outline some future work.

2. Method Outline

The method for timing program fragments presented in this paper assumes input in the form of a control flow graph (CFG) of the analyzed program and assumes an execution time analysis (WCET) module calculating the final WCET estimate based on both flow information, such as maximum number of iterations of loops, and the timing of program fragments provided by our method. A typical context handling this is illustrated in Figure 1. It should be noted that we assume a direct correspondence between the CFG and the assembler code. Hence, there may, or may not, be a correspondence with the source code, depending on the transformations/optimizations performed by the compiler. How optimizations influence execution time analysis and the program flow is treated in [5], but is outside the scope of this paper.

The basic idea of our approach is to describe the entire control flow of a program by a set of linearly independent paths, as used in the classical complexity measure by McCabe [12]. A set of independent paths is for a well-

![Figure 1. The context of our method for timing program fragments](image-url)
structured program just a fraction of the number of possible execution paths through the program. This allows us to reduce the number of measurements to a set of linearly independent paths or basic blocks.

In essence the method works in five steps:

1. Identify a set of linearly independent paths or basic blocks. We will here assume that a separate flow analysis module has done this.

2. Instrument the target software in order to be able to determine which independent paths or basic blocks that are visited when executing the program.

3. Run and measure the execution time of the target software, while monitoring the paths or basic blocks that have been visited. This has to be repeated with different input data until a sufficient number of paths have been covered (see step 4 and 5).

4. Set up an equation system of linearly independent equations representing the visited paths/basic blocks as unknowns and the measured execution times of the target program for these paths/basic blocks.

5. Solve the equations to obtain execution times that can be fed into the calculation of the final WCET value.

As an illustration, consider the simple CFG in Figure 2. Which timing attributes should we derive? For Figure 2, this could for example be the execution time (ET) for the left path through the loop in Figure 2, the ET for basic block \( a \), or the ET for basic block \( a + d \). Typically, to be able to calculate the WCET of a program we need the execution times for the different paths through the program, expressed either as execution times of individual basic blocks or combinations of basic blocks.

Once we know what to derive, the code is instrumented so that we can retrieve information about how many times the different pieces of code are executed. For example, that the left part of the loop was executed 9 times for some input.

Since we assume the sum-of-basic blocks model [13], the ET for a program equals the sum of the ET for the basic blocks that are executed. This means that if we run, for instance, 5 tests with different input data we may end up with the following 5 equations defining the execution time of the different tests (where \( ET(x) \) denotes the execution time of basic block \( x \)):

\[
\begin{align*}
(1) \quad & ET(a) + ET(c) + ET(d) = 12 \\
(2) \quad & 3 \cdot ET(a) + 2 \cdot ET(b) + ET(c) + 5 \cdot ET(d) = 46 \\
(3) \quad & 3 \cdot ET(a) + 2 \cdot ET(b) + ET(c) + 3 \cdot ET(d) = 46 \\
(4) \quad & 2 \cdot ET(a) + ET(b) + ET(c) + 2 \cdot ET(d) = 29 \\
(5) \quad & 25 \cdot ET(a) + 24 \cdot ET(b) + ET(c) + 25 \cdot ET(d) = 420
\end{align*}
\]

To solve such a set of linear equations with four unknowns, we need four linearly independent equations. Unfortunately, we only have three linearly independent equations above (since \( (2) = (3) + 2 \cdot ((2) - (1)) + (1) = (5) \)). Even worse: by a simple inspection of the CFG we realize that basic blocks \( a \) and \( d \) are both executing in each iteration of the loop, i.e., they are linearly dependent. This means that we cannot by measuring execution times of the entire program distinguish them. Hence, we have to view them as the single composite basic block \( ad \). This gives us the three unknowns \( ad \), \( b \) and \( c \), together with the three equations:

\[
\begin{align*}
(6) \quad & ET(ad) + ET(c) = 12 \\
(7) \quad & 3 \cdot ET(ad) + 2 \cdot ET(b) + ET(c) = 46 \\
(8) \quad & 2 \cdot ET(ad) + ET(b) + ET(c) = 29
\end{align*}
\]

Solving this system of equations gives us \( ET(ad) = 10 \), \( ET(b) = 7 \) and \( ET(c) = 2 \).

Now, assuming that the high-level analysis has concluded that the maximum number of iterations of the loop is 30 and that the \( c \)-branch in the loop is executed exactly once, we can conclude that the WCET is \( 30 \cdot ET(ad) + 29 \cdot ET(b) + ET(c) = 305 \). A value which was obtained by a limited number of tests, compared to performing exhaustive testing, which in most realistic cases is intractable.

Limitations of the Approach

It should be noted that this solution only has general validity if the execution times of paths are input data independent. This is true for the majority of instructions for most simple architectures, but does not always hold for more advanced instructions, such as division (and in some cases multiplication). We plan to address this in future work.

Furthermore, the method is not applicable to systems with caches. The class of applications we target almost exclusively configure caches to act as fast RAM/ROM for parts of the address space.

In Section 4 an extension capable of handling simple pipelines is presented. This extension is, however, not applicable for super scalar processors, which on the other hand are rarely used in critical systems.

3. Path Analysis

In this section we elaborate on the timing information that can be derived, and how the code should be instrumented. We also present an algorithm that decides which timing attributes to derive.
For some programs the number of unknowns to solve is larger than the number of measurements that can be obtained, as illustrated by the example in Section 2. In such cases, some blocks can be grouped into larger program fragments to obtain a solvable system. This perfectly valid as long as these groupings are consistent with the entities needed in the calculation phase, which requires the entities to be consistent with those used by the high-level flow-analysis tools, e.g., we may combine two basic blocks that execute in exactly the same loop iterations, but not blocks executing in different alternatives of a selection. By grouping basic blocks we reduce the number of unknowns in the equation system, and consequently the number of measurements that need to be performed.

3.1. Control Flow Graph Reduction

The reduction method starts by analyzing all inner loops in the CFG of the assembler code, and decides which timing attributes that should be solved in each of these loops. The inner loops are then replaced by a single block in the CFG, to denote that it is known what to solve in this loop. This can create new “inner” loops in the CFG, which are handled using the same approach. This is repeated until there are no remaining loops in the CFG.

When deciding what to solve in an inner loop, we first group blocks that always are executed equally many times for all input data. These are grouped into one program fragment, with a single unknown in the equation system.

After “sequential” blocks have been grouped, the remaining unknowns are counted in the inner loop. If there are more paths than unknowns, then a valid grouping has been found (which also can be solved, given that we succeed in measuring sufficient number of independent paths). If there are more unknowns than paths, then the timing for sub-paths within the inner loop is derived. It should be noted that a solvable equation system based on sub-paths can always be found.

3.2. Fragments Identification Algorithm

Here we present an algorithm, which identifies the program fragments that should be monitored. The algorithm uses a CFG of the assembler code for the program, making it applicable to many different kinds of programming languages. The algorithm assumes that the CFG is well structured, i.e., each loop has a single entry and exit point, and backward edges are only allowed to loop entry points.

Algorithm outline:
For all inner-loops:
1) Group “sequential” basic blocks into fragments.
2) Count fragments, $c_f$, and independent paths, $c_p$, (in the “grouped” CFG). If $c_f \leq c_p$ solve timing for fragments, else solve for paths. Instrument each fragment, and enter one unknown for each fragment into the equation system. Replace the inner-loop in the graph with a single block, which can cause new loops to be classified as inner loops.

It is important to note that if we cannot reduce the graph so that sufficiently few unknowns are obtained, we can resort to the method of deriving timing for sub-paths in the program. The condition $c_f \leq c_p$ determines this.

Repeat step (1) and (2) until there are no more loops.

We now have a method to identify the unknowns of a solvable system of linear equations. However, a solution can only be found if sufficiently many independent paths can be executed. Special cases are treated below.

3.3. Feasible Paths

Some real-time programs contain code that has a single feasible path (SFP). For example, filters can have a single path through a large set of basic blocks. This prevents our approach from finding sufficiently many independent measurements. However, it makes little sense to provide detailed timing information for parts of code that has the SFP property. If code that contain SFPs can be identified we can group these basic blocks to a single fragment, i.e., a single unknown in the equation system, and solve the timing for the SFP. A method for determining if parts of a program have SFPs is presented in [8].

The number of feasible paths in a program can limit the number of obtainable independent measurements. Should it be that the infeasible paths prevent us from solving the equations, the grouping into program fragments has to be changed. Paths that the high-level analysis have identified as infeasible can in some cases be compensated for in the above identification algorithm, by adjusting the $c_f$ and $c_p$ counts. In other cases the analysis algorithm will prompt the user to manually change the grouping into program fragments, by for example selecting parts of the feasible paths or the entire paths. It should be noted that such modifications may correspond to removing some paths that have not been proven to be infeasible. Hence, there is definite risk that the obtained WCET is unsafe, since one of the removed paths may correspond to the actual worst-case. On the other hand, if no manual modifications are needed we have complete coverage in terms of the paths considered.

A special case of infeasible paths are those containing dead code. Deciding whether a piece of code is dead is in the general case infeasible [18], but methods exists for special cases [2, 9, 4]. Paths/equations corresponding to dead code can safely be removed.
3.4. Monitoring Executions

The idea of our method is to run the program, measure the execution time, and observe the number of times each program fragment is executed. To retrieve information about the program fragments that are executed there are a couple of approaches that can be used:

**Instrumentation**: Instrument the code in such a way that information on which blocks are executed can be obtained. This is performed by inserting extra code in the beginning of each program fragment to increment an associated counter. A disadvantage with this approach is that these probes increase the code size and execution time. The probes can, however, be avoided by running the test in two phases:

1. Instrument the program (as above) and execute it to observe which fragments are executed.
2. Do the measurement again with the same input data, but without the probes, and measure the execution time. This requires twice as much time, but alleviates the probe-effect problem altogether.

The input data used for the program can be generated randomly, by some heuristic method or taken from logs of the program executing in its target environment.

**Control-flow analysis**: By using control-flow analysis we can derive input data for which the execution path is known. This information can be derived by, e.g., symbolic execution [4]. A disadvantage with this is that it may require considerable amounts of computing resources. However, this method may be faster than other approaches since it can effectively find “good” input data.

3.5. Measuring Execution Times

There are several possible approaches to measuring execution time, each with its own advantages and disadvantages. Here we review some of the possibilities [3, 17, 14].

**Using Clock Cycle Counters**: A clock cycle counter counts clock cycles on the bus or from the CPU core. The benefits are that this approach is exact, time can be measured with a fine granularity, and it is non-intrusive, i.e., it does not disturb the execution of software in any way.

**Using Timers**: Timers can be found in most systems. These can be implemented in two different ways: fully in hardware or a combination of software and a periodic interrupt to the processor. Real-time clocks (RTCs) are used to implement the hardware (only) solution, and is accessed in a similar way as clock cycle counters, with the difference of having a lower resolution. In this paper we assume that the measurements are correct and exact, i.e., that clock cycle counters are used. However, the method can with some effort be used on systems with timers only.

4. Extensions for Pipelines

The method presented this far is appropriate for architectures where the execution time of basic-blocks are fixed and independent of the surrounding code. This is not the case for pipelined architectures. In this section we will show how our method can be extended to also handle architectures with simple pipelines.

In a pipelined architecture the execution of instructions is divided into several phases which for different instructions can be performed in parallel, e.g. while the operation of one instruction is being performed, the result of the preceding instruction may be stored and the operands of the succeeding instruction fetched. This parallelism and instruction overlap increases the processor speed. On the basic-block level, the execution of one basic block may overlap the execution of another, as illustrated in Figure 3. The effect of this overlap is a reduction in execution time, i.e., for the example in Figure 3, \( WCET(A) + WCET(B) > WCET(A + B) \), which violates our sum-of-basic-block assumption.

The way we avoid this problem is for the basic blocks in Figure 3 illustrated in Figure 4. As can be seen, we make an artificial partitioning of the basic blocks so that the “artificial” basic blocks do not overlap. This is done by separating the overlapping portions of the original basic blocks into an interface basic block. In the example we combine the overlapping parts of \( A \) and \( B \) into the artificial block \( AB \) and rename the remaining parts of \( A \) and \( B \). The outer edges of \( A \) and \( B \) will similarly be combined with preceding and succeeding basic blocks. The limitation of this approach is that it does not cover overflows of multiple basic blocks. Our simple pipeline assumption means exactly this: we only allow overlap with immediate neighboring basic-blocks. This is, however, not a severe limitation, since more advanced pipelines are rare in current micro-controllers.

To see the effect of our pipeline handling for the analysis of an entire program, consider the graph in Figure 5 which is derived from the CFG in Figure 2. The added blocks in diamond boxes in the figure denote the artificial blocks corresponding to the overlap of original blocks.
The obtained graph satisfies our requirements on CFGs on which our program fragment identification algorithm (Section 3.2) can be applied (if we regard all nodes as basic blocks). Hence, we can use the algorithm to identify a solvable system of equations.

Instrumentation

To retrieve information about when these new blocks are executed, a new instrumentation policy must be employed.

Each block (boxes in Figure 5) is assigned a unique identifier. When a block is executed, it writes its identifier to the global variable prev_block, which is used to store the identifier of the previously executed block. In addition, each block has one counter for each possible preceding basic block. At the beginning of each basic block, code which increments the counter associated with the block identified by prev_block is inserted. The number of times the block itself has executed equals the sum of its associated counters. From this we can derive how many times the different head and tail blocks are executed.

This instrumentation requires about two extra assembler instructions per basic block, and one counter for each predecessor of each basic block. The overhead (memory and code) can be eliminated by running the tests in two phases as described in Section 3.4.

“Pipeline effects” in non-pipelined architectures

It should be noted that the execution time of a basic block may even for non-pipelined architectures vary depending on which exit is taken, e.g., the execution time of conditional branch instructions is for most architectures dependent on whether the branch is taken or not. There may also be an influence from the preceding basic block, since different incoming paths to the block may lead to different status of prefetching and phasings of interleaved memory interfaces, leading to path dependent execution time of the basic block. Consequently, even for non-pipelined architectures pipeline type of effects between basic blocks are common. Our pipeline modeling will, however, accurately handle all such effects that are limited to only influencing the immediate neighboring basic blocks.

5. Case Study

Since we are not aware of any comparable technique, apart from traditional WCET analysis and random or exhaustive testing, we cannot perform direct comparisons with other work. Instead we present a small case study, which illustrates the usefulness and details of the approach. The CFG for the case-study program after the fragment identification algorithm has been applied is shown in Figure 6, where annotations for blocks due to pipelining have been added. Shaded boxes indicate blocks that have been grouped, the text inside the boxes (A–N) identifies the grouped blocks. Pipelining effects are taken into account.

The reduced CFG’s inner loop contains four fragments and three independent paths. Solving timing for sub-paths is decided for this loop, since the test \( c_f \leq c_p \) fails (4 > 3). (The paths are: (1) \( B3 \to B4 \to B6 \), (2) \( B3 \to B5 \to B6 \) and (1) \( B6'' \to B3' \to (1) \)).

Next we count the number of fragments in Figure 6 and decide if we should go for a path or fragment solution in the outer loop. There are 10 fragments (excluding the inner loop) and 10 independent paths, hence fragments can be solved in the outer loop (the test \( c_f \leq c_p \) succeeds). We
have now identified the fragments in the program and decided which timing information should be solved. Next we have to instrument the program.

Boxes and diamonds in Figure 6 indicate how the fragments need to be instrumented. A group becomes a box if there is at least one basic block in the group, i.e., pipelining effects can in some cases be included in basic blocks.

A simple way to instrument the code is the following: Assign a unique number (id) to each box and basic block preceding a diamond box. Each box is also assigned a set of counters (cn), one for each assigned id. Insert the code below at the start of each box fragment (some processors also require that the status register is stored before this code, and restored afterwards).

Before performing a measurement all counters are set to zero, and after the program has executed the corresponding equation is obtained by inspecting the counters and recording the execution time. The count for a box fragment becomes the sum of its associated counters.

To obtain timing for the decided upon fragments we must find input data, as described in Section 3.4, that reveals sufficiently many equations. An example set of independent paths for Figure 6 are:

\[
\begin{align*}
A + I + O + J + N &= t_1 \\
A + I + O + J + L + O + J + N &= t_2 \\
A + I + O + K + N &= t_3 \\
A + I + O + K + L + O + K + N &= t_4 \\
A + B + C + E + G + N &= t_5 \\
A + B + C + E + H + N &= t_6 \\
A + B + C + F + G + N &= t_7 \\
A + B + C + F + H + N &= t_8 \\
A + B + D + E + G + N &= t_9 \\
A + B + D + E + H + N &= t_{10} \\
A + B + D + F + G + N &= t_{11} \\
A + B + D + F + H + N &= t_{12} \\
2A + 2B + 2D + 2F + 2H + N + M &= t_{13} \\
5A + 3B + 3D + 3F + 3H + N + M &= t_{14}
\end{align*}
\]

All other paths in this program can be expressed as linear combinations of the paths above.

For the left part of the program (fragments \(A-H\) and \(N\)) we solve the equations as follows:

\[
\begin{align*}
X = A + B + N & \quad C = Z + t_3 \\
Y = E + G & \quad D = C + t_6 \\
Z = X + Y & \quad Z = C + t_3 \\
\end{align*}
\]

Similarly we can solve \(E, F, G, H, A + B, N\) and \(M\).

For the right part of the program (\(A, I, L, O\)) we solve \(R, R', S\) and \(S'\):

\[
\begin{align*}
R &= A + I + O + J \\
R &= t_1 \\
R' &= L + O + J \\
R' &= t_2 - N
\end{align*}
\]

If we were to derive timing for the same program on a processor without pipeline, the CFG would contain fewer blocks, but would still be equally many (or more) independent paths. Hence, finding the desired timing information would be easier.

Above it has been shown that the equations can be solved for the example program. However, we did not give any information about how a system designer/engineer actually would obtain sufficiently many equations from a program.

This would typically be carried out in parallel with functional testing of the program, registering executing time and equations while performing the tests. The equations would then be reduced (by tools) to a set of linearly independent equations. The missing equations, which are likely to be few, are obtained by 1) identifying them, which is done by inspecting the current equations, 2) deciding upon fragment counts that should be obtained and 3) finding input data (see Section 3.4) that executes “along” the decided values.

Assuming that we from high-level analysis know that both the inner and outer loop will at most execute 10 times, a safe estimate of the programs WCET can be calculated as follows (not considering infeasible paths):

\[
WCET(P) = \max\left\{10M + N + 11 \cdot \max\left(\frac{R}{100}, S + 10s\right), 10M + N + 11 \cdot \left(W + \max(C, D) + \max(E, F) + \max(G, H)\right)\right\}
\]

### Industrial Practice

The industrial practice used in finding the WCET of a program is to apply different loads (input data) and measure the execution time for a period of time. This process lasts until someone decides that enough time and resources have been spent. Random tests are typically used.

The theoretical upper limit on the number of paths for this program is (backward edges can be traversed at most 10 times each):

\[
\begin{align*}
\text{Inner loop: } p_i &= 2 + 2^2 + 2^3 + \ldots + 2^{11} = 4096 \\
\text{Outer loop: } p_i &= p_i + p_i^2 + p_i^3 + \ldots + p_i^{11} + 8 + 8^2 + 8^3 + \ldots + 8^{11} \\
p_i &\approx 5.42 \cdot 10^{43}
\end{align*}
\]

If we assume that 1 ppm of these paths are feasible and that a measurement/test on average requires 1 ms, the time for performing exhaustive testing becomes \(1.7 \cdot 10^{25}\) years! The time required for finding input data that executes along these paths must also be added (typically done by partitioning the input data into equivalence classes). Clearly, this is not a viable approach.

Instead random testing is performed for, say, a week, and 30% is added to the measured worst-case. During a weeks testing 6 - 10^8 cases can be covered; an extremely small part of the total state space of 5.42 · 10^{43} paths. Hence, no guarantee can be given to whether the measured worst-case is safe or tight. This should be compared to the 14 measurements required by our approach. Even though more tests may be needed due to infeasible paths, still the state space is reduced by several magnitudes!

### 6. Related Work

Not many have studied the problem of finding the execution time of parts of programs, on a processor with an
unknown timing model. Here we review some related work.

In [14] a method for measuring the worst-case execution time of programs is presented. The method relies on the actual hardware for measurements, and tackles the problem of too many paths by dividing the program into so called “measurement blocks”. Compared to our work this method does not produce safe values.

In [3] on-chip performance monitors are used to retrieve information on number of pipeline stalls, cache misses etc. An approximation of the execution time of basic blocks is derived from this. Our work does not require any performance monitors, and hence it can be applied to a wider range of hardware architectures (even though we do not handle caches).

Finding WCET of programs using dynamic testing methods has been studied by several research groups. Approaches using Genetic algorithms seem to be the most successful ones, see e.g. [17]. However, no guarantee can be given regarding the safeness of the results.

Static analysis methods, e.g. [6, 11], use either cycle-level simulators or other models of the hardware to compute the execution time of individual basic blocks. Since, our work is based on the assumption that accurate hardware models are non-existing or difficult to construct these approaches are not sufficient for us. However, if there are such models these approaches are perfectly valid.

7. Conclusion

We have introduced the theoretical foundation for a method deriving the execution time of program fragments. Combined with high-level analysis and calculation tools, this method can derive the worst-case execution time of programs. The method relies only on the target hardware itself for retrieving timing information, making it applicable to a wide range of processors, even those for which timing models are not available.

The main limitation of the approach is that it requires the execution time of basic blocks to be constant. We are currently investigating methods to overcome this limitation. We also plan to validate the approach by applying it to a substantial case-study. This will require development of prototype tools and integration with some high-level analysis and calculation tools.

Acknowledgement

The authors wish to thank Jakob Engblom for his valuable comments on an early version of this paper, which helped improve the quality of it.

References