Efficient Longest Executable Path Search for Programs with Complex Flows and Pipeline Effects

Friedhelm Stappert *
C-LAB
Fürstenallee 11
33102 Paderborn
Germany
fst@c-lab.de

Andreas Ermedahl †
DoCS, Uppsala University
Box 325
SE-751 05 Uppsala
Sweden
ebbe@docs.uu.se

Jakob Engblom ‡
IAR Systems AB
Box 23051
SE-750 23 Uppsala
Sweden
jakob@iar.se

ABSTRACT

Current development tools for embedded real-time systems do not efficiently support the timing aspect. The most important timing parameter for scheduling and system analysis is the Worst-Case Execution Time (WCET) of a program.

This paper presents a fast and effective WCET calculation method that takes account of low-level machine aspects like pipelining and caches, and high-level program flow like loops and infeasible paths. The method is more efficient than previous path-based approaches, and can easily handle complex programs. By separating the low-level from the high-level analysis, the method is easy to retarget.

Experiments confirm that speed does not sacrifice precision, and that programs with extreme numbers of potential execution paths can be analyzed quickly.

Keywords

WCET, hard real-time, embedded systems, path search, program flow, pipeline timing

1. INTRODUCTION

The purpose of Worst-Case Execution Time (WCET) analysis is to provide a priori information about the worst possible execution time of a program before using the program.

∗Friedhelm is a PhD student at C-LAB (www.c-lab.de), which is a cooperation between Paderborn University and Siemens.
†This work is performed within the Advanced Software Technology (ASTEC, http://www.docs.uu.se/astec) competence center, supported by the Swedish National Board for Industrial and Technical Development (NUTEK, http://www.nutek.se).
‡Jakob is an industrial PhD student at IAR Systems (http://www.iar.com) and Uppsala university, sharing his time between research and development work. He is partially supported via ASTEC.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Copyright 2001 ACM 1-58113-399-5/01/0011 ...$5.00.

in a system. Reliable WCET estimates are necessary when designing and verifying embedded real-time systems, especially when used in safety-critical systems like vehicles and industrial plants.

WCET estimates can be used to perform scheduling and schedulability analysis, to determine whether performance goals are met for periodic tasks, to check that interrupts have sufficiently short reaction times, to find performance bottlenecks, and for many other purposes. [1, 9, 12].

WCET estimates must be safe, i.e. guaranteed not to underestimate the execution time, and tight, i.e. provide acceptable overestimations.

Measuring the execution time to find the worst case requires access to the target hardware and is a very time-consuming process. It is also very hard to guarantee to find the worst case with measurements.

Static analysis promises to generate safe and tight estimates by analyzing the source code and object code of the program off-line (without executing it).

When performing static WCET analysis, it is assumed that there are no interfering background activities, such as direct memory access (DMA), and that the program execution is uninterrupted. Extra execution time caused by cache interference between tasks, interrupts, etc. are assumed to be handled in a separate analysis.

To make WCET analysis a mainstream tool for embedded real-time systems development, the analysis should be part of the usual work-flow of edit-compile-test-debug. Just like a program is checked for bugs, it should be checked for timeliness.

For this to be achieved, WCET analysis should be performed inside the compilation system, which demands a very efficient method for calculating the WCET.

Furthermore, due to the fragmented character of the embedded processor market [13], it is necessary that a tool is easily retargeted to new architectures.

In this paper, we present a very fast method for calculating WCET estimates, given information about the program flow and a program timing model (which is given in a target-independent format). The method is path-based in that it explicitly finds the longest path in the control flow of the program, and more efficient than previous path-based approaches (especially in the presence of many potential execution paths). This makes it feasible to use the method inside a compiler, for example.
The program timing model is obtained by using a trace-driven simulator for the target architecture. The separation from the calculation makes it easier to retarget, since the calculation is not affected by target changes.

The concrete contributions of this work are:

- We adapt an acyclic longest-path search algorithm to perform longest executable path search in a program, providing a very efficient path-based WCET calculation algorithm.
- We extend the basic algorithm to handle flow information such as dependent conditional statements and implications, expressed using the flow language presented in [8].
- We extend the algorithm further to handle arbitrary pipeline effects, going beyond pipeline effects between adjacent basic blocks.
- We have implemented the new calculation algorithm within an existing WCET prototype tool, replacing a different calculation module while using the existing hardware model.

The rest of this paper is organized as follows: Section 2 describes previous work in the field of WCET analysis. Section 3 presents our WCET tool framework. Section 4 shows the basic path-based calculation method, while Section 5 shows how to add flow information and Section 6 arbitrary pipeline effects to the calculation. Section 7 contains experimental evaluations, and Section 8 gives conclusions and discusses future work.

2. PREVIOUS WORK AND WCET ANALYSIS OVERVIEW

To generate a WCET estimate, we consider a program to be processed through the phases of program flow analysis, low level analysis and calculation.

The program flow analysis phase determines possible program flows and provides information about which functions get called, how many times loops iterate, if there are dependencies between if-statements, etc. The information can be obtained using manual annotations [11, 18, 25, 28], or automatic flow analysis [3, 10, 14, 21, 30].

The purpose of low-level analysis is to determine the execution time for each atomic unit of flow (e.g., an instruction or a basic block) given the architecture and features of the target system. Low-level analysis can be further divided into global low-level analysis, for effects that require a global view of the complete program, and local low-level analysis, for effects that can be handled locally for an instruction and its neighbors.

In global low-level analysis, instruction caches [11, 14, 19, 30], cache hierarchies [23], data caches [17, 30, 33], and branch predictors [4] have been analyzed. Local low-level analysis has built software models to deal with scalar pipelines [7, 14, 19] and superscalar CPUs [20, 29, 30]. For some complex architectures attempts have been made to use the hardware itself [26].

The purpose of the calculation phase is to calculate the WCET estimate for a program, combining the information derived in the program flow and global and local low-level analysis phases. There are three main categories of calculation methods proposed: path-based, tree-based, and IPET (Implicit Path Enumeration Technique).

In a tree-based approach the WCET is found in a bottom-up traversal of a tree, generally corresponding to a parse tree of the program, using rules defined for each type of compound program statement to determine the execution time of the statement [2, 4, 19, 27]. The method is conceptually simple and computationally quite cheap, but has problems handling flow information, since the computations cannot consider dependencies across statements.

In IPET, program flow and low-level execution time are modeled using arithmetic constraints [8, 11, 18, 24, 28]. Each basic block and program flow edge in the program is given a time variable ($t_{entity}$) and a count variable ($x_{entity}$), and the goal is to maximize the sum $\sum_{i\in entities} x_i \cdot t_i$, subject to constraints reflecting the structure of the program and possible flows. Very complex flows can be expressed, but the computational complexity of solving the resulting problem is potentially very high.

In a path-based approach, the possible execution paths of a program or piece of a program are explored explicitly to find the longest path [14, 15, 30]. In contrast to IPET, the path-based approach explicitly computes the longest executable path in the program.

3. TOOL OVERVIEW

The work presented in this paper is implemented within the framework of our existing WCET tool. In addition to the previous IPET-based calculation [8], we have implemented a path-based calculation module. The pipeline analysis and other components of the system remain unchanged, demonstrating the modular structure of the tool, and in particular the independence of the pipeline analysis and calculation modules.

Figure 1 gives an overview of the WCET analysis system, using a path-based search as described in this paper. The calculation module is shown in detail.
The target chip for the present implementation is the NEC V850E, a typical 32-bit RISC embedded microcontroller architecture [6]. The compiler used is an IAR V850/V850E C/Embedded C++ compiler [32].

Flow analysis is currently performed manually, resulting in a description of the possible program flow in the *scope graph* data structure. The scope graph reflects the structure of the program and the flow, as described in Section 3.1 below.

The *timing graph* data structure represents the low-level view of the program used to build the program timing model. The data structure and the analysis is presented in more detail in Section 3.2.

We do not use cache analysis in the current experiments, since our target hardware does not have a cache, but Figure 1 still shows where such an analysis module fits in.

### 3.1 Scope Graph and Flow Facts

The *scope graph* is a hierarchical representation of the structure of a program. Each scope corresponds to a certain repeating or differentiating execution context in the program, e.g., loops and function calls, and describes the execution of the object code of the program within that context.

Each scope is assumed to iterate, and has a header node. A new iteration starts each time the header node is executed, and a maximal number of iterations must be given for each scope. Scopes are allowed to iterate just once, i.e., not loop. Each scope can carry a set of *flow facts*. The flow facts language allows complex program flows to be represented in a compact and readable manner. In this paper we address a subset of the flow facts presented in [8].

Each flow fact consists of three parts: the defining *scope*, a *context specifier*, and a *constraint expression*. The fact is valid for each *entry* to the defining scope.

The context specifier describes the iterations for which the constraint expression is valid. All iterations of a scope is denoted $\langle\rangle$, while a subrange is given as $\langle\langle\rangle\rangle$.

The constraints are specified as a relation between two arithmetic expressions involving *execution count variables* and constants. An execution count variable, $x_{\text{entity}}$, corresponds to a node or edge in the code of a scope, and represents the number of times the entity is executed in the context given in the fact. Note that for a path-based analysis, constants can only be zero or one.

Figure 2 shows an example of two nested scopes with some attached flow facts. Each scope has an upper bound, to guarantee program termination.

The fact $\text{inner} : \langle \rangle : x_c + x_r \leq 1$ gives that the nodes C and F can never execute on the same iteration of the scope (an infeasible path).

The fact $\text{inner} : \langle \langle 1..8 \rangle \rangle : x_5 \leq x_2$ gives that, during the first eight iterations of an entry to the loop inner, executing node C implies that G is also executed.

The fact $\text{outer} : \langle \langle 1..8 \rangle \rangle : x_2 = 1$ gives that during the first five iterations of outer, the execution has to pass the I node, and can thus not enter inner.

Note that flow facts represent program flows implicitly by constraining the set of possible program flows, in contrast to [15] where feasible paths are represented explicitly. This makes the flow facts usable with calculation techniques which are not path-based [8].

### 3.2 Timing Graph and Pipeline Analysis

The *timing graph* is a flat program flow graph, where the nodes correspond to basic blocks in the code. Each node and edge in the timing graph can be decorated with information about the execution of that piece of code, extracted from some preceding analysis module. Figure 3 shows an example of a timing graph with information about instruction cache behavior (icache hit and icache miss) and memory type accessed (dmem SRAM). Other types of information can be used.

The timing graph is generated for the whole program at once, and the pipeline analysis generates times for all the nodes and edges in the timing graph in one pass. The pipeline analysis is described in more detail in [7]. Pieces of the timing graph are then used in the calculation of the WCET.

Times for nodes correspond to the execution times of nodes in isolation, (e.g. $t_q$ in Figure 4), and times for edges, (e.g. $\delta_{QR}$ in Figure 4), to the pipeline effect when the two successive nodes are executed in sequence (usually an overlap).

Timing effects for sequences of nodes are calculated by first running the individual nodes (plus execution information), in the simulator, then the sequence, and then comparing the execution times. The process is illustrated in Figure 4. The timing effect, $\delta_{QR}$ for the edge QR is $22 - 19 - 11 = 2$. 

![Figure 2: Scopes with Attached Flow Facts](image)

![Figure 3: Timing Graph with Execution Facts](image)

![Figure 4: Timing Effect Calculation](image)
Dijkstra’s (TG):
/** Initialization **/
for each node \( v \) in \( TG \) do
    \( \text{predecessor}[v] := \text{nil} \)
    \( \text{time_sum}[v] := 0 \)
end for
/** Breadth-first-search **/
for each node \( u \) in \( TG \) in breadth-first order do
    for each outgoing edge \( e = (u, v) \) in \( TG \) do
        \( d := \text{time_sum}[u] + \delta_e \)
        /** Is \( u \) on the longest path to \( v \) **/
        if \( \text{time_sum}[v] < d \)
            \( \text{predecessor}[v] := u \)
            \( \text{time_sum}[v] := d \)
        end if
    end for
end for
return \( TG \)

Figure 5: Longest Path Search Algorithm

—4; the time is negative since the execution of the nodes \( Q \) and \( R \) overlap in the CPU pipeline.

There is a potential for timing effects along longer sequences of nodes than just two, usually caused by a node using some CPU resource that is used by a later node in the sequence, but not by the nodes in between.

The advantage of this approach to pipeline analysis is that we only run each basic block through the machine model a few times, that we do not require a special-purpose CPU model, and that the pipeline model and calculation step are kept separate and independent.

4. EFFICIENT PATH SEARCH

Since our pipeline timing model allows us to compose the execution time of a program from smaller pieces, we base our efficient path search on Dijkstra’s algorithm for longest-path search in an acyclic (timing) graph \( TG \) (shown in Figure 5) [5]. The algorithm computes the longest path in \( O(m + n) \) time where \( m \) is the number of edges and \( n \) is the number of nodes, i.e. it is linear in the size of the graph.

Our approach is more efficient than the classic approach of generating all paths, running them through a pipeline model, and then selecting the longest; in this case, the number of paths to explore is up to \( 2^n \), where \( n \) is the number of decisions in the program segment being analyzed\(^1\). The key to the efficiency is the timing model we use.

In order to be able to apply Dijkstra’s algorithm, we must remove all cycles from the timing graph fragment for a scope. We replace all backedges (i.e. edges to the header node of the scope) with edges to a special continuation node \( \perp_c \), and all edges leading out of the scope are redirected to a special exit node \( \perp_x \) (see Figure 6(b)). This is the “Path Search Preprocessing” stage in Figure 1.

After this preprocessing, the algorithm works by breadth-first search. For each node, it computes the predecessor with the greatest total time used from the header node (called \( \text{time_sum} \) in the algorithm). If a node is not reachable from the header node, the \( \text{time_sum} \) is zero. This is the “Longest Path Search” stage in Figure 1.

For each node \( v \), \( \text{predecessor}[v] \) defines the predecessor of node \( v \) on the longest path from the start node to \( v \), and

\(^1\)To keep complexity under control while losing some precision in the pipeline model, it is possible cut a program segment into smaller pieces with a lower number of decisions in each [14].

The advantage of this approach to pipeline analysis is that we do not require a special-purpose CPU

following the predecessor chains allows complete paths to be formed. Figure 6(c) illustrates the result of the algorithm, showing the predecessor and \( \text{time_sum} \) for each node.

When computing the WCET for a looping scope \( S \), the last iteration has to be treated specially, since a different path is usually taken. Therefore, we calculate two longest executable paths in each scope: one to \( \perp_c \) and one to \( \perp_x \). If there is no executable path to \( \perp_c \), \( S \) does not iterate at all, in which case the WCET for the scope is the longest executable path to \( \perp_x \).

If there is a path going to \( \perp_c \) the final WCET for scope \( S \) becomes:
\[
\text{time_sum}(\perp_c) \cdot (\text{loop bound}(S) - 1) + \text{time_sum}(\perp_x).
\]

5. PATH SEARCH WITH FACTS

In this section, we show how flow facts can be used to obtain more precise WCET estimates by removing infeasible paths.

5.1 Ranges and Virtual Scopes

In order to account for flow facts with ranges, we expand the scope graph to a number of virtual scopes. A virtual scope corresponds to a certain range of iterations of a scope. Whenever two consecutive iterations are covered by different sets of facts they should go into different virtual scopes, as illustrated in Figure 7.

The two facts \( s : <1..5> : X_c = 1 \) and \( s : <3..10> : X_b + X_f = 1 \) are specified for the scope \( s \). Since the facts overlap partially, the scope is split into the four virtual scopes shown, each with a set of facts valid for their entire iteration space.
VirtualScopeCreation(S):
VS := ∅, begin := 1
Fcurrent := facts covering iteration begin in S
/** Loop over all iterations in the scope **/
for each iteration iter between 2 and loopbound(S) do
Fiter := facts covering iteration iter in S
/** Has set of covering facts changed **/
if Fcurrent ≠ Fiter then
   end := iter - 1
VS := add virtual scope s : begin..end to VS
begin := iter, Fcurrent := Fiter
end for
return VS

Figure 8: Virtual Scopes Generation

SimpleFactRemoval(TG):
/** Handle 'forbidden' nodes, (xnode = 0): */
for each forbidden node v in TG do
   delete v from TG
   remove resulting dead paths
end for
/** Handle 'must-have' nodes, (xnode = 1): */
for each must-have node v in TG
   mark all transitive predecessors of v
   mark all transitive successors of v
   for each node u in TG
      if u not marked
         delete node u from TG
      end for
end for
return TG

Figure 9: Simple Facts Removal

The algorithm for finding the virtual scopes for a scope S is given in Figure 8. Note that the split of the iteration space of a scope is the inverse of the approach used in [15], where they unify iteration spaces that have any information in common, giving lower precision for facts that partially overlap each other.

5.2 Simple Fact Removal

For efficiency, certain facts that can be expressed by modifying the timing graph are handled in a preprocessing stage (the “Simple Fact Removal” stage in Figure 1).

A fact with a constraint of the form xnode = 0, i.e. node must not be taken, is handled by simply removing node from the graph.

A fact with a constraint expression of the form xnode = 1, i.e. node must be taken, is handled by removing all paths that do not include node. The paths can be found in time linear to the size of the graph by the algorithm shown in Figure 9.

5.3 Path Search with Infeasible Path Removal

After simplifying the graph as described above, the rest of the facts are handled in a search loop.

Figure 10 shows the top-level algorithm. It performs a bottom-up traversal of the scope graph (by recursion), divides scopes to virtual scopes, performs preprocessing on timing graph fragments, and searches for the longest path.

The longest path found for a scope is checked for feasibility against the flow facts not removed in the preprocessing. Fea-

WCETCalculation(S):
/** Check if WCET for S already has been calculated **/
if WCET for S exists in TimeCache then
   return WCET for S from TimeCache
/** If not, we have to calculate WCET **/
WCET := 0
/** Replace call to subscopes with node with time **/
for each subscope sub reachable in S do
   tsub := WCETCalculation(sub)
   replace sub with node taking tsub time
end for
/** Divide scope S into virtual scopes **/
VS := VirtualScopeCreation(S)
/** Calculate times for virtual scopes **/
for each virtual vs in VS in increasing order do
   /** Get and convert timing graph **/
   TG := TimingGraphFragment(S, vs)
   TG := PathSearchPreprocessing(TG)
   TG := SimpleFactRemoval(TG)
   TG := LongTimingEffectExpansion(TG)
   /** Extract longest feasible paths **/
   {tvs, stop} := VirtualScopeTime(TG, vs, S)
   /** Add time for virtual scope to WCET of S **/
   WCET := WCET + tvs
   /** Check if we have an early exit **/
   if stop == true then break
end for
return calculated WCET of S to TimeCache

Figure 10: WCET Algorithm for a Scope

sibility is checked by comparing the number of occurrences of nodes on the longest path with the constraints specified in the facts.

For example, for a fact like “inner : <> : xc + xf ≤ 1”, we check that the path does not contain both node F and node C.

If the path is not feasible, it is removed from the graph and the search begins again, finding the second-longest path. The path is removed using an algorithm by Martins and Santos [22], and the effect is illustrated in Figure 11. The idea is to create a deviation around the path to be removed, by adding some nodes and removing the last part of the original path. In the process, time_sum[v] and predecessor[v] are updated, avoiding the need for another run of Dijkstra’s algorithm.

The process of longest path search and infeasible path detection and removal is repeated until a feasible path is found, which is the longest executable path in the virtual

Figure 11: Infeasible Path Removal
6. HANDLING LONG PIPELINE EFFECTS

Pipeline effects across basic block sequences longer than two must be considered during the path search since they affect the longest path, as exemplified in Figure 15, where the timing effect on the sequence CDE makes the longest path different from the one in Figure 6. Path-based methods have previously required complete paths to be executed to handle such effects [14], while we use graph rewriting to keep the search efficient.

A timing effect over a long sequence should only be accounted for when all the nodes in the sequence have been executed. Since our longest-path-search-algorithm only has a local knowledge of the path (looking at predecessors), we preprocess the graph in such a manner that each long timing effect can be expressed as an extra timing effect on a regular edge. The algorithm is given in Figure 14 and corresponds to the box “Timing Effect Expansion” in Figure 1.

The process is illustrated in Figure 15. The node D is duplicated to make the path for the timing effect CDE distinct, and the timing effect is added to the edge from D* to E. The longest path changes compared to the base version shown in Figure 6.

6.1 Timing Effects Across Scope Boundaries

For programs with long pipeline timing effects there might be effects across loop boundaries, such as the wrap-around timing effect across the back edge illustrated in Figure 16(a), or timing effects between nodes in different scopes when entering or exiting the subscope (border crossing timing effects).

All timing effects of length two are accounted for at the scope where the edges begin, giving us an exact solution for programs without long timing effects.

To handle long effects, we add history nodes to the graph for the scope where the effects end. The history nodes represent the potential paths taken before the beginning of a path search and therefore have a time value of zero.

For example, the sequence GAB in Figure 16, makes us insert the history node “G**”. This changes the longest path...
Program | Description | Properties
--- | --- | ---
compress | Compression using lzw. | Nested loops, goto-loop, function calls.
crc | Cyclic redundancy check computation on 40 bytes of data. | Complex loops, lots of decisions, loop bounds depend on function arguments, function that executes differently the first time it is called.
expint | Series expansion for computing an exponential integral function. | Inner loop that only runs once, structural WCET estimate gives heavy overestimate.
firm | Simple iterative Fibonacci calculation, used to calculate fib(30). | Parameter-dependent function, single-nested loop.
firm | Finite impulse response filter (signal processing algorithms) over a 700 items long sample. | Inner loop with varying number of iterations, loop-iteration dependent decisions.
sinsertsort | Insertion sort on a reversed array of size 10. | Input-data dependent nested loop with worst-case of $n^2/2$ iterations.
jjdcmint | Discrete-cosine transformation on a 8x8 pixel block. | Long calculation sequences (i.e. long basic blocks), single-nested loops.
ledcmult | Read ten values, output half to LCD | Loop with iteration-dependent flow.
natmult | Matrix multiplication of two 20x20 matrices. | Multiple calls to the same function, nested function calls, triple-nested loops.
ns | Search in a multi-dimensional array | Return from the middle of a loop nest, deep loop nesting.
sichneu | Simulate an extended Petri Net | Automatically generated code containing massive amounts of if-statement ($\gg 250$)

Columns including Flow hold WCET estimates resulting from using flow facts. Columns including Pipeline indicate that pipeline effects between nodes have been accounted for. Actual gives the actual WCET of the program, as given by a simulation of the target platform. The numbers in the +% columns give the pessimism of each WCET estimate in percent.

The results for the columns without Pipeline show that the modeling of pipelines is very important for tight WCET analysis. In most cases, the effect of the pipeline is much larger than that of the control flow. The results show that the pipeline analysis is precise.

For two programs (firmcall, natmult), loop bounds are sufficient to get an exact WCET estimate. For jjdcmint, the facts reduce the pessimism somewhat, while compress, expint and ledcmult show dramatic improvements when facts are added (due to the structure of the programs).

The remaining overestimate in firm and insertsort is due to triangular loops that cannot be expressed within the path-based calculation system.

Columns including Flow hold WCET estimates resulting from using flow facts. Columns including Pipeline indicate that pipeline effects between nodes have been accounted for. Actual gives the actual WCET of the program, as given by a simulation of the target platform. The numbers in the +% columns give the pessimism of each WCET estimate in percent.

The results for the columns without Pipeline show that the modeling of pipelines is very important for tight WCET analysis. In most cases, the effect of the pipeline is much larger than that of the control flow. The results show that the pipeline analysis is precise.

For two programs (firmcall, natmult), loop bounds are sufficient to get an exact WCET estimate. For jjdcmint, the facts reduce the pessimism somewhat, while compress, expint and ledcmult show dramatic improvements when facts are added (due to the structure of the programs).

The remaining overestimate in firm and insertsort is due to triangular loops that cannot be expressed within the path-based calculation system.

Figure 19 shows some information about the complexity of the analysis. The Scopes column lists the number of scopes required to model the program, and V.S. the number of virtual scopes after virtual scope expansion. Paths shows the number of possible execution paths in the entire program, and Expl. the number of paths that our search actually explored. The last column shows how Expl. relates to Paths. In every case, our tool explores only a subset of the paths, and the more complex the programs get (many paths compared to the number of virtual scopes), the proportion of paths explored goes down.

In conclusion, our experiments clearly demonstrate the efficiency, precision, and safety of our WCET analysis method.

8. CONCLUSIONS AND FUTURE WORK

In this paper, we have presented an efficient local longest-path search algorithm for worst-case execution time analysis. We have extended the algorithm to handle complex flow facts and arbitrary pipeline effects. Using this approach, we are able to quickly and efficiently calculate the WCET of programs. The calculation method avoids exploring all the
### Figure 18: Execution Time Estimates

<table>
<thead>
<tr>
<th>Program</th>
<th>Basic Cycles</th>
<th>With Flow Cycles</th>
<th>With Pipeline Cycles</th>
<th>Flow &amp; Pipeline Cycles</th>
<th>Actual Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>compress</td>
<td>126242</td>
<td>+1357</td>
<td>10388</td>
<td>+20</td>
<td>92482</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>10062</td>
<td>+17.2</td>
<td>8672</td>
</tr>
<tr>
<td></td>
<td>86077</td>
<td>+693</td>
<td>10062</td>
<td>+17.2</td>
<td>8588</td>
</tr>
<tr>
<td>fibcall</td>
<td>559</td>
<td>+78.6</td>
<td>559</td>
<td>+78.6</td>
<td>313</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>559</td>
<td>+78.6</td>
<td>0</td>
</tr>
<tr>
<td>insertsort</td>
<td>2328</td>
<td>+117</td>
<td>2328</td>
<td>+117</td>
<td>333</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2328</td>
<td>+117</td>
<td>0</td>
</tr>
<tr>
<td>jfdctint</td>
<td>5388</td>
<td>+9.4</td>
<td>5388</td>
<td>+9.4</td>
<td>4912</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>198</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>5388</td>
<td>+9.4</td>
<td>0</td>
</tr>
<tr>
<td>lcdnum</td>
<td>501</td>
<td>+153</td>
<td>341</td>
<td>+72.2</td>
<td>283</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>198</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>341</td>
<td>+72.2</td>
<td>0</td>
</tr>
<tr>
<td>matmult</td>
<td>278859</td>
<td>+24.4</td>
<td>278859</td>
<td>+24.4</td>
<td>221824</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>221824</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>278859</td>
<td>+24.4</td>
<td>0</td>
</tr>
<tr>
<td>ns</td>
<td>22903</td>
<td>+40.1</td>
<td>20633</td>
<td>+48.5</td>
<td>15343</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>+10.9</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>20633</td>
<td>+48.5</td>
<td>13934</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>+10.9</td>
<td>+0.2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>13911</td>
</tr>
<tr>
<td>nsichneu</td>
<td>150841</td>
<td>+196</td>
<td>87193</td>
<td>+70.6</td>
<td>97662</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>+70.6</td>
<td>+91</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>87193</td>
<td>+70.6</td>
<td>51134</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>+70.6</td>
<td>+0.03</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>51116</td>
</tr>
</tbody>
</table>

### Figure 19: Complexity Measures

paths of a program, typically giving it a computational complexity close to linear in the size of the program.

We have implemented the new calculation method within our generic WCET tool framework, demonstrating the flexibility of that framework.

Our experiments show that the new calculation method generates tight and safe WCET estimates, and that flow information can be used effectively to improve the quality of the estimates. The WCET analysis is efficient enough to be integrated in the natural design-flow of real-time software engineers.

For the future, we are considering whether it is possible to create a hybrid approach between the path-based and IPET-based calculation methods, combining the efficiency of path-based approaches with the expressive power of IPET (in particular, extending the sets of flow facts that can be handled exactly).

Considering the availability of WCET tools, we are cooperating with embedded programming-tools vendor IAR Systems [16].

A longer version of this paper is available as a technical report [31].

### 9. REFERENCES


