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 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 a 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.

[*]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.

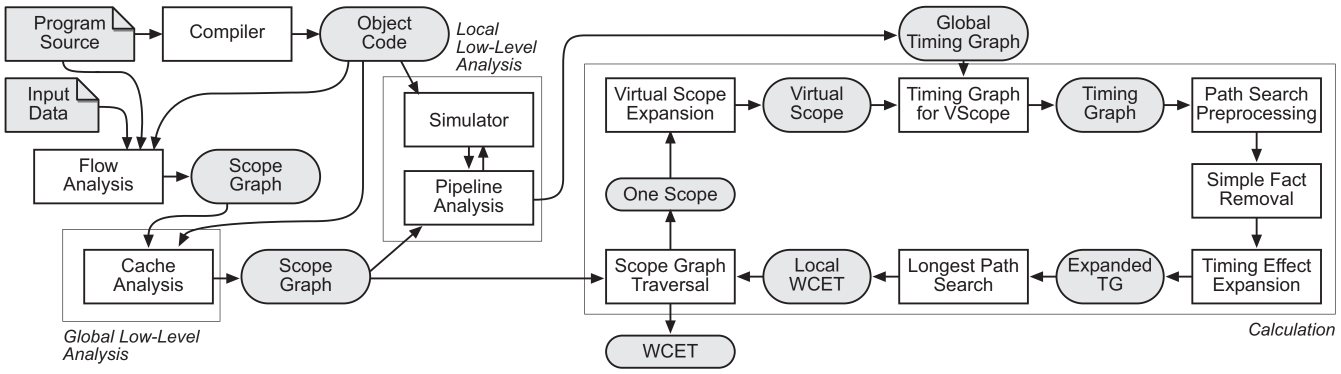


Figure 1: WCET Analysis using Path-Based Calculation

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 implication, 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.

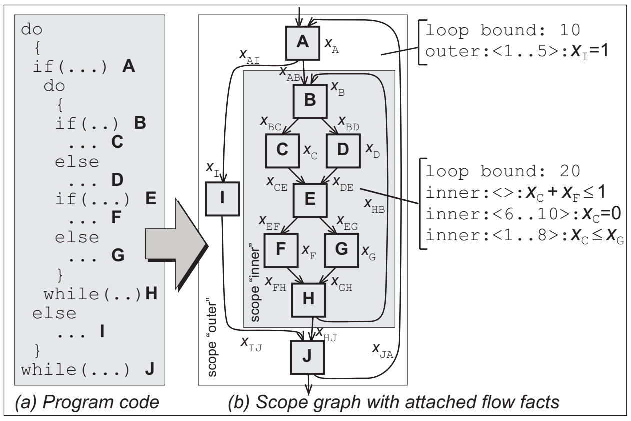
In *IPET*, program flow and low-level execution time are modeled using arithmetic constraints [8, 11, 18, 24, 28]. Each basic block and program flow edge in the program is given a time variable (t_{entity}) and a count variable (x_{entity}), and the goal is to maximize the sum $\sum_{i \in entities} x_i * t_i$, 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



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

```
loop bound: 10
outer:<1..5>:xI=1
```

```
loop bound: 20
inner:<>:xC+xF≤1
inner:<6..10>:xC=0
inner:<1..8>:xC≤xG
```

Figure 2: Scopes with Attached Flow Facts

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.

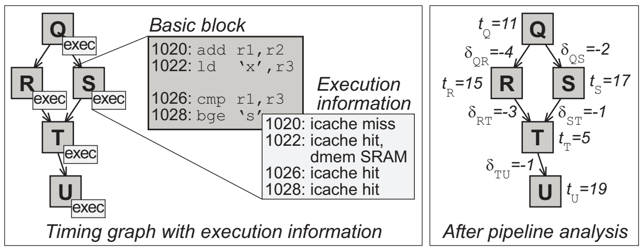
Each scope is assumed to iterate, and has a header node. A new iteration starts each time the header node is executed, and a maximal number of iterations must be given for each scope. Scopes are allowed to iterate just once, i.e. not loop. Each scope can carry a set of *flow facts*. The flow facts language 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 `<>`, while a subrange is given as `<`*min..max*`>`.

The constraints are specified as a relation between two arithmetic expressions involving *execution count variables* and constants. An execution count variable, x_{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



```
1020: add r1,r2
1022: ld  'x',r3

1026: cmp r1,r3
1028: bge 's'
```

1020: icache miss
1022: icache hit,
      dmem SRAM
1026: icache hit
1028: icache hit

Figure 3: Timing Graph with Execution Facts

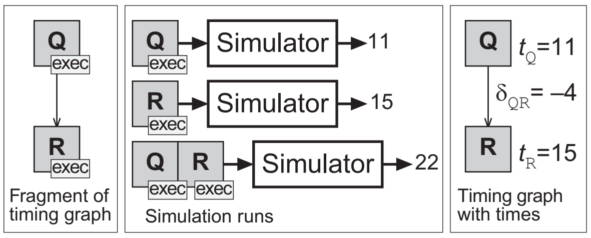


Figure 4: Timing Effect Calculation

attached flow facts. Each scope has an upper bound, to guarantee program termination.

The fact `inner : <> :` $x_{\texttt{C}} + x_{\texttt{F}} \leq 1$ gives that the nodes `C` and `F` can never execute on the same iteration of the scope (an infeasible path).

The fact `inner : <1..8> :` $x_{\texttt{C}} \leq x_{\texttt{G}}$ 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 `outer : <1..5> :` $x_{\texttt{I}} = 1$ 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 by 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. $t_{\texttt{Q}}$ in Figure 4), and times for edges, (e.g. $\delta_{\texttt{QR}}$ 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 compar-

```
$Dijkstra's(TG)$:
  /** Initialization **/
  for each node $v$ in $TG$ do
    $predecessor[v]$ := $nil$
    $time\_sum[v]$ := 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
      $d$ := $time\_sum[u]$ + $t_u$ + $\delta_e$
      /** Is $u$ on the longest path to $v$ **/
      if $time\_sum[v]$ < $d$ then
        $predecessor[v]$ := $u$
        $time\_sum[v]$ := $d$
    end for
  end for
  return $TG$
```

Figure 5: Longest Path Search Algorithm

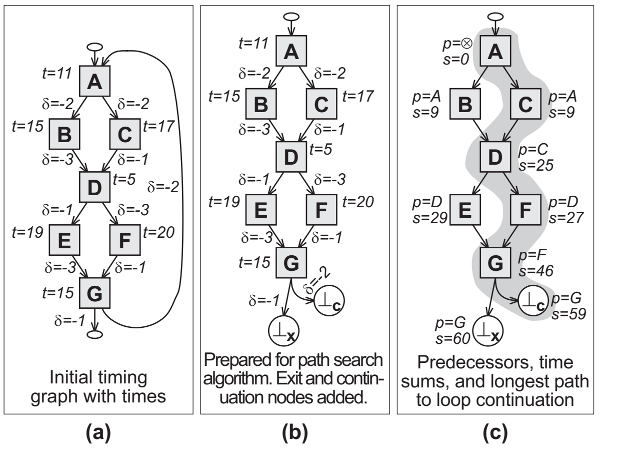

Initial timing graph with times
Prepared for path search algorithm. Exit and continuation nodes added.
Predecessors, time sums, and longest path to loop continuation
(a)
(b)
(c)

Figure 6: Longest Path Search

ing the execution times. The process is illustrated in Figure 4. The timing effect, $\delta_{\texttt{QR}}$, for the edge QR is $22-15-11 = -4$; 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 *time_sum* in the algorithm). If a node is not reachable from the header node, the *time_sum* is zero. This is the "Longest Path Search" stage in Figure 1.

[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].

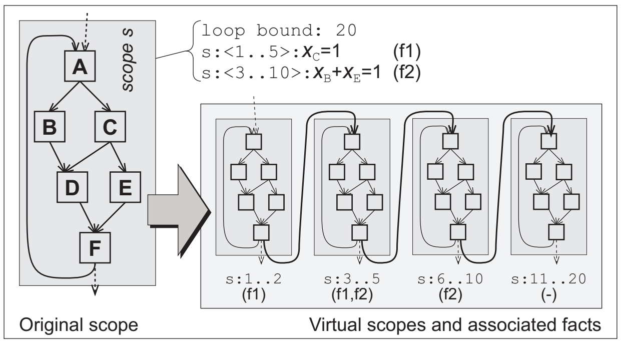


Figure 7: Virtual Scope Expansion

For each node v, *predecessor*$[v]$ defines the predecessor of node v on the longest path from the start node to v, and following the predecessor chains allows complete paths to be formed. Figure 6(c) illustrates the result of the algorithm, showing the predecessor and *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:
$time\_sum(\perp_c) * (loopbound(S) - 1) + time\_sum(\perp_x)$.

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 set of facts they should go into different virtual scopes, as illustrated in Figure 7.

The two facts `s : <1..5>` : $X_{\texttt{C}} = 1$ and `s : <3..10>` : $X_{\texttt{B}} + X_{\texttt{E}} = 1$ are specified for the scope s. Since the facts over-

```
$VirtualScopeCreation(S)$ :
  $VS$ := $\emptyset$, $begin$ := 1
  $F_{current}$ := facts covering iteration $begin$ in $S$
  /** Loop over all iterations in the scope **/
  for each iteration $iter$ between 2 and loopbound($S$) do
    $F_{iter}$ := facts covering iteration $iter$ in $S$
    /** Has set of covering facts changed **/
    if $F_{current} \neq F_{iter}$ then
      $end$ := $iter$ - 1
      $VS$ := add virtual scope $s : begin..end$ to $VS$
      $begin$ := $iter$, $F_{current}$ := $F_{iter}$
  end for
  return $VS$
```

Figure 8: Virtual Scopes Generation

```
$SimpleFactRemoval(TG)$:
  /** Handle 'forbidden' nodes, ($x_{node} = 0$): **/
  for each forbidden node $v$ in $TG$ do
    delete $v$ from $TG$
    remove resulting dead paths
  end for
  /** Handle 'must-have' nodes, ($x_{node} = 1$): **/
  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

lap partially, the scope is split into the four virtual scopes shown, each with a set of facts valid for their entire iteration space.

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 $x_{node} = 0$, 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 $x_{node} = 1$, 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

```
$WCETCalculation(S)$ :
  /** 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
    $t_{sub}$ := $WCETCalculation(sub)$
    replace $sub$ with node taking $t_{sub}$ 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 **/
    {$t_{vs}$, $stop$} := $VirtualScopeTime(TG, vs, S)$
    /** Add time for virtual scope to $WCET$ of $S$ **/
    $WCET$ := $WCET$ + $t_{vs}$
    /** Check if we have an early exit **/
    if $stop$ == $true$ then break
  end for
  add calculated $WCET$ of $S$ to $TimeCache$
  return $WCET$
```

Figure 10: WCET Algorithm for a Scope

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

For example, for a fact like "inner : <> : $x_{\mathtt{C}} + x_{\mathtt{F}} \leq 1$", 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

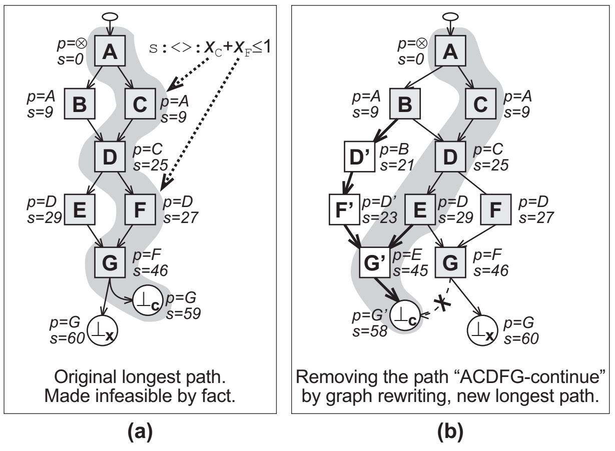


Figure 11: Infeasible Path Removal

```
$LongestFeasiblePathSearch(TG, F, endnode)$ :
  /** Extract longest path $p$ in $TG$ **/
  $TG$ = $Dijkstra's(TG)$
  begin loop
    $t_p$ := $time\_sum(endnode)$
    $p$ := longest path from startnode($TG$) to $endnode$
    /** Is $p$ feasible against flow facts or
    was there no path to $endnode$? **/
    if $IsFeasible(p, F)$ == $true$ or $t_p$ == 0 then
      return execution time $t_p$
    else
      /** Remove $p$ from $TG$
      and extract the next longest path **/
      $TG$ := $DeletePathFromGraph(TG, p)$
  end loop
```

Figure 12: Longest Feasible Path Search

```
$VirtualScopeTime(TG, vs, S)$ :
  $F$ := facts covered by $vs$
  /* Time for longest paths to $\perp_c$ and $\perp_x$ */
  $t_{cont}$ := $LongestFeasiblePathSearch(TG, F, \perp_c)$
  $t_{exit}$ := $LongestFeasiblePathSearch(TG, F, \perp_x)$
  /* Continuation path feasible? */
  if $t_{cont}$ > 0 then
    /* Not the last virtual scope */
    if lastiter($vs$) $\neq$ loopbound($S$) then
      return {$t_{cont}$ * sizeof($vs$),$false$}
    /* Last virtual scope: must exit */
    else if $t_{exit}$ > 0 then
      return {$t_{cont}$ * (sizeof($vs$)-1) + $t_{exit}$,$true$}
  /* Only exit path feasible */
  else if $t_{exit}$ > 0 then
    return {$t_{exit}$,$true$}
```

Figure 13: WCET Calculation for Virtual Scope

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 scope. The algorithm for longest feasible path search, given a timing graph TG, a factset F and a target node *endnode*, is given in Figure 12.

After the initial run of Dijkstra's Algorithm, the path search algorithm runs in $O(K * m)$ time, where m is the number of edges in the graph (the path removal runs in $O(m)$ time), and K is the number of paths removed. In an extreme case, when the only feasible path in the program is the shortest one, we could have to examine all paths (i.e. exponential size). However, for real programs this is very unlikely. It was not a noticeable problem for any of our benchmark programs (see Section 7). Also note that it is possible to interrupt the search at any time, still yielding a safe (but probably pessimistic result).

The algorithm given in Figure 13 returns the WCET for a given virtual scope vs, and a flag indicating if the execution was forced to go to the exit path.

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 examplified 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

```
$LongTimingEffectExpansion(TG)$:
  /** Breadth-first-search **/
  for each node $v$ in $TG$ in breadth-first order do
    if $in\_degree[v]$ > 1 and $v$ in long timing effect then
      for each incoming edge $(u, v)$ inside a sequence do
        /** Copy $v$ and add and redirect edges **/
        add node $v'$ to $TG$
        add edge $(u, v')$ to $TG$
        remove edge $(u, v)$ from $TG$
        for each outgoing edge $e$ = $(v, w)$ in $TG$ do
          add edge $e' = (v', w)$ to $TG$
          /** Add long timing effect to edge **/
          if $e$ is last in a timing sequence $s$ then
            add $\delta_s$ to weight of $e'$
        end for
      end for
end for
```

Figure 14: Long Timing Effects Expansion

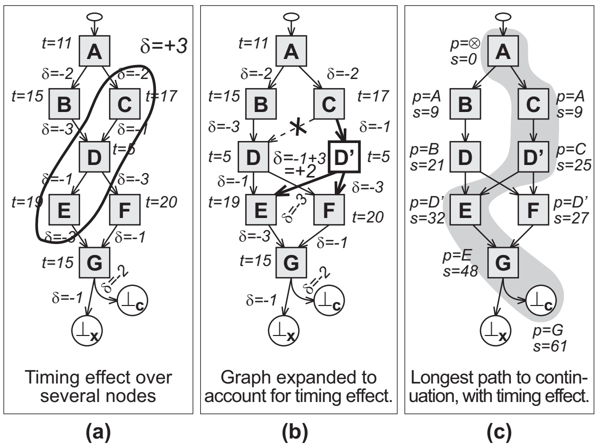


Figure 15: Path Search with Timing Effects

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

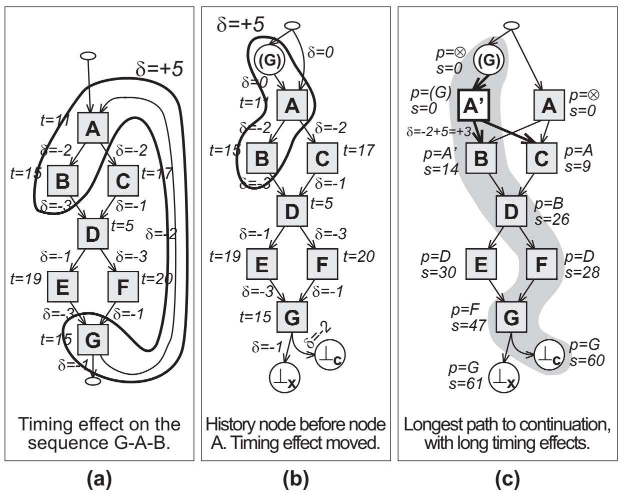
For programs with long pipeline timing effects there might be effects across loop boundaries, such as the *wrap-around timing effect* across the back edge illustrated in Figure 16(a), or timing effects between nodes in different scopes when entering or exiting the subscope (*border crossing timing effects*).

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 rep-

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.
`fibcall`	Simple iterative Fibonacci calculation, used to calculate fib(30).	Parameter-dependent function, single-nested loop.
`fir`	Finite impulse response filter (signal processing algorithms) over a 700 items long sample.	Inner loop with varying number of iterations, loop-iteration dependent decisions.
`insertsort`	Insertion sort on a reversed array of size 10.	Input-data dependent nested loop with worst-case of $n^2/2$ iterations.
`jfdctint`	Discrete-cosine transformation on a 8x8 pixel block.	Long calculation sequences (i.e. long basic blocks), single-nested loops.
`lcdnum`	Read ten values, output half to LCD	Loop with iteration-dependent flow.
`matmult`	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.
`nsichneu`	Simulate an extended Petri Net	Automatically generated code containing massive amounts of `if`-statements ($\gg 250$)

Figure 17: Benchmark Programs



Timing effect on the sequence G-A-B.
History node before node A. Timing effect moved.
Longest path to continuation, with long timing effects.
(a)
(b)
(c)

Figure 16: Timing Effect Across Back-Edge

resent 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 from `ACDFG` as shown in Figure 6 to `ABDFG` as shown in Figure 16(c).

The insertion of history nodes gives a safe but possibly pessimistic estimate of the execution times, since we will always use the worst incoming timing effect. This remaining pessimism is the price we have to pay for the convenience and efficiency of extracting WCET times for scopes in isolation.

7. EVALUATION

In order to demonstrate the effectiveness of our flow specification language and path-based WCET extraction method, we performed a number of experiments, using the programs listed in Figure 17.

The results of the execution time analysis are shown in Figure 18. The column *Basic* gives the WCET estimate using only loop-bounds and ignoring pipeline overlap *between* nodes (but including the pipeline overlap *within* nodes[2]).

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 (`fibcall`, `matmult`), loop bounds are sufficient to get an exact WCET estimate. For `jfdctint`, the facts reduce the pessimism somewhat, while `compress`, `expint` and `lcdnum` show dramatic improvements when facts are added (due to the structure of the programs).

The remaining overestimate in `fir` 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 analy-

[2]Completely ignoring pipeline effects within a block would create a WCET about five times higher (since our chip has a five-stage pipeline).

	Basic		With Flow		With Pipeline		Flow & Pipeline		Actual
Program	Cycles	+%	Cycles	+%	Cycles	+%	Cycles	+%	Cycles
`compress`	126242	+1357	10388	+20	92482	+967	8672	+0.12	8662
`crc`	61624	+104	61624	+104	30389	+0.39	30389	+0.39	30271
`expint`	68077	+693	10062	+17.2	41359	+382	8588	0	8588
`fibcall`	559	+78.6	559	+78.6	313	0	313	0	313
`fir`	487970	+40.2	487808	+40.1	352162	+1.2	352073	+1.1	348095
`insertsort`	2328	+117	2328	+117	1794	+67.0	1794	+67.0	1249
`jfdctint`	5388	+9.4	5388	+9.4	4942	+0.35	4942	+0.35	4925
`lcdnum`	501	+153	341	+72.2	283	+42.9	198	0	198
`matmult`	278859	+24.4	278859	+24.4	221824	0	221824	0	221824
`ns`	22903	+64.6	20653	+48.5	15434	+10.9	13934	+0.2	13911
`nsichneu`	150841	+195	87193	+70.6	97662	+91	51133	+0.03	51116

Figure 18: Execution Time Estimates

Program	Scopes	V.S.	Paths	Expl.	+/−
`compress`	23	27	244	39	-84%
`crc`	8	8	33	12	-64%
`expint`	6	8	25	13	-48%
`fibcall`	3	3	6	4	-33%
`fir`	4	8	34	15	-56%
`insertsort`	3	3	6	5	-17%
`jfdctint`	5	5	10	8	-20%
`lcdnum`	3	5	30	7	-77%
`matmult`	15	15	25	22	-12%
`ns`	6	7	16	11	-31%
`nsichneu`	2	2	3.73E97	3	≈ -100%

Figure 19: Complexity Measures

sis. 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 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 co-operating with embedded programming-tools vendor IAR Systems [16].

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

9. REFERENCES

[1] L. Casparsson, A. Rajnak, K. Tindell, and P. Malmberg. Volcano – A Revolution in On-Board Communications. *Volvo Technology Report*, 1:9–19, 1998.

[2] R. Chapman. Program Timing Analysis. Dependable Computing System Centre, University of York, England, May 1994.

[3] R. Chapman, A. Burns, and A. Wellings. Integrated Program Proof and Worst-Case Timing Analysis of SPARK Ada. In *Proc. ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems (LCT-RTS'94)*, 1994.

[4] A. Colin and I. Puaut. Worst Case Execution Time Analysis for a Processor with Branch Prediction. *Journal of Real-Time Systems*, May 2000.

[5] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* MIT Press, 1990.

[6] NEC Corporation. *V850E/MS1 32/16-bit Single Chip Microcontroller: Architecture*, 3^{rd} edition, January 1999. Document no. U12197EJ3V0UM00.

[7] J. Engblom and A. Ermedahl. Pipeline Timing Analysis Using a Trace-Driven Simulator. In *Proc. 6^{th} International Conference on Real-Time Computing Systems and Applications (RTCSA'99)*. IEEE Computer Society Press, December 1999.

[8] J. Engblom and A. Ermedahl. Modeling Complex Flows for Worst-Case Execution Time Analysis. In *Proc. 21^{th} IEEE Real-Time Systems Symposium (RTSS'00)*, November 2000.

[9] J. Engblom, A. Ermedahl, M. Sjödin, J. Gustafsson, and H. Hansson. Worst-Case Execution-Time Analysis for Embedded Real-Time Systems. *Software Tools for Technology Transfer*, 2001. Accepted for publication.

[10] A. Ermedahl and J. Gustafsson. Deriving Annotations for Tight Calculation of Execution Time. In *Proc. Euro-Par'97 Parallel Processing, LNCS 1300*, pages 1298–1307. Springer Verlag, August 1997.

[11] C. Ferdinand, F. Martin, and R. Wilhelm. Applying Compiler Techniques to Cache Behavior Prediction. In *Proc. ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems (LCT-RTS'97)*, 1997.

[12] J. Ganssle. Really Real-Time Systems. In *Proceedings of the Embedded Systems Conference San Fransisco (ESC SF) 2001*, April 2001.

[13] T. R. Halfhill. Embedded Market Breaks New Ground. *Microprocessor Report, January 17*, 2000.

[14] C. Healy, R. Arnold, F. Müller, D. Whalley, and M. Harmon. Bounding Pipeline and Instruction Cache Performance. *IEEE Transactions on Computers*, 48(1), January 1999.

[15] C. Healy and D. Whalley. Tighter Timing Predictions by Automatic Detection and Exploitation of Value-Dependent Constraints. In *Proc. 5^{th} IEEE*

Real-Time Technology and Applications Symposium (RTAS'99), pages 79–88, June 1999.

[16] IAR Systems WWW homepage. URL: `http://www.iar.com`.

[17] S.-K. Kim, S. L. Min, and R. Ha. Efficient Worst Case Timing Analysis of Data Caching. In *Proc. of RTAS'96*, pages 230–240. IEEE, 1996.

[18] Y-T. S. Li and S. Malik. Performance Analysis of Embedded Software Using Implicit Path Enumeration. In *Proc. of the 32:nd Design Automation Conference*, pages 456–461, 1995.

[19] S.-S. Lim, Y. H. Bae, C. T. Jang, B.-D. Rhee, S. L. Min, C. Y. Park, H. Shin, K. Park, and C. S. Ki. An Accurate Worst-Case Timing Analysis for RISC Processors. *IEEE Transactions on Software Engineering*, 21(7):593–604, July 1995.

[20] S.-S. Lim, J. H. Han, J. Kim, and S. L. Min. A Worst Case Timing Analysis Technique for Multiple-Issue Machines. In *Proc. 19th IEEE Real-Time Systems Symposium (RTSS'98)*, December 1998.

[21] T. Lundqvist and P. Stenström. Integrating Path and Timing Analysis using Instruction-Level Simulation Techniques. In *Proc. SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES'98)*, June 1998.

[22] E. Martins and J. Santos. A New Shortest Paths Ranking Algorithm. *Investigacao Operational*, 20(1):47–62, 2000.

[23] F. Müller. Timing Predictions for Multi-Level Caches. In *Proc. ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems (LCT-RTS'97)*, pages 29–36, Jun 1997.

[24] G. Ottosson and M. Sjödin. Worst-Case Execution Time Analysis for Modern Hardware Architectures. In *Proc. ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems (LCT-RTS'97)*, June 1997.

[25] Chang Yun Park. Predicting Program Execution Times by Analyzing Static and Dynamic Program Paths. *Real-Time Systems*, 5(1):31–62, March 1993.

[26] S. Petters and G. Färber. Making Worst-Case Execution Time Analysis for Hard Real-Time Tasks on State of the Art Processors Feasible. In *Proc. 6th International Conference on Real-Time Computing Systems and Applications (RTCSA'99)*, December 1999.

[27] P. Puschner and C. Koza. Calculating the Maximum Execution Time of Real-Time Programs. *The Journal of Real-Time Systems*, 1(1):159–176, 1989.

[28] P. Puschner and A. Schedl. Computing Maximum Task Execution Times with Linear Programming Techniques. Technical report, Technische Universität, Institut für Technische Informatik, Wien, April 1995.

[29] J. Schneider and C. Ferdinand. Pipeline Behaviour Prediction for Superscalar Processors by Abstract Interpretation. In *Proc. SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES'99)*. ACM Press, May 1999.

[30] F. Stappert and P. Altenbernd. Complete Worst-Case Execution Time Analysis of Straight-line Hard Real-Time Programs. *Journal of Systems Architecture*, 46(4):339–355, 2000.

[31] F. Stappert, A. Ermedahl, and J. Engblom. Efficient Longest Executable Path Search for Programs with Complex Flows and Pipeline Effects. Technical Report 2001-012, Dept. of Information Technology, Uppsala University, 2001.

[32] IAR Systems. *V850 C/EC++ Compiler Programming Guide*, 1st edition, January 1999.

[33] R. White, F. Müller, C. Healy, D. Whalley, and M. Harmon. Timing Analysis for Data Caches and Set-Associative Caches. In *Proc. 3rd IEEE Real-Time Technology and Applications Symposium (RTAS'97)*, pages 192–202, June 1997.