# Efficient Longest Executable Path Search for Programs with Complex Flows and Pipeline Effects [◇]

Friedhelm Stappert[*]
C-LAB
Fürstenallee 11, 33102 Paderborn
Germany
`friedhelm.stappert@c-lab.de`

Andreas Ermedahl[†]
DoCS, Uppsala University
Box 325, SE-751 05 Uppsala
Sweden
`andreas.ermedahl@docs.uu.se`

Jakob Engblom[†‡]
IAR Systems AB
Box 23051, SE-750 23 Uppsala
Sweden
`jakob.engblom@iar.se`

## Abstract

*Current development tools for embedded systems do not efficiently support the timing aspect of embedded real-time systems. 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 pipeline analysis from the calculation, 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

[◇] Uppsala University, Department of Information Technology Technical Report 2001-012, second release, August 2001 (`http://www.it.uu.se/research/reports/`). ISSN 1404-3203.

[*] Friedhelm is a PhD student at C-LAB (`www.c-lab.de`), which is a cooperation of 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.

## 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 real-time systems are used to control 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 many other purposes [1, 9, 12].

To be valid for use in safety-critical systems, WCET estimates must be *safe*, i.e. guaranteed not to underestimate the execution time. To be useful for all kinds of systems, they must be *tight*, i.e. provide acceptable overestimations. The safeness of an estimate is critical when the estimate is used in the construction of a safety-critical system.

WCET estimates can be generated by measurement, by hand, or by static analysis. Measuring the execution time requires access to the target hardware and measuring tools, and is a time-consuming process. Also, the timing estimates obtained by measurement are not guaranteed to be safe, since it is in general very hard to find the input that causes the longest execution time for a certain program.

It is possible to count execution cycles by hand, given a CPU manual and an assembly listing, but for anything but the simplest 8-bit architectures, this is a daunting task due to the more complex execution mechanisms associated with pipelines, caches, and other features found on today's high-end embedded chips.

*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) or refresh of DRAM, and that the program execution is uninterrupted (no task switches or interrupts). Extra execution time caused by cache interference between tasks, interrupts, etc. are deferred to some subsequent analysis or system design phase.

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.

Also, the hardware model used in the WCET tool should be a separate module that is easy to change when changing target processor. Due to the fragmented character of the embedded processor market [13], it is not possible to get away with a WCET tool supporting only a single architecture.

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 the same format regardless of the target architecture). The method is *path-based* in that it explicitly finds the longest path in the program, but 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 approach only runs each basic block in the program through the machine model a few times, and does not require a special-purpose CPU model.

A big advantage of generating the timing model for the program in a separate step is that the hardware model can be replaced independently of the rest of the tool, making a tool much easier to retarget, and that several different calculation methods can be used with the same program timing model, making it possible to trade speed and precision.

The concrete contributions of this work are:

- We adapt an acyclic longest-path search algorithm to perform longest executable path search in a program, which gives us 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.* Most WCET research groups make a similar division notationally, but integrate two or more of the phases into a single algorithm, making parts harder to reuse.

The program flow analysis phase determines possible program flows, without regard to the time for each "atomic" unit of flow. The result of the flow analysis should provide 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* (integrated in the programming language [24], or provided separately [11, 17, 27]), or *automatic flow analysis* [3, 10, 14, 20, 29].

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, 18, 29], cache hierarchies [22], data caches [16, 29, 31], and branch predictors [4] have been analyzed. Local low-level analysis has built software models to deal with scalar pipelines [7, 14, 18] and superscalar CPUs [19, 28, 29]. For some complex architectures attempts have been made to use the hardware itself [25].

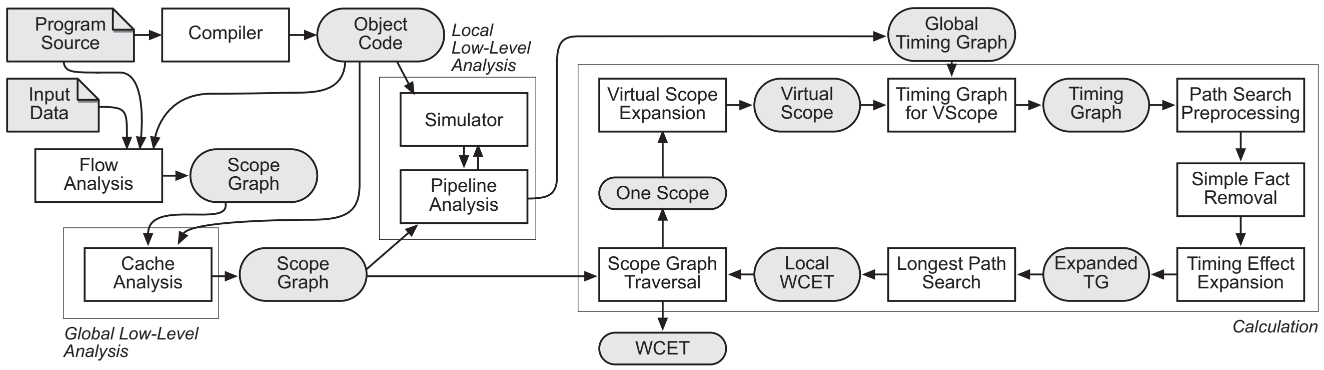


**Figure 1. WCET Analysis using Path-Based Calculation**

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 in literature: *path-based*, *tree-based*, and *IPET* (Implicit Path Enumeration Technique).

In a *tree-based approach* the final WCET is generated by 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, 18, 26]. The method is conceptually simple and computationally quite cheap, but has problems handling flow information, since the computations are local within a single program statement and thus cannot consider dependencies across statements.

In *IPET*, program flow and low-level execution time are modeled using arithmetic constraints [8, 11, 17, 23, 27]. 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. The result is a worst-case count for each node and edge. Very complex flows can be expressed using constraints, but the computational complexity of solving the resulting problem is potentially very high (there have been no conclusive results in the literature regarding the actual complexity for real programs).

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, 29]. In contrast to IPET, the path-based approach explicitly computes the longest executable path in the program. This may be valuable information for the programmer, e.g. for tuning and debugging purposes.

## 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 module [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.

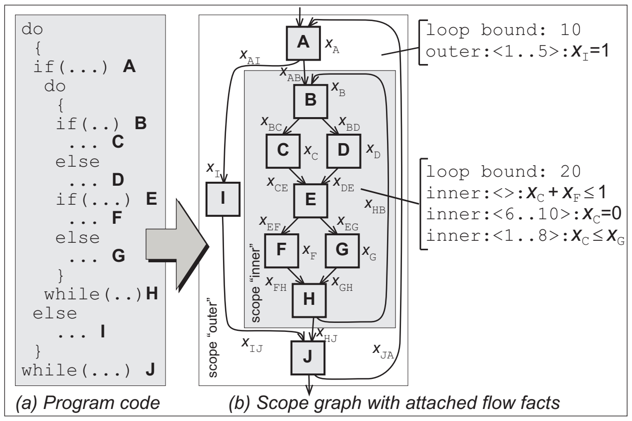
Figure 1 gives an overview of the WCET analysis system, when using a path-based search as described in this paper. The calculation module is shown in detail.

The target chip for the present implementation is the NEC V850E, a typical 32-bit RISC embedded microcontroller architecture [6]. The compiler is a modified IAR V850/V850E C/Embedded C++ compiler [30] which emits the object code of the program in an accessible format.

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 an explicit low-level view of the program and is generated by the local low-level pipeline analysis. The data structure and the analysis is presented in more detail in Section 3.2.

We have implemented an instruction cache analysis similar to the one described by Ferdinand et al. [11], but we do not use the analysis in the current experiments, since our target hardware does not have a cache. Figure 1 still shows that such an analysis would fit in, by modifying the scope graph to include cache information as described in [7, 9]. Cache- and other global low-level analysis results are used in the pipeline analysis as described in Section 3.2 below.



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

**Figure 2. Scopes with Attached Flow Facts**

## 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 maximal number of iterations must be given for each scope, and a new iteration is defined to start each time the header node is passed. Scopes are allowed to iterate only once, i.e. not loop.

Each scope can carry a set of *flow facts*. The flow facts language defined by our group is a powerful language that 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].

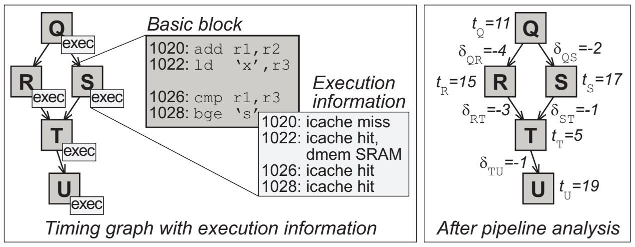
Each flow fact consists of three parts: the name of the *scope* where the fact is defined, a *context specifier*, and a *constraint expression*.

The fact is valid for *each entry* to the scope where it is attached. If the same scope is entered several times, each entry starts a new iteration count from zero.

The context specifier describes the iterations for which the constraint expression is valid. Facts valid for certain iterations are expressed as <*min*..*max*>, where *min* and *max* are integers and $min \leq max$, while facts valid for all iterations of a scope are denoted by `<>`.

These flow facts have a natural relation to path-based calculations, since they talk about what happens on a single iteration of a scope. The facts are used to remove certain paths from the set of possible paths for a scope.

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 an entity in the scope graph



```
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**

(node or edge) and represents the number of times the entity is executed in the context given in the fact. A fact is only allowed to refer to variables located in the scope to where the fact is attached.

Figure 2 shows an example of two nested scopes with some attached flow facts. Note that each scope has an upper loop bound attached to it, used for guaranteeing that the analyzed program terminates.

The fact `inner : <> :` $x_{\mathtt{C}} + x_{\mathtt{F}} \leq 1$ gives that the nodes `C` and `F` can never execute on the same iteration of the scope (an infeasible path), while the fact `inner : <6..10> :` $x_{\mathtt{C}} = 0$ gives that for each entry of the inner loop, node `C` can not be executed during iteration 6 to 10.

The fact `inner : <1..8> :` $x_{\mathtt{C}} \leq x_{\mathtt{G}}$ gives that, for each entry of `inner`, during the first eight iterations, an execution of `C` implies that `G` must also be executed.

The fact `outer : <1..5> :` $x_{\mathtt{I}} = 1$ gives that for each entry of `outer`, during the first five iterations of `outer`, the execution is forced to take the path passing the `I` node, (and can therefore not enter the inner scope during those iterations).

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 or edge in the timing graph can be decorated with information about the execution of that piece of code, (e.g. cache behavior or memory accesses), extracted by some preceding analysis module. The timing graph is generated for the whole program at once, and the pipeline analysis is carried out once for the whole graph. Pieces of the timing graph are then used in the calculation of the WCET.

Figure 3 shows an example of a timing graph: the execution information indicates whether instructions hit or miss the instruction cache (`icache hit` and `icache miss`) and the memory type accessed by a load instruction (`dmem SRAM`). Many other types of facts could be

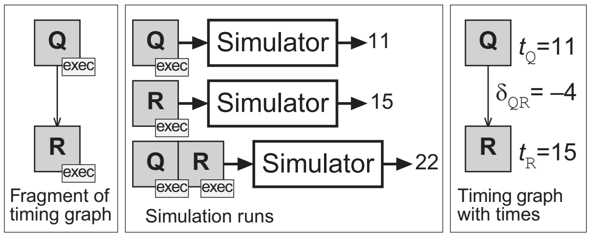


**Figure 4. Timing Effect Calculation**

imagined depending on the properties of the target architecture.

The pipeline analysis generates times for the nodes and edges in the timing graph. 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) [7].

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, $\delta_{\texttt{QR}}$, 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. Such timing effects should only be included in the final WCET estimate if all the nodes in the sequence are executed in sequence. For example, in Figure 18(a) on page 9, we have a timing effect for the sequence `CDE`. This means that if and only if nodes `C`, `D`, and `E` are executed in sequence, 3 clock cycles should be added to the execution time (note that timing effects can be negative as well as positive).

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

The classic approach to longest executable path search in path-based calculation is to generate all possible paths for a certain program segment (function, loop body, or other unit), run all the paths through some kind of hardware model, and select the path with the longest execution time. The unit of analysis is the complete path, and the number of paths to explore is up to $2^n$, where $n$ is the number of decisions in the

```
$Dijkstra's(TG)$:
  /** 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**

program segment being analyzed. The need to handle many complete paths arises from the use of pipelining in modern processors: to get a tight timing estimate, one must account for the overlap between basic blocks, and this can only be done by analyzing all the basic blocks in a path in a continuous sequence.[1]

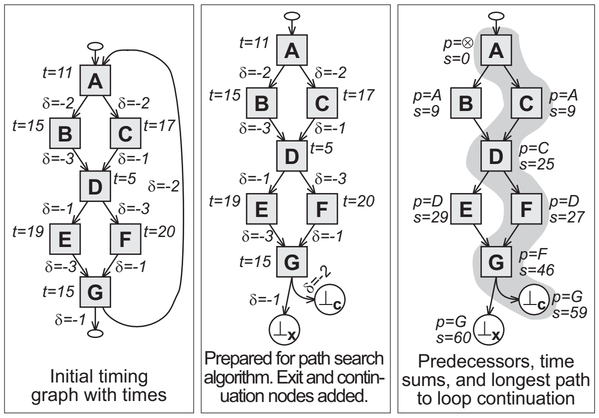
However, since our pipeline analysis allows the timing of a path to be composed from smaller components, it is possible to reformulate the longest path search problem as finding the longest path in a directed acyclic graph, eliminating the need for an explicit enumeration of all paths to handle pipeline effects.

We base our efficient path search on Dijkstra's algorithm (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. In order to be able to apply the algorithm on the timing graph $TG$ for a scope $S$, we must first remove all cycles from the graph, since in general the longest path in a graph is undefined if there are cycles in the graph. Therefore, within each scope, we replace all backedges (i.e. edges back to the header node of the scope) with edges to a special *continuation node* $\perp_c$. Furthermore, all edges leading to nodes outside 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.

Considering pipeline effects across scope boundaries, all timing effects of length two are accounted for at the scope where the edges begin. This solution is exact for programs without long timing effects, since all timing effects will be accounted for.

After this preprocessing, the algorithm works by breadth-first search. For each node, it computes the

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


t=11
A
δ=-2
δ=-2
t=15
B
C
t=17
δ=-3
δ=-1
D
t=5
δ=-2
δ=-1
δ=-3
t=19
E
F
t=20
δ=-3
δ=-1
t=15
G
δ=-1
Initial timing graph with times
⊥c
⊥x
Prepared for path search algorithm. Exit and continuation nodes added.
p=⊗ s=0
p=A s=9
p=C s=25
p=D s=29
p=D s=27
p=F s=46
p=G s=59
p=G s=60
Predecessors, time sums, and longest path to loop continuation

(a) (b) (c)

**Figure 6. Longest Path Search**

predecessor with the greatest total time on the longest path from the start node of the graph (called *time_sum* in the algorithm). If a node is not reachable from the header node, due to the removal of certain paths (Section 5), the corresponding *time_sum* is zero. Figure 6(c) illustrates the result of the algorithm, showing the predecessor and *time_sum* for each node. This is the “Longest Path Search” stage in Figure 1.

After each run, for each node $v$, *predecessor*$[v]$ defines the predecessor of node $v$ on the longest path from the start node to $v$. Thus, a path can easily be constructed backwards by following the *predecessor* chain.

When computing the local WCET for a looping scope $S$, we have to treat the last iteration specially, since when exiting a scope, a different path is usually taken, which may be longer or shorter than the repeating path. Therefore, we calculate *two* longest executable paths in each scope: one to the special node $\perp_c$ and one to the special node $\perp_x$. 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 remove infeasible paths in the path-based calculation, and thus obtain more precise WCET estimates.

### 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, and the virtual scope expansion will, for each virtual scope, create a copy of the original scope.

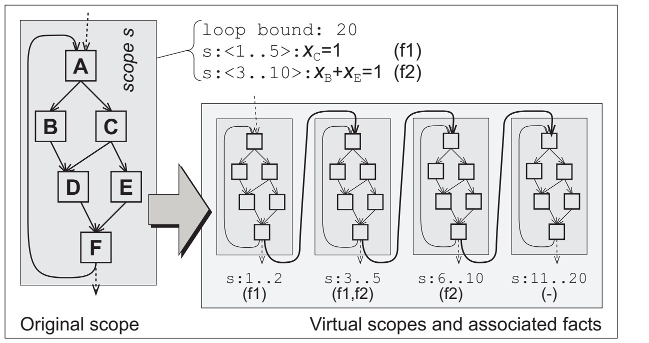


**Figure 7. Virtual Scope Expansion**

```
$VirtualScopeCreation(S)$ :
  $VS$ := $\emptyset$, $begin$ := 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**

The purpose of the virtual scope expansion is to make sure that each fact attached to a virtual scope has a range covering the *entire* iteration range of a virtual scope, as illustrated in Figure 7. Here, the two facts s : <1..5> : $X_{\mathtt{C}} = 1$ and s : <3..10> : $X_{\mathtt{B}} + X_{\mathtt{E}} = 1$ are specified for the scope s. Both facts hold for the iterations 3..5. Only fact $f_1$ holds in iterations 1..2, and $f_2$ in iterations 6..10. In iterations 11..20, none of the facts hold. Thus, the scope is split into the virtual scopes s : 1..2, s : 3..5, s : 6..10, and s : 11..20. After the expansion, we note which facts are valid for each virtual scope.

This split of the iteration space of a scope is the inverse of the approach used by Whalley and Healy in [15]. They form the union of all iteration spaces that have any information in common, giving lower precision for facts that partially overlap each other.

The algorithm for finding the virtual scopes corresponding to a scope is given in Figure 8.

### 5.2. Simple Fact Removal

To make the path search with facts more efficient, certain facts can be handled in a *preprocessing* stage (the “Simple Fact Removal” stage in Figure 1).

A fact with a constraint expression of the form $x_{node} = 0$, stating that *node* must not be taken in the

```
$SimpleFactRemoval(TG)$:
  /** 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**

covered iterations, can be handled by simply removing *node* from the corresponding virtual scope.

A fact with a constraint expression of the form $x_{node} = 1$, stating that *node* must be taken on each iteration, can be handled by removing all paths from the graph that do not include *node*. The paths can be found in time linear to the size of the graph by first marking all transitive predecessors and successors of *node*, and then removing all nodes that are not marked. Any such node would not be on a path from start to end involving *node*, and is thus dead.

The algorithm for this step is shown in Figure 9.

## 5.3. Path Search with Infeasible Path Removal

After applying the modifications to the graph as described above, we start searching for the longest executable path allowed by the remaining facts. We have the fragment of the timing graph corresponding to a virtual scope, with backedges removed and special nodes added, and a set of remaining facts.

Figure 10 shows the top-level algorithm, used for each scope. It performs WCET analysis recursively for the subscopes, divides the scope into virtual scopes, retrieves a piece of the timing graph, and removes the paths corresponding to the simple facts.

The longest path in the graph is found as described in Section 4 above. The longest path is then checked for feasibility against the flow facts not removed in the preprocessing. The checking is done by counting the number of times each node occurs in the path, and comparing this to the constraints specified in the facts. For example, for a fact like "inner : <> : $x_{\mathtt{C}} + x_{\mathtt{F}} \leq 1$", we will check that the path does not contain both node F and node C. Note that for a path-based analysis, each variable can only be zero or one, since the paths do not loop.

```
$WCETCalculation(S)$ :
  /** 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**

If the path is not feasible, it is removed from the graph and the search begins again, now finding the second-longest path. The path is removed using an algorithm by Martins and Santos [21], and the effect is illustrated in Figure 11. The idea is to create a deviation around the path to be removed. This is achieved by adding some new nodes and edges to the graph as shown in Figure 11, and removing the end of the original path. Note that the modified graph still contains all paths of the original graph, except precisely the removed one. All new nodes and edges have the same timing as their originals in the timing graph. During the path removal, the algorithm also computes the next longest path in the graph by updating the path information *time_sum*$[v]$ and *predecessor*$[v]$ (see Section 4 above) for all affected nodes $v$ (avoiding another pass of Dijkstra's algorithm).

The process of longest path search and infeasible path detection and removal is repeated until a feasible path is found. The first feasible path found is the longest executable path in the virtual scope. The algorithm for longest feasible path search is given in Figure 12.

The removal of a path and finding the next longest one runs in $O(m)$ time, where $m$ is the number of edges in the graph. This comes from the fact that for each new node the *time_sum* and *predecessor* are computed

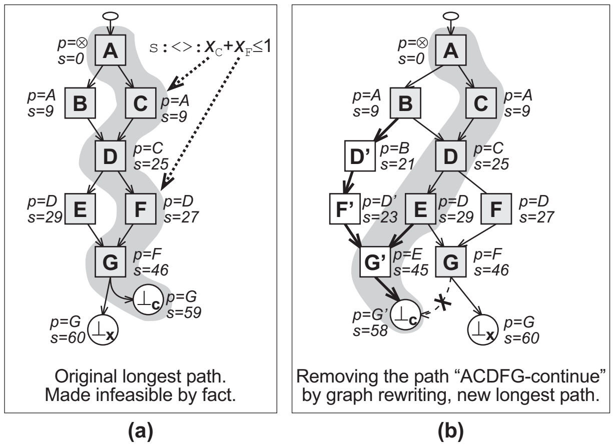


**Figure 11. Infeasible Path Removal**

```
$VirtualScopeTime(TG, vs, S)$ :
  $F$ := facts covered by $vs$
  /** Get time for longest continuation and exit paths **/
  $t_{cont}$ := $LongestFeasiblePathSearch(TG, F, \perp_c)$
  $t_{exit}$ := $LongestFeasiblePathSearch(TG, F, \perp_x)$
  /** Does there exist a feasible continuation path **/
  if $t_{cont}$ > 0 then
    /** Is $vs$ not the last virtual scope covering $S$ **/
    if lastiter($vs$) $\neq$ loopbound($S$) then
      return {$t_{cont}$ * sizeof($vs$), $false$}
    /** We must take the exit path **/
    else if $t_{exit}$ > 0 then
      return {$t_{cont}$ * (sizeof($vs$)-1) + $t_{exit}$, $true$}
  /** We must take the exit path **/
  else if $t_{exit}$ > 0 then
    return {$t_{exit}$, $true$}
```

**Figure 13. WCET Calculation for Virtual Scope**

```
$LongestFeasiblePathSearch(TG, F, endnode)$ :
  /** 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**

```
$LongTimingEffectExpansion(TG)$:
 /** 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 $(v', w)$ 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**

by traversing all incoming edges of the corresponding original node (see [21] for details). However, this complexity is obtained only if the next longest path is found after scanning the entire set of edges, which is the case only when a path passes all nodes. For a typical flow graph, this is not realistic. Thus the actual complexity can be assumed to be much lower.

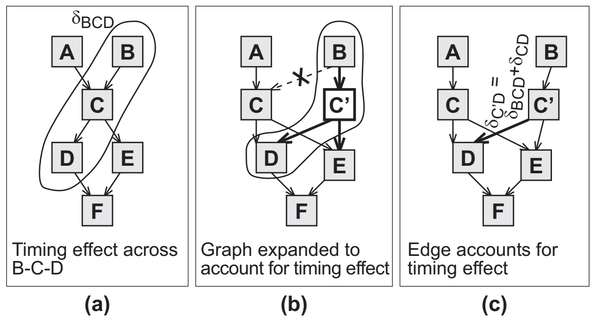
After one initial run of Dijkstra's Algorithm, the path search algorithm runs in $O(K * m)$ time, where $m$ is the number of edges in the graph and $K$ is the number of paths removed. As the number of paths in a flow graph grows exponentially with the number of decisions (see Section 4 above), the whole path search might take exponential time, since in the worst case all paths have to be examined (if the shortest path is the only feasible path). However, for typical programs this is very unlikely and may only happen when we have many complex flow facts covering the same virtual scope. It was not a noticeable problem for any of our benchmark programs (see Section 7). Thus, in general, the number of paths examined should be low compared to the total number of possible paths. Also note that since the calculation successively improves the WCET estimate in each step, it can be interrupted at any time, still yielding a safe, but probably pessimistic result.

The algorithm given in Figure 13 returns the WCET for a virtual scope. It also returns whether the execution was forced to take an exit path, either due to flow facts or because the virtual scope contained the last iteration of the original scope.

The times extracted for each virtual scope are combined together to generate a WCET for the entire scope, as given by the algorithm in Figure 10.

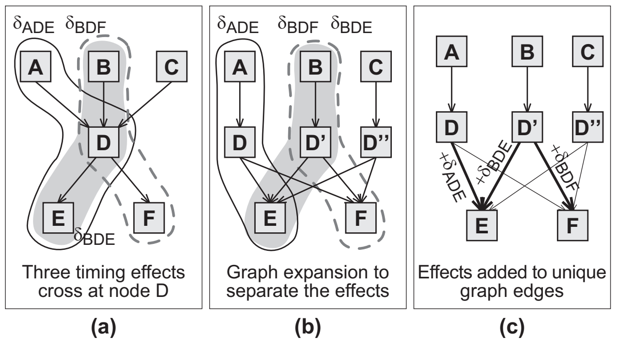
## 5.4. Optimizations

We can use the set of facts covering each virtual scope to reduce the number of calls to the path-removal algorithm. If the set of facts covering a virtual scope $vs_i$ is a subset or equal to the set of facts covering a virtual scope $vs_j$ then all paths removed from $vs_i$ can



δBCD
Timing effect across B-C-D
Graph expanded to account for timing effect
δC'D = δBCD+δCD
Edge accounts for timing effect
(a)
(b)
(c)

**Figure 15. Simple Timing Effect**



δADE δBDF
δBDE
Three timing effects cross at node D
δADE δBDF δBDE
Graph expansion to separate the effects
+δADE
+δBDE
+δBDF
Effects added to unique graph edges
(a)
(b)
(c)

**Figure 16. Crossing Timing Effects**

be safely removed also from $vs_j$. This means that when doing a longest path search over $vs_j$ we can instead start with the resulting graph after the longest-path search over $vs_i$.

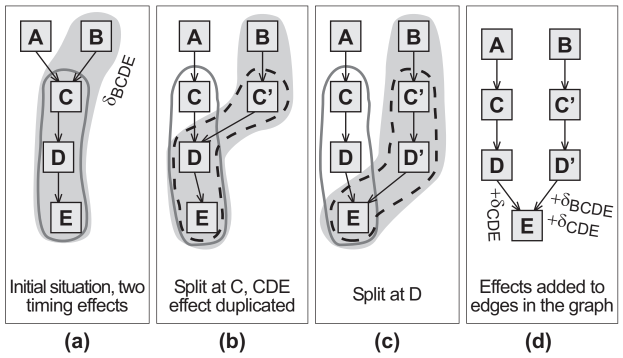
## 6. Handling Long Pipeline Effects

If there are pipeline effects across sequences of nodes longer than two, they must be considered during the path search since they affect the longest path. An example is shown in Figure 18, where the timing effect on the sequence `CDE` increases the execution time of that path by 3 cycles, making the longest path different from the one shown in Figure 6. Path-based methods have previously required complete paths to be executed to capture such effects [14], while here we show how to capture the effects locally by graph rewriting. We have previously demonstrated how to handle such effects in IPET [7].

To account for such effects, we need to know when we have taken a path containing the sequence of nodes corresponding to the timing effect. Since the longest path search only looks at the predecessors for a node, a preprocessing algorithm, given in Figure 14 and corresponding to the box "Timing Effect Expansion" in Figure 1, must be used.

The idea is to make each path that contains a long timing effect separate in the graph, and to add the time of the timing effect to the last edge in the sequence.

This is obtained by traversing the graph in topological order, and for each node visited, if it is inside a timing effect sequence (i.e. not the first or last of a sequence), and has more than one incoming edge, it is copied together with its outgoing edges and the incoming edge that is part of the sequence. The original incoming edge is removed. The copied nodes and edges have the same timing effects as the original nodes and edges.



δBCDE
Initial situation, two timing effects
Split at C, CDE effect duplicated
Split at D
+δCDE
+δBCDE
+δCDE
Effects added to edges in the graph
(a)
(b)
(c)
(d)

**Figure 17. Overlapping Timing Effects**

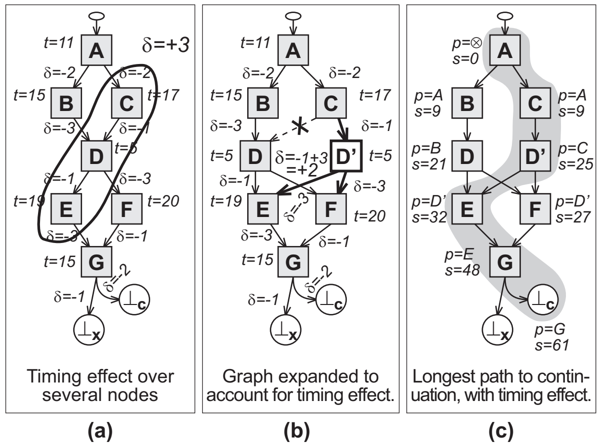


**Figure 18. Path Search with Timing Effects**

Figure 15 shows the basic idea, and how the timing effect is constructed for the final edge in the sequence.

Figure 16 shows what happens when several time effects cross each other. A number of new nodes get added, and three edges rewritten. However, the result is still logical, and the algorithm handles this complex case without a hitch.

Figure 17 shows the case where two timing effects overlap. The graph must be expanded in such a way that we count both effects only if the sequence `BCDE` is taken, and only the effect over `CDE` if only that path is taken. The end result is a graph where the effect $\delta_{\texttt{CDE}}$ is added to two edges.

The process is illustrated in Figure 18. The node `D` is copied to `D'` due to the timing effect CDE, and

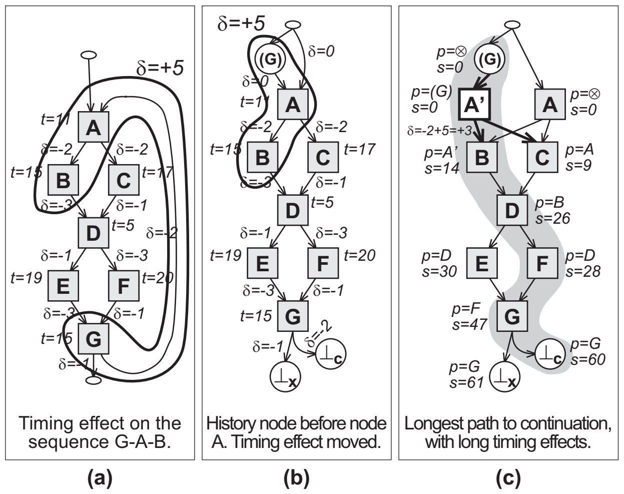


**Figure 19. Timing Effect Across Back-Edge**

the longest path changes compared to the base version shown in Figure 6.

### 6.1. Long Timing Effects over Analysis Boundaries

One problem with a bottom-up WCET calculation such as the one presented in this paper is that there might be long timing effects across the boundaries between successive paths or scopes, which have to be handled in order to ensure a safe WCET estimate.

We can have a *wrap-around timing effect* across the loop back-edge as illustrated in Figure 19(a), or pipeline timing effects between nodes in a parent scope and nodes in a subscope when entering or exiting the subscope (*border crossing timing effects*).

Our solution is to add special *history nodes* to the timing graph. The history nodes represent the potential paths taken *before* the beginning of a path search and therefore have no weight associated with them. Long timing effects begin at the history node and end at the usual end node (as shown for the timing effect from "`(G)`" to `B` in Figure 19(b)).

For each node $v$ in the graph that is the target of a back-edge or an entry to the scope, we collect all timing effects with a length greater than two (those of length two are already handled as regular edge times). For each such timing effect we create a history node labelled with the sequence of nodes up to $v$, and insert it between the start node and the node $v$.

For example, the sequence `GAB` in Figure 19, 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 18(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, while in reality the nodes corresponding to the history node might not have to be executed each time node $v$ is executed. 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 20.

The results of the execution time analysis are shown in Figure 21. The column *Basic* gives the WCET estimate using only loop-bounds as flow limiting information and ignoring pipeline overlap *between* nodes (but including the pipeline overlap *within* nodes[2]). Columns including *Flow* hold WCET estimates resulting from adding flow facts to the programs. Columns including *Pipeline* hold WCET estimates where pipeline effects both within and 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 worse 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.

For two programs (`fibcall`, `matmult`), no special flow information is needed in order to capture the WCET (i.e. the structural WCET is the actual WCET), and the precise estimates we get show that the basic pipeline modeling introduces no pessimism.

For `jfdctint`, the facts we could express reduced 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. In `crc`, there are some complex if-statements that are hard to model exactly.

Figure 22 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 to account for facts with ranges. *Paths* shows the number of possible execution paths (summed over all virtual scopes, before applying facts to the graph), and *Expl.* the number of paths that our search

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

| 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 20. Benchmark Programs**

| | 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 21. 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 22. Complexity Measures**

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.

The computation time needed for the analysis was negligible for all our benchmark programs, except for `nsichneu`, where the analysis took a few seconds on a Sparc Ultra 5 (due to the size of the program).

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 paths of a program, 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 reusability of previously developed modules. Also, by putting the pipeline timing analysis and the calculation into separate modules, the WCET tool is easier to retarget.

Our experiments show that the new calculation method generates tight and safe WCET estimates for many programs, and that flow information can be used effectively to improve the quality of the estimates.

The WCET analysis also generates the precise path that gives rise to the WCET for a program, making it easier for programmers to fix performance problems.

For future work, we plan to compare the calculation method with an IPET-based calculation within the same framework [8]. This comparison will be very fair, since all other components of the tool (test programs, pipeline analysis, compiler, etc.) will be exactly the same. Previously, it has not been possible to make reasonable comparisons between calculation methods, since no two tools have used the same target system and input.

We are also considering whether it is possible to create a hybrid approach between the path-based and IPET-based calculations, 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).

Finally, we would like to consider a novel use for the flow facts: using flow facts to guide cache- and branch-prediction analysis should provide us with the opportunity to obtain more precise results and thus less pessimism.

## 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*, 3rd edition, January 1999. Document no. U12197EJ3V0UM00.

[7] J. Engblom and A. Ermedahl. Pipeline timing analysis using a trace-driven simulator. In *Proc. 6th 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. 21th 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 embeded 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. 5th IEEE Real-Time Technology and Applications Symposium (RTAS'99)*, pages 79–88, June 1999.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

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

[22] 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.

[23] 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.

[24] Chang Yun Park. Predicting program execution times by analyzing static and dynamic program paths. *Real-Time Systems*, 5(1):31–62, March 1993.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

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

[31] 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.