Clustered Calculation of Worst-Case Execution Times

Andreas Ermedahl[*]
Mälardalen Real-Time Research Center
Box 883, S-72123 Västerås
Sweden
andreas.ermedahl@mdh.se

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

Jakob Engblom[*]
Virtutech
Norrtullsg. 15, 11327 Stockholm
Sweden
jakob@virtutech.com

ABSTRACT

Knowing the Worst-Case Execution Time (WCET) of a program is necessary when designing and verifying real-time systems. A correct WCET analysis method must take into account the possible program flow, such as loop iterations and function calls, as well as the timing effects of different hardware features, such as caches and pipelines.

A critical part of WCET analysis is the calculation, which combines flow information and hardware timing information in order to calculate a program WCET estimate. The type of flow information which a calculation method can take into account highly determines the WCET estimate precision obtainable. Traditionally, we have had a choice between precise methods that perform global calculations with a risk of high computational complexity, and local methods that are fast but cannot take into account all types of flow information.

This paper presents an innovative hybrid method to handle complex flows with low computational complexity, but still generate safe and tight WCET estimates. The method uses flow information to find the smallest parts of a program that have to be handled as a unit to ensure precision. These units are used to calculate a program WCET estimate in a demand-driven bottom-up manner. The calculation method to use for a unit is not fixed, but could depend on the included flow information and program characteristics.

Keywords

WCET analysis, WCET calculation, hard real-time, embedded systems.

Categories and Subject Descriptors

D.2.8 [**Metrics**]: Performance Measures; J.7 [**Computers in Other Systems**]: Real Time; D.4.8 [**Performance**]: Modeling and Prediction

General Terms

Algorithms,Performance,Measurement

CASES'03, Oct. 30–Nov. 2, 2003, San Jose, California, USA.
Copyright 2003 ACM 1-58113-676-5/03/0010 ...$5.00.

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 real-time systems, especially when real-time systems are used to control safety-critical systems like vehicles, military equipment and industrial power plants.

WCET estimates are used in real-time systems development to perform scheduling and schedulability analysis, to determine whether performance goals are met for periodic tasks, and to check that interrupts have sufficiently short reaction times. 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, they must also be *tight*, i.e. avoid large overestimations.

A correct WCET calculation method must take into account the possible program flow, like loop iterations and function calls, as well as effects of hardware features, like caches and pipelines. The flow information can be considered as a set of *flow facts*, each providing a certain piece of information about the program (like loop bounds, infeasible paths, etc.). The expressiveness of the flow facts a calculation method can handle is in high degree determining the WCET estimate precision that can be achieved.

In this paper we present a method to handle complex flow information with low computational complexity while still generating safe and tight WCET estimates. We use the key observation that flow facts are usually local in their nature, expressing information that only affects a small region of a program. However, these regions might be larger than the units used in local calculation schemes (a loop nest rather than a loop, or an entire function rather than just a loop inside that function). In general, the boundaries for a flow fact might not agree with the boundaries of a calculation scheme based on the structure of a program. When flow facts cross structural boundaries, and thus calculation boundaries, they cannot be accounted for, which leads to lower precision.

One solution to the boundary problem is to work globally on the entire program at once. However, this has a potentially high complexity, and makes scaling to large pro-

[*] This work is performed within the Advanced Software Technology (ASTEC, www.docs.uu.se/astec) competence center, supported by the Swedish National Board for Industrial and Technical Development (NUTEK, www.nutek.se).

[†] Friedhelm is a PhD student at C-LAB (www.c-lab.de), which is a cooperation of Paderborn University and Siemens.

grams risky. Almost all techniques for performing global calculations are based on integer linear programming (ILP) or constraint programming (CP) techniques, thus having a complexity potentially exponential in the program size.

However, by structuring a local calculation after the boundaries dictated by the flow facts, it is possible to achieve both efficient local calculation and high precision, since all facts can thus be accounted for while still avoiding the need for global calculation (unless there are actual flow facts that make this necessary).

Another key observation is that in many cases it is not sufficient to consider each provided flow fact in isolation. Many different types of flow facts might be generated for the same program and such flow facts can *interact* and together constrain program flow in a manner not possible by single flow facts. A WCET calculation method working over smaller program parts therefore must, to achieve maximum precision, find interacting flow facts and treat them as a unit.

This is achieved by our *clustered calculation*, which basically works as follows: The provided flow information is used to construct units where the included flow facts all have to be considered together. For each such *fact cluster* the part of the program covered by the included flow facts is extracted. The fact clusters and corresponding program parts are used to calculate a program WCET estimate in a demand-driven bottom-up manner. The calculation method to use for a particular fact cluster is not fixed, but could depend on the characteristics of the included flow facts and corresponding program parts.

The concrete contributions of this paper are:

- We introduce the concept of organizing flow information into fact clusters.
- We present various algorithms to construct fact clusters.
- We present an algorithm that uses fact clusters to calculate a program WCET estimate.
- We evaluate the clustered calculation method against global and local calculation schemes.

The rest of this paper is organized as follows: Section 2 introduces previous work, Section 3 presents our WCET tool architecture, and Section 4 presents our flow representation for WCET analysis. Section 5 presents how flow facts can be organized into fact clusters, and Section 6 gives the clustered WCET calculation method. Section 7 presents different calculation alternatives. Section 8 gives an illustrating example of the clustered calculation method. Finally, Section 9 presents our experimental evaluation, and Section 10 gives our conclusions and ideas for future work.

2. WCET ANALYSIS OVERVIEW

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

The purpose of the flow analysis phase is to extract the dynamic behaviour of the program. This includes information on what functions get called, how many times loops iterate, if there are dependencies between `if`-statements, etc. Since the flow analysis does not know the execution path which corresponds to the longest execution time, the information must be a safe (over)approximation including *all* possible program executions. The information can be obtained by *manual annotations* (integrated in the programming language [18] or provided separately [8, 11, 19]), or by *automatic flow analysis* [12, 13, 16, 22, 29].

The purpose of low-level analysis is to determine the timing behaviour of instructions given the architectural features of the target system. For modern processors it is especially important to study the effects of various performance enhancing features, like caches and pipelines. Low-level analysis can be further divided into *global* low-level analysis, for effects that require a global view of the program, and *local* low-level analysis, for effects that can be handled locally for an instruction and its neighbours.

In global low-level analysis, instruction caches [15, 13, 19, 20, 29], data caches [17, 29, 31], and branch predictors [5, 23] have been analyzed. Local low-level analysis has dealt with scalar pipelines [5, 6, 7, 10, 13, 20, 22, 29] and superscalar CPUs [21, 28]. Heckmann et al. [15] present an integrated cache and pipeline analysis, and argue that such integration is necessary for processors with heavy interdependencies between various functional elements. Attempts have also been made to use measurements and the hardware itself to extract the timing [26].

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

In a tree-based approach, the WCET is calculated in a bottom-up traversal of a tree generally corresponding to a syntactical parse tree of the program, using rules defined for each type of compound program statement (like a loop or an `if`-statement) to determine the WCET at each level of the tree [3, 4, 5, 20]. The method is conceptually simple and computationally cheap, but has problems handling flow information, since the computations are local within a single program statement and thus cannot consider dependencies between statements.

In a path-based calculation, the WCET estimate is generated by calculating times for different paths in a program, searching for the overall path with the longest execution time [13, 29, 30]. The defining feature is that possible execution paths are *explicitly* represented. The path-based approach is natural within a single loop iteration, but has problems with flow information stretching across loop-nesting levels.

In IPET, program flow and low-level execution time are modeled using arithmetic constraints [8, 11, 16, 19, 25, 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 WCET is extracted by maximizing $\sum_{i \in entities} x_i * t_i$, where the x_i are subject to constraints reflecting the structure of the program and possible flows. The result is a worst-case count for each node and edge. As shown in [8], very complex flows can be expressed using constraints, but the computational complexity of solving the resulting problem is potentially very high, since the program is completely unrolled and all flow information is lifted to a global level.

Both the path-based and tree-based calculation methods are performed in a *bottom-up* fashion. Bottom-up calculation methods calculate a safe (timing) abstraction for a part of the program, which is later used in the calculation of surrounding parts of the program. For the tree-based calculation the abstraction unit is a program statement, while for

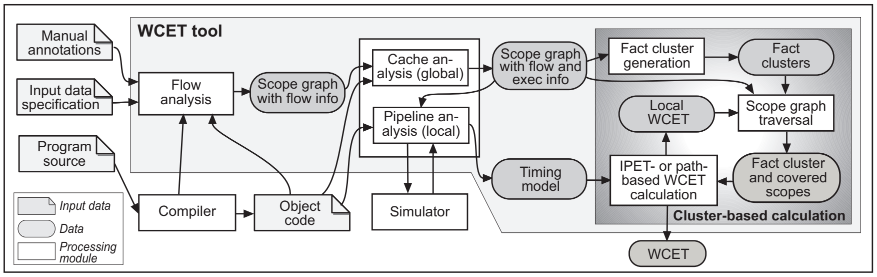


Figure 1: WCET tool architecture

the path-based calculation it is a loop or a function. Bottom-up calculations are beneficial, since the overall WCET calculation problem can be subdivided into smaller easier-to-solve problems. At the other extreme we have the IPET methods, where no subdivision is made and the unit of calculation is the entire program.

In this paper we present an approach where the part of the program calculated locally is not predetermined statically but depends on the flow information available for the program. The calculation is performed bottom-up, but is demand-driven in that a WCET for a program part is only calculated when its timing estimate is needed in a surrounding program part.

3. TOOL OVERVIEW AND TERMINOLOGY

The work presented in this paper is implemented within the framework of our existing WCET analysis tool. In addition to previous implemented extended IPET-based [8, 9] and path-based [9, 30] calculation methods, we have implemented a calculation module based on clustering. Figure 1 gives an overview of the WCET tool, when using a clustered calculation module (as presented in this paper). Compared to our previously presented work [8, 30] all components of the system except the calculation phase remain unchanged, demonstrating the modular structure of the tool. The modular architecture allows independent replacement of the modules implementing the different steps, which makes it easy to customize a WCET tool for particular target hardware and analysis needs.

All data structures and analysis phases in our WCET tool are based on the possibility of partitioning the instructions in the object code into *basic blocks*[1]. Figure 2(a) shows an example C function, Figure 2(b) and Figure 2(c) show the corresponding assembler code and basic block graph.

We have an automatic flow analysis currently under development [12]. The flow analysis results in a description of the dynamic behaviour of the program, consisting of a *scope graph* annotated with *flow facts* (Figure 2(d)), as described in more detail in Section 4 below.

For the current experiments, we rely on a machine model for a NEC V850E [6, 7] that accurately models a processor pipeline using a trace-driven cycle-accurate simulator. For the V850E target, caches are not used. The resulting *timing model*, see Figure 2(f), is a data structure containing times for each entity (node or edge) in the scope graph. Times for nodes correspond to the execution times of basic blocks (with additional execution information[2]) in isolation, e.g. $t_{\mathtt{A}}$ in Figure 2(f), and times for edges, e.g. $\delta_{\mathtt{AC}}$ in Figure 2(f), to the *timing effect* when two successive nodes are executed in sequence [6, 7, 30]. These timing effects are usually negative due to the pipeline overlap between the two nodes. Timing effects reaching across node sequences longer than two are also taken into account where necessary.

This timing model is powerful enough to capture the effects of pipelines and caches, separating the analysis of machine aspects from the calculation phase. It is also not tied to our particular fashion of low-level analysis. For example, the integrated cache and pipeline analysis for the Motorola ColdFire 5307 processor presented by Ferdinand et al. [10] generates a model where times are assigned to basic blocks in a program (including the effect of both pipelines and caches on the timing of each block). Such a timing model can be used within our framework, with the clustered calculation method presented in this paper. Similarly, the timing model for Infineon C167 presented by Atanassov et al. [1] attributes times only to edges in the flow graph, and this model would also fit in our timing model framework.

4. REPRESENTING PROGRAM FLOW

The *scope graph* is a hierarchical representation of the dynamic behaviour of a program suitable for WCET analysis. The graph consists of nodes and edges where each node is referring to a basic block in the object code. A basic block might be referenced by several different scope nodes.

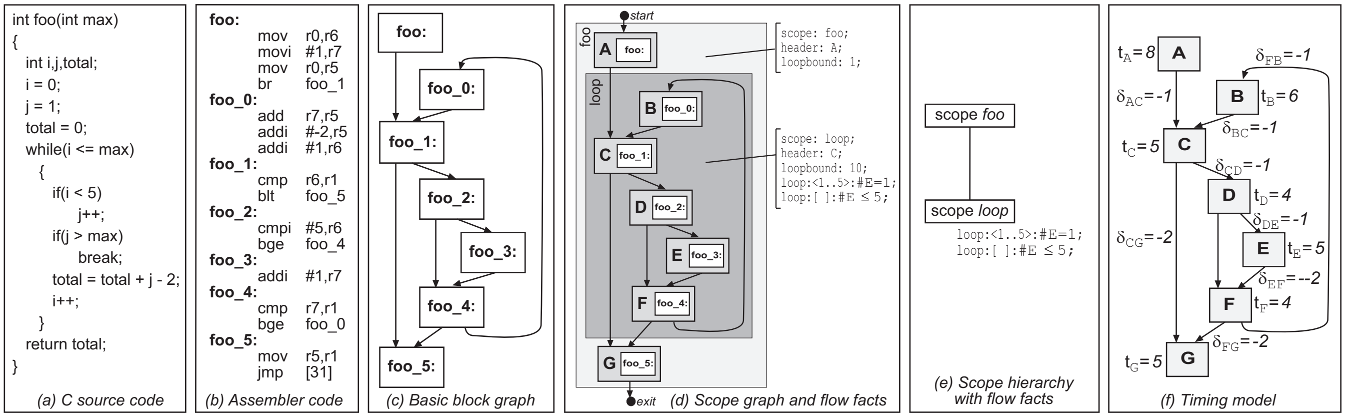
The nodes and edges in the scope graph are partitioned into *scopes* reflecting the dynamic structure of the program in terms of function calls, loops, recursive calls and unstructured code parts. Scopes are necessary in order to carry program flow information, in particular bounds for all loops and context-sensitive flow information for function calls. Figure 2(d) shows the scope graph generated for the code in Figure 2(a).

Each scope has a distinguished header node, (e.g. node `A` resp. `C` in Figure 2(d)), with the property that no other node in the scope can be executed more than once without passing the header node. Each scope should have a loop bound attached to it, providing an upper bound on the number of times its header node can be executed for each entry of the scope.

The scopes in the scope graph are organized in a *scope-hierarchy*, a directed tree with scopes as vertices and edges from a scope going to all its children. Figure 2(e) illus-

[1] A basic block is a maximal sequence of instructions that can be entered only at the first instruction in the sequence and exited only at the last instruction in the sequence [24].

[2] The nodes in the scope graph can be annotated with additional *execution information*, e.g. giving what instructions that will hit or miss the cache and the type of memory being accessed [6, 9] (not explicitly shown in Figure 2(f)).



```
int foo(int max)
{
   int i,j,total;
   i = 0;
   j = 1;
   total = 0;
   while(i <= max)
      {
         if(i < 5)
               j++;
         if(j > max)
               break;
         total = total + j - 2;
         i++;
      }
   return total;
}
```

(a) C source code

```
foo:
         mov     r0,r6
         movi    #1,r7
         mov     r0,r5
         br      foo_1
foo_0:
         add     r7,r5
         addi    #-2,r5
         addi    #1,r6
foo_1:
         cmp     r6,r1
         blt     foo_5
foo_2:
         cmpi    #5,r6
         bge     foo_4
foo_3:
         addi    #1,r7
foo_4:
         cmp     r7,r1
         bge     foo_0
foo_5:
         mov     r5,r1
         jmp     [31]
```

(b) Assembler code

(c) Basic block graph

```
scope: foo;
header: A;
loopbound: 1;
```

```
scope: loop;
header: C;
loopbound: 10;
loop:<1..5>:#E=1;
loop:[ ]:#E ≤ 5;
```

(d) Scope graph and flow facts

```
loop:<1..5>:#E=1;
loop:[ ]:#E ≤ 5;
```

(e) Scope hierarchy with flow facts

(f) Timing model

Figure 2: WCET analysis stages

trates the scope-hierarchy generated for the scope graph in Figure 2(d). In the tree each scope has zero or more *descendants*, i.e. scopes below it in the tree, and zero or more *ancestors*, i.e. scopes above it in the tree. The immediate descendants of a scope are its *child scopes* and the immediate ancestor is its *parent scope*. A scope without any descendant is called a *leaf scope*. E.g. in Figure 2 scope `loop` is a descendant and a child to scope `foo`. Scope `loop` is also a leaf scope.

The *complete subtree* for a scope s is formed by all scopes having s as ancestor in the scope-hierarchy (including s). Each tree of scopes formed by removing the complete subtrees of one or several descendant scopes of s is a *subtree* of s. An *in-edge* of a scope s is an edge having its source node in a scope not within the complete subtree of s and having its target within the complete subtree of s. An *in-node* is a target node of an *in-edge*. An *out-edge* of a scope s is an edge having its source node in a scope within the complete subtree of s and having its target outside the complete subtree of s. Timing effects reaching across scope boundaries are always taken into account via out-edges. That is, the timing effect associated with an out-edge of a scope s is included in the timing calculation of s, and, vice versa, the timing effect of an in-edge of s is taken into account by the calculation of the corresponding source scope. A scope can be entered at several in-nodes, allowing for unstructured jumps into loops, and might have several out-edges. An edge going to a header node of a scope s and having its source node located in the complete subtree of s is a *back-edge* of s. E.g. in Figure 2(d) `A`→`C` is an in-edge, `F`→`G` an out-edge and `B`→`C` a back-edge of scope `loop`.

4.1 Flow facts

To express more complex program flow information than just basic loop bounds each scope can carry a set of *flow facts* [8, 9]. The flow facts combine the expressive power of IPET, using constraints to limit possible executions of scope graph entities, with the ability to give the flow information in a scope-local context.

Each flow fact consists of three parts: the name of the *defining scope* where the fact is attached, a *context specifier*, and a *constraint expression* (see Figure 2(d)). Each flow fact is considered *local* to its defining scope and the fact is interpreted as being valid for *each entry* of the scope.

The context specifier describes the iterations for which the constraint expression is valid. This can be for all iterations or for just some iterations. The type of a context specification is either *total* (written with "`[`" and "`]`"), for which the fact is considered as a sum over all iterations of the specified scopes, or *foreach* (written with "`<`" and "`>`"), which considers the fact as being local to a single iteration of the scope. Facts valid for all iterations are expressed by "`<>`" or "`[]`", while facts valid for certain iterations are expressed as `<`*min..max*`>` or `[`*min..max*`]`, where $min \leq max$ are integers larger than 0.

The constraints are specified as a relation between two arithmetic expressions involving *execution count variables* and constants. An execution count variable, `#`*entity*, corresponds to an entity (node or edge) in the scope graph, and represents the number of times the entity is executed in the context given by the context specification.

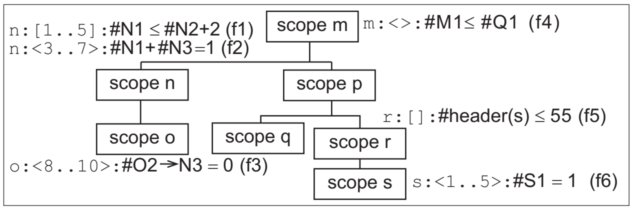
A fact can only refer to count variables corresponding to entities located in the complete subtree of the defining scope of the fact. For example, a fact defined in scope `loop` cannot refer to executions of entities located in the `foo` scope. All scopes between the defining scope and the scopes containing referred count variables are said to be *covered* by the fact. Thus, the scopes covered by a fact form a subtree with the defining scope as root.

For each scope covered by a fact the fact *spans* a number of iterations. For the defining scope the span is the number of iterations specified by the context specifier. For all other covered scopes the span is all iterations of the scope.

In Figure 2(d), the `loop` scope has two flow facts attached to it. The first flow fact specifies that for each time `loop` is entered, node `E` must be taken during each of the first five loop iterations (but not that the loop needs to iterate 5 times). The second fact specifies that for each time `loop` is entered node `E` can be taken at most five times. Observe that the facts are local to scope `loop`, and should be valid for each entry of the `loop`, independently on how many times function `foo` is called from other functions in the program.

5. CLUSTERING OF FLOW FACTS

The goal of clustering is to find the flow facts that need to be considered together in order not to lose precision. Such interacting flow facts are caused by facts sharing application area with some other facts, by reaching down into descendant scopes and by having overlapping range specifications. Together the flow facts also indirectly specify a part of the scope graph that needs to be considered together with the



(a) *Example scope-hierarchy with associated facts*

Fact	Defining scope	Span def scope	Covered scopes
f1	*n*	*1..5*	*{n}*
f2	*n*	*3..7*	*{n}*
f3	*o*	*8..10*	*{o}*
f4	*m*	*1..lb(m)*	*{m,p,q}*
f5	*r*	*1..lb(r)*	*{r,s}*
f6	*s*	*1..5*	*{s}*

(b) *Information about facts*

Fact cluster	Defining scope	Span def scope	Covered scopes
{f1,f2}	*n*	*1..7*	*{n}*
{f3}	*o*	*8..10*	*{o}*
{f4}	*m*	*1..lb(m)*	*{m,p,q}*
{f5,f6}	*r*	*1..lb(r)*	*{r,s}*
{f6}	*s*	*1..5*	*{s}*

(c) *Information about fact clusters*

Figure 3: Fact clustering example

flow information in the calculation.

We define a *fact cluster* to be a set of flow facts. The *defining scope* of a fact cluster is defined to be the first common ancestor of all the facts in the cluster. The *cover* of a fact cluster is all scopes between the defining scope and the scopes containing count variables referred to by a flow fact in the scope. Thus, the covered scopes form a subtree in the scope-hierarchy with the defining scope as root. For the defining scope s of a cluster the *span* is all iterations between the lowest and highest iteration of s spanned by any fact in the cluster. For all other covered scopes the span is equal to all iterations of the scope.

In Figure 3(a) an example scope-hierarchy with associated flow facts is given. In Figure 3(b) we show the defining scopes, defining scope spans, and cover of each given flow fact. The name of a referred count variable gives the scope in which the corresponding entity is located, e.g. `#N1` refers to executions of node `N1` located in scope `n`. The function `lb(`s`)` returns the loop bound for a scope s.

The fact clusters generated from the facts are given in Figure 3(c). For each generated fact cluster we show its defining scope, its defining scope span, and the scopes covered by the cluster. Note that the same flow fact can be present in several clusters, and that not all flow facts in a cluster need to have the same defining scope.

5.1 Flows causing clusters

Program flows causing fact clusters and reaching over several scopes are actually quite common. The simplest example is illustrated in Figure 4. It is the classical "triangular" loop, i.e. a nested loop where the number of iterations of the inner loop depends on the current iteration number of the outer loop (cf. scopes `r` and `s` in Figure 3(a)).

```
for(i=0; i<10; i++)       // Bound:  10, (scope r)
   for(j=i; j<10; j++)    // Local bound:  10, (scope s)
      { ...  }            // executed at most 55 times
```

Figure 4: Triangular loop

The inner loop considered in isolation will have an iteration bound of 10, and so will the outer loop. If WCET calculation is performed locally, the WCET calculation for the inner loop will assume 10 iterations, and the WCET calculation for the outer loop will use 10 executions of the inner loop, leading to the body of the inner loop being counted 100 times, when it is actually never executed more than 55 times. This requires that we handle the inner and outer loop together. Flow fact `f5` in Figure 3(a) shows how this type of triangular loop dependency can be captured, (`#header(`s`)` refers to the count variable of the header node of a scope s).

```
void foo(bool x) {      // Function foo(), (scope m)
   if( cond )
      x = true;         // Block M1
   for( ...  )          // Outer loop, (scope p)
      for (...  )       // Inner loop, (scope q)
         if( x )
            Q1          // Block Q1, execution implied by M1
}
```

Figure 5: Long reaching dependency

Flows in nested scopes can be related in other ways, for example if the outcome of a decision in a scope determines the paths taken in a loop (maybe deeply) nested in the scope (with varying outcome), like e.g. for the scopes `m`, `p` and `q` in Figure 3(a). Figure 5 shows example code with such a long reaching dependency. Flow fact `f4` in Figure 3(a) captures this type of nested dependency. It gives that an execution of `M1` *implies* an execution of `Q1`, (node `Q1` can still be executed on its own).

```
for( ...  )  {          // Bound:  10, (scope o)
   if( cond ) {         // Block O2, false during last 3 iters
      N3;               // Block N3, big chunk of work
      break;
   }
...
}
```

Figure 6: Condition dependent dependency

In the next example, shown in Figure 6, block `N3` does not belong to the loop (due to the `break` statement), and the way the loop is exited will determine whether it should be counted or not. Thus, `N3` depends on the decision `cond` in the loop body, but `N3` is a node in the parent scope of `o` (scope `n`). Fact `f3` in Figure 3 captures this dependency by specifying that the edge `O2`→`N3` can not be taken during the last three iterations of the `o` scope.

Another case of flow information causing clusters is when information from different types of flow analysis methods or manual annotations interact, and therefore need to be considered together in the WCET calculation. An example of such overlapping flow information is shown in Figure 3(a) with flow facts `f1` and `f2`. Both flow facts have the same defining scope `n` and they overlap in the ranges of their context specifications.

5.2 Fact clustering algorithm

An algorithm to create the clusters of flow facts is given in Figure 7. The algorithm makes a post-order traversal of the scope graph, where all clusters for a descendant scope are generated before its parent scope is processed.

For each scope s, we look at the facts defined on the scope, and partition the facts based on their range specifications. Two facts with ranges that overlap, i.e. have some iteration numbers in common, go to the same set: $\forall f_i, f_j \in \texttt{facts}(s) : (\texttt{overlap}(\texttt{span}(f_i, s), \texttt{span}(f_j, s)) \wedge f_i \in c) \Rightarrow f_j \in c$. This creates sets of facts where the defining scope span of each fact overlaps one or more of the facts in the same set.

For example, fact `f1` and `f2` in Figure 3 have overlapping ranges and the same defining scope `n`, and should therefore

```
$ClusterFacts$(scopegraph $sg$):
  $FC$ := $\emptyset$ // To hold generated fact clusters
  // Traverse scopes in scope graph bottom up
  for each scope $s$ in $sg$ in bottom-up order do
    $F$ := flow facts in $sg$ with $s$ as defining scope
    // Partition facts into clusters
    $C$ := partition facts in $F$ over span of $s$ into
      overlapping sets of facts
    // Add fact clusters already created in descendant scopes
    for each fact cluster $c$ in $C$ do
      $Cov$ := scopes covered by $c$ except scope $s$
      for each fact cluster $c_{sub}$ in $FC$ defined in $Cov$ do
        $c$ := $c \cup c_{sub}$
      end for
    end for
    // Update set of fact clusters
    $FC$ := $FC \cup C$
  end for
  return $FC$
```

Figure 7: Minimal fact clustering algorithm

be put in the same set. Note that if there are any "all iterations" facts (using context specification [ ] or < >), there will only be one fact set for this scope since these facts include all iterations, and thus overlap with all other facts defined on the scope.

We also need to consider interactions of flow facts located in different scopes. For each extracted fact-cluster we check if it covers any descendant scopes. For all covered descendant scopes all facts in clusters defined on these scopes are added to the cluster, together covering a set of scopes that have to be considered jointly. Note that this means that a fact can be part of several fact clusters. For example, fact f5 in Figure 3 covers both scope r and s and should therefore be clustered together with fact f6, resulting in the fact cluster {f5,f6} with r as its defining scope. Fact f6 is at the same time the only fact in the cluster having s as defining scope.

The algorithm given in Figure 7 generates minimal fact clusters, i.e. sets of facts where all included facts need to be considered together, but includes as few facts as possible. We call this clustering algorithm *minimal fact clustering*. It is also possible to form larger clusters, (note that any clustering has to put all interacting facts in the same unit), and natural examples of such clusterings are:

- *Scope-based clustering*: All facts defined in a scope are put in the same cluster, together with all the facts in fact clusters defined in covered descendant scopes.
- *Maximum clustering*: All flow facts in the scope graph are put into one big cluster with the first common ancestor scope as its defining scope. Scopes not covered by the resulting fact cluster will be calculated separately from the scopes in the cluster.
- *Global clustering*: All flow facts in the scope graph are put into one big cluster with the root scope of the scope graph as its defining scope. All scopes in the scope-graph are part of the cluster. This is identical to the global calculation view used by our Extended IPET method [9].

Furthermore, we can construct even smaller clusters by subdividing foreach facts into facts valid for smaller ranges. A foreach fact gives flow information valid for *each individual* iteration and therefore does not need to force overlapping subranges to the same cluster. Instead, we apply the algorithm given in Figure 7 to total facts only. The remaining foreach facts are *split* into new foreach facts across the ranges of the resulting clusters. E.g. in Figure 3(a) the total fact f1 does not overlap f2 completely, so we split f2 into the facts n:<3..5>:#N1 + #N3 = 1 (f2') and n:<6..7>:#N1 + #N3 = 1 (f2"). The resulting fact clusters become {f1,f2'} and {f2"}. We call such clustering *split-foreach-fact minimal clustering*. Compared to the minimal clustering algorithm, splitting of foreach facts will result in more fact clusters with smaller covers.

```
$ScopeWCET$(scope $s$, scopegraph $sg$, factclusterset $FC$,
            timedatabase $tdb$):
  // Initialize timing variables for scopes and clusters
  $t_{s,back}$ := $t_{s,out}$ := $t_{c,back}$ := $t_{c,out}$ := 0
  // Get fact clusters for scope $s$
  $C_s$ := fact clusters in $FC$ with $s$ as defining scope
  $C_s$ := add empty cluster for each range of $s$ not
    covered by fact clusters in $C_s$
  // Make WCET calculation over clusters
  for each cluster $c$ in $C_s$ in increasing range order do
    $st_c$ := subtree of scopes in $sg$ covered by $c$
    // Replace non-covered descendant scopes with timing nodes
    for each child scope $sub$ to leaf scopes in $st_c$ do
      // Do demand-driven analysis of descendant scopes
      if time for $sub$ is not in $tdb$ then
        $tdb$ := $ScopeWCET(sub, st, cs, tdb)$
      // Replace calls to descendant scopes with timing nodes
      $t_{sub}$ := time for scope $sub$ in $tdb$
      $st_c$ := in $st_c$ replace call to $sub$ with
        node taking $t_{sub}$ time
    end for
    // Get begin nodes for cluster
    if $c$ spans first iteration of $s$ then $b$ := in_nodes($s$)
    else $b$ := header_node($s$)
    // Calculate time to out-edges for cluster
    $t_{c,out}$ := $ClusterWCET(c, b, \text{out\_edges}(s), st_c, tdb)$
    // Update time to out-edges for scope
    if valid($t_{c,out}$) then
      $t_{s,out}$ := max($t_{s,back} + t_{c,out}$,$t_{s,out}$)
    // Calculate time to back-edges for cluster
    if $c$ does not span last iteration of $s$ then
      $t_{c,back}$ := $ClusterWCET(c, b, \text{back\_edges}(s), st_c, tdb)$
      // Update time to back-edges for scope
      if valid($t_{c,back}$) then
        $t_{s,back}$ := $t_{s,back} + t_{c,back}$
      // Break if execution can't continue
      else break loop
  end for
  // Update timing database and return
  add time $t_{s,out}$ for scope $s$ to $tdb$
  return $tdb$
```

Figure 8: Clustered WCET calculation

6. CLUSTERED WCET CALCULATION

The algorithm for calculating a WCET estimate using fact clusters is shown in Figure 8. The algorithm performs a demand-driven traversal of the scopes in the scope-hierarchy. For each scope we find the fact clusters defined on the scope, and for each fact cluster the scopes covered by the cluster are extracted as a subtree over which a local WCET calculation is made. This means that if there are fact clusters that cover more than one scope, a WCET calculation is performed over all covered scopes as a unit.

The WCET estimate for a scope s is obtained by iterating over the clusters having s as defining scope in range order, i.e. fact clusters spanning the first iterations of s are processed before fact clusters spanning later iterations of s. If some scope range is not spanned by any fact cluster, an *empty fact cluster* is created. Such empty clusters cover just the current scope and span only consecutive iterations not spanned by any fact. For a scope not covered by any flow fact, an empty cluster is created, spanning all iterations of the scope.

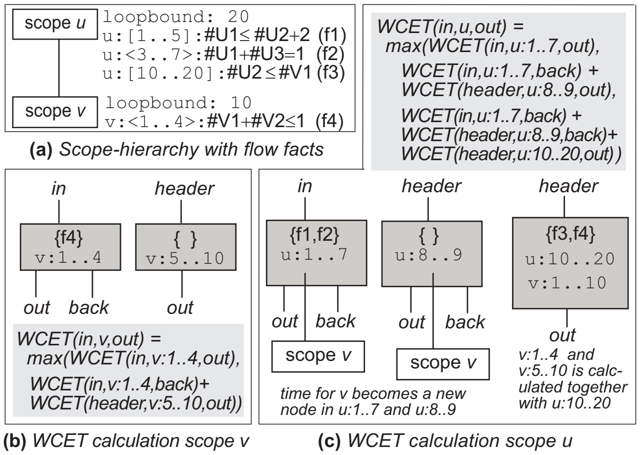

scope u
loopbound: 20
u:[1..5]:#U1≤ #U2+2 (f1)
u:<3..7>:#U1+#U3=1 (f2)
u:[10..20]:#U2≤#V1 (f3)
scope v
loopbound: 10
v:<1..4>:#V1+#V2≤1 (f4)
(a) Scope-hierarchy with flow facts
WCET(in,u,out) = max(WCET(in,u:1..7,out), WCET(in,u:1..7,back) + WCET(header,u:8..9,out), WCET(in,u:1..7,back) + WCET(header,u:8..9,back)+ WCET(header,u:10..20,out))
in
header
{f4}
v:1..4
{ }
v:5..10
out
back
out
WCET(in,v,out) = max(WCET(in,v:1..4,out), WCET(in,v:1..4,back)+ WCET(header,v:5..10,out))
in
header
header
{f1,f2}
u:1..7
{ }
u:8..9
{f3,f4}
u:10..20
v:1..10
out
back
out
back
out
scope v
scope v
time for v becomes a new node in u:1..7 and u:8..9
v:1..4 and v:5..10 is calculated together with u:10..20
(b) WCET calculation scope v
(c) WCET calculation scope u

Figure 9: Calculation over clusters

A timing estimate for a program fragment should be calculated from where the execution can enter the fragment to where the execution can exit the fragment. A calculation for a cluster is therefore performed from some *begin-nodes*, where the execution can enter the covered scopes, to some *end-edges*, where the execution can exit the covered scopes. The cluster spanning the first iteration of a scope has begin-nodes equal to the in-nodes of the scope. For the remaining clusters the begin-nodes are equal to the header-nodes of the scope, since this defines the start of a new iteration.

Similarly, for all clusters except the one including the last iteration, we make two distinct calculations, one ending at an out-edge and one ending at a back-edge. This is because the execution path taken to exit a scope might be different from the path taken when continuing to the next cluster. If a WCET estimate for the back-edges cannot be calculated, e.g. due to some contradicting flow information in the cluster, the execution can not continue. If so, we stop iterating over the ranges and return the total time accumulated for the scope.

Figure 9 shows an example of a WCET calculation working over clusters. The algorithm starts at scope u where there are several facts covering the iteration range. The name of a referred count variable gives the scope in which the corresponding entity is located, e.g. #V1 refers to executions of node V1 located in scope v. The facts f1 and f2 together form a cluster, {f1,f2}, spanning range 1..7 of u. Since neither f1 nor f2 cover v, a local calculation is made for v by a recursive call to the algorithm.

The local WCET calculation for v only needs to consider facts and fact clusters defined on v. Fact f4 creates a fact cluster on its own, {f4}. Two calculations are made for the {f4} cluster: one to the out-edges and one to the back-edges. The remaining iterations (5..10) of v are not spanned by any fact and an empty fact cluster { } is therefore created for these iterations. Since the fact cluster covers the last iteration of v, a WCET estimate is only made to the out-edges, and not to the back-edges as in the previous clusters.

After calculating a WCET estimate for v, the calculation restarts at u with the fact cluster {f1,f2}. There are no facts spanning range 8..9, and an empty fact cluster { } (covering just scope u) is created. For both these calculations, the call to scope v is represented by a call node with the timing of the extracted WCET estimate for v, i.e. no details of v except its timing are included in the calculation.

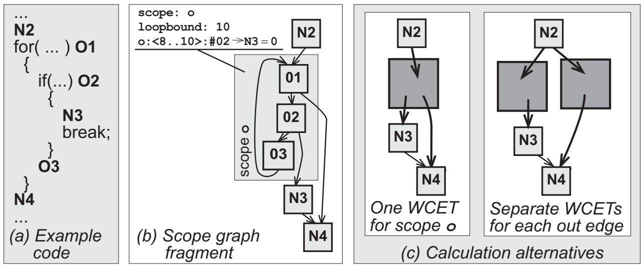


Figure 10: Calculation alternatives for graph fragment with multiple exits

Fact f3, however, covers both scope u and v and will be clustered together with the f4 fact as {f3,f4}. This means that when calculating a WCET estimate for scope u over the range 10..20 we cannot use the previously generated time for scope v, but must do the calculation over *both* u and v. Observe, that {f3,f4} covers the last iteration of u so no calculation for the back-edges is needed.

7. CALCULATION ALTERNATIVES

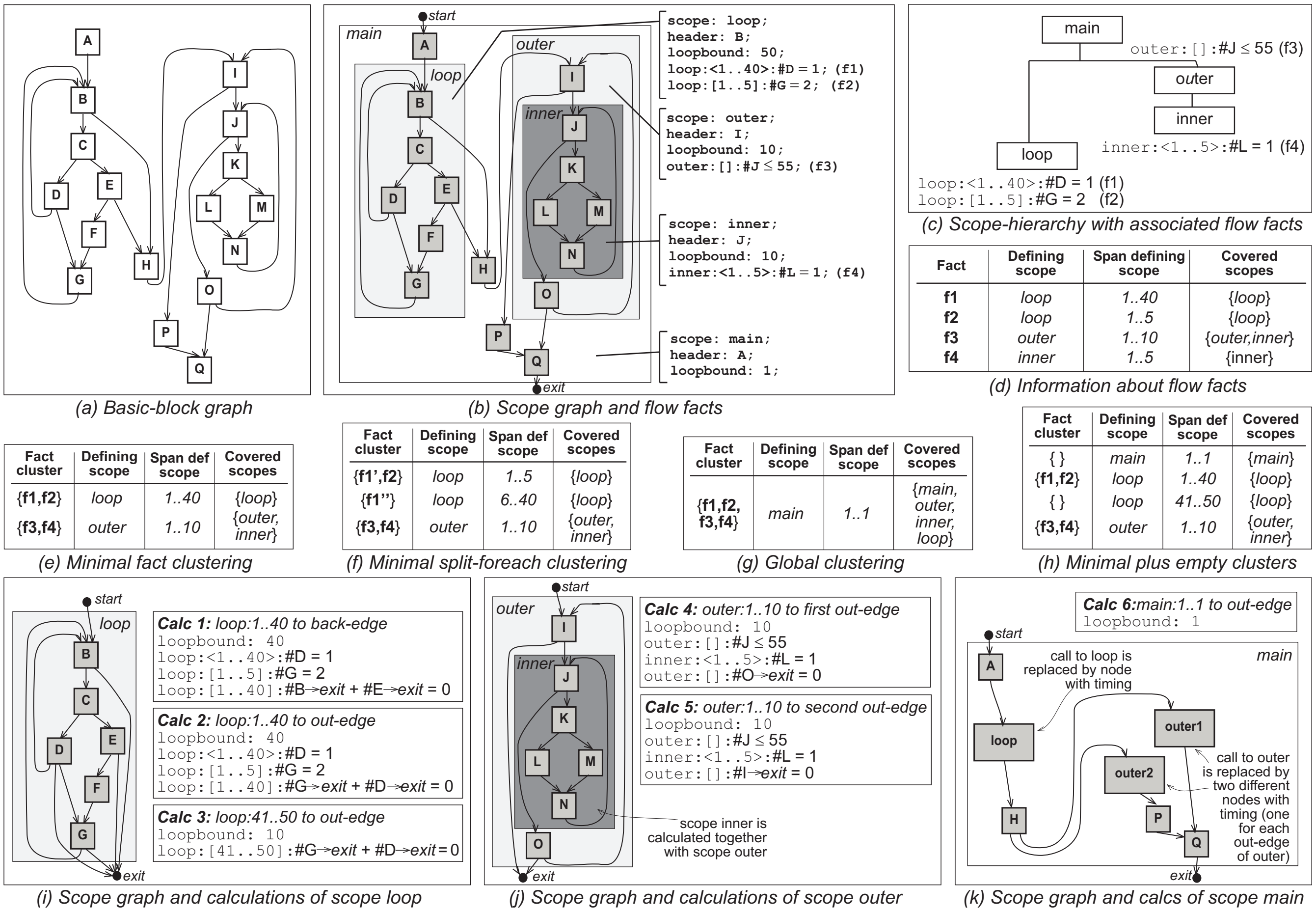
Some graph fragments have several points where the execution can enter or exit. For such fragments we have the option to make a separate WCET calculation for each pair of entry and exit points, or to make just one WCET calculation for all entry or exit points together, or to do something in between. This allows us to trade WCET estimate precision for calculation speed.

Figure 10 gives an example of the need for the calculation to differentiate between different exit points for increased precision. The code and scope graph corresponds to the example in Figure 6, where flow fact f3 specifies that edge O2→N3 cannot be taken during the last three iterations of scope o.

If only one calculation is made for scope o for both its out-edges it will result in a timing estimate for o which gives that the loop is iterated 10 full iterations. Later, when doing a WCET calculation for scope n the worst case path will be passing the call node for scope o together with the nodes N3 and N4. This gives a safe but pessimistic WCET estimate, since the extracted worst case path could not be taken in an actual execution.

The other calculation alternative, which is to make a separate WCET calculation for each out-edge of scope o, will discover that the out-edge to node N3 can not be taken during the last three iterations of o. The WCET estimate for scope o will therefore be different depending on the used out-edge. In the calculation of scope n, this will result in two separate call nodes for scope o, each with different timing. Thus, by making separate calculations for different in-nodes and out-edges the calculation cost increases, but more precise WCET estimates can be achieved.

Note that we only need to extract one single program fragment even though we perform separate calculations for its begin-nodes and end-edges. By adding extra flow facts stating which begin-nodes and end-edges are possible for each particular calculation, the extracted graph fragment can be reused.


A
B
C
D
E
F
G
H
I
J
K
L
M
N
O
P
Q
start
main
loop
outer
inner
exit
main
outer
inner
loop
start
loop
exit
start
outer
inner
exit
scope inner is calculated together with scope outer
start
main
call to loop is replaced by node with timing
loop
outer1
outer2
call to outer is replaced by two different nodes with timing (one for each out-edge of outer)
exit

(a) Basic-block graph

```
scope: loop;
header: B;
loopbound: 50;
loop:<1..40>:#D = 1; (f1)
loop:[1..5]:#G = 2; (f2)

scope: outer;
header: I;
loopbound: 10;
outer:[]:#J ≤ 55; (f3)

scope: inner;
header: J;
loopbound: 10;
inner:<1..5>:#L = 1; (f4)

scope: main;
header: A;
loopbound: 1;
```

(b) Scope graph and flow facts

```
outer:[]:#J ≤ 55 (f3)
inner:<1..5>:#L = 1 (f4)
loop:<1..40>:#D = 1 (f1)
loop:[1..5]:#G = 2   (f2)
```

(c) Scope-hierarchy with associated flow facts

Fact	Defining scope	Span defining scope	Covered scopes
f1	*loop*	*1..40*	{*loop*}
f2	*loop*	*1..5*	{*loop*}
f3	*outer*	*1..10*	{*outer,inner*}
f4	*inner*	*1..5*	{inner}

(d) Information about flow facts

Fact cluster	Defining scope	Span def scope	Covered scopes
{f1,f2}	*loop*	*1..40*	{*loop*}
{f3,f4}	*outer*	*1..10*	{*outer, inner*}

(e) Minimal fact clustering

Fact cluster	Defining scope	Span def scope	Covered scopes
{f1',f2}	*loop*	*1..5*	{*loop*}
{f1''}	*loop*	*6..40*	{*loop*}
{f3,f4}	*outer*	*1..10*	{*outer, inner*}

(f) Minimal split-foreach clustering

Fact cluster	Defining scope	Span def scope	Covered scopes
{f1,f2, f3,f4}	*main*	*1..1*	{*main, outer, inner, loop*}

(g) Global clustering

Fact cluster	Defining scope	Span def scope	Covered scopes
{ }	*main*	*1..1*	{*main*}
{f1,f2}	*loop*	*1..40*	{*loop*}
{ }	*loop*	*41..50*	{*loop*}
{f3,f4}	*outer*	*1..10*	{*outer, inner*}

(h) Minimal plus empty clusters

Calc 1: *loop:1..40 to back-edge*
```
loopbound: 40
loop:<1..40>:#D = 1
loop:[1..5]:#G = 2
loop:[1..40]:#B→exit + #E→exit = 0
```

Calc 2: *loop:1..40 to out-edge*
```
loopbound: 40
loop:<1..40>:#D = 1
loop:[1..5]:#G = 2
loop:[1..40]:#G→exit + #D→exit = 0
```

Calc 3: *loop:41..50 to out-edge*
```
loopbound: 10
loop:[41..50]:#G→exit + #D→exit=0
```

(i) Scope graph and calculations of scope loop

Calc 4: *outer:1..10 to first out-edge*
```
loopbound: 10
outer:[]:#J ≤ 55
inner:<1..5>:#L = 1
outer:[]:#O→exit = 0
```

Calc 5: *outer:1..10 to second out-edge*
```
loopbound: 10
outer:[]:#J ≤ 55
inner:<1..5>:#L = 1
outer:[]:#I→exit = 0
```

(j) Scope graph and calculations of scope outer

Calc 6:*main:1..1 to out-edge*
```
loopbound: 1
```

(k) Scope graph and calcs of scope main

Figure 11: Complete example of clustered calculation

8. COMPLETE EXAMPLE

In Figure 11(a)-(k) we give a compact illustration of the steps involved in our clustered calculation method. To simplify the presentation, no timing for entities is included in the example.

Figure 11(a) shows an example control-flow graph consisting of a single loop and a loop nest consisting of two loops. Figure 11(b) shows the corresponding scope graph with scopes `main`, `loop`, `outer` and `inner`. Each scope has a loop bound and some have flow facts attached. Note that both `loop` and `outer` have multiple out-edges. Figure 11(c) shows the corresponding scope-hierarchy, with flow facts attached to the different scopes. Figure 11(d) shows the defining scope, defining scope span, and cover of the flow facts.

Figure 11(e) shows the fact clusters generated when applying the minimal clustering algorithm given in Figure 7. Since fact `f1` and `f2` overlap in their ranges they will be put in the same cluster. Fact `f3` refers to the header node of scope `inner` and is therefore put in the same cluster as `f4`[3].

Figure 11(f) shows the fact clusters generated when applying the split-foreach-fact minimal clustering (see Section 5.2). Fact `f1` has been split into two new facts `loop:<1..5>:#D` $=$ 1 (`f1'`) and `loop:<6..40>:#D` $=$ 1 (`f1''`). The fact clusters {`f1',f2`} and {`f1''`} together span the same range as the {`f1,f2`} cluster given in Figure 11(e).

Figure 11(g) shows the fact cluster generated when applying the global clustering (see Section 5.2). All facts are put into one cluster, with `main` as defining scope, and will all be considered together as a unit in the final calculation.

For the rest of the example we use the clusters in Figure 11(e) as generated by the minimal clustering. Figure 11(h) shows the resulting set of clusters, after adding empty clusters for all ranges of scopes not covered by any cluster. This is done as part of the algorithm given in Figure 8. An empty cluster for range 1..1 of `main` and one empty cluster for range 41..50 of `loop` is created.

Our demand-driven WCET calculation algorithm given in Figure 8 starts at scope `main`. Since only `main` is covered by the empty fact cluster, recursive calls are made for scope `loop` and `outer`, before calculating the WCET of `main`.

Scope `loop` is covered by two fact clusters, {`f1,f2`} and an empty cluster. The calculation starts with cluster {`f1,f2`}, since it spans the first iteration of `loop`. The scopes covered by the cluster are extracted to form a separate graph fragment as given in Figure 11(i). Two different calculations are made: one ending at the back-edge of `loop` (Calc 1) and one ending at the out-edge of `loop` (Calc 2). The same scope graph is used for both calculations, but some extra flow facts are added in each calculation to constrain where the execution should end.

[3] Fact `f4` is also constituting a fact cluster on its own, {`f4`}, defined on scope `inner`, but this is not included in Figure 11(e), since `f4` will always be calculated together with `f3`.

Program	Properties	BB	Sc	FF
compress	Nested loops, goto-loop, function calls.	91	24	9
crc	Complex loops, lots of decisions, loop bounds depend on function arguments.	29	9	6
duff	Loop with multiple entry points	20	6	2
expint	Inner loop that only runs once, structural WCET estimate gives heavy overestimate.	24	7	4
fibcall	Parameter-dependent function, single-nested loop.	7	4	0
fir	Inner loop with varying number of iterations, loop-iteration dependent decisions.	14	5	7
insertsort	Input-data dependent nested loop with worst-case of $n^2/2$ iterations.	7	4	1
lcdnum	Loop with iteration-dependent flow.	26	4	2
matmult	Multiple calls to the same function, nested function calls, triple-nested loops.	27	16	0
ns	Return from the middle of a loop nest, deep loop nesting.	18	7	1
nsichneu	Automatically generated code containing massive amounts of if-statements ($\gg 250$)	754	3	129

Figure 12: Benchmark programs

The calculation continues with the empty cluster spanning range 41...50 of scope loop. When calculating a WCET estimate for this cluster we reuse the extracted scope graph for scope loop. Since the cluster is empty, no flow facts are included, except one specifying that the execution must end at the out-edge (Calc 3). The three different WCET estimates extracted are used together, as given by the algorithm in Figure 8, to calculate a WCET estimate for scope loop.

The next step is to calculate a WCET estimate for scope outer. Since the fact cluster {f3,f4} covers both scope outer and inner, a WCET estimate will be extracted for both scopes together. A new scope graph is extracted for the two scopes, as shown in Figure 11(j). For the extracted scope graph and fact cluster, two different calculations are made, one to the out-edge with node I as source (Calc 4), and one to the out-edge with node O as source (Calc 5).

After calculating WCET estimates for scope loop and outer a WCET estimate for scope main can be calculated. A scope graph for scope main is extracted, as shown in Figure 11(k), with the calls to scope loop and outer replaced with call nodes. Each node is given a timing equal to the WCET estimate extracted for the call to the corresponding scope. Note that scope outer gets replaced with two different call nodes, since it had multiple out-edges. No fact is covering scope main, and only one calculation needs to be made for this scope (Calc 6). The result of the calculation is a WCET estimate for the whole program.

Note that we do not put any demands on the calculation method to use when calculating a WCET estimate for a fact cluster and its covered scopes. For example, for the calculations of the main scope or the last range of loop, both our Path-based [9, 30] and extended IPET [8, 9] methods can be used. For fact clusters with more complicated flow information, such as {f3,f4}, our extended IPET calculation method is preferably used.

9. MEASUREMENTS AND EVALUATION

In order to demonstrate the precision and effectiveness of our clustered calculation method we have performed a number of measurements, using the programs listed in Figure 12. We have tried to find a number of test programs containing various types program structures and of varying size, in order to test the calculation method thoroughly. The **BB** column gives the number of basic blocks when compiled for the V850E processor, **Sc** gives the number of scopes and **FF** the number of flow facts in the corresponding scope graph. All flow facts where manually added.

Our WCET tool supports three different calculation methods: a Path-based [9, 30], an Extended IPET [8, 9] and a clustered method (as outlined in this article). Each calculation module takes the same two inputs: a scope graph with flow facts (representing possible program flows) and a timing model (representing hardware timing).

Our Path-based method is very fast, only exploring a few of the total number of possible program paths, but can only handle flow facts of foreach type and with a cover of a single scope[4]. The Extended IPET calculation method allows for more complex flow information than classical IPET. It works by converting the whole program into one large constraint system and can handle all type of flow facts. The current implementation of the clustered method does not support the possibility to perform path-based calculation within clusters, i.e. only our extended IPET can be used within clusters, but is planned for future work. The extended IPET and clustered calculation method both rely on the mixed ILP solver lp_solve [2] to solve generated constraint systems.

All measurements were performed on a AMD Athlon 1800+ with 512 MB RAM. Since the V850E processor used does not have a cache, cache analysis was not included in the experiments.

The table in Figure 13 shows the WCET estimate precision (**cycles**) in clock cycles and computation time (**time**) in seconds, of the different calculation methods. The **Actual WCET** column gives the actual WCET of the program obtained by running a worst-case trace of the program through the same CPU simulator used by the WCET analysis. The Path-based method does not work with the duff benchmark, since it contains an unstructured loop.

The path-based WCET estimate precision is of the same quality as the clustered and extended IPET for most programs, indicating that foreach facts with a single scope cover are often sufficient for obtaining precise WCET estimates. However, programs like insertsort and fir need extra flow facts covering *several scopes* for high WCET estimate precision. This indicates that scope-local methods are not always sufficient to achieve high precision. The precision of the clustered method is of the same quality as the extended IPET, the current method with highest precision.

For all our benchmarks, except nsichneu, the time spent in the calculation stages is almost negligible. This is because most of the benchmarks programs given in Figure 12 are quite small and do not really stress our calculation methods.

To evaluate how the different calculation methods scale with added flow facts and the program size, we used an altered version of the nischneu benchmark. The original scope graph generated for nsichneu consists of three scopes (see Figure 12). The innermost scope is very large, containing 752 scope nodes. By adding extra dummy flow facts (i.e. facts that do not reflect the real program execution, but increase the complexity of the resulting constraint system), spanning a particular iteration of the inner scope and not actually removing any possible execution paths, we increase

[4] Path-based methods can be extended to handle triangular loop dependencies and unstructured code [14], this is however not implemented in our path-based method.

Program	Path-based			Ext. IPET			Clustered			Actual
	cycles	+%	time	cycles	+%	time	cycles	+%	time	WCET
compress	8670	+0.09	0.01	8670	+0.09	0.25	8670	+0.09	0.29	8662
crc	30275	+0.01	0.01	30271	0	0.02	30271	0	0.04	30271
duff	-	-	-	1083	0	0.01	1083	0	0.04	1083
expint	8588	0	0.01	8588	0	0.01	8588	0	0.04	8588
fibcall	313	0	0.01	313	0	0.01	313	0	0.01	313
fir	352073	+1.14	0.01	348095	0	0.02	348095	0	0.07	348095
insertsort	1794	+67.04	0.01	1074	0	0.01	1074	0	0.01	1074
lcdnum	198	0	0.01	198	0	0.03	198	0	0.13	198
matmult	221824	0	0.01	221824	0	0.01	221824	0	0.04	221824
ns	23746	+70.49	0.01	17353	+24.59	0.01	17353	+24.59	0.03	13928
nsichneu	51133	+0.03	0.09	51116	0	1.78	51116	0	2.15	51116

Figure 13: WCET estimate precision and calculation time

Extra	Path-based			Ext. IPET				Clustered		
facts	time	expl.	paths	time	lptime	constr.	vars.	time	lptime	calcs
0	0.09	3	3.73E97	1.78	0.66	1651	2139	2.15	0.72	2
1	0.2	5	1.11E98	14.45	5.02	3417	4528	8.64	1.85	4
2	0.31	7	1.87E98	32.84	11.71	4931	6665	13.28	3.06	6
3	0.46	9	2.61E98	58.38	21.80	6445	8802	19.98	4.24	8
4	0.58	11	3.36E98	92.43	35.32	7959	10939	27.04	5.48	10
5	0.74	13	4.11E98	135.90	51.94	9473	13076	36.13	6.73	12
6	0.92	15	4.85E98	187.86	71.35	10987	15213	49.33	7.99	14
7	1.05	17	5.59E98	248.87	95.06	12501	17350	54.35	9.14	16
8	1.19	19	6.34E98	319.51	120.81	14015	19487	59.44	10.34	18
9	1.32	21	7.09E98	390.73	149.54	15529	21624	66.80	11.52	20
10	1.34	23	7.83E98	476.26	182.84	17043	23761	78.20	12.82	22

Figure 14: Scaling measures of calculation methods

the computational load. For example, adding one dummy fact will create an extended scope graph for our extended IPET method consisting of 1508 scope nodes (752 + 752 + 4). The IPET method will create a constraint system over the whole graph while the clustered and path-based calculation methods will partition the problem into smaller subproblems. For each calculation run all WCET estimates achieved were exactly the same as reported in Figure 13.

Figure 14 gives computation times obtained for our calculation methods when adding dummy flow facts. The **Extra facts** column gives the number of added dummy facts. For each calculation method we give some values of interest for understanding its particular execution time properties. For the path-based calculation the computation time (**time**), the number of explored paths (**expl.**) and the number of potential paths (**paths**) are given. For the Extended IPET calculation, the computation time (**time**), the time spent in the linear programming solver lp_solve (**lptime**), and the number of constraints (**constr.**) and variables (**vars.**) generated are given. For the clustered calculation the computation time (**time**), the number of lp_solve calls made and the total time spent in lp_solve (**lptime**) are given. The computation time for Extended IPET and Clustered calculation includes the time spent in lp_solve.

Figure 15 shows computation times of each calculation method plotted against the number of added dummy flow facts. We note that the computation time seems to be linearly increasing with the problem size both for the path-based and clustered calculation, while the extended IPET has a more than linear increase. Both the extended IPET and the clustered calculation method spend most of the calculation time in constructing graphs and generating constraint systems.

The graph also plots the time spent in lp_solve for the extended IPET and clustered calculation. For the extended IPET a single call to lp_solve is made for each calculation, with constraints and variables for the complete virtual scope graph. For the clustered calculation, the number of lp_solve calls increases with the number of added dummy facts, but not the size of each generated constraint system. Each call to lp_solve by the clustered calculation of the innermost scope contained 2156 variables and 1660 constraints and took approximately 0.65 seconds.

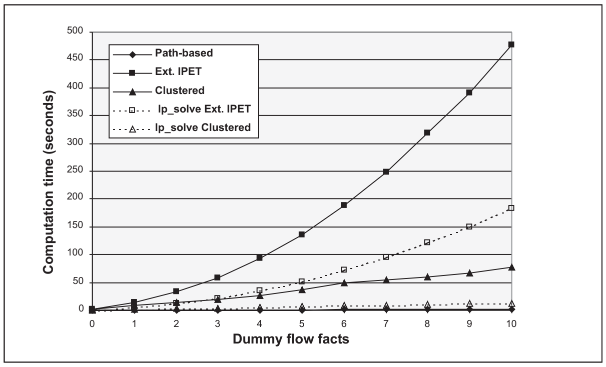


Figure 15: Computation time scaling

We conclude that extended IPET has quite bad scaling properties. This could be a general problem for calculation methods relying on global ILP solvers for calculating WCET estimates. Our path-based calculation method is very efficient, only exploring a few of the total number of possible paths, and seems to scale very well. The clustered calculation is somewhere in between in complexity, scaling reasonably well, while still being able to handle complex flow information.

We have implemented all five clustering algorithms outlined in Section 5.2. The algorithms differ in how many flow facts will be considered together, and consequently in the size of the scope graph that will be covered by each local

Program	min-split			minimum			scope			max			global		
	cl	call	time	cl	call	time	cl	call	time	cl	call	time	cl	call	time
compress	6	34	0.29	4	34	0.28	4	30	0.3	1	23	0.34	1	1	0.28
crc	6	6	0.04	6	6	0.01	6	6	0.04	1	1	0.01	1	1	0.02
duff	2	12	0.04	2	12	0.04	2	12	0.03	1	4	0.02	1	1	0.01
expint	4	11	0.04	4	11	0.04	2	7	0.03	1	6	0.03	1	1	0.01
fibcall	0	4	0.01	0	4	0.01	0	4	0.01	0	4	0.01	0	1	0.01
fir	5	12	0.05	2	6	0.04	2	3	0.03	1	1	0.02	1	1	0.02
insertsort	1	1	0.01	1	1	0.01	1	1	0.01	1	1	0.01	1	1	0.01
lcdnum	2	23	0.13	2	23	0.12	1	19	0.11	1	21	0.12	1	1	0.03
matmult	0	15	0.04	0	15	0.04	0	15	0.05	0	15	0.04	0	1	0.01
ns	1	13	0.03	1	13	0.03	1	10	0.02	1	13	0.03	1	1	0.01
nsichneu	1	2	2.18	1	2	2.17	1	2	2.06	1	2	2.17	1	1	1.72

Figure 16: Clustered calculation measures

WCET calculation performed. Figure 16 shows the effect of applying different fact cluster algorithms to our benchmarks. Columns labelled **cl** give the number of fact clusters generated (not including empty clusters). Columns labelled **call** give the number of local WCET calculations performed, i.e. the number of calls to `lp_solve`, and **time** gives the computation time of the calculation.

The minimum (**minimum**) and split-foreach-fact (**min-split**) fact clustering algorithms generate many small clusters, and result in many local WCET calculations. At the other extreme we have the global clustering (**global**) which performs one single WCET calculation for the whole program or maximum clustering (**max**) which puts all flow facts into one cluster but does not include non-covered scopes. Scopes not covered by any fact clusters are traversed bottom-up generating one or more local WCET calculations, explaining the different number of WCET calculation calls made for different benchmarks. For all benchmarks all clustering algorithms gave the same WCET estimates as presented for the clustered calculation in Figure 13.

As discussed in Section 7, some fact clusters define graph fragments with several entry and exit points, allowing us to trade WCET estimate precision for speed. Figure 17 presents measurements performed using the minimal clustering algorithm. The **diff in-out** measurements differentiate between entry and exit points, while the **no diff** measurements do not. The amount of fact clusters generated is identical for both algorithms (**cl**). The **calls** column gives the number of local WCET estimates performed for each program. We note that for many programs, the number of local WCET estimates decreases quite significantly when not differentiating between entry and exit points. For all programs, except `ns`, the calculated WCET estimates precision is of the same quality. Program `ns` contains a non-local return from a deeply nested loop, causing an overestimation in a fashion similar to the example presented in Section 7.

10. CONCLUSIONS AND FUTURE WORK

In this paper we have presented a new method for calculating the WCET of a program. The method can be considered a hybrid between fast but less precise calculation methods like tree-based and path-based methods, and the precise but potentially slow global IPET method. It is based on finding the smallest possible parts of a program that have to be handled as a unit to ensure precision. The calculation method to use for each such part is not fixed but could depend on the characteristics of the given flow information and program structure. Since these parts are typically small compared to the overall program, the method is fast, but no precision is lost from introducing arbitrary boundaries in the calculation as is done in tree-based and path-based approaches.

Program	diff in-out			no diff		
	cl	call	WCET	cl	call	WCET
compress	6	38	8670	6	31	8670
crc	6	6	30271	6	6	30271
duff	2	12	1083	2	5	1083
expint	4	11	8588	4	10	8588
fibcall	0	4	313	0	3	313
fir	5	12	348095	5	12	348095
insertsort	1	1	1074	1	1	1074
lcdnum	2	23	198	2	7	198
matmult	0	15	221824	0	15	221824
ns	1	13	17353	1	8	23746
nsichneu	1	2	51116	1	2	51116

Figure 17: Effect of differing between flows in and out of clusters

Our experiments indicate that the clustered calculation achieves the same precision as the global extended IPET, while being much less prone to high analysis times. In general, the suitability of a particular calculation method depends on the structure of the program and the properties of the provided flow information. We have outlined several different alternatives to perform clustered calculation, making it easy to adapt the calculation to particular requirements of computation speed and precision.

We are currently working on fully integrating an automatic flow analysis module [12] into our WCET analysis tool. Preliminary results indicate that such analyses are likely to produce a large number of flow facts, while a human user usually only provides a handful of facts for a typical program. In this scenario we believe that the clustered calculation method will become important to keep the calculation time down.

11. REFERENCES

[1] P. Atanassov, R. Kirner, and P. Puschner. Using Real Hardware to Create an Accurate Timing Model for Execution-Time Analysis. In *IEEE Real-Time Embedded Systems Workshop, held in conjunction with RTSS2001*, Dec 2001.

[2] M. Berkelaar. *lp_solve: (Mixed Integer) Linear Programming Problem Solver*, 2003.
URL: `ftp://ftp.es.ele.tue.nl/pub/lp_solve`.

[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 G. Bernat. Scope-tree: a program representation for symbolic worst-case execution time

analysis. In *Proc. 14th Euromicro Conference of Real-Time Systems, (ECRTS'02)*, pages 50–59, 2002.

[5] A. Colin and I. Puaut. Worst Case Execution Time Analysis for a Processor with Branch Prediction. *Journal of Real-Time Systems*, 18(2/3):249–274, May 2000.

[6] J. Engblom. *Processor Pipelines and Static Worst-Case Execution Time Analysis*. PhD thesis, Uppsala University, Dept. of Information Technology, Box 337, Uppsala, Sweden, Apr 2002.

[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, Dec 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)*, Nov 2000.

[9] A. Ermedahl. *A Modular Tool Architecture for Worst-Case Execution Time Analysis*. PhD thesis, Uppsala University, Dept. of Information Technology, Box 325, Uppsala, Sweden, Jun 2003. ISBN 91-554-5671-5.

[10] C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, and R. Wilhelm. Reliable and Precise WCET Determination for a Real-Life Processor. In *Proc. 1st International Workshop on Embedded Systems, (EMSOFT2000), LNCS 2211*, Oct 2001.

[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. Gustafsson, B. Lisper, C. Sandberg, and N. Bermudo. A Tool for Automatic Flow Analysis of C-programs for WCET Calculation. In *8th IEEE International Workshop on Object-oriented Real-time Dependable Systems (WORDS 2003), Guadalajara, Mexico*, Jan 2003.

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

[14] C. Healy, M. Sjödin, V. Rustagi, and D. Whalley. Bounding Loop Iterations for Timing Analysis. In *Proc. 4th IEEE Real-Time Technology and Applications Symposium (RTAS'98)*, Jun 1998.

[15] R. Heckmann, M. Langenbach, S. Thesing, and R. Wilhelm. The Influence of Processor Architecture on the Design and the Results of WCET Tools. *IEEE Proceedings on Real-Time Systems*, 2003.

[16] N. Holsti, T. Långbacka, and S. Saarinen. Worst-Case Execution-Time Analysis for Digital Signal Processors. In *Proceedings of the EUSIPCO 2000 Conference (X European Signal Processing Conference)*, Sep 2000.

[17] S.-K. Kim, S. L. Min, and R. Ha. Efficient Worst Case Timing Analysis of Data Caching. In *Proc. 2nd IEEE Real-Time Technology and Applications Symposium (RTAS'96)*, pages 230–240. IEEE, 1996.

[18] R. Kirner and P. Puschner. Transformation of Path Information for WCET Analysis during Compilation. In *Proc. 13th Euromicro Conference of Real-Time Systems, (ECRTS'01)*. IEEE Computer Society Press, Jun 2001.

[19] 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.

[20] 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, Jul 1995.

[21] 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)*, Dec 1998.

[22] T. Lundqvist and P. Stenström. An integrated path and timing analysis method based on cycle-level symbolic execution. *Journal of Real-Time Systems*, May 2000.

[23] T. Mitra and A. Roychoudhury. Effects of Branch Prediction on Worst Case Execution Time of Programs. Technical Report 11-01, National University of Singapore (NUS), Nov 2001.

[24] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997. ISBN: 1-55860-320-4.

[25] 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)*, Jun 1997.

[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)*, Dec 1999.

[27] P. Puschner and A. Schedl. Computing Maximum Task Execution Times with Linear Programming Techniques. Technical report, Technische Universität Wien, Institut für Technische Informatik, Apr 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] F. Stappert, A. Ermedahl, and J. Engblom. Efficient Longest Executable Path Search for Programs with Complex Flows and Pipeline Effects. In *Proc. 4th International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, (CASES'01)*, Nov 2001.

[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, Jun 1997.