Clustered Worst-Case Execution-Time Calculation
– Tech Report Edition –

Andreas Ermedahl, Friedhelm Stappert, and Jakob Engblom

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.

Index Terms: WCET analysis, WCET calculation, timing analysis, hard real-time, embedded systems.

I. 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 such as 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 [1]–[3]. 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, such as loop iterations and function calls, and the effects of hardware features, such as caches and pipelines. The flow information can be considered as a set of *flow facts*, each giving constraints on the program flow for a certain piece of the program (loop bounds, infeasible paths, execution dependencies, etc.). Flow facts are usually local in their nature, expressing information that only affects a smaller region of a program, such as a single loop or an if-statement. However, sometimes these regions might reach over the basic program structures. For example, a loop nest rather than a loop, or an entire function rather than just a loop inside that function.

In general, the expressiveness of the flow facts which can be handled by a calculation method are to a 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.

Previous WCET calculations have been either *local* or *global* in nature. Local calculation schemes work by only considering a *fixed* granularity of a program at once, such as a single loop or a function. Local calculation schemes are usually performed in a *bottom-up* fashion, calculating a safe (timing) abstraction for a program part, which is later used in the calculation of surrounding parts of the program. Bottom-up calculations are beneficial, since the overall WCET calculation problem can be subdivided into smaller, easier-to-solve problems, thereby achieving high computational efficiency. The drawback of traditional local calculation schemes is that they cannot handle all types of flow facts. Basically, when flow facts reach over the fixed calculation boundaries, they cannot be accounted for, which leads to lower WCET estimate precision.

Global calculation schemes handle this problem by working globally on the entire program at once. This allows most type of flow facts to be handled, regardless of which program region they affect. However, most 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. This makes scaling to large programs risky.

Our *clustered calculation method* in effect achieves the precision of global WCET calculations, while getting close to the efficiency of local calculations. The key idea is to work with a *dynamic* granularity in the WCET calculation: determining the units of work in the calculation based on the actual flow facts present. This is unlike traditional local calculations where the units are statically defined based on the program structure, regardless of the flow information present.

Furthermore, in many cases it is not sufficient to consider each flow fact in isolation. Several flow facts are likely to be present for a program, and these flow facts may *interact* and together constrain the program flow further than each individual flow fact. A WCET calculation method must find such interacting flow facts and treat them as a unit or risk losing precision. The clustered calculation method takes this into account, drawing the boundaries in the calculation in such a way that all flow fact dependencies are indeed accounted for.

The clustered calculation method works as follows: the provided flow facts are used to construct units where all included flow facts are directly or indirectly dependent. For each such *fact cluster* the program region affected by the included flow facts is extracted. The fact clusters and corresponding program regions are used to calculate a program WCET estimate in a bottom-up manner. The method is also *demand-driven* in that a WCET for a program region is only calculated when its timing estimate is needed in a surrounding program part. 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 region.

The boundaries between calculation regions present in clustered and local calculation schemes also interact with the *hardware timing* of the target hardware. Timing dependencies can reach across calculation boundaries, and such cases need to be handled safely in order to generate a correct WCET estimate.

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.
- We present a timing model able to represent various type of hardware timing dependencies.
- We evaluate the effect of long reaching timing dependencies on various calculation schemes.

The rest of this paper is organized as follows: Section II gives an introduction to WCET analysis and previous work. Section III presents our flow representation, Section IV presents how flow facts can be organized into fact clusters, and Sections V-VI give the clustered WCET calculation method. Section VII presents our timing model and Section VIII shows how we handle timing dependencies reaching over calculation borders. Section IX presents our WCET tool architecture, including our pipeline timing analysis (Section IX-A), our efficient path-based calculation method (Section IX-B), and our Extended IPET calculation method (Section IX-C). Finally, Section X presents our experimental evaluation, and Section XI gives our conclusions and ideas for future work.

II. WCET Analysis Overview and Previous Work

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 which 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 ob-

tained by *manual annotations* (integrated in the programming language [4] or provided separately [5]–[7]), or by *automatic flow analysis* methods [8]–[12]. The flow analysis is traditionally called high-level analysis, since it is often done on the source code, but it can also be done on intermediate or machine code level.

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, such as 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 [7], [9], [12]–[14], data caches [12], [15], [16], and branch predictors [17], [18] have been analyzed. Local low-level analysis has dealt with scalar pipelines [9], [11], [12], [14], [17], [19]–[21] and superscalar CPUs [22], [23]. Heckmann et al. [13] 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 [24].

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 calculation, the WCET is calculated in a bottom-up traversal of a tree generally corresponding to a syntactical parse tree of the program [14], [17], [25]. The syntax-tree is a representation of the program whose nodes describe the structure of the program (e.g. sequences, loops or conditionals) and whose leaves represent basic blocks. Rules are given for traversing the tree, translating each node in the tree into an equation that expresses its timing based on the timing of its child nodes. The method is conceptually simple and computationally cheap, but has problems handling long reaching dependencies, since the computations are local within a single program statement.

Figure 1(a) shows an example control-flow graph with timing on the nodes and a loop-bound flow fact. Figure 1(b) illustrates how a tree-based calculation method would proceed over the graph according to the program syntax-tree and given transformation rules. Collections of nodes are collapsed into single nodes, simultaneously deriving a timing for the new node. Since the processing order is pre-defined flow information between non-related program parts, e.g. between C and F, are hard to handle. Similarly, hardware dependencies between non-local parts of the code are difficult to handle, and must be treated in a pessimistic fashion to guarantee the safeness of the analysis.

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 [9], [12], [26]. The defining feature is that possible execution paths are represented *explicitly*. The path-based approach is natural within a single loop iteration, but has problems with flow information stretching across loop-nesting levels.

Figure 1(c) illustrates how a path-based calculation method would proceed over the graph in Figure 1(a). The loop in the graph is first identified and the longest path within the loop is found. The time for the longest path is combined with the loop bound flow fact to extract a WCET estimate for the whole program.

In IPET, program flow and low-level execution time are modeled using arithmetic constraints [5]–[7], [10], [27]. Each basic block and program flow edge in the program is given a time variable (t_{entity}), holding to the contribution of that entity to the total execution time every time it is executed, and a count variable (x_{entity}), corresponding to the number of times the entity is executed. The WCET is extracted by maximizing the sum of products of the execution counts and times ($\sum_{i \in entities} x_i * t_i$), where the execution count variables are subject to constraints reflecting the structure of the program and possible flows. The result of an IPET calculation is a WCET estimate and a worst-case count for each execution count variable.

Figure 1(d) shows the constraints and WCET formula generated by a IPET-based calculation method for the program illustrated in Figure 1(a). The *start* and *exit constraints* gives that the program must be started and exited once. The
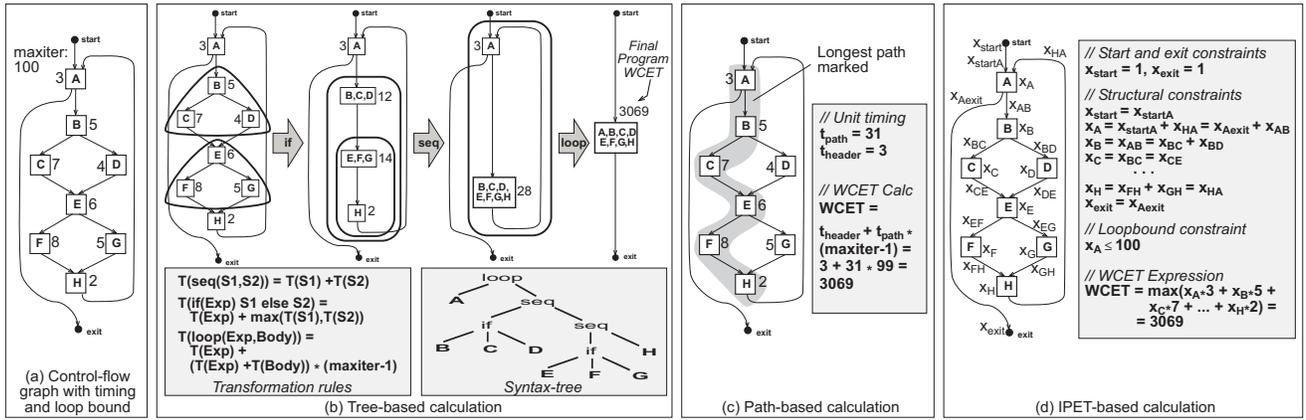
Fig. 1. Different calculation methods

structural constraints reflects the possible program flow, meaning that for a basic block to be executed it must be entered the same number of times as it is exited. The *loop bound* is specified as a constraint on the number of times the loop header node A can be executed.

IPET is able to handle all types of flow information, including flow facts with long reaching dependencies. IPET has traditionally been applied in a global fashion, treating the whole program and all flow information together as a unit. IPET calculations normally rely on integer linear programming (ILP) or constraint programming (CP) techniques, thus having a complexity potentially exponential in the program size. Also, since flow facts are converted to constraints the size of the resulting constraint system grows with the number of flow facts.

We have previously developed an efficient Path-based calculation method (Section IX-B) [26], [28], [29], and an Extended IPET calculation method (Section IX-C) [5], [28]. Both methods are able to handle more types of complex flow facts and timing dependencies than traditional path-based and IPET calculation schemes. The clustered calculation method presented in this article combines the precision of the Extended IPET calculation with the efficiency of the Path-based calculation.

In this article we focus on the calculation phase which uses the results of the other phases (i.e. high-level flow information from the flow analysis phase and detailed timing information from the low-level analysis phase) as input.

III. REPRESENTING PROGRAM FLOW

In order to perform any kind of WCET calculation, we need to be able to represent the flow of a program. The fundamental data structure in our approach is the *scope graph*. The scope graph is based on the partitioning of the instructions in the object code of a program into *basic blocks* [30]. Figure 2(a) shows an example C function, Figure 2(b) show the corresponding assembler code, and Figure 2(c) show the corresponding control flow graph and basic blocks.

The scope graph consists of nodes (including distinguished start and exit nodes) and edges. Each node (except the start and exit node) holds a reference to a basic block in the object code. The flow information representation in the scope graph thus refers to the compiled executable object-code of the program. This is necessary in order to be able to tie the flow information to the hardware timing information of a program, which is by necessity generated on the object-code level.

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 basic block graph in Figure 2(c).

Each scope has a distinguished header node, (e.g. node A resp. C in Figure 2(d)). 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
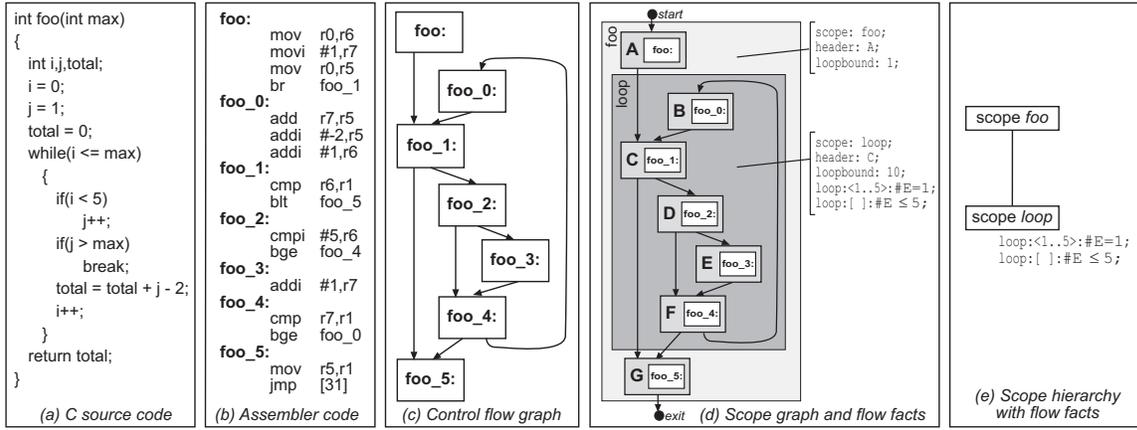
Fig. 2. Scope graph with flow facts and scope-hierarchy

all its children. Figure 2(e) illustrates 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 leaf scope and both a descendant and a child to scope `foo`.

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. 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. For example, in Figure 2(d) A→C is an in-edge, C is an in-node, F→G an out-edge and B→C a back-edge of scope `loop`.

A. Flow facts

To express more complex program flow information than just basic loop bounds, each scope can carry a set of *flow facts* [5], [28]. 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. The latter property makes it possible to use flow facts in local calculations schemes. Furthermore, using flow facts on the scope graph, we are able to represent most types on interesting flows, as discussed in more details in Section IV-A below.

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 of the defining scope for which the constraint expression is valid. This can be for all or 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 *span* of a fact is the iterations of the defining scope for which a fact is valid. Two facts are said to *overlap* if their spans have any iteration in common.

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

```
void foo(bool x){      // scope: m
 ...
 for(i=0;i<10;i++)     // scope: n, loop bound: 10
   for(j=i;j<10;j++)   // scope: o, loop bound: 10
     {... O1; ...}     // code including block O1
 ...
 if(...)
   x=true;             // block M1
 bar(x);
 ...
 for(...)              // scope: r
   {...
    baz();             // code including
    ... }             // blocks R1, R2, R3
 }
```

```
void bar(bool x){   // scope: p
  for(...)           // scope: q
    if(x==true)
      Q1;            // block Q1, execution
  ...                // implied by M1
}
```

```
void baz() {    // scope: s
  ...            // code including blocks S1, S2, S3
  for(...)       // scope: t, loop bound: 10
    if(T2){      // block T2, false during last 3 iters
      S4;        // block S4, big chunk of work
      break; }   // code including block S5
  ...
}
```

Triangular loop

Long reaching dependency

Conditional dependency

Fig. 3.   Example code

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, in Figure 2 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.

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. The two flow facts overlap since they span the first five and all iterations of `loop` respectively.

IV. 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. Two flow facts can interact by giving constraints for the same iterations of a scope, i.e. by overlapping. Interacting flow facts do not need to refer to the same scope graph entity. For example, adding a flow fact which refers to node `F` in Figure 2(d) would indirectly constrain the execution of node `E` and could therefore interact with the two already given flow facts. However, if the new flow fact only were spanning the last three iterations of scope `loop`, it would not directly interact with the first flow fact since they do not overlap. A flow fact can also interact

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

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

(b) *Information about facts*

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

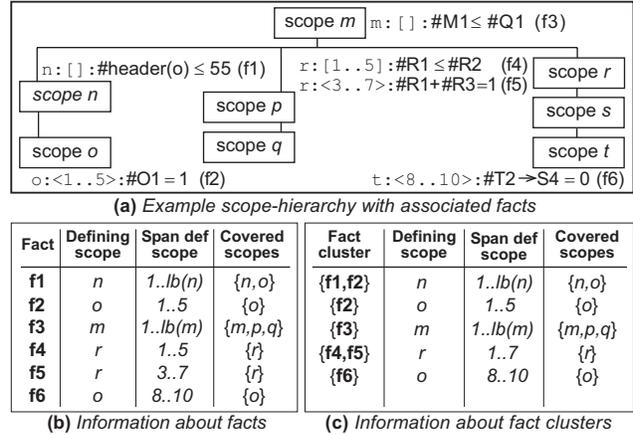**(c)** *Information about fact clusters*

Fig. 4.   Fact clustering example

by constraining executions of entities located in descendant scopes. For example, adding a fact defined on scope `foo` referring to entities in scope `loop` would potentially interact with the two already given facts.

We define a *fact cluster* to be a set of flow facts. The *defining scope* of a fact cluster is 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.

In Figure 3 some example code fragments are given, including three different functions `foo`, `bar` and `baz`, and some loops. In Figure 4(a) a scope-hierarchy generated from the code in Figure 3 is presented, including some flow facts. Figure 4(b) show the defining scopes, defining scope spans, and cover of each given flow fact. The name of a count variable gives the scope in which the corresponding entity is located, e.g. `#M1` refers to executions of node `M1` located in

scope m. The function lb(n) returns the loop bound for a scope n.

The fact clusters generated from the facts are given in Figure 4(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.

A. Flows causing clusters

Program flows causing fact clusters and reaching over several scopes are actually quite common. The simplest example is illustrated in the first loop nest in function foo in Figure 3. 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 4(a)).

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 calculation of the inner loop will assume 10 iterations. The WCET calculation for the outer loop will use 10 iterations of the inner loop for each entry of the inner loop. This gives that the inner loop body being counted 100 times, when it is actually cannot be executed more than 55 times. To solve this problem we need to handle the inner and outer loop together as a unit. Flow fact f1 in Figure 4(a) shows how this type of triangular loop dependency can be captured, (#header(o) refers to the count variable of the header node of scope o).

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), e.g. for the scopes m, p and q in Figure 4(a). The code in foo and bar in Figure 3 experience such a long reaching dependency. Flow fact f3 in Figure 4(a) captures this dependency. It gives that an execution of node M1 *implies* an execution of node Q1, (node Q1 can still be executed on its own).

In the next example, given in function baz in Figure 3, node S4 does not belong to the loop (scope t) due to the break statement. Therefore, the way the loop is exited will determine whether it should be counted or not. Thus, S4 depends on the decision T2 in the loop body, but S4 is a node in the parent scope of t (scope s). Fact f6 in

Figure 4 captures this dependency by specifying that the edge T2→S4 cannot be taken during the last three iterations of the t 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 4(a) with flow facts f4 and f5. Both flow facts have the same defining scope r and their spans have iterations in common.

B. Fact clustering algorithm

An algorithm to create the clusters of flow facts is given in Figure 5. The algorithm makes a post-order traversal of the scope graph, generating clusters in descendant scopes prior to the parents.

For each scope s, we look at the facts defined on the scope, and partition the facts based on their iteration span. Two facts that overlap, i.e. have some iterations in common, go to the same cluster: $\forall f_i, f_j \in$ facts(s) : (overlap(f_i, f_j) $\wedge$ $f_i \in c$) $\Rightarrow f_j \in c$. This creates sets of facts where each fact overlaps one or more of the facts in the same cluster. For example, facts f4 and f5 in Figure 4 have the same defining scope r and have iterations in common, and should therefore be put in the same cluster. Note that if there are any "all iterations" facts (using context specification [ ] or < >), there will only be one fact cluster for this scope since these facts include all iterations, and thus overlap with all other facts defined on the scope.

The algorithm also consider interactions of flow facts located in different scopes. For each extracted fact-cluster c we add fact cluster defined in descendant scopes covered by c. This creates larger fact clusters, each covering a set of scopes that have to be jointly considered. Note that this means that a fact can be part of several fact clusters. For example, fact f1 in Figure 4 covers both scope n and o and should therefore be clustered together with fact f2, resulting in the fact cluster {f1,f2} with n as its defining scope. Fact f2 also forms a fact cluster on its own with o as defining scope.

The algorithm given in Figure 5 generates minimal sets of facts where all included facts should be considered together. We call this clustering algorithm *minimal fact clustering*. The algorithm

```
ClusterFacts(scopegraph sg):
  FC := ∅  // 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 into set of overlapping facts
    // Add fact clusters already created in descendant scopes
    for each fact cluster c in C do
      S := scopes covered by c except scope s
      for each fact cluster c_sub in FC defined in S do
        c := c ∪ c_sub
      end for
    end for
    // Update set of fact clusters
    FC := FC ∪ C
  end for
  return FC
```

Fig. 5.   Minimal fact clustering algorithm

makes sure that all facts that might interact are put in the same cluster.

It is also possible to make more relaxed form of clusterings, i.e. merging some of the minimal fact clusters into larger clusters. These clusterings will put facts into the same cluster even though they do not really interact. Natural examples of such more relaxed 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 [28].

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 5 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 4(a) the total fact f4 does not overlap f5 completely, so we split f5 into the facts r:<3..5>:#R1 + #R3 = 1 (f5') and

r:<6..7>:#R1 + #R3 = 1 (f5"). The resulting fact clusters become {f4,f5'} and {f5"}. 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.

V. CLUSTERED WCET CALCULATION

The algorithm for calculating a WCET estimate using fact clusters is shown in Figure 6. 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.

A WCET 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

```
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, 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, 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
```
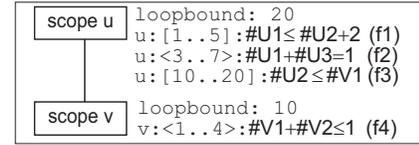
Fig. 6.   Clustered WCET calculation algorithm

(a) *Scope-hierarchy with flow facts*

(b) *WCET calculation scope v*

(c) *WCET calculation scope u*
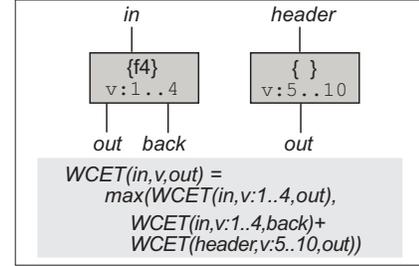
Fig. 7.   Calculation over clusters

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 cannot continue. If so, we stop iterating over the ranges and return the total time accumulated for the scope.
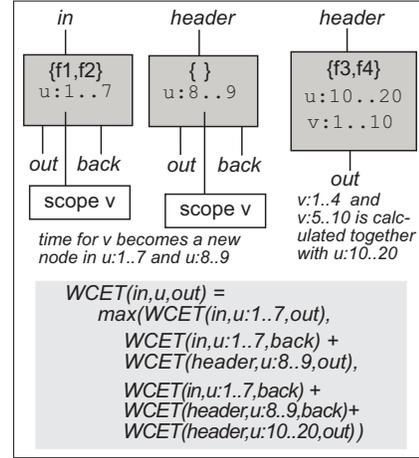
Figure 7 shows an example of a WCET calculation working over clusters. The algorithm starts at scope u where there are several facts covering the iteration range. The name of a referred count variable gives the scope in which the correspond-

ing 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 { } (covering just scope v) 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
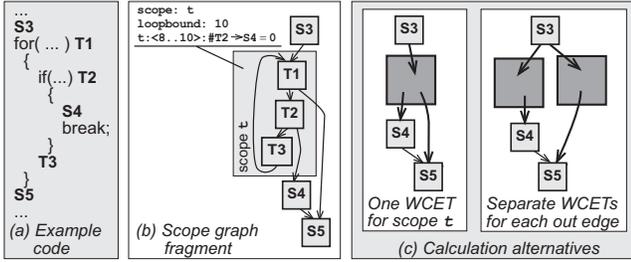
Fig. 8. Calculation options for fragment with multiple exits

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.

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 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.

A. Entry and Exit Options

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 8 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 function baz in Figure 3, where flow fact f6 specifies that edge T2→S4 cannot be taken during the last three iterations of scope t.

If only one calculation is made for scope t for both its out-edges it will result in a timing estimate for t which gives that the loop is iterated 10 full iterations. Later, when doing a WCET calculation for scope s the worst case path will be passing the call node for scope t together with the nodes S4 and S5. 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 t, will discover that the out-edge to node S4 cannot be taken during the last three iterations of t. The WCET estimate for scope t will therefore be different depending on the used out-edge. In the calculation of scope s, this will result in two separate call nodes for scope t, 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.

VI. COMPLETE EXAMPLE OF CLUSTERED CALCULATION

In Figure 9(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 9(a) shows an example control-flow graph consisting of a single loop and a loop nest consisting of two loops. Figure 9(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 9(c) shows the corresponding scope-hierarchy, with flow facts attached to the different scopes. Figure 9(d) shows the defining scope, defining scope span, and cover of the flow facts.

Figure 9(e) shows the fact clusters generated when applying the minimal clustering algorithm given in Figure 5. 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[1].

Figure 9(f) shows the fact clusters generated when applying the split-foreach-fact minimal clustering (see Section IV-B). Fact f1 has been split into two new facts loop:<1..5>:#D = 1 (f1') and loop:<6..40>:#D = 1 (f1'').

---

[1] Fact f4 is also constituting a fact cluster on its own, {f4}, defined on scope inner, but this is not included in Figure 9(e), since f4 will always be calculated together with f3.
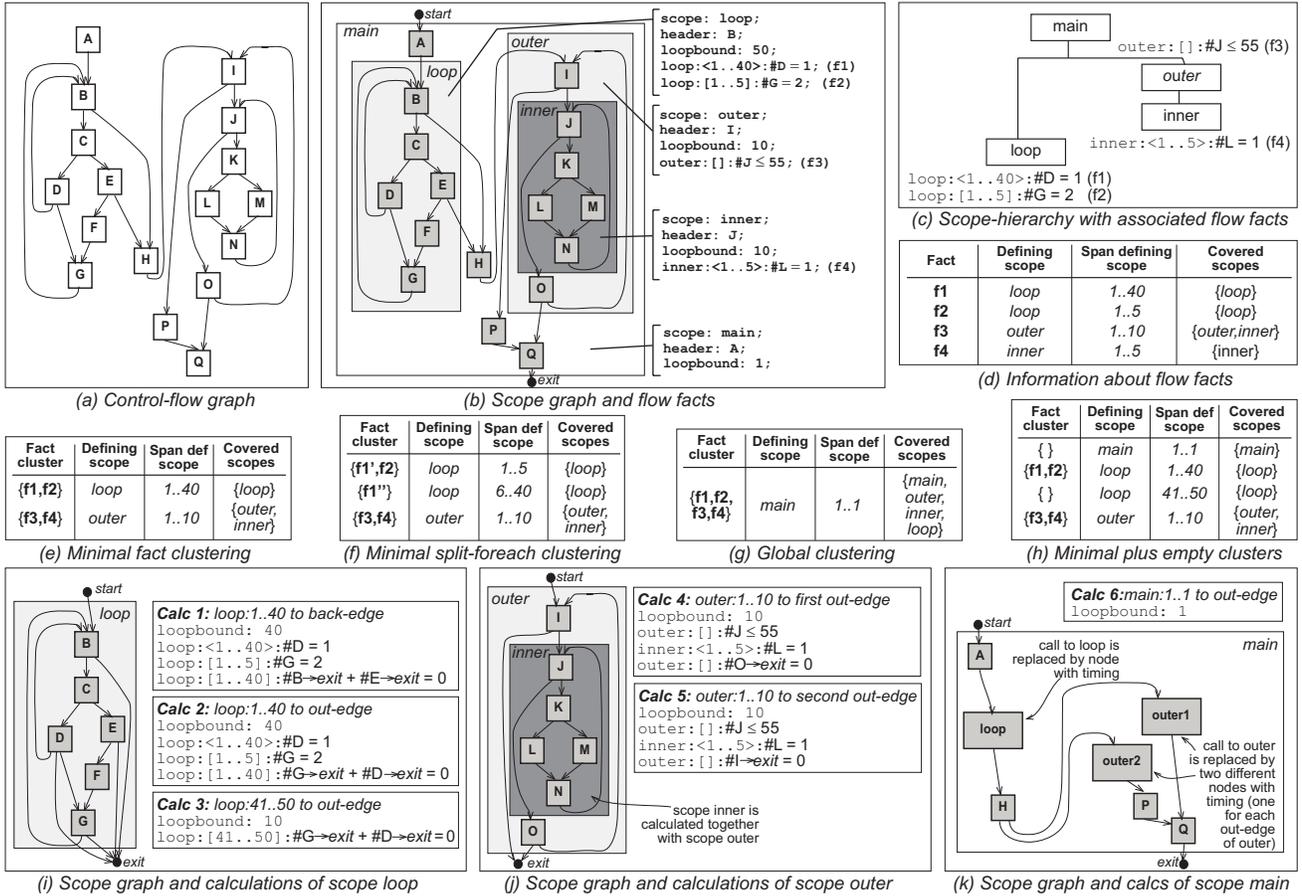
(a) Control-flow graph

(b) Scope graph and flow facts

(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

(i) Scope graph and calculations of scope loop

(j) Scope graph and calculations of scope outer

(k) Scope graph and calcs of scope main

Fig. 9.   Complete example of clustered calculation

The fact clusters {f1',f2} and {f1''} together span the same range as the {f1,f2} cluster given in Figure 9(e).

Figure 9(g) shows the fact cluster generated when applying the global clustering (see Section IV-B). 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 9(e) as generated by the minimal clustering. Figure 9(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 6. 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 6 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 9(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.

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 6, 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 9(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 9(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 and Extended IPET methods can be used (see Section IX). For fact clusters with more complicated flow information, such as $\{f3, f4\}$, our Extended IPET calculation method is preferably used.

VII. REPRESENTING PROGRAM TIMING

As we have defined the clusters in the clustered calculation based on the flow information, it is possible that there are hardware timing effects crossing the boundaries between calculation units. In order to discuss this problem, we need to have a defined *timing model* for the program.

In previous work, we have developed a timing model designed to support the calculation of a program WCET estimate from the timing of smaller program parts, while still capturing all relevant hardware timing effects [19], [20]. This timing model will be explained here and used in Section VIII to attack the problem of timing effects across calculation boundaries.

The timing model represents the execution time using *timing* for nodes, t_{node}, and *timing effects* for sequences of nodes, δ_{seq}. Each node corresponds to one or more instructions on the target, and the edges connecting them to the potential flows between the instructions. For this discussion, we can assume that the nodes in the timing model are identical to the nodes in

the scope graph, and thus refers to a certain basic block in the program executing in a certain context.

Intuitively, a node time t_{node} represents the *time* it takes to execute a node in isolation on the hardware. A timing effect δ_{seq} represents the *change* in execution time encountered when a node sequence *seq* is executed compared to the total time of the constituent nodes (and the timing effects of the subsequences of the sequence). Timing effects are negative to indicate a speedup over a sequence of nodes, and positive to indicate a slowdown.

We use the notation $T(seq)$ for the execution time of a sequence *seq* of nodes. This time should be a safe estimate of the execution time of the corresponding instructions on the target hardware. The execution time for a sequence of nodes is obtained by summing all node times and timing effects defined for the sequence:

$$T(seq) = \sum_{\forall n \in \text{nodes}(seq)} t_n + \sum_{\forall s \in \text{subsequences}(seq)} \delta_s$$

Figure 10(a) shows a small piece of a scope graph, containing three basic blocks `Q`, `R` and `S`. The example processor includes an in-order scalar pipeline containing an instruction fetch (`IF`), an execute (`EX`), and a memory access (`M`) stage. It also contains a separate floating point (`F`) stage, which can be used instead of the (`EX`) and (`M`) stages. Instructions can execute in the (`F`) stage for several clock cycles.

Figure 10(b) shows the pipeline layout of three basic blocks, and that the execution times for the nodes are 8 cycles for `Q`, 6 cycles for `R` and 7 cycles for `S`, respectively. When executing the blocks in sequence, the total execution time for the sequence is usually smaller than the sum of the times for the individual basic blocks, reflecting that the instructions of the basic blocks are overlapping in the pipeline. For example, the timing for the sequence `QR` is 11 cycles, while the sum of the execution times for `Q` and `R` is $8 + 6 = 14$ cycles. Similarly, the time for executing sequence `QS` is 14 cycles, while the execution time sum of `Q` and `S` is $8 + 7 = 15$ cycles. The overlaps give the timing effects $\delta_{QR} = T(QR) - t_Q - t_R = 11 - 8 - 6 = -3$ and $\delta_{QS} = T(QS) - t_Q - t_S = 14 - 8 - 7 = -1$, as shown in Figure 10(c).

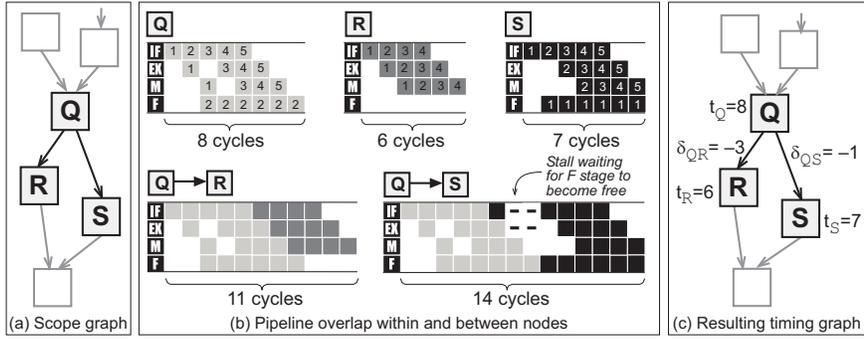An important feature of this model is that we
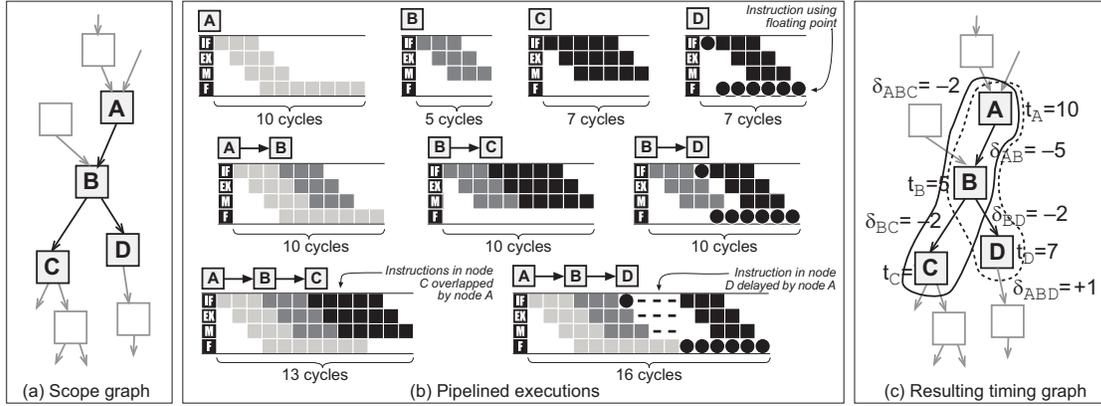
Fig. 10. Pipeline overlap

Fig. 11. Pipeline overlap and interference over three nodes

can get *long timing effects*, timing effects over sequences of nodes longer than two. Such effects occur when there are effects in the hardware reaching across more than adjacent basic blocks. Two examples of when long timing effects appear are given in Figure 11. Execution sequence ABC shows that a basic block can overlap more than one successor, while execution sequence ABD shows how a basic block can delay a later basic block across intervening blocks.

The timing model for these two cases is summarized in Figure 11(c), showing that we have long timing effects $\delta_{ABC} = T(ABC) - \delta_{AB} - \delta_{BC} - t_A - t_B - t_C = 13 + 2 + 5 - 10 - 5 - 7 = -2$ and $\delta_{ABD} = T(ABD) - \delta_{AB} - \delta_{BD} - t_A - t_B - t_D = 16 + 2 + 5 - 10 - 5 - 7 = +1$. In general, long timing effects can be both positive and negative, and can potentially occur for sequences of nodes of arbitrary lengths [19].

Our timing model is not tied to a particular method of low-level analysis; while it has obvious affinities with the pipeline analysis method used in our prototype tool (presented in Section IX), it can also represent the results of other varieties of low-level analyses. For example, the integrated cache and pipeline analysis for the Motorola ColdFire 5307 processor presented by Ferdinand

et al. [21] generates a model where times are assigned to basic blocks in a program (using an iterative analysis to include the effect of both pipelines and caches on the timing of each block). This model can be represented in our timing model framework, using node times only and no timing effects. Similarly, the timing model for Infineon C167 presented by Atanassov et al. [31] attributes times only to edges in the control flow graph. This can be represented in our timing model framework, using timing effects only on sequences of length two and no node times. Thus, both low-level analysis methods can easily be combined with the clustered calculation method presented in this paper.

VIII. CLUSTERING AND LONG TIMING EFFECTS

For efficiency reasons, local and clustered calculation methods perform WCET calculations on small parts of a program. As discussed in Section VII, timing effects can occur between nodes and on longer sequences of nodes. Thus, timing effects can be present which are only partially covered by the nodes included in a local calculation. For example, since fact clusters are formed only from flow information, there are no guarantees that the low-level timing information

will be nicely split along the same boundaries in the program.

A WCET calculation method must be able to handle such border crossing timing effects safely and efficiently to be correct and useful. Our solution to this problem assumes that the timing model is global, i.e. that the low-level analysis has been performed across the entire program. We use this global timing model to construct a local timing model valid for the considered graph fragment and extends this model to include all relevant timing effects.

We note that a timing effect for a sequence of nodes should only be accounted for when the execution goes uninterrupted via all the nodes in the sequence. However, when constructing a timing model for a graph fragment, no knowledge of the worst case execution path in the surrounding fragments can be assumed, i.e. we do not know if the prefix nodes of the sequence were taken or not. Therefore, at the boundaries between fragments, rules are needed to ensure safe timing and (hopefully) precision. We need to use potentially pessimistic but safe assumptions on the possible executions made in the surrounding fragments.

We solve this problem by introducing *history nodes*, special nodes that are added to the timing model and that represent the potential paths taken *before* entering the given graph fragment. In a fragment containing incoming timing effects, such history nodes are added at the start of the fragment.

History nodes have an execution time of zero, and their only purpose is to provide support for the specification of long timing effects. A history node is added for each node belonging to a surrounding fragment which is also included in a long timing effect which ends in a node in the fragment. Edges are added to connect the history nodes, with the result that all incoming timing effects get a corresponding history node sequence. Each added edge has a timing effect of zero.

If there are incoming execution paths not included in any timing effect sequence, extra edges should be added from the start node to the entry of the fragment. This allow the calculations to bypass incoming timing effects if this would result in a larger WCET estimate for the fragment.

Figure 12 shows an example of the handling of long timing effects. Due to the two (overlapping) long timing effects over the sequences CEFG

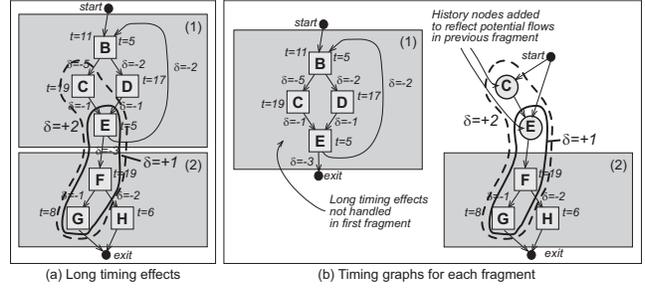(a) Long timing effects   (b) Timing graphs for each fragment

Fig. 12. Managing long timing effects at fragment borders

and EFG, two history nodes corresponding to nodes C and E are added in the timing model for fragment (2), together with the long timing effects. The calculation for fragment (2) will then find the local longest path including the long timing effects overlapping fragments (1) and (2).

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. For example, in Figure 12 the worst case execution path for fragment 1 will exit in sequence DE, giving that the long timing effects should not be used in the calculations in fragment 2. However, this cannot be accurately represented in the local timing model for fragment 2, which will cause an overestimation of the WCET. This imprecision is the price we have to pay for the convenience of using a subdivided calculation method.

In the clustered calculation border-crossing timing effects can come both from surrounding scopes and from subordinate scopes. Figure 13 gives an example of long timing effects reaching over fact cluster calculation borders. Figure 13(a) shows a scope graph fragment with the scopes t and u. Scope t has a flow fact covering scope t but not scope u. This generates a fact cluster only consisting of this single fact covering only the t scope. The corresponding timing graph fragment contains timing for nodes and edges, (not shown in the figure), as well as two long timing effects, illustrated as δ_{ACD} and δ_{EGI}.

When making a clustered calculation the u scope will be calculated in isolation from the t scope. A worst case estimate for scope u is first calculated and given a timing. The resulting scope graph for scope t is illustrated in Figure 13(b). In the graph, the call to scope u has been replaced with a call-node. To be sure to capture the long timing effect δ_{ACD} a history node is created for the A node. Since we can enter t by two different edges, an extra edge from the start node is added
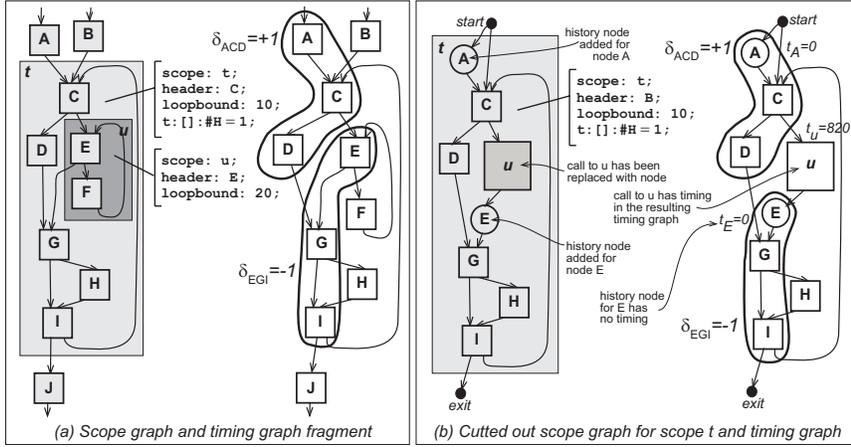
Fig. 13.   Example of clustered calculation and long timing effects

to allow the calculation to bypass the timing effect sequence.

We also have a long timing effect $\delta_{\texttt{EGI}}$ originating in node $\texttt{E}$ in the non-covered scope $\texttt{u}$. To be sure to capture this long timing effect, a history node is created for the $\texttt{E}$ node. Note that this node is connected to the node for the subscope $\texttt{u}$, since the long timing effect starts within $\texttt{u}$.

IX. WCET TOOL IMPLEMENTATION

The work presented in this paper has been implemented within the framework of our WCET analysis tool. Figure 14 gives an overview of our WCET tool, illustrating where the clustered calculation module fits in. Compared to our previously presented work [5], [26] all components of the system except the calculation phase remain unchanged, demonstrating the modular structure of the tool.

The prototype runs under Solaris, Linux, and Windows. The tool currently supports low-level WCET analysis for two different target chips, NEC V850E [32] and ARM9 [33], both typical 32-bit RISC micro-controller architectures with pipelines. For the current experiments, we only use the V850E model since it has more interesting behavior.

The WCET analysis tool supports three different calculation methods: A Path-based local calculation method [26], [28], an Extended IPET calculation method [5], [28], and the clustered calculation method described in this article.

As illustrated in Figure 14 all calculation methods take the same inputs and can be used interchangeably. The clear separation between the flow- and low-level analysis from the calculation phase makes it possible to systematically compare the performance and precision of our differ-

ent calculation methods. Figure 15 illustrates the clustered calculation method in more detail. Note that both the Path-based and the Extended IPET methods are used by the clustered calculation method to compute the WCET for clusters.

The flow analysis is currently performed manually, but we have an automatic flow analysis under development [8]. The flow analysis results in a scope graph annotated with flow facts, representing the dynamic behaviour of the program (see Section III).

A. Low-Level Timing Analysis

To extract the low-level hardware timing of the program we perform a *pipeline timing analysis* on the nodes in scope graph, generating a *timing graph* which is a concrete representation of the timing model presented above.

The pipeline analysis uses a trace-driven cycle-accurate simulator of the target processor to generate times. The core of the method is illustrated in Figure 16. Each node and sequence of nodes is executed in the simulator, and the resulting times are used to construct the timing model. The analysis treats the hardware model as a black box, i.e. it does not need to have access to its internal state. In principle, the weak requirements on the hardware model allow us to use any trace-driven cycle-accurate processor model (or even the hardware itself) to extract timing for instructions.

To include information such as cache misses and hits, and various memory speeds in the analysis, the nodes in the scope graph can be annotated with additional *execution information* [19], [28] as illustrated in Figure 18(a) and Figure 16. The hardware model must be able to correctly mimic
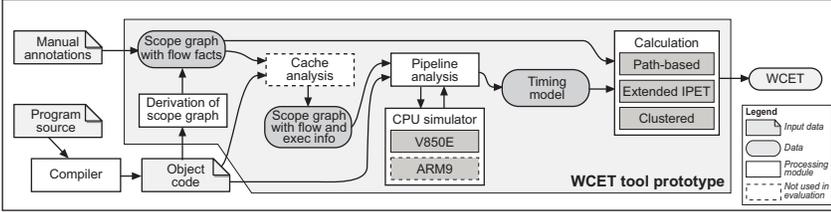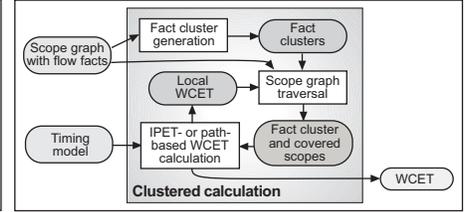
Fig. 14.   WCET tool architecture

Fig. 15.   Clustered calculation unit

the behaviour of the hardware for the provided execution information. We have implemented an instruction cache analysis similar to the one presented by Ferdinand et al. [13], but do not use this analysis in this paper since the V850E typically does not use a cache.

B. Efficient Path-based Calculation

The path-based calculation method used in our tool is very fast and can handle many types of flow facts and all types of long timing effects. The method gains its efficiency from using our timing model which allows the timing of a path to be composed from smaller components (cf. Section VII). This allows us to reformulate the longest path search problem to finding the longest path in a directed acyclic graph, something which can be solved very efficiently using well-known graph-search techniques, such as Dijkstra's Algorithm [34].

Other approaches to longest executable path search for WCET analysis are based on generating 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 for these analyses 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 program segment being analyzed.

Instead, our Path-based calculation method only needs to visit each node and edge once, therefore having a complexity linear to the size of the program graph. An extracted execution path is checked against provided flow facts for feasibility [26]. The method uses graph rewriting in order to delete paths not feasible and to safely handle long timing effects.

The method can only handle a subset of the flow facts expressable in our flow fact language (cf. Section III). Since the method finds the longest path in each scope in isolation, only loop bounds and flow facts of foreach type with

a cover of only one scope can be handled[2]. Another problem for our path-based method is unstructured code, since it makes it difficult to define what constitutes a path or a single iteration.

To handle flow facts valid for certain iterations we perform a graph transformation, unrolling a given scope into a set of *virtual scopes*. Each virtual scope is a copy of the original scope and represents a certain range of iterations for which a given set of flow facts should hold. Figure 17 gives an illustration of a virtual scope expansion. Here, the two facts `s:<1..5>:#C = 1` and `s:<3..10>:#B + E = 1` are specified for the scope `s`. Both facts hold for the iterations $3..5$. Only fact `f1` holds in iterations $1..2$, and `f2` 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 should be valid for each virtual scope, and derive a longest path for each virtual scope in isolation. The derived paths together form a longest path for the whole scope. For a more detailed description of these techniques we refer to [26], [28], [29].

C. Extended IPET Calculation

We also have an Extended IPET calculation method implemented in our WCET tool. The basics of IPET was presented in Section II. We have extended IPET to be able to handle long timing effects [20] and complex flow facts [5], [28]. The Extended IPET calculation can take into account all types of flow facts described in Section III-A, including both foreach and total facts, and flow facts covering several scopes. Similar to other IPET approaches our Extended IPET calculation method considers the whole program at once.

To handle flow facts valid for just some iterations the scope graph is unrolled into a virtual

---

[2]Path-based methods can be extended to handle triangular loop dependencies and unstructured code [35], this is however not implemented in our path-based method.
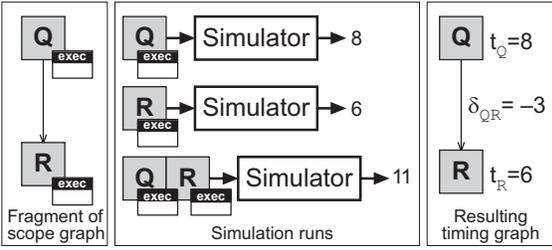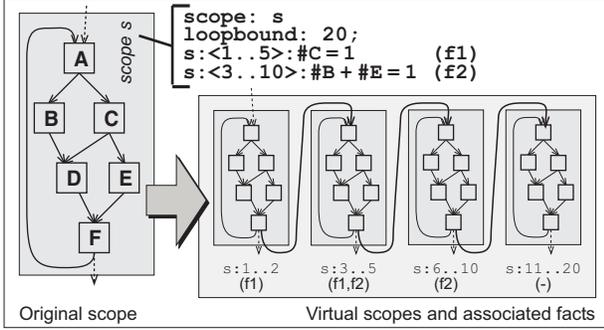
Fig. 16.   Pipeline timing analysis

Fig. 17.   Virtual scope expansion

Fig. 18.   Using instruction cache analysis results

scope graph. Compared to our path-based calculation method, which unrolls only one individual scope at a time, our Extended IPET calculation method considers and unroll all the scopes in the scope graph at the same time. All flow facts are lifted to a global level and converted to constraints over entities in the virtual scope graph. The structure of the virtual scope graph is used to generate structural constraints. Each (long) timing effect is given a count variable, and constraints are generated to limit the number of times each timing effect can be accounted for. For a more detailed description of these techniques we refer to [28].

In the current experiments we rely on the mixed ILP solver `lp_solve` [36] to solve generated constraint systems. We also support the possibility to export generated constraints in a format suitable as input to Sicstus Prolog CP [37] or to the mixed integer and constraint solver CPLEX [38].

D. Clustered Calculation Implementation

We have also implemented the clustered calculation method described in this article. The method comes in two versions. In the first version (Extended) IPET is always used to calculate WCET times for clusters. In the second version, depending on the characteristics of the cluster, either the path-based calculation or IPET is utlized when calculating a WCET estimate for a cluster. Specifically, path-based calculation is used for clusters covering just one scope and containing no total flow facts, while IPET is used for all other clusters. To our knowledge this is the first WCET calculation method making use of two different type of calculation schemes for the calculation of a single program.

In the experiments, the clustered calculation version using both path-based and IPET is called *Clustered Mix*, while the version using only IPET is called *Clustered IPET*.

X. EXPERIMENTS 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 Table I. 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 methods thoroughly[3]. The **BB** column gives the number of basic blocks when compiled for the V850E, **Sc** gives the number of scopes and **FF** the number of flow facts in the corresponding scope graph. All flow facts where added manually. The **Lte** column gives the number of non-zero long timing effects with a length larger than two.

For each program we have derived an *actual WCET* value by running a worst-case trace of the program through the CPU simulator used by the WCET analysis. Thereby, the experiments

---

[3]The benchmarks are available for download at `http://www.c-lab.de/home/en/download.html`

Program	Description	Properties	BB	Sc	FF	Lte
adpcm	adaptive differential pulse code modulation algorithm	Completely well-structured code	148	59	43	0
compress	Compression using lzw.	Nested loops, goto-loop, function calls.	91	24	9	3
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.	29	9	6	0
duff	Using "Duff's device" to copy 43 byte array.	Unstructured loop with known bound, switch statement.	20	6	2	0
expint	Series expansion for computing an exponential integral function	Inner loop that only runs once, structural WCET estimate gives heavy overestimate.	24	7	4	0
fibcall	Simple iterative Fibonacci calculation, used to calculate fib(30).	Parameter-dependent function, single-nested loop.	7	4	0	0
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.	14	5	7	0
isort	Insertion sort on a reversed array of size 10.	Input-data dependent nested loop with worst-case of $n^2/2$ iterations.	7	4	1	0
jfdctint	Discrete-cosine transformation on a 8x8 pixel block.	Long calculation sequences (i.e. long basic blocks), single-nested loops.	14	6	0	3
lcdnum	Read ten values, output half to LCD	Loop with iteration-dependent flow.	26	4	2	0
matmult	Matrix multiplication of two 20x20 matrices.	Multiple calls to the same function, nested function calls, triple-nested loops.	27	16	0	0
ns	Search in a multi-dimensional array	Return from the middle of a loop nest, deep loop nesting.	18	7	1	0
nsichneu	Simulate an extended Petri Net	Automatically generated code containing massive amounts of if-statements ($\gg 250$)	754	3	129	1

TABLE I. Benchmark programs used in experiments

only deal with the effectiveness and precision of the WCET analysis phases, and do not need to take into account any differences between the simulator and the actual hardware. The traces were obtained by manual analysis of the program source and object codes, and extensive testing was undertaken to make sure that they really correspond to the actual WCETs. All measurements were performed on a Pentium 4, 2.66 GHz, with 256 MB RAM.

A. Necessity of correct flow and timing

The necessity for both correct low-level timing and flow information when calculating safe and tight WCET estimates is illustrated in Table II. The column **Basic** gives the WCET estimate when ignoring pipeline overlap between basic blocks, (but including the pipeline overlap within basic blocks), and using only basic loop bounds as flow information. The columns including **Flow** give WCET estimates resulting from adding flow facts to the program. Columns including **Pipe** give WCET estimates where timing effects between nodes have been accounted for. Column **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, relative to the actual WCET.

The need for correct pipeline timing is illustrated by the values in the **With Pipe** column. The WCET overestimation is clearly reduced compared to the **Basic** column. In general, modelling

pipeline overlaps between basic blocks seems to tighten the WCET estimates by at least 20 percent, (ignoring pipeline effects *within* basic block would create WCET estimates about five times higher, since the V850E has a five-stage pipeline). The benefit is greatest for programs with many small basic blocks (such as crc, fibcall and nsichneu), and least for programs with large basic blocks (for example, jfdctint).

The need for correct flow information is illustrated by the values in the **With Flow** column. The improvements in WCET estimate precison due to flow information vary much more for the different benchmarks than the effect of pipeline analysis. Some benchmarks, for example compress and nsichneu, show large decrease in obtained WCET overestimations, while for other programs the improvement is much smaller.

The good results in the **Flow & Pipe** column indicate that to obtain WCET estimates both high quality flow and timing information must be obtained. All WCET estimates are safe, i.e. larger than or equal to the actual WCET. The remaining execution time overestimation is mostly due to the problem of correctly modelling the program flow. All WCET estimates were calculated using the Extended IPET method.

B. Calculation method comparasion

Table III shows the WCET estimate precision (**cycles**) in clock cycles and needed computation time (**time**) in seconds, of the different calculation methods. The **Clustered IPET** holds

Program	Basic Cycles	+%	With Flow Cycles	+%	With Pipe Cycles	+%	Flow & Pipe Cycles	+%	Actual Cycles
adpcm	8954	+42.5	8714	+38.7	6382	+1.59	6282	0	6282
compress	126242	+1357	10388	+20	92482	+967	8672	+0.12	8662
crc	61624	+104	61340	+103	30389	+0.39	30271	0	30271
duff	1823	+86.3	1775	+63.9	1104	+1.9	1083	0	1083
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
isort	2328	+117	1428	+33.0	1794	+67.0	1074	0	1074
jfdctint	5388	+9.4	5388	+9.4	4925	0	4925	0	4925
lcdnum	501	+153	341	+72.2	283	+42.9	198	0	198
matmult	275879	+24.4	275859	+24.4	221824	0	221824	0	221824
ns	25741	+84.8	25713	+84.6	17373	+24.7	17353	+24.6	13928
nsichneu	150841	+195	87193	+70.6	97645	+91.0	51116	0	51116

TABLE II. WCET estimates with and without flow facts and timing effects

Program	Path-based cycles	+%	time	Extended IPET cycles	+%	time	Clustered IPET cycles	+%	time	Clustered Mix cycles	+%	time	Actual WCET
adpcm	6282	0	0.02	6282	0	0.24	6282	0	0.18	6282	0	0.12	6282
compress	8670	+0.09	0.01	8670	+0.09	0.21	8670	+0.09	0.12	8670	+0.09	0.1	8662
crc	58435	+93	0	30271	0	0.03	30271	0	0.04	30271	0	0.03	30271
duff	-	-	-	1083	0	0.01	1083	0	0.04	1083	0	0.04	1083
expint	8588	0	0.01	8588	0	0.01	8588	0	0.04	8588	0	0.03	8588
fibcall	313	0	0.01	313	0	0.01	313	0	0.01	313	0	0.01	313
fir	352073	+1.14	0.02	352073	+1.14	0.03	352073	+1.14	0.04	352073	+1.14	0.04	348095
isort	1794	+67.0	0.02	1074	0	0.01	1074	0	0	1074	0	0.01	1074
jfdctint	4942	+0.40	0.01	4925	0	0.01	4926	+0.02	0.03	4942	+0.40	0.01	4925
lcdnum	198	0	0	198	0	0.04	198	0	0.11	198	0	0.1	198
matmult	221824	0	0.01	221824	0	0.02	221824	0	0.04	221824	0	0.04	221824
ns	17361	+24.6	0.01	17353	+24.59	0.01	17353	+24.59	0.03	17353	+24.59	0.03	13928
nsichneu	51133	+0.03	0.03	51116	0	1.4	51116	0	1.5	51133	+0.03	0.22	51116

TABLE III. WCET estimate precision and calculation time

measurements made when always using IPET to calculate WCETs for clusters. **Clustered Mix** holds measurements made when making path-based calculation within clusters covering just one scope and not containing total flow facts, and IPET calculation within remaining clusters. 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 cover of a single scope often are sufficient for obtaining precise WCET estimates. However, programs such as `isort` 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 methods are of the same quality as the Extended IPET, the current method with highest precision.

For all our benchmarks, except `adpcm` and `nsichneu` the time spent in the calculation stages is almost negligible. This is because most of the benchmarks programs given in Table I are quite small and do not really stress our calculation
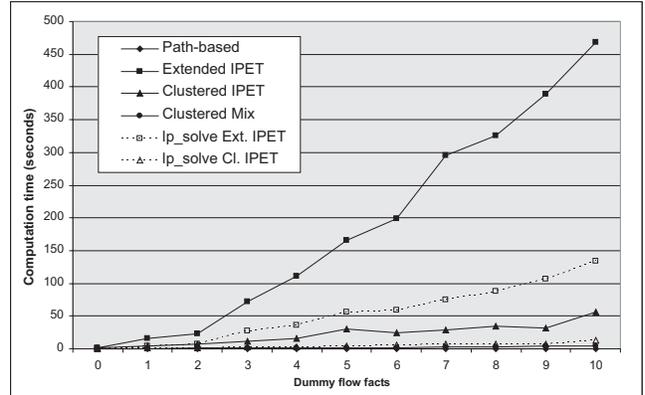
Fig. 19.   Computation time scaling

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 Table I). 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 execution paths, we increase the computational load. For example,

Extra	Path-based			Extended IPET				Clustered IPET			Clustered Mix		
facts	time	expl.	paths	time	lptime	constr.	vars.	time	lptime	calcs	time	icalls	pcalls
0	0.03	5	3.73E97	1.4	0.43	1651	2139	1.5	0.38	2	0.22	0	2
1	0.08	12	1.11E98	15.9	4.45	3417	4528	4.12	0.97	4	0.54	0	4
2	0.13	19	1.87E98	23.03	7.55	4931	6665	7.16	1.7	6	0.99	0	6
3	0.17	26	2.61E98	71.87	26.69	6445	8802	11.79	2.78	8	1.36	0	8
4	0.23	33	3.36E98	110.85	36.56	7959	10939	15.61	3.53	10	1.52	0	10
5	0.28	40	4.11E98	165.31	56.71	9473	13076	30.35	4.79	12	1.83	0	12
6	0.34	47	4.85E98	199.34	59.35	10987	15213	24.59	5.73	14	2.15	0	14
7	0.45	54	5.59E98	295.26	74.88	12501	17350	28.27	6.6	16	2.47	0	16
8	0.46	61	6.34E98	326.21	88.33	14015	19487	34.26	7.76	18	3.2	0	18
9	0.58	68	7.09E98	389.45	106.62	15529	21624	31.61	7.28	20	4.2	0	20
10	0.65	75	7.83E98	468.98	134.47	17043	23761	55.77	12.99	22	4.62	0	22

TABLE IV. Scaling measures of calculation methods

adding one dummy fact creates a virtual scope graph consisting of 1508 (752 + 752 + 4) scope nodes. The Extended IPET method creates a constraint system over the whole graph while the clustered and path-based calculation methods partition the resulting problem into smaller subproblems. For each calculation run all WCET estimates achieved were exactly the same as reported in Table III.

Table IV gives computation times obtained for our calculation methods when adding dummy flow 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 IPET calculation the computation time (**time**), the number of lp_solve calls made (**calcs**) and the total time spent in lp_solve (**lptime**) are given. For the Clustered Mix calculation the computation time (**time**), the number of path calls (**pcalls**) and the number of IPET calls (**icalls**) are given.

Figure 19 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 calculations, while the Extended IPET has a more than linear increase. Both the Extended IPET and the clustered calculations 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 IPET. 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 2159 variables and 1666 constraints and took approximately 0.3 seconds.

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. Clustered IPET is somewhere in between in complexity, scaling reasonably well, while still being able to handle complex flow information. The Clustered Mix scales even better, relying on the fact that our path-based method can be used for all calculations of clusters.

C. Clustered calculation evaluation

We have implemented all five clustering algorithms outlined in Section IV-B. The algorithms differ in how many flow facts will be grouped together, and consequently in the size of the scope graph that will be covered by each fact cluster. Table V 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 **calls** 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. All measurements where used using Clustered IPET.

Program	min-split			minimum			scope			max			global		
	cl	calls	time	cl	calls	time	cl	calls	time	cl	calls	time	cl	calls	time
adpcm	18	59	0.17	18	59	0.18	18	59	0.18	1	36	0.23	1	1	0.23
compress	6	38	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
isort	1	1	0.01	1	1	0.01	1	1	0.01	1	1	0.01	1	1	0.01
jfdctint	0	5	0.02	0	5	0.02	0	5	0.02	0	5	0.02	0	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

TABLE V. Clustered calculation measures

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 Table III.

Table VI shows measurements done by the Clustered IPET and Clustered Mix calculation methods. For the Clustered IPET calculation the computation time (**time**) the number of IPET calls (**icalls**) are given. For the Clustered Mix calculation the computation time (**time**), the number of path calls (**pcalls**) and the number of IPET calls (**icalls**) are given. We note that both methods always make the same number of local calculation calls. For most programs most local calculations can be made using path-based calculation. However, for some programs, such as **crc** and duff, IPET is needed in some calculations. All WCET values except for jfdctint and nsichneu are identical, indicating that Clustered Mix obtain almost the same precicision as Clustered IPET. For all programs the time calculation time of the Clustered Mix is smaller than Clustered IPET. All measurements where made using split-foreach-fact clustering.

As discussed in Section V-A, some fact clusters define graph fragments with several entry and exit points, allowing us to trade WCET estimate

precision for speed. Table VII 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 calculations performed for each program. We note that for many programs, the number of local WCET calculations 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 V-A.

D. Impact of long timing effects

A long timing effect is an effect reaching over a sequence of three or more nodes (see Section VII). The number of long timing effects varies with the processor architecture and the program code properties. For the V850E, only a few of our benchmark programs contained long timing effects, as shown by the **Lte** column in Table I.

We have implemented a model of the NEC V850E with a data memory latency of 6 clock-cycles. This is an artificial model not corresponding to any real setup of the V850E, but very useful in provoking long timing effects. We use the model to investigate how long timing effects affect the WCET estimate precision. This is particularly relevant for calculations that partition the program into smaller parts to increase efficiency, such as our path-based and clustered calculation. When long timing effects reach over calculation boundaries they might introduce pessimism in

Program	cl	Clustered IPET			Clustered Mix			
		icalls	cycles	time	icalls	pcalls	cycles	time
adpcm	18	59	6282	0.18	0	59	6282	0.12
compress	6	38	8670	0.12	0	38	8670	0.1
crc	6	6	30271	0.04	2	4	30271	0.03
duff	2	12	1083	0.04	9	3	1083	0.04
expint	4	11	8588	0.04	0	11	8588	0.03
fibcall	0	4	313	0.01	0	4	313	0.01
fir	5	10	352073	0.04	0	10	352073	0.04
isort	1	1	1074	0	1	0	1074	0.01
jfdctint	0	5	4926	0.03	0	5	4942	0.01
lcdnum	2	23	198	0.11	0	23	198	0.1
matmult	0	15	221824	0.04	0	15	221824	0.04
ns	1	13	17353	0.03	0	13	17353	0.03
nsichneu	1	2	51116	1.5	0	2	51133	0.22

TABLE VI. Detailed Clustered IPET and Clustered Mix comparision

Program	diff in-out			no diff		
	cl	calls	cycles	cl	calls	cycles
adpcm	18	59	6282	18	58	6282
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
isort	1	1	1074	1	1	1074
jfdctint	0	5	4926	0	5	4926
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

TABLE VII. Entry and exit point differentiation effect

the WCET estimate calculation, as discussed in Section VIII. For example, in Table III the one clock cycle difference of WCET estimates for jfdctint of the clustered and the Extended IPET calculation, is caused by a border-crossing long timing effect.

Table VIII gives the WCET precision achieved for our calculation methods when using the V850E model with 6 cycle data memory. We use the same scope graphs, flow facts and basic block graphs as for the runs in Table III. For some programs, such as adpcm, compress and matmult, many long timing effects appear, while for others there is no impact.

More long timing effects make the precision of the clustered WCET estimates for adpcm, compress, jfdctint, matmult and ns to be a little worse than for Extended IPET. Since the WCET estimates for these programs were identical when using the basic V850E model, as given Table III, we conclude that the obtained pessimism is due to long timing effects across calculation boundaries.

For all our benchmarks only negative long timing effects where observed, even though the V850E CPU has a potential of exhibiting positive long timing effects [19]. This means that for the tested programs, a calculation method would obtain pessimistic but safe estimates by ignoring long timing effects.

To evaluate the effect of ignoring long timing effects, we calculated WCET estimates for programs using the slow memory and Extended IPET calculation method, while ignoring all long timing effects. The result is shown in the **IPET, no Lte** column in Table IX. For all programs except isort, ignoring long timing effects leads to an overestimation of the WCET estimate. For some programs, such as duff and lcdnum, the consequences of ignoring long timing effects are quite substantial. Altogether, we conclude that long timing effects must be modelled in order to generate safe and tight WCET estimates.

XI. 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 such as 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. 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, the properties of the provided flow information and the timing characteristics of the target hardware. 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 [8] into our WCET analysis tool. Preliminary results indicate that such analyses are likely to produce a large

Program	Path-based		Clustered IPET		Ext. IPET		Actual WCET	Lte
	Cycles	+%	Cycles	+%	Cycles	+%		
adpcm	12217	+2.02	11985	+0.08	11983	+0.07	11975	36
compress	15457	+1.10	15306	+0.10	15304	+0.10	15289	26
crc	34432	+1.53	33917	+0.01	33917	+0.01	33914	8
duff	-	-	1468	0	1468	0	1467	2
expint	8720	0	8720	0	8720	0	8720	0
fibcall	332	0	332	0	332	0	332	0
fir	597206	+1.19	590168	0	590168	0	590168	0
isort	3963	+40.55	2361	+0.21	2361	+0.21	2356	3
jfdctint	6626	+3.77	6393	+0.13	6392	+0.12	6385	3
lcdnum	268	+12.61	238	0	238	0	238	5
matmult	312550	+9.53	287470	+0.74	287468	+0.74	285348	13
ns	29243	+70.56	21350	+24.53	21349	+24.52	17145	1
nsichneu	135793	+0.01	135776	0	135776	0	135776	3

TABLE VIII. WCET precision using V850 model with 6 cycle memory

Program	IPET, Lte		IPET, no Lte		Actual WCET
	Cycles	+%	Cycles	+%	
adpcm	11983	+0.07	12252	+2.31	11975
compress	15304	+0.10	15461	+1.12	15289
crc	33917	+0.01	34427	+1.50	33914
duff	1468	0	1765	+20.31	1467
isort	2361	+0.21	2361	+0.21	2356
jfdctint	6392	+0.12	6626	+3.77	6385
lcdnum	238	0	268	+12.61	238
matmult	287468	+0.74	312550	+9.53	285348
ns	21349	+24.52	21350	+24.53	17145
nsichneu	135776	0	135793	+0.01	135776

TABLE IX. Precision when ignoring long timing effects

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.

REFERENCES

[1] N. Audsley, A. Burns, R. Davis, K. Tindell, and A. Wellings, "Fixed priority pre-emptive scheduling: an historical perspective," *Real-Time Systems*, vol. 8, no. 2/3, pp. 129–154, 1995.

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

[3] J. Ganssle, "Really Real-Time Systems," in *Proceedings of the Embedded Systems Conference (ESC SF) 2001*, Apr 2001.

[4] 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.

[5] 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.

[6] 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.

[7] 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*, 1995, pp. 456–461.

[8] 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'03)*, Jan 2003.

[9] C. Healy, R. Arnold, F. Müller, D. Whalley, and M. Harmon, "Bounding Pipeline and Instruction Cache Performance," *IEEE Transactions on Computers*, vol. 48, no. 1, Jan 1999.

[10] 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.

[11] 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.

[12] F. Stappert and P. Altenbernd, "Complete Worst-Case Execution Time Analysis of Straight-line Hard Real-Time Programs," *Journal of Systems Architecture*, vol. 46, no. 4, pp. 339–355, 2000.

[13] 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.

[14] 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*, vol. 21, no. 7, pp. 593–604, Jul 1995.

[15] 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)*. IEEE, 1996, pp. 230–240.

[16] 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)*, Jun 1997, pp. 192–202.

[17] A. Colin and I. Puaut, "Worst Case Execution Time Analysis for a Processor with Branch Prediction," *Journal of Real-Time Systems*, vol. 18, no. 2/3, pp. 249–274, May 2000.

[18] T. Mitra and A. Roychoudhury, "Effects of Branch Prediction on Worst Case Execution Time of Programs," National University of Singapore (NUS), Tech. Rep. 11-01, Nov

2001.

[19] J. Engblom, "Processor Pipelines and Static Worst-Case Execution Time Analysis," Ph.D. dissertation, Uppsala University, Dept. of Information Technology, Box 337, Uppsala, Sweden, Apr 2002, ISBN 91-554-5228-0.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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)*, 2002, pp. 50–59.

[26] 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.

[27] P. Puschner and A. Schedl, "Computing Maximum Task Execution Times with Linear Programming Techniques," Technische Universität Wien, Institut für Technische Informatik, Tech. Rep., Apr 1995.

[28] A. Ermedahl, "A Modular Tool Architecture for Worst-Case Execution Time Analysis," Ph.D. dissertation, Uppsala University, Dept. of Information Technology, Box 325, Uppsala, Sweden, Jun 2003, ISBN 91-554-5671-5.

[29] F. Stappert, "From Low-Level to Model-Based and Constructive Worst-Case Execution Time Analysis," Ph.D. dissertation, Faculty of Computer Science, Electrical Engineering, and Mathematics, University of Paderborn, 2004, C-LAB Publication, Vol. 17, Shaker Verlag, ISBN 3-8322-2637-0.

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

[31] 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.

[32] NEC Corporation, *V850E/MS1 32/16-bit Single Chip Microcontroller: Architecture*, 3rd ed., Jan 1999, document no. U12197EJ3V0UM00.

[33] *ARM 9TDMI Technical Reference Manual*, 3rd ed., ARM Ltd., Mar 2000, document no. DDI 0180A.

[34] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 2nd ed. MIT Press, 2002.

[35] 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.

[36] M. Berkelaar, *lp_solve: (Mixed Integer) Linear Programming Problem Solver*, 2004, URL: `ftp://ftp.es.ele.tue.nl/pub/lp_solve`.

[37] "SICStus Prolog user's manual," Swedish Institute of Computer Science," ISBN 91-630-3648-7, 1995.

[38] "ILOG CPLEX homepage," 2004, `http://www.ilog.com/products/cplex/`.