Modeling Complex Flows for Worst-Case Execution Time Analysis

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

Andreas Ermedahl[†]
Dept. of Computer Systems, Uppsala University
Box 325, SE-751 05 Uppsala, Sweden
email: andreas.ermedahl@docs.uu.se

Abstract

Knowing the Worst-Case Execution Time (WCET) of a program is necessary when designing and verifying real-time systems. The WCET depends both on the program flow (like loop iterations and function calls), and on hardware factors like caches and pipelines.

In this paper we present a method for representing program flow information, that is compact while still being strong enough to handle the types of flow previously considered in WCET research. We also extend the set of representable flows compared to previous research.

We give an algorithm for converting the flow information to the linear constraints used in calculating a WCET estimate in our WCET analysis tool.

We demonstrate the practicality of the representation by modeling the flow of a number of programs, and show that execution time estimates can be made tighter by using flow information.

Keywords: WCET, flow information, hard real-time, embedded systems, static program analysis, IPET.

1. Introduction

The purpose of *Worst-Case Execution Time* (WCET) analysis is to provide a priori information about the worst possible execution time of a piece of code before using it in a system.

Knowing the WCET of a program or piece of a program is necessary when designing and verifying real-time systems. Considering that every day, more and more devices are being controlled by embedded real-time systems (from kitchen appliances through power grids to cars and other vehicles), the value of having reliable software cannot be overestimated. In many cases, a failure of an embedded real-time system will lead to a disaster, sometimes including the loss of human life.

WCET estimates are used in real-time systems development to perform scheduling and schedulability analysis, to determine whether performance goals are met for periodic tasks, and to check that interrupts have sufficiently short reaction times.

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

The WCET depends both on the program flow (like loop iterations and function calls), and on architectural factors like caches and pipelines. Thus, both the program flow and the hardware the program runs on must be modelled by a WCET analysis method.

When performing WCET analysis, it is assumed that the program execution is uninterrupted (no preemptions or interrupts) and that there are no interfering background activities, such as direct memory access (DMA) and refresh of DRAM. Timing interference caused by such resource contention is assumed to be handled by some subsequent analysis, for instance schedulability analysis.

In this paper, we investigate how to model program flow to achieve the tightest possible worst-case execution time estimates. There is a need for a comprehensive flow modeling mechanism, since previous research has only modeled some subset of the interesting flows found in actual code [2, 5]. We are able to represent the types of flows previously considered in the WCET field, and some that have not been considered before.

In order to use the flow information for actual WCET estimates, we provide an algorithm for converting the representation to a format suitable for our WCET calculation method (based on the implicit path

[*] Jakob is an industrial PhD student at IAR Systems (http://www.iar.com) and Uppsala university, sharing his time between research and development work.

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

enumeration technique, IPET).

The contributions of this paper are:

- We introduce a high-level flow fact specification language that balances strength of expression and compactness in a manner appropriate for WCET analysis. The language is strong enough to handle the flow information generated by existing program analysis methods, while providing compact representations for even complex flows.
- We show how the flow facts are converted for use in an IPET-style WCET calculation.
- We perform experiments to demonstrate the usefulness of our flow information language, and demonstrate that flow information can provide tighter WCET estimates.

Paper outline: Section 2 presents previous work and Section 3 gives an overview on the issues involved in representing the flow information. In Section 4 we present our representation. Section 5 presents the conversion algorithm. Section 6 gives an overview of our WCET analysis tool and Section 7 contains our experimental evaluation. Finally, Sections 8 and 9 present conclusions and plans for future work.

2. WCET Analysis Overview and Previous Work

To generate a WCET estimate, we consider a program to be processed through the phases of *program flow analysis*, *low level analysis* and *calculation*. (For a more detailed division see [8]).

The program flow analysis phase determines possible program flows, without regard to the time for each "atomic" unit of flow. The result of the flow analysis should provide information about which functions get called, how many times loops iterate, if there are dependencies between `if`-statements, etc. The information can be obtained using *manual annotations* (integrated in the programming language [21], or provided separately [10, 14, 23]), or *automatic flow analysis* [1, 9, 11, 18, 25].

The purpose of low-level analysis is to determine the execution time for each atomic unit of flow (e.g. an instruction or a basic block) given the architecture and features of the target system. For WCET analysis, instruction caches [10, 11, 15, 25], cache hierarchies [19], data caches [13, 25, 27], branch predictors [3], scalar pipelines [7, 11, 15] and superscalar CPUs [16, 24, 25] have been analysed.

The purpose of the calculation phase is to calculate the WCET estimate for a program, given the program flow and global and local low-level analysis results. There are three main categories of calculation methods proposed in literature: *path-* [11, 25] (the final WCET estimate is generated by calculating times for explicitly represented paths in a program, searching for the path with the longest execution time), *tree-* [3, 15] (the final WCET is generated by a bottom-up traversal of a tree representing the program), or *IPET* (Implicit Path Enumeration Technique)-based.

We use an IPET-style calculation [10, 14, 20, 23] where program flow and atomic execution times are represented using algebraic and/or logical constraints. The WCET estimate is calculated by maximizing an objective function, while satisfying all constraints.

3. Issues of Flow Information

We consider flow information handling to be divided into three phases:

1. **Flow analysis**: Obtaining flow information. By manual annotations or automatic flow analysis.
2. **Flow representation**: Representing the results of the flow analysis.
3. **Calculation**: Using the control flow (as represented in the flow representation) in the final WCET calculation. Not all calculation methods can take advantage of all types of flow information.

In this paper we will deal with the last two phases by presenting a language for representing flow information and describing how the flow information can be converted to the IPET style of calculation.

3.1. Expressing program flow

The most complete form of program flow information is a simple explicit enumeration of all possible paths through the program. Every possible execution would be represented using an ordered list of statements or instructions. This representation of the dynamic behaviour of the program is too expensive for most programs. Instead, we need a compact approximate way of representing the dynamic behavior of the program. The approximation must be:

- *Safe* – no feasible worst-case executions of the program should be excluded.
- *Tight* – as few infeasible executions as possible should be included.

Note that we assume that flow analysis is performed prior to low-level analysis, which means that the flow analysis does not have access to any information about the execution time for a particular piece of code.

The largest set of executions of a program is given by the *structure* of the program: all paths that can be traced through the flow-graph of the program, regardless of the semantics of the code are considered

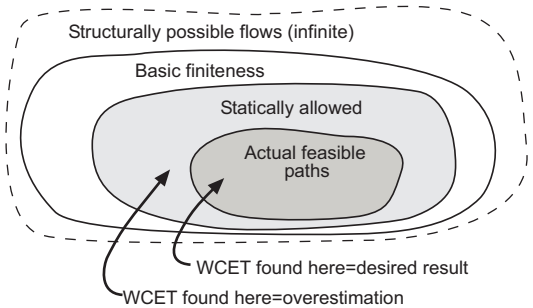


Figure 1. Relation between possible executions and flow information

possible. This set is usually infinite, since all loops[1] can be taken an arbitrary number of times. The executions are made finite by introducing *basic finiteness information*, where we bound all loops with some upper limit on the number of executions.

Adding even more information allows the set of executions to be narrowed down further, to a set of *statically feasible paths*. This is the "optimal" outcome of the flow analysis. Figure 1 provides an illustration of the sets of paths.

The calculation, finally, uses information about the execution time of each piece of code to find the paths in the set of statically feasible paths that correspond to actual worst-case execution times.

Flow information can be provided on the source code or object code level. If provided on source code level, the information must be mapped to the object code to be used in the WCET calculation. In the presence of optimization compilers, this problem is non-trivial [6, 17].

4. Representing Flow Information

To represent the dynamic behaviour of a program we introduce the concept of a *scope*. Intuitively, each scope corresponds to a certain repeating or differentiating execution environment in the program, like a function call or loop. The exact definition of a scope is up to the flow analysis algorithms employed: they can define whatever types of scopes as best suited to represent the information generated. All scopes are supposed to be looping, even if they just iterate zero or one times. Thus there exists a concept of *iterations* of scopes.

The scopes form a tree, called a *scope tree*, with the entry point to the program given by the root node. An example of a small scope hierarchy containing loops and function calls is shown in Figure 2.

[1] We consider iterative loops and recursive loops to be equivalent, and will simply use the word "loop" in this paper.

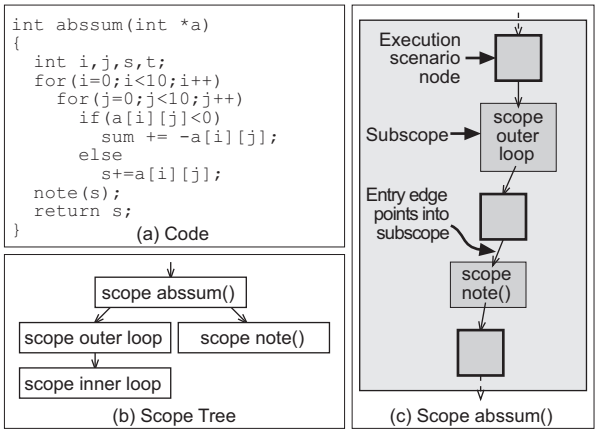


Figure 2. Example of code with associated scopes

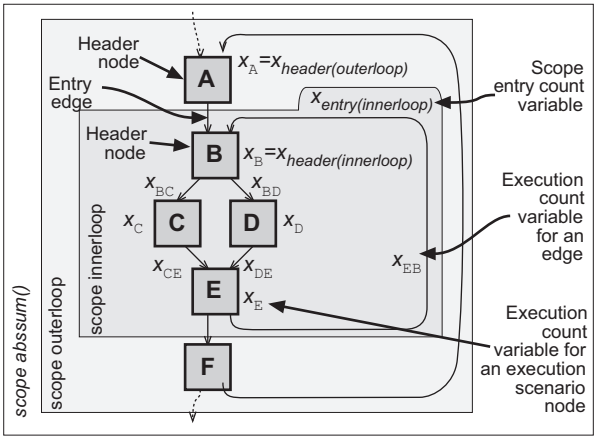


Figure 3. Example details of a scope

A scope is a set of *nodes* and *edges*. A node belongs to exactly one scope, and represents the execution of a certain basic block[2] in the program in the environment given by the scope and its superscopes. We call such a contextualized basic block an *execution scenario*. This means that there can be several "copies" of the same basic block in the program flow representation.

Each scope has a *header node* which has the property that no other node in the scope can be executed more than once without every possible execution path passing the header node.

An *edge* connects two nodes and represents potential program flow. Edges may cross into subscopes or superscopes, and the source and sink of an edge may be in different scopes.

Each scope has a set of *entry edges*, which are edges coming into the scope from surrounding scopes. Note that the entry edges *may* go to other nodes than the

[2] A basic block is a piece of object code that is always executed in sequence, i.e. it contains no branches in or out.

Fact	→	*Scope* : *ContextSpec* : *Constraint*
ContextSpec	→	<>
	|	[]
	|	< *RangeList* >
	|	[*RangeList*]
RangeList	→	*Range*
	|	*Range* , *RangeList*
Range	→	*Integer* .. *Integer*
	|	*Integer*
Constraint	→	*Expression Relop Expression*
Expression	→	*CountVariable*
	|	*Integer*
	|	(*Expression Op Expression*)
Relop	→	$\leq$ | = | $\geq$
Op	→	+ | − | ∗ | /

Figure 4. Flow Fact Specification

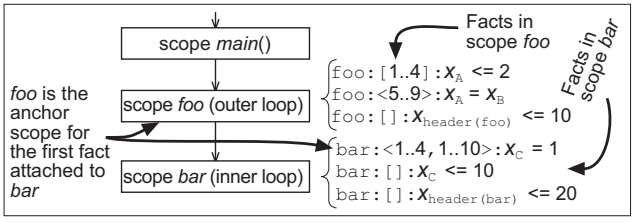


Figure 5. Example of facts attached to scopes

header node (in this case the scope describes an unstructured loop).

A *node sequence* is an ordered list of nodes from the same scope, such that there is an edge between each adjacent pair of nodes (and the edges point in the direction of the sequence).

Figure 3 shows the details of the nested loop scopes from Figure 2 (the names given for the nodes are just a notational convenience), and some details related to flow information.

If all possible paths through the program given by the edges between the nodes are considered, we get the structurally possible flows discussed above.

4.1. Constraining the flow through the scopes

Each scope has as a set of associated flow information facts. Each flow information fact consists of three parts, the name of the *scope* where the fact is defined, a *context specifier*, and a *constraint expression*. Intuitively, the constraint expression should hold when the execution is within the context specification. Figure 4 gives the grammar of the specification language.

All facts in the program are considered to be true at the same time. There are no "alternative" sets of facts.

4.1.1. Scope and Context Specifiers

The context specifier in a flow information fact gives the iterations of the scopes in which the constraint expression must be valid. The specifiers are defined using the two dimensions of *type* and *iteration space*, as shown in Figure 6.

Operator	Type	Iterations
<>	foreach	all
[]	total	all
<*ranges* . . .>	foreach	range
[*ranges* . . .]	total	range

Figure 6. Context Specification Operators

The type is either *total* or *foreach*. For a *total* operator, the fact is considered as a sum over all iterations of the context, while the *foreach* operators consider the fact as being local to a single iteration of the defining scope.

The iteration space is the set of iterations of the scope that a fact is valid for. This can either be *all* or *range*. For "all" facts, ([], <>), the facts should be valid for all iterations in the scope, while "ranges" facts, ([*RList*], <*RList*>), only are valid for some iterations. The ranges specified inside the context delimeters specifies during what iterations that the fact should be valid.

Range specifications can have one or more dimensions. Each range corresponds to precisely one scope. The last range in the list corresponds to the defining scope for the fact, the second last corresponds to the parent scope of the defining scope, the third last corresponds to the parent scope to the parent scope of the defining scope, etc. The scope that corresponds to the first range in the list is called the *anchor scope* of the fact. For *all* operators and one-dimensional ranges the anchor scope and defining scope are the same.

For example, the fact `bar : <1..4,1..10> :` $x_{\texttt{C}} = 1$, in Figure 5, has a "foreach/range" context specification with two ranges and is defined in scope `bar` and has scope `foo` as anchor scope. The fact should be applied to each individual iteration of the `bar` scope when `bar`'s iteration counter is within 1 to 10 (inclusive) and, at the same time, the iteration counter of the parent scope `foo` is within 1 to 4.

4.1.2. Constraint Specification

The constraints are specified as a relation ($\leq$,=,$\geq$) between two arithmetic expressions involving *execution count variables* and integers. The arithmetic operations allowed depends on the power of the calculation method used (the syntax specification in Figure 4 uses the common arithmetic operators +, −, ∗, /, since that is appropriate for the IPET calculation technique).

The execution count variables, (x_{entity}), correspond to nodes, sequences of nodes, and edges. The values of the variables correspond to the number of times the entities are executed, i.e. an execution profile of the program. By constraining the possible values of the vari-

ables, we limit the the set of possible execution paths. A fact can only refer to execution count variables in the defining scope and its subscopes.

The value of a variable corresponds to the number of times the entity is executed within the context. This is a *local* value, and not global to the entire program execution.

The execution count of the headernode of a scope is used to limit the number of iterations of a certain scope. The execution count variable of the headernode of a scope will be referred to as $x_{header(scope)}$.

There are also execution count variables for scopes, corresponding to the number of times the scope is entered from its parent scope. Those variables are called *entry count variables*, and will be referred to as $x_{entry(scope)}$. Note that the value of the entry count variable is equal to the sum of all the entry edge variables. Figure 3 shows an example of the variables for entries, headers, nodes, and other details of a scope.

4.1.3. Detailed Fact Semantics

The semantics of the flow facts are defined as *restrictions* on the flow of a program. We define the exact restriction by reasoning over all possible executions of a program.

An *execution path* is a sequence of nodes and edges from the scope tree such that the nodes are connected by the edges. It begins at the entry node of the program, and ends at the exit. It might be infinite (if the program contains loops). A *subpath* is a consecutive segment of an execution path. A path p must *satisfy* all flow facts in order to be included in the set of possible executions.

Whether a path p satisfies a certain fact $f = scope : context : contraint$ is determined by finding all parts of p that correspond to the $scope : context$ specification and checking for constraint satisfaction in each. For example, a fact defined for an inner loop in a loop nest will need to be checked against every execution of the inner loop in p.

A path p enters the *cover* of a fact f when the iteration counter of the anchor scope becomes equal to the lower range limit of the anchor scope. A path p exits the cover of a fact when the execution enters a scope which is not a descendant of the anchor scope, or when the iteration counter of the anchor scope becomes larger than upper range limit of the anchor scope. For "all" facts, (`<>`,`[]`), the enter and exit of the fact cover occur when the defining scope is entered resp. exited (since no iteration numbers are given).

Within a cover of a fact we will count the number of occurrences of the nodes, edges, and sequences of nodes referred to in *constraint*. Note that a fact only

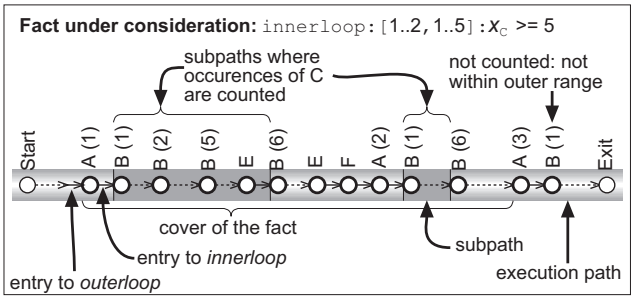


Figure 7. Example for fact semantics

can refer to variables belonging to entities within the defining scope, or the subscopes of the defining scope. The occurrences will only be counted when all iteration numbers between the anchor and defining scope are within the corresponding ranges specified by the fact.

The number of occurrences of each entity will be checked against the counts allowed in *constraint*. If the fact is a "total" fact the sum of all occurrences of each entity should be checked against the constraint. If the fact is a "foreach" fact we will further subdivide the number of occurrences of each entity so that we get a separate sum for each iteration of the defining scope in the cover. Each such sum is then checked against the constraint. This provides iteration-local semantics of *foreach* facts.

A single iteration of a scope is defined as beginning when the scope is entered or the header node executed, and ending the next time the header node is executed, or the scope is exited to a scope above it. For unstructured loops, there might be an iteration zero, since the loop can be entered without immediately passing the header node.

Figure 7 shows an example of a multidimensional fact defined for the scope tree shown in Figure 3. Dashed arrows represents (long) sequences of nodes and edges, and solid arrows a single edge from the scope tree.

The cover of the `innerloop:[1..2,1..5]`$: x_{\mathtt{C}} \geq 5$ fact starts with the first iteration in the outer scope, and ends when the third iteration of the outer scope begins. Inside the cover, we find the subpaths where the iteration numbers are within all the ranges of the fact. Inside these subpaths, the number of occurences of node `C` are counted. Since the fact has "total" semantics, we check this sum check against the constraint.

Had the fact used "foreach" semantics, each iteration of the inner loop would be counted by itself.

4.1.4. Examples

The use of constraints to describe program flow is well-known from the implicit path-enumeration technique (IPET) [10, 14, 20, 23]. The use of context specifications, however, makes this approach much more

powerful than previous methods.

For example, we can specify information for each iteration of a loop, by using one fact for each iteration. Information related to certain paths through a loop can be expressed using constraints on sequences of nodes. The specification of information locally in a scope is much more convenient than previous global approaches.

Some examples of types of flows that can be described are:

- A simple loop bound is specified by using an "all/total" operator: $scope$: `[]` : $x_{header(scope)} \leq bound$. If the header of the loop contains the loop exit check (e.g. `for`-loops), the *bound* is the number of loops+1.
- $scope$: `<>` : $x_{\texttt{A}} + x_{\texttt{B}} = 1$ The nodes `A` and `B` cannot execute on the same iteration of the scope. Corresponds to Park's `nopath(A,B)` annotation [21] and implies an *infeasible* or *false* path.
- $scope$: `<>` : $x_{\texttt{A}} = x_{\texttt{B}}$ The nodes `A` and `B` execute on the same iterations of the scope. Corresponds to Park's `samepath(A,B)` annotation [21].
- The *loop sequence* annotations of the MARS system [22] is used to indicate that a number of successive loops shares a total number of iterations. It is expressed as $scope$: `<>` : $x_{header(loop1)} + x_{header(loop2)} = total$, where $scope$ is the scope containing the sequenced loops and *total* is the total number of iterations they share.
- $scope$: `<>` : $x_{\texttt{A}} \leq x_{\texttt{B}}$ For every iteration of the scope an execution of `A` *implies* an execution of `B` (node `B` can still be executed on its own).
- $scope$: `<5..10>` : $x_{\texttt{A}} = 1$ Node `A` must execute on all the iterations numbered 5 thru 10. This type of iteration-based facts can be derived using flow analysis methods like [9, 12].
- $scope$: `[1..2,1..10]` : $x_{\texttt{A}} \leq 2$ Node `A` cannot be executed more than twice over the first ten iterations of the inner scope for the first two iterations of the outer scope.

5. Conversion of flow information to IPET Calculation

Our WCET tool uses the IPET calculation method to find the WCET and thus we need to convert the flow information facts to a form appropriate for IPET.

5.1. IPET Calculation

In IPET the flow of a program is modeled as an assignment of values to execution count variables. The variables are considered *global*, and the values reflect the total number of executions of each node for each execution of the program.

Each entity with a count variable also has a time variable (t_{entity}) giving the contribution of that part of the program to the total execution time for each time it is executed. As shown in our previous work, the modeling method can handle both individual execution scenario nodes and sequences of nodes [7].

The flow possible given the structure of the program is modeled using *structural constraints*. For each node, the sum of the incoming flows is equal to the outgoing flows. For example, for node `B` in Figure 3 the constraints $x_{\texttt{AB}} + x_{\texttt{EB}} = x_{\texttt{B}}$ and $x_{\texttt{B}} = x_{\texttt{BC}} + x_{\texttt{BD}}$ would be generated. The structural constraints are implicit and need not be described by the flow facts.

In addition to the structural constraints, we have constraints given by the flow information facts. We also generate additional constraints for node sequences, as shown in [7].

The WCET estimate is generated by maximizing the sum of the products of the execution counts and execution times (subject to the flow constraints):

$$WCET = maximize(\sum_{\forall entity} x_{entity} \cdot t_{entity})$$

This maximization problem is then solved using a constraint solver or integer linear programming (ILP) system.

Observe that IPET will not explicitly find the worst case execution path but just give the worst case count on each node. There is no information about the precise execution order.

5.2. Conversion

The problem in converting flow facts to IPET constraints is that our specification language uses *scope-local* semantics, while IPET relies on *execution-global* variable values: all instances of a certain entity across an the entire execution path is counted. We need to join all the local information we have about the flow into a consistent set of constraints reasoning about the global variables.

For example, consider the loop nest shown in Figure 3. A fact such as `innerloop` : `[]` : $x_{\texttt{C}} \leq 10$, stating that each time the `innerloop` is entered, block `C` will be executed at most 10 times. Thus, globally, we need to constrain $x_{\texttt{C}}$ by $x_{\texttt{C}} \leq 10 * x_{entry(innerloop)}$, and just not $x_{\texttt{C}} \leq 10$. Performing this raising of facts to the global level is the work of our conversion algorithms.

5.3. Converting "All" Facts

For facts with an "all/total" context specification, (`[]`), the conversion is simply to multiply all constants in the constraint by the $x_{entry(scope)}$ of the defining

```
for each fact s:c:constr in Facts do
  if c == <> then    // Convert foreach fact
    add constr[constants * x_header(c)] to Csp
    remove s:c:constr from Facts
  if c == [] then    // Convert total fact
    add constr[constants * x_entry(c)] to Csp
    remove s:c:constr from Facts
  else Ranged fact, save it for later
    let s:c:constr remain unchanged in Facts
end for
```

Figure 8. "All" Conversion

scope. By a constant we mean an integer not part of any multiplication or division.

For "all/foreach" facts, (<>), we need to consider the number of iterations executed by the scope. This is equal to the number of times the header node of the scope is executed, and thus the conversion is to multiply all constants by $x_{header(scope)}$. For unstructured scopes, we must use $x_{header(scope)} + 1$, since there exists an iteration where the header node does not get executed.

Figure 8 shows the algorithm for converting the "all" facts. The notation *constr*[constants $* x$] indicates that all constants in the constraint *constr* are multiplied by the variable (or constant) x. The constraints generated are put into the set Csp. A constant is defined as an expression containing no variables.

5.4. Converting "Range" Facts

The complex case is the range facts, since they only provide *partial* information about the value of a variable. For example, given the fact `foo : [1..4]` : $x_{\texttt{A}} \leq 2$ in Figure 3, we cannot simply generate the constraint $x_{\texttt{A}} \leq 2 * x_{entry(\texttt{foo})}$, since the node `A` might be executed outside the specified range.

The algorithm shown in Figure 9 solves this problem by treating each range as a *virtual scope*. A virtual scope is defined by *scope* : *range* (two identical ranges in different scopes generate different virtual scopes). The ranges are specified without context type. Each virtual scope has a header and entry variable, (x^{vscope}_{entry} and x^{vscope}_{header}), together with subvariables x^{vscope}_{entity} which holds the number of times entities, like nodes and edges, gets executed within the virtual scope. For example, the fact `s : [1..5]` : $x_{\texttt{A}} \geq 3$ would generate a subvariable named $x^{s:1..5}_{\texttt{A}}$, an entry variable $x^{s:1..5}_{entry}$ and a header variable $x^{s:1..5}_{header}$.

Each virtual scope is then treated as an "all" scope, but with subvariables instead of the main variables, using the same rules for inserting references to entry and header node variables. The following sections outlines how the generated virtual scope variables are constrained and related to the global IPET variables.

```
for each scope s in ScopeTree do
  F_s := all facts in ScopeTree attached to s
  a := s, F_l := ∅
  while F_s ≠ ∅ do
    // Convert facts with same anchor scope a together
    F_a := all facts in F_s with anchor = a
    F_a := F_a + F_l // Add lifted facts
    F_l := F_a, F_s := F_s - F_a
    // Generate set of non-intersecting rangelists
    R := all rangelists among facts in F_a
    R_sub := split R into non-overlapping rangelists
    R_sub := R_sub + rangelists not covered by R_sub
    // Relate global vars to subrange vars
    for each variable v among facts in F_a do
      add v = Σ_{r∈R_sub} v^{s:r} to Csp
    end for
    add x_header(s) = Σ_{r∈R_sub} x^{s:r}_header to Csp
    add x_entry(s) = x^{s:smallest(R_sub)}_entry x to Csp
    // Create constraints from facts
    for each fact s:c:constr in F_a do
      R_c := all ranges in R_sub covered by c
      for each variable v in constr do
        constr := constr[v replaced by Σ_{r∈R_c} v^{s:r}]
      end for
      if is_total(c) then
        add constr[constants * x^{s:smallest(R_c)}_entry] to Csp
      if is_foreach(c) then
        add constr[constants *Σ_{r∈R_c} x^{s:r}_header] to Csp
    end for
    // Constrain subrange vars and header vars
    for each rangelist rl in R_sub do
      for each variable v among facts in F_a do
        m := 1
        if v belongs to scope lower than s then
          m := multiply sizes of lower scopes
        add v^{s:rl} ≤ x^{s:rl}_header * m to Csp
      end for
      add x^{s:rl}_header ≤ sizeof(rl) * x^{s:rl}_entry to Csp
      add x^{s:rl}_header ≥ x^{s:rl}_entry to Csp
    end for
    // Constrain ordering between subrange entry vars
    for all rangelists rl_i and rl_j in R_sub do
      if immediate_predecessor(rl_i,rl_j) then
        add (sizeof(rl_i) - 1) * x^{s:rl_j}_entry ≤ x^{s:rl_i}_header - x^{s:rl_i}_entry
        and x^{s:rl_i}_entry ≥ x^{s:rl_j}_entry to Csp
    end for
    // Lift facts to next anchor scope
    u := upperbound(parent_scope(a))
    for each fact s:c:constr in F_l do
      RList := rangelist(c)
      if is_total(c) then
        R_c := all ranges in R_sub covered by c
        c := [1..u,RList]
        constr := constr[constants * x^{s:smallest(R_c)}_entry]
      if is_foreach(c) then
        c := <1..u,RList>
    end for
    update s:c:constr in F_l
  end while
end for
```

Figure 9. "Range" Conversion

5.4.1. Virtual Scope Variables

The algorithm loops over all scopes, and for each scope, over all anchor scopes for the facts attached to

the scope. For each anchor scope, we collect the set of ranges (potentially multidimensional) used in the facts with the given anchor scope.

In the case that several ranges for the same scope overlap, the ranges are split into disjoint subranges. For example, for the facts $\texttt{s:[1..5]}:x_{\texttt{A}} \geq 3$ and $\texttt{s:[3..7]}:x_{\texttt{A}} \leq x_{\texttt{B}}$, we get the three virtual scopes s:1..2, s:3..5, and s:6..7. The subvariables for each range are then replaced with the sum of the "split" variables, for example $x_{\texttt{A}}^{s:1..5} = x_{\texttt{A}}^{s:1..2} + x_{\texttt{A}}^{s:3..5}$ and $x_{\texttt{A}}^{s:3..7} = x_{\texttt{A}}^{s:3..5} + x_{\texttt{A}}^{s:6..7}$.

In the case that the ranges in the facts don't cover the complete iteration space of the scope, extra virtual scopes are created to fill the gaps, so that every iteration number from 1 to the upper bound of the scope is covered. E.g. assuming the scope `s` has an upper bound of 20 we get the set of virtual scopes {s:1..2, s:3..5, s:6..7, s:8..20}.

In the special case that there exists an iteration zero (i.e. unstructured loops) it is communicated to the conversion algorithm by having a range with zero as its lower bound, since that is going to be used to construct the set of subranges.

The value of each global variable is set equal to the sum of the corresponding subvariables, e.g. in our previous example $x_{\texttt{A}} = x_{\texttt{A}}^{s:1..2} + x_{\texttt{A}}^{s:3..5} + x_{\texttt{A}}^{s:6..7} + x_{\texttt{A}}^{s:8..20}$. This ties the virtual scope variables to the real, global, IPET variables.

The facts are converted to constraint by replacing all variables with the sum of virtual scope variables corresponding to the range of the fact, in a way analogous to the "all" facts. For "range/total" facts, ([*RList*]), the constants are multiplied by the header count variable, and for "range/foreach" facts, (<*RList*>), the constants are multiplied by the entry count variable.

5.4.2. Constraining Virtual Scope Variables

All subvariables must constrained so that they cannot contribute more than the number of iterations of the virtual scope to the total sum. For an entity in the defining scope, this is done by constraining the variable by the header node for the virtual scope (i.e. $x_{entity}^{vscope} \leq x_{header}^{vscope}$). For subvariables belonging to scopes below the defining scope, we must multiply the subheader variable by the iteration bounds for all the intervening scopes, since we would otherwise incorrectly constrain iterations in lower scopes.

Lower and upper bounds are generated for the subheaders, in order to constrain how much the subvariables of each virtual scope can contribute to the total. The lower bound is given by $x_{header}^{vscope} \geq x_{entry}^{vscope}$, stating that we must take at least one iteration in the scope each time the virtual scope is entered (for scopes with a range starting a zero, no constraint is generated). An upper bound is provided by $x_{header}^{vscope} \leq \texttt{sizeof}(vscope) * x_{entry}^{vscope}$. This states that for every entry we cannot execute more iterations than the size of the range for the virtual scope. The size of a range is the number of iterations covered by the range. E.g. the virtual scope s : 3..5 has the size 3, generating the constraints $x_{header}^{s:3..5} \geq x_{entry}^{s:3..5}$ and $x_{header}^{s:3..5} \leq 3 * x_{entry}^{s:3..5}$.

The entry variables must be constrained to ensure that each virtual scope is only executed if the previous virtual scope is executed long enough to come to the start of the virtual scope. E.g. a virtual scope s:3..5 can not be entered unless the preceding virtual scope s:1..2 was entered and at least two iterations executed in it.

First, the constraint $x_{entry}^{prev} \geq x_{entry}^{next}$ is generated, where *prev* is the virtual scope preceding *next*: we cannot enter a subrange more often than its preceding subrange.

To ensure that we only enter a subrange if the preceding subrange has executed all its iterations, the constraint $(\texttt{sizeof}(prev) - 1) * x_{entry}^{next} \leq x_{header}^{prev} - x_{entry}^{prev}$ is used. The idea is that we need to "fill up" the preceding subrange before continuing into the next subrange, and that we need to compensate for the cases where we do not. Notice that the constraint both provides a lower bound for x_{header}^{prev} and an upper bound for x_{entry}^{next}.

The constraint is based on the observation that executions of x_{header}^{prev} can be divided into those where *next* get entered and those where not. The number of iterations spent leading up to *next* is $\texttt{sizeof}(prev) * x_{entry}^{next}$.

The number of iterations spent not leading up to *next* is derived from the entry variables: $x_{entry}^{prev} - x_{entry}^{next}$ is the number of entries of *prev* which didn't lead to an entry of *next*. This provides a lower bound (for each such entry we must iterate at least once in *prev*). The upper bound is $(\texttt{sizeof}(prev) * (x_{entry}^{prev} - x_{entry}^{next})$ (each entry iterates the whole iteration space of *prev* but does not enter *next*). After some algebraic manipulations the lower bound becomes the constraint above, and the upper bound is the same as $x_{header}^{prev} \leq x_{entry}^{prev} * \texttt{sizeof}(prev)$, which is already given.

The entry variable for a range is equal to the entry variable for the (split) subrange entry variable corresponding to the first covered subrange, since when that subrange is entered, the range itself is entered. I.e. $x_{entry}^{s:3..7} = x_{entry}^{s:3..5}$. In the same way we can constrain the global entry variable of a scope with the first subrange, i.e. $x_{entry(s)} = x_{entry}^{s:1..2}$. The function `smallest()` returns the first smallest subrange among a set of subranges.

5.4.3. Multidimensional Ranges

Generating constraints for multidimensional ranges requires some extra consideration. When creating disjoint virtual scopes all dimensions of the ranges have to be taken into consideration. For example, the facts $\texttt{s:[1..5,3..10]} : x_{\texttt{C}} \geq 12$ and $\texttt{s:[6..10,1..7]} : x_{\texttt{C}} \leq x_{\texttt{D}+2}$ would generate the following set of virtual scopes (assuming that 10 is the upper bound for both s and its parent scope p): $\{s{:}1..5, 1..2,\ s{:}1..5, 3..7,\ s{:}1..5, 8..10,\ s{:}6..10, 1..2,\ s{:}6..10, 3..7,\ s{:}6..10, 8..10\}$.

The size of a multidimensional range is the product of the sizes of all the ranges in it (e.g. $\texttt{sizeof}(s{:}1..5, 3..7) = 25$). The first virtual scope among a set of virtual scopes is the one which has all the smallest ranges, (i.e. $s{:}1..5, 1..2$ in the above example).

The relation between entries becomes more complex, since we do not have a linear order between virtual scopes. For example, the number of times we are entering $s{:}6..10, 1..2$ can not be constrained by $s{:}1..5, 1..2$ since we don't have to enter s during the first five iterations of p to be able to enter s during the remaining iterations of p.

Instead, we get an ordering between entry variables which have identical subranges except for the defining scope. E.g. the above example gives: $x_{entry}^{s:1..5,1..2} \geq x_{entry}^{s:1..5,3..7} \geq x_{entry}^{s:1..5,8..10}$ and $x_{entry}^{s:6..10,1..2} \geq x_{entry}^{s:6..10,3..7} \geq x_{entry}^{s:6..10,8..10}$. Also, the first entry variable in the chain can be constrained by the number of times the corresponding range for just the parent scope gets entered. This gives us: $x_{entry}^{p:1..5} \geq x_{entry}^{s:1..5,1..2}$ and $x_{entry}^{p:6..10} \geq x_{entry}^{s:6..10,1..2}$. Finally, the parent scope entry variables are then related to each other and to the global entry variable: $x_{entry(p)} = x_{entry}^{p:1..5} \geq x_{entry}^{p:6..10}$. Thus, we have a partial order which lets us relate all entries of all generated virtual scopes to the global entry variables.

We also need to handle the problem with facts of different dimension but with the same defining scope are interacting with each other. For example, $\texttt{s:[3..7]} : x_{\texttt{A}} \leq 3$ interacts with $\texttt{s:[1..2,1..4]} : x_{\texttt{A}} \geq 5$. The solution is to *lift* the facts defined for lower anchor scopes to the higher scopes. This is done by prefixing a new range to the context specification of the fact, which includes the whole iterations space of the new anchor scope. The fact should only be valid if the original defining scope of the fact is entered, and therefore all constants in the constraint is multiplied with the entry variable of original fact.

Lifting makes the set of constraints for a certain scope and anchor scope consistent. Note it also expands the set of facts defined for a certain scope.

```
acc_length = 18, len = 35, ptr = start + 17, end = start + 700;
for(i=0;i<700;i++) { /* Outer loop */
  for(j=1;j<acc_length;j++) /* Inner loop */
    { /* calculation */ }
  if(ptr == end)
    acc_length--; /* node A */
  else {
    if(acc_length < len)
      acc_length++; /* node B */
    ptr++; /* node C */
    }
} /* end of loop */
```

Basic finiteness	$outer : \texttt{[]} : x_{header(outer)} \leq 701$
	$inner : \texttt{[]} : x_{header(inner)} \leq 35$
A taken last 18	$outer : \texttt{<683..700>} : x_A = 1$
B taken first 17	$outer : \texttt{<1..17>} : x_B = 1$
C taken first 682	$outer : \texttt{<1..682>} : x_C = 1$
Precise inner loop	$outer : \texttt{[1..17]} : x_{header(inner)} = 442$
	$outer : \texttt{<18..683>} : x_{header(inner)} = 35$
	$outer : \texttt{[684..700]} : x_{header(inner)} = 442$

Figure 10. `fir` kernel with facts

5.5. Conversion Example

For an example on how the conversion process works, consider the code given in Figure 10. It shows the flow-relevant parts of the kernel of a DSP algorithm, and has the flow as shown below the code. Note that since we are not doing any heavy compiler optimizations in our experiments, the structure of the source code and object code are the same.

The (header of) inner loop executes $(18, 19 \ldots 34, 35 \ldots 35, 34 \ldots 19, 18)$ times over the iterations of the outer loop. This gives a basic finiteness bound of 35, and the more precise loop bounds given in the lower section of the figure. The ramp-up and ramp-down each contains a total of $\sum_{i=18}^{34} i = 442$ iterations, and are given one fact each.

The conversion to constraints using our algorithm is illustrated in Figure 11. (A) shows the virtual scopes resulting from the split operation. Note the virtual scope $outer{:}\texttt{701..701}$ which models the last execution of the header (the exit from the loop). (B) and (C) shows the generated subentry variables and subheader variables and (D) shows the constraints generated between them.

Figure 11(E) shows the subvariables for the `C` node in the `fir` code, and how they are constrained by the main variable for x_C and the subheaders. Finally, (F) shows how a flow fact was converted.

6. Our WCET Analysis Tool

We have used the low-level module of our WCET tool [7] to perform experiments with our flow facts.

Figure 12 gives an overview of our WCET analysis system as implemented today. In order to generate a WCET estimate, a program is processed through a number of modules:

The *compiler* is a modified IAR V850/V850E

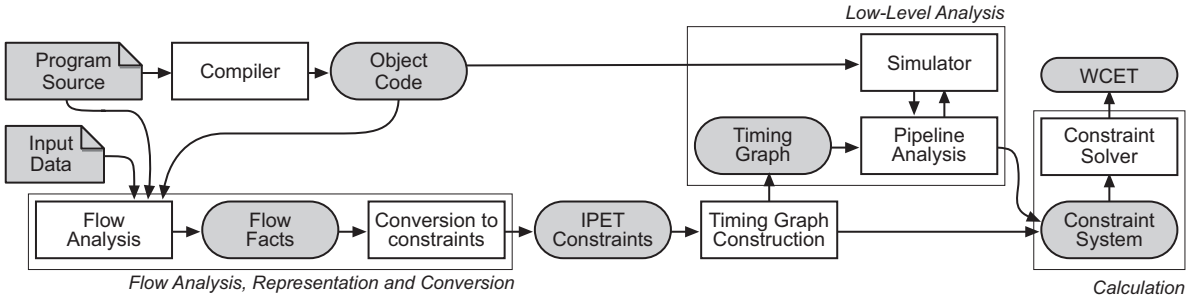


Figure 12. Overview of our current WCET analysis system

A: Disjunct covering virtual scopes

$outer$:1..17, $outer$:18..682, $outer$:683..683, $outer$:684..700, $outer$:701..701

B: Subentry variables

$$x_{entry(outer)} \geq x_{entry}^{outer:1..17} \geq x_{entry}^{outer:18..682} \geq x_{entry}^{outer:683..683} \geq x_{entry}^{outer:684..700} \geq x_{entry}^{outer:701..701}$$

C: Subheader variables

$$x_{header(outer)} = x_{header}^{outer:1..17} + x_{header}^{outer:18..682} + x_{header}^{outer:683..683} + x_{header}^{outer:684..700} + x_{header}^{outer:701..701}$$

D: Relation between subheader and subentry variables

$$x_{header}^{outer:1..17} \leq 17 * x_{entry}^{outer:1..17} \qquad x_{header}^{outer:1..17} \geq x_{entry}^{outer:1..17}$$
$$x_{header}^{outer:18..682} \leq 665 * x_{entry}^{outer:18..682} \qquad x_{header}^{outer:18..682} \geq x_{entry}^{outer:18..682}$$
$$x_{header}^{outer:683..683} \leq 1 * x_{entry}^{outer:683..683} \qquad x_{header}^{outer:683..683} \geq x_{entry}^{outer:683..683}$$
$$x_{header}^{outer:684..700} \leq 17 * x_{entry}^{outer:684..700} \qquad x_{header}^{outer:684..700} \geq x_{entry}^{outer:684..700}$$
$$x_{header}^{outer:701..701} \leq 1 * x_{entry}^{outer:701..701} \qquad x_{header}^{outer:701..701} \geq x_{entry}^{outer:701..701}$$
$$(17-1) * x_{entry}^{outer:18..682} \leq x_{header}^{outer:1..17} - x_{entry}^{outer:1..17}$$
$$(665-1) * x_{entry}^{outer:683..683} \leq x_{header}^{outer:18..682} - x_{entry}^{outer:18..682}$$
$$(1-1) * x_{entry}^{outer:684..700} \leq x_{header}^{outer:683..683} - x_{entry}^{outer:683..683}$$
$$(17-1) * x_{entry}^{outer:701..701} \leq x_{header}^{outer:684..700} - x_{entry}^{outer:684..700}$$

E: Subvariables and constraints for variable x_C

$$x_C = x_C^{outer:1..17} + x_C^{outer:18..682} + x_C^{outer:683..683} + x_C^{outer:684..700} + x_C^{outer:701..701}$$
$$x_C^{outer:1..17} \leq x_{header}^{outer:1..17}$$
$$x_C^{outer:18..682} \leq x_{header}^{outer:18..682}$$
$$x_C^{outer:683..683} \leq x_{header}^{outer:683..683}$$
$$x_C^{outer:684..700} \leq x_{header}^{outer:684..700}$$
$$x_C^{outer:701..701} \leq x_{header}^{outer:701..701}$$

F: Conversion of fact $outer$: <1..682> : $x_C = 1$

$$x_C^{outer:1..17} + x_C^{outer:18..682} = 682 * (x_{header}^{outer:1..17} + x_{header}^{outer:18..682})$$

Figure 11. Example of Conversion

C/Embedded C++ [26] compiler which emits the object code of the program in an accessible format.

The *program flow analysis* is performed manually at present. It results in a description of the possible program flow using our scopes-and-facts representation. The information facts are converted to IPET constraints as described above.

We then construct a *timing graph*, which contains the same execution scenario nodes and edges as the scope tree, but without the scopes. In addition to the execution count variables, the edges and nodes in the timing graph are given *execution time* variables.

The *Pipeline Analysis* runs nodes and sequences of nodes in the simulator, and calculates the values of the execution time variables accordingly.

The *Simulator* is fed instructions together with information about how the instructions execute (provided by the execution scenarios). It must have a cycle-accurate model of the CPU, but it does not need to model the semantics of the object code.

Times for nodes correspond to the execution time of a basic block in isolation, and times for sequences to the effect of the processor pipeline when the basic blocks are executed in sequence (usually an overlap). Timing effects for sequences of nodes are calculated by first running the individual nodes in the simulator, and then the sequence and comparing the execution times. For details see [7].

We do not include any cache or branch predictors in the current version of the tool, since our target system does not have a such features. The analysis method could easily be extended to include the effects of such features as outlined in [8].

For this set of experiments, the target system was the NEC V850E [4] 32-bit embedded microcontroller. We have designed our own cycle-accurate simulator for the V850E core using a generic framework, and we plan to support other architectures in the future.

7. Experiments

In order to demonstrate the effectiveness of our specification language and conversion to IPET, we have performed a number of experiments on example programs containing various types of flow.

The execution times in our experiments are shown in Figure 13. The column *Basic* gives the WCET estimate using only basic finiteness constraints. The column *Improved* gives the estimate resulting from adding more flow information facts. The 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.

The better results for the "improved" column indi-

Program	Basic Cycles	Basic +%	Improved Cycles	Improved +%	Actual
`fir`	326967	1.1	323277	0.0	323277
`insertsort`	2077	66.3	1249	0.0	1249
`duff`	1248	1.8	1226	0.0	1226
`jfdctint`	5550	0.0	5550	0.0	5550

Figure 13. Execution Time Estimates

```
i = 2;
while(i<=10) { /* Outer loop */
  j = i;
  while (a[j] < a[j-1]) { /* Inner loop */
    swap(a[j],a[j-1]);
    j--;
  }
  i++;
}
```

Basic finiteness	$outer : [] : x_{header(outer)} \leq 9$
	$inner : [] : x_{header(inner)} \leq 9$
Triangular	$outer : [] : x_{header(inner)} \leq 45$

Figure 14. Triangular loop: `insertsort`

cates the advantage of using more complex flows that simple loop bounds in WCET analysis, and shows that our method is able to capture complex flows in an effective and efficient manner.

We choose to present a few example programs in detail in order to give an understanding of how our flow modeling works. The `fir` program was described above, and `insertsort` and `duff` are presented below. For `jfdctint` simple loop bounds are sufficient.

7.1. InsertSort

The key problem in insertion sort is a triangular loop, i.e. a loop where the number of iterations of the inner loop depends on the iteration of the outer loop. Figure 14) shows the kernel of the code (from [15]).

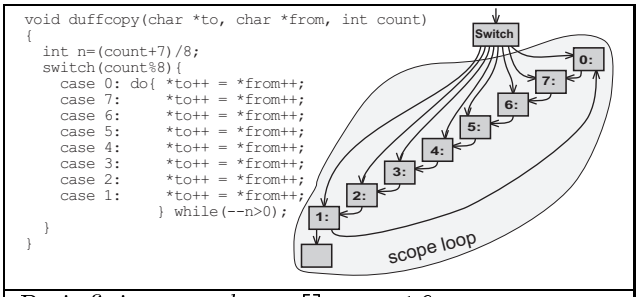
The loop iterates nine times in the outer loop, and, assuming worst-case input, at most nine times in the inner loop. This gives the basic finiteness facts.

However, we can deduce that the inner loop will not execute more than $1 + 2 + \ldots + 9 = 45$ times for each entry to the outer loop. This constant represents the same information as the results of the summation formulae used to determine the number of iteration of nested loops in [3, 11].

7.2. Duff

The core of `duff` (see Figure 15) is a loop with multiple entry points. It is a benevolent case of unstructured code. We use this example to show that our representation and conversion mechanisms are capable of handling more than just structured loops.

For a multiple-entry loop, the definition of a header given above allows *any* of the nodes in the loop be the header node. We use the *last* node in the loop as the



Basic finiteness	$loop : [] : x_{1:} \leq 6$
Exact bound	$loop : [] : x_{0:} + x_{7:} + x_{6:} + x_{5:}$ $+x_{4:} + x_{3:} + x_{2:} + x_{1:} = 43$

Figure 15. Multiple-entry loop: `duff`

header, since it is always executed when the loop is executed.

In our test, we use `duffcopy` to copy 43 bytes. The basic finiteness constraint is that the loop iterates 6 times, and for greater precision, we sum over the execution of all the nodes in the loop.

8. Conclusions

This paper has presented a new representation for program flow information. The representation is strong enough to handle the types of flow considered by previous WCET research, and also some additional types of flow (most notably unstructured code).

The representation allows facts to be presented both over all iterations and over some specific iterations of a loop, where previous approaches have choosen one particular style. It is compact yet expressive.

We have given an algorithm for converting the flow information facts to a form suitable for IPET WCET calculation. Without such a conversion, the facts would not be useful for WCET analysis.

The conversion algorithm and expressiveness of the representation have been demonstrated using a set of example programs. In order to further the understanding for our representation, we have given detailed flow information for a number of programs.

As a side effect, the tighter execution times obtained show that the IPET calculation method can take advantage of complex flow information.

9. Future Work

Our research strategy has been to start from the machine modeling [7] and work upwards in the tool chain. The next step is to extend our machine modeling to include caches and to investigate automated program flow analysis. We plan to investigate how cache and branch-prediction analysis can take advantage of flow information.

Furthermore, we would like to investigate how flow information can be converted to be used in path-based or tree-based calculation methods, and compare the effectiveness of various approaches.

The long term goal is to integrate a WCET analysis tool into the IAR Systems Embedded Workbench integrated development environment (`http://www.iar.com`), and thus provide WCET analysis as a standard and accessible tool for embedded real-time systems developers.

Acknowledgements: IAR Systems has provided the opportunity to work with the internals of their compilers and other development tools.

We would also like to thank Friedhelm Stappert and Mikael Sjödin for their fruitful comments on drafts of this article. The comments from the anonymous reviewers further improved the quality.

References

[1] R. Chapman, A. Burns, and A. Wellings. Integrated program proof and worst-case timing analysis of SPARK Ada. In *Proc. ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems (LCT-RTS'94)*, 1994.

[2] A. Colin and I. Puaut. Worst-case execution time analysis of the rtems real-time operating system. Technical Report Publication Interne No 1277, IRISA, November 1999.

[3] A. Colin and I. Puaut. Worst case execution time analysis for a processor with branch prediction. *Journal of Real-Time Systems*, May 2000.

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

[5] J. Engblom. Static properties of embedded real-time programs, and their implications for worst-case execution time analysis. In *Proc. 5^{th} IEEE Real-Time Technology and Applications Symposium (RTAS'99)*. IEEE Computer Society Press, June 1999.

[6] J. Engblom, P. Altenbernd, and A. Ermedahl. Facilitating worst-case execution times analysis for optimized code. In *Proc. of the 10^{th} Euromicro Workshop of Real-Time Systems*, pages 146–153, June 1998.

[7] J. Engblom and A. Ermedahl. Pipeline timing analysis using a trace-driven simulator. In *Proc. 6^{th} International Conference on Real-Time Computing Systems and Applications (RTCSA'99)*. IEEE Computer Society Press, December 1999.

[8] J. Engblom, A. Ermedahl, M. Sjödin, J. Gustafsson, and H. Hansson. Towards industry-strength worst case execution time analysis. Technical Report ASTEC 99/02, Advanced Software Technology Center (ASTEC), April 1999.

[9] A. Ermedahl and J. Gustafsson. Deriving annotations for tight calculation of execution time. In *Proc. Euro-Par'97 Parallel Processing, LNCS 1300*, pages 1298–1307. Springer Verlag, August 1997.

[10] 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.

[11] C. Healy, R. Arnold, F. Müller, D. Whalley, and M. Harmon. Bounding pipeline and instruction cache performance. *IEEE Transactions on Computers*, 48(1), January 1999.

[12] C. Healy, M. Sjödin, V. Rustagi, D. Whalley, and R. van Engelen. Supporting timing analysis by automatic bounding of loop iterations. *Journal of Real-Time Systems*, May 2000.

[13] S.-K. Kim, S. L. Min, and R. Ha. Efficient worst case timing analysis of data caching. In *Proc. of RTAS'96*, pages 230–240. IEEE, 1996.

[14] Y-T. S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. In *Proc. of the 32:nd Design Automation Conference*, pages 456–461, 1995.

[15] S.-S. Lim, Y. H. Bae, C. T. Jang, B.-D. Rhee, S. L. Min, C. Y. Park, H. Shin, K. Park, and C. S. Ki. An accurate worst-case timing analysis for risc processors. *IEEE Transactions on Software Engineering*, 21(7):593–604, July 1995.

[16] S.-S. Lim, J. H. Han, J. Kim, and S. L. Min. A worst case timing analysis technique for multiple-issue machines. In *Proc. 19^{th} IEEE Real-Time Systems Symposium (RTSS'98)*, December 1998.

[17] S-S. Lim, J. Kim, and S. L. Min. A worst case timing analysis technique for optimized programs. In *Proc. of the fifth International Conference on Real-Time Computing Systems and Applications (RTCSA); Hiroshima, Japan*, pages 151–157, Oct 1998.

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

[19] F. Müller. Timing predictions for multi-level caches. In *Proc. ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems (LCT-RTS'97)*, pages 29–36, Jun 1997.

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

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

[22] P. Puschner and C. Koza. Calculating the maximum execution time of real-time programs. *The Journal of Real-Time Systems*, 1(1):159–176, 1989.

[23] P. Puschner and A. Schedl. Computing maximum task execution times with linear programming techniques. Technical report, Technische Universität, Institut für Technische Informatik, Wien, April 1995.

[24] 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, May 1999.

[25] F. Stappert and P. Altenbernd. Complete worst-case execution time analysis of straight-line hard real-time programs. *Journal of Systems Architecture*, 46(4):339–355, 2000.

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

[27] R. White, F. Müller, C. Healy, D. Whalley, and M. Harmon. Timing analysis for data caches and set-associative caches. In *Proc. 3^{rd} IEEE Real-Time Technology and Applications Symposium (RTAS'97)*, pages 192–202, June 1997.