Modeling Complex Flows for Worst-Case Execution Time Analysis

Jakob Engblom\textsuperscript{*}
IAR Systems AB
Box 23051, SE-750 23 Uppsala, Sweden
email: jakob.engblom@iar.se

Andreas Ermedahl\textsuperscript{†}
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
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 of 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 [?].)

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...
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 in formation, 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.

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 subsopes 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.
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.

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 delimiters 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>: x0 = 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 (≤, =, ≥) 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 pro-
4.1.3. Detailed Fact Semantics

The semantics of the flow facts are defined as restrictions on the flow of the program. We define the exact conditions under which a fact is considered satisfied by a program. A fact is considered satisfied if all required conditions are met.

4.1.4. Examples

The use of constraints to describe program flow is well-known in the literature [10, 14, 21, 23]. The use of context-specific constraints to describe program flow is also well-known [10, 14, 21, 23].

Figure 7. Example of fact semantics

A fact can only refer to execution count variables in the defining scope and its subscopes.

Figure 8. Example of fact semantics

The value of a variable is equal to the sum of all the entry edge variables.

Figure 9. Example of fact semantics

The execution count of the headnode of a scope is used to limit the number of iterations of a certain scope. The execution count variable is called entry edge variable.

The execution count variable is equal to the sum of all the entry edge variables. Figure 3 shows an example of a scope with entry edge variables.
ifications, 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: \( \text{scope} : [] : x_{\text{header(scope)}} \leq \text{bound} \). If the header of the loop contains the loop exit check (e.g., \texttt{for}-loops), the \textit{bound} is the number of loops + 1.

- \( \text{scope} : <> : x_{A} + x_{B} \leq 1 \) The nodes A and B cannot execute on the same iteration of the scope. Corresponds to Park’s no\textup{path}(A,B) annotation [21] and implies an \textit{infeasible} or \textit{false} path.

- \( \text{scope} : <> : x_{A} = x_{B} \) The nodes A and B execute on the same iterations of the scope. Corresponds to Park’s same\textup{path}(A,B) annotation [21].

- The \textit{loop sequence} annotations of the MARS system [22] is used to indicate that a number of successive loops share a total number of iterations. It is expressed as \( \text{scope} : <> : x_{\text{header(loops1)}} + x_{\text{header(loops2)}} = \text{total} \), where \textit{scope} is the scope containing the sequenced loops and \textit{total} is the total number of iterations they share.

- \( \text{scope} : <> : x_{A} \leq x_{B} \) For every iteration of the scope an execution of A \textit{implies} an execution of B (node B can still be executed on its own).

- \( \text{scope} : <5..10> : x_{A} = 1 \) 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].

- \( \text{scope} : [1..2,1..10] : x_{A} \leq 2 \) 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 \textit{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 \( t_{\text{entity}} \) 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 \textit{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 \( x_{A} + x_{B} = x_{B} \) and \( x_{B} = x_{BC} + x_{BD} \) 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):

\[
WCET = \maximize \left( \sum_{\text{entity}} x_{\text{entry}} \cdot t_{\text{entry}} \right)
\]

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 \textit{scope-local} semantics, while IPET relies on \textit{execution-global} variable values: all instances of a certain entity across 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 \texttt{innerloop} : [] : \( x_{C} \leq 10 \), stating that each time the \texttt{innerloop} is entered, block C will be executed at most 10 times. Thus, globally, we need to constrain \( x_{C} \) by \( x_{C} \leq 10 \cdot x_{\text{entry(innerloop)}} \), and just not \( x_{C} \leq 10 \). 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
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 \( x_{\text{foo}} : [1..4] : x_1 \leq 2 \) in Figure 3, we cannot simply generate the constraint \( x_1 \leq 2 \times x_{\text{foo}} \), 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 \( \text{scope} : \text{range} \) (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, \( x_{\text{scope}} \) and \( x_{\text{header}} \), together with subvariables \( x_{\text{entity}} \) which holds the number of times entities, like nodes and edges, get executed within the virtual scope. For example, the fact \( x : [1..5] : x_3 \geq 3 \) would generate a subvariable named \( x_{3:1..5}^{1..5} \), an entry variable \( x_{3:1..5}^{1..5} \) and a header variable \( x_{3:1..5}^{1..5} \).

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.

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 \( s: [1..5]: x_1 \geq 3 \) and \( s: [3..7]: x_2 \leq x_3 \), 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 \( x^{3.5}_{1} = x^{1.2}_{1} + x^{3.5}_{1} \) and \( x^{3.7}_{1} = x^{3.5}_{1} + x^{6.7}_{1} + x^{8.20}_{1} \).

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 \( x_1 = x^{1.2}_{1} + x^{3.5}_{1} + x^{6.7}_{1} + x^{8.20}_{1} \). This ties the virtual scope variables to the real, global, IPET variables.

The facts are converted to constraints 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. \( x_{entry} \leq x_{scope} \)). 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 \( x_{scope} \geq x_{entry} \), 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 \( x_{scope} \leq \text{sizeof}(v_{scope}) \times x_{entry} \). 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 \( x^{s:3.5}_{entry} \geq x^{s:3.5}_{header} \) and \( x^{s:3.5}_{entry} \leq 3 \times x^{s:3.5}_{entry} \).

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 \( x^{prev}_{entry} \geq x^{next}_{entry} \) 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 \( \text{sizeof} (prev) - 1 ) \times x^{prev}_{entry} \leq x^{next}_{entry} - x^{prev}_{entry} \) is generated. 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 \( x^{prev}_{entry} \) and an upper bound for \( x^{next}_{entry} \).

The constraint is based on the observation that executions of \( x^{prev}_{header} \) can be divided into those where \( next \) get entered and those where not. The number of iterations spent leading up to \( next \) is exactly bounded by \( \text{sizeof}(prev) \times x^{next}_{entry} \).

The number of iterations spent not leading up to \( next \) is derived from the entry variables: \( x^{prev}_{entry} - x^{next}_{entry} \) 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 (\( \text{sizeof}(prev) \times (x^{prev}_{entry} - x^{next}_{entry}) \) (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 \( x^{prev}_{header} \leq x^{prev}_{entry} \times \text{sizeof}(prev) \), 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. \( x^{s:0.7}_{entry} = x^{s:3.5}_{entry} \). In the same way we can constrain the global entry variable of a scope with the first subrange, i.e. \( x^{s:entry(s)}_{entry} = x^{s:1.2}_{entry} \). The function \text{smallest()} returns the first 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 $s: [1..5, 3..10]: \text{set}_c \geq 12$ and $s: [6..10, 1..7]: \text{set}_c \leq x_2 + 2$ 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. $\text{sizeof}(s: 1..5, 3..7) = 25$). 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: $x_{\text{entry}}^{s: 1..5, 1..2} \geq x_{\text{entry}}^{s: 1..5, 3..7} \geq x_{\text{entry}}^{s: 1..5, 8..10}$ and $x_{\text{entry}}^{s: 6..10, 1..2} \geq x_{\text{entry}}^{s: 6..10, 3..7} \geq x_{\text{entry}}^{s: 6..10, 8..10}$. 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: $x_{\text{entry}}^{p: 1..5} \geq x_{\text{entry}}^{p: 1..5, 1..2}$ and $x_{\text{entry}}^{p: 6..10} \geq x_{\text{entry}}^{p: 6..10, 1..2}$. Finally, the parent scope entry variables are then related to each other and to the global entry variable: $x_{\text{entry}}(p) = x_{\text{entry}}^{p: 1..5} \geq x_{\text{entry}}^{p: 6..10}$. 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, $s: [3..7]: x_3 \leq 3$ interacts with $s: [1..2, 1..4]: x_3 \geq 5$. 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 ex-

Figure 10. fix kernel with facts

pands the set of facts defined for a certain scope.

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 $\sum_{i=18}^{1} i = 442$ 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 $\text{outer}: 701 \ldots 704$ 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 fix 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/V830E
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 caché 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-
**Figure 13. Execution Time Estimates**

<table>
<thead>
<tr>
<th>Program</th>
<th>Basic Cycles</th>
<th>Basic +%</th>
<th>Improved Cycles</th>
<th>Improved +%</th>
<th>Actual Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>fir</td>
<td>326967</td>
<td>1.1</td>
<td>323277</td>
<td>0.0</td>
<td>323277</td>
</tr>
<tr>
<td>_insertsort</td>
<td>2077</td>
<td>66.3</td>
<td>1249</td>
<td>0.0</td>
<td>1249</td>
</tr>
<tr>
<td>duff</td>
<td>1248</td>
<td>1.8</td>
<td>1226</td>
<td>0.0</td>
<td>1226</td>
</tr>
<tr>
<td>jfdctint</td>
<td>5550</td>
<td>0.0</td>
<td>5550</td>
<td>0.0</td>
<td>5550</td>
</tr>
</tbody>
</table>

**Figure 14. Triangular loop: insertsort**

calculates 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 + ... + 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 header node, 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 chosen 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


