Informations Traceability in Control Flow to Compute WCET with Compiler Optimizations

Internship at INRIA Rennes from 26th May 2014 to 24th August 2014

Benjamin Rouxel - benjamin.rouxel.1@etudiant.univ-rennes1.fr
Supervised by Isabelle Puaut - isabelle.puaut@irisa.fr

Abstract. Real-time systems are omnipresent in our life. Such kind of systems require more attention as human lifes depend on them. Designers need to compute the Worst Case Execution Time (WCET) in order to guarantee they respect their timing constraints. Many WCET techniques exist, the safest one is based on static analysis. This method analyzes the source code structure, and model a target architecture to compute the WCET.
To compute a WCET close to the reality, it must be done at the binary level (target platform modeling is easier at this level). Also, information flow are required to improve the precision of the WCET. Infeasible paths property is one of them. It may come from mutually exclusion between conditional branch in the execution flow of a program. Those properties are extracted at a high-level language design.
As most of real-time applications are compiled to C code, flow informations must be propagated through compiler steps to WCET estimation tools. But compilers propose to optimize the code's structure, and most of the flow information depends on the code structure. So infeasible paths properties become invalid when the compiler optimizes the code.
In order to reconcile real-time system developers, we created a framework to trace infeasible paths properties into compilers. It is designed to be fully optimizations independent t(if an optimization is added/removed or modified, our framework will still be working properly). The resulted WCET estimation are very encouraging, our traceability allowed to improve the WCET by around 30% in our test case.

1 Introduction

The need to respect timing constraints mark the difference between real-time systems and normal ones. Such systems must ensure they react on-time, not before, not after a predefined time. There exists two categories of real-time systems, soft and hard real-time system. For the former one it's tolerable to miss a deadline, only the quality of service will be degraded. For the latter one, missing a deadline could result in catastrophic consequences (e.g: human loss with airplane system). To ensure timing constraints in hard real-time system, it's mandatory to know the Worst Case Execution Time (WCET) of every real-time application. And the WCET must be as close as possible to the reality to improve precision.

However computing the WCET is very complex as this problem is equivalent as the halting problem from Turing, a program needs to know if another program will stop and how much time it will take to execute.

With the purpose of abstracting WCET computation and other timing constraints in real-time system, a new paradigm appeared: the synchronous programming. Synchronous applications are written in a high-level language [1] and are usually compiled into C-code before generated binary code. As timing constraints check is not anymore a matter of developers, compilers must provide the needed information to check them by other tools. So a full chain of traceability is needed between the different steps of the used compilers in order to propagate some properties from the high language to binary code.

One of the interesting property is the set of infeasible path in a program flow. It is usefulness comes from one of the technique used to compute the WCET which is based on finding the longest execution path in all possible execution paths of a program. As the set of paths can be huge even in simple application (the number of paths grows exponentially with tests and loops), it becomes crucial to reduce it [2]. This longest path will give us an upper bound of the execution time. However it's possible that this path is not feasible due to conditional branch contradiction. The detection of infeasible path will then evict this path and take the next longest one and so on until a longest feasible path is found.

However the infeasible path property is linked to a program structure at the time it is extracted. Due to a large set of optimizations compilers can radically modify the program structure, infeasible paths properties might no longer refer to existing paths. Also new infeasible paths could have been introduced by the compiler modifications on the program. So to obtain a more precise WCET, developers generally turn off optimizations during the compilation process. Nevertheless optimizations are important especially for real-time application on embedded system. Indeed such devices will have strong constraints around memory space (code size) and processor speed (number of instructions to execute, jumps, ..).

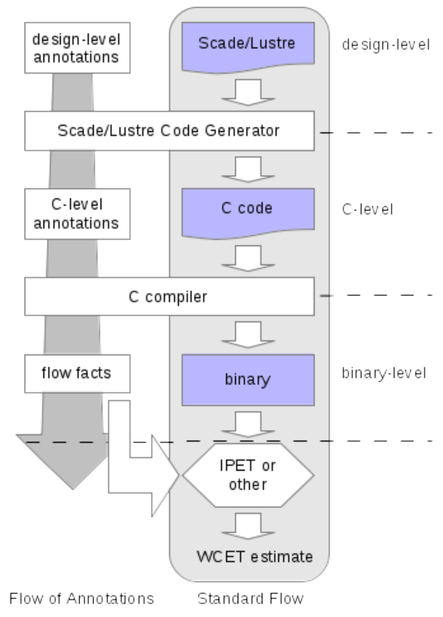


Fig. 1: Overall flow

This work is part of the W-SEPT ANR[1] project where the overall flow is presented by the figure 1. Its purpose is to estimate, as precisely as possible, a WCET from a synchronous program written in Lustre. As shown by figure 1 annotations (infeasible paths properties, loop bounds ..) are extracted from the design-level and must be propagated through C-level to binary-level in order to compute the WCET using some methods, here IPET [3]. Other parts of this project are studied at IRISA[2]/INRIA[3] and focus on another interesting property which is *loop bound*. Our work is complementary with Li's work [4] to compute a precise WCET in general cases.

The point of interest of our work is on traceability of infeasible paths properties into C-compiler optimizations (e.g: LLVM[4] tool chain) while minimizing the needed intervention into the compiler's code. We created a framework to react to modifications realized by the compiler's optimizations on the structure of a C-code generated from a synchronous program. The benefit of the presented approach is in optimizations independence while allowing any kind of optimizations to be activated by the compiler.

The structure of this document is as follows: Section 2 first introduces the needed background material on compilation and WCET estimation. Then Section 3 presents an overview on the past work related to this research. The theoretical part of the framework is introduced in Section 4. Section 5 gives an overview of the implementation and provides some experimental results. Finally Section 6 concludes the document by exposing some limits and possible improvements of our framework.

2 Background

This section presents all background material required to fully understand the rest of this document: some basic notions of compilation, synchronous programming, and WCET computation methods.

[1] Agence National de la Recherche
[2] Institut de Recherche en Informatique et Systèmes Aléatoires
[3] Institut National de Recherche en Informatique et en Automatique
[4] http://www.llvm.org

2.1 Compilation Methods

The compilation is the translation or transformation of a program from a source language (e.g:C, C++, ..) to a target language (e.g: assembly). Most compilers use an Intermediate Representation (IR). This is useful to improve the amount of supported programming language by compilers by only modifying the front-end (to add a source language), or the back-end (to add a target language). So most optimizations, transformations, analyses, etc.. are done on the IR.

For the life time of the compilation, the IR can be represented with different structure depending on the current needs.

Control Flow Graph (CFG) As a formal definition the CFG is a directed graph $G =< V, E >$, where V and E are respectively the set of nodes (basic blocks), and the set of edges. Figure 3b shows an example of a CFG. A basic block is a sequence of instructions ended by a terminator instruction (jump, return, etc..). An edge is a directed link between two basic blocks, so there is one In-Block and one Out-Block.

Passes and Optimizations Compilers are in charge of optimizing the generated code. Each optimization requires some analyses to be done before optimizing the code, and then applies some transformations on the IR. Such task are called passes.

More informations on compilation can be found in [5,6].

2.2 WCET Estimation

In order to respect timing constraints in real time system, the Worst Case Execution Time (WCET) and the Best Case Execution Time (BCET) must be known. It exists three main ways to compute them: *measurement-based* analysis, *static* analysis and *hybrid* methods. The whole aspect of the WCET's computation is referred in this survey [7].

Measurement-based analysis use tests. As in most cases, it is impossible to test all possibilities, the WCET/BCET computed through this manner will not be safe as it will be the WCET/BCET related to the input data used during tests and not in a general case.

Static methods are based on code structure and target platform model analysis. As they don't depend on data input, the computed WCET/BCET will be an upper bound for any possible execution. So this method is much more safe than the *measurement-based* one. Also, to obtain the tightest WCET possible the target platform is modeled to handle caches, pipelines, branch predictions ... . Such features can considerably modify the resulting WCET/BCET.

The presented work acts at the compilation level, and the preferred method is the static-based one because it is safer for hard real-time system.

IPET Implicit Path Enumeration Technique (IPET) [8] is a method used to build an Integer Linear Programming (ILP) system in order to compute the WCET from a program's CFG. The ILP system comprises equations of the form: $\sum_{p \in pred(B)} e_{pB} = \sum_{s \in succ(B)} e_{Bs}$ where B is a basic block, $pred(B)$ are the predecessors of B, $succ(B)$ are the successors of B. The equations' system is then solved by a solver like *lp_solve*[5] or *CPLEXSolver*[6]

[5] http://sourceforge.net/projects/lpsolve/

[6] http://www-01.ibm.com/software/commerce/optimization/cplex-optimizer/

Infeasible Path An infeasible path is a path in the CFG which can never occur in any possible execution of the program. A simple example is presented in figure 2, the dashed red line show the path that could never occur due to the two conditional branch in node A and node B which are contradictory. To extract infeasible paths, different methods are available. They are mostly based on a dynamic analysis and/or model checking [9,10]. The purpose of this property is to improve the quality of the WCET by making it tighter to the reality. Infeasible paths are introduced in the ILP system with equations of the form: $\sum_{e \in I} \leq n - 1$ where I is the set of edges in an infeasible path and n the number of edges in the set. This is a particular case when all edges are executed only ones, and that will be used in this study.

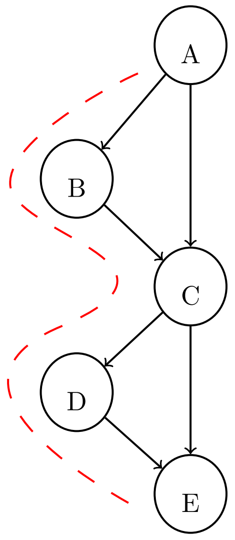


Fig. 2: Infeasible Path Example

2.3 Synchronous programming

The synchronous paradigm has been introduced in the early 80's. It was designed to help reactive state machine (real-time system) programmer by abstracting timing constraints from their point of view. A synchronous program can be seen as an automaton composed of a transition function and states. The state evolution is triggered by the data input arrival which occur at a precise time. The transition function is deterministic and each state must compute their data on time. So WCET has to be computed for every state transition to ensure, in an automatic way, that initial timing constraints are respected.

To facilitate the WCET computation, synchronous language compilers do not use unboundedness code structures : no dynamic memory allocation, no recursion, no unbounded loops, no goto ...[7]. The only existing loop is the main loop, which call the transition function to make a step. Thus the generated CFG will end up in a sequence of basic block, conditional/unconditional jumps and call instructions.

The approach in this document uses Lustre[8] as the synchronous programming language for the experiment. Then the Lustre program is compiled into C-code, which is then compiled to binary code through assembly. Our work take place into the C compiler and the WCET analysis is done at the binary level. So we will no longer directly speak about synchronous programming. For a full description of this programming paradigm in general and Lustre in particular see [1].

Figure 3a is an example of Lustre program and compilation taken from [11]. It shows an simple automaton for synchronous programming design. This program performs some computation depending on control signals (*onoff*, *toggle*). The data part is voluntary abstracted as its not the relevant part for our approach. We focus on the small arrows (*idle*, *low*, *high*) which act as activators of their under box. When the signal is true, the corresponding box compute the input data. What is important for the sequel, is the mutually exclusivity satisfied by controls, which then guarantee the (pairwise) exclusivity of activator signals.

Figure 3b shows a sample of the CFG from the transition function. The whole program CFG is too long to be drawn here, and as the presented part acts as a pattern which is repeated many times, this portion is enough to fully understand the next sections.

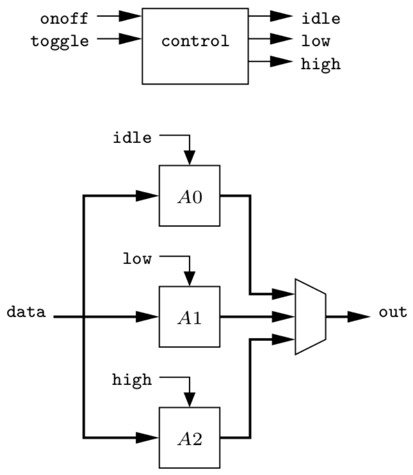
3 Related work

Three main methods exist to compute WCET : static-based, measurement-based, hybrid methods and plenty of tools [7]. Because we target hard real-time system, this work use static analyze and so needs flow information to apply the IPET technique [8].

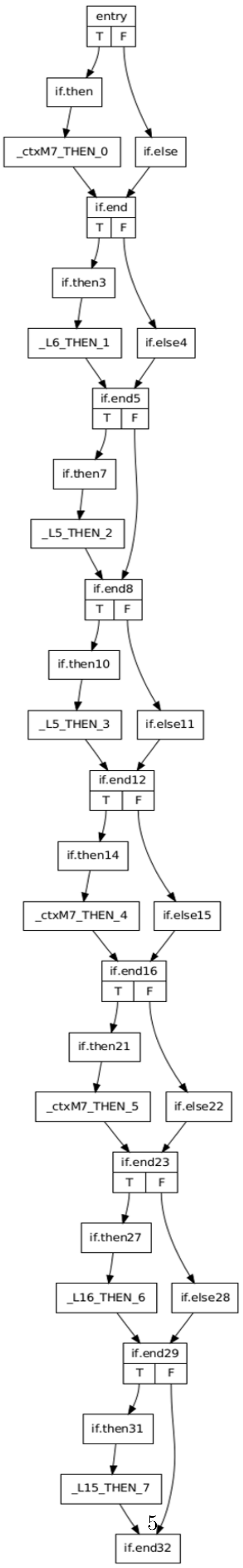
It exists two main flow information : paths that can not be executed (*infeasible paths*) and maximum number of loops' iteration (*loop-bound*). The latter one is the most important because if this flow information

[7] It exists static allocated array and *for-loops* to walk trough them. But those loops represent a deterministic execution path, so we can forget them

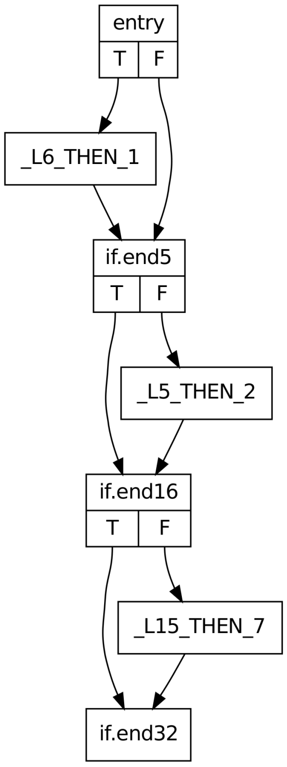
[8] http://www-verimag.imag.fr/Lustre-V6.html



(a) Example of Lustre schéma



(b) CFG corresponding to a sample of the generated C code



(c) CFG from 3b after optimizations

Fig. 3: Synchronous program example: automaton and CFG

is not available then the WCET is impossible to compute. So a lot of work has be done to determine and to exploit this property, [12,13,14,15] focus on different techniques to extract this information. Besides the extraction, Engblom and Ermedahl [16] presented an improvement of the IPET technique to improve the expressiveness of *loop-bound* constraints as they consider the WCET of the loop body may not be the same at each iteration. Li [4] et al. work on the same project as this study. As he is focusing on *loop-bound*, the presented work focus on *infeasible paths*. Nevertheless some work has also be done to extract and exploit infeasible paths when using IPET technique [10,15].

This work is part of the W-SEPT ANR[9] project[10]. As said, it comes in complement of Li et al. [4] for *loop-bound* traceability, and Raymond et al. [11] who also work on information traceability but has a fully different approach as he tries to match patterns on program's CFG while we react on CFG's update to propagate infeasible paths properties.

The infeasible path annotations are determined, in this study, from very high level language such as Lustre using dynamic approach [9] and model checking [17,10]. Then they must be propagated to the binary level, therefore through compiler optimizations. [18,19] studied the propagation of annotations through compiler optimizations on CFG. Their approaches are similar to each other in the way they studied every optimization to find a transition function to update the properties according to program modifications. This approach is quite time consuming because of the number of available optimizations (≥ 30) [5,6]. Also, their description of the optimization must be updated when the optimization semantic is. And when adding new optimization, they need some time to upgrade their system. Nevertheless this kind of approach is the most pertinent because properties are fully updated in accordance of the optimization semantic. So we are looking for a framework which is efficient while minimizing the amount of work when modifying the compiler behavior.

4 A Framework to trace Infeasible Paths Properties through Compiler Optimizations

The purpose of this framework is to allow the traceability of the given infeasible paths while the compiler is updating the CFG. Infeasible paths properties are extracted from the source program, then they are updated according to the compiler's optimizations. The ILP system is finally construct with the infeasible paths constraints in order to compute the WCET of the compiled program at the binary level.

Compare to [19,18], our framework is optimization independent. It is based on the knowledge of actions involving modifications of the CFG. So the base object composing the CFG must notify properties handler object, which will update them accordingly to the nature of the modification.

We now see our CFG as a quadruplet $G =< V, E, I, R >$ where V and E are still respectively the set of nodes and the set of edges. Then we add I as a set of infeasible path where $i \in I$ as : $i = (e1, e2) \in E^2$. The infeasible path i contains only two edges because our input properties only refer blocks by two, see subsection 5.2 for more details. R has the same structure as I but it will be use as a container for removed edge.

Starting from the definition of the CFG, we define what actions must be undertaken on infeasible path for any type of primitive modification done on the CFG:

- Connecting two basic blocks : adding an edge between two basic blocks
- Disconnecting two basic blocks : removing an edge between two basic blocks
- Modifying predecessors/successors list of an existing basic block : updating an edge between two basic blocks

All the presented modifications involve edges and no basic blocks adding/removing because such actions are strongly linked to edge's operations (e.g: adding a block is linked to connecting two basic blocks).

[9] Agence National de la Recherche
[10] http://wsept.inria.fr/

4.1 Connecting two Basic Blocks

When a basic block is added to the CFG, we have nothing to do the infeasible paths' set. This lack of propagation is due to our approach which is optimization independent. We are working at a low level so we don't know why this edge has been added. To have some actions to do we would need to know what would be the future actions, and so we would need to know what the optimization do.

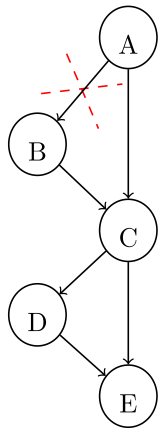
4.2 Disconnecting two known Basic Blocks

Disconnecting two basic blocks means removing an edge between two of them. So we observe 3 different cases going from the simplest one to the most tricky one which works because we don't have any back edge (no loop, no goto).

The absence of remove When removing an edge which is not part of any infeasible paths' properties, we trivially can say we have no change to make on the set I.

The regular remove When removing an edge which is part of an infeasible path, we will move the infeasible path from the set I to the set R. The infeasible path is not fully removed because a following CFG's modification may trigger an update action. And as shown in subsection 4.3, both I and R are used to update infeasible paths, this can bring back lost information.

$I = \{(A \to B, C \to E), (B \to C, C \to E)\}$
$R = \emptyset$



$I = \{(B \to C, C \to E)\}$
$R = \{(A \to B, C \to E)\}$

Fig. 4: Example of an edge disconnection

Figure 4 shows an example of application of this rule. We disconnect the 2 basic blocks A and C. So we can move all rules containing the edge $A \to C$ from the set I into the set R.

This regular remove can formally be expressed :

Rule 1 *Let e be the removed edge:*

$$\forall (e1, e2) \in I, (e == e1 \vee e == e2) \Rightarrow I = I / \{(e1, e2)\} \wedge R = R \cup \{(e1, e2)\}$$

The particular remove This remove rule is a particular case due to our work context which does not contain any back edge[11]. It happen when the removing edge is part of an infeasible path and when the following constraints are respected:

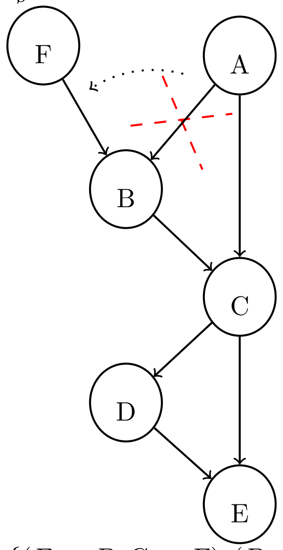
- the In-block of the edge does not have a predecessor
- the Out-block of the edge will have one and only one predecessor after removal
- the Out-block does not contain a ϕ-instruction[12], meaning all predecessors could be executed in parallel as nothing local to the basic block depends on all of them.

If the removed edge respects those constraints, then the removing edge can be replaced by the edge between the Out-block and it's only one remaining predecessor.

[11] A back edge is an edge who go back to previous seen edge (closer to the entry point) instead of going to a new one (closer to the out point)

[12] The ϕ-instruction comes from a special form of the CFG: Single Statement Assignment (SSA). It can be interpreted as : "if the given variable is not null then return it, else return the other given one". More details on SSA and ϕ-instruction are described in [5]

$I = \{(A \to B, C \to E), (B \to C, C \to E)\}$
$R = \emptyset$



$I = \{(F \to B, C \to E), (B \to C, C \to E)\}$
$R = \emptyset$

Fig. 5: Example of an edge disconnection

In figure 5, edge $B \to C$ is removed, and respect all the above constraints. This kind of modification is an update in two phases, first adding edge between $A \to C$, second removing edge $B \to C$. The order of actions is important as if the adding one is done in second time, we will not be able to detect the two phases update action.

To formally express this complex rule 2 we needed to introduce some functions:

- **count** : return the number of elements in a set
- **predecessors** : return the predecessors set of a basic block
- **instructions** : return the instructions set of a basic block
- **is_phinode** : return true if the given instruction is a ϕ-instruction
- **replace** : return the given infeasible path after replacing the edge given in argument 2 by the one given in argument 3

Rule 2 *Let e be the removed edge:*
$\forall (i_{e1}, i_{e2}) \in I \cup R,$

$$\begin{aligned}
&(e == i_{e1} \vee e == i_{e2}) \\
&\wedge count(predecessor(i_{e1})) == 0 \\
&\wedge count(predecessors(i_{e2})) == 1 \\
&\wedge (\forall inst \in instructions(i_{e2}), \neg is\_phinode(inst)) \\
\Rightarrow\ & I = (I \backslash (i_{e1}, i_{e2})) \cup \{replace((i_{e1}, i_{e2}), e_{old}, e_{new})\}
\end{aligned}$$

4.3 Updating In/Out blocks of an edge

As an edge can have only one In-block and one Out-block, the edge update can only refer to the modification of only one of them. Such modification can be seen as a sequence of actions composed of a remove followed by an add.

We skip the simplest case concerning an edge not referred in any infeasible path as we saw in the disconnect section 4.2 there is nothing to do.

Modifying the In block To update infeasible path's rules when updating the In-block, we have to consider two cases: the edge is part of a condition, or not[13]. If not, we can simply replace the old edge by the new one (see rule 2). If yes, the only safe action is to remove infeasible path containing the old edge (see rule 1) as nothing guarantees that the new edge will be part of a conditional jump or if the condition will be the same as in the old block.

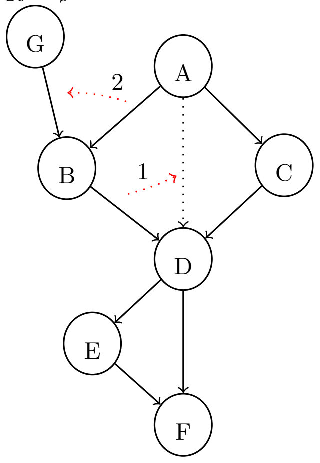
Figure 6 shows the 2 possibilities discussed above. Case 1 shows the replacement of edge $B \to D$ by $A \to D$. This edge is not part of a conditional jump, so we can just replace it in the set of infeasible paths. At the opposite, case 2 shows the replacement of the edge $A \to C$ by $E \to C$, but the edge $A \to C$ is part of a conditional jump, so we can only removed infeasible path referring it.

Modifying the Out-block Replacing the Out-block of an edge is easier than for In-block. The old edge must just be replaced by the new one in every infeasible path referring it.

Besides we noted a particular case which is illustrated in figure 7. When replacing the edge $A \to B$ by $A \to D$, we must also replace any infeasible path referring $B \to D$ by $A \to D$. The consequences of such a replacement is simple:

[13] In LLVM implementation, we know an edge is part of a condition when the branch instruction contain more than one operand: constraint operand + two basic blocks operands

$I = \{(A \to B, D \to F), (B \to D, D \to F)\}$
$R = \emptyset$

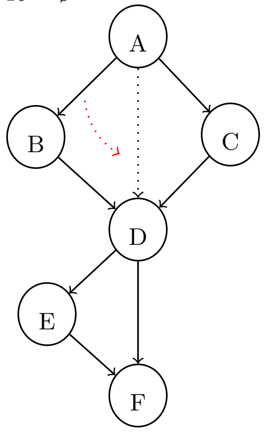


Case 1:
$I = \{(A \to B, D \to F), (A \to D, D \to F)\}$
$R = \emptyset$

Case 2:
$I = \{(B \to D, D \to F)\}$
$R = \{(A \to B, D \to F)\}$

Fig. 6: Two different examples of edge update

$I = \{(A \to B, D \to F), (B \to D, D \to F)\}$
$R = \emptyset$



$I = \{(A \to D, D \to F), (B \to D, D \to F)\}$
$R = \emptyset$

Fig. 7: Example of an edge disconnection

– if $A \to B$ and $B \to D$ was both referred in some infeasible paths, properties will be duplicated, so must not be added.
– if $A \to B$ is not referred in infeasible paths, but $B \to D$ is, then we will correct some pessimistic decisions we made on the $A \to B$ edge. This situation can happen when modifications on the CFG involved adding an edge.

The constraint on this kind of scenario is: The Out block of the new edge is the same block as the only one successor of the Out-block from the old edge.

The formal rule corresponding to this example:

Rule 3 *Let e_{old} be the old edge, e_{new} be the new edge and e_{succ} be the edge from the old Out-block to its unique successor :*

$$\forall (i_{e1}, i_{e2}) \in I \cup R, get\_unique\_successor(Out(e_{old})) == Out(e_{new}) \wedge (e_{succ} == i_{e1} \vee e_{succ} == i_{e2}) \Rightarrow I = (I/(i_{e1}, i_{e2})) \cup \{replace((i_{e1}, i_{e2}), e_{succ}, e_{new})\}$$

4.4 Other IR Manipulations

In the previous sections we worked at the edge level. We present here three more actions from the basic block level, they can help to lose less infeasible paths and so gain in precision. Unfortunately, we did not experienced then when compiling the C-code generated from our Lustre example 3a. The purpose of this subsection is to give the idea of the possible benefit when using those actions.

Those three actions from the basic block level can be seen as *meta-actions* from the edge level. *Meta-actions* are a group of actions which when applied at once on the infeasible paths set could improve the quality of the resulting set. It is important to note that all those actions are valid for our approach as they are still optimization independent and generally will be coded at the lowest structure level.

Splitting a basic block This action can be done by the following sequence:

– adding a basic block to the CFG
– moving the edge from the initial block to the new one
– moving some instructions from the initial block to the new one
– then adding a jump from the initial block to the new one

As explained in subsection 4.1, the adding result in a loss of informations, which can be retrieved when knowing the next actions.

Merging two basic blocks This action may be done by the following sequence:

– moving every instructions from a node to one of its successor ; respectively to one of this predecessors
– updating every edges going to the successor ; respectively coming from the predecessors

Swapping two Out-blocks in a conditional jump This action is done just by updating the branch statement, and it's easy to feel how the infeasible path will evolve when knowing this kind of transformation.

4.5 Definitely Deleting Infeasible Path

Because of big program with a large set of infeasible paths properties, we must find a way to decrease this number to gain speed during the compilation process. So, we wait for a basic block deletion from the CFG. When a basic block is removed, we also delete every infeasible paths rule we have either in I and R, what gives the rule 4 for the set I and its equivalent when replacing the set I by the set R.

Rule 4 *Let B be the removed basic block:*

$$\forall(i_{e1}, i_{e2}) \in I, In(i_{e1}) == B \vee Out(i_{e1}) == B \vee In(i_{e2}) == B \vee Out(i_{e2}) == B \Rightarrow I = I/(i_{e1}, i_{e2})$$

5 Experiments and Results

The previous section presented a framework to trace CFG's modification. This section exhibits at a glance the implementation of the framework. Its goal is to give the idea of the needed work to transmit annotations between the different steps of the compilation process.

5.1 Implementation Details

This work is part of the W-SEPT ANR project, thus our framework has been implemented into the LLVM[14] compiler suite. It is a three phases compiler developed around a strong Object Oriented model. Figure 8 shows the different steps from the C-source file to the WCET estimation. The central part details the LLVM steps as follows:

– **clang** : the front-end which is in charge of lexing/parsing and building the LLVM's intern IR
– **opt** : the optimizer which walks through the IR to analyze and optimize it
– **llc** : the back-end which is in charge of applying optimization specific to a target platform and pretty printing to the target assembly language

[14] http://www.llvm.org

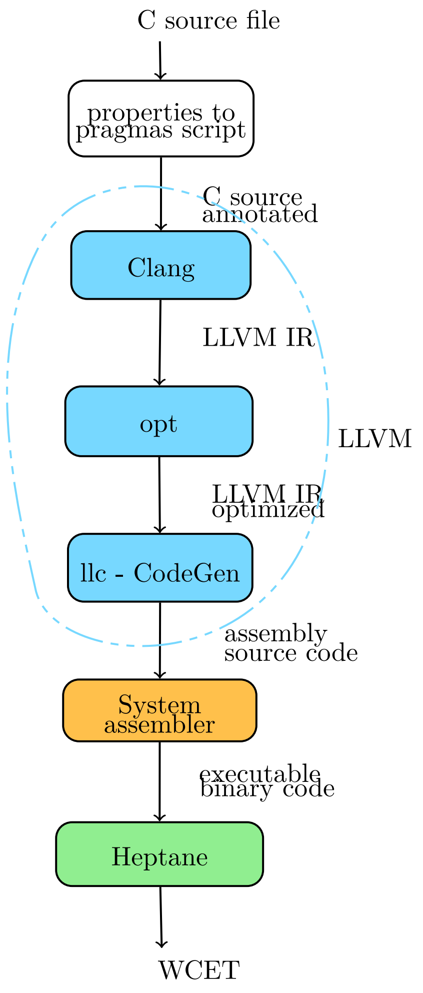


Fig. 8: Chain of traceability

To ensure our approach is safe, the compiler must realize its modifications on the CFG in a thread-safe environment. Meaning every modification on the CFG must be applied sequentially.

5.2 Incorporating properties into code

The first box of the figure 8 is necessary because the infeasible path properties are extracted and stored in a separated file. Then we want to re-introduce it in the C-code.

As Kirner [18] mentioned pragmas are a good candidate to add annotation to modify the behavior of a compiler. Fig 9a shows an example of property extracted from a Lustre program, and 9b the corresponding pragmas.

You must note the initial properties are noted in term of condition variable name[15], but we need the properties to be written in term of basic block[16]. So we modify the C source code by adding labels at the beginning of each block. As label must be unique, it is necessary to duplicate initial properties to refer to the corresponding group of labels.

5.3 Propagating properties into LLVM

The first step of the LLVM compiler is done by Clang which is in charge of parsing and transforming C-code into LLVM IR-code. Once this is done, we no longer deal with C-code, and so no more *pragmas*. Luckily, LLVM propose a mechanism of *metadatas* which can be added to any module (C-file), so we transform our pragmas into *metadatas* to transmit them to the next step.

As described in section 4, we are interested into edges, not basic blocks anymore. So for the next steps of the compiler, *metadatas* will refer to edges as shown in figure 9c.

5.4 From LLVM to binary code

To compute the WCET, we use Heptane which acts at the binary level. So we need to transmit the computed infeasible path written with LLVM IR *metadatas*. There is no way to directly transmit such kind of information into the binary code. So we have to produce a separate file. Because of the link between this study and the W-SEPT project, we will use the FFX format from the OTAWA software[17]. This format is based on XML, and a small example for one infeasible path is shown in figure 9e.

5.5 Getting Events from CFG Transformations

This part is the heart of the framework described in section 4, and will be implemented into the *opt* step of LLVM

To stay as much as possible independent from optimizations and other part of LLVM, we choose to implement the framework with a design pattern derived from the *Observer* from [20]. Basically the observee

[15] In 9a, *L15* and *L5* are both variables into the generated C-code
[16] It is too early to be at an edge-level as the CFG has not been built yet
[17] OTAWA is an open-source tool dedicated to binary code analysis and to WCET computation.

```
#IMPOSSIBLE: [L15 AND L5]
```

(a) Example of Property from automaton in 3a

```
#pragma WCET infeasible_path: [_L15_THEN_7 AND _L5_THEN_2]
#pragma WCET infeasible_path: [_L15_THEN_7 AND _L5_THEN_3]
```

(b) Pragmas corresponding to properties in 9a

```
!wcet.infeasible_path = !{!2}
!2 = metadata !{
metadata !"#~#if.end29~if.then31/if.end29~if.end32",
metadata !"#~#if.then31~_L15_THEN_7/if.end29~if.end32",
metadata !"#~#_L15_THEN_7~if.end32/if.end29~if.end32",
metadata !"#~#if.end29~if.then31/if.end5~if.then7",
metadata !"#~#if.end29~if.then31/if.then7~_L5_THEN_2",
metadata !"#~#if.end29~if.then31/_L5_THEN_2~if.end8",
metadata !"#~#if.then31~_L15_THEN_7/if.end5~if.then7",
metadata !"#~#if.then31~_L15_THEN_7/if.then7~_L5_THEN_2",
metadata !"#~#if.then31~_L15_THEN_7/_L5_THEN_2~if.end8",
metadata !"#~#_L15_THEN_7~if.end32/if.end5~if.then7",
metadata !"#~#_L15_THEN_7~if.end32/if.then7~_L5_THEN_2",
metadata !"#~#_L15_THEN_7~if.end32/_L5_THEN_2~if.end8",
metadata !"#~#if.end29~if.then31/if.end8~if.then10",
metadata !"#~#if.end29~if.then31/if.then10~_L5_THEN_3",
metadata !"#~#if.end29~if.then31/_L5_THEN_3~if.end12",
metadata !"#~#if.then31~_L15_THEN_7/if.end8~if.then10",
metadata !"#~#if.then31~_L15_THEN_7/if.then10~_L5_THEN_3",
metadata !"#~#if.then31~_L15_THEN_7/_L5_THEN_3~if.end12",
metadata !"#~#_L15_THEN_7~if.end32/if.end8~if.then10",
metadata !"#~#_L15_THEN_7~if.end32/if.then10~_L5_THEN_3",
metadata !"#~#_L15_THEN_7~if.end32/_L5_THEN_3~if.end12",
metadata !"#~#if.end5~if.then7/if.end5~if.end8",
metadata !"#~#if.then7~_L5_THEN_2/if.end5~if.end8",
metadata !"#~#_L5_THEN_2~if.end8/if.end5~if.end8",
metadata !"#~#if.end8~if.then10/if.end5~if.end8",
metadata !"#~#if.then10~_L5_THEN_3/if.end5~if.end8",
metadata !"#~#_L5_THEN_3~if.end12/if.end5~if.end8",
metadata !"#~#if.end5~if.then7/if.end8~if.else11",
metadata !"#~#if.end5~if.then7/if.else11~if.end12",
metadata !"#~#if.then7~_L5_THEN_2/if.end8~if.else11",
metadata !"#~#if.then7~_L5_THEN_2/if.else11~if.end12",
metadata !"#~#_L5_THEN_2~if.end8/if.end8~if.else11",
metadata !"#~#_L5_THEN_2~if.end8/if.else11~if.end12",
metadata !"#~#if.end8~if.then10/if.end8~if.else11",
metadata !"#~#if.end8~if.then10/if.else11~if.end12",
metadata !"#~#if.then10~_L5_THEN_3/if.end8~if.else11",
metadata !"#~#if.then10~_L5_THEN_3/if.else11~if.end12",
metadata !"#~#_L5_THEN_3~if.end12/if.end8~if.else11",
metadata !"#~#_L5_THEN_3~if.end12/if.else11~if.end12"}
```

(c) Corresponding infeasible paths to pragmas 9b

```
!wcet.infeasible_path = !{!2}
!2 = metadata !{
metadata !"#~#_L15_THEN_7~if.end32/if.end5~_L5_THEN_2",
metadata !"#~#if.end5~_L5_THEN_2/if.end5~if.end16",
metadata !"#~#_L15_THEN_7~if.end32/_L5_THEN_2~if.end16",
metadata !"#~#_L5_THEN_2~if.end16/if.end5~if.end16"}
```

(d) Resulting *metadatas* from 9c

```
<le>
  <add>
    <count src="if.end5" dst="_L5_THEN_2"/>
    <count src="if.end5" dst="if.end16"/>
  </add>
  <const int="1"/>
</le>
```

(e) Sample XML corresponding a property in 9d

Fig. 9: Transformation chain of infeasible paths properties

fires an event which is caught by a dispatcher that notifies all listeners (listeners must be registered at the beginning of the program). The difficulties were to find where to place event firing, but with the wish to be optimization independent, we placed them into methods from the base element of the CFG (eg: BasicBlock, Instruction, ..).

This implementation has the benefit of being scalable (just add event type, and new event firing spot), reusable (just write a new listener and register it), and non-intrusive (if no listeners, then event will do nothing special).

Optimization level / Infeasible paths properties traceability	O0-level (no optimizations)	O1-level	O2-level	O3-level (every optimizations)
Disabled	2896	1523	1542	1542
Enabled	2014	997	998	998

Table 1: Estimated WCET from the CFG 3a with and without traceability

5.6 Results

The first results are presented by the figure 9d. This is indeed the remaining infeasible paths properties with the optimization level *-O1* of LLVM's *opt*. The figure 3c shows the corresponding optimized CFG.

The estimated WCET presented in the table 1 from the CFG in the figure 3a is computed by Heptane which was configured as follows:

- one 2-way level cache with a latency of 1 and with a size of 512bytes
- a memory with a latency of 15
- an ideal processor with 1 cycle for 1 instruction

The displayed results shows the increase of the WCET's precision when infeasible paths traceability is enable or not. They also show that the compiler has well optimized the code between the two first level of optimizations. But with the two superior levels, the WCET is worst than for O1-level, this comes from certain optimizations which in our context do worst than best.

6 Conclusion

Computing the WCET is an important part of the real-time system development. And it is a complex task especially when using compiler optimizations. But optimizations are important to respect hardware constraints (memory space, processor speed ..). In hard real-time system, the WCET must be safe and tight. So we use a static method to compute a safe WCET, and we do it at the binary level to get a tighter WCET.

One way to improve the tightness of the WCET is to handle infeasible paths properties. Such properties are extracted from a high-level language and must be propagated through compilers to the binary level. We created a framework to trace this property into compiler steps. The framework is based on a system of event catching. Compiler base objects which represent the program (basic blocks, instructions, ...) must send a notification when they are being updated. Those events are then caught by our framework which applied a set of rules corresponding to the event on the infeasible paths properties.

Our framework works at a low level into the compiler architecture, thus we are optimizations independent. This means that if an optimization is added/removed or even modified, our framework will still be working properly.

Now we know it is possible to have safe results just by reacting on the base structure of a program, which is the CFG. But the presented work is much a test of feasibility than a real proof. In order to prove the validity of the exposed rules in section 4, we should implement them into a proved compiler.

The robustness of the presented framework is based on the developer ability to use the good method when modifying the CFG. As we noticed, this is not always true (eg: using add+rem on an edge instead of update). So an important improvement would be to handle a group of events and react to that group. This would help to have better results without loosing the benefit of being optimization independent.

This approach may also be used to propagate other properties than just the infeasible paths. It could easily be possible to react on other events placed on other type of objects such as Functions, Instructions, Variables ... . Also, we worked with infeasible paths properties with only two edges. Being able to handle longer infeasible path will add a lot of precision to the WCET computation. In the whole approach we assumed the test of the conditional branch remain the same, so it would be interesting to handle a modification of the test as some optimizations may use it.

References

1. N. Halbwachs, Synchronous Programming of Reactive Systems. Berlin, Heidelberg: Springer-Verlag, 2010.
2. H. S. Negi, A. Roychoudhury, and T. Mitra, “Simplifying wcet analysis by code transformations,” in Int’l Workshop on WCET Analysis, 2004.
3. Y.-T. Li and S. Malik, “Performance analysis of embedded software using implicit path enumeration,” IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems, vol. 16, no. 12, 1997.
4. E. R. Hanbing Li, Isabelle Puaut, “Traceability of flow information: reconsiling compiler optimizations and wcet estimation,” 2014.
5. A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, Compilers: Principles, Techniques, and Tools (2Nd Edition). Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006.
6. D. F. Bacon, S. L. Graham, and O. J. Sharp, “Compiler transformations for high-performance computing,” ACM Comput. Surv., vol. 26, pp. 345–420, Dec. 1994.
7. R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström, “The worst-case execution-time problem&mdash;overview of methods and survey of tools,” ACM Trans. Embed. Comput. Syst., vol. 7, pp. 36:1–36:53, May 2008.
8. Y.-T. S. Li and S. Malik, “Performance analysis of embedded software using implicit path enumeration,” in Proceedings of the 32Nd Annual ACM/IEEE Design Automation Conference, DAC ’95, (New York, NY, USA), pp. 456–461, ACM, 1995.
9. V. Suhendra, T. Mitra, A. Roychoudhury, and T. Chen, “Efficient detection and exploitation of infeasible paths for software timing analysis,” in Proceedings of the 43rd Annual Design Automation Conference, DAC ’06, (New York, NY, USA), pp. 358–363, ACM, 2006.
10. S. Andalam, P. Roop, and A. Girault, “Pruning Infeasible Paths for Tight WCRT Analysis of Synchronous Programs,” in Intl’ Conf. on Design, Automation and Test in Europe (DATE), 2011.
11. P. Raymond, C. Maiza, C. Parent-Vigouroux, and F. Carrier, “Timing analysis enhancement for synchronous program,” in Proceedings of the 21st International Conference on Real-Time Networks and Systems, RTNS ’13, (New York, NY, USA), pp. 141–150, ACM, 2013.
12. P. Lokuciejewski, D. Cordes, H. Falk, and P. Marwedel, “A fast and precise static loop analysis based on abstract interpretation, program slicing and polytope models,” in Int’l Symp. on Code Generation and Optimization (CGO), 2009.
13. M. de Michiel, A. Bonenfant, H. Cassé, and P. Sainrat, “Static loop bound analysis of C programs based on flow analysis and abstract interpretation,” in IEEE Int’l Conf. on Embedded and Real-Time Computing Systems and Applications (RTCSA), 2008.
14. J. Gustafsson, A. Ermedahl, C. Sandberg, and B. Lisper, “Automatic derivation of loop bounds and infeasible paths for WCET analysis using abstract execution,” in IEEE Real-Time Systems Symposium (RTSS), 2006.
15. C. Healy, M. Sjödin, V. Rustagi, D. Whalley, and R. van Engelen, “Supporting timing analysis by automatic bounding of loops iterations,” Journal on Real-Time Systems, vol. 18, May 2000.
16. J. Engblom and A. Ermedahl, “Modeling complex flows for worst-case execution time analysis,” in IEEE Real-Time Systems Symposium (RTSS), 2000.

17. A. Metzner, “Why model checking can improve wcet analysis,” in Computer Aided Verification (R. Alur and D. Peled, eds.), vol. 3114 of Lecture Notes in Computer Science, pp. 334–347, Springer Berlin Heidelberg, 2004.
18. R. Kirner, P. Puschner, and A. Prantl, “Transforming flow information during code optimization for timing analysis,” Journal on Real-Time Systems, vol. 45, no. 1-2, 2010.
19. J. Engblom, A. Ermedahl, and P. Altenbernd, “Facilitating worst-case execution times analysis for optimized code,” in Euromicro Conf. on Real-Time Systems (ECRTS), 1998.
20. E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Patterns: Elements of Reusable Object-oriented Software. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.