Traceability of Flow Information: Reconciling Compiler Optimizations and WCET Estimation

Hanbing Li
Inria/IRISA
hanbing.li@inria.fr

Isabelle Puaut
University of Rennes 1/IRISA
isabelle.puaut@irisa.fr

Erven Rohou
Inria/IRISA
erven.rohou@inria.fr

ABSTRACT

Real-time systems have become ubiquitous. For this class of systems, correctness implies not only producing the correct result, but also doing so within specified timing constraints. Designers are required to obtain the worst-case execution time (WCET) of their systems to guarantee that all applications meet their time constraints. Many WCET estimation methods have been proposed. They operate through static code analysis, measurements, or a combination of both. Such methods give an upper bound of the time required to execute a given task on a given hardware platform. To be useful, WCET estimates have to be as tight as possible.

Information on possible flows of control (the so-called *flow information*) improves the tightness of WCET estimates. Flow information, should it be produced automatically or be inserted manually from annotations, is typically inserted at source code level. On the other hand, WCET analysis is performed at machine code level. Between these two levels, compiler optimizations may have a dramatic effect on the structure of the code, resulting in a loss of useful information. For this reason, many WCET tools for real-time systems turn off compiler optimizations when computing WCET. In this paper, we propose a framework to trace and maintain flow information from source code to machine code to benefit from optimizations, yet improving the WCET estimates. Our implementation in the LLVM compiler shows that we can improve the WCET of Mälardalen benchmarks by 60 % in average, and up to 86 %. We also provide new insight on the impact of existing optimizations on the WCET.

Keywords

WCET estimation, optimization, compilation, precise and safe, LLVM

1. INTRODUCTION

In real-time systems, knowing the Worst-Case Execution Time (WCET) of pieces of software is required to demonstrate that the system meets its timing constraints, in all situations, including the worst case. WCET calculation methods have to be *safe* and as *tight* as possible. Safety means that the estimate must be higher than or equal to the actual worst-case execution time. Tightness makes the estimate useful: to avoid over-provisioning processor resources, the estimated WCET has to be as close as possible to the actual WCET.

RTNS 2014 , October 8 - 10 2014, Versailles, France
Copyright 2014 ACM 978-1-4503-2727-5/14/10 ...$15.00.
http://dx.doi.org/10.1145/2659787.2659805.

WCET estimation has to be computed at the machine code level, because the timing of processor operations can only be obtained at this level. Moreover, in processors with cache memories, the addresses of memory locations – necessary to analyze the contents of caches – are only known at binary code level.

Information on program control flow is required to calculate tight WCETs. The most basic *flow information* consists in loop bound information (the maximum number of times a loop iterates, regardless of the program input). More elaborate flow information help tighten WCETs, for example by expressing that a given path is infeasible, or that some program points are mutually exclusive during the same run.

Flow information may be obtained by using static analysis techniques or added manually by the application developer through annotations. In both situations, it is convenient to extract or express flow information at the source code level. When using manual annotations, the application developer can focus on the application semantics and behavior, ignoring the compiler and the binary code. When extracted automatically, more flow information can be gathered at source code level than at binary code level because of the higher level of the analyzed language.

Compilers translate high level languages written by programmers into binary code fit for microprocessors. Modern compilers also typically apply hundreds of optimizations to deliver more performance. Some of them are local (i.e. at the granularity of the basic block), they usually do not challenge the consistency of flow information. Other optimizations radically modify the program control flow. As a result, it is usually very difficult to match the structure of the binary code with the original source code, and hence to *port* flow information from high-level to low-level representations. Even when the structure of the binary and source code seem to match, there may be important changes of loop bound information, through optimizations such as loop unrolling or loop re-rolling.

Using optimizing compilers is key to deliver performance. From the point of view of the programmer, compilers are black boxes that take source code as input, and produce binary code. Some compilers can produce dumps of the transformations they applied, but these dumps are very limited.

for(i=0; i<2*n; i++)	for(i=0; i<2*n; i+=2)
// *MAXITER(100)*	// *MAXITER?*
{	{
body(i);	body(i);
	body(i+1);
}	}
(a) Original source code	(b) Optimized (unrolled)

Figure 1: CFG matching and WCET overestimation

Yet, modern compilers apply hundreds of transformations, some very aggressive, that radically modify the structure of loops (consider unrolling, software pipelining, fusion, tiling, polyhedral transformations...) and even functions (inlining, specialization, processing OpenMP directives).[1]

Using the flow information obtained at the source code level, or using best-effort methods for matching source code and binary code may be misleading. In the favorable case, the WCET is "simply" overestimated. Consider the example of Figure 1. The loop on the left has been annotated by the programmer. After optimization, in particular loop unrolling, the code will be similar to the right part of the figure (shown in C language for readability, although it will be expressed in compiler IR, or binary code). Both contain a single loop, and a tool could be tempted to match the CFGs and port the flow information to the binary representation. In this particular case, the result remains safe, but precision is lost since the new loop obviously iterates only 50 times at the maximum, whereas the original loop iterates 100 times. On the other hand, loop rerolling (implemented in some compilers – including LLVM – to reduce code size) results in a increase of the number of loop iterations. Using graph matching would result in underestimated WCETs, that jeopardize the system safety.

In this paper, we propose a framework to systematically transform flow information from source code level down to binary code level. The framework defines a set of formulas to transform flow information for standard compiler optimizations. What is crucial is that transforming the flow information is done *within* the compiler, in parallel with transforming the code (as illustrated in Figure 2). There is no *guessing* what flow information have become, it is transformed along with the code they describe. In case the transformation is too complex to update the information, we always have the option to drop it. The result is then safe, even though it will probably result in a loss of precision. However, we also notify the programmer (or the compiler developer) that this optimization causes problems in a real-time context, making it possible to disable it. Note that only problematic optimizations must be disabled, as opposed to all of them in most current real-time systems.

Our framework is designed to transform flow information as expressed by the most prevalent WCET calculation technique: Implicit Path Enumeration Technique (IPET) [1]. More precisely, flow constraints are expressed as linear relations between execution counts of basic blocks in the program control flow graph. As shown later in the paper, the framework is general enough to cover all linear flow constraints and all typical optimizations implemented in modern compilers. The proposed framework was integrated into the LLVM compiler infrastructure. For the scope of this paper, our experiments in LLVM will concentrate on *loop bounds* as sources of flow information. Experimental results show that LLVM optimizations not only reduce the average-case execution times but also allows to significantly reduce estimated WCETs as well.

[1]Debugging highly optimized programs gives a sense of such surprising transformations.

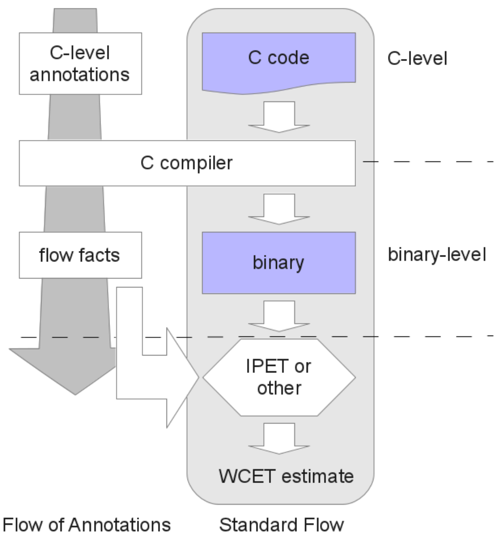


Figure 2: Overall flow

The remainder of the paper is organized as follows. Related work is briefly surveyed in Section 2. Section 3 then details the context in which our work was developed: type of WCET calculation used, types and formats of flow information supported. Section 4 describes how flow information is transformed, independently of the compiler framework. Implementation within the LLVM compiler infrastructure is presented in Section 5. We provide experimental data in Section 6, before concluding with a summary of the paper contributions and plans for future work.

2. RELATED WORK

WCET calculation techniques can be classified into two categories: static and measurement-based methods [2]. Static methods analyze the set of possible execution paths from the program structure. They derive upper bounds of the WCET from the program structure and a model of the hardware architecture. By design, static methods are guaranteed to identify the longest execution path, and therefore, are safe. On the other hand, measurement-based methods use end-to-end measurements on the processor or a cycle-accurate simulator, with different inputs, in search for the input that exercises the longest execution path. This approach may miss the actual worst case. We do not address measurement-based techniques in this paper.

Information on the flow of control of applications improves the tightness of WCET estimates. Beyond loop bounds, which are mandatory for WCET calculation, examples of flow information include infeasible paths, or other properties constraining the relative execution counts of program points. Flow information can be obtained via two basic methods: static analysis or annotations added by the application developer. Among many other examples using static analysis, Healy and Whalley [3] concentrate on the detection and exploitation of branch constraints, and Lokuciejewski et al. [4] extract loop bounds. Tools for WCET estimation also have support for expressing flow information, either using annotations in the source code or through flow expression languages. The former approach is used for example in the Heptane static analysis tool [5], whereas the Otawa static WCET estimation tool[2] defines a language dedicated to the expression of flow information. For the scope of this paper, we assume that flow information is known and the focus is on the traceability of flow information all along the compilation process.

WCC[3] is a WCET-aware compiler that integrates optimizations for WCET minimization. Our work takes a different angle, by addressing general-purpose optimizations and compilers. Experimental results show that most optimizations designed for average-case performance are also beneficial in the worst-case.

Raymond et al. [6] focus on timing analysis enhancement through traceability of flow information for synchronous programs. Full traceability is guaranteed within the Lustre to C compiler, whereas error-prone graph matching is used so far for C to binary compilation. Our work is intended to complement theirs, with the overall objective of having *full* traceability of flow information from very high level languages to binary code.

An early approach was presented by Engblom et al. [7] to derive WCET when code optimizations are applied. According to the authors, there data structures were not powerful enough to support the most complex loop optimizations such as loop unrolling. In contrast, our mechanism can handle most LLVM optimizations, including loop unrolling.

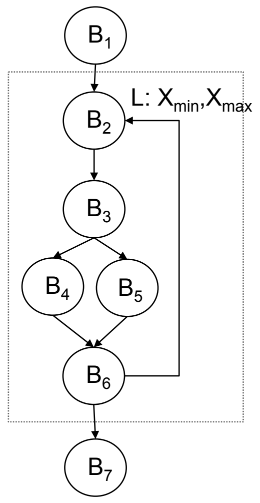
The SATIrE system was introduced [8] as a source-to-source analysis that can map source code annotations to the intermediate representation. Barany et al. [9] use this system to build a WCET analysis tool which combines source-level analysis, optimization and a back-end compiler performing WCET analysis. The connection to several other timing analysis tools is also implemented. Comparing with their source-to-source analysis, our method works on source-to-binary transformation.

Huber et al. propose [10] an approach to relate intermediate code and machine code when generating machine code in compiler back-ends. The approach is based on a novel representation, called *control flow relation graph*, that is constructed from partial mapping provided by the compiler. In contrast to them, we focus in this paper on optimizations performed at the intermediate code level.

The most related work is from Kirner et al. [11, 12], who present a method to maintain correct flow information from source code level to machine code level. It transforms flow information in parallel to the code transformations performed by the compiler. We differ in the following respects. Their first implementation goes back to GCC-2.7.2, a compiler released in 1995, lacking a modern higher-level intermediate representation (GIMPLE was introduced much later), and featuring only "a small number of code transformations that change the control flow of a program significantly". We rely on state-of-the-art technology, and we can handle most optimizations (see Table 1 for details). In a more recent implementation, Kirner et al. rely on source-to-source transformations, while we focus on traceability within the compiler, down to the code generator.

[2]`http://www.otawa.fr`

[3]`http://ls12-www.cs.tu-dortmund.de/daes/en/forschung/wcet-aware-compilation.html`



Objective function

$$\sum_{i \in CFG} f_i \times T_i$$

Structural constraints

$$f_1 = 1$$
$$f_{12} + f_{62} = f_{23} = f_2$$
$$f_{23} = f_{34} + f_{35} = f_3$$
$$f_{34} = f_{46} = f_4$$
$$f_{35} = f_{56} = f_5$$
$$f_{46} + f_{56} = f_{62} + f_{67} = f_6$$

Additional constraints

$$f_6 \leq X_{max}$$
$$f_4 \leq 2 \times f_5$$

Figure 3: CFG and WCET calculation using IPET

3. BACKGROUND: WCET CALCULATION AND EXPRESSION OF FLOW INFORMATION

3.1 WCET calculation using IPET

The static WCET calculation method used in this paper is the most common technique, named IPET for *implicit path enumeration technique* [1]. This method operates on control flow graphs (CFG), extracted from binary code. IPET models the WCET calculation problem as an *Integer Linear Programming (ILP)* formulation.

An example CFG is depicted in the left part of Figure 3. Branch free sequences of code (*basic blocks*) are depicted as circles, whereas arrows represent possible flows between basic blocks. The example program includes one loop, depicted by a rectangular box. Notation X_{min}, X_{max} states that the loop iterates at least X_{min} times, and at most X_{max} times.

The right part of Figure 3 depicts the ILP system used to calculate the WCET. Every basic block i has a worst-case execution time, denoted as T_i, and considered constant in the ILP system. Calculating the WCET is done by maximizing the objective function, in which f_i represents the execution count of basic block i. The control flow is subject to structural flow constraints, that come directly from the structure of the CFG and are generated automatically. From top to bottom, the first one states that the entry point to be analyzed is executed exactly once. The next constraints state that the execution count of a basic block is equal to the sum the execution counts of its incoming edges, as well as outgoing edges, where f_{ij} represents the execution count of the edge from node i to j. Finally, additional constraints specify flow information that cannot be obtained directly from the control flow graph. The first kind of additional information is loop information ($f_6 \leq X_{max}$ in the example). It gives the maximum number of iterations for loops, and is mandatory for WCET estimation. Some other linear constraints such as $f_4 \leq 2 \times f_5$ may also be specified to constrain the relative numbers of executions of basic blocks in the CFG. Additional constraints may be inserted manually by the programmer, through annotations, or be obtained automatically using static analysis methods.

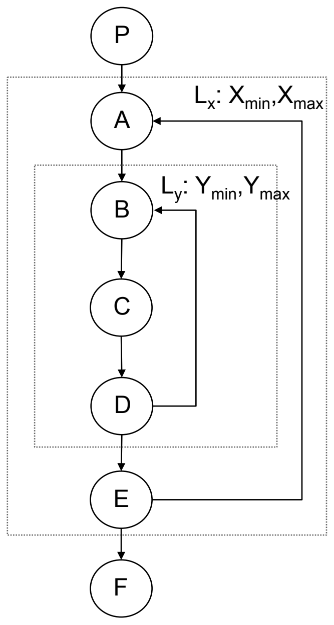


Figure 4: Running example including nested loops

3.2 Notations and assumptions

Transformation of flow information operates on the program control flow graph (CFG)[4]. A CFG is a (possibly cyclic) directed graph made of a set of nodes $\mathcal{N}$ representing basic blocks, and a set of edges $\mathcal{E}$ representing control flow between basic blocks.

In the example program of Figure 3, we have:

$$\begin{aligned}&\text{CFG} = \{\mathcal{N}, \mathcal{E}\}\\ &N = \{B_1, B_2, B_3, B_3, B_4, B_5, B_6, B_7\}\\ &\mathcal{E} = \{B_1 \to B_2, B_2 \to B_3, B_3 \to B_4, B_3 \to B_5,\\ &\quad B_4 \to B_6, B_5 \to B_6, B_6 \to B_2, B_6 \to B_7\}\end{aligned}$$

The proposed framework for traceability of flow information assumes reducible and properly nested loops. Information on loop nesting is captured through the *LoopScope* data structure, made of a set of pairs $\langle L_o, L_i\rangle$ with L_o and L_i as loops. L_i is the inner loop and is completely nested in the outer loop L_o. This data structure is useful because some compiler optimizations involve multiple loops (e.g. loop interchange), their maximum number of iterations have to be modified jointly. The *LoopScope* data structure for our running example depicted in Figure 4 contains:

$$\text{LoopScope} = \{\langle \_, L_x\rangle, \langle L_x, L_y\rangle\}$$

with "_" denoting the absence of enclosing loop for the outermost loop.

Loop bounds are the maximum number of executions of any node in the loop body, regardless of the position of the node(s) testing the loop exit. Without loss of generality, *local* loop bounds are considered, representing the maximum number of iterations of a loop for each entry. Loop bounds are constant and context-independent.

[4]For presentation clarity, we will concentrate in this paper on a single CFG, although the framework supports multiple functions and function calls.

4. A FRAMEWORK FOR THE TRACEABILITY OF FLOW INFORMATION

In this section, we present a transformation framework that conveys flow information from source code level to machine code level. The transformations are expressed in an abstract way, independently of the compiler infrastructure in which they will be integrated.

The transformation framework, for each compiler optimization, defines a set of formulas, that rewrite available flow constraints into new constraints. As mentioned in Section 3, flow information is available in two forms:

- Loop bounds, for every program loop:

$$\textit{Loopbounds} = \{\langle L_x, \langle l_{bound}, u_{bound}\rangle\rangle\}$$

  with L_x the loop identifier and l_{bound} and u_{bound} denoting respectively the minimum and maximum number of iterations for loop L_x, and this for each entry in L_x.

- Additional flow constraints that are linear relations on execution counts of basic blocks (f_i):

$$\textit{Constraints} = \{C_1 + \sum_{i \in CFG} C_2 \times f_i \;\; op \;\; C_3 + \sum_{j \in CFG} C_4 \times f_j\}$$

  with $C_{1/2/3/4}$ non-negative integer constants and *op* an operator in set $\{=, >, \geq, <, \leq\}$. These constraints, to be rewritten jointly with code optimizations, do not include the structural constraints presented in Section 3, but only additional flow information. Indeed, structural constraints can be derived automatically from the CFG at the end of the compilation process.

The transformation framework supports any linear constraint on executions counts of basic blocks. However, in the implementation presented in Section 5, we focus on tracing only the loop bounds.

Note that loop bounds, when finally used to calculate WCET using IPET, will eventually be encoded as linear constraints. However, as illustrated later in the paper, keeping the notion of loops is a richer information, and it integrates more naturally in a compiler.

4.1 Transformation rules

There are three basic rewriting rules for transforming flow information: *change rule*, *removal rule* and *addition rule*.

Change rule

This rule is used to express changes of the execution counts of basic blocks or changes of loop bounds, resulting from compiler optimizations. It is expressed as $\alpha \to \beta$, which means α is substituted by β in the constraints.

In case of a change in the execution count of a basic block, α is f_i, with i one of the basic blocks in the original CFG. β is an expression $\{C + \sum_{j \in newCFG} M \times f_j\}$, with C a constant and M a multiplicative coefficient, that can be either a non-negative integer constant, an interval [a,b] or an interval [a,+∞) in which both a and b are non-negative constants. For example, given the constraint $3f_A \leq 7f_D$ and the rule $f_A \to 4f_B$, we transform the old constraint into the new constraint $3 \times (4 \times f_B) \leq 7 \times f_D$.

In case of a change in loop bounds, α is a loop bound constraint $L_x\,\langle l_{bound}, u_{bound}\rangle$, with $L_x \subset$ original CFG, and

`for (i=0;i<10;i++)`	`for (j=0;j<20;j++)`
`  for (j=0;j<20;j++)`	`  for (i=0;i<10;i++)`
`    a[i][j] = i + j;`	`    a[i][j] = i + j;`
(a) Original source code	(b) Optimized code

Figure 5: Loop interchange optimization

β is $L_x \langle l_{bound'}, u_{bound'} \rangle$. The new loop bounds $l_{bound'}$ and $u_{bound'}$ can be non-negative integer constants or any expression involving only constants (e.g. ceiling or floor) whose result is a non-negative integer.

Removal rule

This rule is used whenever a basic block or a loop is removed from the CFG due to some code optimization. We express it as $\alpha \rightarrow \emptyset$. α can be f_i ($i \in$ original CFG) or $L_x \langle l_{bound}, u_{bound} \rangle$ ($L_x \subset$ original CFG) depending on the object (basic block, loop) that is removed. Through this transformation, α is deleted from the constraints.

For example, with initial constraint $3 \times f_A \leq 7 \times f_D$, the rule $f_A \rightarrow \emptyset$ removes the constraint. With initial constraint $3 \times f_A + 2 \times f_B \leq 7 \times f_D$ and rule $f_A \rightarrow \emptyset$, we get the new constraint $2 \times f_B \leq 7 \times f_D$.

Addition rule

This last rule is meant to be used by optimizations that add new objects (basic block/loop) in the CFG. When a new term is introduced into the CFG, the new constraint is added directly. The constraint should be linear, and should only involve objects (basic blocks, loops) from the new CFG.

For example, if a new constraint $3 \times f_A \leq 7 \times f_D$ appears in the rules set, we just add this constraint into the constraints set.

4.2 Supported compiler optimizations

We defined transformation rules for LLVM optimizations at the `-O1` level. Since the number of optimizations is relatively limited, we also considered other standard compiler optimizations. Table 1 lists the supported optimizations. Note that control flow preserving optimizations do not need specific traceability of flow information, and thus are not listed. Due to space constraints, the full list of rules will be published as a separate report, posted on the project's website[5]. The subsequent paragraphs illustrate our approach through two case studies: interchange, and unrolling.

4.3 Case studies

4.3.1 Loop Interchange

Loop interchange is a standard loop optimization exchanging the order of two iteration variables used by a loop nest. The variable used in the inner loop switches to the outer loop, and vice versa. Loop interchange is typically applied to ensure that elements of a multi-dimensional array are accessed in the order in which they are represented in memory, improving locality of reference and thus performance in architectures with data caches. An example of loop interchange performed at the source code level is given in Figure 5.

The modifications of the CFG due to loop interchange are shown in Figure 6. In the figure, it is assumed that the structure of the CFG (structure of loops, basic blocks) is not altered, although the contents of individual basic blocks due to loop interchange may change. These local changes are denoted by a prime following the basic blocks names.

[5]`http://wsept.inria.fr`

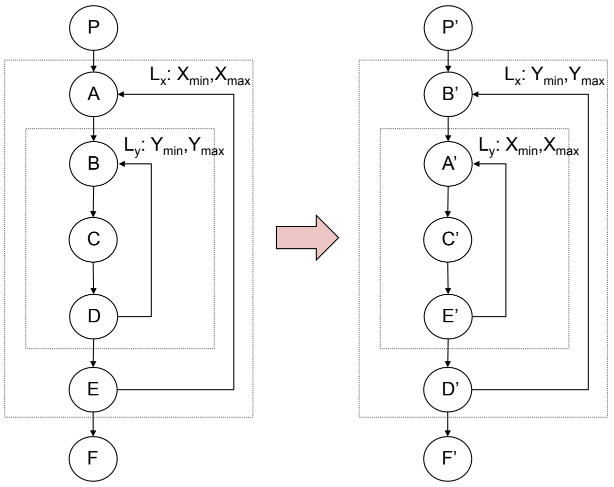


Figure 6: Loop interchange example. The left part of the figure shows the original CFG, whereas the right part shows the optimized one.

The transformation of flow information for the loop interchange optimization only requires the application of the change rule since there is no addition or removal of nodes and basic blocks. The set of rules describing the flow transformation is given below.

$$\begin{aligned}
L_X \langle X_{\text{min}}, X\text{max} \rangle &\rightarrow L_X \langle Y_{\text{min}}, Y\text{max} \rangle \\
L_Y \langle Y_{\text{min}}, Y\text{max} \rangle &\rightarrow L_Y \langle X_{\text{min}}, X\text{max} \rangle \\
f_B &\rightarrow [X_{\text{min}} \cdots X\text{max}] f_{B'} \\
f_A &\rightarrow [\frac{1}{Y\text{max}} \cdots \frac{1}{Y_{\text{min}}}] f_{A'} \\
f_D &\rightarrow [X_{\text{min}} \cdots X\text{max}] f_{D'} \\
f_E &\rightarrow [\frac{1}{Y\text{max}} \cdots \frac{1}{Y_{\text{min}}}] f_{E'}
\end{aligned} \tag{1}$$

The first two lines show that the respective loop bounds of the two loops have been swapped. The following four lines update the constraints of the basic blocks to reflect the alteration of their execution count. For example, the execution count f_B of node B changes from $[X_{\text{min}} \dots X\text{max}] \times [Y_{\text{min}} \dots Y\text{max}]$ to $[Y_{\text{min}} \dots Y\text{max}]$, so the original f_B is replaced by the $[X_{\text{min}} \dots X\text{max}] f_B$. If Y_{min} is 0, we should use $+\infty$ instead of $\frac{1}{Y_{\text{min}}}$.

4.3.2 Loop Unrolling

Another example used to demonstrate constraint transformation is *Loop Unrolling*, illustrated in Figure 1. Loop unrolling replicates the loop body a number of times, called the unrolling factor. Unrolling reduces loop branching overhead and increase instruction level parallelism.

The modifications of the CFG are shown in Figure 7, in the general case where the number of iterations is not known to be a multiple of the unrolling factor. In the figure, the loop body B is replicated k times and the structure of the CFG is changed; a new loop is created to cope with number of iterations not multiple of k. The loop bound of these two

Optimization name	**Description**
Redundancy elimination, control-flow and low-level optimizations of LLVM	
adce	Aggressive dead code elimination
correl.-prop.	Correlated value propagation
deadargelim	Deletes dead arguments from internal functions
dse	Intra basic-block elimination of redundant stores
early-cse	Early common subexpression elimination
functionattrs	Interprocedural deduction of function attributes
globalopt	Transforms simple global variables that never have their address taken
ipsccp	Interprocedural conditional constant propagation
jump-threading	Reduction of the number of branch instructions in case of chained branching.
mem2reg	Promote memory reference to be register references
memcpyopt	Transformations related to eliminating calls to memcpy
prune-eh	Remove unused exception handling info
reassociate	Reassociate expressions to promote better constant propagation
simplifycfg	Dead code elimination and basic block merging
sroa	Scalar replacement of aggregates
tailcallelim	Elimination of tail recursion
Loop Optimizations of LLVM	
loop-simplify	Canonicalize natural loops to make subsequent analyses and transformations simpler and more effective
lcssa	Transform loops in closed SSA form
licm	Loop invariant code motion (move invariant code outside loop body)
loop-unswitch	Transforms loops that contain branches on loop-invariant conditions to have multiple loops
indvars	Canonicalize induction variables: analyzes and transforms the induction variables into simpler forms suitable for subsequent analysis and transformation
loop-idiom	Loop idiom recognizer: transforms simple loops into a non-loop form
loop-deletion	Deletion of loops with non-infinite computable trip counts that have no side effects and do not contribute to the computation of the function's return value
loop rotation	Replacement of a loop with the exit test at the start of a loop with an equivalent one, with the test at the end of the loop.
loop-unroll	Replication of loop body by some unrolling factor to reduce branches and increase instruction-level parallelism
Other supported optimizations not implemented in LLVM	
if simplification	Removal of empty or not taken branches in conditional constructs.
if conversion	Replacement of flow of control by predicated instructions when applicable
loop interchange	Exchange the order of two loops in a perfect loop nest. In general, it switches the outer loop to the inner position and vice versa.
loop fission	Split a loop into multiple loops with the same iteration space as the original one and a subset of the original loop body.
loop fusion	Replace multiple loops with the same loop bound with a single one

Table 1: Supported optimizations. The optimizations included in LLVM appear on top.

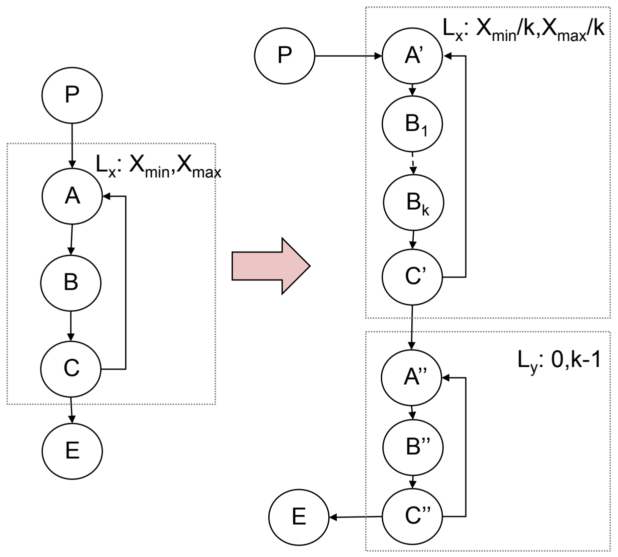


Figure 7: Loop unrolling example. The left part of the figure shows the original CFG, whereas the right part shows the optimized one.

loops are also different from the original one.

The transformation of flow information for loop unrolling requires the application of the change rule and the addition rule because of the addition of the new loop. The set of rules describing the flow transformation is given below.

$$\begin{gathered}
L_X \left\langle X_{\min}, X_{\max} \right\rangle \rightarrow L_X \left\langle \lfloor \frac{X_{\min}}{k} \rfloor, \lfloor \frac{X_{\max}}{k} \rfloor \right\rangle \\
L_Y \left\langle 0, k-1 \right\rangle \\
f_A \rightarrow k \times f_{A'} + f_{A''} \\
f_C \rightarrow k \times f_{C'} + f_{C''} \\
f_B \rightarrow f_{B_1} + \ldots + f_{B_k} + f_{B''}
\end{gathered} \tag{2}$$

The first line (change rule) expresses that the loop bound of the first loop is derived from the loop bound of the original loop by dividing it by the unrolling factor k. The second line (addition rule) expresses the loop bound of the new loop. The following three lines (change rules) update the constraints on the basic blocks to reflect the alteration of their execution count. For example, the execution count f_A of node A is replaced by $k \times f_{A'} + f_{A''}$.

5. IMPLEMENTATION IN THE LLVM COMPILER INFRASTRUCTURE

5.1 The LLVM compiler infrastructure

We integrated the transformation rules described in Section 4 in the LLVM compiler infrastructure [13], version 3.4. LLVM is a collection of modular and reusable compiler and toolchain technologies. As shown in Figure 8, it consists in a three-phase compiler. The first phase is the compiler front-end, named *clang*, which parses, validates and diagnoses errors in the C/C++ code, and then translates the code into the LLVM Intermediate Representation (IR). Then, in a second phase, named *opt* (the LLVM optimizer), a series of analyses and optimizations are performed, with the objective of improving the code quality. Finally, the compiler backend, named *codegen* produces native machine code from IR.

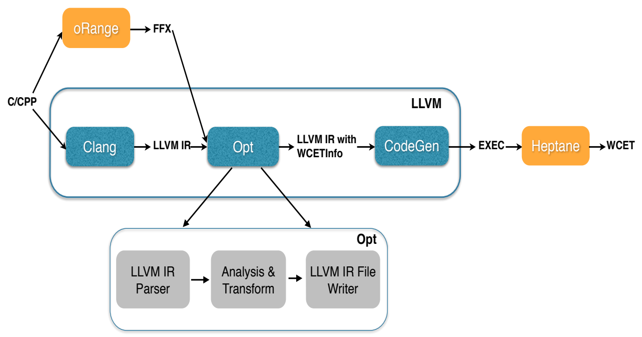


Figure 8: Implementation of traceability in LLVM

LLVM is built around the notion of passes. A pass performs an action on the program. They consist in *Transform* and *Analysis* passes (and a few *Utility* passes). Analysis passes compute various information relevant to subsequent transform passes, such as dominator trees, alias analysis, or loop forests. Each pass can specify its impact on already available information. It may specify that particular information is preserved (allowing further passes to reuse it without recomputing it), while others are invalided, hence destroyed, and must be recomputed.

Yellow boxes in Figure 8 represent external components we added for our WCET flow (oRange and Heptane – described in subsequent sections). Furthermore, we modified *opt* in three places: the parser, to read in annotations from an external file; the individual transformations to convey annotations all the way down; and the IR writer to dump the updated annotations to file.

5.2 Representation and transfer of flow information (WCETInfo)

We added to LLVM a new type of information, named *WCETInfo*, to be attached to the program. Its current purpose is to map loops (Loop objects in LLVM) to the corresponding estimate of loop bounds. Similarly to other information in LLVM, transformation passes may have one of the following behaviors with respect to WCETInfo:

Preserve WCETInfo. This is when the transformation does not modify loops, or when loops are modified, but we know that their bound remain unchanged. Constant propagation is an example of this case.

Update WCETInfo. This happens when loops are modified, but we are able to apply the corresponding transformation to the loop bound information, according to one of the rules of Section 4.

Delete WCETInfo. This occurs when the transformation is unknown, or is known to be too complex to propagate loop bounds correctly. These optimizations are WCET unfriendly, and may render the WCET impossible to compute. Thus, they should be disabled from a compiler targeting real-time systems.

The default for every pass is to delete the WCETInfo, as this is the safe behavior.

The modification of *opt* to read loop bounds reads bounds generated by the oRange static analysis tool [14]. Loop bounds are expressed in the portable flow fact annotation language FFX [15]. Code generation was also modified after all optimization passes to output the final loop bounds in the binary code in a specific section of the binary, for subsequent use in the WCET calculation.

As a side product of using automatically generated loop bounds, we were able to compare the loop bounds generated by oRange with those available inside LLVM using the *scalar evolution* analysis pass. All loop bounds calculated by *scalar evolution* were also computed by oRange and the bounds were identical.

6. EXPERIMENTAL RESULTS

The benefits of compiler optimizations cannot be assessed by enabling them one-at-a-time: optimizations depend on one-another, and many have an impact only when many others have been applied before to prepare the code. In the following, we first examine the impact on WCET of *all* LLVM optimizations of level (`-O1`). Then we evaluate the impact of an optimization by disabling it, and comparing its *negative effect* (we first disabled one optimization out of n, then two out of n).

6.1 Experimental Setup

We demonstrate the impact of our mechanism on program optimization and annotation transformation with the standardized set of WCET benchmarks from Mälardalen University[6].

WCET are estimated using the Heptane timing analysis tool [5], implementing the Implicit Path Enumeration Technique (IPET) for WCET calculation. The ILP solver is CPLEX. For the scope of this paper, to ease the understanding of results, a very simple hardware model is used by Heptane. A 32-MIPS processor is considered, with a 2-level hierarchy of caches and a perfect data cache. The L1 instruction cache is a 2-way 512-byte cache with 8-byte lines, and the L2 cache is a 8-way 16-Kbyte cache with 64-byte lines (the L1 cache size is voluntarily small to match the small size of Mälardalen benchmarks). The cache latency is set to 1 cycle for L1, 10 cycles for L2, and the memory latency to 50 cycles. Both cache levels implement LRU replacement. No instruction-level parallelism (pipeline) is assumed in the architecture.

Code is compiled to assembly using LLVM. The GNU assembler then compiles assembly code to binary that is used to feed Heptane. Optimized codes use the `-O1` option of

[6]`http://www.mrtc.mdh.se/projects/wcet/benchmarks.html`

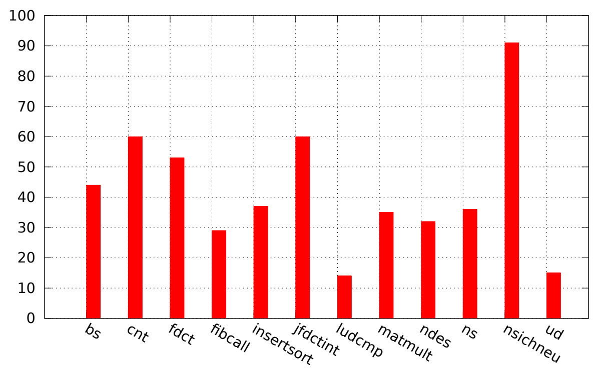


Figure 9: Impact of optimizations (-O1) on WCET. The y-axis represents the WCET with optimizations, normalized with respect to the WCET without optimization (-O0)

LLVM (with the exception of inlining and sparse conditional constant propagation which are not fully supported yet). Note that LLVM has most of its optimizations in `-O1`. Higher levels only add global value numbering, vectorization, global dead code elimination, and constant merging for `-O2`, and argument promotion for `-O3`.

6.2 Impact of optimizations on WCET

Figure 9 shows the impact of compiler optimizations on the WCET computed by Heptane. Results are normalized with respect to the code compiled at `-O0`.

Firstly, we are able to transform all flow information from C code to binary without loss of information. This is shown by the fact that we can compute the WCET of all benchmarks (a single missing loop bound would make the computation impossible).

Secondly, we observe that option `-O1` yields to an important reduction of estimated WCETs: 60 % in average, and up to 86 % (optimized WCET is 14 % of unoptimized) for benchmark *ludcmp*, which contains deeply-nested loops.

6.2.1 *Individual impact of optimizations (1-off)*

Disabling *instcombine* and *loop-rotate* causes problems to the compilation flow of LLVM. Further optimizations crash, and cause an abort of LLVM. Loop rotation transforms top-tested loops into bottom-tested loops. While this transformation has a marginal impact on performance, it is an enabler for others. We hypothesize that further optimizations assume this transformation has been applied, hence crash when we disable it. *Instcombine* also applies some normalization, such as moving constant operand of a binary operator to the right hand side. Again, further optimizations probably assume the IR is in a normal form and fail when it is not the case. Thus, we kept both optimizations enabled.

Table 2 reports our results. For each benchmark (horizontally), we report in the first row the WCET (in cycles) when all optimizations are enabled. Following rows report the estimated WCET when disabling each optimization individually. For example, in the bottom left corner, we see that disabling *simplifycfg* causes an increase of 7 % of WCET of *bs*. In other words, *simplifycfg* improves the estimate. On the contrary, it has an adversary impact on *ud* (last column): the WCET estimate is better by 2 % when *mem2reg* is not run. A dash sign means no change.

General Comments.

Disabling some optimizations has no impact on the WCET. Some simply do not apply to our real-time benchmarks. For example: all loops compute useful values, hence loop deletion has nothing to do; *prune-eh* removes unused exception handlers, which do not exist in C code. Others, such as tail call elimination or memcopy optimization recognize specific patterns that do not occur in our benchmarks. *Globalopt* considers global variables whose addresses are never taken, and optimizes away constant and write-only variables.

Some optimizations, such as *loop-simplify*, do modify the code. It turns out that, in our configuration and for our benchmarks, the WCET remains unaltered.

Register promotion is implemented by *mem2reg*. This is a key optimization that replaces costly memory accesses by much faster register uses. It is a priori surprising that turning it off does not result in major degradation. The reason is that *sroa* achieves the same effect. This is further discussed in Section 6.2.2.

Common subexpression elimination (*early-cse*) and induction variable canonicalization (*indvars*) are basic optimizations of any compiler targeting Average Case Execution Time (ACET). Our results show that they also have a dramatic impact on WCET. These two classic optimizations alone can improve the tightness of WCET by valuable amounts.

Code Layout and I-Cache Effects.

Some transformations result in a minor improvement or degradation (±2 % or so) of the WCET. We suspected this could be a random effect due to a slightly different code layout, resulting in marginally different misses in the cache. To validate our hypothesis, we re-executed the entire experiments, disabling the I-cache (i.e. assuming a perfect cache). As expected we observed that these differences vanish.

Scalar Replacement of Aggregates.

Disabling *sroa* only impacts *ndes*, but the effect is significant: it results in more than 44 % increase in WCET. Visual inspection confirms that this benchmark makes intensive use of small structs (of two and three elements) that can easily be promoted.

As mentioned, *sroa* also captures the register promotion, but this effect is visible when both optimizations are turned off (see Section 6.2.2).

Loop Invariant Code Motion.

Disabling *licm* has a dramatic impact on *ud*, increasing the WCET estimate by 98 %. However, assuming a perfect instruction cache, the impact is reduced to less than 5 %. *Ud* consists in depth-3 loop nests (LU decomposition). The compiler is able to hoist a few instructions outside the loops, explaining the slight WCET improvement.

When the cache is analyzed, and *licm* is not run, many instructions in loops that iterate more than 100 times cannot be proved resident in the cache after the first access. They are thus classified as *always miss*. The effect is exacerbated by the fact that cache lines in our setup are small.

Loop Unrolling.

Loop unrolling produces mixed results: it improves *matmult* (2 %) and *ndes* (12 %), but degrades *ludcmp* (-4 %), *cnt* (-21 %) and *ud* (-7 %). With perfect I-cache, loop unrolling is always worthwhile, improving *cnt, ludcmp, matmult, ndes* and *ud* respectively by 22 %, 5 %, 3 %, 1 %, and 5 %.

	bs	cnt	fdct	fibcall	insertsort	jfdctint	ludcmp	matmult	ndes	ns	nsichneu	ud
WCET (cycles) at -O1	760	7320	18550	396	1426	19239	15201	108614	129345	8882	129687	10795
deadargelim	-	-	-	-	-	-	-	-	-	-	-	-2 %
early-cse	25 %	-	-1 %	-	23 %	-	22 %	-	14 %	0.1 %	-	16 %
indvars	-	15 %	-2 %	54 %	5 %	1.0 %	45 %	8 %	2 %	-	-	49 %
licm	-	-	-	-	-	-	-	-	-	-	-	98 %
loop-unroll	-	-21 %	-	-	-	-	-4 %	2 %	12 %	-	-	-7 %
mem2reg	7 %	-	-	-	-	-	-	-	1 %	-	-	-2 %
reassociate	-	-	-	-	-	1 %	-	-	-	-	-	-
simplifycfg	7 %	-	-	-	-	-	-6 %	-	4 %	-	-	-2 %
sroa	-	-	-	-	-	-	-	-	44 %	-	-	-
adce	-	-	-	-	-	-	-	-	-	-	-	-
correl.-prop.	-	-	-	-	-	-	-	-	-	-	-	-
dse	-	-	-	-	-	-	-	-	-	-	-	-
functionattrs	-	-	-	-	-	-	-	-	-	-	-	-
globalopt	-	-	-	-	-	-	-	-	-	-	-	-
ipsccp	-	-	-	-	-	-	-	-	-	-	-	-
jump-threading	-	-	-	-	-	-	-	-	-	-	-	-
lcssa	-	-	-	-	-	-	-	-	-	-	-	-
loop-deletion	-	-	-	-	-	-	-	-	-	-	-	-
loop-idiom	-	-	-	-	-	-	-	-	-	-	-	-
loop-simplify	-	-	-	-	-	-	-	-	-	-	-	-
memcpyopt	-	-	-	-	-	-	-	-	-	-	-	-
prune-eh	-	-	-	-	-	-	-	-	-	-	-	-
tailcallelim	-	-	-	-	-	-	-	-	-	-	-	-

Table 2: Change in WCET when one optimization is disabled (1-off). Reference is `-O1`. Positive numbers denote a beneficial effect of the optimization (WCET degrades when it is disabled).

Loop unrolling is well known to compiler developers to be a double-edged sword. Average performance improves as long as the working set stays in the cache. The additional misses in the instruction cache cancel the benefits of the optimization. Our results show that the same holds for WCET. In the case of *cnt*, the reason is slightly different. The loop is unrolled ten times (fully unrolled). Its new size is about 400 bytes, which fits in the L1 I-cache. However, due to the structure of the application, there is no reuse of this code. The increased number of cycles comes from additional cold misses. Note, though, that in case of reuse, additional capacity misses are expected, because the unrolled loop size is close to the cache size (512 bytes), and most of its contents is evicted.

6.2.2 Combined impact of optimizations (2-off)

As a final experiment, we disabled pairs of optimizations out of `-O1`. We tried all pairs. Given the amount of data, we only report highlights in Table 3.

The most effective optimizations for our benchmarks are *early-cse*, *indvars*, and *licm*. We often observe additive effects, with up to 2.47× increase in WCET estimate when *licm* and *indvars* are both disabled. While not reported here, this observation is generally true for our set of optimizations and benchmarks. *fdct* without *early-cse* and *indvars* is an exception, resulting in a minor degradation (-1 % vs. -1 % -2 % $\sim$ -3 %). However, with a perfect cache, we observe the opposite behavior, confirming that slightly different code layout is at play.

As for ACET, optimizations are not always additive. This is also true for WCET. It can be illustrated by the pair *simplifycfg + early-cse*. *Bs* shows a sub-additive impact, while on *ndes* the effect is amplified.

As anticipated in the previous section, *sroa* and *mem2reg* have overlapping effects. As mentioned in the LLVM documentation, *sroa* also performs *alloca promotion*, which serves the purpose of SSA formation and results in an effect similar to register promotion. Disabling both optimizations results is a considerable degradation of the WCET estimate.

7. CONCLUSION

Designers of real-time systems are required to compute the WCET of the components of their systems. This is accomplished by combining information provided at high level by programmers (e.g. loop bound information) and generated at low level by compilers. This combination is possible if a mapping is maintained between high- and low-level representations. Optimizing compilers typically break this simple mapping, and developers turn all optimizations off. We propose a framework, built within the LLVM compiler, that traces information through compiler optimizations. We illustrate it on loop bounds, and we show that many optimizations can be turned on. Not only do we not lose any precision, the resulting WCET is much tighter, even in the presence of CFG restructuring transformations such as loop unrolling. We also provide insight about the advantage of running particular optimization of the well accepted Mälardalen benchmarks.

Our ongoing work regarding C to binary traceability consists in extending traceability beyond loop bound information (*e.g.* mutually exclusive branches, which are common in C-code generated by Lustre). Another ongoing work is to introduce contextual information (*e.g.* semantics information that depend on the execution context, such as call context, number of the iteration, etc). Besides, we need to consider global loop bounds because they can result in more precise WCET when triangular loops are analyzed.

Acknowledgments

The research was funded by W-SEPT[5], a collaborative research project focusing on identification and traceability of semantic information for worst-case execution time estimation. W-SEPT is funded by ANR under grant ANR-12-INSE-0001, and supported by the competitiveness clusters

	bs	cnt	fdct	fibcall	insertsort	jfdctint	ludcmp	matmult	ndes	ns	nsichneu	ud
WCET (cycles) at -O1	760	7320	18550	396	1426	19239	15201	108614	129345	8882	129687	10795
mem2reg + sroa	70 %	29 %	71 %	96 %	163 %	58 %	291 %	161 %	181 %	43 %	15 %	227 %
early-cse + indvars	25 %	15 %	-1 %	54 %	24 %	1 %	76 %	8 %	18 %	0.1 %	-	109 %
licm + early-cse	25 %	-	-1 %	-	23 %	-	18 %	-	14 %	0.1 %	-	118 %
licm + indvars	-	15 %	-2 %	54 %	5 %	1 %	45 %	8 %	2 %	-	-	147 %
simp.cfg + early-cse	25 %	-17 %	-1 %	-	23 %	-	22 %	-	30 %	0.1 %	-	22 %

Table 3: Change in WCET when two optimizations are disabled (2-off). Reference is `-O1`. Positive numbers denote a beneficial effect of the optimizations (WCET degrades when they are disabled).

Aerospace Valley and Minalogic. The work was also partially funded by COST Action IC1202: Timing Analysis On Code-Level (TACLe).

We express special thanks to Damien Hardy for his time solving tricky issues when using Heptane and his comments on earlier versions of this paper.

8. REFERENCES

[1] Y-T. S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 16(12):1477–1487, 1997.

[2] 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–overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7(3):36:1–36:53, 2008.

[3] C. A. Healy and D. B. Whalley. Automatic detection and exploitation of branch constraints for timing analysis. *IEEE Trans. on Software Engineering*, 28(8), 2002.

[4] 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.

[5] A. Colin and I. Puaut. Worst case execution time analysis for a processor with branch prediction. *Real-Time Systems*, 18(2-3):249–274, 2000.

[6] P. Raymond, C. Maiza, C. Parent-Vigouroux, and F. Carrier. Timing analysis enhancement for synchronous program. In *Int'l Conference on Real-Time and Network Systems (RTNS)*, pages 141–150, 2013.

[7] J. Engblom, A. Ermedahl, and P. Altenbernd. Facilitating worst-case execution times analysis for optimized code. In *Euromicro Workshop on Real-Time Systems*, pages 146–153, 1998.

[8] M. Schordan. Source-to-source analysis with SATIrE - an example revisited. In *Scalable Program Analysis*, number 08161 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2008. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany.

[9] G Barany and A. Prantl. Source-level support for timing analysis. In *Conference on Leveraging Applications of Formal Methods, Verification, and Validation*, pages 434–448, 2010.

[10] B. Huber, D. Prokesch, and P. Puschner. Combined WCET analysis of bitcode and machine code using control-flow relation graphs. In *Conference on Languages, Compilers and Tools for Embedded Systems (LCTES)*, pages 163–172, 2013.

[11] R. Kirner. *Extending optimising compilation to support worst-case execution time analysis*. PhD thesis, Technische Universität Wien, 2003.

[12] R. Kirner, P. Puschner, and A. Prantl. Transforming flow information during code optimization for timing analysis. *Real-Time Systems*, 45(1):72–105, 2010.

[13] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *Int'l Symp. on Code Generation and Optimization (CGO)*, pages 75–88, San Jose, CA, USA, March 2004.

[14] M. de Michiel, A. Bonenfant, C. Ballabriga, and H. Cassé. Partial Flow Analysis with oRange (short paper). In *Symposium On Leveraging Applications of Formal Methods, Verification and Validation*, number 6416 in LNCS, pages 479–482, 2010.

[15] J. Zwirchmayr, A. Bonenfant, M. de Michiel, H. Cassé, L. Kovacs, and J. Knoop. FFX: A Portable WCET Annotation Language. In *Int'l Conference on Real-Time and Network Systems (RTNS)*, pages 91–100, 2012.