Timing Analysis of Optimised Code *

Raimund Kirner
Institut für Technische Informatik
Technische Universität Wien
Treitlstraße 3/182/1
A-1040 Wien, Austria
raimund@vmars.tuwien.ac.at

Peter Puschner
Institut für Technische Informatik
Technische Universität Wien
Treitlstraße 3/182/1
A-1040 Wien, Austria
peter@vmars.tuwien.ac.at

Abstract

*Timing analysis is a crucial test for dependable hard real-time systems (DHRTS). The calculation of the worst-case execution time (WCET) is mandatory. As modern compilers are capable to produce small and efficient code, software development for DHRTS today is mostly done in high-level languages instead of assembly code. Execution path information available at source code (*flow facts*) therefore have to be transformed correctly in accordance with code optimisations by the compiler to allow safe and precise WCET analysis. In this paper we present a framework based on abstract interpretation to perform this mandatory transformation of flow facts. Conventional WCET analysis approaches use this information to analyse the object code.*

Keywords: Worst-Case Execution Time Analysis, Execution Times, Real-Time Languages, Compiler Optimisations, Code Transformation

1 Introduction

To guarantee timeliness of hard real-time systems the designer needs to pay special attention to the worst-case execution time (WCET). The calculation of tight WCET bounds relies on a precise hardware model and precise knowledge about the possible execution paths. The latter will be called *flow facts*.

As it is not possible to extract all flow facts from the source code due to the *Halting Problem*, manual code annotation is sometimes required. As code development and WCET analysis are usually done at different code representation levels (high-level language coding vs. machine code analysis), it is required to safely transform the flow facts.

Methods that try to transform flow facts simply based on debug information are not able to cooperate with more complex optimisations performed by the compiler. Manual transformations done in [11] would be an error-prone and time-consuming task. A more advanced but still limited approach was proposed in [5] by designing an optimisation description language (ODL). ODL is restricted to simple simple flow facts and in addition is not capable to deal with optimisations like *branch elimination* or *loop unrolling*.

To overcome these limitations, we developed an approach where the transformation of flow facts is integrated into the compiler [10]. In this paper we present the theoretic foundations of this approach. Based on abstract interpretation we obtain the required safe and flexible transformation of those flow facts.

2 Dependable Flow Facts Transformation

To visualise our transformation method we give a short survey on abstract interpretation. Abstract interpretation is a formalised method that supports the systematical construction of a safe and correct interpretation, based on a given concrete interpretation [3]. The basic principle of abstract interpretation is shown in figure 1. The concrete Domain $\langle \mathfrak{D}, \sqsubseteq \rangle$ and the abstract domain $\langle \widetilde{\mathfrak{D}}, \tilde{\sqsubseteq} \rangle$ are here both chain complete partial ordered sets (cpo). The sound mapping between $\mathfrak{D}$ and $\widetilde{\mathfrak{D}}$ is established by a pair of functions $\langle \alpha, \gamma \rangle$. Assuming a Galois connection we get the following properties:

*This work has been supported by the IST research project "High-Confidence Architecture for Distributed Control Applications (NEXT TTA)" under contract IST-2001-32111.

$$\forall d \in D \wedge \forall \widetilde{d} \in \widetilde{D} : \ \alpha(\gamma(\widetilde{d})) \sqsubseteq \widetilde{d} \ \wedge \ d \sqsubseteq \gamma(\alpha(d)) \quad (1)$$

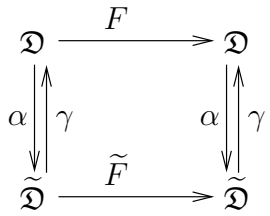

Figure 1. Principle of Abstract Interpretation

2.1 The Correctness of the Transformation

Assuming that the semantic domain $\mathfrak{D}$ itself represents the program to be optimised which is done by the operation F. The correctness is proven by showing observational equivalence [4]: an abstraction function α_o is used to extract the relevant properties for correctness.

An example prepared for our needs is given in figure 2 for the transformation of flow facts in parallel to the code transformation. As already mentioned, the calculation of flow facts cannot be complete. Therefore, certain flow facts ff_a are given manually by the user (denoted by the operation a). Further flow information ff_{impl} is extracted by semantic code analysis denoted by operation F_s. The resulting flow information is denoted $ff = ff_a \cup ff_{impl}$. Finally, the operation $F_t = F_{t1} \times F_{t2}$ represents the code optimisation of the compiler in parallel with the flow facts transformation. The correctness condition shown in figure 2 requires that the observational abstraction α_o has an unchanged semantics for both code annotation and transformation.

$\mathcal{P}$ is the program which has been annotated with flow facts and transformed by the compiler into $\mathcal{P}_t$. $\mathcal{S}[\![\mathcal{P} \times ff]\!]$ denotes the semantics of program $\mathcal{P}$ under consideration of the flow facts ff. Conventional WCET analysis tools require $\mathcal{P}_t$ and ff_t as input.

2.2 Transformation of Flow Facts

Based on the code annotation and transformation shown in figure 2 we perform an abstract interpretation with control-flow path abstraction to correctly transform the flow facts in parallel to the code transformation F_t. The extraction of flow facts ff_{impl} from the source code is not topic of our work. There exists work like [6] tackling this problem.

The concept of our method is shown in figure 3. It is important to note that we use a component-wise combination of the Galois connection $\langle \mathcal{P}, \alpha_s, \gamma_s, \widetilde{\mathcal{P}} \rangle$ and the Galois isomorphism $\langle ff, \alpha_{\equiv}, \gamma_{\equiv}, \widetilde{ff} \rangle$ which therefore again forms a Galois connection $\langle \mathcal{P} \times ff, \alpha_s \times \alpha_{\equiv}, \gamma_s \times \gamma_{\equiv}, \widetilde{\mathcal{P}} \times \widetilde{ff} \rangle$ [12]. The semantic $\mathcal{C}[\![\widetilde{\mathcal{P}} \times \widetilde{ff}]\!]$ of the abstract domain $\widetilde{\mathcal{P}} \times \widetilde{ff}$ describes all possible control flow paths during the execution of $\mathcal{P}$. Our flow fact transformation is done by the operation $\widetilde{F}_t = \widetilde{F}_{t1} \times \widetilde{F}_{t2}$ which is the abstract counterpart to $F_t = F_{t1} \times F_{t2}$. The construction of a sound operation $\widetilde{F}_t$ is done by fulfilling equ. 2.

$$\begin{aligned} \forall \widetilde{p} \in \widetilde{\mathcal{P}} \quad \wedge \quad & \forall \widetilde{f} \in \widetilde{ff} : \\ & F_{t1}(\gamma_s(\widetilde{p})) \sqsubseteq \ \gamma_s(\widetilde{F}_{t1}(\widetilde{p})) \wedge \\ & F_{t2}(\gamma_{\equiv}(\widetilde{f})) \sqsubseteq \ \gamma_{\equiv}(\widetilde{F}_{t2}(\widetilde{f})) \end{aligned} \quad (2)$$

The function $\widetilde{F}_{t1}$ performs the update of the syntactic part $\alpha_s(\mathcal{P})$ of the control flow path description in direct relation to the concrete program transformation F_{t1} (the code optimisation). The operation of the function $\widetilde{F}_{t2}$ is directly obtained from $\widetilde{F}_{t1}$ since $\widetilde{F}_{t2}$ only updates the control-flow dependent part of the flow facts.

$$\begin{aligned} \forall p \in \mathcal{P} \ \wedge \ \forall f \in ff : & \\ & F_{t1}(p) \times F_{t2}(f) \ \sqsubseteq \\ & F_{t1}(p) \times \gamma_{\equiv}(\widetilde{F}_{t2}(\alpha_{\equiv}(f))) \end{aligned} \quad (3)$$

That this approach yields correct flow facts ff_t for the transformed code $\mathcal{P}_t$ is shown in equ. 3 which follows from the fact that the abstraction relation $\alpha_s \times \alpha_{\equiv}, \gamma_s \times \gamma_{\equiv}$ is a Galois connection.

3 Flow Facts for WCET Calculation

Flow facts ff are hints that describe constraints on the possible *control flow paths* (*CFP*). Possible sources for ff are syntax and semantics of the program code or additional annotations (ff_a).

Our WCET calculation is based on integer linear programming (ILP) [14]. This calculation method transforms the structure of a program into an ILP problem and allows to incorporate arbitrary flow facts that describe iteration counts. Other methods like tree-based [2, 13] or path-based [7] WCET calculation are in contrast limited to certain classes of structured flow facts.

The power of the ILP based WCET calculation can be fully exploited by describing ff with *scopes*, *markers*, *restrictions* and *loop bounds* [14, 9]. A sample code that

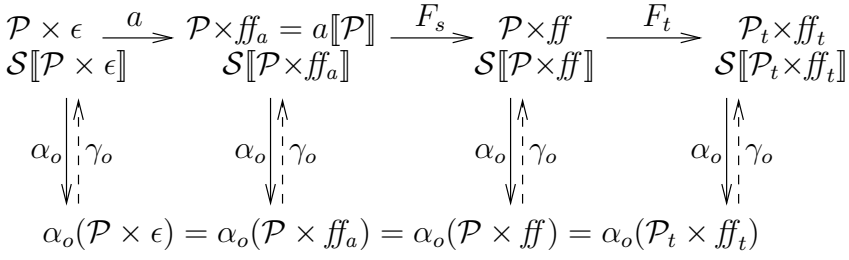

Figure 2. Observational Correctness of Transformation

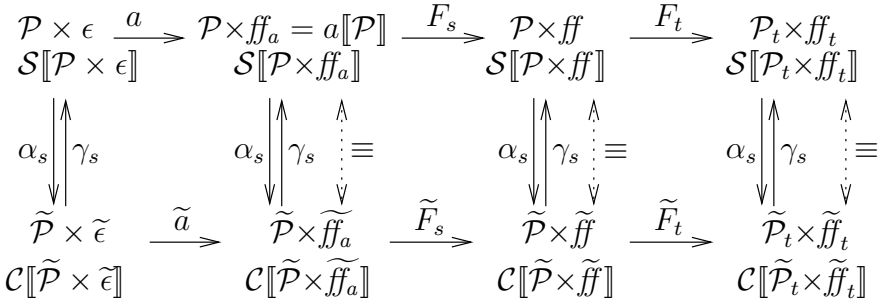

Figure 3. Transformation of Flow Facts

demonstrates the usage of these code annotations is given in figure 4. The code is written in WCETC, a language derived from ANSI C with grammar extensions to express ff_a inside the source code [8]. Each loop is assigned with a lower and upper iteration bound. The safe modelling of certain code transformations requires both the lower and upper iteration bounds. This is true for BCET(best-case execution time) calculation as well as WCET calculation. *Markers* are used to label execution paths of the code. The *restrictions* are used to set the execution counts of several markers in relation to each other. Numeric factors in a restriction without a marker represent execution counts relative to the execution count of the surrounding *scope*. Restrictions have to be valid under all considered possible execution scenarios, e.g. using specific knowledge about the possible input data of a program.

In the sample code of figure 4 we have used the implicit semantic information of the code to specify two restrictions for the lower and upper execution bound of the conditional branch labelled by marker `m2`. For tight WCET results it is required to specify both upper and lower bounds since we do not know which branch will contribute more to the execution time. The analogous argument is valid for the calculation of the BCET.

```
scope
{
  for (i=0,i<=m,i++)
      range 4...10 iterations
  {
      marker m1;
      if (i%2 == 0)
      {
          marker m2;
          arr[i] = d;
      }
      else
          arr[i] = i;
  }

  /* min.  exec.  count of m2 */
  restriction m1 <= 2*m2;

  /* max.  exec.  count of m2 */
  restriction 2*m2 <= m1+1;
}
```

Figure 4. Sample Code, annotated with Flow Facts

CFPS:	$\wp$(StatConn)$\times\wp$(LoopScope)
StatConn:	P$\times$P$\times$FType
FType:	Num $\cup$ {seq, bra}
LoopScope:	Lid$\times$Lid$\times$P$\times$P
P:	*... reference to basic block*
Lid:	*... identifier for loop scope*

Table 1. Data Structure of *CFP* ($\widetilde{\mathcal{P}}$)

FF:	$\wp$(MB)$\times\wp$(Restr)$\times\wp$(FFLF)
MB:	Marker$\times$StatConn
Restr:	$\wp$(Term)$\times$Rel$\times\wp$(Term)
Term:	Num$\times$Marker
Rel:	$\{=,<,\leq\}$
FFLF:	Lid$\times$Bound$\times$LoopMarker
Bound:	Num$\times$Num
LoopMarker:	Marker$\times$Marker
Marker:	*... reference to a marker name*

Table 2. Data Structure of Flow Facts ($\widetilde{ff}$)

3.1 Representation of Flow Facts

The representation of $\widetilde{ff}$ is required to be simple but powerful enough to support the correct $\widetilde{ff}$ update during code optimisation. As described in section 2.2, the construction of the $\widetilde{ff}$ transformation function $\widetilde{F}_{t2}$ is directly induced by the *CFP* update function $\widetilde{F}_{t1}$.

The data structure CFPS, suitable for the modification by function $\widetilde{F}_{t1}$ is described in table 1. CFPS represents the *CFP* (i.e., $\widetilde{\mathcal{P}}$) that can be derived from the syntactic structure of the program $\mathcal{P}$. $\widetilde{\mathcal{P}}$ alone without ff is simply the control flow graph (CFG) of $\mathcal{P}$ extended with loop scope information. The loop scope information is required for correct $\widetilde{ff}$ update. The nodes in the CFG are single basic blocks of $\mathcal{P}$. The edges also encode whether the control flow is just a sequential or branching control flow or simply labelled by a numeric index to support generic CFGs.

It is important to note that $\widetilde{\mathcal{P}}$ does not have to be calculated explicitely since in most compiler architectures it is implicitely represented by $\mathcal{P}$. Only the loop scopes may be an additional data structure that has to be maintained.

The representation of $\widetilde{ff}$ is given in table 2. FF consists of a set of marker bindings for execution edges (MB), a set of restrictions (Restr) and a set of additional loop information (FFLF). The set of restrictions is the same for calculating the WCET and the BCET. Information like the loop bounds given in FFLF could be expressed directly by restrictions but is treated separately for flexibility reasons. When parsing the restrictions in the ff they are inserted into the restriction set.

The structural changes resulting from several code optimisations make it necessary to keep the loop bounds as explicit values. As already mentioned, we maintain an upper and a lower loop bound value. For the final calculation of the WCET only the upper loop bound and for the BCET calculation only the lower loop bound is required. But for the safe ff update in case of certain loop transformations (e.g., loop unrolling) they are required for both calculations.

The above described data structure FF for $\widetilde{ff}$ is flexible enough to safely update $\widetilde{ff}$ during transformations of the program $\mathcal{P}$.

3.2 Required Transformation of Flow Facts

The update of $\widetilde{ff}$ is induced by the *CFP* transformations done by $\widetilde{F}_{t1}$. We use the data symbols $\widetilde{ff} = \langle \Xi, \Gamma \rangle, \ell \rangle \in$ FF where Ξ is the set marker bindings, Γ the set of restrictions, and ℓ is the set of loop frames.

Typical compiler optimisations consist of a program analysis phase and a resulting program transformation phase which can also be performed interleaved. By using the abstracted program transformation function $\widetilde{F}_{t1}$ it becomes obvious that different code optimisations fall into the same class of abstract *CFP* transformations. This fact simplifies the design of a transformation function $\widetilde{F}_{t2}$ that is complete and correct.

Analysing the actions performed by $\widetilde{F}_{t1}$, we can identify the following operations performed at instruction level:

- insert
- move
- copy
- delete
- replace

Changing only the statements within a single basic block does not require to update Ξ, Γ or ℓ. But it becomes more complex when $\widetilde{F}_{t1}$ also includes structural changes of the *CFP*. Facing the operations of $\widetilde{F}_{t1}$ at single instruction level does not allow to induce the required operations to be performed by $\widetilde{F}_{t2}$. Depending on the context of the operation done by $\widetilde{F}_{t1}$ it could be required to

- duplicate involved $b \in \Xi$ and $r \in \Gamma$.
- duplicate involved $b \in \Xi$ and update $r \in \Gamma$ by the sum of original and new markers.
- duplicate involved $b \in \Xi$ and create new restrictions using the old and new markers.
- update the multiplication factor of certain terms in $r \in \Gamma$.

- delete involved $b \in \Xi$ and maybe also $r \in \Gamma$.
- no update of $b \in \Xi$ or $r \in \Gamma$ required.

Without the knowledge of the overall structure update of $\widetilde{\mathcal{P}}$ it is not possible to decide which of the above $\widetilde{ff}$ updates would be required to maintain semantic correctness of the flow facts. As a consequence, we have to group these atomic operations done by $\widetilde{F}_{t1}$ into operations of coarser granularity and use the semantic context of the operations done by $\widetilde{F}_{t1}$ to induce the $\widetilde{ff}$ update.

The challenge for designing the $\widetilde{ff}$ update function is that there exist numerous different code optimisations and even each compiler may handle them slightly different. To overcome this infeasible complexity, we systematically abstract the impact of each code optimisation to the changes of the *CFP*. As a result we get generic *CFP* update patterns for which we can induce the required $\widetilde{ff}$ update:

- SPLIT EXECUTION PATHS: A branch in block b_x in the *CFP* which leads to a block b_y is changed leading to another block b_z. An example for this transformation pattern are *jump optimisations*.

  Transformation: After setting the branch target in b_x to b_z, the execution count of b_y is decreased by the branching count of b_x. All restrictions using markers of the original path of b_y have to be updated with an additional marker that is assigned to the branching edge of b_x.

- DELETE EXECUTION PATHS: One ore more blocks of code are deleted. Typical examples for such a transformation pattern are *dead code, unreachable code or common subexpression elimination*.

  Transformation: For all markers in the blocks to be deleted we have to update in a safe way all restrictions that use them. For the case that we do not know anything about the execution count of the blocks to be deleted, a safe approximation has to be used. If it is for example known at compile time that the code is unreachable (unreachable code can also be produced by prior code optimisations), the transformation is precise by just removing in all restrictions globally the terms using markers that are defined inside the blocks to be deleted.

Possible structural changes on the *CFP* involving iterations are:

- SPLIT ITERATION SPACE OF LOOP: The loop body gets to be executed outside the original loop. Examples for this would be *loop unrolling, loop peeling*.

  Transformation: The original loop and any potentially new created loop have to get an updated loop bound. Additionally, a restriction for limiting the execution count of the original loop and the copies to the original loop bound is emitted.

- CHANGING LOOP SCOPE OF CODE: Blocks are moved from a certain loop scope to another loop scope. Examples for this are *loop unswitching, loop-invariant code motion*.

  Transformation: The restriction multiplication factor for all markers within the blocks have to be updated by the iteration bound of the new loop scope.

- CHANGE OF ITERATION COUNT: The control code of a loop is updated to perform a new number of iterations. Examples for this are *loop interchange, loop coalescing or loop vectorisation*.

  Transformation: The loop bound of the loop has to be updated. If the overall iteration count of the loop body is also changed, then all restrictions using markers from within the loop body have to corrected. The iteration of the body may not be changed if this transformation pattern is performed on nested loops.

Using these abstractions, it becomes possible to define universal $\widetilde{ff}$ update functions. $\widetilde{F}_{t2}$ has to compose simple $\widetilde{ff}$ updates to perform the induced operations. Using these generic pattern also simplifies the correctness proof of the induced function $\widetilde{F}_{t2}$ over the abstract interpretation of the code transformation.

4 Evaluation

To show the importance of supporting compiler optimisations for WCET calculation we have done experiments with a first prototype implementation as shown in table 3. The columns "Calc(x)" and "Meas(x)" show the WCET, obtained by the WCET analysis tool [1] and by measurement respectively, where x denotes the optimisation level used with the compiler(O0 for none and O3 for full optimisation).

The column "*rel*" shows the relation Calc(*O0*)/Calc(*O3*). The potential performance improvement by supporting compiler optimisations is more than a factor of three. This is not a surprising result but confirms that it is quite important to be able to cope with compiler optimisations when doing WCET analysis. Our approach is capable to constructively design a correct transformation of flow facts.

Algorithm	Calc(*O0*)	Meas(*O0*)	Calc(*O3*)	Meas(*O3*)	*rel*
bubble	1 651 040	1 651 040	496 920	495 920	3.32
discrep	1 393 820	1 393 820	439 760	439 760	3.16
mat_mul	5 359 920	5 359 920	1 482 320	1 482 320	3.61

Table 3. WCET Results for Sample Algorithms

5 Summary and Conclusion

In this paper we have presented a novel method to perform precise WCET analysis at object code level using the flow facts obtained at source level. This transformation method is also capable to deal with complex code transformations in a safe manner. Therefore, this method is well-suited for dependable hard real-time systems where both code performance and safety are important. Since this approach is based on the semantics of the performed code transformations it can be systematically integrated into a compiler.

References

[1] P. Atanassov, R. Kirner, and P. Puschner. Using real hardware to create an accurate timing model for execution-time analysis. In *International Workshop on Real-Time Embedded Systems RTES (in conjunction with 22nd IEEE RTSS 2001)*, London, UK, December 2001.

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

[3] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[4] P. Cousot and R. Cousot. Systematic design of program transformation frameworks by abstract interpretation. In *Conference Record of the Twentyninth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 178–190, Portland, Oregon, January 2002. ACM Press, New York.

[5] J. Engblom, A. Ermedahl, and P. Altenbernd. Facilitating Worst-Case Execution Time Analysis for Optimized Code. In *Proceeding of the 10th Euromicro Real-Time Workshop*, Berlin, Germany, June 1998.

[6] J. Gustafsson. *Analysing Execution-Time of Object-Oriented Programs Using Abstract Interpretation.* PhD thesis, Uppsala University, Uppsala, Sweden, May 2000.

[7] C. A. Healy, R. D. Arnold, F. Mueller, D. Whalley, and M. G. Harmon. Bounding Pipeline and Instruction Cache Performance. In *IEEE Transactions on Computers*, number 48 in 1, January 1999.

[8] R. Kirner. The Programming Language WCETC. Research Report 2/2002, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, 2002.

[9] R. Kirner and P. Puschner. Supporting Control-Flow-Dependent Execution Times on WCET Calculation. In *Deutschsprachige WCET-Tagung*, Paderborn, Germany, October 2000. C-Lab.

[10] R. Kirner and P. Puschner. Transformation of Path Information for WCET Analysis during Compilation. In *Proceedings of the 13th Euromicro Conference on Real-Time Systems*, pages 29–36, Delft, The Netherlands, June 2001. Technical University of Delft, IEEE.

[11] S.-S. Lim, J. H. Han, J. Kim, and S. L. Min. A worst case timing analysis technique for multiple-issue machines. In *Proceedings of the 19th Real-Time Systems Symposium (RTSS)*, December 1998.

[12] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis.* Springer, 1999. ISBN: 3-540-65410-0.

[13] P. Puschner and C. Koza. Calculating the Maximum Execution Time of Real-Time Programs. *The Journal of Real-Time Systems*, 1:159–176, 1989.

[14] P. Puschner and A. V. Schedl. Computing Maximum Task Execution Times – A Graph-Based Approach. *The Journal of Real-Time Systems*, 13:67–91, 1997.