IMPROVING THE WCET COMPUTATION TIME BY IPET USING CONTROL FLOW GRAPH PARTITIONING

C. Ballabriga, H. Cassé[1]

Abstract

Implicit Path Enumeration Technique (IPET) is currently largely used to compute Worst Case Execution Time (WCET) by modeling control flow and architecture using integer linear programming (ILP). As precise architecture effects requires a lot of constraints, the super-linear complexity of the ILP solver makes computation times bigger and bigger. In this paper, we propose to split the control flow of the program into smaller parts where a local WCET can be computed faster - as the resulting ILP system is smaller - and to combine these local results to get the overall WCET without loss of precision. The experimentation in our tool OTAWA with lp_solve solver has shown an average computation improvement of 6.5 times.

1. Introduction

Hard real-time systems are composed of tasks which must imperatively finish before their deadlines. To guarantee this termination, scheduling analysis requires the knowledge of each task WCET. To obtain this WCET, three steps are necessary: (1) the task control flow analysis, which determines the possible program paths, (2) the architecture effects analysis, which takes into account the various hardware components (CPU pipeline, instruction cache, etc) to produce timings for program paths, and (3) the final WCET computation.

A widely-used approach for the last step is the Implicit Path Enumeration Technique (IPET) [10]. The different components of the computation (control flow, pipeline execution, etc) are represented as integer linear constraints and the WCET as a linear expression to maximize. The result is then obtained using an Integer Linear Programming (ILP) solver that consumes a lot of time, and is often one of the most time consuming tasks in the WCET computation. The resolution time is usually a function (often non linear) of the number of constraints and variables in the ILP system. For example, the approach for instruction caches of Ferdinand [7] provides good results but the virtual loop unrolling multiplies the number of variables and constraints of a block nested in n level of loops by 2^n.

In this paper, we attempt to reduce the time cost of the WCET computation by splitting the WCET computation into smaller ILP systems and by merging the intermediate results. According to the built constraints, the program is split into small regions where a local safe WCET can be computed. This result is then re-used to compute bigger regions until the whole program is covered. Notice that the first two steps - control flow and architecture effects analyses - are left untouched. This approach has shown time computation improvements in different contexts (taking into account various architecture effects) for the ILP solver lp_solve.

[1]IRIT, Université de Toulouse, CNRS - UPS - INP - UT1 - UTM, 118 rte de Narbonne, 31062 Toulouse cedex 9, email: {ballabri,casse}@irit.fr

The approach is related to [5], where the author partitions the WCET computation into *fact clusters*. The fact clusters are composed of sets of flow facts and the scopes they are applied to (a scope is a structured CFG subgraphs). The clusters bound the range of flow facts in order to get an accurate and feasible WCET computation. Our approach uses a different method to partition the CFG (producing smaller regions) and to take hardware effects into account (especially the pipeline and the instruction cache) without loss of precision.

In the next section, we describe precisely the ILP constraints generated to model the control flow of the task and the architecture effects. Then we describe our approach for splitting the WCET computation and, in particular, we describe the problems caused by hardware modeling constraints for local effects and global effects. The CPU pipeline and the instruction cache are taken as examples. The next section presents the experimentation results of our method and we conclude in the last section.

2. Description of the ILP systems.

As there are a lot of different methods to handle the hardware features in the IPET approach, we present in this section the variant our survey is based on.

2.1. Constraints related to the task control flow

The task control flow analysis as presented in [10] involves representing the analyzed program by a Control Flow Graph (CFG), a directed graph whose nodes are Basic Blocks (BB). A BB is a sequence of instructions with exactly one entry point and one exit point (i.e. no jump in or from the middle). An edge connects two BB if the control can pass directly from the predecessor BB to the successor BB (which can happen if the two BB are in sequence, or in case of a branch, a function call/return, etc).

For each BB and each edge, an ILP variable is created, representing the number of times we execute the BB or take the edge. We name x_i the variable associated with basic block i, and name $e_{i,j}$ the variable associated with edge going from basic block i to j. To represent the structure of the CFG, structural constraints are created for each BB: the sum of execution count for all incoming edges must be equal to the sum of execution count for all outgoing edges, and to the execution count of the basic block (example for BB B_i: $\sum e_{*,B_i} = x_{B_i} = \sum e_{B_i,*}$).

An additional set of constraints is used to represent the loop bounds. A loop can be identified by a loop header L. The edges ingoing L can be divided into *entry edges*, and *back edges*. The entry edges are entering the loop, while the back edges correspond to edge allowing to perform a new iteration. The iteration bound for a loop being N means that the loop is executed at most for N iterations every time it is entered. This is matched by the constraint: $\sum backEdges \leq \sum entryEdges \times (N - 1)$. The WCET is computed by maximizing an objective function that contains a term $t_{B_i} \times x_{B_i}$ for each basic block. Therefore the resulting WCET is $max(\sum t_{B_i} \times x_{B_i})$.

2.2. Instruction Cache modeling constraints

C. Ferdinand et al. presents in [7, 6, 15, 1] a method to handle the instruction cache in the IPET approach. This article proposes a method to predict by abstract interpretation the behavior of set-associative instruction caches with a Least Recently Used (LRU) replacement policy. Thanks to three distinct analyses, the *Must*, *the May*, and the *Persistence* analyses, it categorizes the basic blocks[1] according their behavior in the cache as *Always Hit*, *Always Miss, Persistent,* or *Not Classified. Always Hit* and *Always Miss* means that the access to the cache block results ever, respectively, in a cache hit or in a cache miss. *Persistent* means, intuitively, that the first block access is a miss and the next accesses give hits. *Not Classified* means that we do not know what will happen to the block when it is accessed.

Former approach of C. Ferdinand was only considering May and Must analyses and was using loop unrolling in order to cope with a limited form of the persistence, that is, miss at the first iteration and hit in the following iteration. In [13], and later in [2], an improved approach for dealing with Persistence is proposed. In these papers, it is proposed a new form of Persistent category that takes as parameter the loop that is associated with the Persistent instruction: if the block B is categorized as Persistent at loop L, it means that L is the outer-most loop such that B can not be replaced from the cache within L. That is, L and its inner loops does not wipe out the cache block containing B and the Persistent block cause at most one miss each time the loop L is entered.

To integrate this approach with IPET, the number of hits and miss for cache blocks is represented by the variables x_i^{hit} and x_i^{miss}. In all cases, we assert that $x_i^{hit} + x_i^{miss} = x_i$. The most straightforward constraints are derived from the *Always Hit*, resp. *Always Miss*: $x_i^{miss} = 0$ (resp. $x_i^{hit} = 0$). As the $Persistent$ category means that the block will cause a miss at most each time the related loop L is entered, we can safely generate the constraint $x_i^{miss} <= \sum e_{*,L}$ (where $e_{*,L}$ represents the entry edges of the loop L).

2.3. Constraints for the CPU pipeline

In the original article of Li and Malik [10], the execution time of a basic block is viewed as a constant time produced, for example, by measuring the basic block execution time on a real processor. In [4], Engblom proposes to use simulation to compute the basic block execution times and to refine them on pipelined processor according to the neighbouring basic blocks. Yet, this approach is not safe if the processor exhibits "timing anomalies" or "long time effects".

We have preferred the approach presented in [14] that allows to take into account the basic block context modeled from a set of parameters. The execution time can be expressed as a function of these parameters. This method is based on [9], which expresses the execution pattern of a basic block by an execution graph. This graph conveys the precedence constraints between instructions, and analyses it to derive the basic block execution time considering pessimistic assumptions about the execution context.

If the method is simply used to compute the execution time of a basic block by considering the union of possible contexts, no constraints are generated: the execution time of the basic block is modified according to the pipeline effects, and this impacts only the objective function. However, for more precision, it is possible to consider the basic block predecessors: if a basic block has several predecessors, then due to pipeline effects the basic block execution time is different, depending on the previously executed predecessor. In this case, the differences in execution time are associated to the in-edges by additional terms in the objective function.

[1]For the sake of clarity in the paper, we consider that the basic blocks are split according to cache block boundaries (allowing us to have a single category for each basic block), but our implementation handles regular basic blocks.

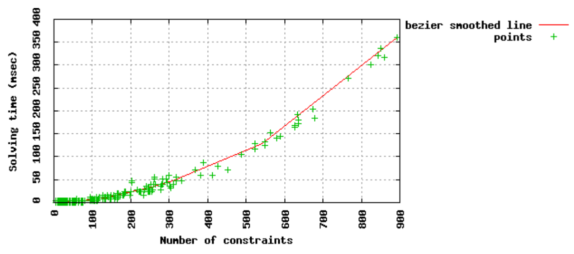


Figure 1. lp_solve solving time

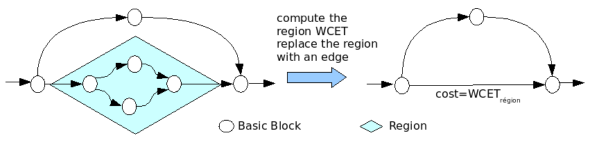


Figure 2. region building

For example, if a basic block B has two predecessors P_1 and P_2, then the terms $diff_{P1,B}. \times e_{P1,B}$ and $diff_{P2,B} \times e_{P2,B}$ are added to the objective function, where $diff_{P1,B}$ and $diff_{P2,B}$ represent the execution time of B when the control flow comes from P_1 and P_2.

2.4. ILP solving performance problems

The last few sections have shown that a lot of variables and constraints are required to compute the WCET. To sum it up, in the control flow constraints, we need a variable for each basic block and for each edge, resulting in two constraints for each basic block. Moreover, we need one more constraint for each loop header. To handle the cache, we need two additional variables and constraints (one for the category and one for $x_i = x_i^{hit} + x_i^{miss}$) for each basic block.

Finally, the use of the contextual execution graph requires only to add terms to the objective function. For a task with n_N basic blocks, n_L loops and n_E edges, usually $n_E > n_N$, we produce an ILP system with $3n_N$ variables and $5n_N + n_L$ constraints). With tasks of several thousands of basic blocks, this makes huge system to resolve and things become worse with the loop unrolling that duplicates constraints for block in nested loops ($\times 2^n$ for n nesting levels).

As the basic approach to resolve ILP systems has a super-linear complexity relatively to the constraint number, the resulting systems are very costly to solve, and takes a large part of the WCET computation time. This is illustrated in Figure 1 that shows the measured execution time for a collection of WCET computation on the lp_solve solver [11]. We have performed our measures on 100 pieces of code coming from parts or whole programs from the Mälardalen benchmark [12]. One may notice that it would be efficient to split the ILP system into subsystems smaller enough (about 100-300 constraints) because the sum of their computation times would be much lesser than the computation of the whole system. Next section introduces a method to approach such a solution.

3. Our approach

The main constraint in splitting the problem into smaller ILP subsystems is to keep the same WCET when the ILP systems are computed separately (our approach does not cause loss of WCET precision). The idea is to isolate sub-systems such such that (1) their local WCET is a constant and (2) the sub-system can be replaced in the global WCET by the product of the local WCET and of the number of times it is executed. Therefore, the subsystem isolation depends on the structure of the system, i.e. on the constraints generated to model the control flow and architecture effects.

3.1. Single-Entry Single-Exit regions

The structural constraints, for each basic blocks, relates the sum of incoming and outgoing edge count to the basic block execution count. We can observe that if a region of the CFG is connected to the outside by one entry edge, and one exit edge, the only constraints connecting region blocks with blocks outside the region are the structural constraints at the entry edge successor (BB_{entry}), and at the exit edge predecessor (BB_{exit}), whose form are $BB_{entry} = entryEdge$ and $BB_{exit} = exitEdge$. Furthermore, since we have a region with a single entry and single exit, all control flow entering by the entry edge leaves by the exit edge such that $BB_{entry} = BB_{exit} = entryEdge = exitEdge = x_{region}$, with x_{region} representing the execution count of the whole region.

Therefore, all variables representing execution count of basic blocks in the region depends ultimately only on x_{region}. Thus we can compute locally the WCET of the region, assuming $x_{region} = 1$ and taking into account only structural constraints and objective function terms corresponding to the region blocks. When the main program is analyzed, we can substitute a single edge for the region, remove from the whole system the constraints of the subsystem, remove from the objective function the terms depending on $x_i \in region$ and add to the objective function $x_{regionEdge} \times WCET_{region}$. This is illustrated in Figure 2.

It is important to keep in mind that it is consistent because the objective function terms that represents the region contribution to the WCET depends only (directly or indirectly) on x_{region}. In other words, it means that the region WCET is always the same, regardless of the execution context. Those single-entry single-exit regions are called *SESE regions* and surveyed in [8].

3.2. Handling CPU pipeline analysis

We consider here the pipeline analysis method already discussed in section 2.3. This method can compute each basic block execution time for the union of all possible contexts. In this case, the execution time of each basic block is independent, and the WCET of the region is the same in every context. Only the objective function needs to be modified: each term $t_i \times x_i$ is adjusted (t_i is set to the execution time of basic block i computed by the pipeline analysis). In this case, no additional problem appears, and the SESE region approach presented in the last section can be used as is.

The CPU pipeline analysis method can also compute the basic block execution depending on the predecessors. In this case, the objective function is enhanced by $diff_{i,B}. \times e_{i,B}$ terms for each predecessor of each basic block B. The unique basic block whose predecessor may be outside the region is the entry basic block (that is, the sink block of the region entry edge). Because the regions are single-entry single-exit, the entry basic block has only one predecessor, and thus, one possible execution time. Therefore, in this case, too, no additional problem appears. To sum it up, the CPU pipeline handling does not cause problems with our approach, provided we limit ourselves to (1) a single time for each basic block or (2) a time dependent on the predecessor only. Experimental results found in [14] shows that WCET tightness is pretty good with a context of only one predecessor.

3.3. Handling cache related constraints

The cache categorization method by C. Ferdinand et al [7, 6, 15, 1] computes categories such as *Always Hit*, *Always Miss*, *Persistent*, and *Not Classified*, and generates ILP constraints based on those categories as described in section 2.2. In order to know if the cache categorization raise any issue

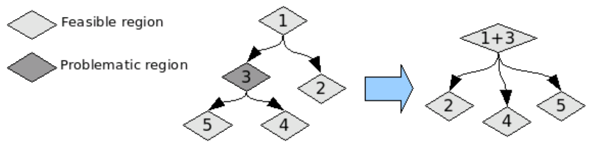


Figure 3. From PST to Feasible Tree

regarding our region partitioning approach, we need to study the ILP constraints generated by each of the possible categories (except for *Not Classified*, which does not generate any constraints).

The *Always Hit* and *Always Miss* categories means that the block causes, respectively, always a hit or a miss. Therefore, the generated constraints, $x_i^{hit} = x_i$ or $x_i^{miss} = x_i$, and the objective function contribution, $x_i^{hit} \times t_i^{hit}$ and $x_i^{miss} \times t_i^{miss}$, does not cause any difficulty because their value is proportional to x_i (either equal, or zero) that, in turn, in derived from x_{region}. So the terms of the objective function to handle the cache may be embedded in $x_{region} \times WCET_{region}$. The loop unrolling approach does not induce any problem as it only duplicates some parts of the code.

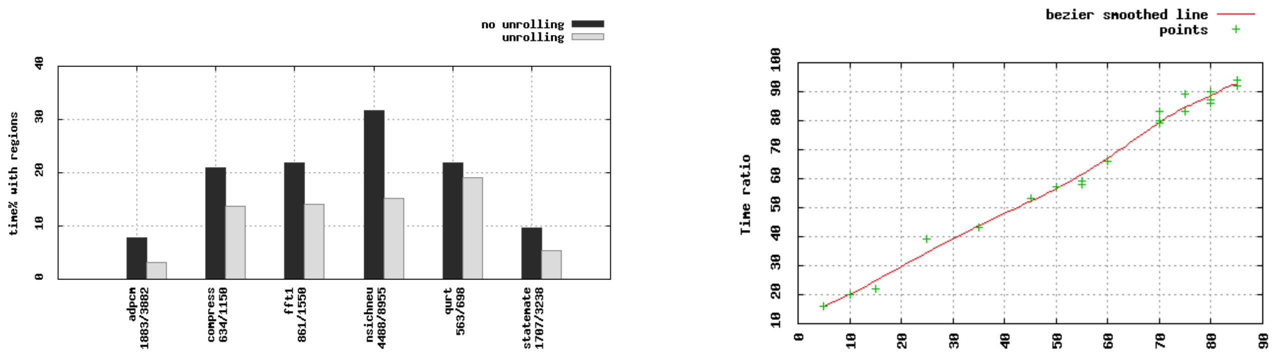
The *Persistent* category have more complex constraints that link together basic blocks count variables from distant CFG parts. The *Persistent* category means that the first execution of the instruction may result in a miss, but all the subsequent executions will result in hits. The corresponding constraint has the form $x_i^{miss} <= \sum e_{*,H_L}$ (H_L is the loop header of L, the loop associated with the *Persistent* block). If the loop header is outside the region, then x_i^{miss} depends on something that can not be derived from the execution count of the region, x_{region}. In other words, for each execution of the region, the hit/miss status of the *Persistent* basic block depends on the presence of the block in the cache at the region entry. Therefore, the region WCET is different depending on the context.

The Persistent category show us that the constraints induce specific limitations to the local computation of region WCET: a region can be computed independently if for any *Persistent* basic block, the associated loop header is also in the region, else the region is infeasible. In a more generic way, the region partition is driven by the nature of the constraints and the objective terms applied to the basic blocks of the region. A region is feasible if (1) the objective function terms tied to the region blocks are proportional to the number of times the region is executed, x_{region}, and (2) any constraint containing variables tied to the region blocks contains only variables tied to the region blocks.

3.4. The Program Structure Tree

We have seen that hardware effects (cache) may disrupt our approach by making some regions infeasible, thus slowing the overall WCET computation. So, to improve the efficiency of our method, we propose to compensate the apparition of non-feasible regions by organizing the integration of local WCET in a hierarchy of computations. This forms a tree called *Program Structure Tree* (PST) [8]. This paper introduces the concept of Single-Entry Single-Exit (SESE) regions, a sub-graph of the CFG with one entry edge and one exit edge. It defines a *canonical* SESE region as a SESE region that does not contain any other SESE region that shares its entry or its exit edges. It is then shown that the canonical SESE regions can be structured in a PST that can be computed in linear time.

The PST is not usable as is, because the hardware effects makes some regions infeasible. The PST building must be modified to remove the infeasible regions, as described in the Figure 3. The PST is visited from the leaves to the root. When an infeasible region is found, it is removed by moving



(a) time gained with lp_solve

(b) sensitivity to random dependencies

Figure 4. Experimental results

its basic blocks and its children regions to its parent region. The result is a tree containing only feasible regions. Finally, the whole WCET is computed by performing a bottom-up visit of the PST, computing each region WCET once all its children are processed and by reporting the children WCET into the parent region system.

4. Evaluation

To validate our approach, we have experimented with it using OTAWA [3], our WCET computation framework. The target architecture features a simple pipelined processor (handled by contextual execution graph) and a 4-way set-associative instruction cache (handled by cache categorization). The measurements have been done on a subset of big-enough Mälardalen benchmarks [12]), with the lp_solve solver [11]. A “big-enough” benchmark means that the generated ILP system provides enough space to extract regions (generating more than 300 constrains).

4.1. Comparison of analysis times

For each selected test of the Mälardalen benchmarks, we have computed the WCET with both the traditional and the region partitioning approaches, and measured the computation time (excluding program path and architecture effects analysis that remains unchanged in our approach). In the case of the region partitioning approach, we have taken into account the time needed to do the partitioning (PST building, identification of infeasible regions, etc).

On average, with our approach the analysis is 6.5 times faster, and results exactly in the same computed WCET. Details can be found on figure 4(a), which gives for each test the percent of the ratio $\frac{time_{new-approach}}{time_{old-approach}}$ (for example, 33% means that we go approximately 3 times faster), with and without loop unrolling.

4.2. Results with random dependencies

To evaluates the limits of our approach, we have also experimented measurement with additional random dependencies. These dependencies generalize the concept of long range dependencies as generated by First Miss blocks in the cache analysis: such a block is linked to the loop header by a constraint. In other words, the dependency from a source basic block BB_1 to a sink basic block BB_2

represents the fact that we need information about BB_1 number of executions to know the execution time of BB_2.

This property could also be applied on any analysis involving categorization, not just cache behavior prediction. As we have no WCET algorithm to produce these dependencies, we have used a random generator. The generated dependencies prevents the feasibility of some regions and stresses our algorithm.

The figure 4(b) shows the impact of these random dependencies on the analysis time. The X axis represents the ratio in percent of the basic blocks affected by a dependency, while the Y axis represents the mean analysis time over all tests for the same Mälardalen benchmarks as in previous section. The results shows that, even if almost every other basic block is affected by a dependency, we still have some time gain.

5. Conclusion

This paper proposes an approach to alleviate the problem of high analysis time for WCET computation with the IPET method by partitioning the program into smaller regions that can be computed independently. We have presented the general principle of our approach (building the PST and visiting it in a bottom-up fashion to compute the WCET of the program) in the trivial case where we do not take into account architecture effects. We have then identified the limitations of our approach in more realistic conditions, including cache behavior prediction, CPU pipeline analysis and even random generation of dependencies between basic blocks. The experimentation conducted in several situations shows a significant time gain.

In future works, we plan to support more architecture effect modeling like the branch prediction unit or the data caches. Although the random dependency generation seems to show that the approach remains efficient, it would be interesting to stress the algorithm with more complex constraints as found in Cache Conflict Graph for example. Another way of experimentation concerns the validation of the method with other industrial or publicly available ILP solvers. Any solver based on the dual simplex algorithm should exhibit some improvements in computation time but additional ILP techniques may offset the time spent in our analysis.

References

[1] M. Alt, C. Ferdinand, F. Martin, and R. Wilhelm. Cache behavior prediction by abstract interpretation. In *SAS '96: Proceedings of the Third International Symposium on Static Analysis*, pages 52–66, London, UK, 1996. Springer-Verlag.

[2] C. Ballabriga and H. Cassé. Improving the First-Miss Computation in Set-Associative Instruction Caches. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2-4 July 2008.

[3] H. Cassé and P. Sainrat. OTAWA, a framework for experimenting WCET computations. In *3rd European Congress on Embedded Real-Time Software*, 25-27 December 2005.

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

[5] Andreas Ermedahl, Friedhelm Stappert, and Jakob Engblom. Clustered worst-case execution-time calculation. *IEEE Transaction on Computers*, 54(9):1104–1122, September 2005.

[6] C. Ferdinand. A fast and efficient cache persistence analysis. *Technical report, Universitat des Saarlandes*, Sept. 1997.

[7] C. Ferdinand, F. Martin, and R. Wilhelm. Applying compiler techniques to cache behavior prediction. In *Proceedings of the ACM SIGPLAN Workshop on Language, Compiler and Tool Support for Real-Time Systems*, pages 37–46, 1997.

[8] R. Johnson, D. Pearson, and K. Pingali. The program structure tree: Computing control regions in linear time. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 171–185, 1994.

[9] X. Li, A. Roychoudhury, and T. Mitra. Modeling out-of-order processors for WCET analysis. *Real-Time Syst.*, 34(3):195–227, 2006.

[10] Y.-T. S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. In *Workshop on Languages, Compilers, and Tools for Real-Time Systems*, pages 88–98, 1995.

[11] lp_solve ILP solver, http://lpsolve.sourceforge.net/.

[12] Mälardalen benchmarks, http://www.mrtc.mdh.se/projects/wcet/benchmarks.html.

[13] F. Mueller. Timing analysis for instruction caches. *Real-Time Systems*, pages 209–239, 2000.

[14] C. Rochange and P. Sainrat. A Context-Parameterized Model for Static Analysis of Execution Times. *Transactions on High-Performance Embedded Architecture and Compilation*, 2(3):109–128, 2007.

[15] H. Theiling, C. Ferdinand, and R. Wilhelm. Fast and precise WCET prediction by separated cache and path analyses. *Real-Time Systems*, 18(2/3):157–179, 2000.