Partial Worst-Case Execution Time Analysis

Rabab Bouziane, Erven Rohou, Abdoulaye Gamatié

Compas'2018 : Parallélisme/ Architecture / Système
ENSEEIHT - Toulouse, France, du 03 au 06 juillet 2018

Partial Worst-Case Execution Time Analysis

Rabab Bouziane[1], Erven Rohou[1], Abdoulaye Gamatié[2]

[1] Univ Rennes, Inria, CNRS, IRISA [2] LIRMM, CNRS, Univ. Montpellier

---

Résumé
Computing the worst-case execution time (WCET) of tasks is important for real-time system design. The industry and research communities have developed a wealth of techniques to compute relevant WCET approximations. Traditionally, WCETs are estimated at the granularity of a function (or task). We propose an approach to estimate partial WCET (pWCET), i.e., the worst-case execution time between two locations in a function, such as basic blocks or instructions. Our technique is derived from the well-known implicit path enumeration technique. It takes into account both the control flow graph and the architecture (pipeline and cache hierarchy). Some useful applications of such pWCETs are motivated in this paper.

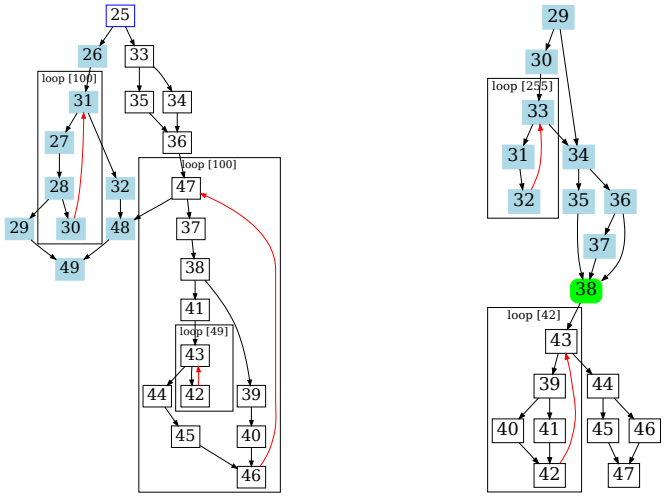
Mots-clés : Worst-case execution time, Program analysis

---

1. Motivation for Partial WCET Estimates and Positioning

Real-time systems are composed of tasks that must deliver their results within a well-defined time-frame. Designers of such systems compute an upper bound of the worst-case execution time of the tasks of a system such that any execution of the task takes less time than the estimate (the bound is said to be *safe*). In addition, to be useful, the bound shall be as close as possible to the actual worst-case (the bound is *tight*).
Traditional WCET estimates are computed at the granularity of a function (or task). This is convenient for both computation and exploitation of the results. Functions are well-defined code fragments with a single entry and few exits, a few parameters, and one or no return value. Compilers and analysis tools have developed extensive theory and a wealth of tools based on basic blocks, control flow graphs, and graph theory to deal with functions.
Yet, designing computing systems requires careful attention to a number of aspects such as performance, energy consumption, or security concerns. Computing partial worst case execution time (pWCET) estimates proves useful in many contexts as discussed below.

Example 1: Performance debugging. Many tools exist to measure the actual (average) execution time of regions of interest in code and identify performance bottlenecks. Pinpointing fragments with high WCET is more difficult because the worst-case may occur in rare circumstances and may not be easy to actually expose, hence usual profiling techniques do not apply. pWCET is useful for developers to identify potential (worst-case) performance bottlenecks in their applications and focus their effort in the relevant code fragments. In multicore systems, performance can be limited by contention on shared resources (caches, bus...), and interactions between tasks must be taken into account when estimating WCET. Without precise knowledge about the occurrence of competing events in the different cores, pessimistic values must be considered. pWCETs are useful to split a task in finer grain fragments, whose executions can be



(a) *expint* benchmark (b) *crc* benchmark

Figure 1: Motivation for pWCET on sample control-flow graphs.

proven to not overlap. This results in fewer contentions and tighter overall WCET. Execution time also matters for security: some software attacks consist in injecting new code in a target application. A protection may consist in computing the WCET of a task and verifying at runtime that the actual time does not exceed the computed value. Any timing anomaly suggests an intrusion. Hence, if we consider the example of Figure 1(a) (taken from the Mälardalen benchmarks [4]) which consists of an unbalanced if-statement: the left branch, starting at block 26, has a pWCET of 78 900 cycles and the right branch, starting at block 33, has 629 524 cycles, then bounding the execution time of the function to the latter value would be overly pessimistic when the left branch executes. pWCET for each branch is much tighter.

Example 2: Energy-efficiency. Energy consumption is another major concern driving the design of a computing system. It led to the development of many techniques such as power gating, DVFS, non-volatile memories... Knowing the WCET of the remainder of a computation makes it possible to apply optimizations. Consider Figure 1(b), and assume that the execution has reached block 38 in advance with respect to its (pessimistic) bound. This slack time combined with the knowledge of the rest of the computation makes it possible to apply energy saving techniques, such as reducing the clock frequency or switching the task to as less power hungry core (as in Arm's big.LITTLE architecture). In the context of Internet of Things (IoT), many systems do not have any exchangeable battery. They harvest energy from physical phenomena (light, vibration...) into a capacitor, and run as long as there is energy left. For these intermittently powered systems, it is crucial to guarantee that the system stops at predefined locations. In other words, when a fragment starts executing, we must guarantee that execution will reach the next checkpoint where the state can be safely stored. Combining pWCET estimates with an energy model of the system is a promising way to ensure that the system makes forward progress and never runs out of power at unwanted locations. Designers also stated incorporating non-volatile memories in their products. On standard STT-RAM, non-volatility refers to a 10-year retention period. However, whenever shorter retention is acceptable, cheaper designs are possible [8]. We envisioned a system with several memory banks designed at various energy/retention points [2]. By computing the WCET between a memory write and its subsequent reads, we can assign writes to the most appropriate bank.

```
int main(void) {
  int i;
  int a=10, b=20, c=0;
  a=6;
  if (a>b) {
    b=a;
    a=20;
    c=10;
    for (i=0; i<c; i++) { ANNOT_MAXITER(10);
      a=i+c;
    }
  } else c=30;
}
```

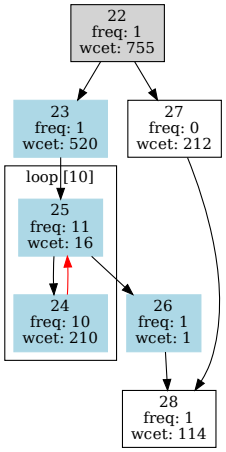


Figure 2: A sample program with its associated Control Flow Graph (CFG). The loop is marked to iterate a maximum of ten times. **ANNOT_MAXITER** is a macro that stores this information in a dedicated ELF section of the binary, to be retrieved by Heptane and attached to the CFG.

Some related work on WCET estimation. Wilhelm et al. [9] presented an overview of methods and existing WCET estimation tools. Two classes of methods are distinguished: static versus measurement-based. Static methods do not rely on real hardware executions. They analyze the code itself, combine the control flow graph with a model of the hardware architecture, and produce an upper bound of this combination. On the other hand, measurement-based methods execute the code on real hardware or a simulator for certain inputs. Then, based on the measured times, the minimal and maximal execution times are derived. Therefore, static methods are safer and guarantee that the execution time will not be higher than the obtained bound.
Recently, Jacobs et al. [6] considered a special case of pWCET. They focus on interference of concurrent tasks sharing a bus, and they compute for how many cycles concurrent cores may be granted access to the resource in any time interval of a given length.

2. Background: the Heptane WCET Estimation Tool

We consider Heptane [5], a static WCET estimation tool, divided in two parts: HeptaneExtract and HeptaneAnalysis. HeptaneExtract generates the control flow graph G from a program written in C language. Then, it identifies the different loops, attaches the loop bounds information provided in the source file and attaches the instruction addresses based on the binary file. Heptane does not include the analysis of maximum number of loops iterations, which are not always statically computable in the general case. Thus, loops must be annotated by the user with their maximum number of iterations (*maxiter*)[1]. Afterwards, HeptaneAnalysis implements IPET (Implicit Path Enumeration Technique) along with cache analysis techniques for several cache architectures. Static WCET estimation methods are divided into two steps: *high level* analysis and *low level* analysis (see Figure 3). The *high level* analysis consists of determining the longest execution path. The *low level* analysis takes into consideration the micro architecture.

[1] External tools such as oRange [3] are able to provide loops upper bounds of C programs in some cases.

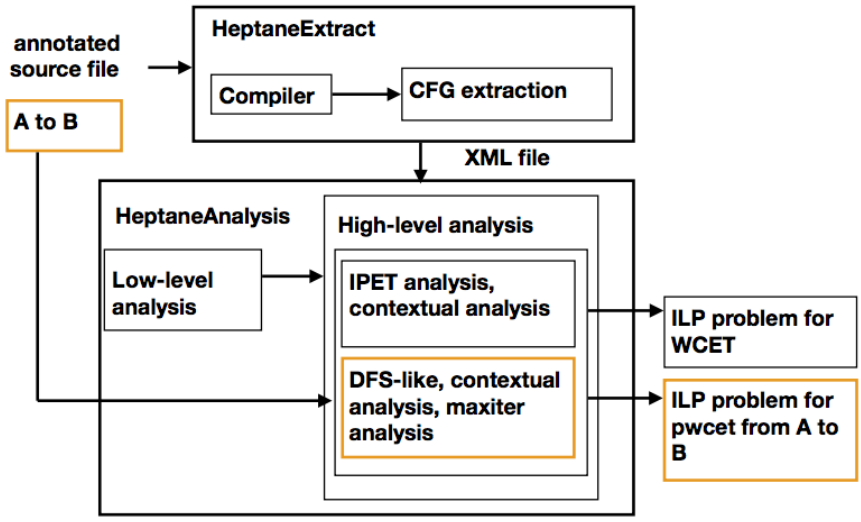


Figure 3: Heptane implementation enhanced with the proposed pWCET analysis (additions shown in yellow).

For the *high level* analysis, Heptane performs an IPET analysis, based on Integer Linear Programming (ILP) formulation of the WCET estimation problem. The program flow is mapped into a set of graph flow constraints. An upper bound of the program's WCET is then obtained by maximizing the following objective function : $max \sum_i n_i \times w_i$ where w_i is the duration of the basic block i (constant in the ILP problem) determined by the low-level analysis, and n_i is the number of times the basic block i is executed (variable in the ILP problem). For the *low level* analysis, Heptane performs data address analysis, a cache analysis and pipeline analysis. Note that Heptane performs context-sensitive analysis, which means that every call path of a function is analyzed separately.

3. Contribution: Computing pWCET Estimates

We first present the general principle of the pWCET Estimation, described through an algorithm. Then we validate it on the Mälardalen Benchmarks [4][2].

3.1. pWCET Estimation Algorithm

A given program is given as a regular executable (both ARM and MIPS instruction sets are supported), in binary format, and an entry point (function `main`). Considering two basic blocks A and B, we are interested in estimating the worst-case execution time from block A to block B. We first compute the WCET estimate for the entire program, which consists in computing for each basic block its WCET estimate, and its worst case execution frequency. Through this step, we obtain the system constraints of the original ILP that we want to use later for pWCET estimations, as well as the contextual analysis. Secondly, we compute the set of blocks and edges that can be traversed in any path from A to B. Consider the example on Figure 2, where we selected block 23 for A and 26 for B. All blocks colored in light blue must be considered, which means that all the blocks in the different paths leading to B must be considered. We achieve

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

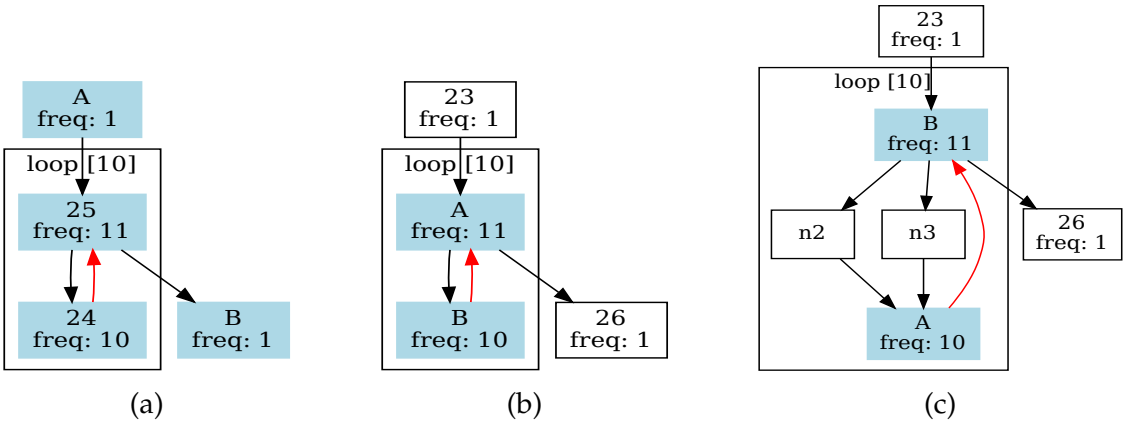


Figure 4: Handling of maxiter

this by a Depth First Traversal -like (DFS-like) walk on the CFG, starting from node A, and we reach B or a block that reaches B. Note that the WCET path from A to B is not necessarily on the overall WCET, i.e, the path from A to B may not be a part of the longest path in the program, thus, the WCET from A to B is not a sub-WCET of the program's WCET.
Then, we compose a new ILP problem for the sub-graph G′ obtained from DFS-like, to compute the WCET from A to B, see Algorithm 1 for the details. Therefore, the objective function will include the nodes in G′ along with the callee nodes (with the callee context) if there is a function call. Moreover, in order to tighten the WCET (make it less pessimistic), we analyze the *maxiter* annotations. The potential for improvement comes from the fact that the annotation applies to the execution of the entire function, while we consider only a subgraph. We take into account the following cases, as illustrated in Figure 4.

1. The backedge is part of the subgraph, the loop may execute its maximum number of iterations on a path from A to B (as in Figure 4(a)). We do not modify the value of *maxiter*.

2. The backedge is not part of subgraph, as in Figure 4(b). B is necessarily reached in the same iteration as A. We set maxiter=0 so that the ILP formulation for the subgraph does not consider pessimistic frequencies due to the loop structure.

3. The backedge is part of subgraph, but it does not contribute to any cycle, as in Figure 4(c). Reaching B requires taking the backedge just once, hence, we set maxiter=1.

So, the new ILP problem has a new system constraints, slightly different from the original one to consider the above cases. Note that the maxiter modification is applied to all backedges that are not part of the subgraph. The whole implementation of the pWCET algorithm has been done inside Heptane, as shown in Figure 3. Given a start node A and an end node B, the high-level analysis of Heptane performs IPET analysis along with the contextual analysis to estimate WCET and then performs the DFS-like search and creates a new ILP problem using the contextual analysis related to the output of the DFS-like and the maxiter analysis.

3.2. Application to a benchmark-suite

We experimented with the Mälardalen Benchmarks, a typical suite for WCET-related experiments. Benchmarks were compiled for ARM using GCC. As customary with real-time systems, no optimization is applied (optimization level `-O0`). The reason for this is the need to

Algorithm 1 pWCET algorithm

procedure DFS-LIKE(A,B)
    DFS(A, B)
    for all nodes N visited in the DFS and not included in the output **do**
        DFS(N, B)
    return L ▷ the list of nodes encountered in possible paths
procedure GENERATION OF THE ILP PROBLEM FROM A TO B
    $max \sum_i n_i \times w_i$ where $i \in L$ ▷ the rest of the basic blocks are excluded by setting their w_i to 0
    for all calls in L **do**
        add to the objective function the callee nodes with their callee context
    Modify w_i based on whether A and/or B are inside a loop or not ▷ Maxiter analysis (see Figure 4)

```
int main(void) {
  int i, j=0;
  for (i=0; i<3; i++) {
    ANNOT_MAXITER(3);
    if (j==1) j=3;
    else j=j+i;
  }
}
```

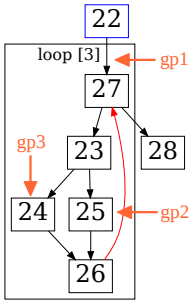
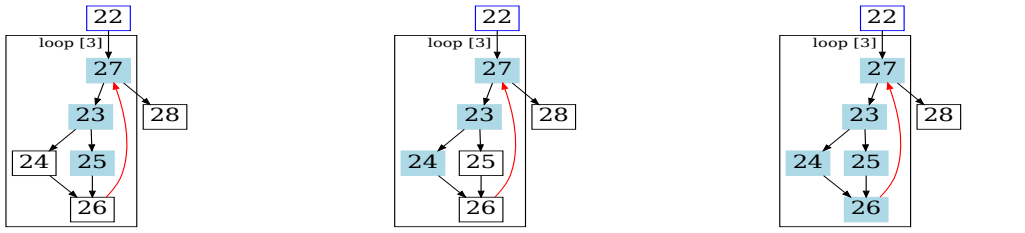


Figure 5: Gain time computation on example from Avila et al. [1].

keep source-level annotations consistent with the binary representation. Compiler optimizations heavily restructure the program representation, to the point that the CFG representation at binary level cannot be matched with the source level, making annotations invalid[3]. The ILP problems are solved by *lp_solve* or *cplex*. For the sake of brevity, we present two Mälardalen benchmarks: *expint* and *crc*, see Figure 1. For *expint*, we observe that the WCET of the subgraph from the block 33 to 49 is much higher that the WCET of the subgraph from the block 26 to 49. For *crc*, we see that the CFG can be divided to two parts: from block 29 to block 38 and from block 38 to block 47. This kind of graphs can actually be illustrated in security where attacks like code injection may occur. Thus, by computing the pWCETs of the two sub-graphs, we can verify at run-time that the real execution time is not higher than the estimated WCET.

We are not aware of any previous work computing estimations of partial WCET, which makes it impossible to compare with prior techniques. To illustrate the potential of our technique, we show that we can encompass previous work, proposed by Avila et al. [1]. The difference between the estimated WCET and the actual execution time is known as *gain time*. Early identification of gain time requires to obtain pWCETs of the code instead of considering the code as a whole. We successfully applied our algorithm on their toy program. In Figure 5, the example is

[3] Li et al. [7] have successfully traced annotations throughout compiler optimizations, but this requires heavy compiler machinery and it is not the focus of this work.



gp1→gp2: pWCET= 348 gp1→gp3: pWCET= 345 gp2↔gp3: pWCET=888

Figure 6: pWCET estimates (cycles) between gain points

taken from [1] where the authors placed three gain points: gp1, gp2 and gp3, in which the time actually consumed by the program is measured. These measurements are used to identify all sources of pessimism of WCET analysis. In Figure 6, different pWCETs are computed to cover possible paths between gain points.

Complexity of pWCET computation. Computation time is hardly noticeable on a modern PC for the Mälardalen benchmarks. For larger problems, solving the ILP problem can require large amounts of time. We add invocations to the solver for each pWCET, but these problems are also much smaller than the overall problem. Since solving ILP problem has exponential complexity, we expect that smaller problems are much faster to solve.

	crc		*expint*	
subgraphs (first node ID → last node ID)	29 → 38	38 → 47	26 → 49	33 → 49
pWCET (cycles)	953 779	34 590	78 900	629 524

Table 1: pWCET Estimation for *crc* and *expint* benchmarks

Table 1 reports several pWCET computed for some nodes within the CFGs of *expint* and *crc*, presented in Figure 1. In the case of *expint*, the function consists in a conditional statement whose branches are heavily unbalanced: the right branch is almost 9× longer than the left one in the worst case. This is mostly due to the presence of a doubly nested loop, and a more complex computation. Recalling the discussion on performance debugging in Section 1 (Example 1), the much tighter bound for the WCET on the left branch makes it possible to make more informed decisions regarding performance analysis or security assessment. Regarding *crc*, the function consists in two computations, scheduled one after the other. Again, we observe a large unbalance between the two subgraphs, the first one requiring 27× more cycles in the worst case, due to a larger tripcount, and the presence of a function call in the loop body. Recalling Example 2 from Section 1, the slack time when reaching block 38 may well be in the order of the remaining of the computation, making it possible to apply aggressive energy saving techniques, or, in the IoT context, skipping a checkpoint.

Note that the pWCET estimates depend on the structure of the CFG and the number and type of instructions inside each block, but also on the capability of Heptane to apply address and cache analysis to bound the number of cache misses.

4. Concluding remarks

In this work, we presented many uses of partial worst case execution time (pWCET) estimates. We showed how to use Heptane, a static WCET estimation tool, to compute pWCETs, based on ILP formulations. Different applications of such an approach are possible as it can be linked to other WCET estimation problems like gain time identification. Hence, pWCET opens many opportunities as discussed in the introductory section of this paper. Future work will consist in leveraging this information in order to deal with the energy efficiency of programs. In particular, we consider exploiting pWCET for reducing energy consumption.

Acknowledgments

We would like to thank Loïc Besnard, Damien Hardy, and Isabelle Puaut for fruitful discussion and feedback on this work. This work has been funded by the ANR Continuum project, under the grant ANR-15-CE25-0007-01.

Bibliographie

1. Avila (M.), Glaizot (M.) et Puaut (I.). – Impact of automatic gain time identification on tree-based static WCET analysis. – In *WCET*, 2013.
2. Bouziane (R.), Rohou (E.) et Gamatié (A.). – How Could Compile-Time Program Analysis help Leveraging Emerging NVM Features? – In *EDIS - First international conference on Embedded & Distributed Systems*, pp. 1–6, Oran, Algeria, décembre 2017.
3. de Michiel (M.), Bonenfant (A.), Ballabriga (C.) et Cassé (H.). – Partial flow analysis with oRange. – In *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*, *LNCS*, number 6416 in LNCS, pp. 479–482, 2010.
4. Gustafsson (J.), Betts (A.), Ermedahl (A.) et Lisper (B.). – The Mälardalen WCET benchmarks: Past, present and future. – In *10th International Workshop on Worst-Case Execution Time Analysis (WCET)*, p. 136, 2010.
5. Hardy (D.), Rouxel (B.) et Puaut (I.). – The Heptane Static Worst-Case Execution Time Estimation Tool. – In *17th International Workshop on Worst-Case Execution Time Analysis (WCET)*, p. 12, Dubrovnik, Croatia, juin 2017.
6. Jacobs (M.), Hahn (S.) et Hack (S.). – WCET analysis for multi-core processors with shared buses and event-driven bus arbitration. – In *Proceedings of the 23rd International Conference on Real Time and Networks Systems*, *RTNS*, RTNS, pp. 193–202. ACM, 2015.
7. Li (H.), Puaut (I.) et Rohou (E.). – Traceability of Flow Information: Reconciling Compiler Optimizations and WCET Estimation. – In *RTNS - 22nd International Conference on Real-Time Networks and Systems*, Versailles, France, octobre 2014.
8. Sun (Z.), Bi (X.), Li (H.), Wong (W.-F.), Ong (Z.-L.), Zhu (X.) et Wu (W.). – Multi retention level STT-RAM cache designs with a dynamic refresh scheme. – In *International Symposium on Microarchitecture (MICRO)*, pp. 329–338. IEEE, 2011.
9. Wilhelm (R.), Engblom (J.), Ermedahl (A.), Holsti (N.), Thesing (S.), Whalley (D.), Bernat (G.), Ferdinand (C.), Heckmann (R.), Mitra (T.), Mueller (F.), Puaut (I.), Puschner (P.), Staschulat (J.) et Stenström (P.). – The worst-case execution-time problem – overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, vol. 7, n3, mai 2008, pp. 36:1–36:53.