Fine-Grain Iterative Compilation for WCET Estimation

Isabelle Puaut, Mickaël Dardaillon, Christoph Cullmann, Gernot Gebhard, Steven Derrien

Fine-Grain Iterative Compilation for WCET Estimation

Isabelle Puaut
Univ Rennes, Inria, CNRS, IRISA
Isabelle.Puaut@irisa.fr

Mickaël Dardaillon
Univ Rennes, Inria, CNRS, IRISA
Mickael.Dardaillon@irisa.fr

Christoph Cullmann
AbsInt Angewandte Informatik GmbH
Science Park 1, D-66123 Saarbrücken, Germany
Cullmann@absint.com

Gernot Gebhard
AbsInt Angewandte Informatik GmbH
Science Park 1, D-66123 Saarbrücken, Germany
Gebhard@absint.com

Steven Derrien
Univ Rennes, Inria, CNRS, IRISA
Steven.Derrien@irisa.fr

Abstract

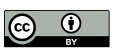
Compiler optimizations, although reducing the execution times of programs, raise issues in static WCET estimation techniques and tools. Flow facts, such as loop bounds, may not be automatically found by static WCET analysis tools after aggressive code optimizations. In this paper, we explore the use of iterative compilation (WCET-directed program optimization to explore the optimization space), with the objective to (i) allow flow facts to be automatically found and (ii) select optimizations that result in the lowest WCET estimates. We also explore to which extent code outlining helps, by allowing the selection of different optimization options for different code snippets of the application.

2012 ACM Subject Classification Computer systems organization → Real-time systems, Computer systems organization → Embedded systems

Keywords and phrases Worst-Case Execution Time Estimation, Compiler optimizations, Iterative Compilation, Flow fact extraction, Outlining

Digital Object Identifier 10.4230/OASIcs.WCET.2018.9

Acknowledgements This work was funded by European Union's Horizon 2020 research and innovation program under grant agreement No 688131, project Argo. The authors would like to warmly thank Benjamin Rouxel, Stefanos Skalistis and Imen Fassi and the anonymous reviewers, for their helpful comments on earlier drafts of this paper.



1 Introduction

Real-time systems play an important role in our daily life. In hard real-time systems, computing correct results is not the only requirement. Results must be also produced within pre-determined timing constraints, typically deadlines. To obtain strong guarantees on the system temporal behavior, designers must compute upper bounds of the Worst-Case Execution Times (WCET) of the tasks composing the system, in order to finally guarantee that they meet their deadlines. Standard static WCET estimation techniques [18] compute such bounds from static analysis of the machine code. Their goal is to obtain a *safe* and *accurate* estimation of a task execution time on a given hardware platform. The *safety* criterion ensures that the WCET holds for any possible execution of the task on the target platform, whereas *accuracy* avoids resource over-provisioning.

WCET analysis is confronted with the challenges of extracting knowledge of the execution flow of an application from its machine code. In particular, loop bounds are mandatory to estimate WCETs. Extraction of flow information can be performed automatically by static WCET analysis tools, or guided by the designer through *flow facts* (loop bounds, unfeasible paths) expressed using source-level annotations.

Compiler optimizations are well known to significantly improve the (average-case) performance of programs, but raise issues regarding WCET estimation. On the one hand, automatic detection of loop bounds may not be feasible anymore because the generated code is more complex and less amenable to static analysis. On the other hand, manual annotations may not be valid anymore after the code optimizations (loops may have been unrolled, re-rolled, split, fused, or may simply have disappeared from the code).

To safely benefit from compiler optimizations, in this paper, we explore the use of *iterative compilation* (exploration of the optimization space) to minimize WCETs instead of average-case in the original use of iterative compilation [6, 2, 4]. More precisely, our contributions are the following:

- We propose and evaluate coarse-grain (application-level) WCET-oriented optimization exploration strategies. Each of the two proposed strategies selects a sequence of optimization passes that (i) allows static WCET analysis tools to automatically detect loop bounds (i.e. disregards optimizations making the WCET estimation fail because it is unable to detect some loop bounds without the use of annotations); (ii) results in the lowest estimated WCET. Two strategies are proposed, the former based on random selection of optimization sequences and the latter using a genetic algorithm.
- We detail and provide preliminary experimental data on a fine-grain (code snippet-level) WCET-oriented optimization exploration strategy, that allows different optimization per code snippet. Interesting code snippets (for the scope of this paper, loops) are outlined, to allow different optimization sequences within a same function. This enables selective application of optimizations: code snippets for which static WCET estimation tools can detect loop bounds with optimizations can be aggressively optimized, whereas the remaining parts can be left un-optimized and later fed with source-level flow fact annotations [13].

Experiments were conducted using the LLVM [12] compilation framework, that allows fine control over optimization passes, and aiT [1], the industry standard for static WCET analysis. The target architecture is the Leon3 core, used to build a predictable multi-core architecture in the framework of the Argo H2020 project[1].

[1] The work presented in this paper is part of ARGO (`http://www.argo-project.eu/`), funded by the European Commission under Horizon 2020 Research and Innovation Action, Grant Agreement Number 688131.

The rest of this paper is organized as follows. We first introduce in Section 2 the background on compilation optimization and static WCET estimation. Section 3 presents the coarse-grain and fine-grain strategies to explore the optimization space in order to both enable automatic flow fact derivation and minimize WCET estimates. Section 4 then describes the experimental setup used to evaluate their quality. Section 5 is devoted to an extensive experimental evaluation of the impact of optimizations on WCET estimates, with and without the proposed techniques. We conclude in Section 6.

2 Background & Related Work

Turning on compiler optimizations impacts static WCET estimation from two respects. First, automatic detection of flow information, in particular loop bounds that are mandatory for WCET estimation, may not be feasible anymore because the transformed code is more complex and less amenable to static analysis. Second, manual source-level annotations may not be valid anymore after the code transformations, in particular the ones deeply modifying loops (loop unrolling, re-rolling, fusion, splitting, among others).

To address these issues, one solution would be to use some feedback provided by the compiler [3] to identify which optimizations were applied, in which ordering and with which parameter, and to transform manually-provided source-level flow information accordingly. However, the level of feedback provided by state-of-the-art compilers is still very limited. Another approach is to instrument the compiler such that it transforms source-level flow information jointly with code transformations, as done by several authors within *gcc* [11], *LLVM* [14, 15] or *WCC* [17]. This however imposes to stick to a given compiler version, or to maintain the flow-fact co-transformation framework along compiler versions. This is the approach followed by the WCC compiler infrastructure [9], containing WCET-oriented optimization and flow fact traceability features. Although a compiler designed specifically for WCET has many benefits, it lacks many optimizations available in standard compilation toolchains such as *gcc/LLVM*.

In this paper, we experiment a completely different approach based on the principles of *iterative compilation*, which is a now mature technology in compilers for optimizing (average-case) performance [6, 2]. The benefits of our approach are twofold. First, we rely on standard industrial strength compiler toolchain to benefit from their large number of available optimizations. Second, we consider the compiler as a black box and adapt the optimization sequences to the code under study to minimize the WCET. Our metrics for evaluating the quality of optimization sequences differ from standard iterative compilation: WCET is optimized instead of average-case performance, and optimizations sequences may be regarded as invalid if static WCET estimation fails at determining loop bounds. One of the hardest challenge in iterative compilation is dealing with the sensitivity of execution performance to input data. Interestingly, this issue does not manifest in our case, because WCET estimation is by definition insensitive to input data, making our approach even more relevant.

3 Proposed WCET-directed Optimization Strategies

Our approaches combine two techniques, with the variation of the optimized code granularity presented in Section 3.1, and the iterative strategies to explore the resulting optimization space in Section 3.2.

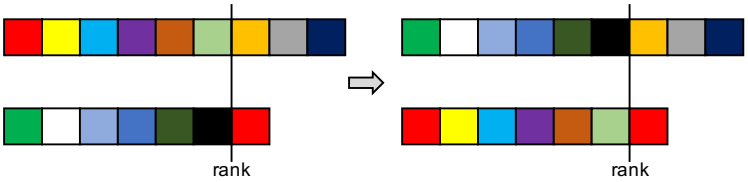


Figure 1 Crossover operation in genetic exploration.

3.1 Optimization Granularity

The intuitive way to optimize an application is to compile all its functions with the same optimization options. We refer to this approach as *coarse-grain optimization* in the rest of the paper.

However, a single problem in a part of the application code, due to the application of an optimization, can forbid the use of that optimization on the whole application. To circumvent this difficulty, we propose to use different optimizations sequences on different parts of the code. In order to isolate a block of code to apply different optimizations we use *outlining* [16]. With this technique, the code snippet to isolate is replaced by a call to a new function that implement the same functionality. All variables used by the code snippet are passed as parameters to the generated function. The naive way to pass arguments is to pass everything by reference, which may significantly increase the average and the worst case execution time. Using liveness [5] properties of the used variables, we can filter which variable needs to be passed by reference or value. In our implementation all arrays are passed by reference; scalars are passed by reference if they are live-out, and by value otherwise; pointers which may be modified are similarly passed by reference.

In this work we systematically outline all outer loops, using to the GeCoS source-to-source code transformation framework [10]. Different optimizations sequences are generated, for the original functions, and also for each new function generated by loop outlining. We refer to this approach as *fine-grain optimization* in the rest of the paper.

3.2 Iterative Optimization Space Exploration Strategies

We designed two strategies to explore the optimization space:

- **Random exploration.** For this strategy, for each experiment the number of optimization passes to be applied is selected randomly. The sequence of passes to be applied is then constructed, each optimization in the sequence being selected randomly, with no attention paid to duplicated optimization passes. This random selection of optimizations is repeated a fixed number of times.
- **Genetic exploration.** A population is set-up, each individual representing a sequence of optimization passes to be applied. Individuals in the population are selected for breeding. At every generation, there is a probability of *mutation* of individuals (here change of one pass in the optimization sequence, selected randomly). Then, the population is doubled in size by randomly selecting N pairs of individuals for breeding. Each pair gives birth to two children by *crossover*. Figure 1 gives an example of *crossover*. A rank in the optimization sequence is selected randomly and the sequences of optimizations are swapped. Similarly to *random exploration*, to keep the implementation simple, no attention is paid to duplicated passes in an optimizations sequence. The optimizations sequences for the initial individuals are selected randomly (using the same techniques

as in the *random exploration* strategy). At each generation, the N best individuals (optimizations sequences for which aiT succeeds in estimating loop bounds, keeping the N lower WCET values) are kept.

4 Experimental Setup

This section presents our experimental setup. We first briefly discuss our choice of input benchmarks. We then describe the compiler and WCET tools used in our experiments, before detailing the parameters of our optimization space search strategies.

4.1 Corpus of Codes

Experiments were conducted on two image processing data benchmarks (Harris and PIPS, see description in Table 1) from the Mälardalen WCET benchmark suite[2] and from the PolyBench/C benchmark suite[3]. We restricted our study to the benchmarks analyzable by aiT with no additional information when compiled without optimization (-O0). This excludes the benchmarks that call library functions (*libmath*, *libc*) that need manual flow annotations. The complete list of benchmarks is given in Table 1 with a small description of each benchmark.

4.2 Compiler and WCET Estimation Tools

Programs are compiled using LLVM [12], version 4.0.0, targeting the Leon3 architecture (Sparc instruction set). Programs are first compiled into LLVM bitcode using the *clang* front-end, before using the *opt* LLVM optimizer to selectively apply optimization passes and then generating a Leon3 executable. *opt* takes as parameters an ordered list of optimization passes. *opt* automatically applies any analysis passes required when turning on a given optimization. The order of application of optimizations is respected unless there are dependencies between passes, in which case *opt* reorders the passes to respect the dependencies. At this stage of our work, we assume the combination of optimization passes to be correct. This will need to be verified for certification concerns, and is considered outside the scope of the paper.

Programs WCETs are estimated using aiT, the industry standard for static WCET analysis, version 17.04, for the Leon3 target [1], configured with no cache. No flow annotations are given to aiT, resulting in situations where the tool is not able to derive them automatically on the optimized code. The virtual unrolling factor of aiT used by its value analysis is set to 2.

Detection of loop bounds in aiT uses an interprocedural data-flow based analysis operating at assembly level. The analysis first searches for loop counters (registers or memory cells with known value when entering the loop). Potential loop counters are further examined by a data-flow analysis to derive *loop invariants* (expressions indicating how the loop counter is modified at each iteration). More details can be found in [7].

4.3 Parameters of Optimization Exploration Strategies

To provide a fair comparison, the same number of optimization sequences were experimented on each benchmark. For *random exploration*, 1000 random optimizations sequences were generated. By default, our *genetic exploration* generated optimizations sequences from 10

[2] `http://www.mrtc.mdh.se/projects/wcet/benchmarks.html`
[3] `http://web.cse.ohio-state.edu/~pouchet.2/software/polybench/`

Table 1 Corpus of programs.

Harris	Classical Harris corner detection algorithm.
PIPS	Industrial use case from the ARGO project [8]. Image processing pipeline for post-processing raw data from a polarized image sensor.
cnt	Counts non-negative numbers in a matrix
cover	Program for testing many paths.
crc	Cyclic redundancy check computation on 40 bytes of data.
des	DES and Triple-DES encryption/decryption algorithm.
duff	Using "Duff's device" from the Jargon file to copy 43 bytes array.
expint	Series expansion for computing an exponential integral function.
fdct	Fast Discrete Cosine Transform.
fir	Finite impulse response filter (signal processing algorithm) over a 700 items long sample.
jfdctint	Discrete-cosine transformation on a 8x8 pixel block.
lcdnum	Read ten values, output half to LCD.
ludcmp	LU decomposition algorithm.
matmult	Matrix multiplication of two 20x20 matrices.
ns	Search in a multi-dimensional array.
nsichneu	Simulate an extended Petri Net.
qurt	Root computation of quadratic equations.
sqrt	Square root function implemented by Taylor series.
statemate	Automatically generated code.
ud	Calculation of matrices.
covariance	Co-variance computation.
2mm	2 Matrix multiplications.
3mm	3 Matrix multiplications.
atax	Matrix transpose and vector multiplication.
bicg	BiCG sub kernel of BiCGStab linear solver.
doitgen	Multi-resolution analysis kernel (MADNESS).
mvt	Matrix vector product and transpose.
gemm	Matrix multiply.
gemver	Vector multiplication and matrix addition.
gesummv	Scalar, vector and matrix multiplication.
symm	Symmetric matrix-multiply.
syr2k	Symmetric rank-2k operations.
syrk	Symmetric rank-k operations.
trmm	Triangular matrix-multiply.
durbin	Algorithm for solving Yule-Walker equations.
lu	LU decomposition without pivoting.
ludcmp	Solving a system of linear equations using LU decomposition followed by forward and backward substitutions.
trisolv	Triangular solver.
floyd-warshall	Finds the shortest path in a graph.
nussinov	Algorithm for predicting RNA folding using dynamic programming.
adi	Alternating Direction Implicit solver.
fdtd-2d	2-D finite different time domain kernel.
heat-3d	Solving of heat equation over 3D space.
jacobi-1D	1-D Jacobi stencil computation.
jacobi-2D	2-D Jacobi stencil computation.
seidel-2D	2-D Seidel stencil computation.

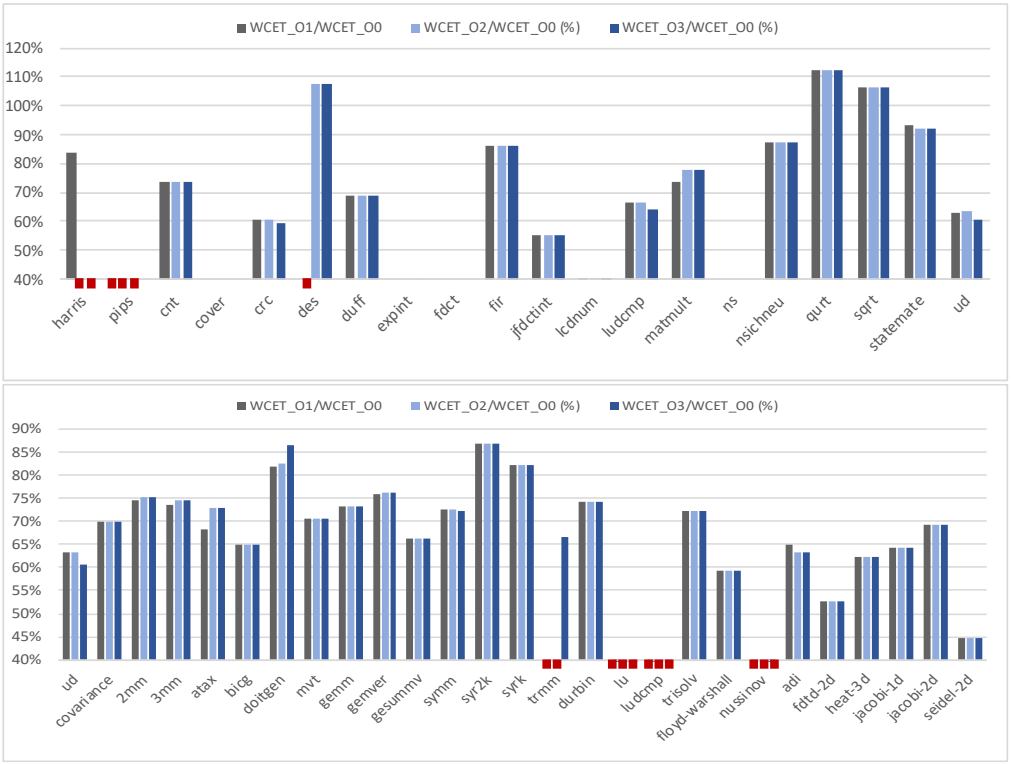


Figure 2 Ability to derive WCETs at -O1, -O2 and -O3.

generations of 100 individuals each (same number of distinct optimization sequences as *random exploration*). 15% of individuals are mutated at each generation.

In the *coarse-grain* case, all files are compiled using the same optimizations sequence, while *fine-grain* optimization uses different sequences for each file. Regarding the extension of the *genetic exploration* strategy to *fine-grain*, the implemented *mutation* operator mutates all files (with a different mutation per file). Similarly, we chose to apply the *crossover* operation to all files when breeding two individuals.

5 Experimental Evaluation

In this section, we look at the impact of standard optimizations levels on the WCET estimates, and then evaluate our *coarse-grain* and *fine-grain* optimization selection strategies.

5.1 Impact of Optimizations on the Ability to Derive Flow Information

LLVM comes with four optimization levels *-O0* (no optimization applied) to *-O3* (highly optimized code). Figure 2 gives for all optimizations levels beyond *-O0* (*-O1, -O2, -O3*) the ratio WCET_{-Oi} / WCET_{-O0} expressed in percentage (the lower the better). No bar for a given optimization level means that aiT was not able to detect loop bounds automatically.

The results show that in most situations, turning on optimizations results in lower WCET estimates than when compiling with option -O0 (ratio WCET_{-Oi} / WCET_{-O0} lower than 100 %). However, in some cases (benchmarks *qurt* and *sqrt* and *des*) optimized codes result in larger WCETs than non-optimized ones. For some benchmarks (*harris*, *pips*, *trmm, lu*, *ludcmp*, *nussinov*), aiT was not able to extract loop bounds when optimizations are turned on

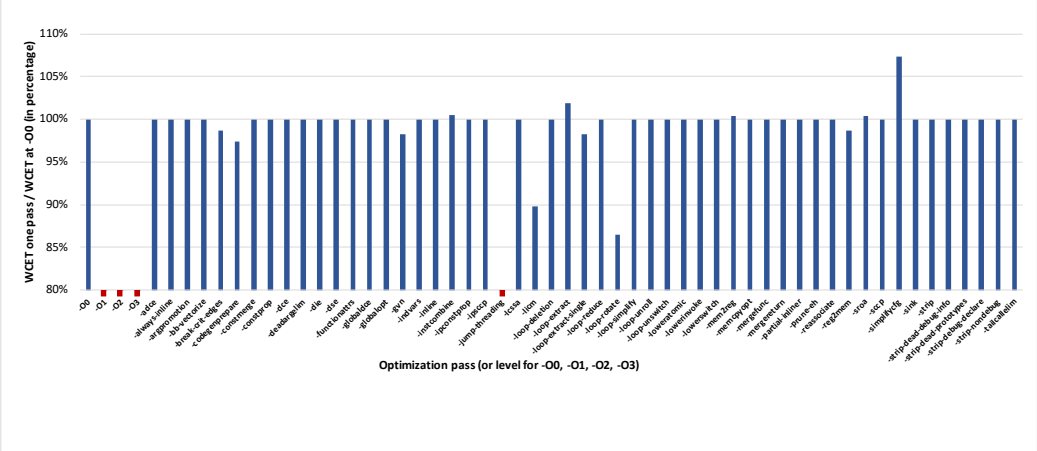


Figure 3 Impact of independent optimization passes on estimated WCET (application PIPS).

(depicted as red bars below the x-axis in the figure). Finally and surprisingly, for benchmark *trmm* aiT was not able to detect loop bounds at *-O1* and *-O2* but was able to estimate loop bounds at *-O3*. One can also note that the levels of optimization (from -O1 to -O3) have similar impact on estimated WCETs.

To detect which individual optimizations make automatic for loop bounds detection fail, we activated each optimization pass individually and estimated the resulting WCET using aiT. Experimental results are presented in Figure 3, that gives for each optimization WCET_{optim} / WCET_{-O0}, when optimization pass *optim* is activated on benchmark *pips*. Results show that optimization passes, even if only one of them is activated at a time, can significantly lower estimated WCETs, but may also have a negative impact. For example in benchmark *pips*, pass *-jump-threading*[4] makes aiT unable to estimate loop bounds; optimization pass *-loop-rotate* (classical loop rotation) reduces the estimate WCET of 15% and optimization pass *-simplifycfg* (dead code elimination and basic block merging) augments it of 7%.

Note that this experiment does not allow us to identify optimization passes that made aiT *systematically* fail to identify loop bounds. This is because activating an optimization pass does not imply that the optimization is actually triggered (e.g. pre-conditions are not always met). Passes *-gvn, -jmp-threading, -instcombine, -licm -mem2reg, -sroa, -lowerswitch* when activated made estimation of loop bounds fail for some programs. Pass *-jmp-threading* was the most harmful (aiT was unable to estimate a WCET on 20 benchmarks out of 46 when this optimization turned on).

We performed an in-depth analysis of the code generated by LLVM on a very small code snippet (simple loop initializing an array), with optimization *-jmp-threading* turned on, to identify why loop bound estimation fails on the generated code. It turns out that LLVM duplicates the loop induction variable: the first variable is used to index the array, whereas the second one is used in the loop exit test; the first variable is incremented, and then copied into the second one. The initial loop bound analysis of aiT was not able to demonstrate that the two induction variables were actually a single one and were equal at all times. The issue was fixed in aiT to correctly detect the loop bound when such a code is

[4] This pass looks at blocks that have multiple predecessors and multiple successors. If one or more of the predecessors of the block can be proven to always cause a jump to one of the successors, it forwards the edge from the predecessor to the successor by duplicating the contents of this block.

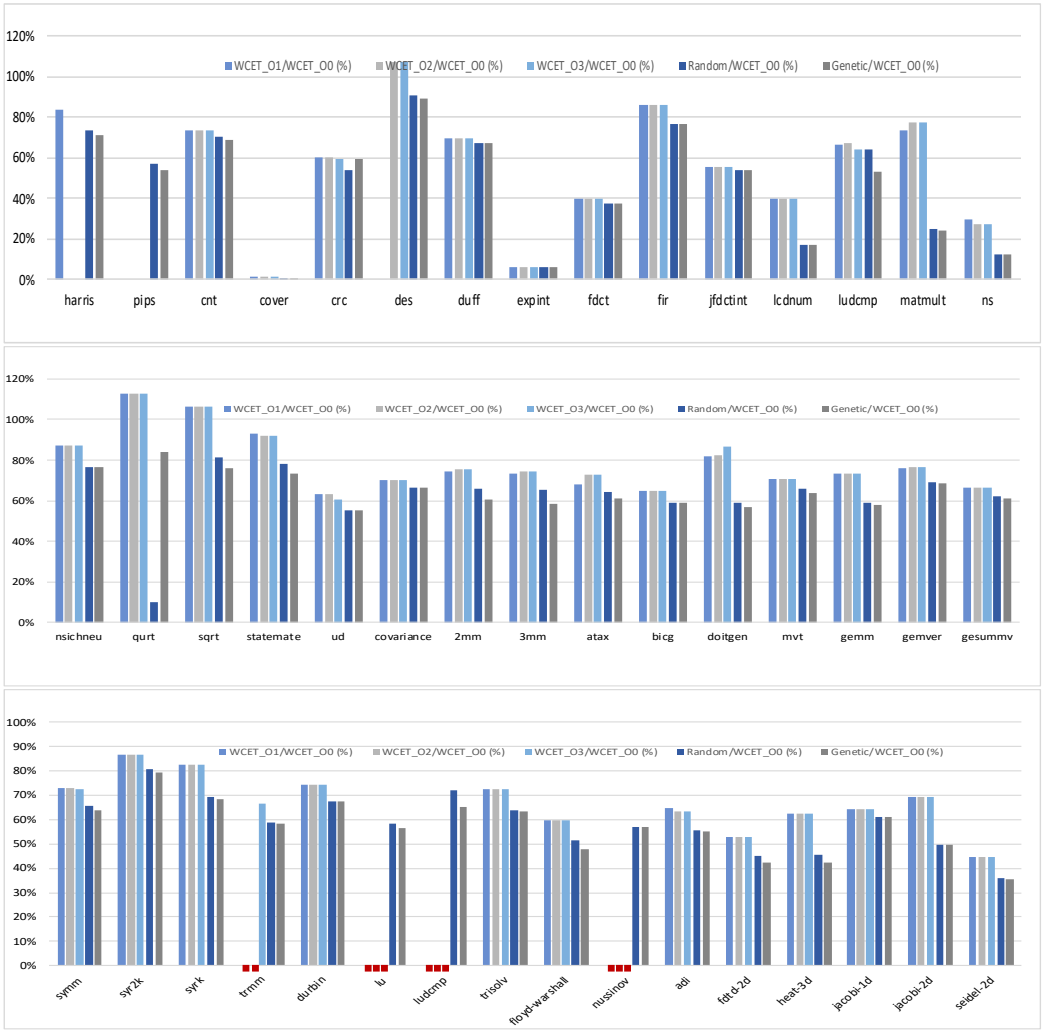


Figure 4 Coarse grain exploration of optimization space.

generated. Understanding why the optimization passes other than *-jmp-threading* generate hard-to-analyze code is left for future work.

5.2 Evaluation of Coarse-Grain Optimization Selection Strategies

Figure 4 presents the experimental results for all benchmarks. The first important remark on the experimental results is that for all benchmarks aiT was able to derive loop bounds automatically, even in the situations where some optimization levels made it impossible before (benchmarks *harris, pips, trmm, lu, ludcmp, nussinov). On all benchmarks, exploring the optimization space using random exploration resulted in WCET estimates lower than the best WCET possible with -O1, -O2 and -O3. The gain is most of the time significant (21% on average as compared with the best optimization level). Finally, except for benchmarks crc and qurt, genetic exploration outperforms random exploration. Preliminary experiments with genetic exploration* made us select large populations and low number of generations, that turned out to give better WCET estimates than lower population sizes and larger number of generations.

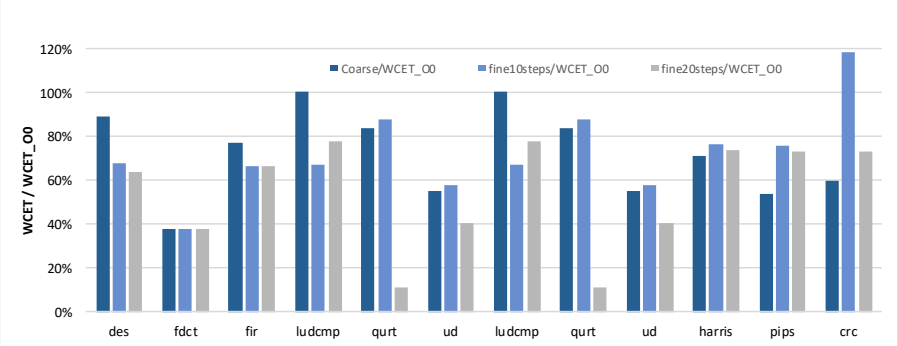


Figure 5 Fine grain exploration of optimization space.

5.3 Evaluation of Fine-Grain Optimization Selection Strategies

Due to time constraints, experiments of the proposed fine-grain optimization strategies were conducted on the image processing benchmarks *harris*, *pips* and the Mälardalen benchmarks only. We further restricted the benchmarks to the ones containing loops, and avoided those containing a single loop that covers the entire code. Results are given in Figure 5, showing three values for each benchmark: $\text{WCET}_{coarse}/\text{WCET}_{-O0}$, $\text{WCET}_{fine10steps}/\text{WCET}_{-O0}$ (10 generations) and $\text{WCET}_{fine20steps}/\text{WCET}_{-O0}$ (20 generations instead of the default value of 10).

The first results obtained are encouraging (improvement of 37 % of the WCET estimates on average). On 6 out of the 9 benchmarks analyzed (all but the 3 ones at the right of the figure), the fine-grain *genetic exploration* outperforms the coarse-grain exploration. Moreover, for all benchmarks except one, having 20 generations instead of 10 significantly improves WCETs, at the cost of an analysis time twice longer. This result is expected, since the optimization space to be explored is much larger than for the coarse-grain strategy. We believe there is room left for improvements, by tuning the parameters of the genetic algorithm to better deal with the very large optimization space to be explored, or avoid the cost of outlining when not beneficial to the WCET.

6 Conclusion

Compiler optimizations are known to add challenges when estimating the WCET of applications. Hence it is quite common to disable them when dealing with critical systems. In this paper, we proposed an iterative compilation workflow to reconcile timing critical applications with compiler optimizations. Our methods, based on optimization space exploration, show a significant tightening of the estimated WCETs. Our first exploration of fine-grain application of optimizations demonstrated opportunities to further reduce WCET estimates (improvement of 37 % of the WCET estimates on average). Future work is still needed to take full benefit of fine-grain exploration of optimizations. A first direction is to better explore the very large optimization space, for example by concentrating the optimization effort of regions having the most impact on worst-case performance. Another direction is to develop techniques to better select the code snippets to be outlined: outlining has a cost (extra function call and parameter passing) that has to be avoided when outlining is not beneficial to the WCET. Symmetrically, we still need to explore which code sequences would benefit from being outlined and *not* optimized, such that manual source-level annotations can be given when more beneficial to WCET estimates than compiler optimizations and automatic flow fact extraction.

References

1 aiT:the industry standard for static timing analysis. http://www.absint.com/ait.

2 Felix V. Agakov, Edwin V. Bonilla, John Cavazos, Björn Franke, Grigori Fursin, Michael F. P. O'Boyle, John Thomson, Marc Toussaint, and Christopher K. I. Williams. Using machine learning to focus iterative optimization. In *Fourth IEEE/ACM International Symposium on Code Generation and Optimization (CGO 2006), 26-29 March 2006, New York, New York, USA*, pages 295–305, 2006.

3 Guillem Bernat and Niklas Holsti. Compiler support for WCET analysis: a wish list. In *Proceedings of the 3rd International Workshop on Worst-Case Execution Time Analysis, WCET 2003 - a Satellite Event to ECRTS 2003, Polytechnic Institute of Porto, Portugal, July 1, 2003*, pages 65–69, 2003.

4 Yang Chen, Shuangde Fang, Yuanjie Huang, Lieven Eeckhout, Grigori Fursin, Olivier Temam, and Chengyong Wu. Deconstructing iterative optimization. *TACO*, 9(3):21:1–21:30, 2012. doi:10.1145/2355585.2355594.

5 Keith Cooper and Linda Torczon. *Engineering a compiler*. Morgan Kaufmann, 2011.

6 Keith D. Cooper, Alexander Grosul, Timothy J. Harvey, Steven W. Reeves, Devika Subramanian, Linda Torczon, and Todd Waterman. ACME: adaptive compilation made efficient. In *Proceedings of the 2005 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'05), Chicago, Illinois, USA, June 15-17, 2005*, pages 69–77, 2005.

7 Christoph Cullmann and Florian Martin. Data-Flow Based Detection of Loop Bounds. In Christine Rochange, editor, *7th International Workshop on Worst-Case Execution Time Analysis (WCET'07)*, volume 6 of *OpenAccess Series in Informatics (OASIcs)*, Dagstuhl, Germany, 2007. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/OASIcs.WCET.2007.1193.

8 Steven Derrien, Isabelle Puaut, Panayiotis Alefragis, Marcus Bednara, Harald Bucher, Clement David, Yann Debray, Umut Durak, Imen Fassi, Christian Ferdinand, Damien Hardy, Angeliki Kritikakou, Gerard K. Rauwerda, Simon Reder, Martin Sicks, Timo Stripf, Kim Sunesen, Timon D. ter Braak, Nikolaos S. Voros, and Jürgen Becker. Wcet-aware parallelization of model-based applications for multi-cores: The ARGO approach. In *Design, Automation & Test in Europe Conference & Exhibition, DATE 2017, Lausanne, Switzerland, March 27-31, 2017*, pages 286–289, 2017.

9 Heiko Falk, Peter Marwedel, and Paul Lokuciejewski. Reconciling compilation and timing analysis. In *Advances in Real-Time Systems (to Georg Färber on the occasion of his appointment as Professor Emeritus at TU München after leading the Lehrstuhl für Realzeit-Computersysteme for 34 illustrious years).*, pages 145–170, 2012.

10 A. Floc'h, T. Yuki, A. El-Moussawi, A. Morvan, K. Martin, M. Naullet, M. Alle, L. L'Hours, N. Simon, S. Derrien, F. Charot, C. Wolinski, and O. Sentieys. Gecos: A framework for prototyping custom hardware design flows. In *Source Code Analysis and Manipulation (SCAM), 2013 IEEE 13th International Working Conference on*, pages 100–105, Sept 2013.

11 Raimund Kirner, Peter P. Puschner, and Adrian Prantl. Transforming flow information during code optimization for timing analysis. *Real-Time Systems*, 45(1-2):72–105, 2010. doi:10.1007/s11241-010-9091-8.

12 Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *International Symposium on Code Generation and Optimization*, pages 75–88, San Jose, CA, USA, Mar 2004.

13 Thomas Lefeuvre, Imen Fassi, Christoph Cullmann, Gernot Gebbard, Emin Koray Kasnakli, Isabelle Puaut, and Steven Derrien. Using polyhedral techniques to tighten wcet estimates of optimized code: a case study with array contraction. In *Design, Automation*

& Test in Europe Conference & Exhibition, DATE 2018, Dresden, Germany, March 19-23, 2018, 2018.

14 Hanbing Li, Isabelle Puaut, and Erven Rohou. Traceability of flow information: Reconciling compiler optimizations and wcet estimation. In *Proceedings of the 22Nd International Conference on Real-Time Networks and Systems*, RTNS '14, pages 97:97–97:106, New York, NY, USA, 2014. ACM.

15 Hanbing Li, Isabelle Puaut, and Erven Rohou. Tracing flow information for tighter WCET estimation: Application to vectorization. In *21st IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA 2015, Hong Kong, China, August 19-21, 2015*, pages 217–226, 2015.

16 Chunhua Liao, Daniel J Quinlan, Richard Vuduc, and Thomas Panas. Effective source-to-source outlining to support whole program empirical optimization. In *Languages and Compilers for Parallel Computing*, pages 308–322. Springer, 2010.

17 Paul Lokuciejewski, Sascha Plazar, Heiko Falk, Peter Marwedel, and Lothar Thiele. Approximating pareto optimal compiler optimization sequences - a trade-off between wcet, ACET and code size. *Softw., Pract. Exper.*, 41(12):1437–1458, 2011. doi:10.1002/spe.1079.

18 Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per 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.