Winston: Revisiting iterative compilation for WCET minimization

Valentin Pasquale, Isabelle Puaut

Winston: Revisiting iterative compilation for WCET minimization

Valentin Pasquale
ENS Lyon
France
valentin.pasquale@ens-lyon.fr

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

ABSTRACT

Static Worst-Case Execution Time (WCET) estimation techniques take as input the binary code of a program and output a conservative estimate of its execution time. While compilers, and iterative compilation, usually optimize for the average-case, previous work such as [7, 23] has shown that it is also possible to use existing optimization and iterative compilation techniques to lower the WCET estimates drastically.

In this paper, we revisit the use of iterative compilation for WCET minimization and show that previous work can be improved both in terms of complexity and reduction of WCET estimates. In particular, we found that the use of long chains of compilation flags, from a few hundred to a few thousand, allows a significant reduction of WCET estimates, of 35% on average, and up to 70% on some benchmarks, compared to the best compilation level (-O0 .. -O3) applicable. These gains are significantly better than the reductions of WCET estimates obtained by [7], which, on the same benchmarks and experimental conditions, reduce the WCET estimates by 20% on average.

KEYWORDS

WCET estimation, compilation, compiler optimizations, iterative compilation

ACM Reference Format:
Valentin Pasquale and Isabelle Puaut. 2022. Winston: Revisiting iterative compilation for WCET minimization. In *Proceedings of the 30th International Conference on Real-Time Networks and Systems (RTNS '22), June 7–8, 2022, Paris, France.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3534879.3534899

1 INTRODUCTION

Obtaining safe upper bounds on the execution time of programs (WCET, for Worst-Case Execution Time) is crucial in real-time systems to prove that tasks are completed before their deadline. A way to estimate this value is to use static WCET estimation techniques [26] that provide safe upper bounds using the program's binary code.

Reducing WCET estimates motivates the use of optimizing compilers. However, optimization passes in compilers are not designed for real-time systems. Some of them may complicate static WCET estimation. For example, they can make loop-bound estimation impossible. Some optimizations may also worsen the WCET estimate, whereas it improves the average-case execution time. The sequence of optimizations that minimize the WCET estimate of a code is therefore not necessarily the same as the one minimizing average-case performance [9], motivating research on WCET-oriented compilation.

Compilers come with standard optimization levels (*-O1* to *-O3*, *-O0* meaning non-optimized code). Previous work in the compiler community has shown that better performance than the one obtained with the *-O3* level can be obtained by using *iterative compilation* [2, 5, 10, 14]. Iterative compilation techniques generate many program versions with different sequences of optimization flags. The different program versions are then executed and their execution time is measured to choose the sequence with the lower execution time. The number of different optimization flags in compilers (more than 100 in *gcc*) is large, and therefore the number of possible optimization sequences is huge, in particular, given that the same optimization flag can appear several times in a sequence. Metaheuristics (e.g., genetic algorithms) or machine-learning-based algorithms, given the enormous size of the search space, are appropriate techniques to generate sequences of optimization flags based on previously tested sequences. Previous research on iterative compilation has shown that the optimization sequences that minimize execution time are application-dependent, making standard optimization levels *-O1* to *-O3* sub-optimal. Moreover, an optimization sequence well-suited to one code is not necessarily well-suited to other types of code. Finally, the huge size of the optimization space makes it difficult to manually introspect for optimization sequences that are well suited to a given code.

The fundamental difference between standard (performance-oriented) iterative compilation and WCET-oriented iterative compilation is that the metric to be optimized for the latter is the WCET. As the exact WCET of a given code is in general unknown, an upper bound of the WCET, the *WCET estimate*, is produced instead. Static WCET estimation tools have to infer flow information (e.g., loop bounds, infeasible paths) to estimate WCETs. It may happen that certain sequences of optimization flags while resulting in significant performance improvement, strongly affect the structure of the code and render static WCET estimation tools unable to accurately (or simply) infer flow information. WCET-oriented iterative compilation not only determines the sequence of optimization flags that results in the lowest WCETs, but also excludes optimization sequences that render the tools unable to estimate WCETs accurately or simply estimate them.

RTNS '22, June 7–8, 2022, Paris, France

ACM ISBN 978-1-4503-9650-9/22/06…$15.00
https://doi.org/10.1145/3534879.3534899

The use of iterative compilation for minimizing WCET estimates instead of average execution times was extensively explored in [7]. A new technique was proposed in [7] to generate optimization sequences based on *learning* the impact of individual optimization passes on the WCET estimate. In this paper, we propose Winston, for Wcet mInimizatioN uSing iTerative cOmpilatioN, a new technique for WCET-oriented iterative compilation that outperforms [7] both in terms of complexity and WCET reduction. The better behavior of Winston as compared to [7] is based on: (i) the observation that very long optimization sequences result in lower WCET estimates than the ones produced by [7]; (ii) the introduction of a *cache*, that avoid re-calculating the WCETs several times for the same optimization sequences, and therefore allows to better explore the optimization search space, for a given search time budget; (iii) the proposal of a new algorithm for searching optimization sequences results, well suited to the use of long optimization sequences.

Note that given the huge size of the search space of optimization sequences, Winston is a *best-effort* algorithm. Winston does not guarantee finding the optimal optimization sequence, which would require an exhaustive search space traversal.

The contributions of this paper are the following:

- We show that the use of long chains of compilation options, from a few hundred to a few thousand, allows a significant reduction of WCET estimates. These long chains of compilation flags can afterward be shortened by safely removing useless flags.
- We show that using a cache of previously analyzed optimization sequences speeds up the search for the sequence that results in the lower WCET estimate.
- We present a new algorithm, named Winston, for Wcet mInimizatioN uSing iTerative cOmpilatioN, that iteratively generates long and varied optimization sequences.
- We give an extensive performance evaluation of Winston, using the same experimental setup as in [7] (benchmark suites, WCET estimation tool – aiT [22] –, target processor – LEON3). Experimental results show a reduction of WCET estimates of 35% on average and up to 70% on some benchmarks, compared to the best compilation level applicable. These gains are far better than the WCET reductions obtained by [7], which, on the same benchmarks and experimental conditions, reduce the WCET estimate by 20% on average.

The outline of this paper is as follows. Section 2 first presents the general workflow of WCET minimization using iterative compilation as defined and implemented in [7]. Section 3 presents preliminary findings, in particular the fact that long optimization chains allow for a drastic reduction of WCET estimates for some codes. We present in Section 4 Winston, a new iterative compilation algorithm that aims at reducing WCET estimates based on these preliminary findings. An experimental evaluation of Winston is presented in Section 5. We compare Winston to related work in Section 6 and give concluding remarks in Section 7.

2 BACKGROUND ON WCET-ORIENTED ITERATIVE COMPILATION

The typical workflow of WCET-oriented iterative compilation, as defined and used in [7], is depicted in Figure 1. Given a program in a high-level language and a compiler (LLVM) for the target architecture (LEON3), the approach first compiles the program and analyzes it with a WCET estimation tool (the industry-standard WCET estimation tool aiT [22]). Subsequently, a new optimization sequence is generated. The program is then recompiled and analyzed, resulting in a new WCET estimate. This iterative process is repeated until a given number of calls to the WCET estimation tool aiT has been made.

In [7], no flow information is manually given to aiT to augment the precision of WCET estimation. This choice was taken because compiler optimizations modify flow information. Therefore, manually-provided flow information could be incorrect unless a traceability mechanism such as the one proposed in [15], which unfortunately is not available in standard compilers, is used. Thus, some WCET estimates may not be as precise as if manual annotation of flow information was provided.

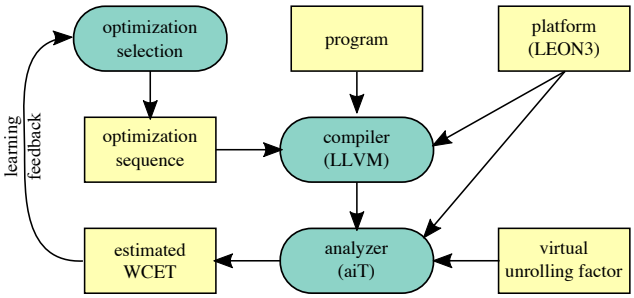


Figure 1: Iterative WCET-oriented compilation workflow (from [7])

The aiT WCET estimation tool uses a parameter named *virtual unrolling factor* to trade off WCET estimation precision with analysis complexity. The virtual unrolling factor defines the number of contexts used to analyze loops. The higher the virtual unrolling factor, the more precise the WCET estimate. On the other hand, the higher the virtual unrolling factor, the longer the analysis time and consumed memory. In [7], the following process is used to select the unrolling factor for each benchmark: for each benchmark, we start from a low value and increase it as long as the analysis runs in a reasonable time (less than 30 seconds) and as long as it increases the precision (see [7] for details).

Four techniques are presented in [7] to generate sequences of optimization passes:

- *Association:* this technique is based on a ranking of the impact of individual optimization passes in a sequence on the WCET estimate (as *positive, negative, neutral*). This ranking is obtained by analyzing previously generated optimization sequences. More precisely, the technique uses a lattice of the sets of optimization passes, with set inclusion as partial order in the lattice. A new optimization flag in a long sequence is evaluated (as *positive, negative, neutral* within the sequence)

by comparing the estimated WCET of the long sequence with the largest subset of previously evaluated sequences of the lattice. The classification of a flag impacts the probability of selecting it in future generated sequences, promoting flags with positive rankings over flags with neutral or negative rankings. By construction, by using sets to represent optimization sequences, [7] disregards the ordering of flags in the optimization sequence and the multiple occurrences of a given flag in a sequence.

- *Cleaning:* this technique, based on *Association* removes from the execution sequence generated by *Association* the optimization passes that do not improve the WCET estimate. *Cleaning* is shown in [7] to outperform *Association*. *Cleaning* results in an average reduction of WCET estimates of 20% compared to the best optimization level applicable.
- *Random* and *Genetic* as baselines that naively generate optimization sequences. *Genetic* was shown in [7] to outperform *Random*.

Since the impact of a given optimization pass depends on which pass was applied previously in the optimization sequence, a sequence in [7] may contain multiple occurrences of the same pass.

The work presented in [7] used four different benchmark suites. The first one is the Mälardalen benchmark suite [11] which is a reference for WCET estimation techniques[1]. The second one is the Polybench suite [21], which contains a lot of linear algebra codes. Few benchmarks also come from the MiBench suite[12] and the PolyMage suite[19].

3 PRELIMINARY FINDINGS

Two preliminary findings, respectively presented in Sections 3.1 and 3.2, allowed us to outperform [7]: the fact that long optimization sequences result in lower WCET estimates as compared to shorter sequences, and the use of a cache solution to speed-up the iterative compilation process.

3.1 Longer optimization chains result in lower WCET estimates

To evaluate the impact of the length of sequences, we run the *Random* and *Genetic* algorithms of [7], both with sequences of maximum sizes 50 and 1000. We compared them on a subset of the Polybench benchmark suite that runs in a reasonable time (which is the full set of the Polybench suite, except *3mm*, *fdtd-2d*, *gesummv*, *jacobi-1d*, and *jacobi-2d*). We performed these preliminary experiments on the Polybench benchmarks because they are the largest programs analyzed in [7] (much larger than those in the Mälardalen/Heptane benchmark suite) and also because they contain mainly loop nests and, as such, are more sensitive to loop optimization than benchmarks from the Mälardalen/Heptane suite.

Results show that long sequences lead to lower WCET estimates than when using short sequences, with an average decrease of 10%. Sometimes, there are cases where the WCET estimate is significantly lower than the usual WCET estimation found. Let us take the example of the benchmark *2mm*, and let us denote by $\text{WCET}_{bestopt}$ the WCET estimate when the best optimization level is used. The WCET estimate of *2mm* when using long sequences was 40% of $\text{WCET}_{bestopt}$, whereas when using small sequences it was 90% of $\text{WCET}_{bestopt}$. Even by removing these few outliers where the WCET estimate is significantly better, on average, it is still better, however by only 2% of the WCET estimate of the best optimization applicable.

We analyzed the compilation sequence resulting in the lower WCET, for each algorithm on each benchmark. As already observed in [7], a lot of options in generated large sequences are useless. To eliminate these useless options, we tested each option in the optimization sequence one by one. If removing the option results in a WCET estimate lower or equal than when the option is in the sequence, this one is removed. Otherwise, it is kept. We obtain a "minimal" sequence of options by iterating this process. The minimal sequence we obtain from the algorithms that use large sequence ranges from 20 to 30 options, which is far longer than minimal sequences obtained by the same algorithms with small sequences of options that range from 5 to 10.

In summary, generating long sequences of optimizations, even if most of them are useless and can be safely removed, result in lower WCET estimates than when using shorter sequences and longer minimal sequence that will not be found otherwise.

One may wonder why long sequences work better if only 30 flags of them are sufficient to obtain the same result. Our intuition is that out of all sequences of size below 50, few result in a significant reduction of the WCET estimate (few is relative to the number of options), so they are incredibly hard to find. Besides, inserting new flags in the sequence has minor impact on the WCET estimate: if a sequence of size 50 works well, then inserting random flags, even if we insert 1000 flags, rarely damages the WCET. So by testing a sequence of 1000 flags, we are actually testing if one of the subsequences of size 50 works well. Hence, testing sequences of size 1000 test largely more sequences than testing sequences of size 50, therefore, statistically we will often find a sequence of size 50 hidden in a sequence of size 1000 that works well.

Moreover, it seems that classical search algorithms cannot find these short sequences that work well because the neighbors of a good solution are not systematically good solutions (also, these neighbors are very rare and hard to find), however, this is not the case for long sequences because most flags are useless. This may also explain why meta-heuristic algorithms work better when they use long sequences and why it seems hard to design an algorithm to find the rare sequences of size below 50 that works very well. In addition, in section 5.2 we explain that the compiler sometimes generates incorrect code when we use long sequences. Maybe the same phenomenon is involved, one could also probably isolate a sequence of size below 50 that generates incorrect code too, yet taking randomly 50 options rarely produces incorrect code while taking 1000 options does this with a higher probability.

Further observations on the impact of long sequence lengths on WCET estimates.

Further observations on the impact of long sequences of optimizations on WCET estimates are given below and are illustrated in Figure 2. Two sequences of optimizations were generated. The

[1]The actual source code used is the benchmark suite coming with the Heptane static WCET analysis tool [13], very close to the Mälardalen benchmark suite. This code will be named Mälardalen/Heptane in the following.

first one, named $SO1$ in the following, is of length 250 and is used in Figures 2a and 2b. The second one, named $SO2$, is of length 6000, and is used in Figure 2c. Both sequences were obtained by a variation of the *Random* algorithm, which appends random sequences of options to the best optimization sequence already found (and rotates the best optimization sequence used to avoid being blocked in a local minimum like [4]). The benchmark under analysis in the Figure is *qurt*.

Figure 2a first depicts the evolution of the WCET estimates when only the first x optimizations of $SO1$ are applied, with x in the interval [0..250]. The figure shows that these short optimization chains may result in modifications of the WCET in non-significant ways or even make the WCET analysis fails, which is represented by a value of -1 and depicted by red crosses in the figure. However, after applying several options from $SO1$, some options significantly reduce the WCET estimate. This shows that some optimizations, although not decreasing the WCET estimate themselves, allow other optimizations placed further in the sequence to decrease the WCET estimate significantly. This Figure shows that not only the set of optimization matters but also their ordering in the optimization chain.

Figure 2b further illustrates the impact of the optimizations sequence order. The figure depicts the impact on the estimated WCET when applying x options (from index 150 to 150 + x in $SO1$) to a program that was at first compiled with *-O0*. In contrast to Figure 2a, which applied options from indices 0 to 149 before applying the x optimizations, Figure 2b shows that applying the x optimizations results in a bad WCET estimate, even worse than when compiled with *-O0*. These results show that long optimization chains may be required to allow further optimizations in the sequence to be effective.

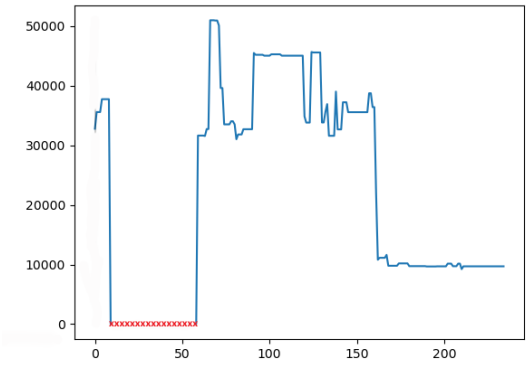
Finally, Figure 2c shows the evolution of WCET estimates when only the first x optimizations of $SO2$ are applied, with x in the interval [0..6000]. The graph is similar to the first one, lousy WCET estimates are found along the way, and once the minimum is found, the WCET estimate does not vary much. This effect is observed with large sequences in general, independently of their size.

These observations show that long options sequences may significantly allow further optimizations to decrease WCET estimates. Most papers on iterative compilation take for granted that the list of options to be explored is short. The observations reported in this section show the opposite.

3.2 Caching WCET estimates helps

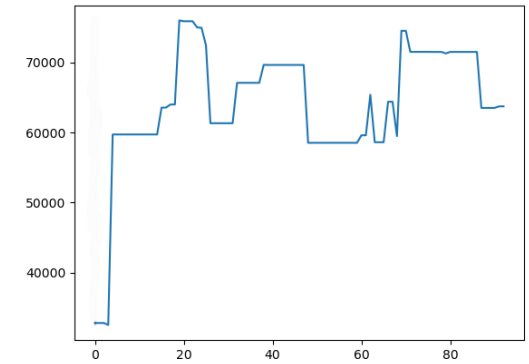
To speed up the search for the optimization sequence resulting in the lower WCET estimate, we used a *cache*. This cache stores for each optimization sequence the resulting WCET estimate. It, therefore, maps strings, which represent an optimization flag sequence, to integers, which represent the WCET estimates. Each time an optimization sequence is evaluated, the resulting WCET estimate is stored in the cache. If the same sequence is evaluated again, then the WCET estimate cached by the first evaluation is used, avoiding compiling the code and calling the WCET estimation tool.

We observed that when using *Random* selection of optimization sequences, the cache has a negligible impact since the probability



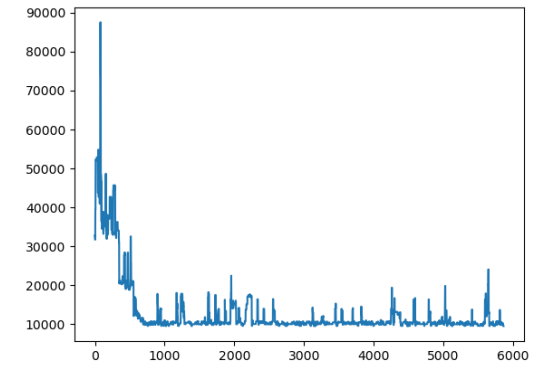
wcet (3).png

(a) Evolution of the WCET estimate when only the first x options are taken (optimization sequence $SO1$ of size 250).



wcet (2).png

(b) Evolution of the WCET estimate when only the options from index 150 to 150+x are taken (optimization sequence $SO1$ of size 250).

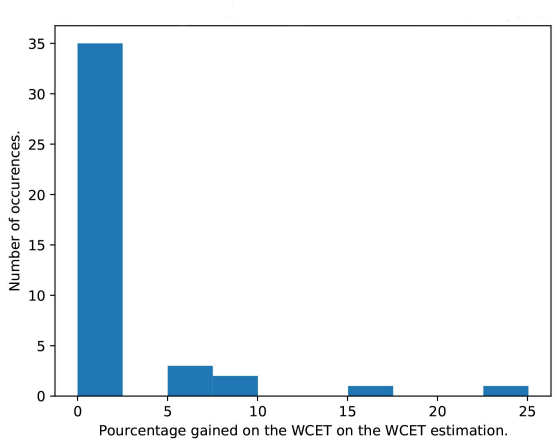


wcet (1).png

(c) Evolution of the WCET estimate when the first x option are taken (optimization sequence $SO3$ of size 6000).

Figure 2: Impact of number of optimizations and position in optimization chain on the WCET estimate. A value of -1 means the analyzer fails to provide a WCET estimate.

Figure 3: Impact of the cache on Winston



of evaluating the same sequence several times is very low. However, for the algorithms that may explore over and over the same solutions, for instance, the *Genetic* algorithm, it has, on average, on all benchmarks with a cache of unbounded size, 10% of cache hits. This 10% of cache hits enlarge the search space of optimizations by 10% and for a fixed number of calls to the WCET estimation tool allows further WCET estimate reductions.

Figure 3 details how the introduction of a cache impacts the WCET estimates found by Winston, the iterative compilation algorithm we propose, and detail in Section 4. These experiments were performed by applying Winston on all benchmarks used in the evaluation of Winston, given in Table 1, with a number of calls to aiT set to 1000, and no limit on the cache size [2]. The figure shows the histogram of the gain brought by the caching mechanisms on the WCET estimates for a fixed number of calls to the WCET estimation tool (the cache allows a larger exploration of the space of optimization sequences by avoiding useless WCET estimations. When there is enough locality in the search (7 experiments out of 42, i.e., 17%), the same sequence is explored several times, and the WCET estimate is decreased (by up to 25%). When the search does not have locality, there is no improvement of the WCET estimate (but also almost no overhead, the time overhead for accessing the cache is negligible compared to the WCET estimation delay).

Globally, caching optimization sequences WCET estimates help to reduce the future WCET estimation, at least for search algorithms that exhibit some locality in their search.

4 WINSTON: ITERATIVE COMPILATION WITH LONG OPTIMIZATION SEQUENCES

Winston is inspired by the work presented in [18] which performs iterative compilation with the Nelder–Mead method [20]. The Nelder–Mead technique is a heuristic technique, often used in nonlinear optimization problems, that finds the minimum of an objective function in a multidimensional space. The concept of simplex in [18] is a geometrical figure consisting of n+1 points in n-dimensions (in our case a simplex is a sequence of optimization flags). Through a sequence of geometric transformations (*reflection, expansion, contraction, shrink* as defined in [18]), the initial simplex moves towards minimum and away from maximum. The idea behind [18] is that the geometric transformations proposed in the Nelder–Mead technique will give better results than those of genetic algorithms that use randomly generated values when performing mutation and cross-over.

Our adaptation of [18] to long optimization sequences in Winston makes Winston very similar to a genetic algorithm [6] that would only do cross-over operations and no mutation. Indeed, since we aim at generating large sequences of varied compilation options, the mutation operator of the genetic algorithm does not seem appropriate to our case because it would have an insignificant impact in the context of long optimization sequences. In the same way, trying to understand which option contributes to the lower WCET estimate like the best algorithm of [7] would be too time-consuming as there are too many compilation options. Moreover, the idea described in [18] was interesting and it seemed to work with a large sequence of options, it creates new points to explore from previous ones, and no knowledge is required about them, which is exactly what we are dealing with.

More precisely, Winston first creates a few random solutions by choosing a sequence length between 10 and 1000, and then picks optimization flags within the sequence randomly. The selected random sequences form the initial pool of known good solutions. Then, at each iteration, Winston takes two random solutions from this pool, combines them to create a new one, and similarly to [18] adds it to the pool if the resulting WCET estimate is lower. The process of combining two sequences to form a new one executes one of the following procedures with equal probability:

- Concatenation of two sequences. For instance the concatenation of "-loop-rotate -sink" and "-inline -sink" is "-loop-rotate -sink -inline -sink", with "-sink" now being duplicated.
- Intersection of two sequences: the result is the first sequence, where each element that is not on the second sequence is removed. For instance the intersection of "-loop-rotate -sink" and "-inline -sink" is "-sink". This can result in an empty optimization sequence.
- Concatenation of two subsets of two sequences. To pick a subset of options, we pick two indexes randomly between 0 and the sequence size. For instance the concatenation of "-loop-rotate -sink" and "-inline -sink" is "-loop-rotate -sink -sink", or "-sink -inline -sink".

There is no limit on the size of sequences, hence many ones are explored. We tried to fine-tune the probabilities to select the combination procedure, but it makes only a tiny difference (or no difference at all).

Note that similarly to [7] and as shown in the examples above, an optimization sequence may contain multiple occurrences of the same optimization flag.

[2] This choice of not limiting the cache size was motivated by the selected number of calls to the WCET estimation tool aiT (1000), which was low enough to have a reasonable number of entries in the cache and consequently fast lookup.

5 EXPERIMENTAL EVALUATION

Our experimental setup, largely inspired from [7] is presented in Section 5.1. Section 5.2 then compares the WCET estimates obtained using Winston to those obtained by the state-of-the-art iterative compilation techniques, and details the results on the benchmarks we have analyzed.

5.1 Experimental setup

Our experimental setup is, unless otherwise stated, the same as in [7]. The target processor is a 32-bit RISC LEON3 processor, with separate 16 KB 2-way instruction and data caches. Programs are compiled with the same compiler as in [7]: LLVM 4.0 for the SPARC architecture, using Gaisler BCC 2.0.5 implementation [1]. The LLVM optimizer *opt* was applied separately, and all its 53 optimizations passes were used in the experiments.

Timing analysis is carried out by AbsInt aiT [22] version 19 targeting the LEON3 architecture (the version of absInt aiT is therefore a slightly more recent version than used in [7]). Similarly to [7], sequences that make the compiler crash or infinitely loop are disregarded. The same process as in [7] is used the *virtual unrolling factor* used by aiT (see Section 2).

One slight difference with the experimental setup used in [7] is the introduction of a correctness check of compiled programs. This check is motivated by the observation that some programs are compiled successfully but produce incorrect results, probably due to untested long sequences of optimization passes. The correctness check is performed by comparing the program's output with the one obtained without optimization flags. For the sake of simplicity and efficiency, we compile the LLVM IR on the processor of the machine running the experiment, which therefore compiles the LLVM IR to x86, instead of executing a LEON3 interpreter to get the output of the program. The correctness check is performed on the input data provided with the code of the benchmarks. As further discussed in Section 5.3, this very simple correctness check does not capture all incorrectly generated code. A more complex correctness check (targeting LEON3 code and exploring varied input data) would have to be performed on the solutions with the lower WCET estimates, in order to remove the ones with incorrect code (if any). Winston was evaluated on the benchmarks listed in Table 1. They come from the Mälardalen WCET benchmark suite as modified for their use in the Heptane static WCET analysis tool [13], and the Polybench benchmark suite [21] for linear algebra, physics and statistics. The benchmarks we have evaluated are a large subset of those used in [7][3].

5.2 Experimental results

We now compare Winston to the best algorithm of [7] (named *Cleaning* in the original publication), to a *Random* algorithm operating on sequences of maximum size 1000, and to a *Genetic* algorithm operating with sequences of maximum size 1000. Based on our preliminary findings, *Random* and *Genetic* are configured to generate longer sequences than in [7], in which a maximum sequence length of 50 was used. The *Cleaning* algorithm operates on sequences of size 50, and Winston, by construction, has no limit on the size of sequences it manipulates. All algorithms, except the *Cleaning* algorithm, have a cache implemented, as previously described in section 3.2. All techniques iterate until the WCET estimation tool aiT is called 1000 times.

The results are given in Figure 4, which represents the evolution of the results of *Random, Genetic, Cleaning* and Winston over the 1000 iterations. The x-axis of the figure is the number of non-cached calls to the WCET estimation tool aiT. The y-axis of the figure is the geometric mean over the benchmarks of the ratio between the estimated WCET, with the technique under consideration, over the estimated WCET with the best optimization level applicable (-O0..-O3). A value lower than 1 therefore means that the estimated WCET is improved. Only the very first iterations may result in WCET estimates worst than the best optimization level applicable. The geometric mean is used as it is employed in [7], and also because it amplifies the effect of benchmarks where a significant reduction of the WCET estimate is found.

Figure 4 shows that Winston provides better WCET estimates than all other algorithms of the state-of-the-art, even better than a naive random/genetic algorithm that uses a large sequence too. The fact that both the random and genetic algorithms outperform the state-of-the-art is due to the use of long sequences. We tried to test the cleaning algorithm with large sequences to provide a better comparison but it does not work well as it tries to learn which options work best, which is unpractical when 1000 options are used, and it tries to clean the sequence from unnecessary options, which is also unpractical with large sequences.

The runtime of Winston is very similar to the one reported in [7] (around 10000 seconds on average on all the benchmarks). Notwithstanding that some methods have more overhead than others (especially when it comes to the time-consuming task of compiling the code with large sequences of options), the WCET estimation time is always longer than the compilation time, and computing the WCET estimate in parallel to other operations would reduce all overheads to a negligible value.

An important remark about Winston (that also applies to the state-of-the-art algorithms Winston is compared with) if that it is an *anytime* algorithm: the longer it runs, the better the results. As shown in Figure 4 major improvements appear with a low number of iterations in the iterative compilation process. The number of iterations can then be tuned to match the complexity of the code under optimization, and make Winston and similar techniques applicable to real-world code and not only benchmarks.

The detailed results by benchmarks can be found in Figures 5 (for the Mälardalen/Heptane benchmarks) and 6 (for the Polybench benchmarks). Again in these figures, the x-axis is the number of non-cached computations of the WCET estimate. Except for a few benchmarks, Winston provides the lower WCET estimate. Moreover, Winston often significantly reduces it compared to the other algorithms.

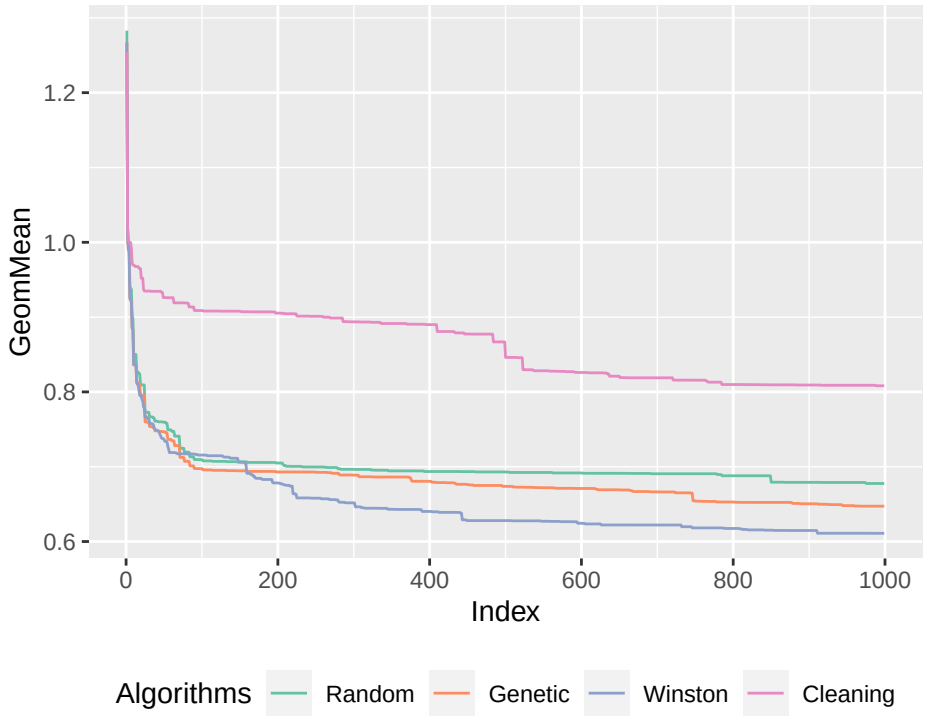
The likeliness for Winston to find a significant improvement in the WCET estimate varies with the benchmark considered. For instance, for *jacobi-1d* one is found approximately 60% of the time over the 1000 iterations, whereas on small benchmarks, like *qurt*, it is way rarer to find a significant reduction of the WCET estimate

[3]We manually removed some benchmarks when we found out after the entire experiment that the results were incorrect due to a mistake we made in the logging of the output of programs

Table 1: Benchmarks used in the experimental evaluation

Collection	Benchmarks
Mälardalen/Heptane	crc, des, duff, expint, fdct, jfdctint, ludcmp, matmult, ns, nsich-neu, qurt, statemate
Polybench	2mm, 3mm, adi, atax, bicg, covariance, durbin, fdtd-2d, floyd-warshall, gemm, gemver, gesummv, heat-3d, jacobi-1d, jacobi-2d, lu, mvt, symm, syr2k, syrk, trisolv, trmm

Figure 4: Result of Winston, against various algorithms of the state-of-the-art [7]. A geometrical mean over the benchmarks is used to represent the ratio between the estimated WCET, with the technique under consideration, over the estimated WCET with the best optimization level applicable (-O0..-O3).



total bis.pdf

(approximately 5% of the time). Overall the best improvements are found in the Polybench suite.

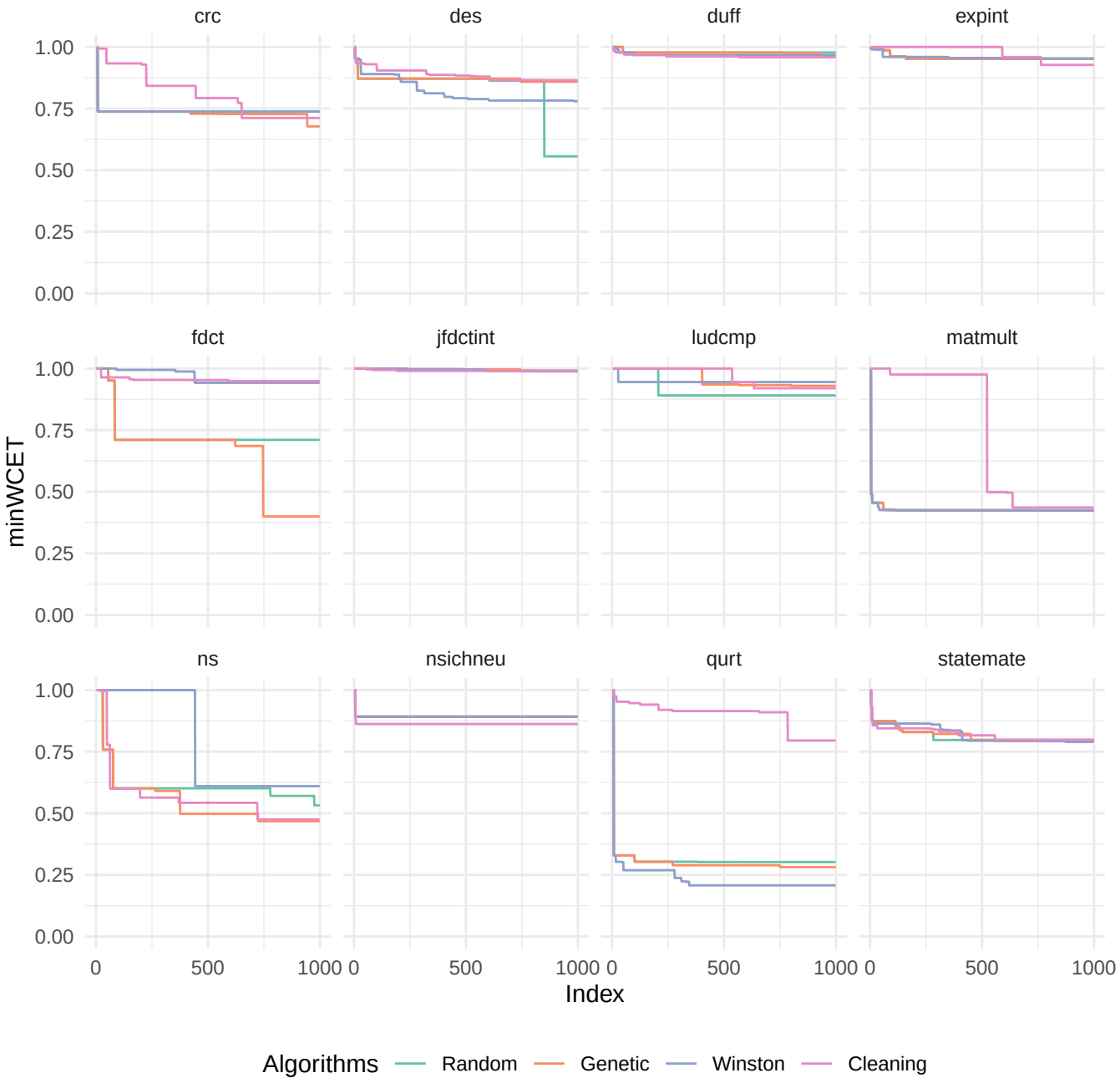
These significant improvements are not found for all benchmarks. For instance, with the *covariance* benchmark, Winston does not obtain a significant reduction of the WCET estimate, whereas other algorithms do.

As a final remark, all algorithms, Winston and all the other algorithms Winston is compared with, start with a set of randomly selected solutions. For this reason, all the algorithms are sensitive to the set of initial solutions, albeit in a different way. For the sake of reproducibility, the seed of the random number generator was set to the same value when evaluating all algorithms, which therefore start with the same set of initial solutions. It may happen that a good solution is found "by chance" because of the initial seed, minimizing the observed difference between algorithms. We additionally observed that changing the seed can (although marginally) change the solution found by each algorithm. A deeper comparison between the algorithms would require an exploration of the seed values. Such an exploration is hard to achieve due to the run-time of iterative compilation (hours for the more complex benchmarks).

5.3 Discussion

Some interesting problems arose with the use of large optimization sequences. First, the time it takes to compile benchmarks can be long, and on a few benchmarks it turns out to be much longer than the time taken by the WCET analysis tool. This could be improved by parallelizing the overall iterative compilation process (compiling a benchmark while the WCET analysis is run on another one). But overall, using large sequences can make the algorithm significantly

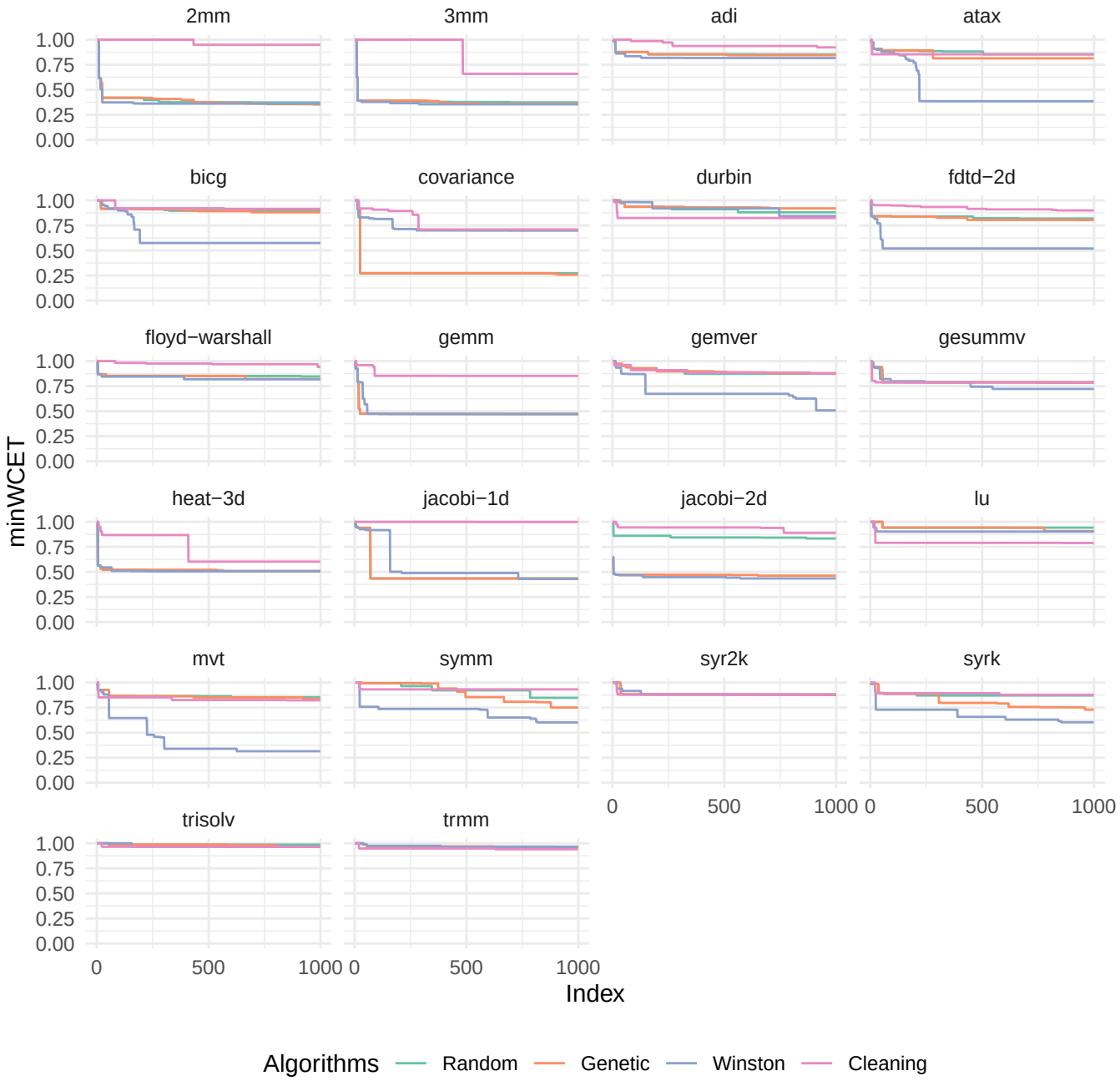
Figure 5: Detailed results per benchmark - Mälardalen/Heptane



longer to execute, especially for real-world programs where the compilation is already an expensive process. Because Winston can sometimes multiply by over 10 the compilation time (due to the long sequence of options used), it may be impractical to use Winston for real-world programs where the compilation already takes a few minutes/hours. In the process of designing Winston we have experimented the cleaning of optimization sequences after a number of iterations. We observed that too frequent cleaning reduces the benefits of using long sequences. Defining the best frequency of cleaning is left for future work.

Second, we observed that sometimes the compiled code is incorrect. This is due to bugs in *llvm/opt* when applying too many optimizations passes. While it rarely happens with short sequences

Figure 6: Detailed results per benchmark - Polybench



of options, it was sufficiently frequent in our experiments to motivate a correctness check of the compiled code. To do so, we compiled the intermediate code to the assembly code of the host machine (x86) and executed it on one input: the code prints the result so we can compare it with the result when no optimization is applied. The result of this process is sometimes strange. For instance, it can have a finite WCET estimate, but when executed on our machine, it has an infinite loop (or an abnormally long execution time). Also, as already observed in [7], sometimes *llvm/opt* crashes when it tries to apply optimization passes. All in all, it seems quite difficult to be sure of an optimization list that will not break the program. All sequences resulting in incorrect code are safely ignored in our

experiments. Generating correct code with complex and long optimization chains looks like an interesting area for future research, but is outside this paper's scope.

One interesting observation is that a significant reduction of WCET estimates happens mainly on the Polybench benchmark suite. It seems that benchmarks where significant reductions are found, maybe the ones where polyhedral optimizations find significant improvements, if so, the more complex the code, the better we can optimize the WCET. We also tried to lower the virtual unrolling factor, and significant reductions were no longer found, which may indicate that the simpler the program the less opportunity for these significant reductions to be found.

Unsurprisingly, the sequence of options that work best on a benchmark cannot in general be re-used elsewhere. What works best for a benchmark does not give a good result to others. The sequence of options resulting in the lower WCET estimate for each benchmark is also very different, we could not find obvious similarities between them, even when looking at cleaned sequences that contain only 20 to 30 options.

Another situation where this algorithm finds significant reduction is the *qurt* example from the Mälardalen benchmark. In this example, one function calls another one three times, and it seems the significant reduction of the WCET comes from optimizations that delete the first two calls, as their results are never used. This shows that our method discovers compilation sequences that are able to remove aggressively dead code, whereas traditional compiler options specialized in dead code elimination are unable to remove it.

On one instance of the Polybench benchmarks, the way the result was logged was wrong: only the first element of the result array was logged instead of the whole array. Our algorithm finds in this example a sequence of options that safely delete all the code that was not dealing with this first element. Although, in this example, a large amount of dead code was introduced by mistake, this situation shows that our algorithm is able to perform aggressive dead code elimination, which is a significant improvement over a single *-O3* optimization flag that rarely eliminates a large amount of dead code.

6 RELATED WORK

Iterative compilation techniques aim at minimizing an objective (most of the time the average-case performance) by searching the huge space of compilation options as efficiently as possible. To do so, meta-heuristic algorithms are usually used, which are high-level heuristics that work well for a wide range of optimization problems [8]. Therefore, the literature about meta-heuristic algorithms is very developed, it goes from the genetic algorithm [6, 24] that we used in this work to more exotic algorithms [4, 18], which inspired us when we created Winston. Some of these meta-heuristic algorithms have been used for iterative compilation, for example, the genetic algorithm in [7], or the algorithm derived from the simplex method in [18]. Based on our observation that long optimization sequences significantly reduce WCET estimates, we have designed Winston, which uses the meta-heuristics well-suited to derive long optimization chains.

The iterative compilation has been an actively researched topic for decades [2, 5, 10, 14], however, to the best of our knowledge, none of these works tried to use a tremendous amount of optimization options. Our work differs from these in the sense that algorithms described in this paper find optimization sequences that are impossible to find by hand.

Many techniques have been developed to determine safe and as precise as possible WCET estimates [26]. While research on WCET estimation led to a large amount of research, optimizing code for WCET minimization comparatively received less attention. Existing techniques for WCET-oriented optimization may optimize the usage of the memory hierarchy [16, 25] branch predictors [3], or perform WCET-aware loop transformations [17]. In contrast to these approaches, we take benefit from the full set of compiler optimizations from mainstream compilers, designed for average-case performance, to automatically select the ones that minimize WCET estimates. The research about iterative compilation and WCET is reduced to [7, 23]. Our main finding as compared to [7, 23] is that long optimization sequences allow larger reductions of WCET estimates.

Finally, our method, in opposite to [23], compiles the whole source code with the same sequence of options instead of using a different set of options per function. While this reduces the search space, it may give a better result to use a fine-grain approach, as options that work best for certain benchmarks do not work for others, and therefore using a different set of options for a different part of the code seems appropriate.

7 CONCLUSION

In this work, we have discovered the counter-intuitive observation that using long sequences of optimization passes helps lower WCET estimates. Based on that observation, we designed an algorithm named Winston, suited to the generation of long sequences. This algorithm performs significantly better than [7], allowing a reduction of WCET estimates of 35% on average, as compared to the 20% of [7] using the same benchmarks and experimental conditions.

In future work, we will refine Winston to behave more consistently on the benchmarks, as there are still some benchmarks on which more naive methods perform better than Winston. We will also consider cleaning optimization sequences to reduce compilation time while not altering the positive impact of long sequences. Another direction for future work would be to optimize different portions of the code from a given application using different optimization flags to better adapt the optimization flags to that portion, with the pitfall of further enlarging the search space. Finally, an open question we would like to answer is whether large sequences are useful when the average-case is considered instead of the worst-case. Answering this question may lead to interesting discoveries and new algorithms for iterative compilation. .

Acknowledgments

The authors would like to thank Abderaouf (Nassim) Amalou, Hugo Reymond and the anonymous reviewers for their fruitful comments on earlier drafts of this paper.

REFERENCES

[1] Gaisler bare-c cross-compiler system.

[2] François Bodin, Toru Kisuki, Peter Knijnenburg, Mike O'Boyle, and Erven Rohou. Iterative compilation in a non-linear optimisation space. In *Workshop on Profile and Feedback-Directed Compilation*, 1998.

[3] François Bodin and Isabelle Puaut. A wcet-oriented static branch prediction scheme for real time systems. In *17th Euromicro Conference on Real-Time Systems (ECRTS 2005), 6-8 July 2005, Palma de Mallorca, Spain, Proceedings*, pages 33–40, 2005.

[4] Edmund K Burke and Yuri Bykov. The late acceptance hill-climbing heuristic. *European Journal of Operational Research*, 258(1):70–78, 2017.

[5] Yang Chen, Shuangde Fang, Yuanjie Huang, Lieven Eeckhout, Grigori Fursin, Olivier Temam, and Chengyong Wu. Deconstructing iterative optimization. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(3):1–30, 2012.

[6] GV Conroy. Handbook of genetic algorithms by lawrence davis (ed.), chapman & hall, london, 1991, pp 385,£ 32.50. *The Knowledge Engineering Review*, 6(4):363–365, 1991.

[7] Mickaël Dardaillon, Stefanos Skalistis, Isabelle Puaut, and Steven Derrien. Reconciling compiler optimizations and wcet estimation using iterative compilation. In *2019 IEEE Real-Time Systems Symposium (RTSS)*, pages 133–145. IEEE, 2019.

[8] Tansel Dokeroglu, Ender Sevinc, Tayfun Kucukyilmaz, and Ahmet Cosar. A survey on new generation metaheuristic algorithms. *Computers & Industrial Engineering*, 137:106040, 2019.

[9] Heiko Falk and Paul Lokuciejewski. A compiler framework for the reduction of worst-case execution times. *Real Time Syst.*, 46(2):251–300, 2010.

[10] GG Fursin, Michael FP O'Boyle, and Peter MW Knijnenburg. Evaluating iterative compilation. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 362–376. Springer, 2002.

[11] Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. The Mälardalen WCET Benchmarks: Past, Present And Future. In Björn Lisper, editor, *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*, volume 15 of *OpenAccess Series in Informatics (OASIcs)*, pages 136–146, Dagstuhl, Germany, 2010. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. The printed version of the WCET'10 proceedings are published by OCG (www.ocg.at) - ISBN 978-3-85403-268-7.

[12] Matthew R Guthaus, Jeffrey S Ringenberg, Dan Ernst, Todd M Austin, Trevor Mudge, and Richard B Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Proceedings of the fourth annual IEEE international workshop on workload characterization. WWC-4 (Cat. No. 01EX538)*, pages 3–14. IEEE, 2001.

[13] Damien Hardy, Benjamin Rouxel, and Isabelle Puaut. The Heptane Static Worst-Case Execution Time Estimation Tool. In Jan Reineke, editor, *17th International Workshop on Worst-Case Execution Time Analysis (WCET 2017)*, volume 57 of *OpenAccess Series in Informatics (OASIcs)*, pages 8:1–8:12, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[14] Toru Kisuki, P Knijnenburg, M O'Boyle, and H Wijshoff. Iterative compilation in program optimization. In *Proc. CPC'10 (Compilers for Parallel Computers)*, pages 35–44. Citeseer, 2000.

[15] Hanbing Li, Isabelle Puaut, and Erven Rohou. Traceability of flow information: Reconciling compiler optimizations and WCET estimation. In Mathieu Jan, Belgacem Ben Hedia, Joël Goossens, and Claire Maiza, editors, *22nd International Conference on Real-Time Networks and Systems, RTNS '14, Versaille, France, October 8-10, 2014*, page 97. ACM, 2014.

[16] Paul Lokuciejewski, Heiko Falk, and Peter Marwedel. Wcet-driven cache-based procedure positioning optimizations. In *20th Euromicro Conference on Real-Time Systems, ECRTS 2008, 2-4 July 2008, Prague, Czech Republic, Proceedings*, pages 321–330, 2008.

[17] Paul Lokuciejewski and Peter Marwedel. Combining worst-case timing models, loop unrolling, and static loop analysis for wcet minimization. In *2009 21st Euromicro Conference on Real-Time Systems*, pages 35–44. IEEE, 2009.

[18] Pingjing Lu, Yonggang Che, and Zhenghua Wang. Combining model and iterative compilation for program performance optimization. *J. Softw.*, 4(3):240–247, 2009.

[19] Ravi Teja Mullapudi, Vinay Vasista, and Uday Bondhugula. Polymage: Automatic optimization for image processing pipelines. *ACM SIGARCH Computer Architecture News*, 43(1):429–443, 2015.

[20] John A. Nelder and R. Mead. A simplex method for function minimization. *Comput. J.*, 7(4):308–313, 1965.

[21] Online. The Polybench benchmark suite. https://web.cse.ohio-state.edu/~pouchet.2/software/polybench.

[22] Online. "aiT:the industry standard for static timing analysis.". http://www.absint.com/ait.

[23] Isabelle Puaut, Mickaël Dardaillon, Christoph Cullmann, Gernot Gebhard, and Steven Derrien. Fine-grain iterative compilation for wcet estimation. In *WCET 2018-18th International Workshop on Worst-Case Execution Time Analysis*, pages 1–12, 2018.

[24] L Shi and K Rasheed. A survey of fitness approximation methods applied in evolutionary algorithms. In *Computational intelligence in expensive optimization problems*, pages 3–28. Springer, 2010.

[25] Vivy Suhendra, Tulika Mitra, Abhik Roychoudhury, and Ting Chen. WCET centric data allocation to scratchpad memory. In *Proceedings of the 26th IEEE Real-Time Systems Symposium (RTSS 2005), 6-8 December 2005, Miami, FL, USA*, pages 223–232, 2005.

[26] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, et al. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3):1–53, 2008.