Using Polyhedral Techniques to Tighten WCET Estimates of Optimized Code: A Case Study with Array Contraction

Thomas Lefeuvre, Emin Kasnakli, Imen Fassi, Isabelle Puaut, Christoph Cullmann, Steven Derrien, Gernot Gebhard

Using Polyhedral Techniques to Tighten WCET Estimates of Optimized Code: A Case Study with Array Contraction

Thomas Lefeuvre
Univ. Rennes 1/IRISA

Imen Fassi
Univ. Rennes 1/IRISA

Christoph Cullmann
AbsInt GmbH

Gernot Gebhard
AbsInt GmbH

Emin Koray Kasnakli
Fraunhofer IIS

Isabelle Puaut
Univ. Rennes 1/IRISA

Steven Derrien
Univ. Rennes 1/IRISA

***Abstract*—The ARGO H2020 European project aims at developing a Worst-Case Execution Time (WCET)-aware parallelizing compilation toolchain. This toolchain operates on Scilab and XCoS inputs, and targets ScratchPad memory (SPM)-based multi-cores. Data-layout and loop transformations play a key role in this flow as they improve SPM efficiency and reduce the number of accesses to shared main memory. In this paper[1], we study how these transformations impact WCET estimates of sequential codes. We demonstrate that they can bring significant improvements of WCET estimates (up to 2.7×) provided that the WCET analysis process is guided with automatically generated flow annotations obtained using polyhedral counting techniques.**

I. Introduction

Real-time systems are ubiquitous, and many of them play an important role in our daily life. In hard real-time systems, computing the correct results is not the only requirement. In addition, the results must be produced within predetermined 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 deadlines. Standard static WCET estimation techniques [1] compute such bounds from a 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 over-provisioning the system.

WCET analysis is confronted with two challenges: (i) extracting knowledge of the execution flow of an application from its machine code, and (ii) modeling the temporal behavior of the target platform. The former issue can benefit from the designer guidance through *flow facts* (loop bounds, infeasible paths) expressed using annotations. Multi-core platforms make the latter issue even more challenging, as interference caused by concurrent accesses to shared resources have also to be modeled. In this context, the goal of the ARGO H2020 project (http://www.argo-project.eu/) is to develop a WCET-aware parallelizing compilation toolchain, operating from Scilab and XCoS inputs specifications, and targeting scratchpad memory (SPM) based embedded multi-cores.

Obviously, accurate WCET analysis is facilitated by *predictable* hardware architectures, for which the hardware state during execution can be modeled without introducing too much pessimism in the timing analysis. For example, hardware platforms using ScratchPad Memories (SPMs) instead of caches are considered as more predictable. The reason is that accurately modeling the temporal behavior of data and instruction caches is very challenging, and leads to very pessimistic WCET estimates, for some cache designs (replacement policies, cache hierarchies, write policies, cache coherency).

SPMs are not handled by the hardware, such as caches, but are managed by the software. When SPM management is left to the programmer-managed, it makes them very difficult to use. Many researches have thus studied how to implement automatic SPM management at the compiler level, either as a standalone optimization [2]–[4] or in combination with loop and array-layout transformations [5], [6]. For the latter, it has been shown that impressive average-case performance improvements could be obtained on compute intensive kernels, but their ability to reduce WCET estimates remains to be demonstrated. This is the topic we address in this work. More precisely, our contributions are the following:

- We demonstrate the ability of automatic polyhedral optimization techniques to reduce WCET estimates in the case of sequential codes, with a focus on locality improvement and array contraction. We show on representative real-time image processing use cases that they can bring significant improvements of WCET estimates (up to 40%) provided that the WCET analysis process is guided with automatically generated flow annotations.
- We present an automatic source-to-source compiler toolchain (part of the ARGO flow) which circumvents some limitations of modern static WCET estimation tools in presence of aggressive code transformations. The approach helps WCET estimation tools with automatically-generated flow annotations.

[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 remainder of this paper is organized as follows. Section II provides background information on the ARGO project and recalls how polyhedral compilation techniques can be used to take advantage of SPMs to improve performance. Section III describes our approach based on automatic derivation of flow annotations, and Section IV provides preliminary results on two representative use cases (real-time image processing). Conclusion and future work are sketched in Section V.

II. Background

In this Section, we first provide an overview of the ARGO project. Then, we describe the loop and array-layout transformations considered in this work.

A. The ARGO project

The goal of the ARGO project is to design a WCET-oriented parallelizing tool-chain for embedded multi-core architectures. The ARGO flow [7], operates on user applications expressed either as XCoS models (Simulink-like) or in the Scilab language. This input specification is then parallelized and optimized for WCET on an embedded multi-core architecture.

A fundamental requirement for obtaining WCET estimates using WCET analyzers, such as AbsInt's aiT [8], is that the timing behavior of all processor cores is *predictable*. For example, predictability at the core level is compromised by speculative hardware mechanisms such as caches or branch prediction. On multi-core platforms, the additional problem of shared hardware resources for which the interference due to concurrent accesses have to be considered, arises. To keep the problem tractable, we only consider in ARGO hardware platforms with the following characteristics for each core:

- A multi-core platform composed of time-predictable processors (here Leon3 or Xentium processors), supported by tools such as AbsInt's aiT [8]
- Memory hierarchies based on private SPMs instead of data and instruction caches, in order to avoid challenging cache hit/miss analysis

In this paper, we will restrict ourselves to one core of the InvasIC platform, a predictable configurable and tile-based multi-core platform based on Leon3 cores.

One of the challenges for ARGO is to be able to take advantage of the use SPMs as a replacement for data and instruction caches. Although SPM-based platforms are much easier to model from a WCET point of view, they raise many challenges from the compiler point of view, as scratchpad must be explicitly managed by the software. Because of their widespread use in embedded systems, there has been a lot of research on compiler support for dynamic scratchpad management [2]–[4]. Most of this work aim at determining (i) which variable should be allocated to scratchpad (ii) when a given variable should be moved in scratchpad and copied back to shared memory. The problem is that in many cases (e.g real-time image processing), the arrays manipulated by the program are way too large to fit in a scratchpad memory.

It is however possible to circumvent this issue by resorting to array layout transformations. Such transformations, explored in this paper, can be used to reduce the array sizes (array contraction) and/or partition arrays into regions so that they can fit the scratchpad.

B. Polyhedral based SPM management

Polyhedral based loop and data-layout transformation techniques are very efficient when it comes to optimizing memory accesses for SPM-based platforms [5]. For example, they offer very efficient array contraction algorithms [6].

Such algorithms consists in reducing the footprint of all temporary/intermediate arrays. This is achieved through complex re-indexing transformations involving circular (i.e. modulo based) addressing functions. The goal is generally to find the re-indexing transformation leading to the smallest possible array. The technique is based on array cell live range analysis, which is made possible thanks to the polyhedral representation. The result of this analysis is then used to identify conflicting array cells locations (cells that are alive at the same time). From there, finding a legal contraction consists in obtaining a re-indexing function that does not remap two conflicting cells to the same memory location.

When used alone, array contraction brings few benefits, but, its efficiency can be significantly improved when combined with locality improving loop transformations (tiling and/or fusion). Thanks to steady improvements in the polyhedral compilation techniques, nowadays, we have algorithms that automatically find such program transformations [9].

Since a detailed discussion on array contraction is out of the scope of this work, we simply illustrate the effect of the transformation on an example, given in Figure 1. As visible in the figure, and further detailed in Section III, the optimized code now contains complex control flow, which may challenge WCET analysis tools. To address this issue, we enrich transformed code with annotations, that we describe in the following Section.

III. Tightening WCET estimates using automatically generated flow annotations

This Section is organized as follows. We first discuss the scalablity issue with static WCET estimation tools for complex loop nests, and then describe our approach based on source level flow annotations.

A. Scalability of static WCET analysis

As mentioned in the previous Section, polyhedral transformations often lead to complex loop structures (with guarded statements and non constant loop bounds for example). For average-case performance, this overhead is largely outweighed by others benefits (such as parallelism and locality). This is not true when dealing with worst-case performance, as this complexity can result in very pessimistic WCET estimates, or no WCET estimate at all when the estimation tool cannot derive loop bounds.

This complexity is illustrated in Figure 1, which shows a code excerpt taken from the optimized PIPS use case (see Section IV for more details). The interested reader may notice that, in this code snippet, loop `L3` is guarded by a predicate expression based of the outer loop index $c1$. Similarly, statements S1 and S2 are guarded by complex combinations of guards. Because it cannot always determine precisely how often a guard will be valid, a static WCET tool may end-up overestimating the WCET by considering the worst-case path

```
L1: for(c0 = 0; c0 <= 14; c0 = c0 + 1)
      if(c0 >= 11)
L2:     for(c1 = 0; c1 <= 959; c1 = c1 + 1) {
          ...
          if (c1 < 640)
L3:         for(c2 = 0; c2 <= 479; c2 = c2 + 1) {
S1:           if (c0 == 11)
                d[c1][c2] = ... ;
              if ((c1 >= 2) && (c2 >= 2)) {
                s[(c1 - 1) % 512][(c2 - 1)] = ...;
                if ((c1 >= 320) && (c2 >= 240)) {
S2:                ix[1][...][...] = ...;
                }
              }
            }
        ...;
```

Fig. 1. Excerpt of the PIPS kernel after polyhedral transformation.

for every loop iteration. This will then result in a pessimistic WCET estimate.

As mentioned earlier, the ARGO project relies on the AbsInt aiT tool to perform binary level WCET analysis. As for all static WCET estimation methods [10], the tool must be able to determine an upper bound of the number of iterations for every loop to be able to estimate the WCET. Because inferring loop bounds from binary is challenging, aiT offers two ways to derive as precise as possible loop bounds.

- The first approach relies on *annotations*, which are used to specify loop bounds that could not otherwise be inferred by the tool. Such annotations have to be provided by the user/programmer, and because of this are error-prone and may lead to unsafe WCETs (due to incorrect annotations in the case of too complex source code for example).
- The second one is based on *virtual unrolling*, in which several instances of a same loop/functions are considered during the analysis (each instance corresponding to a specific context). This approach is very efficient for nested loops operating on small/medium data-sets. In most cases, the use of aggressive virtual unrolling leads to very accurate WCET estimates, as the context associated to each loop instance can be used to eliminate dead guards in the execution path. As a matter of fact, in our case, the choice of a *large enough* unrolling factor would always lead to a tight WCET estimate, as we end-up performing a complete symbolic execution of the loop nest.

Unfortunately, none of these two approaches can be used in the context of the ARGO project.

Since ARGO operates from very high-level models (Scilab, XCos), an approach based on user-provided loop bound annotations is not practical, because the objective is to hide details of code transformations to the end-user.

In addition, we also experimentally observed that virtual unrolling does not scale well for very large loop trip counts and large unrolling factors. In some cases, (see Section IV for details) the analysis had to be stopped after two hours, or failed due to memory exhaustion, which is not compatible with the user-in-the-loop approach as advocated by the ARGO toolset.

To address these shortcomings, we propose a meet-in-the-middle approach, which combines polyhedral-based static analysis (implemented at the C level in the Gecos source-to-source compiler [11]), with flow annotations (used at the binary/assembly level in AbsInt aiT WCET analysis tool).

B. Polyhedral based flow analysis

Figure 2 summarizes the proposed loop and array transformation framework. Source-to-source transformations are first applied using a polyhedral representation of loop nests. The polyhedral transformations achieved in this Figure are loop fusion combined with loop shifting; the two for-loop nests are fused and the inner-most loop (j-loop) is shifted by a factor equal to 2 in order to respect the dependencies. Flow annotations are then generated and inserted in the source code using aiT flow annotation facilities.

Flow annotations are associated to basic blocks and can be used, among others, to specify the exact execution count of a given block along the worst-case execution path. As currently implemented in aiT, the annotation follows the syntax described below, where ***labelX*** is the address of a machine instruction (identified through a label) and ***cnt*** is the computed execution count. Similar annotations are provided to specify maximal execution counts of basic blocks.

$$\textit{ais2 } \{\textit{flow sum: point("}\textbf{\textit{labelX}}\textit{") == }\textbf{\textit{cnt}}\textit{; }\}.$$

In a polyhedral representation of a program, a *domain* is associated to every statement in the loop nest. This *domain* is defined using a set of affine constraints over loop indices and parameters. The dimension of the domain (i.e. the polyhedron) corresponds to the depth of the loop-nest it represents. In our use cases, we did not observe any scalability problem. Since the code regenerated after optimization also admits a polyhedral representation, we can reconstruct the domains of all the statements in the regenerated code, as we do from the original program. Because the program transformation is not bijective statement wise (a statement in the original program may be transformed into more than one statement because of loop peeling or loop unrolling for example) we cannot operate on the initial polyhedral representation, but need to reconstruct the domains of the transformed statements.

Deriving the flow information associated to a statement then amounts to counting the number of iterations in its corresponding domain. Counting the number of integral points in a polytope is a well studied problem [12], [13], and most state-of-the-art polyhedral compilation toolboxes offer this ability. In the general case, the outcome of the counting process takes the form of a piecewise polynomial expression. In our case, since we currently only deal with constant sized domains, the result is always a constant.

For example, consider the basic-block containing the statement `S1` in the transformed code example from Figure 2. After extracting the polyhedral representation of the transformed program, we obtain its domain $\mathcal{D}_{S1}$ defined by the following equalities:

$$\mathcal{D}_{S1} = \{i, j | i \geq j \wedge 0 \leq i < 640 \wedge 0 \leq j < 480\}$$

The flow annotation that we are interested in corresponds to the number of integral points in its associated domain (that we write $card(\mathcal{D}_{S1})$), in this case $card(\mathcal{D}_{S1}) = 153280$.

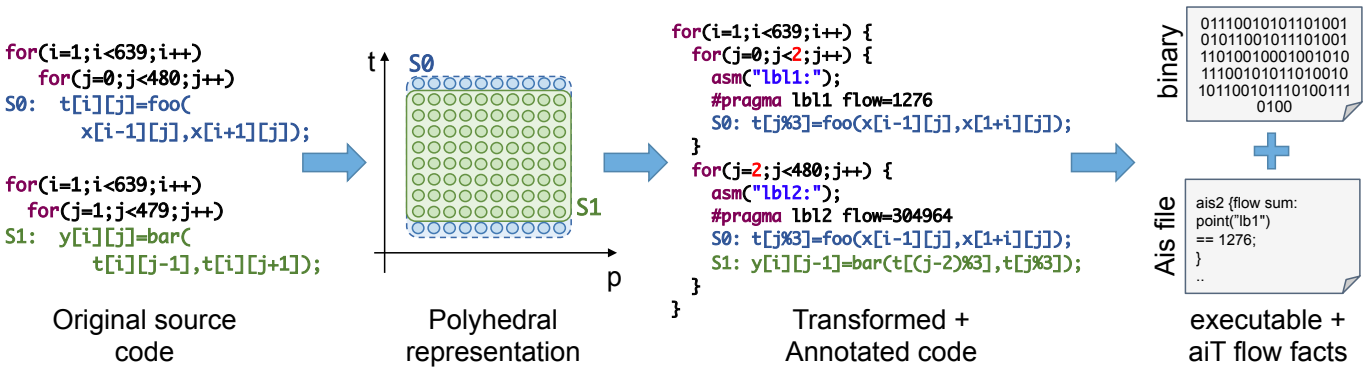

Fig. 2. Summary of the proposed WCET aware loop and array transformation framework.

The derived flow information remains attached to a source level statement (and its corresponding basic-block in the Gecos compiler intermediate representation), not to a machine instruction label as required by aiT. To bridge the gap between the two tools, we insert custom labels using inlined `asm` statements, as depicted in Figure 2, and make sure the resulting program is compiled without optimization which could make the annotation invalid, in case the C compiler applies loop transformations.

IV. Experiments

As mentioned in the introduction, the goal of this work is to tighten WCET estimates through the combined use of SPMs, locality enhancing loop transformations and data layout optimizations. Our goal is to demonstrate that, a meet-in-middle approach involving both source-level and binary-level analysis is mandatory to observe WCET performance improvements for *complex* compute-intensive loop kernels.

A. Benchmarks and experimental setup

In our experiments, we considered two real-time image processing applications:

- The first one is a classical image processing pipeline which implements the classical Harris Corner detection algorithm. The implementation operates on high definition images, and consists of 4 stages. The implementation uses 10 distinct arrays of 4 MB each.
- The second application is used for post-processing raw data from a polarized image sensor, used in factory automation for product defect detection. This application, named PIPS hereafter, is an industrial use case from the ARGO project. Given its size and relative complexity, it was deemed relevant as a real world example code for our experiments. The application consists of several processing stages, each of them operating on its own intermediate version of the image. The implementation uses 15 distinct statically allocated arrays for storing intermediate results, with sizes ranging from 2.46 MB to 9.83 MB.

In this work, we use the Leon3 core as target processor, as it is one of the building blocks for the InvasIC multi-core architectures designed in the ARGO project. The Leon3 core is an open-source implementation of the Sparc-V8 instruction set by Gaisler research (http://www.gaisler.com/). In our experiments, the processor does not include cache memory, but instead uses a 64 KB on-chip SPM. The compiler used for the experiments is *gcc 4.9.4* for Sparc. In the following, we always consider the code to be executed from the SPM. We configured aiT for the InvasIC core, by specifying two address ranges: one for the SPM, and one for the external memory, with distinct access times. We considered 0 cycle delay penalty for SPM access (i.e a load/store instruction takes 1 cycle to execute) and used several distinct values (16, 32 and 64 cycles) for the access delay to shared main memory. The informed reader may observe that these values are higher than typical DRAM access latencies. However, although this paper concentrates on sequential codes, we are interested in the scope of ARGO in multi-cores, where tens of cores may have interfering accesses to this memory through its controller, explaining our choice. The same configuration for aiT is used in all experiments reported in this Section, except if the opposite is explicitly stated in the text.

B. Impact of compiler optimizations on WCET estimates

As a first experiment, we use aiT to estimate WCETs for the original version of our target algorithms under different levels of compiler optimizations (from -O0, no optimization, to -O3). The goal of this experiment is to observe the impact of optimizations on the tool ability to derive WCET estimates. We performed this experiment with various levels of *virtual enrolling* (from 2, default value, to 256 in order to test for scalability) to help the tool determining flow information. The results of this experiment are provided in Table I for both Harris and PIPS. Shared memory access time for this experiment is set to 32, and without array contraction, manipulated arrays are too large to fit in SPM, that is then used to store the code only.

Our results show that the ability of aiT to derive WCET estimates is highly dependent on the program control flow complexity. For example, aiT is able to derive bounds for all versions of the original Harris kernel. On the PIPS use case, the tool was unable to derive a WCET estimate for PIPS when optimizations are enabled, even with the help of *virtual unrolling*. As depicted in Table I for the PIPS example, aiT either terminates, but fails to estimate the WCET because it

kernel	opt	Virtual unrolling factor				
		2	4	16	64	256
Harris	-O0	2.512	2.507	2.507	2.507	$\perp$
	-O1	1.073	1.072	1.072	1.072	$\perp$
	-O2	1.074	1.073	1.073	1.073	$\perp$
	-O3	1.173	1.173	1.173	1.173	$\perp$
PIPS	-O0	20.54	20.54	20.53	20.52	$\perp$
	-O1	∞	∞	∞	∞	$\perp$
	-O2	∞	∞	∞	∞	$\perp$
	-O3	∞	∞	∞	∞	$\perp$

TABLE I
WCET ESTIMATES FOR ORIGINAL CODE (IN GCYCLES).

kernel	opt	Virtual unrolling factor				
		2	4	16	64	256
Harris	-O0	2.086	2.078	2.078	2.078	$\perp$
	-O1	∞	∞	∞	∞	$\perp$
	-O2	0.563	0.561	0.561	0.561	$\perp$
	-O3	0.566	0.564	0.564	0.564	$\perp$
PIPS	-O0	1548.48	1545.53	26.62	23.33	$\perp$
	-O1	∞	∞	∞	∞	$\perp$
	-O2	∞	∞	∞	∞	$\perp$
	-O3	∞	∞	∞	∞	$\perp$

TABLE II
WCET ESTIMATES FOR TRANSFORMED CODE (GCYCLES) - NO FLOW ANNOTATIONS.

could not infer some loop bounds (marked as ∞) or causes memory exhaustion (marked as $\perp$) after two hours of runtime.

C. Impact of polyhedral transformations

As a second experiment, we have used aiT to estimate the WCET on the transformed version of our examples, after loop fusion and array contraction is applied. To avoid costly *modulos* operation, we increased the size of all circular arrays so as to reach a power of two (*modulos* are then translated into simple bitmasking operations). SPM access access delay is set 0 cycle, whereas shared memory access time is set to 32 cycles.

The results of the transformation for our two use cases are summarized below:

- For the Harris use case, the size of all 10 intermediate arrays was reduced thanks to the combination of fusion and contraction (4 arrays reduced to scalars, 5 arrays sizes reduced by more than 98%). All intermediate arrays could then fit in the SPM.
- For the PIPS use case, 6 arrays out of 15 benefited from our transformation (4 were reduced to scalars, one was reduced from 2.46 MB to 11.52 KB, and one from 7.3 MB to 24 Bytes. Here again, all intermediate arrays could fit in the SPM.

We first exercised the aiT ability to estimate a WCET using its *virtual unrolling* feature enabled, and this *without* flow annotations. The results of this experiment are provided in table II for Harris and PIPS, for an external memory access latency of 32 cycles. In the following Tables, UF stands for aiT Unrolling Factor.

They show that for Harris, aiT is still able to derive a WCET estimate without annotations, for all optimization levels except -O1 (at the time of writing, we have no explanation for this observation). For PIPS, the tool can only find a WCET estimate at -O0 level. For higher optimization levels, the analysis fails because of some missing loop bounds or because of memory exhaustion. Virtual unrolling significantly tightens WCET estimates (by a factor 66 for PIPS). However, this tighter WCET estimate comes at the cost of increased analysis time (for PIPS compiled at -O0, the analysis time increased from 2 s for UF=2 to 23 mn for UF=64, still running after 2 hours for UF=256).

As explained earlier, we propose to guide the WCET analyzer by computing flow annotations at the source code level. A key issue is that flow annotations are only valid if the compiler does not alter the control flow exposed in the original source code. Although we observed that the use of assembly labels in the source prevents many *gcc* optimizations, and somewhat preserves the validity of the flow annotations, we currently have no systematic way of ensuring they remain valid other than compiling the code with the -O0 flag. Resorting to -O0 is probably too conservative, as there exist many optimizations that are *flow-safe* in the sense that they would not impact the validity of the source level flow annotations at assembly level [14]. However, identifying such flow-safe optimizations remains an open and challenging problem, especially given the complexity of current compiler frameworks (for example *gcc* has more than 150 user-exposed optimization flags).

kernel	opt	Virtual unrolling factor				
		2	4	16	64	256
Harris	-O0	2.06	2.06	2.06	2.06	$\perp$
PIPS	-O0	12.78	12.75	12.58	12.58	$\perp$

TABLE III
ANNOTATED WCET FOR TRANSFORMED PIPS AND HARRIS (GCYCLES) -O0, FLOW ANNOTATIONS.

The results of the WCET analysis using flow annotations and compiled with optimization level -O0 is provided in Table III for both Harris and PIPS. They show that we are able to obtain WCET estimates in both cases, although the results for Harris are not as good as those obtained by aiT alone.

D. Overall benefits and discussion

The question raised by this work is the following: are polyhedral compiler optimizations useful for reducing WCET estimates?

To provide a fair answer to this question, we need to compare the *best WCET estimate* for the original program with the *best WCET estimate* for the transformed program. In our context, the *best WCET estimate* corresponds to the lowest safe[2] WCET estimate obtained using the various compiler optimization levels available (-O0, -O1, -O2 and -O3). In

[2]This excludes WCET estimates at -O3 needing additional flow annotations, because compiler optimizations may make the annotations invalid.

	Latency	Original	Transformed	Ratio
Harris	16	0.818	0.558	1.47
	32	1.072	0.563	1.91
	64	1.582	0.573	2.7
PIPS	16	14.25	10.49	1.36
	32	20.52	12.58	1.64
	64	32.98	16.7	1.97

TABLE IV
BEST WCET ESTIMATES FOR ORIGINAL AND TRANSFORMED EXAMPLES FOR VARIOUS MEMORY LATENCIES (RATIO = ORIGINAL/TRANSFORMED).

the case of Harris, we compare the WCET estimate obtained for the original code at -O1 (surprisingly slightly lower than at -O2 and -O3) with the WCET estimate obtained for the transformed code at -O2 (slightly lower than at -O3). In the case of PIPS, we are only able to estimate the WCET at optimization level -O0 for both the original and transformed versions, and therefore compare these two results.

The WCET improvements observed for these two application are provided in Table IV and show improvement ratios between 1.2 and 2.7. For each example, we show the performance improvements for a given access delay to the shared external memory. As expected, for larger memory access times, the improvement offered by array contraction increases.

Of course, there are certainly examples where the locality enhancing polyhedral optimizations would end-up to be counter productive from a WCET point of view. A typical scenario would be a program for which aiT could automatically derive a WCET at -O3 before the polyhedral transformation, but would fail after transformation for all optimization levels but -O0 (thanks to flow annotations). However, we did not witness such a case so far.

Besides, in the context of the ARGO flow, the use of polyhedral loop transformations is not restricted to improve the efficiency of SPMs, it is also used to expose task level parallelism that can be later used by the scheduler/mapper to map and schedule tasks on cores. We therefore expect these transformations to also contribute in improving system-level WCET performance (i.e. the WCET of the parallel application).

V. CONCLUSION

The ARGO H2020 European project aims at designing a WCET-aware parallelizing compilation flow for predictable embedded multi-cores. Among the optimizations involved in such a flow, data-layout optimizations and parallelizing loop transformations obviously play a key role. In this paper, we study how such transformations effectively impact WCET analysis. We demonstrate on two use cases that polyhedral transformation can bring significant improvements of WCET estimates (up to 2.7 $\times$ provided they can also drive the WCET analysis process through flow annotations, that can be computed automatically using well known polyhedral counting techniques.

Another outcome of this work is the observation that understanding the impact of combinations of complex compiler optimizations on WCET is a very challenging problem. This complexity is also acknowledged in the context of average-case performance, and has motivated research work on iterative compilation, which consists in exploring many combination of optimizations to pick-up the combination offering the best performance. We believe that a similar approach for WCET would be very relevant, especially in the context of complex toolchains such as ARGO, where both the compiler and WCET analyzers are used as grey boxes.

REFERENCES

[1] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström, "The worst-case execution-time problem – overview of methods and survey of tools," *ACM Trans. Embed. Comput. Syst.*, vol. 7, pp. 36:1–36:53, May 2008.

[2] J. Deverge and I. Puaut, "Wcet-directed dynamic scratchpad memory allocation of data," in *19th Euromicro Conference on Real-Time Systems, ECRTS'07, 4-6 July 2007, Pisa, Italy, Proceedings*, pp. 179–190, 2007.

[3] M. R. Soliman and R. Pellizzoni, "WCET-Driven Dynamic Data Scratchpad Management With Compiler-Directed Prefetching," in *29th Euromicro Conference on Real-Time Systems (ECRTS 2017)* (M. Bertogna, ed.), vol. 76 of *Leibniz International Proceedings in Informatics (LIPIcs)*, (Dagstuhl, Germany), pp. 24:1–24:23, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017.

[4] O. Avissar, R. Barua, and D. Stewart, "An optimal memory allocation scheme for scratch-pad-based embedded systems," *ACM Trans. Embed. Comput. Syst.*, vol. 1, pp. 6–26, Nov. 2002.

[5] M. M. Baskaran, U. Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan, "Automatic data movement and computation mapping for multi-level parallel architectures with explicitly managed memories," in *Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, (New York, NY, USA), pp. 1–10, 2008.

[6] A. Darte, R. Schreiber, and G. Villard, "Lattice-based memory allocation," *IEEE Trans. on Computers*, vol. 54, no. 10, pp. 1242–1257, 2005.

[7] S. Derrien, I. Puaut, P. Alefragis, M. Bednara, H. Bucher, C. David, Y. Debray, U. Durak, I. Fassi, C. Ferdinand, D. Hardy, A. Kritikakou, G. K. Rauwerda, S. Reder, M. Sicks, T. Stripf, K. Sunesen, T. D. ter Braak, N. S. Voros, and J. 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*, pp. 286–289, 2017.

[8] C. Ferdinand and R. Heckmann, "aiT: worst case execution time prediction by static program analysis," in *Building the Information Society, IFIP 18th World Computer Congress, Topical Sessions, 22-27 August 2004, Toulouse, France*, pp. 377–383, 2004.

[9] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "A practical automatic polyhedral parallelizer and locality optimizer," in *Conference on Programming Language Design and Implementation*, PLDI '08, (New York, NY, USA), pp. 101–113, ACM, 2008.

[10] S. M. ann Yau-Tsun Steven Li, "Performance analysis of embedded software using implicit path enumeration," in *32nd Design Automation Conference*, pp. 456–461, 1995.

[11] A. Floch, T. Yuki, A. E. 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 *13th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2013, Eindhoven, Netherlands, September 22-23, 2013*, pp. 100–105, 2013.

[12] P. Clauss, "Counting solutions to linear and nonlinear constraints through ehrhart polynomials: Applications to analyze and transform scientific programs," in *Proceedings of the 10th International Conference on Supercomputing*, ICS '96, (New York, NY, USA), pp. 278–285, ACM, 1996.

[13] S. Verdoolaege, R. Seghir, K. Beyls, V. Loechner, and M. Bruynooghe, "Counting integer points in parametric polytopes using barvinok's rational functions," *Algorithmica*, vol. 48, pp. 37–66, May 2007.

[14] H. Li, I. Puaut, and E. Rohou, "Traceability of flow information: Reconciling compiler optimizations and WCET estimation," in *22nd International Conference on Real-Time Networks and Systems, RTNS '14, Versaille, France, October 8-10, 2014*, p. 97, 2014.