Applied Computing and Informatics 13 (2017) 184–193

Contents lists available at ScienceDirect

Applied Computing and Informatics

journal homepage: www.sciencedirect.com

Original Article

Combining loop unrolling strategies and code predication to reduce the worst-case execution time of real-time software

Andreu Carminati [a,b,*], Renan Augusto Starke [c], Rômulo Silva de Oliveira [a]

[a] Universidade Federal de Santa Catarina, DAS-CTC-UFSC, Caixa Postal 476, Florianópolis, SC, Brazil
[b] Instituto Federal de Santa Catarina, Campus Gaspar, Gaspar, SC, Brazil
[c] Instituto Federal de Santa Catarina, Campus Florianópolis, Florianópolis, SC, Brazil

A R T I C L E I N F O

Article history:
Received 19 October 2016
Revised 29 March 2017
Accepted 31 March 2017
Available online 13 April 2017

Keywords:
Real-time systems
Loop unrolling
Worst-case execution time
Predication

A B S T R A C T

Worst-case execution time (WCET) is a parameter necessary to guarantee timing constraints on real-time systems. The higher the worst-case execution time of tasks, the higher will be the resource demand for the associated system. The goal of this paper is to propose a different way to perform loop unrolling on data-dependent loops using code predication targeting WCET reduction, because existing techniques only consider loops with fixed execution counts. We also combine our technique with existing unrolling approaches. Results showed that this combination can produce aggressive WCET reductions when compared with the original code.

© 2017 Production and hosting by Elsevier B.V. on behalf of King Saud University. This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/).

1. Introduction

WCET is an important parameter necessary for the development of real-time applications and systems. Usually denoted by C, or computation time of a task, this parameter is required by virtually all scheduling strategies that can be used to guarantee that all tasks of a given real-time system will meet their deadlines [1–3]. The schedulability of a real-time system can be enhanced if we reduce tasks' worst-case execution times and consequently the processor demand. For this objective, we can employ a faster processor, which is usually an expensive approach, or optimize the software responsible for the tasks function. The second approach can be performed automatically at compile time using code optimization strategies, as done by [4,5].

The key aspect of optimizations directed to WCET reduction is the use of timing analyzers to evaluate the result of code transformations in terms of WCET. Such timing analyzers are frequently connected with compilers to inform if a code transformation increases, decreases or has no effect concerning WCET. This compiler integration with WCET analyzers is important to discover which path is responsible for generating the worst-case execution time, and to track any path change due to successive code transformations.

Loops are frequently good target candidates for compiler optimizations to extract performance of modern processor architectures. Loop unrolling is a well-known technique used to improve average-case performance of programs. This technique consists in replicating the loop body for a certain number of times to avoid branch and jump overhead and to reduce the number of increment/decrement operations, inserting extra code to verify exiting corner cases, if necessary. The number of body replications is often called *unrolling factor* and the original loop is called *rolled loop*.

Loop unrolling can contribute to improve the instruction level parallelism (ILP) and execution performance of programs, by enabling more optimization that are affected by code expansion. Although, this code expansion can lead to instruction-cache performance degradation, if not carefully applied. If loop unrolling is applied before the register allocation phase, register pressure can be increased, leading to the insertion of more spill and reload operations in the code. However, a standard compiler cannot use loop unrolling directly if worst-case execution time (WCET) reduction is desirable, due to the instability of the execution path that generates the worst possible execution time and negative cache effects. Some techniques were proposed in the literature to achieve WCET reduction using loop unrolling, as in [4,6]. In these works, loops are

* Corresponding author at: Universidade Federal de Santa Catarina, DAS-CTC-UFSC, Caixa Postal 476, Florianópolis, SC, Brazil.
E-mail address: andreu.carminati@posgrad.ufsc.br (A. Carminati).

Peer review under responsibility of King Saud University.

Production and hosting by Elsevier

http://dx.doi.org/10.1016/j.aci.2017.03.002
2210-8327/© 2017 Production and hosting by Elsevier B.V. on behalf of King Saud University.
This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/).

carefully unrolled to promote WCET reduction and limit code increase. But, only loops with fixed execution counts are considered.

The contribution of this paper is twofold. Firstly, we propose an alternative way to perform loop unrolling on loops with arbitrary (or variable) execution counts. Traditionally, loops with unknown execution counts are unrolled with fixed unrolling factors, with the corner conditions (i.e, the unrolling factor is not a multiple of the number of iterations) treated with branch instructions. The approach adopted in this work is to treat the same corner conditions using code predication instead of instructions that perform control flow changes. Code predication is already used in software pipelining of loops, but its application directly with loop unrolling was not reported in the literature. Code predication also can be explored using a transformation called *If-Conversion*, which is a standard compiler optimization that converts control dependencies into data dependencies, removing branches.

The second contribution of this paper is the combination of our technique with other standard unrolling approaches for data dependent loops and loops with fixed execution counts. In this way, we can decide on a per loop level which of the approaches should be used for loop unrolling. This combination of techniques is important because not every loop can be unrolled in the same way. For example, loops with variable number of iterations must include compare and branch instructions to treat different exit conditions, but loops with static execution counts can be unrolled without these instructions. The necessity of compare and branch instructions is not the only difference when unrolling these two types of loops, but the selection of a valid unrolling factor is also different. In loops with a static number of iterations, we can only consider unrolling factors that perfectly divide such number of iterations. Until the present moment, no work addressing the combination of different unrolling techniques was identified in the literature.

We evaluated the results on an experimental processor, which implements a subset of the *very long instruction word* (VLIW) ST231 ISA that was extended to support a simplified full-predication mechanism.

The remainder of this paper is organized as follows: Section 2 outlines the related work on loop unrolling directed to WCET reduction. Section 3 shows the motivations of this work. Section 4 explains the proposed approach to perform loop unrolling targeting real-time applications. In Section 5 we describe briefly our testbed. Section 6 presents the obtained results using a benchmark suite. In Section 7 presents our conclusions and final remarks.

2. Related work

The first work that concerns WCET reduction using loop unrolling, consists in applying this optimization directly at assembly level [4]. In this work, only innermost loops with fixed number of iterations are unrolled and the unrolling factor used for all loops is 2. Although, not all candidate loops are unrolled, but only those that are present in the worst-case execution path (WCEP), and they are kept unrolled only if WCET reduction is achieved. At every optimization application, the WCET information must be re-calculated to update the worst-case path information that drives the algorithm. This recalculation is necessary because any code change that affects the WCET may result in a WCEP change. These WCET recalculations are a common strategy employed by compilers focused in worst-case execution time reduction. Experiments using a processor with no caches showed that a WCET reduction up to 10% was achieved for all benchmarks.

Another approach to perform loop unrolling aiming at WCET reduction was proposed in [6]. Here, the optimization is applied at the source code level and uses a processor with instruction cache and scratchpad memory. As the optimization is applied at the source code level, the success of next optimizations performed by the compiler is enhanced, specially for those that benefit from code expansion. The key aspects of the technique are: (1) choose the most profitable loops concerning WCET reduction and (2) calculate an unrolling factor considering memory constraints. Consequently, the algorithm balances memory utilization and WCET reduction.

Both the previously presented approaches consider only loops with fixed number of iterations. In fact, both techniques can be used to unroll loops with arbitrary counts or data-dependent loops, providing necessary code to exit the loop when the termination condition is reached. This code is commonly generated as branch instructions.

If-Conversion [7] is a technique used to convert control dependencies into data dependencies. The basic principle consists in eliminating gotos and branches and inserting logical variables to control the execution of instructions in the program. *If-Conversion* can be performed at IR-level or machine-level as stated by [8] and is related to region enlargement techniques used to expand the instruction scheduling scope beyond a single basic block, which is specially beneficial for *very long instruction word machines* (VLIW).

The application of *If-Conversion* techniques in loops is not a novel idea. Software pipeline [9] can benefit from *If-Conversion* and code predication to control the execution of prologue and epilogue of pipelined loops [10]. Another technique that can benefit from *If-Conversion* is loop flattening [11]. Loop flattening is a form of software pipelining that merges nested loops into a single loop body, providing necessary code to control the execution and the flow of data between blocks. In [12] *If-Conversion* is used to eliminate back-edges of flattened loops. The next section outlines the motivation and the key ideas behind the proposed unrolling technique.

3. Motivation

We can consider the loop of the code listing of Listing 1 as a motivational example. This code shows a loop with the number of iterations dependent on the value of a variable (called data-dependent loop). For this loop, a compiler commonly generates a control flow structure that is shown in Fig. 1. In this structure, a simple *for* loop has two basic blocks called *header* and *body* which are surrounded by an *entry* and an *exit* basic blocks.

There are some approaches to perform the loop unrolling optimization considering this loop. The simpler strategy consists in optimizing only loops with fixed counts. In this case, the compiler chooses an unrolling factor that exactly divides the number of iterations of the loop. If a compiler is able to optimize data-dependent

```
1  void loop(int a){
2      int i, j = 0, k = 0;
3
4      for(i = 0; i < a ; i++){
5          j++;
6          k++;
7      }
8  }
9
10 int main(int a){
11         loop(90);
12 }
```

Listing 1. Simple data-dependent loop.

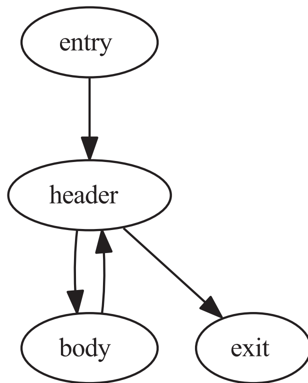

Fig. 1. Control flow graph of Listing 1.

loops with unknown number of iterations, it must take care of left-over iterations. Another problem with data-dependent loops is the difficulty to choose an effective unrolling factor.

Listing 2 shows the application of loop unrolling on the data-dependent loop of Listing 1 (in C code for simplicity). Is this case, if the compiler is not able to calculate the number of iterations for the loop, or determine whether this number is odd or even, it must check the exit condition on every body replication, as done by the *if* statements. This condition checking leads to a control flow graph that is shown in Fig. 2.

Note that with this approach, the number of branching instructions is augmented, also increasing the number of basic blocks. From the WCET perspective, by increasing the number of basic blocks, we increase the search space that contains the worst-case execution path that produces the WCET. From the code generation point of view, a branch may need up to 3 intermediate operations that are not necessarily in this order: (1) condition calculation, (2) target address calculation and (3) branch execution. Depending on the target architecture, all previous operations are executed by one instruction or are segmented in sequences of 2 or 3 instructions.

Considering the use of a target architecture with instruction predication support, it is possible to remove branch operations (if any) from loops that are unrolled. For this purpose, we can consider the loop of Listing 3 and its respective CFG shown in Fig. 3. This loop is semantically equivalent to the loop of Listing 2. If we can rewrite an unrolled loop in terms of conditional expressions, as done to Listing 2 to obtain Listing 3, it is possible to apply *If-Conversions* to the code. Until the writing of this paper, no techniques exist in the literature to perform these transformations to unrolled data-dependent loops. As we stated before, *If-Conversion* is an optimization technique that converts control dependence into data dependence through the definition of guards to control the execution of instructions. If the target architecture supports instruction predication, *If-Conversion* can result in branchless code, reducing code size and number of basic blocks. Generically, an application of *If-Conversion* to the code of Listing 3 would produce the control flow graph of Fig. 4. The prefix (p) means that the execution of the basic blocks *body 2* and *body 3* are conditioned to some predication guard p.

From the WCET perspective, the common behavior of analyzers is to consider the complete execution of the loop iterations. In this way, the last iteration will be considered fully executed, even with the possibility of an early loop exit if the condition is reached. If a

```
1  void loop(int a){
2      int i,j = 0,k = 0,l = 0;
3
4      for(i = 0; i < a; i+=l){
5          l = 0;
6          j++;
7          k++;
8          l++;
9          if(i+l >= a) break;
10         j++;
11         k++;
12         l++;
13         if(i+l >= a) break;
14         j++;
15         k++;
16         l++;
17     }
18 }
```

Listing 2. Unrolled loop.

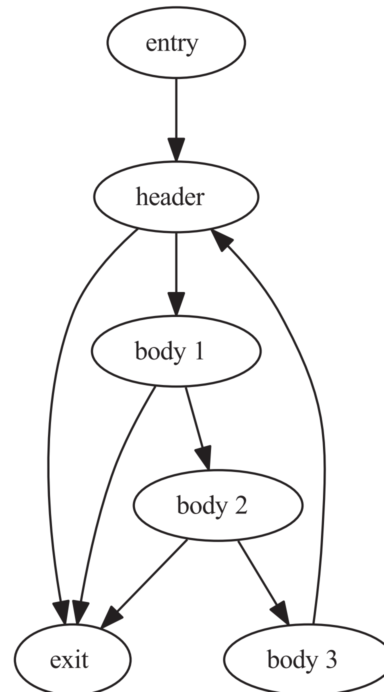

Fig. 2. Control flow graph of Listing 2.

```
1  void loop(int a){
2      int i, j = 0, k = 0;
3
4      for(i = 0; i < a;){
5          j++;
6          k++;
7          i++;
8
9          if(i < a){
10             j++;
11             k++;
12             i++;
13         }
14         if(i < a){
15             j++;
16             k++;
17             i++;
18         }
19     }
20 }
```

Listing 3. Unrolled loop rewritten with conditional expressions.

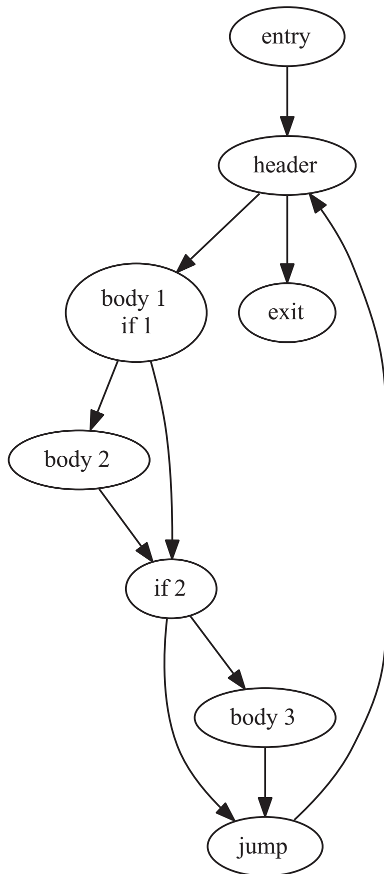

Fig. 3. Control flow graph of Table 3.

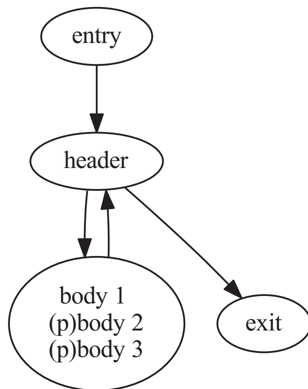

Fig. 4. Control flow graph representing an *If-Conversion* of the code from Listing 3.

loop is always fully executed in the worst case, it is beneficial to reduce the number of instructions of the unrolled loop, and if premature exits will never be taken (branch instructions), we can eliminate them from the code using predication. Using code predication we decrease the number of instructions while preserving the semantics of the code.

In the next section, we present our approach to perform loop unrolling which applies simultaneously code predication directly in machine code. The technique starts from a simple data-dependent loop and directly generates an unrolled and predicated version, as done step-by-step in this section. The main improvement of our approach is that it avoids the use of branch instructions, differently from what is usually done by traditional techniques.

4. Our loop unrolling approach

Our loop unrolling algorithm performs code predication in conjunction with the unrolling steps. In this way, sophisticated *If-Conversion* strategies can be avoided. The algorithm must be used directly in assembly representation. The architectural requirement of the technique is the existence of full-predication mechanisms to control the execution of instructions. As example of such mechanisms, we can cite IA-64 [13] and ARM [14] (except for Thumb instructions). The technique also benefits from branches that are segmented in sequences of more than one operation.

The steps to unroll a loop are shown by Algorithm 1. The algorithm assumes that every loop that will be unrolled is composed by a header and a body. This constraint must be ensured by the caller of the algorithm procedure. Another requirement is the implementation of loop headers with compare instruction followed by branch instructions to control the loop exit.

Algorithm 1. Predicated Loop Unrolling algorithm.

1: **procedure** PREDICATEDLOOPUNROLLING($Loop, U, P$) ▹Unroll loop u times
2: $Header \leftarrow loopHeader(Loop)$
3: $Body \leftarrow loopBody(Loop)$
4: $removeUncondBranch(Body)$
5: $BodyCopy \leftarrow createCopy(Body)$
6: **for** $i \leftarrow 1$ to $U - 1$ **do**
7: $NewHeader \leftarrow createCopy(Header)$
8: $removeConditionalBranch(NewHeader)$
9: $changeCompareOutput(NewHeader, P)$
10: $Body \leftarrow unify(Body, NewHeader)$
11: $NewBody \leftarrow createPredCopy(BodyCopy, P)$
12: $Body \leftarrow unify(Body, NewBody)$
13: **end for**
14: $insertUncondBranch(Body, Header)$
15: **end procedure**

The algorithm works as follows: First, header and body are identified, which is done by Lines 2 and 3. The second step removes the unconditional branch from the loop body to the header. This branch instruction will be re-inserted at the end of the algorithm, as a last instruction (Line 14). The next step is to unroll the loop using the provided unrolling factor, using the original loop body as first copy.

For each unroll step, we create a copy of the header, converting control flow instructions into instructions that control the predication of subsequent copies of the loop body (Lines 7–10). Then, we make a predicated copy of the body that is amended at the end of the original body (Lines 11 and 12). The algorithm basically removes forward branches used to exit the loop and inserts boolean guards to control the execution of the remaining part of the loop. These guards are stored in the P variable.

It is relevant to notice that the first copy of the loop body does not need to be predicated, because the header condition verification ensures that at least one iteration (in relation to the rolled loop) must be executed, otherwise the loop must be already terminated. In this way, the first copy of the body represents exactly the original basic block of the loop.

4.1. Example

As an example, the algorithm is applied to the Listing 1. The assembly code dialect used is referent to the ST231 ISA, which is

also used in our testbed. We omit bundles delimitation in code listings for simplicity. Before the unroll, the sequence of instructions generated is shown in Listing 4.

After the application of the algorithm and using an unrolling factor of 2, we obtain the code as shown in Listing 5. The sequence (*p*) means that the operation execution is conditioned to the content of the flag register *p*, which is a common notation of predicated code. For comparison purposes, the same code is unrolled in the standard way as shown in code listing of Listing 6. Comparing the two approaches, we can see that the predicated version presented fewer instructions than the standard counterpart (with branches).

4.2. Combining loop unrolling techniques

As each loop unrolling technique can be applied to a set of loops that share a certain characteristic, makes more sense to combine the techniques to get a more aggressive WCET reduction, instead of comparing them. In this way, we decide in a per loop level which approach should be applied.

Depending on loop attributes, we consider three unrolling alternatives:

```
1     add $r8 = $zero, 0
2     add $r9 = $zero, 0
3     add $r10 = $zero, 0
4   HEADER:
5     cmplt $br0, $r9, $r16
6     brf $br0, $EXIT
7   BODY:
8     add $r10 = $r10, 1
9     add $r8 = $r8, 1
10    add $r9 = $r9, 1
11    goto $HEADER
12  EXIT:
13
14
15
16
17
18
```

Listing 4. Example of loop in assembly code.

```
1     add $r8 = $zero, 0
2     add $r9 = $zero, 0
3     add $r10 = $zero, 0
4   HEADER:
5     cmplt $br0, $r9, $r16
6     brf $br0, $EXIT
7   BODY:
8     add $r10 = $r10, 1
9     add $r8 = $r8, 1
10    add $r9 = $r9, 1
11    cmplt $p, $r9, $r16
12    (p)add $r10 = $r10, 1
13    (p)add $r8 = $r8, 1
14    (p)add $r9 = $r9, 1
15    goto $HEADER
16  EXIT:
17
18
```

Listing 5. Example of unrolled loop using code predication.

```
1     add $r8 = $zero, 0
2     add $r9 = $zero, 0
3     add $r10 = $zero, 0
4   HEADER:
5     cmplt $br0, $r9, $r16
6     brf $br0, $EXIT
7   BODY0:
8     add $r10 = $r10, 1
9     add $r8 = $r8, 1
10    add $r9 = $r9, 1
11    cmplt $br0, $r9, $r16
12    brf $br0, $EXIT
13  BODY1:
14    add $r10 = $r10, 1
15    add $r8 = $r8, 1
16    add $r9 = $r9, 1
17    goto $HEADER
18  EXIT:
```

Listing 6. Example of unrolled loop using the standard approach.

Standard without branches
For loops that are not data-dependent (fixed execution counts), we can use the simplest loop unrolling approach. This approach replicates the loop body using an unrolling factor that divides the execution count of the loop. As this approach is a common strategy considering compiler optimization, we will omit its representation in pseudo-code. If we use this approach, we will refer to *simpleLoopUnrolling*(*loop*, *unrollingFac*) as the algorithm representation and its parameters.

Standard with branches
For data-dependent loops with some kind of control flow change inside of the loop body, we can use loop unrolling with compare and branch instructions to exit the loop when the condition is reached. For simplicity, we apply this unrolling alternative to loops with call instructions in the body. This approach is also a common strategy considering compiler optimization, and we will omit its representation in pseudo-code. If we must use this approach, we will refer to *branchedLoopUnrolling*(*loop*, *unrollingFac*) as the algorithm representation and its parameters. This approach cannot be used with function inlining, although this is not a problem because we do not use this type of optimization for two reasons: (1) we do not apply any optimization when we cannot quantify WCET effects. (2) with inlining, we lose the one-to-one mapping between the object code and source code, which is necessary to perform WCET calculation.

Predicated
For data-dependent loops with simple loop bodies, we can use the predicated version. We cannot use this type of unrolling in loops with call instructions because condition or flag registers are not commonly exposed to the calling conventions used in processors. If we had to save the flag registers, it would be better to use the previous approach. We will call this approach as *predicatedLoopUnrolling*, as presented by Algorithm 1.

Algorithm 2 chooses the adequate unrolling technique through inspection of the loop characteristics. The field *loop.uf* represents the unrolling factor that must be used for a specified loop. We cannot choose unrolling factors arbitrarily if our objective is WCET reduction. In the next section, we will show how to use WCET information to choose adequate unrolling factors.

Algorithm 2. Optimization algorithm that is executed by the compiler.

```
 1: procedure OptimizeLoops(Program)
 2:     LoopList ← getLoops(Program)
 3:     for each loop ∈ LoopList do
 4:         if not loop.isDataDep then
 5:             simpleLoopUnrolling(loop, loop.uf)
 6:         else if loop.hasCall then
 7:             branchedLoopUnrolling(loop, loop.uf)
 8:         else
 9:             predicatedLoopUnrolling(loop, loop.uf, P)
10:         end if
11:     end foreach
12: end procedure
```

In relation to the predication flag that must be given as parameter of Algorithm 1 in Line 9, the same register can be used to hold all conditions for all loops, because each copy of the loop body must be guarded by only one condition, i.e., that related to the exit condition of the loop, which is updated before the execution of this body copy. In this way, we can pass any register or flag that can be used to predicate instructions. In this algorithm, we consider candidates for loop unrolling: (1) innermost loops and (2) loops composed by two basic blocks header and body, as the example of Fig. 1. We use these restrictions to process only small loops, where we can easily achieve gains using loop unrolling.

4.3. Ensuring WCET reduction by unrolling factor selection

The previous algorithm is responsible for unrolling the loops of a program using a set of unrolling factors. It is also necessary to choose a unrolling factor for each loop that minimizes the WCET. As we are interested only in verifying the effectiveness of our technique, we are not concerned in choosing an optimal unrolling factor considering code increase and WCET reduction.

We adopted a scheme that tries to iteratively choose an unrolling factor for each loop in the program. If the loop has no impact on the worst-case execution time, i.e. resides outside the WCEP (*worst-case execution path*), it will be kept rolled, otherwise it will be unrolled. The set of unrolling factors will vary according to characteristics of the loop, such as data dependency and parity of execution counts.

If the unrolled loop increases the WCET, then it will be also kept rolled. Otherwise it will be maintained unrolled using the factor that best minimizes the WCET considering the previously considered ones from the set. Each loop is processed exactly once, and after each loop handling the WCET (and WCEP) information must be updated to guide the treatment of the next loops. To verify if a WCET increase occurs, it is necessary to perform a program recompilation and an invocation to the WCET analyzer. We do not reconsider loops in case of path changes, since typically all loops in a program are on the WCEP, as stated by [6]. We only check if the current loop is on the WCEP.

Algorithm 3 presents our approach for selection of unrolling factors. This algorithm is designed to be executed as a complementary part of the compilation process, and can be implemented as a separated tool. Regarding the flow of information point of view, it is necessary the following interactions between the compiler and the algorithm:

- Compiler → Algorithm: the compiler must export all information related to all loops that can be unrolled. The information must allow the correlation between the loops and the worst-case execution time related data. Execution counts must be exported as well. In case of data-dependent loops, execution counts can be provided as annotations in the source code, for example. These execution counts are also necessary for the calculation of the worst-case execution time.
- Algorithm → Compiler: The algorithm can provide unrolling factors for all loops that were exported for a determined program. If such unrolling factors are not provided, the compiler keeps the loops rolled. To decide which unrolling factor to use, the algorithm uses WCET analysis and loop information.

As we can see, the previous relation between compiler and algorithm forms a cyclic and incremental approach to optimize loops. The parameter of Algorithm 3 is the representation of a compiled program. The first step of the algorithm is to retrieve a list of (exported) loops of the program representation (Line 2) followed by a WCET analysis (Line 3). The main loop of the algorithm iterates over the loop list (Line 4), considering only loops that are in the WCEP (Line 5). Then, we assume that it will be kept rolled (Line 6) if its is not possible to choose an unrolling factor. The next step consists into test different unrolling factors in the interval $[2, 17]$. If a loop can be unrolled, we have basically two alternatives to consider an unrolling factor as valid:

Data independent loops for data-independent loops (Line 8), the unrolling factor must exactly divide the execution count of the loop (Line 9), because we do not want to generate instructions to control the reaching of the exit condition inside the replicated copies of the body.

Data dependent loops for data-dependent loops (sentence of Line 8 is evaluated to false) we do not test if the unrolling factor divides the execution count, because leftover iterations are treated explicitly by compare and branch or compare and predicate instruction, depending on the approach. However, we use a heuristic approach that considers factors whose parity is equal to the loop bounds' parity (Lines 13).

Note that a loop can have more than one possible unrolling factor from the interval $[2, 17]$. In this case, we will use the one that most reduces the WCET. Experience says that even small unrolling factors can produce instruction cache degradation, so, the interval $[2, 17]$ is able to cover a useful range of unrolling factors for real programs. Although, depending on the compiler used and processor characteristics, these values can be tuned experimentally. If such unrolling factor exists, we recompile the program and test for WCET changes. In case of WCET increase (Line 21), we use the last chosen unrolling factor (Line 22) and skip to the next loop. Otherwise we use the actual factor updating the WCET (Line 24 and 25).

The algorithm only cares about data dependency and parity of execution counts to choose unrolling factors. The final decision about which unrolling approach must be applied to data-dependent loops is left to the compiler that implements Algorithm 2.

Algorithm 3. Algorithm that defines unrolling factors for all optimizable loops in *Program*.

```
1:  procedure calculateUnrollingFactors(Program)    ▷ Algorithm executed by the optimization planning tool
2:      LoopList ← getLoops(Program)
3:      wcetData ← calculateWCET(Program)
4:      for each loop ∈ LoopList do
5:          if isInWCEP(loop, wcetData) then
6:              lastUF ← 0
7:              for i ← 2 to 17 do
8:                  if not loop.isDataDep then
9:                      if not divides(loop.bound, i) then
10:                         continue
11:                     end if
12:                 end if
13:                 if parity(loop.bound) = parity(i) then
14:                     loop.uf ← i
15:                     recompile(Program)
16:                     newWcet ← calculateWCET(Program)
17:                     if newWcetData.value >= wcetData.value then
18:                         loop.uf ← lastUF
19:                     else
20:                         wcetData ← newWcetData
21:                         lastUF ← i
22:                     end if
23:                 end if
24:             end for
25:         end if
26:     end for each
27: end procedure
```

The time complexity of Algorithm 3 is $O(n^2)$, where n is the number of loops. In fact, there will be, for any program, one initial invocation to the analyzer to estimate the WCET and other invocations for each loop to test the considered unrolling factors, giving a total of $1 + 16 \times n$ invocations to the analyzer. The worst case occurs when all loops are data independent and can be divided by factors in the interval $[2, 17]$. In practice, this situation only occurs when the loop count represents a common multiple of all considered unrolling factors. For each loop considered in this algorithm, an invocation to *recompile(Program)* must be performed (Line 15). In this invocation, all loops will be unrolled (Algorithm 2 is invoked inside the compiler), justifying the quadratic complexity. Although our approach is simple, complex heuristics that try to balance code expansion and WCET as proposed by [6] can be applied as well. Our combination of loop unrolling strategies can increase the compilation time due to the fact that we need to process more code that in the original program. Choosing adequate unrolling factors can also increase considerably the compilation time due to the necessity of WCET analyses. As pointed by [6], performance improvements are a primary focus of embedded systems, being longer compilation times of less importance.

5. Evaluation

To evaluate the technique proposed in this paper we conducted experiments using an architecture with a simplified complete predication support. In the next subsections, we will give a short description of the target architecture, compiler support and WCET analyzer. The concluding part of this section presents numerical results obtained from the use of the proposed technique applied to a set of benchmarks commonly used in the literature.

5.1. Target architecture

The architecture used to evaluate the proposed technique was an experimental deterministic 32-bit microprocessor [15,16] with RISC instructions that follows a subset of the HP VLIW ST231 ISA [17]. The processor has a VHDL model, which can be both simulated and synthesized to a FPGA. This processor has a VLIW (*very long instruction word*) design. VLIW is a design philosophy where the hardware is not only exposed by instructions but the instruction level parallelism (ILP) is exposed as well. Since the processor is intended for real-time applications, n-associative or shared caches and out-of-order execution are not utilized. The processor has an instruction cache memory comprising 32 lines with 256 bits per line forming a 1024 kb direct-mapped cache memory. The processor does not have a data cache, but has a scratchpad memory.

The processor has a four issue five stage static scheduled pipeline. Each instruction (or bundle) encodes up to four operations that will be dispatched in parallel. The used processor has an ISA extension that enables code predication in a simplified way through the thirtieth bit, which is otherwise unused. If an operation has its 30th bit enabled, then the result of the instruction will only be committed if the predication flag is configured to true. If the predication flag has false as value, then the operation will produce no effect (or nop). The predication flag is a 1-bit register that can be accessed through comparison instructions. This flag is connected to the branch register number 4 that is already defined by the ISA. In this way, the branch register number 4 controls the execution of predicated instructions.

5.2. Compiler and WCET analysis

We used a custom compiler back-end developed for the target architecture using the LLVM [18] infrastructure. The compiler also produces all necessary information for WCET calculation, as an annotated CFG containing information like loop bounds and mappings between CFG nodes (basic blocks) and a target program code. For data-dependent loops, worst-case execution counts (or bounds) must be provided through source code annotations.

We also used a custom WCET analyzer implemented in C++, that produces cycle-accurate estimates of the worst-case execution time for the target architecture considering the program binary and the data produced by the compiler. This WCET analyzer produces an annotated CFG that includes worst-case counts for each basic block and a single numerical value that represents the calculated WCET.

The unrolling technique was implemented in our back-end at the end of machine code generation. It is difficult to perform WCET-oriented optimization using LLVM due to its highly optimized pass-manager that isolates the treatment of each function of a compilation unit. Due to this fact, we cannot optimize the program as a whole aiming at WCET reduction using the standard LLVM pass-manager because the generated code is only fully materialized at the end of the complete process. Moreover, the pass-manager deallocates any machine related code representation structures of a function after writing its generated object code to file at the end of the pass-manager execution. So, when we can finally calculate the WCET of a program, we cannot use this data to change the code (optimization application), because the needed intermediate structures no longer exist. Due to this fact, strategies like that proposed by [5], where the analyzer is invoked directly by the compiler to take optimization decisions cannot be used.

To overcome this limitation, we adopted an approach similar to [19]. In this approach, a tool in a higher or planning level is

responsible to select the parts of the program that must be optimized, using WCET information as guidance. This tool shares a database with the compiler that is used as communication channel. This database stores facts about the structure of the program and values that specify if such structure must be touched by a specific optimization. The tool invokes the compiler to generate the object code and data used as input for the WCET analyzer. After that, WCET information is obtained through the WCET analyzer. Using this information, the planning tool updates the database using heuristics like the one proposed in Section 4.3, which chooses the loops and unrolling factors and invokes the compiler again. This task repeats until WCET stabilization or when the entire code is already analyzed by the planning tool.

Using this strategy, we can perform any WCET-oriented optimization in an iterative way. Optimizations must keep consistency between the transformed code and the annotations provided in the source. In this way, if a data-dependent loop is unrolled, the execution bound provided as an annotation must be updated before the WCET calculation. A simplified diagram of our tools and their connection can be seen in a figure on the Supplementary material. As we can see, both LLVM infrastructure and planning tool share a data-base containing information about loops, which is empty on the first program compilation. From the first compilation, the planning tool can invoke the WCET analyzer tool and execute Algorithm 3 to choose an unrolling factor for each loop.

6. Results

We used the Mälardalen WCET benchmarks [20] to evaluate the effectiveness of the proposed technique. These benchmarks are widely used to evaluate and compare methods and techniques related to WCET analysis. We excluded benchmarks with indirect recursion. We considered a constant time for complex library function calls, as those that are used to handle floating point numbers. The description of each benchmark can be seen in a table on the Supplementary material.

The results of the experiments are shown in Table 1. The column *Initial WCET* shows the WCET of the benchmark without the application of the loop unrolling. *Initial code size* presents the size of the code (in bytes) in this initial scenario. *Optimized WCET* presents the WCET of the optimized version with its respective code size (*Optimized code size*). The columns *WCET reduction* and *Code increase* present the percentage of WCET reduction and its relative code augmentation, respectively. *WCET reduction* is calculated as $\frac{\text{Initial WCET} - \text{Optimized WCET}}{\text{Initial WCET}} \times 100$ and *Code increase* as $\frac{\text{Optimized code size} - \text{Initial code size}}{\text{Initial code size}} \times 100$. We omitted in this table benchmarks where no gain was obtained. In this way, a total of 18 from 33 benchmarks are shown.

Table 2 shows how many loops of each type were unrolled and the maximum unrolling factor (*Max. uf*) in each benchmark.

Analyzing the obtained results, we can see that the combination of techniques was able to reduce the WCET of half of the benchmarks. For example, considering the *adpcm.c* benchmark, we achieved a small WCET reduction (1.19%) in contrast with a higher code increase (31.10%). If we look at Table 2 we can see that two loops of *adpcm.c* were unrolled (one with fixed execution count and another with a call instruction), and the maximum unrolling factor used was 2. On the other hand, we can see a high WCET reduction for the *exptint.c* benchmark, with less code increase then in the *adpcm.c*. In this benchmark, only one loop was unrolled, with an unrolling factor of 7. The average WCET reduction considering all benchmarks was 6.72%, while the average code increase was 15.56%. As maximum values, we got 32.44% and 80.19%, for WCET reduction and code increase, respectively.

Table 2
Obtained results

Benchmark	Simple	With pred.	With branch	Max. uf
adpcm.c	2	0	2	2
bsort100.c	1	0	0	5
cnt.c	0	0	1	2
compress.c	1	1	0	2
crc.c	0	0	1	2
duff.c	0	1	0	10
edn.c	4	1	0	9
expint.c	0	1	0	7
fft1.c	0	1	2	13
fir.c	0	1	0	6
insertsort.c	0	1	0	3
jfdctint.c	1	0	0	4
lms.c	0	1	3	15
ludcmp.c	0	0	1	5
matmult.c	1	0	1	4
ndes.c	2	0	1	2
qsort-exam.c	0	0	1	2
st.c	0	0	1	4

Table 1
Obtained results

Benchmark	Initial WCET	Initial code size	Optimized WCET	Optimized code size	WCET reduction (%)	Code increase (%)
adpcm.c	19,607	10,208	19,373	14,816	1.19	31.10
bsort100.c	272,623	432	271,985	560	0.23	22.86
cnt.c	9046	752	8566	864	5.31	12.96
compress.c	140,139	4912	137,162	5488	2.12	10.50
crc.c	113,846	2048	112,671	2624	1.03	21.95
duff.c	1859	592	1397	816	24.85	27.45
edn.c	96,871	2336	73,042	3296	24.60	29.13
expint.c	113,473	1540	76,661	1812	32.44	15.01
fft1.c	1,034,634	19,984	727,844	44,272	29.65	54.86
fir.c	509,930,221	976	444,387,758	1616	12.85	39.60
insertsort.c	2720	304	2111	432	22.39	29.63
jfdctint.c	3947	1264	3568	1536	9.60	17.71
lms.c	352,360,015	14,016	303,674,957	70,768	13.82	80.19
ludcmp.c	43,902	3296	43,622	4320	0.64	23.70
matmult.c	268,362	1008	225,657	1968	15.91	48.78
ndes.c	146,612	5376	144,282	7248	1.59	25.83
qsort-exam.c	503,582	2528	480,816	2784	4.52	9.20
st.c	1,480,353	5088	1,198,582	5856	19.03	13.11
				Average	6.72	15.56
				Maximum	32.44	80.19

Table 3
Comparing *predicatedLoopUnrolling* with *branchedLoopUnrolling*

Benchmark	Branched unroll. WCET	Branched code size	Predicated unroll. WCET	Predicated code size	WCET reduction (%)	Code size reduction (%)
compress.c	138,875	5968	137,162	5488	1.23	8.04
duff.c	1515	912	1397	816	7.79	10.53
edn.c	73,181	3072	73,042	3296	0.19	−7.29
fft1.c	727,868	44,272	727,844	44,272	0.00	0.00
fir.c	509,930,221	976	444,387,758	1616	12.85	−65.57
insertsort.c	2720	304	2111	432	22.39	−42.11
lms.c	305,531,002	83,248	303,674,957	70,768	0.61	14.99

Our approach could be applied to 7 benchmarks, which are *compress.c*, *duff.c*, *edn.c*, *fft1.c*, *fir.c*, *insertsort.c* and *lms.c*. To understand how much WCET reduction we can achieve with the predicated loop unrolling, we unrolled those three benchmarks with the *branchedLoopUnrolling* instead of *predicatedLoopUnrolling* algorithm, because every loop that can be unrolled with the last method can be unrolled with the first as well. The results are shown in Table 3. We can observe that the predicated loop unrolling has noticeable effects considering the *duff.c*, *fir.c* and *insertsort.c* benchmarks. For *insertsort.c* and *fir.c* benchmarks, using the branched approach, we simply do not achieve WCET reduction, so the loop is kept rolled, which also explains the difference in code sizes. As we can see, the predicated approach, even with its limited applicability, can exploit cases where the standard approach fails to get WCET reduction. In the *edn.c* case, we achieved WCET reduction with negative code decrease because the algorithm could use a higher unrolling factor with the predicated version (9 instead 5 for a branch).

It is important to say that these results can be enhanced using heuristics to find better unrolling factors to control code expansion [6], which is out of the scope of this paper.

7. Conclusion

The correct scheduling of tasks in a real-time system demands the knowing of the worst-case execution time of each of these tasks. The higher the worst-case execution times, the higher will be the resource demand for the associated system. To reduce the WCET of tasks, one approach that was proposed in the literature consists in applying compiler optimizations on the software that implements the task's behavior in a feedback oriented way. Since worst-case reduction is the objective, WCET timing analyzers provide this feedback, instead of execution profiles commonly used in average-case optimization. Loop unrolling for WCET reduction is considered by [4,6]. Though, in both works only loops with fixed iteration counts are unrolled.

We proposed in this paper an alternative way to perform loop unrolling with arbitrary iteration counts. Traditionally, this type of loop is unrolled using compare and branch operations to control different exit conditions or contexts. What we propose is the use of code predication to control the loop execution under different exit conditions, since worst-case analyzers tend to consider that each loop, even unrolled, is always fully executed up to its execution bound. The approach can be used in architectures with full predication support and is better applicable when branch operations are segmented in more than one step.

We introduced an algorithm that performs this code transformation directly at the machine code level (or assembly). In our framework, each data dependent loop of each benchmark is annotated with a safe loop bound that represents an upper bound on the execution count. After loop unrolling application, the annotation is transformed to reflect the new loop bound of the unrolled loop. Since our technique does not depend on branches, the number of instructions is reduced and the instruction scheduling scope is enhanced, as the whole body of the unrolled fits in a single basic block. This scope enhancement can enable more optimizations to be applied to the code.

We also proposed a strategy that selects which unrolling technique to apply in a per loop basis. For loops with fixed execution counts, we applied the standard technique that unrolls loops using unrolling factors that perfectly divide execution counts to avoid compare and branch instructions. For data dependent loops, we used our predicated or the branch-based approach, depending on the case.

We observed in the experiments that the combination of unrolling techniques was able to reduce the WCET of 18 from 33 benchmarks. For six benchmarks we obtained gains above 20%. In the experiments, we also showed that the predicated approach, even with its limited applicability, can exploit cases where the standard approach fails to get WCET reduction.

As we are not interested in code increase limitation, higher code expansion was observed as well. To work around this situation, techniques like [6] can be applied to our heuristic of unrolling factor selection.

Acknowledgments

The authors would like to thank CAPES and CNPq for the partial funding of this research.

Appendix A. Supplementary material

Supplementary data associated with this article can be found, in the online version, at http://dx.doi.org/10.1016/j.aci.2017.03.002.

References

[1] L. Sha, T. Abdelzaher, K.-E. Årzén, A. Cervin, T. Baker, A. Burns, G. Buttazzo, M. Caccamo, J. Lehoczky, A.K. Mok, Real time scheduling theory: a historical perspective, Real-Time Syst. 28 (2/3) (2004) 101–155, http://dx.doi.org/10.1023/B:TIME.0000045315.61234.1e.

[2] H. Back, H.S. Chwa, I. Shin, Schedulability analysis and priority assignment for global job-level fixed-priority multiprocessor scheduling, in: Proceedings of the Real-Time Technology and Application, 2012, pp. 297–306, http://dx.doi.org/10.1109/RTAS.2012.33.

[3] M. Short, Timing analysis for embedded systems using non-preemptive EDF scheduling under bounded error arrivals, Appl. Comput. Infor. 13 (2) (2017) 130–139. http://dx.doi.org/10.1016/j.aci.2016.07.001.

[4] W. Zhao, W. Kreahling, D. Whalley, C. Healy, F. Mueller, Improving WCET by applying worst-case path optimizations, Real-Time Syst. 34 (2) (2006) 129–152, http://dx.doi.org/10.1007/s11241-006-8643-4.

[5] H. Falk, P. Lokuciejewski, A compiler framework for the reduction of worst-case execution times, Real-Time Syst. 46 (2) (2010) 251–300, http://dx.doi.org/10.1007/s11241-010-9101-x.

[6] P. Lokuciejewski, Peter Marwedel, Worst-case Execution Time Aware Compilation Techniques for Real-time Systems, Springer Science & Business Media, 2010.

[7] J.R. Allen, K. Kennedy, C. Porterfield, J. Warren, Conversion of control dependence to data dependence, in: Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages – POPL '83, ACM Press, New York, New York, USA, 1983, pp. 177–189, http://dx.doi.org/10.1145/567067.567085.

[8] A. Jordan, N. Kim, A. Krall, IR-level versus machine-level if-conversion for predicated architectures, in: Proceedings of the 10th Workshop on

Optimizations for DSP and Embedded Systems – ODES '13, ACM Press, New York, New York, USA, p. 3, http://dx.doi.org/10.1145/2443608.2443611.
[9] A.E. Charlesworth, An approach to scientific array processing: the architectural design of the AP-12OB/FPS-164 family, IEEE Comput. (1981) 18–27.
[10] J.C. Dehnert, P.Y.-T. Hsu, J.P. Bratt, Overlapped loop support in the Cydra 5, in: ACM SIGARCH Computer Architecture News, vol. 17 (2), 1989, pp. 26–38. http://dx.doi.org/10.1145/68182.68185.
[11] R.V. Hanxleden, K. Kennedy, Relaxing SIMD control flow constraints using loop transformations, ACM SIGPLAN Not. 27 (7) (1992) 188–199, http://dx.doi.org/10.1145/143103.143133.
[12] Q. Pop, S. Yazdani, R. Neill, Improving GCC's auto-vectorization with if-conversion and loop flattening for AMD's Bulldozer processors, in: Proceedings of the GCC Developers' Summit 2010, 2010.
[13] R. Geva, D. Morris, IA-64 Architecture Disclosures White Paper, Tech. Rep., HP/Intel, 1999.
[14] S.B. Furber, ARM System Architecture, Addison-Wesley Longman Publishing, 1996.
[15] R.A. Starke, A. Carminati, R.S. de Oliveira, Investigating a four-issue deterministic VLIW architecture for real-time systems, in: 2015 IEEE 13th International Conference on Industrial Informatics (INDIN), 2015, pp. 215–220. http://dx.doi.org/10.1109/INDIN.2015.7281737.
[16] R.A. Starke, A. Carminati, R.S.D. Oliveira, Evaluating the design of a VLIW processor for real-time systems, ACM Trans. Embed. Comput. Syst. 15 (3) (2016) 1–26, http://dx.doi.org/10.1145/2889490.
[17] J.A. Fisher, P. Faraboschi, C. Young, Embedded Computing: A VLIW Approach to Architecture, Compilers and Tools, Elsevier, 2005.
[18] C. Lattner, V. Adve, LLVM: a compilation framework for lifelong program analysis & transformation, in: Proceedings of the International Symposium on Code Generation and Optimization (CGO'04), IEEE Computer Society, 2004, pp. 75–86, http://dx.doi.org/10.1109/CGO.2004.1281665.
[19] P. Puschner, D. Prokesch, B. Huber, J. Knoop, S. Hepp, G. Gebhard, The T-CREST approach of compiler and WCET-analysis integration, in: Proceedings of the 16th IEEE International Symposium on Object/Component/Service-oriented Real-time Distributed Computing (ISORC 2013), 2013, pp. 1–8. http://dx.doi.org/10.1109/ISORC.2013.6913220.
[20] J. Gustafsson, A. Betts, The Mälardalen WCET benchmarks: past, present and future, in: Proceedings of the 10th International Workshop on Worst-Case Execution Time Analysis (WCET'10), Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2010, pp. 136–146. http://dx.doi.org/10.4230/OASIcs.WCET.2010.136.