Worst-Case Execution Time Analysis of Predicated Architectures*

Florian Brandner[1] and Amine Naji[2]

1 **LTCI, Télécom ParisTech, Université Paris-Saclay, Paris, France**
`florian.brandner@telecom-paristech.fr`

2 **U2IS, ENSTA ParisTech, Université Paris-Saclay, Palaiseau, France**
`amine.naji@ensta-paristech.fr`

Abstract

The time-predictable design of computer architectures for the use in (hard) real-time systems is becoming more and more important, due to the increasing complexity of modern computer architectures. The design of predictable processor pipelines recently received considerable attention. The goal here is to find a trade-off between predictability and computing power.

Branches and jumps are particularly problematic for high-performance processors. For one, branches are executed late in the pipeline. This either leads to high branch penalties (flushing) or complex software/hardware techniques (branch predictors). Another side-effect of branches is that they make it difficult to exploit instruction-level parallelism due to control dependencies.

Predicated computer architectures allow to attach a *predicate* to the instructions in a program. An instruction is then only executed when the predicate evaluates to true and otherwise behaves like a simple `nop` instruction. Predicates can thus be used to convert control dependencies into data dependencies, which helps to address both of the aforementioned problems.

A downside of predicated instructions is the precise worst-case execution time (WCET) analysis of programs making use of them. Predicated memory accesses, for instance, may or may not have an impact on the processor's cache and thus need to be considered by the cache analysis. Predication potentially has an impact on all analysis phases of a WCET analysis tool. We thus explore a preprocessing step that explicitly unfolds the control-flow graph, which allows us to apply standard analyses that are themselves not aware of predication.

1998 ACM Subject Classification C.3 [Special-Purpose and Application-Based Systems] Real-Time and Embedded Systems

Keywords and phrases Predication, Worst-Case Execution Time Analysis, Real-Time Systems

Digital Object Identifier 10.4230/OASIcs.WCET.2017.6

1 Introduction

Embedded real-time systems, as most computer systems, faced a steady increase in requirements [4] during more than three decades. An increase in requirements frequently also translates into an increased performance need. Simple and predictable micro-controllers cannot satisfy these needs in many cases. To address this issue new architecture designs have recently been explored that promise a high degree of (time-)predictability while offering an acceptable performance level [20, 16, 23, 18]. Due to its high-latency, designs focusing on the memory hierarchy have been explored extensively [17, 13, 19, 2]. Also the design

* This work was supported by a grant (2014-0741D) from Digiteo France: "Profiling Metrics and Techniques for the Optimization of Real-Time Programs" (PM-TOP).

17th International Workshop on Worst-Case Execution Time Analysis (WCET 2017).
Editor: Jan Reineke; Article No. 6; pp. 6:1–6:13
Open Access Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

of processor pipelines has recently received considerable attention [3, 20, 16, 24]. The goal behind all this work is to find a trade-off between time-predictability and computing power.

The handling of branches and jumps is particularly problematic for the design of time-predictable high-performance processors. The new value of the program counter (`PC`) can usually only be computed late in the processor pipeline.[1] As the branch advances through the pipeline the processor is unable to tell which instructions need to be fetched next. A simple solution is to simply fetch the instructions immediately after the branch. These instructions are typically *flushed* from the pipeline, i.e., discarded, once the actual value of the `PC` is available. This solution induces a considerable branch penalty. *Branch prediction* techniques may be used to guess the branch direction and/or address earlier. Static techniques rely on the compiler to provide good predictions, while dynamic predictors require additional hardware. Both approaches allow to reduce the branch penalty. However, the state of hardware-based branch predictors needs to be taken into consideration during *Worst-Case Execution Time* (WCET) analysis. Conflicts between branches of the program itself and interference from other tasks further complicate the analysis. An alternative solution is to let the instructions following a branch execute. The instructions in these *branch delay slots* might then perform useful work – if the compiler is able to rearrange the instructions accordingly. This approach has drawbacks, despite being predictable: (a) unused branch delay slots have to be filled with `nop` instructions and thus increase code size and (b) tools, including the compiler and WCET analyzer, have to be aware of the branch delay slots.

Branches, furthermore, introduce control dependencies, i.e., instructions *after* a branch can only be executed safely when the branch direction/address has been determined. This even applies when branch prediction is used: the processor may only execute instructions *speculatively* as long as the instructions do not cause any side-effects.[2] Inversely, instructions *before* the branch can usually not be moved to locations after the branch. Branches consequently make it more difficult to exploit *instruction-level parallelism* [7].

One solution to these issues is *predication*, where an additional predicate is attached to instructions. The predicate is evaluated at runtime and allows to conditionally nullify the effect of the instruction, i.e., the instruction is discarded and behaves like a simple `nop` instruction when the predicate evaluates to false. The aforementioned problems can be addressed using predication by moving instructions past branches. The control dependence, with regard to the original branch instruction, is then effectively transformed into a data dependence on the predicate. This, for instance, simplifies the compiler's task to fill branch delay slots or to exploit instruction-level parallelism. In some cases branches can even be eliminated completely. This is particularly advantageous for short code sequences, where the branch penalty often outweighs the cost of executing a few predicated instructions. Some real-time systems actually take advantage of the possibility to entirely eliminate all conditional branches. This is known as the single-path programming paradigm [15].

Despite these advantages, predicated instructions can be problematic during WCET analysis. Side-effects of predicated instructions need to be analyzed, which depend on the runtime value of the instruction's predicate. This may have an impact on many analysis steps, including value range analysis, loop bounds analysis, infeasible path analysis, but also the cache and pipeline analyses. Predicated memory accesses, for instance, may or may not have an impact on the processor's cache, depending on the predicate. The simplest solution

[1] After reading register values at the level where regular arithmetic operations are often handled.

[2] Visible according to the processor's programming conventions, which usually does not include hidden states (caches, branch predictors, ...).

for the analysis would be to ignore predicates and conservatively consider the effect of both cases. This may result in very conservative results, since the implicit information available in the program's original control flow before the elimination of branches is entirely lost. The analysis could also be extended to be aware of predicates. Note, however, that this may require changes to virtually all analysis steps in a WCET analyzer and may thus require a considerable engineering effort. We thus explore a much simpler solution that consists in recovering the (hidden) control flow. Instructions that *define* (set) a predicate are handled similar to branch instructions and lead to a control-flow split. The succeeding instructions are then *duplicated*, once assuming that the predicate evaluates to `true` and once assuming that the predicate value is `false`. Subsequent instructions that are predicated with that predicate are now trivial to handle: an instruction either always corresponds to a `nop` or always corresponds to the regular unpredicated instruction.

The remainder of this paper is structured as follows. We will first give some background regarding the Patmos platform on which our work builds, covering the architecture design as well as the tool suite (compiler, analyzer). Section 4 then describes a simple algorithm that allows us to recover the hidden control flow of predicated code. We then present the results from preliminary experiments in Section 5. Related work, concerning predication in real-time systems and related analysis techniques, is finally discussed in Section 6 before concluding.

2 Background

Patmos Architecture

The Patmos architecture [20] is intended as a test bed to evaluate and design new time-predictable computer architecture concepts, covering cache designs [19, 2], on-chip networks [12], as well as instruction set architecture design. Among many other features, Patmos supports predicated execution. An additional predicate operand is attached to each instruction, which allows to refer to one out of the 8 predicate registers (`p0` through `p7`). The predicate operand, in addition, allows to invert the predicate value (e.g., `!p0`). Consequently, 4 bits of the instruction encoding are reserved for the predicate operand (3 bits for the predicate register, 1 bit for negation). The predicate registers themselves consist of a single bit each. Predicate register `p0` always evaluates to `true` and cannot be overwritten.

Predicate registers can be defined using dedicated comparison instructions (e.g., `cmpeq`), which allow to compare 32-bit integer values from general-purpose register operands and immediate values. These instructions allow to specify a destination predicate, which sets the predicate register accordingly. In addition, basic logical operations can be performed on two predicate register operands using dedicated logical-predicate instructions (e.g., `por`). The result of the operation is again written into a predicate register. Note that these instructions can be predicated themselves, which facilitates the handling of nested `if` statements.

Patmos is a *Very Long Instruction Word* (VLIW) architecture that can issue and execute multiple instructions in parallel in a five-stage, in-order pipeline: fetch (`FE`), decode (`DEC`), execute (`EX`), memory access (`MEM`), and register writeback (`WB`). Instructions are fetched from a special instruction cache, the so-called method cache [2], which guarantees that an instruction fetch always hits in the cache. Method cache misses may only occur in the `EX` stage during the execution of dedicated branch instructions (e.g., `brcf`) or function calls (e.g., `call`, `ret`). Several variations of branch and call instructions exist, having (a) no branch delay slots (e.g., `callnd`, `brcfnd`), (b) 2 branch delay slots (e.g., `br`), and (c) 3 branch delay slots (e.g., `call`, `brcf`). Similarly, misses in the data cache may only occur in the `MEM` stage. The `FE` and `DEC` pipeline stages are thus free from undesirable side-effects (apart from updating the PC), while the EX, MEM, and WB stages may cause side-effects on the processor's registers or caches. Predicate registers are read and written in the execute stage (EX). The processor thus is able to detect whether a predicated instruction needs to be nullified before any undesirable side-effects may become visible.

Compiler

The Patmos toolchain is based on the LLVM compiler framework,[3] which compiles all source files to an internal bitcode representation. In our case, this also applies to user- and system libraries, which are also linked together at the bitcode level. The final machine-code generation is postponed to the very end of the compilation process when *all* bitcode is available [1]. The code generator thus has a complete view of the entire program, which can be optimized and analyzed before the final executable file is generated.

The LLVM code generator is able to produce predicated code at several stages. Firstly, the instruction selector is able to recognize simple *select statements* that are directly compiled to conditional moves. A generic if-conversion optimization is also available, which allows to eliminate conditional branches of complex control-flow and produce predicated code. Finally, patmos-specific code transformations are available to generate single-path programs [8].

The *Application Binary Interface* (ABI) defined by Patmos states that all predicate registers are callee saved. This means that, during a function call, the called function needs to save and restore any predicate register that it modifies. Predicate registers cannot be used to pass arguments to other functions. Predicates are, in terms of the ABI, strictly function local. Consequently, predicates can be freely used across function calls, while called functions are independent from predicates computed before entering the function. It is still possible that call instructions themselves are predicated, i.e., the function is called conditionally.

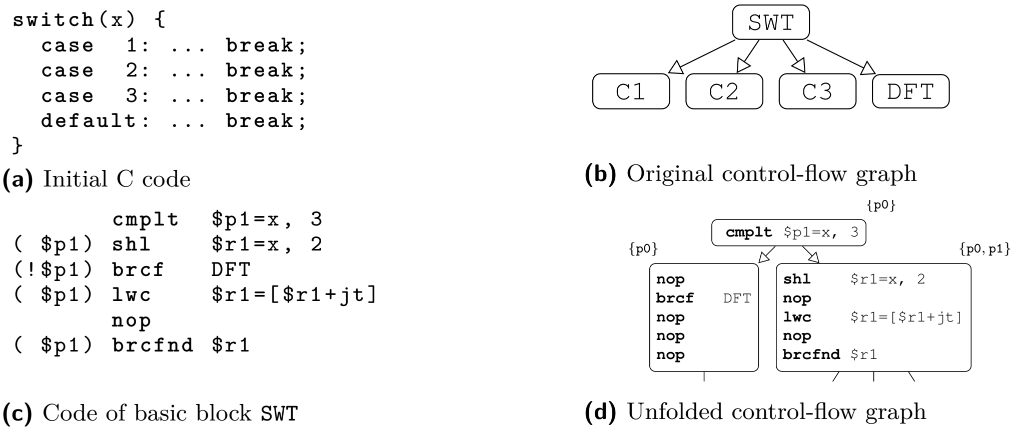
WCET Analyses

As mentioned above, all code of the final program is available within the compiler. This enables us to immediately perform machine-code-level analyses covering the entire program (and all system libraries) before the final code emission. Several such analyses [19] are available *in the compiler* whose results can be exported to other WCET analyzers or used within the compiler itself to perform further analyses or optimizations [10]. The internal analyses either operate directly on the intermediate representation of the LLVM code generator or an enriched inter-procedural representation that allows to easily attach various analysis results to the code generator's intermediate representation in a context-dependent manner.

3 Motivating Example

Before giving a formal description of our proposed approach, we will give a simple motivating example. Figure 1 illustrates the implementation of a `switch` statement using a jump table and predication. The original C code is shown in Subfigure 1a and the resulting *Control-Flow Graph* (CFG) from LLVM in Subfigure 1b. Basic blocks C1 through C3 represent the three `case` statements, while basic block DFT corresponds to the `default` statement. The code of the `switch` statement itself can be found in the SWT block, whose machine code is shown in Subfigure 1c.

[3] http://www.llvm.org



```
switch(x) {
  case  1: ... break;
  case  2: ... break;
  case  3: ... break;
  default: ... break;
}
```

(a) Initial C code

(b) Original control-flow graph

```
        cmplt   $p1=x, 3
( $p1)  shl     $r1=x, 2
(!$p1)  brcf    DFT
( $p1)  lwc     $r1=[$r1+jt]
        nop
( $p1)  brcfnd  $r1
```

(c) Code of basic block SWT

```
cmplt $p1=x, 3
```

```
nop
brcf   DFT
nop
nop
nop
```

```
shl     $r1=x, 2
nop
lwc     $r1=[$r1+jt]
nop
brcfnd  $r1
```

(d) Unfolded control-flow graph

Figure 1 Implementation of a simple `switch` statement using a jump table, the corresponding control-flow graph, the predicated machine code of basic block `SWT`, and the unfolded control flow.

The machine code uses a jump table (`jt`) that is implemented as an array holding the addresses of basic blocks `C1` through `C3`. After verifying that the value of variable `x` is within the array bounds (`cmplt`), the address of the destination block is loaded (`lwc`) and control is transferred (after a 1 cycle load delay slot) via an indirect branch (`brcfnd`). If `x`'s value exceeds the array bounds, a conditional branch (`brcf`) immediately transfers control to basic block `DFT`. Note that this branch instruction has 3 branch delay slots and that one of these slots even contains another branch instruction.

The predicated code may pose several challenges in a WCET analyzer. One particular challenge is the reconstruction of the program's CFG from the binary machine code, which usually represents the input to most WCET analysis tools. The compiler placed a branch instruction in one of the branch delay slots of another branch. In this example it is trivial to detect that the predicates of the respective branches are disjoint. However, different predicates might be used, which makes it difficult to reconstruct the actual control flow from such code. In our case this is not necessary, since the analysis is part of the compiler and thus has direct access to its intermediate representation.[4]

Another challenge, as noted before, are potential side-effects on caches or registers caused by predicated instructions. This issue is resolved by unfolding the hidden control flow from the predicated code – as depicted by Subfigure 1d. Each time when a predicate register is defined (`cmplt`) a control-flow split is performed at the level of the control-flow graph. The instruction defining the predicate is then treated in a similar way as conditional branches and subsequent code is duplicated considering both of the potential predicate values (`true` or `false`). The basic block on the left side of the subfigure here corresponds to an execution where predicate register `p1` evaluates to `false`, while the basic block on the right is executed only when `p1` evaluates to `true`. In fact, each basic block is associated with a set of predicates that are known to be `true` when entering the basic block (indicated in the top corner of each block). Inversely, predicates that do not appear in this set are known to be `false`.[5] Note that the predicates in the code are no longer needed. Predicated instructions are either duplicated *unconditionally* or are otherwise replaced by an explicit `nop`.

[4] Note that this also solves many unrelated issues during the control-flow reconstruction from binary code such as computed branch targets, function pointers, et cetera.

[5] This is safe, since LLVM inserts pseudo definitions on all program paths where a register is not defined.

In the unfolded control-flow graph it is now much easier to analyze the instructions' side-effects. The load (`lwc`) from the jump table, for instance, is only executed when the variable `x` is known to be less than 3 (`cmplt`). This means that any side-effects of this instruction on the data cache are only visible in basic blocks `C1` through `C3`, but not in basic block `DFT`. Another implicit side-effect concerns the value of variable `x` after the comparison. Due to the control-flow split at the `cmplt` instruction, it is very easy for a value range analysis to show that the value of `x` has to be larger then 3 when reaching basic block `DFT`. Delayed branches and potential redefinitions of register operands make this much more challenging in the original CFG. The algorithm to construct such an unfolded CFG, while considering predication and delayed branches, is discussed in the next section.

4 Control-Flow Unfolding

Due to space considerations, Algorithm 1 only shows a simplified version of our approach. The presented algorithm assumes a single issue architecture, which avoids the need to handle several parallel uses and (re-)definitions of predicates, parallel branches and predicate operations, et cetera. The algorithm also assumes that the predicates of branches that appear in branch delay slots are disjoint, i.e., only a single branch is known to be taken at any moment at runtime. Lastly, the presented approach only operates on the CFG of a single function. Extensions to the algorithm, included in the actual implementation, which allow us to handle these cases are briefly highlighted later. Finally, the algorithm invokes several helper functions whose code is not shown. We will briefly define these functions in an informal way before discussing the algorithm in detail.

4.1 Helper Functions

Several helper functions are needed in order to operate on individual instructions in LLVM's intermediate representation. The function NEXT allows to obtain the instruction immediately following an instruction i in its parent basic block, PKILL returns the set of predicate registers whose live ranges end after instruction i, while the functions PDEF and ISPREDDEF allow to obtain/test whether an instruction defines a predicate register. The function ISNOP is used to test whether an instruction i is nullified given the current set of predicates P.

Several helper functions are related to branches, allowing to test for branch instructions (ISBRANCH), obtain the number of the branche's delay slots (BRANCHDELAY), and obtain the successor basic blocks to which control may be transferred by a branch (BRANCHTARGETS). FALLTHROUGHTARGET is used to obtain the fall-trough target basic block of the last instruction of a basic block, i.e., control is transferred to another basic block without an explicit jump or branch instruction.

Finally, three functions are related to the construction of the enriched intermediate representation of our analysis tool. GETCFNODE allows to obtain the control-flow node associated with a start instruction f, the remaining number of delay slots d, and a set of predicates P – if such a control-flow node was created before. Nodes are created using the function MAKECFNODE, which *duplicates* all instructions between the instruction f and e provided as arguments. The new node is also associated with d, the number of branch delay slots remaining, and P, the set of predicates. Finally, MAKECFEDGE creates control-flow edges between two control-flow nodes provided as arguments.

Algorithm 1 Simplified algorithm to recover the hidden control flow from predicated code by code duplication on a single-issue architecture.

```
 1: function UnFold(MachInstr $f$, MachInstr $l$, Pred $d$, BasicBlockSet $T$, PredSet $P$)
 2:     if $n$ =GetCFNode(f, d, P) then return $n$        ▷ Check if control-flow node exists
 3:     PredSet $L = P$; Pred $pd = \texttt{p0}$; MachInstr $e = f$        ▷ Initialize variables
 4:     for each instruction $i$ between $l$ and $f$ do
 5:         $L = L \setminus \text{PKill}(i)$        ▷ Remove dead predicates
 6:         $e = i$        ▷ Track end of control-flow node
 7:         if $\neg\text{IsNop}(i, P)$ then        ▷ Skip nop instructions
 8:             if IsPredDef(i) then        ▷ Predicate definition
 9:                 $pd = \text{PDef}(i)$        ▷ Track defined predicate
10:                 break        ▷ Immediately split control flow
11:             else if IsBranch(i) then        ▷ Branch instruction
12:                 $d = \text{BranchDelay}(i)$        ▷ Track branch delay slots
13:                 $T = \text{BranchTargets}(i)$        ▷ Track branch target(s)
14:         if $d = 0$ then break        ▷ Split control flow after branch delay
15:         $d = d - 1$        ▷ Update remaining branch delay slots
16:     CFNode $n = \text{MakeCFNode}(f, e, d, P)$        ▷ Create a new control-flow node
17:     if $e = l \wedge T = \emptyset$ then        ▷ Handle fall-through
18:         $T = \text{FallThroughTarget}(l)$
19:     else if $d \neq 0 \wedge pd \neq \texttt{p0}$ then        ▷ Handle split due to predicate definition
20:         for each $P' \in \{L \cup pd, (L \setminus pd) \cup \{\texttt{p0}\}\}$ do        ▷ Compute successor predicates
21:             CFNode $n' = \text{UnFold}(\text{Next}(e), l, d, T, P')$
22:             $\text{MakeCFEdge}(n, n')$
23:         return $n$
24:     for each $s \in T$ do        ▷ Create successor control-flow nodes
25:         for each $P' \in \{L \cup pd, (L \setminus pd) \cup \{\texttt{p0}\}\}$ do        ▷ Compute successor predicates
26:             Let $f'$, $l'$ be the first/last instruction of $s$ in
27:                 CFNode $n' = \text{UnFold}(f', l', \infty, \emptyset, P')$
28:                 $\text{MakeCFEdge}(n, n')$
29:     return $n$
30: procedure UnFoldCFG($G$)
31:     Let $f$, $l$ be the first/last instruction of the entry block of $G$ in UnFold($l, f, \infty, \emptyset, \{\texttt{p0}\}$)
```

4.2 Discussion of the Algorithm

Algorithm 1 consists of two functions: the algorithm's main function UnFoldCFG and the recursive function UnFold, which actually constructs the unfolded CFG. The latter function's parameters f (first) and l (last) represent machine instructions that need to be unfolded next. The integer argument d (delay) is needed to track branch delay slots across control-flow node boundaries. The argument T (targets), likewise, is used to track the set of potential branch targets during the handling of branch delay slots. The function's last argument P (predicates) represents the set of active predicates known to be `true`.

The algorithm starts off by processing the CFG of a function provided by LLVM (l. 31), considering the instructions at the function's entry point, which are not in a branch delay slot ($d = \infty$) and not executed under any specific predicate condition ($P = \{\texttt{p0}\}$). This triggers the recursive processing of all instructions in the CFG provided by LLVM and the construction of the unfolded CFG. The actual unfolding then proceeds in two steps.

First, all the instructions between the arguments l and f of function UnFold are analyzed (l. 4 – 15) in order to find locations where the control-flow needs to be split. A split may be necessary due to one of the following reasons: (a) the end of the original basic block of LLVM is reached (fall-through), (b) a branch effectively transfers control to another basic block after its branch delay slots, or (c) a predicate definition is encountered.

Each instruction is analyzed in turn. Instructions that are nullified under the current predicate set P (IsNop, l. 7) are ignored. Two instruction classes need special attention, since they may cause a control-flow split: instructions that define a predicate (IsPredDef) and branches (IsBranch). Predicate definitions are handled similar to branches in traditional CFGs and immediately lead to a control-flow split (**break**), remembering the current end location (e) and the newly defined predicate (pd). Branches, on the other hand, may cause a delayed control-flow split (BranchDelay). The variable d tracks the number of remaining branch delay slots (the variable is initialized to ∞ if the analyzed code is not in a branch delays slot). Once this counter reaches 0 the actual control-flow split occurs (l .14). Note that predicate definitions may also appear in branch delay slots. In this case the variable d is passed as an argument to subsequent recursive calls to the function UnFold. Since a branch was encountered, the branch targets also need to be remembered and potentially passed on the recursive calls using variable T. If no control-flow split is encountered by the analysis, i.e., no instruction defines a predicate or branches, the **for** loop terminates normally. This only happens for basic blocks with a fall-through.

The set of live predicates (L) is tracked additionally, while instructions are processed. This set is initialized with the incoming argument P (l. 3) and updated whenever the live range of a predicate ends (l. 5). Note that a location, where the live range of a predicate ends, could be exploited to rejoin the control flow. The algorithm does not take advantage of this and essentially *extends* the live ranges of predicates up to a control-flow split.

The second step of the algorithm (l. 17 – 29) is concerned with the actual construction of the unfolded CFG. After leaving the **for** loop, a new control-flow node is created representing the instructions between f and e that are executed under the predicate set P (l. 16). It remains then to discover and unfold the successor control-flow nodes, depending on the nature of the control-flow split determined during the first step.

Fall-throughs (case a from above) and completed branches (case b) always transfer control to another basic block in LLVM's CFG. The main difference is that no branch target is known for fall-throughs (case a), which can simply be obtained using the function FallThroughTarget (l. 18). The remainder of the processing is identical (l. 24 – 29). Each branch target is analyzed through a recursive call to UnFold, considering the new set of active predicates P' as well as the branch target's first/last instruction. Note that a predicate definition (case c) may coincide with cases (a) and (b). The active predicates are thus computed from the live predicates L and the newly defined predicate pd (l. 25). The targets are then visited by the algorithm with the predicate `true` ($L \cup pd$) and `false` ($L \setminus pd$). Note that `p0` always remains `true` and consequently needs to be readded.

The remaining control-flow splits, that are not covered by the previous paragraph, are due to predicate definitions (case c), which may either occur in the middle of basic blocks or in a branch delay slot. Both situations require a slightly different handling (l. 19 – 23), since control is *not* (yet) transferred to another basic block of LLVM's CFG. The set of active predicates P' is computed as in the regular case. However, the remaining instructions

(Next, l. 21) of LLVM's current basic block need to be analyzed after the control-flow split, while remembering potentially ongoing branches. This is accomplished by passing the values of d and T to the recursive invocation of UnFold. This allows the recursive invocation to correctly track the branch targets and the number of branch delay slots, i.e., if the split actually occurred in a branch delay slot.

Extensions

The presented algorithm is a somewhat simplified version of the actual implementation. Most notably, the Patmos processor can fetch and issue multiple operations in parallel using instruction bundles. This means that corner cases may arise that need to be considered. For instance, predicate definitions and branches can be combined into the same bundle. The implemented algorithm considers function calls and returns, whose branch delay slots also need to be accounted for. The handling of function calls was omitted for brevity. These extensions slightly complicate the algorithm, but do not impact its overall structure.

Another, more involved extension, is the handling of branches nested within other branch delay slots. The presented algorithm is only correct as long as the predicates of the nested branches are disjoint. This can be handled by replacing the arguments d and T of the function UnFold by a stack data structure. This allows to track all executing branches at the same time, detect the completion of a branch, and split the control flow accordingly.

Complexity

The UnFold function essentially performs a depth first search on the CFG provided by LLVM. Each instruction is processed once for every set of potentially active predicates, whose number can be bounded by 128 (2^7). Note that `p0` is always `true` and thus cannot impact this bound. The algorithm thus is linear in the number of instructions and control-flow edges.

5 Preliminary Experiments

The following section presents the results from preliminary experiments measuring the overhead induced by unfolding. The implementation is part of the analysis framework of the Patmos compiler, which is based on LLVM 3.5. The unfolded CFG is merely used for analysis purposes and essentially represents an additional annotation layer on top of the data structures of LLVM. The binary code of the analyzed programs is thus not modified. The extended version of the previously described algorithm was applied to a subset of the TACLe benchmarks [6], i.e., those adopted from the MiBench suite. The programs were compiled with optimizations enabled (`-O2`), while varying the issue-width (single-issue vs. VLIW) and the compiler's handling of branch delay slots (non-delayed only, delayed only, mixed). This results in 6 configurations overall. The size of the unfolded CFGs for each of these configurations is compared against the original instruction count in LLVM's CFG.

Figure 2 shows the normalized increase in the number of instructions for the three configurations with VLIW instruction bundles. As can be seen, the overhead induced by unfolding is usually low, ranging between 10% and 20%. The `susan` benchmark is the only exception, showing an increase between 43% and 45%. The if-conversion optimization is particularly effective for this benchmark, covering larger regions and producing more complex predicates. Note that the observed overhead does not come as a surprise. It is well known that the share of conditional branches in typical programs roughly falls into a similar range as the observed overhead. This indicates that, overall, only a few instructions are duplicated

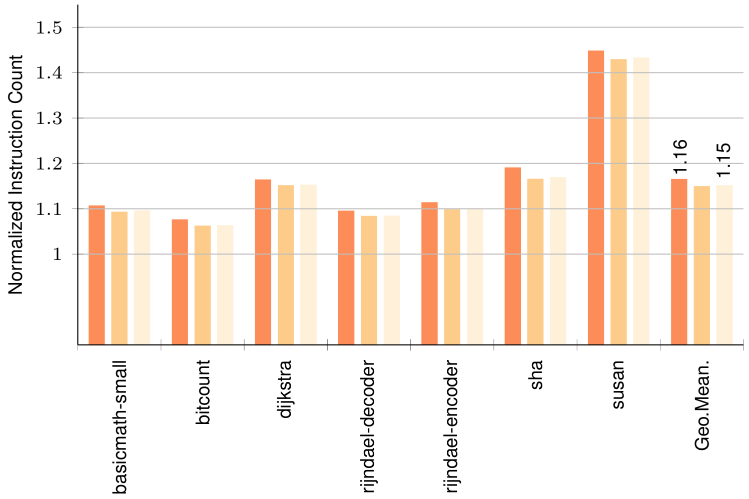


Figure 2 Increase in the number of instructions due to unfolding for the delayed (■), mixed (■), and non-delayed (□) configurations with VLIW instruction bundles, normalized to the size of LLVM's original CFG (lower is better).

by the unfolding algorithm for each computed *condition* – despite the fact that the algorithm artificially *extends* the live ranges of predicates. The configurations for the single-issue processor follow a similar trend (not shown for brevity).

The runtime overhead of the proposed algorithm is negligible and amounts to $0.1s$ on average, which represents 0.9% of the code generation time (excluding other WCET analysis steps). Also note that the unfolded CFG allowed us to improve other analyses. The value range analysis, for instance, is able to take advantage of control-flow splits at predicate definitions as explained in Section 3.

6 Related Work

Predicated computer architectures received considerable attention in the 1990s with the development of VLIW and EPIC architectures that tried to exploit instruction-level parallelism through static compilation techniques rather than hardware [7].

Various compiler optimizations have been developed targeting the transformation of regular code into predicated code [14, 25] and the optimization of predicated code [5, 22]. A common problem for these optimizations is the need to understand the relations between predicates [11, 21], i.e., which predicates can be live at the same time. The underlying machine code may evolve through optimizations in the compiler, which might require these analyses to be performed multiple times. The analyses thus need to be fast and only reason about predicate relations that can be deduced from the *structural* relations between predicates. Information on the actual conditions, e.g., the tested values, are not captured. The work is somewhat orthogonal to our approach and might help to reduce some of the overhead induced by useless code duplications. The techniques are, in addition, concerned with the analysis of the predicates themselves and do not allow to obtain other analysis results.

Hu [9] addressed this issue by refining the semantics of predicated code and redefining several typical concepts used in compilers/static analyzers (e.g., dominance and data dependencies). She also showed how predicate-aware data-flow analysis can be realized using the example of reaching definitions. Similar techniques could be applied to many other analysis techniques, including those used in typical WCET analyzers. However, this would require a considerable engineering effort in order to adapt all existing analyes accordingly.

The single-path programming paradigm, which can often be found in the context of real-time systems, takes predication to the extreme: (almost) all control-flow is eliminated from the program and replaced by predicated code [15]. The idea is to avoid WCET analysis altogether and instead generate code that exhibits the same execution time under *all* execution conditions. Geyer et al. [8] propose a code generator that is able to produce single-path code from a given input program. The proposed approach heavily relies on predication, which generally leads to constant timing. However, some timing variations can still be encountered due to variations of the program inputs and due to memory accesses to the data cache.

Starke et al. [24] propose a lightweight approach for predicated execution for their time-predictable VLIW. The proposed approach consist of only a single predicate register, which limits the compiler's ability to handle complex predicate expressions and may also impact the attainable instruction-level parallelism. The results indicate that predicated execution is mostly beneficial with regard to the WCET. However, the authors do not describe how the WCET analysis handles predicated operations. The technique proposed in the previous section might allow to further improve the estimation of the WCET for their processor.

7 Conclusion

In this work a lightweight approach to the handling of predicated code in WCET analyzers was presented. Predicate definitions are treated similar to conventional branches and immediately lead to a control-flow split. Subsequent instructions are then analyzed twice, once assuming that the predicate evaluates to `true` and once assuming it evaluates to `false`. The hidden control flow in predicated code is recovered and explicitly represented in an unfolded CFG. The presented algorithm is able to perform the desired control-flow unfolding and keep track of branch delay slots for a simplified single-issue architecture. The actual implementation is able to handle parallel instruction bundles, function calls, and nested delayed branches. Our preliminary evaluation shows that the unfolding does not result in excessive code duplication and yields a moderate code size increase of about 16% on average.

References

1 F. Brandner, S. Hepp, and D. Prokesch. D5.2 – Initial compiler version, 2012. Report of T-CREST Deliverable D5.2, `http://www.t-crest.org/page/results`.

2 P. Degasperi, S. Hepp, W. Puffitsch, and M. Schoeberl. A method cache for Patmos. In *Proc. of the Symposium on Object/Component/Service-oriented Real-time Distributed Computing*. IEEE, 2014.

3 M. Delvai, W. Huber, P. Puschner, and A. Steininger. Processor support for temporal predictability – the SPEAR design example. In *Proc. of the Euromicro Conference on Real-Time Systems*, pages 169–176. IEEE, 2003. `doi:10.1109/EMRTS.2003.1212740`.

4 D. L. Dvorak. NASA study on flight software complexity, 2009. NASA Office of Chief Engineer, Technical Excellence Initiative.

5 A. E. Eichenberger and E. S. Davidson. Register allocation for predicated code. In *Proc. of the Int'l Symposium on Microarchitecture*, pages 180–191. IEEE, 1995.

6 H. Falk, S. Altmeyer, P. Hellinckx, B. Lisper, W. Puffitsch, C. Rochange, M. Schoeberl, R. B. Sørensen, P. Wägemann, and S. Wegener. TACLeBench: A Benchmark Collection to Support Worst-Case Execution Time Research. In *Proc. of the Int'l Workshop on Worst-Case Execution Time Analysis*, volume 55 of *OASIcs*, pages 1–10. Schloss Dagstuhl, 2016. `doi:10.4230/OASIcs.WCET.2016.2`.

7 J. A. Fisher, P. Faraboschi, and Y. Cliff. *Embedded Computing: A VLIW Approach to Architecture, Compilers and Tools.* Morgan Kaufmann (Elsevier), 2005.

8 C. B. Geyer, B. Huber, D. Prokesch, and P. Puschner. Time-predictable code execution – instruction-set support for the single-path approach. In *Proc. of the Int'l Symposium on Object/component/service-oriented Real-time distributed Computing*, pages 1–8, 2013. doi:10.1109/ISORC.2013.6913195.

9 P. Hu. Static analysis for guarded code. In *Proc. of the Int'l Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*, pages 44–56. Springer, 2000.

10 B. Huber, D. Prokesch, and P. Puschner. Combined WCET analysis of bitcode and machine code using control-flow relation graphs. In *Proc. of the Conference on Languages, Compilers and Tools for Embedded Systems*, pages 163–172. ACM, 2013. doi:10.1145/2465554.2465567.

11 R. Johnson and M. Schlansker. Analysis techniques for predicated code. In *Proc. of the Int'l Symposium on Microarchitecture*, pages 100–113. IEEE, 1996.

12 E. Kasapaki, M. Schoeberl, R. B. Sørensen, C. Müller, K. Goossens, and J. Sparsø. Argo: A real-time network-on-chip architecture with an efficient GALS implementation. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 24(2):479–492, Feb 2016. doi:10.1109/TVLSI.2015.2405614.

13 S. Metzlaff, I. Guliashvili, S. Uhrig, and T. Ungerer. A dynamic instruction scratchpad memory for embedded processors managed by hardware. In *Proc. of the Architecture of Computing Systems Conference*, pages 122–134. Springer, 2011. doi:10.1007/978-3-642-19137-4_11.

14 J. C. H. Park and M. Schlansker. On predicated execution. Technical report HPL-91-58, HP Laboratories, 1991.

15 P. Puschner. Transforming execution-time boundable code into temporally predictable code. In *Proc. of the IFIP World Computer Congress*, pages 163–172. Kluwer, 2002.

16 J. Reineke. A PRET microarchitecture implementation with repeatable timing and competitive performance. In *Proc. of the Int'l Conference on Computer Design*, pages 87–93. IEEE, 2012. doi:10.1109/ICCD.2012.6378622.

17 J. Reineke, I. Liu, H. D. Patel, S. Kim, and E. A. Lee. PRET DRAM controller: Bank privatization for predictability and temporal isolation. In *Proc. of the Conference on Hardware/Software Codesign and System Synthesis*, pages 99–108, 2011.

18 C. Rochange, S. Uhrig, and P. Sainrat. *Time-Predictable Architectures.* ISTE Wiley, 2014. doi:10.1002/9781118790229.

19 S.Abbaspour, A. Jordan, and F. Brandner. Lazy spilling for a time-predictable stack cache: Implementation and analysis. In *Proc. of the Workshop on Worst-Case Execution Time Analysis*, volume 39 of *OASIcs*, pages 83–92. Schloss Dagstuhl, 2014. doi:10.4230/OASIcs.WCET.2014.83.

20 M. Schoeberl, P. Schleuniger, W. Puffitsch, F. Brandner, and C. W. Probst. Towards a time-predictable dual-issue microprocessor: The Patmos approach. In *Bringing Theory to Practice: Predictability and Performance in Embedded Systems*, volume 18 of *OASIcs*, pages 11–21. Schloss Dagstuhl, 2011. doi:10.4230/OASIcs.PPES.2011.11.

21 J. W. Sias, W.-M. W. Hwu, and D. I. August. Accurate and efficient predicate analysis with binary decision diagrams. In *Proc. of the Int'l Symposium on Microarchitecture*, pages 112–123. ACM, 2000. doi:10.1145/360128.360141.

22 M. Smelyanskiy, S. A. Mahlke, E. S. Davidson, and H.-H. S. Lee. Predicate-aware scheduling: A technique for reducing resource constraints. In *Proc. of the Int'l Symposium on Code Generation and Optimization*, pages 169–178. IEEE, 2003.

23 R. A. Starke, A. Carminati, and R. S. De Oliveira. Evaluating the design of a VLIW processor for real-time systems. *ACM Trans. Embed. Comput. Syst.*, 15(3):46:1–46:26, 2016. doi:10.1145/2889490.

24 R. A. Starke, A. Carminati, and R. S. de Oliveira. Evaluation of a low overhead predication system for a deterministic VLIW architecture targeting real-time applications. *Microprocessors and Microsystems*, 49:1–8, 2017. doi:http://doi.org/10.1016/j.micpro.2016.11.017.

25 A. Stoutchinin and G. Gao. If-conversion in SSA form. In *Proc. of the Int'l Euro-Par Conference*, pages 336–345. Springer, 2004. doi:10.1007/978-3-540-27866-5_43.