# Towards Automated Generation of Time-Predictable Code*

**Daniel Prokesch, Benedikt Huber, and Peter Puschner**

**Institute of Computer Engineering**
**Vienna University of Technology, Austria**
**{daniel,benedikt,peter}@vmars.tuwien.ac.at**

## Abstract

Knowledge of the worst-case execution time of software components is essential in safety-critical hard real-time systems. The analysis thereof is not trivial as the execution time depends on many factors, including the underlying hardware platform, the program structure, and the code produced by the compiler. Often, the execution time is variable and highly sensitive to the input data the program has to process. This paper presents a code transformation applicable in a compiler backend that produces time-predictable code. The resulting code contains a single input-data independent execution path, in order to obtain programs of stable timing behaviour. The transformation technique has been validated by applying it on a number of benchmarks. Experiments show a reduction of execution time variability, at acceptable costs for the single execution path.

**1998 ACM Subject Classification** C.3 Real-Time and Embedded Systems

**Keywords and phrases** Single-Path, Graph Transformation, Predictable Code, Compiler

**Digital Object Identifier** 10.4230/OASIcs.WCET.2014.103

## 1 Introduction

Hard real-time systems are characterised by the fact that their correctness does not only depend on the computational results but also on the timely delivery thereof – failing to provide a result in time will potentially have catastrophic consequences. For this reason, knowledge of the worst-case execution time (WCET) of a software component in the context of a hard real-time system is essential. Determining the WCET of a program is hard in general, and implies solving two sub-problems: modelling the timing behaviour of the hardware components and determining the possible program execution paths. On the software side, components tend to be highly complex with regard to their behaviour in the presence of different input data and their context. This makes precise automatic analysis of the possible program execution paths intractable in general, as the number of paths grows exponentially in the number of control flow alternatives. These practices are contradictory to at least two key principles of hard real-time systems design: simple structures and composability.

One proposed solution to get around the complexity of WCET analysis is the *single-path approach* [4]. A single-path program is characterised by the fact that it has a singleton program execution path, which makes path analysis superfluous and reduces the need for complex timing models. The execution time of the program is stable with respect to varying

* This work was partially funded by the European Union's 7th Framework Programme under grant agreement no. 288008: Time-predictable Multi-Core Architecture for Embedded Systems (T-CREST) and the EU COST Action IC1202: Timing Analysis on Code Level (TACLe).

```
     cond := ...
     if (!cond) goto Lelse
Lthen:
     x := a + 1
     goto Lend
Lelse:
     x := b - 2
Lend:
     ...
```

**(a)** Update of `x` by branching code.

```
         cond := ...
( cond)  x := a + 1
(!cond)  x := b - 2
         ...
```

**(b)** Update of `x` by predicated assignments.

**Figure 1** Whereas 1a has two alternative execution paths, each containing a different assignment of `x`, 1b consist of a single linear sequence of instructions containing both assignments of `x` with disjoint predicates, such that only one of the two assignments has an effect.

input data, making its temporal behaviour predictable. Ideally, the WCET of a single-path program should be obtainable by simple measurement of one execution.

A set of rules required to transform a piece of code given in a high-level representation to a single-path version was presented in [5]. These rules express the translation from a high-level, well structured source language to predicated statements. While they provide a clear conceptual understanding, they are not sufficient to implement the single-path conversion in a compiler backend. In modern state-of-the-art compilers, source languages are translated to a common intermediate representation (IR), on which optimisations are performed in a language-independent way. A code generator (compiler backend) then translates the optimised IR to target-specific machine language. Control flow is explicit in this representation and typically more general than what can be expressed by a well-structured programming language. This work describes the single-path transformation from a low-level perspective, as graph transformation technique on program control flow graphs, amenable for implementation in a compiler backend. This facilitates automated generation of predictable machine code from any piece of WCET-analysable code.

The rest of this paper is organised as follows: Section 2 provides the theoretical foundations of the single-path approach and basic definitions required for the transformation technique. Section 3 describes the transformation technique with the corresponding execution semantics, to transform almost arbitrary control flow graphs to graphs yielding a single execution path. Section 4 documents the validation of the approach by experiments and the effect on the execution behaviour obtained from these experiments. A short overview of related work is given in Section 5, before we conclude with Section 6.

## 2 Background and Preliminaries

### 2.1 Predicated Execution

The single-path transformation is based on *predicated execution.* A predicated instruction is executed conditionally depending on the value of a Boolean predicate, referred to as *guard*: if the predicate value is true, the instruction is *enabled* and executes as expected, otherwise it is *disabled* and exposes the behaviour of a no-op, that is, the hardware state (register file, memory contents) is not altered as a result of the instruction. By means of predicated instructions it is possible to replace changes in control flow by conditional execution of an instruction sequence, like the code snippets in Figure 1 suggest. In both variants the contents of variable `x` are updated depending on the evaluation of `cond`, but by different means.

Predicated execution, as we intend to exploit it, requires certain assumptions about the target hardware. We assume that all instructions of our target instruction set are predicatable and that the instruction latencies are independent of the operand values, in particular for the predicate operand.

## 2.2 Basic Definitions

Before we detail on the transformation procedure, we briefly give some basic definitions.

A *basic block* (BB) is a straight-line sequence of instructions with one entry point and one exit point. A *control flow graph* (CFG) is a directed graph with basic blocks as nodes and models possible execution paths through a function. We require that a CFG has a distinguished entry node and a distinguished exit node. A node $v$ with more than one successor is associated with a *branch condition*, $cond_v$, a Boolean condition that determines which successor to take on a path.

A node $x$ *dominates* a node $y$ (denoted as $x \operatorname{dom} y$) if every path from the start node to $y$ must go through $x$. A node $x$ *postdominates* a node $y$ (denoted as $x \operatorname{pdom} y$) if every path from $y$ to the exit node must go through $x$. $x$ strictly (post-)dominates $y$ if $x \operatorname{dom} y$ (resp. $x \operatorname{pdom} y$) and $x \neq y$. The immediate (post-)dominator $x$ of a node $y$ is the unique node that strictly (post-)dominates $y$ but does not strictly (post-)dominate any other node that strictly (post-)dominates $y$. (Post-)dominator information commonly is presented in form of a *(post-)dominator tree* in which the entry (resp. exit) node is the root and each node (post-)dominates only its descendants in the tree, i.e., the parent node of each node is its immediate (post-)dominator.

A *loop* $L$ is a strongly connected set of nodes in the flow graph. A *natural loop* has a distinguished entry node, the loop *header*, which dominates all nodes in the loop, and a *back edge* that enters the loop header. Given a back edge $(n, d)$, the natural loop of the edge is defined as $d$ plus the set of nodes that can reach $n$ without going through $d$. $n$ is also called *latch*. An edge $(u, v)$ is an *exit edge* of loop $L$ if $u \in L$ and $v \notin L$. In a *reducible CFG*, every cycle contains a back edge that can be associated with a natural loop. Unless two natural loops have the same header, they are either disjoint or one is nested within the other. If two natural loops share the same header, we treat them as a single loop identified by the header.[1] Furthermore, every node $v$ has a unique header, denoted as $header(v)$, which is the header of the innermost loop it is contained in. We consider an entire procedure as pseudo-loop with the entry node as pseudo-header, such that every node of the CFG except the entry node has a header. Removing all back edges from a reducible CFG results in an acyclic flow graph, the *forward control flow graph* (FCFG).

Given a CFG $G = (V, E)$, nodes $u$, $v$, $x \in V$ and an edge $(u, x) \in E$, $v$ is *control dependent* on $u$ (or on edge $(u, x)$) if $v$ postdominates $x$ but not $u$ [1]. For any node $v \in V$ the set of its control dependence edges is denoted by $CD(v)$:

$$CD(v) \equiv \{(u, x) \mid v \text{ is control dependent on } (u, x)\}$$

The control dependence function induces a partitioning on the nodes of the CFG into equivalence classes: If two nodes $v, w \in V$ are control dependent on the same set of edges, then on every path $\pi$ in $G$ from the entry to the exit node, $v$ is on path $\pi$ if and only if $w$ is on $\pi$.

[1] This differs from the common practice that if a loop is a proper subset of another loop, the former is treated as inner loop and the latter as outer loop.

We require two properties of the CFG, which can be provided by suitable preprocessing: We assume that the CFG is reducible, and that every node in the CFG must have an outdegree of at most two.[2] The second requirement implies that we can name the *dual edge* of an edge if its source node has more than one successor: Let $u \in V$ be a node with two successors $v$, $w \in V$. Then, the *dual edge* of edge $(u, v)$ is $(u, w)$, and the dual edge of $(u, w)$ is $(u, v)$.

## 3 The Single-Path Graph Transformation

We describe our technique as a graph transformation from a source CFG to a target CFG that is extended by predicated execution semantics. In the beginning, we define admissible executions in the source CFG in the presence of loop bounds and present our model of predicated execution in a graph.

An *execution* of a control flow graph $G$ is a path from the entry to the exit node. We require that every loop header in the CFG is associated with a number, the *(local) loop bound*, that limits the number of times the header is (re-)entered on a path before the corresponding loop is left via an exit edge.

▶ **Definition 1** (Admissible execution of a graph). An *admissible execution of a control flow graph* $G$ is a path $\pi$ in $G$ on which each loop bound for any header $h \in \pi$ is respected.

For predicated execution, every node is associated with a guarding predicate that determines whether the node is enabled or disabled. Predicates can be seen as part of state which is altered as nodes are visited along a path. Following semantic actions can be performed by nodes and edges:

- A node may set a predicate to the branch condition or its negation. This action is predicated itself by the node's guard.
- An edge may set or clear a set of predicates (set them to true/false), unconditionally.
- An edge may copy the value of one predicate to another predicate, unconditionally.

▶ **Definition 2** (Single-Path Graph Transformation). Let $G = \langle V, E \rangle$ be a graph with local loop bounds. The *Single-Path Graph Transformation* obtains a graph $G^{SP}$ extended by predicated execution, and a path $\pi^{SP}$ in $G^{SP}$, such that for any admissible path $\pi$ in $G$, the sequence of nodes along $\pi$ equals the sequence of enabled nodes along $\pi^{SP}$ in $G^{SP}$.

The single-path graph transformation computes a singleton path $\pi^{SP}$ through $G^{SP}$ that includes every admissible execution path in the source graph. In the following, we describe how we compute $G^{SP}$ and $\pi^{SP}$. Section 3.1 reviews the computation of predicates and the conversion of acyclic graphs. In Section 3.2, these ideas are extended to obtain single-path loops, that is, loops that may contain nested loops and have a fixed iteration count. Section 3.3 summarises the construction of the single-path graph $G^{SP}$ by composition of single-path loops.

### 3.1 From Control-Flow to Predicates

Our transformation technique is based on the *RK algorithm* of Park and Schlansker [3]. Their motivation is the ability to speed up the execution of innermost loops by means of a software

[2] In practice, this implies that jump tables (e.g. resulting from `switch` statements) must be replaced by cascades of two-way alternatives.

pipelining technique, which requires a linear sequence of instructions without control-flow changes. As the algorithm is applied to innermost loops only, the CFGs under consideration are acyclic in nature. We show how to apply the algorithm to transform an acyclic CFG to a linear sequence of predicated basic blocks.

Given an acyclic graph $G = (V, E)$, the algorithm assigns a unique predicate to each of the equivalence classes induced by the control dependence relation. The set $CD(v)$ for all $v \in V$ is computed by means of the postdominator tree (PDT) of $G$ [1]: Each node $v \in V$ with two successors is identified. Then, for each successor $w$ of $v$, the PDT is walked upward, starting at $w$, until (excluding) the immediate postdominator of $v$ is reached. As each node $u$ is visited during the walk, the edge $(v, w) \in E$ is added to the set $CD(u)$.

The set of predicates is denoted as $P$. Each predicate $p_i \in P$, $i \geq 0$ corresponds to a set of (control dependence) edges, which is expressed as function $K(p_i)$. The function $R(v)$ associates each $v \in V$ with a predicate $p_i \in P$ such that, for $v$, $w \in V$,

$$CD(v) = CD(w) \qquad \Leftrightarrow \qquad R(v) = R(w) = p_i \;\; \wedge \;\; K(p_i) = CD(v) = CD(w)$$

Control flow is mapped to predicate values in the resulting sequence of guarded blocks with the following goal: In any execution, for any $p_i \in P$, if a control dependence edge $e \in K(p_i)$ would have been taken in the original acyclic graph, $p_i$ should be true, such that every node $v \in V$ with $R(v) = p_i$ is enabled, otherwise $p_i$ should be false (and the corresponding nodes disabled). To this end, following steps need to be performed:

1. For each predicate $p_i \in P$ and each edge $(u, v) \in K(p_i)$, add a predicate assignment of the form $p_i \leftarrow cond_u$ as semantic action to node $u$, if $v$ follows $u$ when $cond_u$ is true, else add assignment $p_i \leftarrow \neg cond_u$ to $u$.
2. Guard nodes $v \in V$ by predicate $R(v)$.
3. As predicate assignments are potentially disabled in an execution, it is necessary to care for a correct initialisation of predicates. Therefore, predicates are initialised to false at an artificial entry edge as necessary.
4. The nodes are reconnected in the order of a topological sort, such that a linear sequence of predicated nodes is obtained.

## 3.2 Single-Path Loops

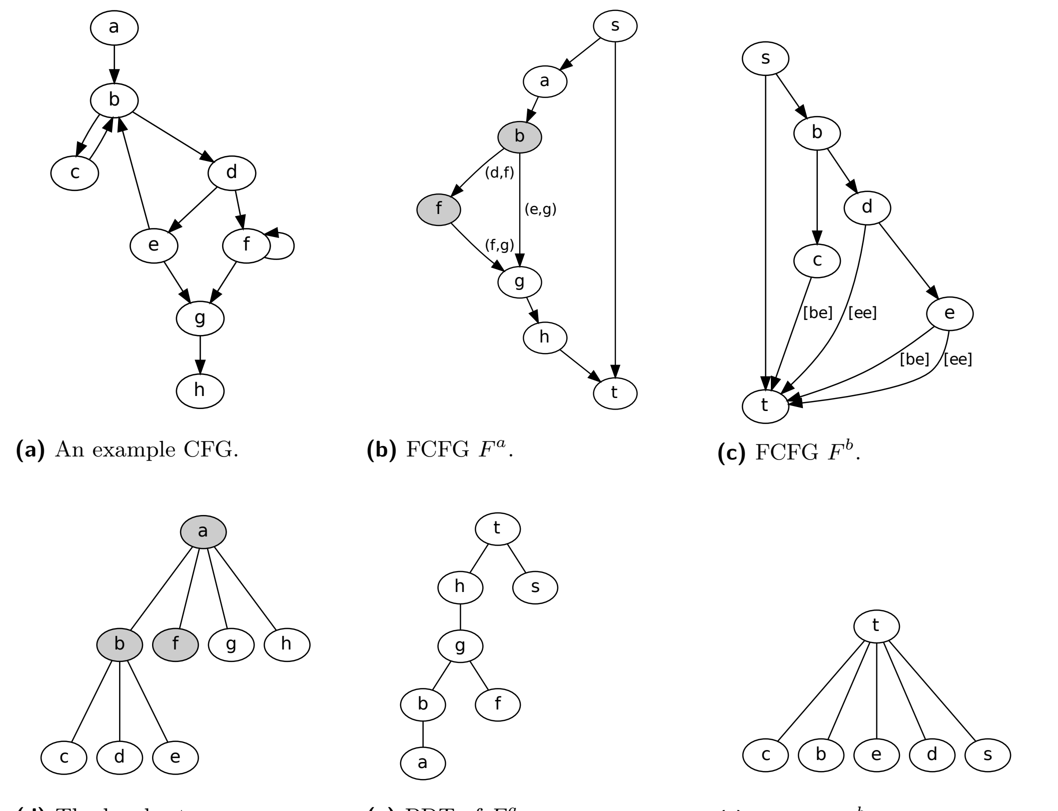
At the beginning of the transformation, we identify loops in the CFG and compute $header(v)$ for each $v \in V$, e.g, by following the procedure in [8].[3] For each loop, we identify back edges and exit edges.

We now consider a loop of the CFG in isolation. Let $L^h$ be a loop of the CFG with header $h$.[4] We construct the acyclic FCFG induced by $L^h$, and augment it by two distinguished nodes $s^h$ and $t^h$, and edges $(s^h, h)$, $(s^h, t^h)$, and $(\ell, t^h)$ for all latches $\ell$ of $L^h$, and $(e, t^h)$ for all sources $e$ of exit edges of $L^h$. Every node in the resulting graph, denoted as $F^h$, is dominated by $s^h$ and postdominated by $t^h$.

A key insight for the extension of the procedure in Section 3.1 is that loops of a reducible flow graph can be represented compacted into a single node. Consequently, each contained inner loop is compacted into a single node in $F^h$, namely its header. As a result, the outgoing edges of an inner loop header correspond to the exit edges of the inner loop.

[3] $header(v)$ is the node into which $v$ is eventually contracted during reduction of the CFG (procedure REDUCE in [8]).

[4] In the following, we use the header $h$ in superscript to denote information specific to a single loop identified by $h$, to avoid ambiguities.



**(a)** An example CFG. **(b)** FCFG $F^a$. **(c)** FCFG $F^b$.

**(d)** The header tree. **(e)** PDT of $F^a$. **(f)** PDT of $F^b$.

**Figure 2** An example to illustrate the single-path graph transformation steps.

▶ **Example 3.** The transformation is best illustrated by an example (Figure 2). Figure 2a shows the CFG, with entry node $a$ and exit node $h$. Figure 2d depicts the header information for each $v \in V$ as tree: the parent of node $v$ is $header(v)$, and each header is drawn shaded. Apart from the outermost pseudo-loop $L^a$ with header $a$, the CFG contains two loops, loop $L^b$ with header $b$ and the self-loop $L^f$ consisting of node $f$. Figure 2b depicts the FCFG for $L^a$, where $b$ and $f$ represent inner loops $L^b$ and $L^f$ compacted into a single node, and the outgoing edges are labelled with the corresponding exit edges. Figure 2c depicts the FCFGs for $L^b$, where the back edges and exit edges are represented by the edges labelled with [be] and [ee], respectively. The FCFG $F^f$ for the self-loop $L^f$ is not shown.

Next, $CD^h(v)$ is computed for all $v \in F^h$, with the aid of the postdominator tree of $F^h$, as described in Section 3.1. The main difference lies in the inclusion of control dependence edges: Edges originating from inner loop headers in $F^h$ correspond to exit edges of the inner loops, and these exit edges have to be added to the respective control dependence sets.

Following the description in Section 3.1, $R^h$ and $K^h$ are computed from $CD^h$, nodes are guarded according to $R^h$ and predicate assignments are added according to $K^h$. $R^h$ maps predicates $p_i^h \in P$ to each node $v \in F^h \setminus \{s^h, t^h\}$. By convention, let $R^h(h) = p_0^h$. If $v$ represents an inner loop, $R^h(v)$ is not the final guard of the inner loop header, but a predicate whose value decides whether the inner loop is enabled at least once. Note that an edge $(u, v) \in K^h(p_i^h)$ may be an exit edge originating from an inner loop, and hence node $u$ is not necessarily a node in $F^h$. For edge $(s^h, h)$, we add the semantic action to clear all predicates $p_i^h \in P$ for $i > 0$, to provide a correct initialisation of predicates.

In the acyclic graph $F^h$, the header $h$ and every node that postdominates $h$ are only control dependent on the edge $(s^h, h)$, which corresponds to the loop entry. If the header predicate $R^h(h) = p_0^h$ is true, all associated guards of the nodes on any path in $F^h$ from $s^h$ to $t^h$ are set to true as a result of the semantic actions, while the guards of the nodes that are not on that path remain false. In the transformed single-path graph, in any execution, all nodes of $F^h \setminus \{s^h, t^h\}$ are visited in the order of a topological sort.

So far, we have dealt with a path through a single iteration of the loop. In the source CFG, except for the top-level pseudo loop, a loop $L^h$ is entered via its header $h$ up to $N$ times on an admissible path, where $N$ denotes the local loop bound of $L^h$. As we want to obtain a single execution path that contains all admissible paths through the source CFG, this path must contain the sequence of nodes in $L^h$ at least $N$ times. Therefore, the single-path graph $G^{SP}$ will contain a back edge from the last node in the sequence of nodes of $L^h$ to the header $h$, which is taken exactly $N - 1$ times in the single execution path $\pi^{SP}$. The semantic action of this edge is, like on $(s^h, h)$, to clear all predicates $p_i^h \in P$ for $i > 0$.

The semantic actions on predicates need to be extended, otherwise the header predicate $p_0^h$ is never set to false. If $p_0^h$ is false at the beginning of an iteration, no guard from a loop member ever becomes true, and all nodes of $L^h$ remain disabled in that iteration. Hence, on the single execution path, $p_0^h$ must be false starting with the $(i+1)$-th up to the $N$-th iteration, if on an admissible path in the source graph the loop iterates $i$ times. Recall that a loop is left via one of its exit edges. As each exit edge necessarily has a dual edge (otherwise, the loop header would not be reachable from the edge source, disqualifying the latter as loop member), we add the dual edge of each exit edge of $L^h$ to $K^h(p_0^h)$.

▶ **Example 4.** Figure 2e depicts the PDT for the computation of $CD^a$ in FCFG $F^a$. $CD^a(x) = \{(s, a)\}$, $\forall x \in \{a, b, g, h\}$, and $CD^a(f) = \{(d, f)\}$. Note that $(d, f)$ is the exit edge of $L^b$ that corresponds to $(b, f)$ in $F^a$. Consequently, we assign $R^a(x) = p_0^a$, $\forall x \in \{a, b, g, h\}$ with $K^a(p_0^a) = \{(s, a)\}$, and $R^a(f) = p_1^a$ with $K^a(p_1^a) = \{(d, f)\}$. A topological order of the nodes is $\langle a, b, f, g, h \rangle$.

Figure 2f shows the PDT for $F^b$. $CD^b(b) = \{(s, b)\}$, $CD^b(c) = \{(b, c)\}$, $CD^b(d) = \{(b, d)\}$, and $CD^b(e) = \{(d, e)\}$. We assign $R^b(b) = p_0^b$, $R^b(d) = p_1^b$, $R^b(e) = p_2^b$, $R^b(c) = p_3^b$. Obtaining the respective $K^b(p_i^b)$ is straightforward, but we have to extend $K^b(p_0^b)$ by the dual edges $\{(e, b),\ (d, e)\}$ of the exit edges of $L^b$. A topological order of the nodes is $\langle b, c, d, e \rangle$.

For the single node self loop $f$, $R^f(f) = p_0^f$ with $K^f(p_0^f) = \{(s, f),\ (f, f)\}$.

## 3.3 Composition of the Single-Path Graph

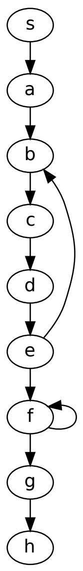
We construct the single-path graph $G^{SP} = (V, E^{SP})$ by a preorder traversal of the loop header tree, visiting the nodes on each level in a topological sort order, and adding edges for $G^{SP}$ as required. The recursive construction algorithm is sketched in Figure 3. It is invoked with the entry node of the source CFG.

When a loop header $h$ is visited, we construct $F^h$, compute $R^h$, $K^h$, guard nodes and add semantic actions, as described in Section 3.2. On the entry edge to an inner loop $L^{h'}$, we add the semantic actions of $(s^{h'}, h')$ (clearing the predicates). Additionally, we add another semantic action to copy the predicate of $h'$ in the outer loop to the predicate of $h'$ in the inner loop. For the CFG, an artificial entry edge is introduced to initialise all predicates of the top-level pseudo loop to false and the predicate of the entry node to true.

For our example, the resulting single-path graph is depicted in Figure 4, together with the guards and semantic actions.

```
Function COMPOSESP($h$: header node)
Compute $R^h$, $K^h$
Add predicate assignments according to $K^h$
Guard header node $h$ by $R^h(h)$ $(= p_0^h)$
$last = h$          // keep a reference to the last node visited
for each $n \in F^h \setminus \{h, s^h, t^h\}$ in topological sort order :
    $E^{SP} = E^{SP} \cup \{(last, n)\}$
    if $n$ is a loop header :
        Add semantic actions to entry edge $(last, n)$: $p_0^n \leftarrow R^h(n)$; $\forall i > 0 : p_i^n \leftarrow 0$
        $last = $ COMPOSESP$(n)$                 // process inner loop
        $E^{SP} = E^{SP} \cup \{(last, n)\}$          // add back edge to inner loop
        Add semantic actions to back edge $(last, n)$ of inner loop: $\forall i > 0 : p_i^n \leftarrow 0$
    else :
        Guard node $n$ by $R^h(n)$
        $last = n$
return $last$
```

**Figure 3** Algorithm for the composition of the single-path graph.



| Node/Edge | Guard | Semantic action |
|---|---|---|
| $a$ | $p_0^a$ | – |
| $b$ | $p_0^b$ | $p_1^b \leftarrow cond_b;\ p_3^b \leftarrow \neg cond_b$ |
| $c$ | $p_3^b$ | – |
| $d$ | $p_1^b$ | $p_0^b, p_2^b \leftarrow \neg cond_d;\ p_1^a \leftarrow cond_d$ |
| $e$ | $p_2^b$ | $p_0^b \leftarrow cond_e$ |
| $f$ | $p_0^f$ | $p_0^f \leftarrow cond_f$ |
| $g$ | $p_0^a$ | – |
| $h$ | $p_0^a$ | – |
| $(s, a)$ | – | $p_0^a \leftarrow 1;\ p_1^a \leftarrow 0$ |
| $(a, b)$ | – | $p_0^b \leftarrow p_0^a;\ p_1^b, p_2^b, p_3^b \leftarrow 0$ |
| $(e, f)$ | – | $p_0^f \leftarrow p_1^a$ |

**Figure 4** The complete single-path graph resulting from the transformation. For semantic actions, 0 is false, 1 is true, and $cond_v$ is the branch condition in node $v$.

## 4 Experiments

To validate the single-path graph transformation, we have created a simulation framework, which serves two purposes. First, it allows for an experimental validation of the transformation procedure, with arbitrary CFGs as input. Second, it provides means to evaluate the estimated execution cost of the single-path-transformed graph relative to the original CFG.

In the framework, after a given CFG $G$ is transformed to $G^{SP}$ and extended by semantic actions on predicates, admissible paths $\pi$ through $G$ are chosen by random. For one admissible path, the branch conditions are recorded. Then, the single execution path $\pi^{SP}$ through $G^{SP}$ is walked, and predicates are updated according to the recorded branch conditions in $\pi$. The

**Table 1** Experiments on the Mälardalen benchmarks. Functions *adpcm/decode*, *fdct/fdct*, *jfdctint/jpeg_fdct_islow*, *loop3/main*, and *matmult/Test* are omitted because their generated CFGs already have a single execution path. Functions *adpcm/encode*, *bs/binary_search*, *bsort100/BubbleSort*, *cnt/Test*, *crc/icrc*, *duff/duffcopy*, and *edn/main* have almost a single execution path with a ratio below 1.10.

| Benchmark/Function | Mean | Std.Dev. | Min | Max | SP | $\|P\|$ | Ratio |
|---|---|---|---|---|---|---|---|
| adpcm/upzero | 73.21 | 21.57 | 53 | 96 | 125 | 3 | 1.30 |
| compress/compress | 1178.13 | 840.65 | 451 | 3589 | 4200 | 28 | 1.17 |
| cover/swi120 | 1512.05 | 18.08 | 1475 | 1570 | 2655 | 7 | 1.69 |
| expint/expint | 1940.28 | 18.45 | 1892 | 1984 | 2736 | 4 | 1.38 |
| fir/fir_filter_int | 1704.76 | 451.32 | 595 | 2618 | 3236 | 6 | 1.24 |
| insertsort/main | 496.24 | 109.97 | 246 | 750 | 832 | 6 | 1.11 |
| janne_complex/complex | 695.98 | 181.55 | 215 | 1135 | 1381 | 6 | 1.22 |
| lcdnum/num_to_lcd | 33.94 | 0.98 | 30 | 36 | 190 | 45 | 5.28 |
| lms/main | 94794.29 | 4106.37 | 86184 | 103839 | 150209 | 32 | 1.45 |
| ludcmp/ludcmp | 301329.73 | 16345.35 | 275478 | 328187 | 415544 | 16 | 1.27 |
| minmax/main | 55.21 | 7.18 | 48 | 70 | 83 | 7 | 1.19 |
| minver/minver | 135115.74 | 55424.42 | 70709 | 197066 | 371563 | 20 | 1.89 |
| qsort-exam/sort | 4609.72 | 1128.15 | 908 | 7253 | 12004 | 22 | 1.66 |
| qurt/qurt | 590.77 | 686.31 | 44 | 1838 | 2208 | 15 | 1.20 |
| select/select | 4345.63 | 896.34 | 2043 | 7196 | 11219 | 16 | 1.56 |
| statemate/FH_DU | 232.39 | 39.56 | 157 | 305 | 360 | 25 | 1.18 |

sequence of enabled nodes in $\pi^{SP}$ is compared against the sequence of nodes of $\pi$ and must be identical.

Furthermore, we generated the CFGs from functions from the well-established WCET benchmarks of the Mälardalen WCET research group[5], providing a simple cost model and local loop bounds.[6] We recorded the execution cost for 100 randomly generated paths through each CFG, and obtained the mean cost, the standard deviation, and the minimum and maximum observed cost. We computed the ratio of the cost of the single-path execution (SP) to the maximum observed cost (Max). As we are interested in the worst case, we forced the paths chosen in $G$ to always execute the maximum number of loop iterations. In addition, we listed the required number of predicates ($|P|$). The results of this comparison are shown in Table 1. The ratio SP/Max was mostly below 1.9. In one case (lcdnum/num_to_lcd), the ratio is about 5.4, which stems from the fact that the function contains a switch statement that is serialised in the single-path graph.

## 5 Related Work

Techniques other than the single-path approach have been proposed to make code more predictable for WCET-analysis. Apart from avoiding problematic code constructs in the first place (e.g. indirect calls, irreducible and input-data dependent loops, recursion), code transformations have been suggested to reduce the number of paths required to be analysed by making infeasible paths explicit or factor out code blocks with constant execution time [2]. Both transformations seem hard to be performed automatically in a compiler and no general

[5] Accessible online: `http://www.mrtc.mdh.se/projects/wcet/benchmarks.html`
[6] The cost of executing a block is the number of instructions in the block. We used local loop bounds bounds recorded by simulation, and for the loops not executed in the simulation, we added a bound of 20.

solution has been supplied so far. Compiler support to aid WCET analysis by providing information available during compilation to the timing analysis is an orthogonal approach to obtain more predictable code [6].

## 6 Discussion and Outlook

We have presented the single-path graph transformation, a technique to transform any reducible control-flow graph into a graph with a single execution path of predicated nodes. The goals of the single-path approach are to minimise control flow complexity and to obtain code with stable, input-data independent timing behaviour. The here-presented technique can be implemented as part of a compiler backend to generate time predictable code in an automated way.

Because the single-path approach serialises control flow, it has been criticised to be too costly to be applied in practice. In our experiments, the costs stay within reasonable limits (the ratio is below 1.9 in all but one cases) when the worst case is considered, for a simple model. Furthermore, the cost could be compensated partially by means of hardware support particularly suitable for single-path code, e.g., by providing a multiple-issue pipeline, instruction prefetching, or hardware loops.

We are implementing the single-path transformation as part of a compiler backend for the Patmos processor, a multi-core processor designed for high performance at high time-predictability [7], in the T-CREST project (`http://www.t-crest.org/`). A next logical step to improve the transformation would be to omit input-data independent regions during if-conversion, as they do not contribute to input-data induced variability.

## References

1 Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, July 1987.

2 Hemendra Singh Negi, Abhik Roychoudhury, and Tulika Mitra. Simplifying wcet analysis by code transformations. In *Workshop on Worst-Case Execution-Time Analysis (WCET)*, September 2004.

3 Joseph C.H. Park and Mike Schlansker. On predicated execution. Technical report, Hewlett Peckard Software and Systems Laboratory, May 1991.

4 Peter Puschner. The single-path approach towards wcet-analysable software. In *2003 IEEE International Conference on Industrial Technology*, volume 2, pages 699–704 Vol.2, 2003.

5 Peter Puschner, Raimund Kirner, Benedikt Huber, and Daniel Prokesch. Compiling for time predictability. In *Proc. SAFECOMP 2012 Workshops (LNCS 7613)*, pages 382–391. Springer, 2012.

6 Peter Puschner, Daniel Prokesch, Benedikt Huber, Jens Knoop, Stefan Hepp, and Gernot Gebhard. The T-CREST approach of compiler and WCET-analysis integration. In *Proceedings of the 9th Workshop on Software Technologies for Future Embedded and Ubiquitious Systems (SEUS 2013)*, 2013.

7 Martin Schoeberl, Pascal Schleuniger, Wolfgang Puffitsch, Florian Brandner, Christian W. Probst, Sven Karlsson, and Tommy Thorn. Towards a time-predictable dual-issue microprocessor: The Patmos approach. In *First Workshop on Bringing Theory to Practice: Predictability and Performance in Embedded Systems (PPES 2011)*, pages 11–20, March 2011.

8 Robert Endre Tarjan. Testing flow graph reducibility. *J. Comput. Syst. Sci.*, 9(3):355–365, December 1974.