# Constant-Loop Dominators for Single-Path Code Optimization

**Emad Jacob Maroun**
Institute of Computer Engineering, TU Wien, Vienna, Austria

**Martin Schoeberl**
Department of Applied Mathematics and Computer Science, Technical University of Denmark, Lyngby, Denmark

**Peter Puschner**
Institute of Computer Engineering, TU Wien, Vienna, Austria

## Abstract

Single-path code is a code generation technique specifically designed for real-time systems. It guarantees that programs execute the same instruction sequence regardless of runtime conditions. Single-path code uses loop bounds to ensure all loops iterate a fixed number of times equal to their upper loop bound. When the lower and upper bounds are equal, the loop must iterate the same number of times, which we call a constant loop.

In this paper, we present the constant-loop dominance relation on control-flow graphs. It is a variation of the traditional dominance relation that considers constant loops to find basic blocks that are always executed the same number of times. Using this relation, we present an optimization that reduces the code needed to manage single-path code. Our evaluation shows significant performance improvements, with one example of up to 90%, with mostly minor effects on code size.

**2012 ACM Subject Classification** Computer systems organization → Real-time systems; Theory of computation → Graph algorithms analysis; Theory of computation → Control primitives; General and reference → Performance

**Keywords and phrases** single-path, dominators, algorithms, optimization, control-flow graph

**Digital Object Identifier** 10.4230/OASIcs.WCET.2023.7

**Supplementary Material** *Software (Source Code)*: `https://github.com/t-crest/patmos-llvm-project/tree/82eb73bff7336674027afecb254f1e3ebd1c23c2`
archived at `swh:1:rev:82eb73bff7336674027afecb254f1e3ebd1c23c2`

**Acknowledgements** This work has been supported by the Doctoral College Resilient Embedded Systems, which is run jointly by the TU Wien's Faculty of Informatics and the UAS Technikum Wien.

## 1 Introduction

Real-time systems are unique in their timing requirements. In addition to producing the correct logical results, real-time programs must produce these results within a specific time frame called the *deadline*. A result produced after the deadline is unacceptable, regardless of its logical correctness. Real-time systems must statically guarantee that a task terminates within the deadline. Here, a program's *worst-case execution time* (WCET) is the critical metric. In the simplest case, if the WCET can always be shown to be shorter than the deadline, we know that the program will always produce its result in time for it to be useful. For multi-task and multi-processor systems, scheduling must be done using each task's WCET to ensure all tasks adhere to their deadlines.

It is almost impossible to know the actual WCET of a program. Therefore, WCET analysis provides an upper bound for it. This *WCET bound* can be used instead of the real WCET when designing the system and verifying its timings. However, the halting problem


has shown that creating a program (in this case, a WCET analyzer) that can tell whether any given program will terminate is impossible [18]. To get around this inconvenience, real-time programs are developed with certain restrictions that allow the code to be analyzable without running afoul of the halting problem. One such restriction is to have loops with a bound on the maximum number of iterations. In a real-time program, all loops must have an upper bound on the number of iterations they may perform at runtime. This is a guarantee that the programmer provides – often in the form of an annotation in the code – to the WCET analyzer, which allows the analyzer to calculate an upper bound to the execution time. A *best-case execution time* (BCET) is also often of interest for task scheduling [23]. To enable efficient BCET analysis, a lower bound on loops is also often provided such that the analyzer does not have to use zero as the default lower bound. If a loop's lower and upper bounds are equal, we call it a *constant loop*; a loop that always executes the same number of iterations.

*Single-path code* generation is a code-generation technique that ensures that programs execute the same sequence of instructions regardless of runtime conditions [22]. This type of code makes WCET analysis much easier, as the analyzer does not have to account for the program executing different code traces based on what happens at runtime. The properties of single-path code can significantly affect execution time [21]. Therefore, it must be optimized to reduce the execution-time overhead.

The control-flow graph (CFG) is a directed graph that shows how execution can flow through a function.[1] Each node represents a block of sequential code, with edges specifying where execution can continue. We use *block* and *node* interchangeably in this paper. If a node in the CFG has multiple outgoing edges, we call that a branch. Depending on some runtime condition, execution continues at the target node of one edge. Loops in a function are represented by cycles in the CFG. The dominance relation identifies whether a node is guaranteed to be executed before another node. This relation is critical in compiler construction to ensure correct code generation and optimization [2]. However, the relation does not account for loop bounds and constant loops.

In this paper, we present a new CFG relation called *constant-loop dominance*. It is a variation of dominance that accounts for whether loops are constant to find blocks executed a fixed number of times. Functions called from such blocks can be optimized to reduce the overhead of converting them to single-path code. The contributions of this paper are: (1) a definition of the constant-loop dominance relation and an algorithm for calculating it; (2) a description of an optimization to single-path code that makes use of the relation; and (3) an implementation of the algorithm and optimization in a compiler that produces single-path code.

The paper is organized into six sections: The following section presents related work. Section 3 provides background information to support the understanding of the rest of the paper. Section 4 introduces the constant-loop dominance relation, an algorithm for finding constant-loop dominators, how it is used to optimize single-path code, and a brief description of the implementation. Section 5 evaluates our optimization's performance and code size impacts. Section 6 concludes the paper.

## 2 Related Work

The dominance relation is fundamental within compiler construction. Its first description was given with a simple, $\mathcal{O}(n^4)$ algorithm [17]. It was used to implement global common expression elimination and loop identification. Its use continued in other important advances

[1] We do not consider inter-procedural CFGs in this paper.

like enabling the efficient computation of static single assignment form [6], which opens up further optimization opportunities [10, 24, 30]. Significant work has been put into reducing the runtime complexity of computing the dominance relation [1, 12, 29]. The state-of-the-art includes an algorithm that runs in $\mathcal{O}(m\alpha(m, n))$, where $n$ is the number of nodes, $m$ is the number of edges, and $\alpha$ is the inverse Ackermann's function [16]. Finally, the quest for a linear time algorithm has resulted in several proposals [3, 5, 9]. The challenge has been translating the theoretical runtime complexity into practical implementations that outperform the older, non-linear algorithms.

Knowledge about loop bounds is a fundamental requirement for analyzing WCET. As such, any annotation language must include the ability to specify bounds [14]. However, since manual annotations can be tedious for programmers to provide and be a source of imprecision and errors, significant effort has gone into automatic methods for finding loop bounds [4, 11, 28]. Effort has been put into finding scenarios that can automatically derive loop bounds. E.g., upper loop bounds can be derived by assuming a loop terminates and then enumerating the state-space of the variables that influence the loop exit [7]. Machine learning has also been used to try and find loop bounds [13]. While our work only uses annotated loop bounds to find constant loops, any method for finding loop bounds is compatible.

Single-path code was introduced as a code generation technique specifically for real-time systems [22]. It can be automatically generated from any WCET analyzable source code, with a significant but manageable performance cost [21]. Single-path code is challenged by its execution-time overhead. One avenue for improving this is to take advantage of its inherent instruction-level parallelism when scheduling on a VLIW architecture [20]. In [19], we extend single-path code with techniques to compensate for execution-time variability from memory accesses. This ensures that single-path code has a constant execution time, eliminating the need for WCET analysis.

# 3 Background

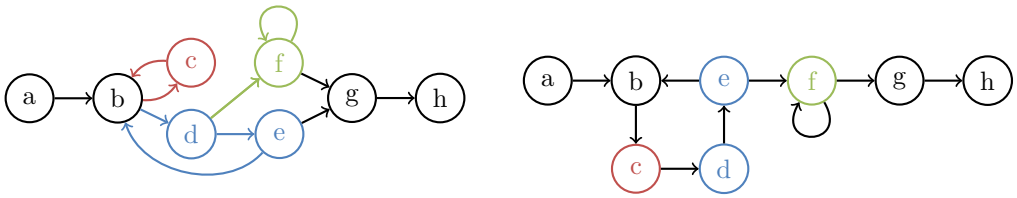
## 3.1 The Patmos Processor

Patmos was specifically designed for real-time systems. It is a RISC-style instruction-set architecture with features that make it time-predictable and optimized for a low WCET [27]. Patmos has an in-order, dual-issue pipeline that maximizes throughput while being time-predictable. All instructions are predicated by one of eight boolean predicate registers. If the value of the predicate is true, an instruction is *enabled*, which means it executes normally. If the predicate is false, the instruction is *disabled*. It still gets executed in the same amount of time. However, it does not read from or write to any memories or update any registers; effectively, the instruction becomes a no-op.

## 3.2 Single-Path Code

Single-path code was initially intended to make it computationally easier to perform WCET analysis. Single-path code uses predication to convert the branching control flow of a function into an instruction stream with only one execution path. To convert a function into single-path code, three techniques are used:

**If-Conversion.** Any conditional branching is converted into predicated instructions, such that only the needed path's instructions are enabled at runtime. The resulting code always executes all instructions in both paths, with only one path being enabled at a time. Looking




**(a)** Traditional. **(b)** Single-Path.

**Figure 1** Conversion of a function with branching control flow (left) to single-path code (right).

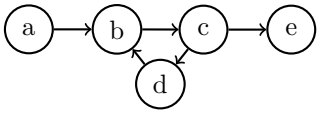
at Figure 1, we can see the result of transforming a function to single-path code. Block `b` conditionally branches to either `c` or `d`. The color coding of Figure 1a's blocks matches the conditions that led to that path being taken. In Figure 1b, the colors indicate that only if the corresponding condition is true will the block's instructions be enabled at runtime. As such, we can see how if-conversion results in `b` always leading to first `c` and then `d`. However, only if the red condition holds at runtime will `c`'s instructions be enabled. The same holds for `d`, leading to either `e` or `f` in the traditional code, but eventually leading to both in the single-path version. Notice how we have not colored the edges in the single-path version, as they are always taken.

**Loop-Conversion.** Loops may iterate a variable number of times depending on runtime conditions. To avoid this variability, single-path code converts loops to always iterate the maximum possible number of times. Any superfluous iterations are instead disabled using predication. Looking at our example, we can see the function has two loops, one containing the blocks `b`, `c`, `d`, and `e`, and the second containing only `f`. A single-path loop maintains a count of how many iterations have been executed and keeps looping until the maximum is reached. Inside the loop, the condition that traditionally breaks out of the loop is instead used as the predicate to all the instructions. This condition will become false at some point, meaning any further iterations will have their instructions disabled.

**Function-Conversion.** Single-path code also has to account for function calls. Say we have two branching paths, one of which performs a function call while the other does not. If we predicate the function call as we do for the rest of the instructions, it will not cause control to shift to the called function. This means the function's instructions are not executed (neither enabled nor disabled) if the path it was called from was disabled. Function-conversion ensures every function call is performed the same number of times, analogously to loop-conversion. Any call not logically necessary is instead disabled. Function-conversion copies all functions that are called within single-path code. The copies are then modified to take an extra predicate register argument, which specifies whether the function was called from an enabled or disabled path. The body of the copied function is then predicated on that register; if it is called from a disabled function, it will be disabled, too, and vice versa. The call instruction in the caller is not predicated, instead being provided with the predicate of the calling code to pass on. This ensures all functions are always called and executed, regardless of whether their callers are enabled or disabled.

## 3.3 Definitions

A *loop* in a CFG is a set of strongly connected nodes, i.e., a path exists between any two nodes. A *natural loop* additionally has an entry node, the *header*, which dominates all other nodes in the loop, and a *back edge* that enters the header from another node in the loop.




| | Dom | CLDom ($const(b)$) | CLDom ($\neg const(b)$) |
|---|---|---|---|
| a | a, b, c, d, e | a, b, c, d, e | a, b, c, d, e |
| b | b, c, d, e | b, c, d, e | |
| c | c, d, e | c, d, e | |
| d | d | d, e | |
| e | e | e | e |

**Figure 2** Example CFG with the traditional dominator and constant-loop dominator relations.

The source node of a back edge is called a *latch*. An *exit edge* is an edge that connects a node in the loop to one outside of it. If two natural loops have the same header, we treat them as the same loop. We refer to these "merged" natural loops as a single *loop*. We define the number of iterations a loop performs as the number of times a path enters its header. All loops are either disjoint or nested within one another and identified by their headers. As such, every node has a header, which is the header of the innermost loop it is contained in. For consistency, we also consider the entire function as a pseudo-loop with the entry node as the header. Nodes without successors are *end nodes* and are assumed to return from the function.

Removing all back edges from the CFG results in an acyclic graph called a *forward control-flow graph* (FCFG). We partition the FCFG into *loop FCFGs* of the subgraphs containing only nodes whose header is the loop header. This means that for each loop, we now have a dedicated FCFG. Each node in the graph is only in one FCFG, except the headers, which reside in the FCFG of their enclosing loop and in the FCFG of the loop for which they are headers. Exit edges are represented as edges from the header of the inner loop to the original target node in the outer loop.

## 4 Constant-Loop Dominance

We consider CFGs with an optional label, `const`, on the headers of loops. If the label is present, it means paths through the loop header must visit the header a fixed number of times before exiting its loop. Otherwise, the number of visits may fluctuate between different paths. We define the constant-loop dominance relation as follows: **A node `x` constant-loop dominates a node `y` (*x cldom y*) if every path from the entry to `y` visits `x` a fixed number of non-zero times.**

Looking at Figure 2, we can see the traditional dominator relation (Dom) and the constant-loop dominator relations (CLDom) for the given CFG. CLDom is shown with the loop headed by `b` being both constant and variable. The most obvious difference is that when the loop is variable, none of its nodes constant-loop dominate other nodes or themselves. If the loop is constant, we can see the result is the same except `d` also constant-loop dominates `e`, unlike with traditional dominance. The last iteration of the loop must exit through `c`, meaning `d` is always visited $i - 1$ times, where $i$ is the max iteration count. Note that constant-loop dominators behave the same as traditional dominators in the absence of loops.

### 4.1 Algorithm

Finding traditional dominators on *acyclic directed graphs* (DAGs) is done by calculating the dominance for each node in topological order. Using topological order ensures that the dominance of a node's predecessors is established before getting their intersection to result in the dominance of the current node (and remembering to add self-dominance.)

**Algorithm 1** Constant-Loop Dominators.

CLDom(s): ▷ Starting node as input

1: $H \leftarrow$ Inner loop headers
2: $D \leftarrow \emptyset$ ▷ Dominator set for nodes of $fcfg(s)$
3: $ID \leftarrow \emptyset$ ▷ Dominator sets for inner loops
4: $IED \leftarrow \emptyset$ ▷ End-Dominator sets for inner loops
5: **for** $h$ *in* $H$ **do** ▷ Analyze inner loops
6: $\quad ID[h], IED[h] \leftarrow CLDom(h)$
7: **end for**
8: **for** $b \mid b \in topological\_sort(fcfg(s))$ **do** ▷ Find dominators
9: $\quad P \leftarrow \bigcap\{D[a] \cup IED[a] \mid \forall(a,b) \in fcfg(s) \wedge a \in H \wedge const(a)\} \cap$
$\qquad \bigcap\{D[a] \mid \forall(a,b) \in fcfg(s) \wedge (a \notin H \vee \neg const(a))\}$
10: $\quad D[b] \leftarrow \begin{cases} P \cup b & \text{if } b \notin H \vee const(b) \\ P & \text{otherwise} \end{cases}$
11: **end for**
12: **for** $h, v \mid \forall h \in H \wedge \forall v \in ID[h]$ **do** ▷ Extract dominators from loops
13: $\quad D[v] \leftarrow \begin{cases} D[h] \cup ID[h][v] & \text{if } const(h) \\ D[h] & \text{otherwise} \end{cases}$
14: **end for**
15: $L \leftarrow \bigcap\{header\_end\_dominators(l, D, IED) \mid \forall(l,s)\}$
16: $E \leftarrow \bigcap\{header\_end\_dominators(e, D, IED) \mid \forall(e,c) \in exit\_edges(s)\}$
17: $C \leftarrow \{c \mid \forall c \notin exits(s) \wedge \forall e \in exits(s) \wedge c \in header\_end\_dominators(e, D, IED)\}$
18: **return** $D, (L \cap (E \cup C))$

This traditional algorithm is the basis for our algorithm for finding constant-loop dominators. It can be seen in Algorithm 1 on lines 8-11, where $P$ (the intersection of predecessors) has been edited for our relation. Instead of operating on the CFG, our algorithm operates on the FCFG of the start node ($fcfg(s)$), which is a DAG. Since traditional and constant-loop dominance are equivalent when there are no loops, they are also equivalent between pairs of nodes within the same FCFG. This baseline, therefore, finds the correct constant-loop dominance between nodes in the same FCFG. The rest of the algorithm accounts for dominance between nodes of different loops (nested or in sequence).

In addition to returning the constant-loop dominator sets for each node in the given FCFG, CLDom returns a second, helper set we will call the *end dominators*. The set is calculated on lines 15-18. It represents the set of nodes in the current FCFG that would constant-loop dominate a hypothetical successor node to the FCFG's loop – assuming that node did not have any other predecessors. It is calculated by finding the nodes that constant-loop dominate all latches ($L$) and constant-loop dominate all exits ($E$) or are strictly constant-loop dominated by all exits ($C$). The nodes adhering to these requirements are precisely those that will always be visited a fixed number of times; they either are visited in every iteration ($L \cap E$) or will be skipped in the last iteration only ($L \cap C$). The helper function *header_end_dominators* does the following: If the given node is in the FCFG ($n \in fcfg(s)$), the function returns that node's constant-loop dominators ($D[n]$). Otherwise, the node must be in one of the inner loops. *header_end_dominators* finds the header in the FCFG ($h \in H$) whose loop contains the node; either directly or in a nested loop. If that header is constant, it returns the constant-loop dominators of the header and end dominators of that inner loop ($D[h] \cup IED[h]$). Otherwise, it returns the header's constant-loop dominators alone ($D[h]$).

Our algorithm starts by recursing on the headers of inner loops in the FCFG (lines 5-7) and storing the results for each. During dominator calculation for each FCFG node, we add the end dominators of any constant headers to their dominator sets before intersecting with the other predecessors ($D[a] \cup IED[a]$). This is what ensures that any constant-loop dominators are extracted from inner loops into the dominator sets of the current loop's nodes. Notice that *header_end_dominators* serves the same purpose in the end-dominators calculation.

Lastly, we also need to extract the constant-loop dominators of the nodes of inner loops into the current dominator set (lines 12-14). We give the loops' nodes the dominators of the header in the current dominator set, as those would not have been available in the recursive call (since $fcfg(h)$ does not contain nodes from outside the loop.) For constant loops, we also add the dominator sets of each node from their loop's recursive call ($ID[h][v]$) so they are included in the final result. Not doing so for variable loops ensures that nodes within a variable loop do not dominate anything, not even themselves.

To use CLDom for getting the constant-loop dominators of a function, we call it on the entry node and ignore the end-dominators result. It will always be empty since functions have no latches or exits.

## 4.2 Pseudo-Root Optimization

Single-path code can take advantage of constant-loop dominators to reduce execution times. The optimization focuses on those blocks that constant-loop dominate all end blocks, which means they will always be executed the same number of times per function call. We will refer to these blocks as *constant-loop dominant*.

As described in Section 3, function-conversion makes functions in single-path code take a predicate argument to enable or disable their bodies. However, this is not always necessary. This is most obvious for any single-path *root function*; a function that is itself single-path but is called from a non-single-path function (e.g., the main function.) A root function is guaranteed to be enabled, making the predicate argument unnecessary. The original single-path implementation recognized this and special-cased root functions not to need the predicate argument [21]. This reduces the number of instructions needed for predicate management, which results in reduced execution time.

The optimization of root functions can also be used for other functions. Any function that we can guarantee is always called enabled can be optimized as if it was a root. Taking this further, any function called from a constant-loop dominant block can also be optimized. We can do so because it means we know exactly how many times the function is called from that point, and function-conversion therefore does not need to account for variations in call numbers (as there is no variation). The callee in cases like these is called a *pseudo-root* since its code generation can be identical to a root's. Any function called from a root or pseudo-root in a constant-loop dominant block is also a pseudo-root.

The pseudo-root optimization uses constant-loop dominance to explore the call tree from the root function(s) and identifies all pseudo-root functions. The single-path transformation then uses the information to optimize all pseudo-roots to omit the predicate argument. It also changes all call instructions to pseudo-roots to be predicated, so the functions are not called when a block is disabled. Note that a function may be called from both a constant-loop dominant block and one that does not dominate. E.g., it could be called both in the entry block of a function and within only one side of a branch. In such cases, functions are duplicated, such that two versions are used: one that takes an additional argument and one that does not.

## 4.3 Implementation

We extend the open-source work presented in [21] with implementations of the constant-loop dominance algorithm and the described optimization. Patmos' compiler is based on the LLVM compiler framework [15]. Its frontend, called Clang, produces the LLVM intermediate representation called Bitcode. Bitcode is then compiled by the backend into machine code. The previous work and our extensions all reside in the backend.

We have implemented our algorithm as a `MachineFunctionPass` in the LLVM backend. At this stage, functions are in an intermediate representation close to Patmos machine code. Our algorithm is run on each function and returns a map from their blocks to the set of blocks that constant-loop dominate them. Our CFG does not have a `const` label. Instead, we provide the algorithm a function that, when given a header, returns whether it should be treated as constant. It does so by looking at the loop iteration bounds; if they are equal, the loop must be constant.

Identifying pseudo-roots is done in the `SPMark` pass of the single-path transformation [21]. We update it so that while identifying functions that will be called from a single-path context, it also identifies which calls are coming from a constant-loop dominant block and marks the target functions as pseudo-roots.

The `SPReduce` pass assigns each instruction its predicate. It also removes predication from call instructions and provides the additional predicate argument to functions. When it sees a call instruction in a constant-loop dominant block in a pseudo-root function, it omits the predicate argument, predicates the call instruction, and targets the pseudo-root version of the function (instead of the version that takes a predicate argument.)

# 5 Evaluation

We use a subset of the TACLe benchmark suite [8] to evaluate the effect of enabling the pseudo-root optimization for single-path code. We only include those programs that compile and run correctly for single-path code with and without the optimization. We exclude the `duff` program, as it has no branching and is fully inlined by the compiler, meaning no changes are made to it by the single-path transformation. The `filterbank` program is also excluded because it is so long-running that the simulator we use to run all the programs, Pasim, saturates its cycle counter, meaning we do not know what the execution times are.

## 5.1 Performance

In Figure 3, we show the performance increase (blue bars) of enabling the pseudo-root optimization ($\frac{disabled-enabled}{enabled} \times 100$). First, note how 11 programs see no execution time differences. All these programs – except `huff_dec` and `gsm_dec` – only have one function. This can be seen in the first row of Table 1, where nine programs only have one function with and without our optimization. The second row shows how many functions were recognized as pseudo-roots. For all these functions, including `huff_dec` and `gsm_dec`, only the root was recognized as a pseudo-root, meaning there is nothing to optimize.

Enabling the pseudo-root optimization produces wildly different results for the other programs. In the lower end, `cosf` sees a small performance decrease. Looking at the third row of Table 1, we see that many more instructions are used by single-path code with the optimization (600 → 726). The fourth row also shows an increase in the total number of call instructions (159 → 181), while the fifth row shows that there are very few calls between pseudo-roots (8). This must mean the five pseudo-root functions found did not make up

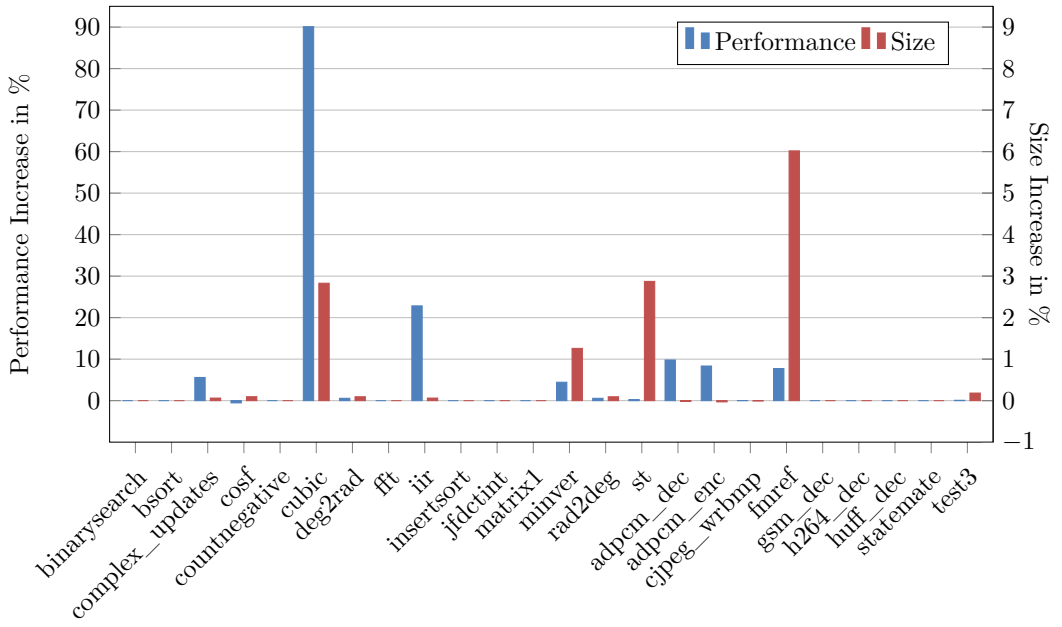



**Figure 3** Performance and code size increase of enabling the pseudo-root optimization for single-path code.

for the increase in code size from duplicating three of them. On the other hand, we have the `cubic` program, which sees a 90 % performance increase. This number is all the more impressive when we look at Table 1. First, notice that the number of functions increases from 33 to 47. Notice also that the number of pseudo-roots found was 17 (including the root). This means that 14 pseudo-roots are also used in a non-pseudo-root context, which means two copies of each original function must be used. The rest of the functions are either only used in a pseudo-root context (3) or in a non-pseudo-root context (16). The additional copies of some functions also translate to an increased total of instructions used for managing the single-path code (627 → 921) and the number of total call instructions (136 → 215), with 58 calls being between pseudo-roots. So from where does all that performance come? The source code shows that the main function is four constant loops nested within each other. The function `cubic_solveCubic` is called four times before the loop and once in each iteration of the inner-most nested loop. Cumulatively, the main function has 879 calls to this function, all from constant-loop dominant blocks. Therefore, recognizing `cubic_solveCubic` exclusively as a pseudo-root likely produces most of this substantial increase in performance.

## 5.2 Code Size

As we have explained earlier and seen in our results so far, using the pseudo-root optimization can increase code size. The first source of this increase is the additional copies of functions used in both pseudo-root contexts and non-pseudo-root contexts. Code size can also be reduced when functions are exclusively pseudo-roots and therefore need fewer instructions for managing predicates and calling other pseudo-roots.

We measure the total size of the final executable of each program with and without the optimization and can see the result in the red bars of Figure 3. We can see that the difference is negligible for most of the programs that were affected by our optimization. For others,

**Table 1** Compiler statistics for each program using single-path code. For each entry, the pseudo-root optimization is disabled for the upper number and enabled for the lower. The metrics given are the total number of functions, the number of pseudo-root (PR) functions, the number of single-path management instructions, the total number of call instructions, and the number of calls between pseudo-roots.

| | binary.. | bsort | complex.. | cosf | countn.. | cubic | deg2rad | fft | iir | insert.. | jfdctint | matrix1 | minver | rad2deg | st | ad..dec | ad..enc | cjpeg.. | fmref | gsm.. | h264.. | huff.. | state.. | test3 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Functions | 1 | 1 | 21 | 30 | 1 | 33 | 27 | 1 | 21 | 1 | 1 | 1 | 33 | 27 | 34 | 3 | 3 | 3 | 72 | 3 | 1 | 2 | 1 | 101 |
| | 1 | 1 | 23 | 33 | 1 | 47 | 30 | 1 | 23 | 1 | 1 | 1 | 36 | 30 | 48 | 3 | 3 | 3 | 103 | 3 | 1 | 2 | 1 | 101 |
| PRs | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| | 1 | 1 | 10 | 6 | 1 | 17 | 12 | 1 | 10 | 1 | 1 | 1 | 4 | 12 | 18 | 3 | 3 | 3 | 36 | 1 | 1 | 1 | 1 | 101 |
| Instructions | 19 | 32 | 302 | 600 | 32 | 627 | 433 | 91 | 303 | 34 | 20 | 40 | 827 | 433 | 546 | 112 | 142 | 71 | 1458 | 389 | 116 | 199 | 47 | 1510 |
| | 19 | 32 | 264 | 726 | 32 | 921 | 378 | 91 | 264 | 34 | 20 | 40 | 922 | 378 | 854 | 96 | 126 | 64 | 2070 | 389 | 116 | 199 | 47 | 1210 |
| Calls | 0 | 0 | 64 | 159 | 0 | 136 | 87 | 0 | 65 | 0 | 0 | 0 | 116 | 87 | 121 | 4 | 4 | 4 | 377 | 8 | 0 | 5 | 0 | 200 |
| | 0 | 0 | 64 | 181 | 0 | 215 | 87 | 0 | 65 | 0 | 0 | 0 | 143 | 87 | 200 | 4 | 4 | 4 | 523 | 8 | 0 | 5 | 0 | 200 |
| PR-Calls | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | 0 | 0 | 21 | 8 | 0 | 58 | 24 | 0 | 22 | 0 | 0 | 0 | 9 | 24 | 46 | 4 | 4 | 4 | 78 | 0 | 0 | 0 | 0 | 200 |

there is a significant increase but none prohibitively so. We can see no correlation between the increase in performance and code size. E.g., while `cubic` sees an enormous performance increase, it only sees a 2.8 % size increase. `fmref`, on the other hand, sees a 6 % size increase for a comparatively modest 7.7 % performance increase.

Lastly, we also need to note that the executables we have measured do not exclusively contain single-path code. They also include all original versions of any single-path function, any initialization code that eventually calls the benchmark function, and the standard library. This means the size differences are likely bigger for both increases and decreases in a real-world, single-path-only scenario.

## 5.3 Source Access

Patmos and its platform, T-CREST [25], are available as open-source and include the contributions of this paper. The Patmos homepage can be found at http://patmos.compute.dtu.dk/ and provides a link to the Patmos Reference Handbook [26], which includes build instructions.

The T-CREST project repositories can be found at https://github.com/t-crest, with the repository for the compiler used in this work at https://github.com/t-crest/patmos-llvm-project (commit hash: 82eb73bff7336674027afecb254f1e3ebd1c23c2).

# 6 Conclusion

In this paper, we presented the constant-loop dominance relation and how it can be used for optimizing single-path code. We first defined the relation as a variation of the traditional dominance where the number of visits to a node must be constant. This takes loop bounds into account to recognize constant loops. We then presented a recursive algorithm for finding the constant-loop dominators. It first explores (nested) loops and uses the intermediate results for the outer loops. We showed how the relation can be used to identify pseudo-root functions in single-path code. These have the quality of being called a fixed number of times. We used this property to optimize single-path code to require fewer instructions to manage predicates and to reduce unnecessary calls. Our evaluation showed sporadic but significant performance improvements from applying our optimization. While some programs saw no execution-time differences, others saw an up to 90 % performance increase. We also showed that the optimizations do affect code size, with executable sizes increasing by up to 6 %.

## References


1 Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. On finding lowest common ancestors in trees. In Alfred V. Aho, Allan Borodin, Robert L. Constable, Robert W. Floyd, Michael A. Harrison, Richard M. Karp, and H. Raymond Strong, editors, *Proceedings of the 5th Annual ACM Symposium on Theory of Computing, April 30 - May 2, 1973, Austin, Texas, USA*, pages 253–265. ACM, 1973. doi:10.1145/800125.804056.

2 Alfred V. Aho and Jeffrey D. Ullman. *Principles of Compiler Design (Addison-Wesley Series in Computer Science and Information Processing)*. Addison-Wesley Longman Publishing Co., Inc., USA, 1977.

3 Stephen Alstrup, Dov Harel, Peter W. Lauridsen, and Mikkel Thorup. Dominators in linear time. *SIAM J. Comput.*, 28(6):2117–2132, 1999. doi:10.1137/S0097539797317263.

4 Armelle Bonenfant, Marianne de Michiel, and Pascal Sainrat. oRange: A tool for static loop bound analysis. In *Proceedings of the Workshop on Resource Analysis*, volume 42, 2008.

5 Adam L. Buchsbaum, Loukas Georgiadis, Haim Kaplan, Anne Rogers, Robert Endre Tarjan, and Jeffery R. Westbrook. Linear-time algorithms for dominators and other path-evaluation problems. *SIAM J. Comput.*, 38(4):1533–1573, 2008. doi:10.1137/070693217.

6 Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, 1991. doi:10.1145/115372.115320.

7 Andreas Ermedahl, Christer Sandberg, Jan Gustafsson, Stefan Bygde, and Björn Lisper. Loop bound analysis based on a combination of program slicing, abstract interpretation, and invariant analysis. In Christine Rochange, editor, *7th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis, Pisa, Italy, July 3, 2007*, volume 6 of *OASIcs*. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2007. URL: http://drops.dagstuhl.de/opus/volltexte/2007/1194.

8 Heiko Falk, Sebastian Altmeyer, Peter Hellinckx, Björn Lisper, Wolfgang Puffitsch, Christine Rochange, Martin Schoeberl, Rasmus Bo Sørensen, Peter Wägemann, and Simon Wegener. TACLeBench: A benchmark collection to support worst-case execution time research. In Martin Schoeberl, editor, *16th International Workshop on Worst-Case Execution Time Analysis, WCET 2016, July 5, 2016, Toulouse, France*, volume 55 of *OASIcs*, pages 2:1–2:10. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016. doi:10.4230/OASIcs.WCET.2016.2.

9 Dov Harel. A linear time algorithm for finding dominators in flow graphs and related problems. In Robert Sedgewick, editor, *Proceedings of the 17th Annual ACM Symposium on Theory of Computing, May 6-8, 1985, Providence, Rhode Island, USA*, pages 185–194. ACM, 1985. doi:10.1145/22145.22166.

10 Rebecca Hasti and Susan Horwitz. Using static single assignment form to improve flow-insensitive pointer analysis. In Jack W. Davidson, Keith D. Cooper, and A. Michael Berman, editors, *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI), Montreal, Canada, June 17-19, 1998*, pages 97–105. ACM, 1998. doi:10.1145/277650.277668.

11 Christopher A. Healy, Mikael Sjödin, Viresh Rustagi, David B. Whalley, and Robert van Engelen. Supporting timing analysis by automatic bounding of loop iterations. *Real Time Syst.*, 18(2/3):129–156, 2000. doi:10.1023/A:1008189014032.

12 Paul Walton Purdom Jr. and Edward F. Moore. Immediate predominators in a directed graph [H] (algorithm 430). *Commun. ACM*, 15(8):777–778, 1972. doi:10.1145/361532.361566.

13 Dimitar Kazakov and Iain Bate. Towards new methods for developing real-time systems: Automatically deriving loop bounds using machine learning. In *Proceedings of 11th IEEE International Conference on Emerging Technologies and Factory Automation, ETFA 2006, September 20-22, 2006, Diplomat Hotel Prague, Czech Republic*, pages 421–428. IEEE, 2006. doi:10.1109/ETFA.2006.355425.

14 Raimund Kirner, Jens Knoop, Adrian Prantl, Markus Schordan, and Albrecht Kadlec. Beyond loop bounds: comparing annotation languages for worst-case execution time analysis. *Softw. Syst. Model.*, 10(3):411–437, 2011. doi:10.1007/s10270-010-0161-0.

15 Chris Lattner and Vikram S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*, pages 75–88. IEEE Computer Society, 2004. doi:10.1109/CGO.2004.1281665.

16 Thomas Lengauer and Robert Endre Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Trans. Program. Lang. Syst.*, 1(1):121–141, 1979. doi:10.1145/357062.357071.

17 Edward S. Lowry and C. W. Medlock. Object code optimization. *Commun. ACM*, 12(1):13–22, 1969. doi:10.1145/362835.362838.

18 Salvador Lucas. The origins of the halting problem. *J. Log. Algebraic Methods Program.*, 121:100687, 2021. doi:10.1016/j.jlamp.2021.100687.

19 Emad J. Maroun, Martin Schoeberl, and Peter Puschner. Compiler-directed constant execution time on flat memory systems. In *2023 IEEE 26th International Symposium on Real-Time Distributed Computing (ISORC)*. IEEE, 2023.

20 Emad J. Maroun, Martin Schoeberl, and Peter P. Puschner. Compiling for time-predictability with dual-issue single-path code. *J. Syst. Archit.*, 118:102230, 2021. doi:10.1016/j.sysarc.2021.102230.

21 Daniel Prokesch, Stefan Hepp, and Peter P. Puschner. A generator for time-predictable code. In *IEEE 18th International Symposium on Real-Time Distributed Computing, ISORC 2015, Auckland, New Zealand, 13-17 April, 2015*, pages 27–34. IEEE Computer Society, 2015. doi:10.1109/ISORC.2015.40.

22 Peter P. Puschner and Alan Burns. Writing temporally predictable code. In *7th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2002), 7-9 January 2002, San Diego, CA, USA*, pages 85–94. IEEE Computer Society, 2002. doi:10.1109/WORDS.2002.1000040.

23 Ola Redell and Martin Sanfridson. Exact best-case response time analysis of fixed priority scheduled tasks. In *14th Euromicro Conference on Real-Time Systems (ECRTS 2002), 19-21 June 2002, Vienna, Austria, Proceedings*, pages 165–172. IEEE Computer Society, 2002. doi:10.1109/EMRTS.2002.1019196.

24 Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Global value numbers and redundant computations. In Jeanne Ferrante and Peter Mager, editors, *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego, California, USA, January 10-13, 1988*, pages 12–27. ACM Press, 1988. doi:10.1145/73560.73562.

25 Martin Schoeberl, Sahar Abbaspour, Benny Akesson, Neil C. Audsley, Raffaele Capasso, Jamie Garside, Kees Goossens, Sven Goossens, Scott Hansen, Reinhold Heckmann, Stefan Hepp, Benedikt Huber, Alexander Jordan, Evangelia Kasapaki, Jens Knoop, Yonghui Li, Daniel Prokesch, Wolfgang Puffitsch, Peter Puschner, André Rocha, Cláudio Silva, Jens Sparsø, and Alessandro Tocchi. T-CREST: time-predictable multi-core architecture for embedded systems. *J. Syst. Archit.*, 61(9):449–471, 2015. doi:10.1016/j.sysarc.2015.04.002.

26 Martin Schoeberl, Florian Brandner, Stefan Hepp, Wolfgang Puffitsch, and Daniel Prokesch. Patmos reference handbook. Technical report, Technical University of Denmark, 2014.

27 Martin Schoeberl, Wolfgang Puffitsch, Stefan Hepp, Benedikt Huber, and Daniel Prokesch. Patmos: a time-predictable microprocessor. *Real Time Syst.*, 54(2):389–423, 2018. doi:10.1007/s11241-018-9300-4.

28 Thomas Sewell, Felix Kam, and Gernot Heiser. Complete, high-assurance determination of loop bounds and infeasible paths for WCET analysis. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), Vienna, Austria, April 11-14, 2016*, pages 185–195. IEEE Computer Society, 2016. doi:10.1109/RTAS.2016.7461326.

**29** Robert Endre Tarjan. Finding dominators in directed graphs. *SIAM J. Comput.*, 3(1):62–89, 1974. doi:10.1137/0203006.

**30** Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Trans. Program. Lang. Syst.*, 13(2):181–210, 1991. doi:10.1145/103135.103136.