Online Iterative Compilation Guided by Work-based Profiling

Rodrigo Caetano de Oliveira Rocha

Master of Science by Research
Institute of Computing Systems Architecture
School of Informatics
University of Edinburgh
2017

Abstract

Acknowledgements

This work was supported in part by the EPSRC Centre for Doctoral Training in Pervasive Parallelism, funded by the UK Engineering and Physical Sciences Research Council (grant EP/L01503X/1) and the University of Edinburgh.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Rodrigo Caetano de Oliveira Rocha*)

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Modern optimising compilers have reached a high level of sophistication, providing a large number of optimisations. Because of the unpredictability of optimisation interactions, in addition to the growing complexity of processor architectures and applications, the correct choice of optimisations and their ordering can have a significant impact on the performance of the code being optimised (Pan and Eigenmann, 2006; Fursin et al., 2007; Kulkarni and Cavazos, 2012).

Each of these optimisations interacts with the code in complicated ways, depending on all other optimisations and the order they were applied to the code being optimised. Understanding the interactions of optimisations is very important in determining a good solution to the phase-ordering problem (Touati and Barthou, 2006; Kulkarni and Cavazos, 2012). Because of all those dependencies and interactions, although most compiler optimisations yield significant improvements in many programs, the potential for performance degradation in certain program patterns is known to compiler writers and many users (Pan and Eigenmann, 2006; Zhou and Lin, 2012; Kulkarni and Cavazos, 2012).

Efficiently selecting the best combination of compiler optimisations for a given program or program section remains an open problem. Compiler writers typically use a combination of experience and insight to construct the default sequences of prearranged optimisations found in compilers. However, these default pre-arranged set of optimisations do not include all possible optimisations and they are always applied in the same pre-defined fixed order, without regard the code being optimised (Pan and Eigenmann, 2006; Cavazos et al., 2007; Zhou and Lin, 2012; Kulkarni and Cavazos, 2012).

A well known compilation technique that addresses this challenge is iterative com-

pilation. Iterative compilation has the ability to adapt to new platforms, program and workload while still having a systematic and simple optimisation process. It works by repeatedly evaluating a large number of compiler optimisations until the best combination is found for a particular program (Fursin et al., 2007; Chen et al., 2010). The main challenge concerning iterative compilation is the need for efficiently exploring such a large optimisation space (Fursin et al., 2007; Cavazos et al., 2007; Zhou and Lin, 2012).

Until recently, most of existing work had been focusing on finding the best optimisation through repeated runs using a single input. Although they demonstrate the potential of iterative compilation, in real scenarios the user rarely executes the same input dataset multiple times (Bodin et al., 1998; Kisuki et al., 1999; Stephenson et al., 2003; Kulkarni et al., 2004; Agakov et al., 2006). Applying iterative compilation in light of a single input may not result in good performance when executing the optimised code with different inputs.

Most of real world applications are complex enough so that a single input case does not capture the whole range of possible scenarios and program behaviour (Haneda et al., 2006; Fursin et al., 2007; Chen et al., 2010, 2012b). Because programs can exhibit behaviours that differ greatly depending on the input, using a single input for iterative compilation can produce poor performance when executed with different inputs.

Recent work have been studying the impact of using multiple input datasets for performing iterative compilation. This previous work suggests that optimising based on a representative range of input datasets allows for selecting a robust compiler optimisation that produces good performance in real scenarios where the input is expected to change constantly (Haneda et al., 2006; Fursin et al., 2007; Chen et al., 2010, 2012b,a; Fang et al., 2015; Mpeis et al., 2016). Their results show that a limited number of input datasets may be sufficient to obtain a well optimised program for a wider range of different inputs (Haneda et al., 2006; Fursin et al., 2007; Chen et al., 2010, 2012b).

Finding such a robust combination of compiler optimisations by means of iterative compilation across a large range of inputs may be very time consuming. In order to speed up this process we intend to reduce the number of actual executions during the exploration of the optimisation space without much degradation of the final selected optimisation.

In this work, our main goal is to enable iterative compilation in *online* scenarios with the restriction of executing distinct inputs only once, while targeting optimal per-

formance across different inputs. This online aspect is usually found in mobile and data centre platforms (Chen et al., 2012a; Fang et al., 2015; Mpeis et al., 2016), where the goal is to optimise programs based on the workload of a particular user (or group of users) between executions.

Measuring just execution time, for example, is useful only if the amount of work is constant between executions with different inputs. For this reason, we propose the use of a work-based metric in order to compare the performance of different optimisations across multiple executions of the program with distinct inputs.

Because of the restriction of not repeating inputs, we instrument the program for measuring the amount of work it performed during its execution. Having a low overhead instrumentation is essential in this online scenario for two main reasons: (i) the user is directly affected by large overheads; (ii) a highly intrusive instrumentation can have a significant impact on the effect of the optimisations.

Our main contributions are the following:

- The use of a work-based performance metric in order to enable *online* iterative compilation by comparing different combination of compiler optimisations even when executed with distinct inputs.

- We propose a relaxed instrumentation for low overhead profiling, with a controlled trade-off between accuracy and overhead.

Chapter 2

Background

2.1 Basic Concepts of Graph Theory

2.2 LLVM Compiler Infrastructure

LLVM was originally proposed as a *Low-Level Virtual Machine*[1], extending previous work on virtual instruction set architectures (Adve et al., 2003; Lattner and Adve, 2004). Since then, LLVM has evolved into an umbrella project that comprises a collection of modular and reusable compiler and toolchain technologies. The main components under the LLVM umbrella is the LLVM intermediate representation (IR) and the LLVM *Core* libraries.

The LLVM compiler infrastructure implements the classical *three-phase compiler* infrastructure, which consists of a frontend, an optimiser, and a backend, as depicted in Figure 2.1. The frontend is responsible for parsing, validating and diagnosing errors in the source code. This parsed source code is then translated into an intermediate representation, which is the LLVM IR in this case. The optimiser is responsible for doing a broad variety of transformations, that are usually independent of language and target machine, to improve the code's performance. The backend, also known as the code generator, then translates the code from the intermediate representation onto the target instruction set. It is common for the backend to also perform some low-level optimisations that take advantage of unusual features of the supported architecture.

[1] Although LLVM was initially an acronym for Low-Level Virtual Machine, it is now a brand that applies to the whole LLVM umbrella project.

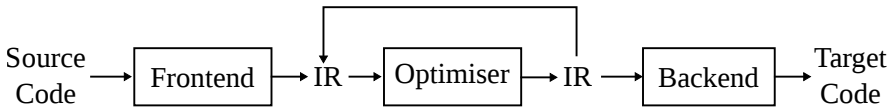


Figure 2.1: Overview of the three-phase compiler infrastructure.

2.2.1 LLVM Virtual Instruction Set

LLVM IR is a low-level RISC-like virtual instruction set. It differs from other intermediate representations (e.g. GCC's GENERICS or the most recent GCC's GIMPLE) as it is defined as a first class language with well-defined semantics. Beyond being implemented as a language, LLVM IR is actually defined in three isomorphic forms: the textual format above, an in-memory data structure inspected and modified by optimizations themselves, and an efficient and dense on-disk binary *bitcode* format.

Unlike most RISC instruction sets, LLVM is strongly typed with a simple language-independent type system. LLVM's type system can be used to implement data types and operations from high-level languages exposing their implementation behaviour to all stages of optimisation. This type system includes the type information used by sophisticated techniques, such as algorithms for pointer analysis, dependence analysis, and data transformations. LLVM also offers instructions for performing type conversions and low-level address arithmetic while preserving type information. Furthermore, LLVM IR also differs from RISC instruction sets as some details of the machine are abstracted away. For example, the calling convention is abstracted through call and ret instructions and explicit arguments. Another significant difference from machine code is that the LLVM IR has an infinite set of virtual registers (which are named with a % character), insted of having just a fixed set of named registers.

The LLVM type system is considered one of the most important features of its intermediate representation, as it enables several optimisations to be performed directly on the IR, without having to do extra analyses on the side before the transformation. The main first class types supported are: single value types, aggregate types, and labels. Single value types consist of integers of arbitrary bit width (e.g. `i32` denotes a 32-bit integer), floating-point of commonly used width (e.g. `half`, `float`, `double`, and `fp128`), pointers (e.g. `i32*`) and vector types. Vectors are used when multiple primitive data are operated in parallel using a single instruction (SIMD). A vector type is represented by a the number of elements and an underlying primitive data type, e.g. `<4 x i32>` is a vector of four 32-bit integer values. Aggregated types consist of arrays and structures. Vectors are not considered to be aggregate types.

In addition to type information, LLVM IR also provides other high-level information that are useful for effectively performing several code analysis and transformations. This includes explicit control flow graphs (CFG) (Allen, 1970) and an explicit dataflow representation, by means of the infinite register set in *static single assignment* (SSA) form (Alpern et al., 1988; Cytron et al., 1989, 1991).

A control flow graph (CFG) is a directed graph in which the nodes represent basic blocks and the edges represent control flow paths, i.e. edges represent transfers of control (jumps) between basic blocks. A basic block is a straight-line sequence of instructions having only one entry point, i.e. the first instruction to be executed in the basic block, and only one exit point, i.e. the last instruction executed (Allen, 1970; Cytron et al., 1991). Figure 2.2 shows an example of a CFG constructed from the code in Listing 2.1.

An IR is in SSA form if and only if each virtual register is assigned exactly once and every use of registers occur after their definition. The primary advantage of using the SSA form is that it simultaneously simplifies and improves several compiler optimisations and analysis (Alpern et al., 1988; Cytron et al., 1991). Most of the industrial-strength compilers for imperative programming language rely heavily on the SSA form.

```
1  define i32 @odd_inc(i32 %arg) {
2  entry:
3    %rem = srem i32 %arg, 2
4    %cmp = icmp eq i32 %rem, 0
5    br i1 %cmp, label %if.then, label %if.else
6  if.then:
7    %add.1 = add i32 %arg, 1
8    br label %if.end
9  if.else:
10   %add.2 = add i32 %arg, 2
11   br label %if.end
12 if.end:
13   %ans = phi i32 [ %add.1, %if.then ], [ %add.2, %if.else ]
14   ret i32 %ans
15 }
```

Listing 2.1: An illustrative example of a function in textual LLVM IR. This function returns the argument incremented by one if it is even or by two if it is an odd integer.

Listing 2.1 shows an example of a function written using textual LLVM IR. Names starting with the @ character have a global scope, while names starting with % have a local scope. As stated previously, LLVM IR is strongly typed, which means that every virtual register is attributed a specific type (e.g. `i32 %arg` is of type `i32`, namely a 32-bit integer) as well as every operation (e.g. `add i32` expects all operands of type `i32`). Instructions use the three address format, which refers to the use of three operands by

most of the instructions. However, instructions with fewer or more operands may occur, e.g. the `ret` instruction have fewer operands, while the `phi` and the `call` instructions may have more than three operands.

Furthermore, the SSA form requires a special assignment statements called the ϕ-function (see line 13 of Listing 2.1). The ϕ-function receives as argument a list of virtual registers from different control flow predecessors of the point where the ϕ-function occurs. The control flow predecessors of each point in the CFG are listed in some arbitrary fixed order, and the i-th operand of ϕ-function is associated with the i-th predecessor. If control reaches the ϕ-function from its i-th predecessor, then the value of the i-th operand is attributed in the assignment. Each execution of a ϕ-function uses only one of the operands, but which one depends on the flow of control just before the ϕ-function.

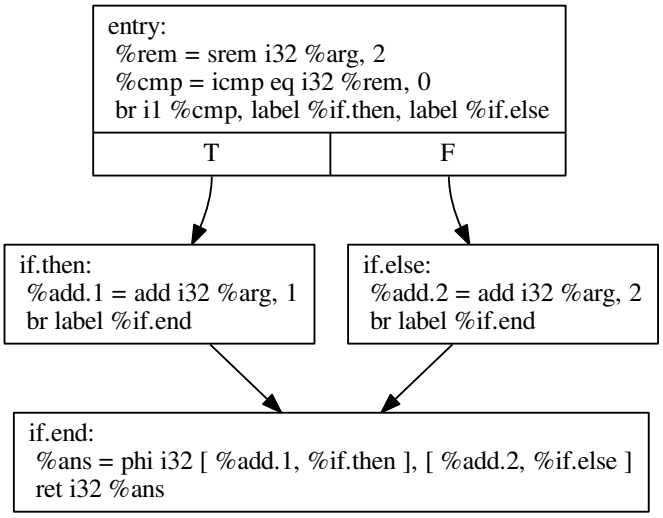


Figure 2.2: Control flow graph of the `odd_inc` function (see Listing 2.1).

In addition to virtual registers, LLVM also allows for stack allocated local variables. These are created by allocating data on the stack frame of the currently executing function. Data from the stack frame can be manipulated by using explicit memory access operations. Figure 2.2 shows an implementation of the `odd_inc` function using data allocated on the stack frame. This implementation avoids using the ϕ-function by keeping the answer on the stack. This is the preferred way of generating LLVM code by most frontends, as it simplifies the frontend's own implementation without

incurring any serious detriment to the generated code.

LLVM provides an optimisation pass for promoting memory references to be register references (called `-mem2reg`). It promotes alloca instructions which only have loads and stores as uses. An alloca is transformed by using dominator frontiers to place phi nodes, then traversing the function in depth-first order to rewrite loads and stores as appropriate.

```
1  define i32 @odd_inc_stack(i32 %arg) {
2  entry:
3    %addr = alloca i32
4    %rem = srem i32 %arg, 2
5    %cmp = icmp eq i32 %rem, 0
6    br i1 %cmp, label %if.then, label %if.else
7  if.then:
8    %add.1 = add i32 %arg, 1
9    store i32 %add.1, i32* %addr
10   br label %if.end
11 if.else:
12   %add.2 = add i32 %arg, 2
13   store i32 %add.2, i32* %addr
14   br label %if.end
15 if.end:
16   %ans = load i32, i32* %addr
17   ret i32 %ans
18 }
```

Listing 2.2: An example of the odd_inc function implemented using data allocated on the stack frame.

2.2.2 Analysis and Transformation Passes

The LLVM optimiser offers several of different passes in order to provide analysis and transformation capabilities. These passes are written using the LLVM Core libraries and they are as loosely coupled as possible. In other words, each pass is either a stand-alone pass or it explicitly declares its dependencies among other passes, in the case where it depend on some other analysis.

The LLVM Core libraries provide both analysis and transformation passes for different levels of the input program. These levels, in hierarchical order, are: module, call-graph SCC, function, loop, single-entry single-exit region (or only region for short), and basic block. A module is an entire *translation unit* (e.g., a C/C++ file with all its headers included). A pass level only allows modification inside the component in focus. For example: while a module-level pass can operate on the entire module at once, a function-level pass prohibits any modification on the module-level (or other functions).

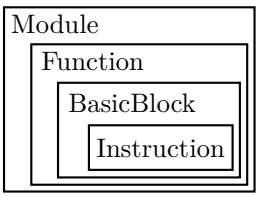


Figure 2.3: The hierarchical levels in an LLVM input program. A call-graph SCC is a particular pattern amongst the functions in a module. Similarly, a loop and a region are particular patterns on the CFG of a function.

The LLVM Pass Manager is responsible for scheduling the passes to be executed efficiently. It also makes sure that the interactions among the passes are correctly fulfilled. For that purpose, when given a series of passes to execute, the Pass Manager uses the explicit dependency information to satisfy these dependencies and optimise the execution of passes.

A pass specifies its dependency on other passes, i.e., which other passes need to be executed before its own execution, Moreover, it can also specify the passes that will be invalidated by its execution.

The Pass Manager aims at avoiding to repeat the execution of analysis passes. It keeps track of which analyses are already available, which are invalidated, and which analyses are pending. It also tracks the lifetimes of the analysis results and frees memory of the analysis results, when appropriate, managing memory usage.

The Pass Manager pipelines the passes together to get a better memory and cache usage, improving the overall cache behaviour of the compiler. When performing a series of function-level passes, it executes all these passes in only one function, before moving to the next function, in order to improve cache usage.

2.3 Early Work in Iterative Compilation

In this section we present the basic concepts of iterative compilation and discuss some of the early work on this research topic. We also consider some of the challenges on reducing its compilation time and present recent work that addresses these challenges.

Iterative compilation is a well known compilation technique that searches the optimisation space in order to find the best optimisation for a particular program. Iterative compilation has the ability to adapt to new platforms, program and workload while still having a systematic and simple optimisation process. It works by repeatedly evaluating a large number of compiler optimisations, by means of an execution-driven search,

until the best optimisation is found for a particular program (Kisuki et al., 1999; Fursin et al., 2007; Chen et al., 2010).

Figure 2.4 shows an overview of the architecture necessary for performing iterative compilation. After executing an optimised version of the program, profiling data is provided as a feed-back to the iterative search. This profiling data can be as simple as the program's execution time, or more complex, including hardware performance counters for cache behaviour, measurements of energy consumption, etc. The evaluator is able to use the feed-back in order to rank the optimisations and select the best one. The optimisation generator can be a fixed sequence of optimisations or a dynamic mechanism for suggesting the next optimisation.

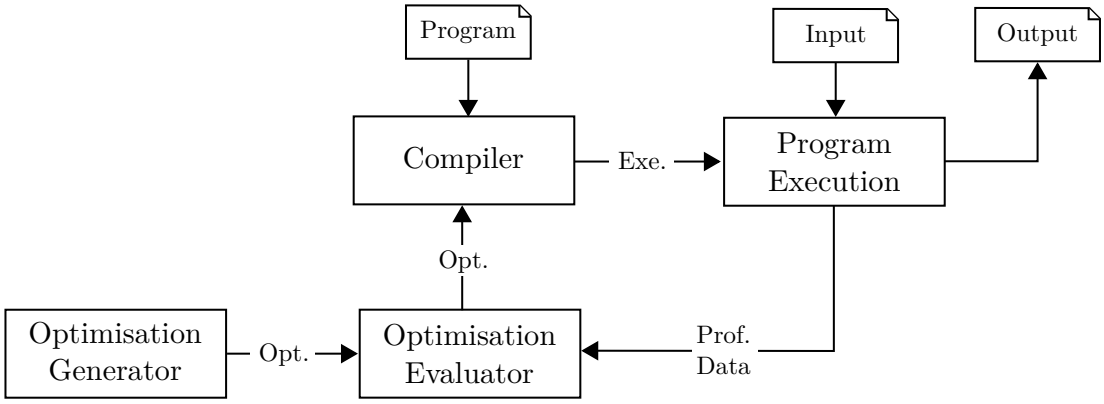


Figure 2.4: A simplified overview of the architecture of an iterative compiler.

Early work on iterative compilation demonstrated its use for determining simultaneously optimal tile sizes and unroll factors for any given loop nest Kisuki et al. (2000); Knijnenburg et al. (2004). These two transformations are highly interdependent with a very irregular optimisation space, as both of them affect, in different ways, cache behaviour and instruction-level parallelism. Because of their close interaction, with non-trivial trade-offs, designing a static cost model, that enables the compiler to automatically select the best configuration, is a very laborious and impractical task. Kisuki et al. (2000) propose the use of iterative compilation to address this problem, where it is able to outperform several static techniques. However, its success comes at the cost of a significant increase in compilation time, which involves several runs of the program for the execution-driven search.

Despite the high cost of compilation time, there are scenarios where this approach is highly attractive due to high-performance requirements, such as embedded systems and library codes. Moreover, iterative compilation was originally intended to be applied in an *offline* scenario, where the software vendor optimises the program

before shipping, the compilation time can be amortised across the number of products shipped, the lifetime of the product, or a much larger number of executions in production Kisuki et al. (1999, 2000); Chen et al. (2010). It is also useful in contexts where the underlying architecture changes frequently, as the iterative search dismisses the arduous task of manually optimising the program for the new platform.

Numerous researchers have addressed the problem of reducing the optimisation space. For the same problem of selecting optimal tile sizes and unroll factors, Knijnenburg et al. (2004) suggest the use of a cache model to avoid executing candidates during the execution-driven search of iterative compilation. By querying the cache model, the compiler is able to rank the optimisation candidates, filtering out candidates below a given threshold. Their results show that it is possible to reduce the number of executions by up to about 70%, without a significant degradation of the resulting optimisation.

Agakov et al. (2006) suggest using machine learning techniques to speed up iterative compilation. They propose a mechanism that learns a predictive model from a training set of benchmarks. This predictive model will later be used for predicting regions of the optimisation space that are more likely to contain promissing results. Their approach is able to significantly reduce the number of executions necessary for achieving good performance improvements with the iterative search.

More recently, Ogilvie et al. (2017)...

2.4 Program Profiling

Basic background about profiling in general

Progrma profiling concerns with the acquisition of dynamic information collected during the program's execution. Progrma profiling is particularly useful for providing

2.5 0-1 Knapsack Problem

The *0-1 knapsack problem* is a well known *NP-hard* problem in combinatorial optimisation Martello et al. (2000). For a given set of items, where each item has a profit and a weight, the problem consists of selecting a sub-set of the items, such that the total weight of the sub-set does not exceed a pre-defined maximum capacity and whose total profit is a maximum. Although the general knapsack problem allows to repeatedly

pack the same item, in the 0-1 knapsack problem each item can only be selected once, always considering the maximum capacity.

Because it is an NP-hard problem, there is no algorithm with polynomial-time complexity on all cases. There are many heuristics for efficiently selecting the subset of item while trying and maximise the total profit. For example, a *greedy* heuristic tries to maximise the total profit by sorting all items based on their profit-weight ratio and then eagerly selecting all items allowed by the maximum capacity as it iterates over the sorted items(Dantzig, 1957). Although heuristics are not guaranteed to find the best solution, they usually tend to efficiently find good or close to optimal solutions.

2.6 Summary

Chapter 3

Related Work

3.1 Iterative Compilation

Until recently, most of existing work had been focusing on finding the best optimisation through repeated runs using a single input. Although they demonstrate the potential of iterative compilation, in real scenarios the user rarely executes the same input dataset multiple times (Bodin et al., 1998; Kisuki et al., 1999; Stephenson et al., 2003; Kulkarni et al., 2004; Agakov et al., 2006). Applying iterative compilation in light of a single input may not result in good performance when executing the optimised code with different inputs.

Most of real world applications are complex enough so that a single input case does not capture the whole range of possible scenarios and program behaviour (Haneda et al., 2006; Fursin et al., 2007; Chen et al., 2010, 2012b). Because programs can exhibit behaviours that differ greatly depending on the input, using a single input for iterative compilation can produce poor performance when executed with different inputs.

For this reason, researchers have been studying the impact of using multiple input datasets for performing iterative compilation. Chen et al. (2010, 2012b) evaluate the effectiveness of iterative compilations across a large number of input test cases. Their main motivation is to answer the question: *How data input dependent is iterative compilation?* When selecting the best optimisation sequence for each input of each program, these optimal optimisation sequences are program-specific and yield average speedups up to 3.75× over the highest optimisation level of compilers, namely `-O3`. Their results show that, for all the evaluated benchmarks, it is possible to find an optimisation sequence that achieves at least 83% of the maximum speedup across all input

test cases. In other words, although the best optimisation sequences are both program and input-dependent, it is possible to find a program-specific optimisation sequence that achieves good performance on average.

When optimising a program, the main method for iterative compilation used by Chen et al. (2010, 2012b) evaluates each combination of compiler optimisations across all the available inputs, i.e., if N is the number of input test cases and M is the total number of combinations of compiler optimisations, they perform a total of $O(NM)$ runs of the program being optimised. Furthermore, they use a pre-defined set of only 300 different combinations of compiler optimisations, which represents a very small sample of the optimisation search space for most modern compilers, e.g. LLVM has 56 distinct optimisation passes and GCC has about 47 high-level (SSA form) optimisation passes and about 25 low-level (RTL) optimisation passes, which in both cases result in much more than 2^{50} distinct combinations of compiler optimisations, without considering repetition.

Some recent work (Chen et al., 2012a; Fang et al., 2015) have applied the same idea of performing input-dependent iterative compilation to distributed applications on data centres. In summary, each worker receives a subset of the input dataset, called the evaluation dataset, to perform an *online* iterative compilation of the code being executed. Each worker performs the same the method for iterative compilation used by Chen et al. (2010, 2012b), i.e., they evaluate each combination of compiler optimisations across all the evaluation dataset. However, because the optimisation is performed online, they usually consider a small evaluation dataset and a small number of compiler optimisations.

Fursin et al. (2007) addressed the problem of comparing the effect of two optimisations on two distinct inputs. For that purpose, they proposed to use instruction per cycle (IPC) as the metric for performing such comparison. Their result show that using IPC seems promising as a robust metric for iterative compilation across large input datasets. However, some specific optimisation techniques may affect the use of IPC as a robust metric, and specially IPC has been shown to provide poor performance estimation for multi-threaded programs (Alameldeen and Wood, 2006; Eyerman and Eeckhout, 2008). In particular, IPC can give a skewed performance measure if threads spend time in *spin-lock loops* or other synchronisation mechanisms. Some existing work on performance assessment suggest that total execution time should be used for measuring performance of multi-threaded programs (Alameldeen and Wood, 2006; Eyerman and Eeckhout, 2008). Alameldeen and Wood (2006) in particular suggest

that a simple work-related metric should be used if the unit of work is representative enough. Work-related metrics have already been largely used for measuring performance of throughput-oriented applications, for other applications, however, choosing an appropriate unit of work can be more challenging (Alameldeen and Wood, 2006).

3.2 Work and Input Size Metrics

Talk about work-based metrics

Previous work have proposed profiling-based mechanism to estimate input sizes (Zaparanuks and Hauswirth, 2012; Coppa et al., 2014). Coppa et al. (2014) in particular propose the concept of *read memory size* for automatically estimating the size of the input passed to a routine, where *read memory size* represents the number of distinct memory cells first accessed by a read operation. In other words, the *read memory size* metric measures the size of the useful portion of the input's memory footprint. However, because we are interested in the amount of computational work performed in respect of a given input, the memory footprint of the input may not always have a direct correspondence to the amount of computational work.

Goldsmith et al. (2007) use *block frequency* as the measure for performance for empirically describing the asymptotic behaviour of programs, which is known as empirical computational complexity. Block frequency is a relative metric that represents the number of times a basic block executes (Ball and Larus, 1994, 1996). They argue in favour of block frequency due to its portability, repeatability and exactness, since it does not suffer from timer resolution problems or non-deterministic noises. Block frequency also has the advantage of being efficiently profiled by means of automatic code instrumentation (Knuth and Stevenson, 1973; Ball and Larus, 1994).

However, in the context of comparing different optimisations, although block frequency would be able to capture aspects of optimisations that simplify the control-flow graph (CFG), measuring work at the basic block resolution would not capture effects of optimisations at the instruction level. Because of that, we extend the idea of using basic block frequency to measure computational work by also considering the computational cost of each basic block. The computational cost of a basic block is given by weighting the instructions that it contains.

3.3 Optimal Instrumentation

In order to profile block frequency, the program can be instrumented with counters that determine how many times each basic block in a program executes. A naive instrumentation would consist basically of having a counter for each basic block which is incremented every time the basic block is reached. Although the naive instrumentation was commonly used in practice (Knuth, 1971), it is a very invasive instrumentation that imposes an unnecessarily high overhead in the instrumented program. An optimal instrumentation based on the principle of *conservation of flow* (*Kirchhoff's first law*[1]) have been originally proposed by Nahapetian (1973) and Knuth and Stevenson (1973). While Knuth and Stevenson (1973) proposed an optimal solution for basic block profiling with *vertex counters*, Ball and Larus (1994) showed that an optimal basic block profiling with *edge counters* provides the best instrumentation for block frequency profiling. Further overhead reduction of the optimal instrumentation was later proposed by placing the counters in edges that are less likely to be executed Forman (1981); Ball and Larus (1994).

Definition 3.3.1 (Kirchhoff's first law)**.** The amount of flow into a vertex equals the amount of output flow, i.e. the sum of the incoming edges of a vertex equals the sum of outgoing edges of the same vertex.

The optimal instrumentation places probes in edges as the basic block frequency can be derived by summing either the flow of the incoming or outgoing edges. However, it uses the Kirchhoff's first law in order to place probes in subset of the edges that allows to later infer the flow of all edges. Previous work have shown that a set of edges represents the minimum number of probes for profiling block frequency if and only if the complementary set of edges forms a spanning tree (Nahapetian, 1973; Ball and Larus, 1994). In other words, after determining a spanning tree of the CFG, probes need to be placed only in the edges from the complement of a spanning tree, usually called *cotree*. Because the edge frequencies satisfy Kirchhoff's first law, each edge flow can be uniquely determined as an algebraic sum of the known edge flows from the cotree (Nahapetian, 1973; Ball and Larus, 1994).

The optimal block frequency instrumentation happens in two main stages: *(i.) Before execution.* The code is instrumented with the edge counters, i.e., it requires one global counter for each edge selected to contain a probe. *(ii.) After execution.* The

[1]Gustav Kirchhoff defined two equalities about electric circuits, known as Kirchhoff's circuit laws. The first one is about current and and the second about potential difference.

```
1 // Inputs: CFG with the known edges flows from the cotree (collected probes).
2 // Output: Updated CFG with all edge flows.
3 populateEdgeFlows(G) {
4   changed = true
5   while changed:
6     changed = false
7     for B in G.vertices():
8       unIN = count( G.unknownIncomingEdges(B) )
9       unOUT = count( G.unknownOutgoingEdges(B) )
10      if unIN==0 and unOUT==1:
11        //sum known incoming and outgoing edges in B
12        sIN = sum( G.incomingEdges(B) )
13        sOUT = sum( G.outgoingEdges(B) )
14        //update unknown outgoing edge in B with (sIN-sOUT)
15        G.setUnknownOutgoingEdge(B, (sIN-sOUT))
16        changed = true
17      if unIN==1 and unOUT==0:
18        //sum known incoming and outgoing edges in B
19        sIN = sum( G.incomingEdges(B) )
20        sOUT = sum( G.outgoingEdges(B) )
21        //update unknown incoming edge in B with (sOUT-sIN)
22        G.setUnknownIncomingEdge(B, (sOUT-sIN))
23        changed = true
24 }
```

Listing 3.1: Post-processing of the CFG for populating all edge flows based on the collected probes.

information from the recorded probes is propagated in the CFGs of the program, in a post-processing phase. Listing 3.1 shows the algorithm for the post-processing of a CFG, once we have the profiling information collected by the probes.

Listing 3.1 is guaranteed to terminate because the probed edge flows on the complement of a spanning tree are necessary and sufficient to compute all edge flows (Nahapetian, 1973; Forman, 1981). Intuitively, if all the edge flows are known for the complement of a spanning tree then at any leaf of the spanning tree there is only one unknown edge flow. This unknown edge flow can be calculated by Kirchhoff's first law. This process repeats until all the unknown edge flows have been calculated. Although this instrumentation algorithm is proved to produce the optimal placement of probes for well-structured CFGs, it may produce sub-optimal placement for some unstructured CFGs (Ball and Larus, 1994).

This briefly described proof suggests that the edges can be more efficiently populated by a bottom-up propagation in the spanning tree. By performing a post-order traversal of the spanning tree, i.e. starting from the leaves, we can then apply the flow equation from the Kirchhoff's first law. At each node of the spanning tree, we first sum the known incoming and outgoing edges, and then the unknown edge flow will by computed by subtracting the minimum of the two sums from the maximum (as before). This bottom-up propagation allows to populate the edge flows in a single pass over the basic blocks.

Forman (1981) and Ball and Larus (1994) propose to optimise the placement of the probes with respect to edges that are less likely to be executed. It works by considering a weighting that assigns a non-negative value to each edge in the CFG. The overhead cost of profiling a set of edges is considered to be proportional to the sum of the weights of the edges. These weights can be obtained either by empirical measurements from previous executions or by static heuristic estimations at compile-time. In order to minimise the profiling overhead, the instrumentation computes the maximum spanning tree in order to avoid probing in frequently executed edges.

Notice that when instrumentation is performed guided by empirical measurements from previous executions, it means that edge profiling information is used in order to produce a better edge profiling instrumentation. The next section presents the main algorithms for producing static estimates for the weights of the edges in a CFG.

3.3.1 Static Estimates of Edge Frequencies

Ball and Larus (1993) presented a simple algorithm that predicts the outcome of conditional branches with a reasonably good accuracy. For this purpose, they used several branch heuristics that were derived by measuring, on a large number of programs, the probability of branches being taken in respect of some *ad-hoc* features from the programs. Their algorithm selects, for each branch, the first heuristic that applies to the branch, in a given priority order of the heuristics. The *ad-hoc* heuristics defined by Ball and Larus (1993) are:

Loop branch heuristic (probability 88%): Probability of an edge back to a loop's head being executed.

Loop exit heuristic (probability 80%): Probability that a comparison inside a loop will *not* exit the loop. This heuristic does not apply to latch blocks, i.e. basic blocks that contain a branch back to the header of the loop.

Pointer heuristic (probability 60%): Probability that a comparison between two pointers, where one of them can be a null pointer, will fail.

Opcode heuristic (probability 84%): Probability that a comparison of an integer being less than zero, less than or equal to zero, or equal to a constant will fail.

Guard heuristic (probability 62%): For a comparison with a register as operand, where the register is defined in a successor basic block which is not a post-dominator. The probability that the successor basic block is reached.

Loop header heuristic (probability 75%): Probability of reaching a successor block

that is a loop header (or pre-header) but not a post-dominator.

Call heuristic (probability 78%): Probability of reaching a successor block that is not a post-dominator but contains a function call.

Store heuristic (probability 55%): Probability of reaching a successor block that is not a post-dominator but contains a store instruction.

Return heuristic (probability 72%): Probability of reaching a successor block that contains a return instruction.

Wu and Larus (1994) proposed an algorithm for statically estimating edge frequencies, which improves on the work of Wagner et al. (1994) and Ball and Larus (1993). This algorithm is able to combine several heuristics of the outcome of a branch into an estimated probability of the branch being taken. Wu and Larus (1994) use the *ad-hoc* heuristics defined by Ball and Larus (1993) as their initial predictions. They also use the Dempster-Shafer theory (Shafer et al., 1976) that provides the necessary mathematical technique for combining evidences from the different heuristics in order to produce more accurate estimates. These branch probabilities can then be used to estimate execution frequencies for all the edges in a CFG.

3.4 Summary

Chapter 4

Online Iterative Compilation

4.1 Work-based Metric

In this section we define the work-based performance metric proposed for comparing different optimised versions of a program when executing with different inputs. We define the performance metric as the ratio between the amount of *work*, ΔW, performed during a period of time, Δt.

$$P = \frac{\Delta W}{\Delta t}$$

By measuring the amount of *work* done per unit of time we reduce the impact of input-dependent aspects and focus instead on the efficiency of the optimised program. For this metric, the main challenge is to precisely define what represents *work*.

We model the computational work ΔW as a linear equation based on block frequency information and a cost-model of the instruction set.

$$\Delta W = \varepsilon + \sum_{B} w(B) f(B)$$

where $f(B)$ represents the frequency of basic block B and $w(B)$ represents the computational work of executing B. We define the work of a basic block B as the sum of the cost of its instructions, i.e.,

$$w(B) = \sum_{i} w_i N_B(i)$$

where w_i is the cost of instruction i and $N_B(i)$ is the number of occurrences of instruction i in basic block B.

In this simplified model, we consider that w_i is constant across all programs and executions in the given target platform. However, $N_B(i)$ is program dependent but constant across executions, while $f(B)$ is both program and execution dependent, since

$f(B)$ can change when executing with different inputs. In other words, $N_B(i)$ is a static value known at compile-time and $f(B)$ is a dynamic value known only at run-time.

Similarly to previous work (Giusto et al., 2001; Powell and Franke, 2009; Brandolese et al., 2011), we derive the cost model for the instruction set by modelling the problem as a multi-variable linear regression, where the *regression coefficients* are the costs of the instructions and the *regressors* (or *explanatory variables*) are computed as $\sum_B N_B(i) f(B)$ for each instruction i.

$$\Delta W = \varepsilon + \sum_i \left(w_i \sum_B N_B(i) f(B) \right)$$

By having some empirical data after executing several benchmarks with different inputs, we can fit the linear model with this empirical data in order to obtain the costs of the instructions.

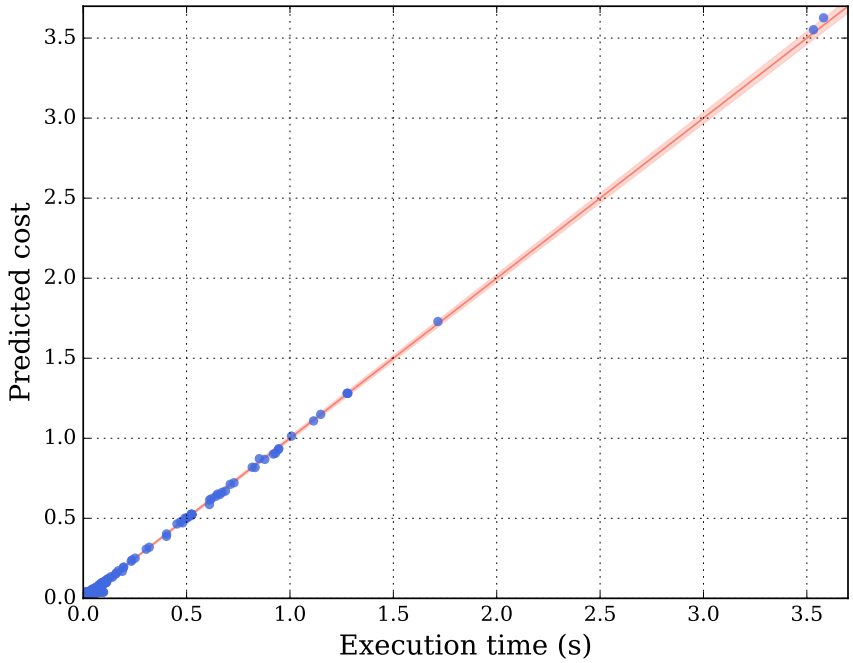


Figure 4.1: Linear model fitted from empirical data. The mean absolute error (MAE) for the fitted curve is of 7 milliseconds.

4.2 Online Iterative Compilation Infrastructure

Talk about LLVM as a lifelong optimisation framework. Talks about idle-time (between run) optimisations in the online scenario.

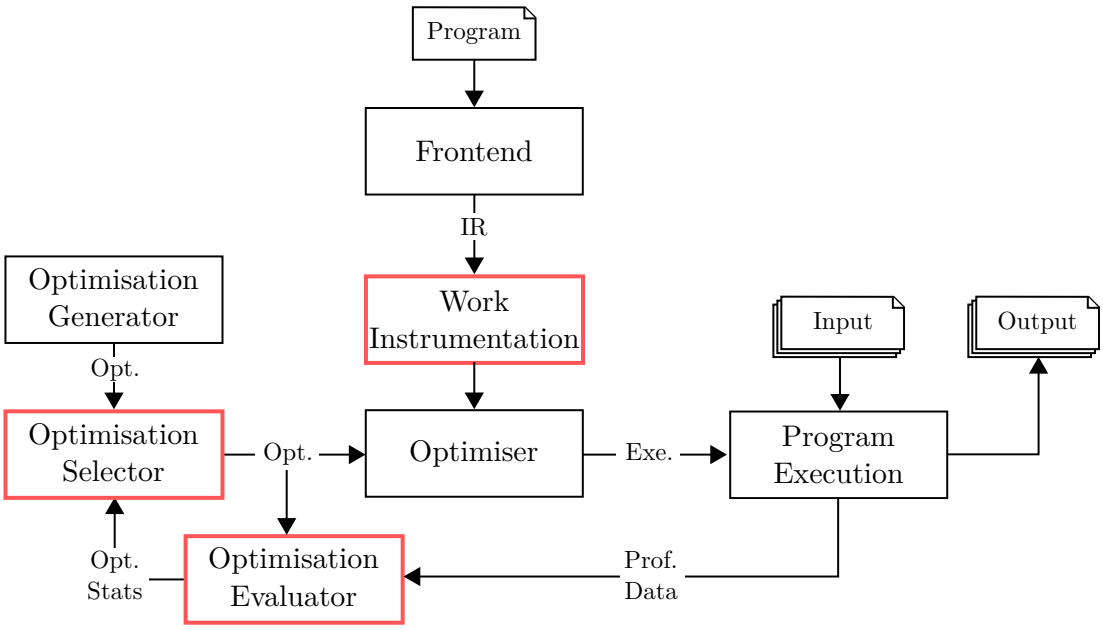


Figure 4.2: Overview of the execution engine for applying iterative compilation.

Describe the process step by step.

Figure 4.2 shows an overview of the infrastructure required for applying online iterative compilation. In this work we focus mainly on the two highlighted components. In both phases the goal is to improve the performance of the execution phase, but from different perspectives. *(i.)* In the compilation phase we focused on providing a low-overhead instrumentation for profiling the work metric. This phase is responsible to improve performance by lowering the overhead of the work profiling. *(ii.)* In the evaluation phase we proposed the use of the work-based metric in order to enable iterative compilation in an online scenario. This phase is responsible to improve performance by being able to properly assess different optimisations such that the best optimisation can be selected. Chapter 5 describes our work profiling strategy and Section 4.1 describes the work-based metric.

LLVM is particularly useful for iterative compilation as it makes possible to cache a pre-compiled, but still unoptimised, version of the input program in the bitcode format of the LLVM IR. This caching allows to speedup the time required for re-compilation as it is able to bypass the frontend phase. If re-compilation time is critical, it would also be possible to keep only the hot portion of the code in the LLVM bitcode format, while the remaining portion of the code is already in the format of object code. However, this is out of the scope of this work and we will always re-compile the whole program.

Talk about the process of keeping the same optimization for a given input-

window and how the optimizations as selected based on the evaluation of these ”evidences”. Perhaps we could suggest using the Theory of Evidences at this point?!

In most online scenarios, it is common for periods of peak usage and idle periods. For example, mobile devices are usually intensely used during the day, with some idle periods at night while its battery is being re-charged. The proposed infrastructure is very well suited for these online scenarios as multiple runs of the program can be monitored during peak time, by collecting the work profiling and measuring its execution time, while at the idle time, the profiling statistics can be used for selecting better optimisations and re-compiling the program. However, if idle time is almost non-existent, the proposed infrastructure can still be used by re-compiling the program with a different optimisation while multiple runs of the program are being executed.

Chapter 5

Work Instrumentation

In this section we describe how the computation of the work-metric can be performed during runtime by means of instrumenting the code. In particular, we adapt the optimal algorithm proposed originally for profiling block frequency (Nahapetian, 1973; Knuth and Stevenson, 1973; Ball and Larus, 1994). Afterwards, we propose a relaxed instrumentation that focus on further reducing the overhead by considering the trade-off between profiling accuracy and instrumentation overhead.

Because we define work as a linear equation on the block frequency counters, it is possible to embed its computation into the execution of the program. A naive instrumentation would consist basically of having a global counter that starts with the interception value, ε, and each basic block increments its own cost into the global counter. Although this instrumentation is easily implemented, it imposes a large overhead on the instrumented program. However, it is possible to insert fewer probes by carefully placing the probes in a way that is possible to reconstruct the complete profiling information Knuth and Stevenson (1973); Ball and Larus (1994).

We adapt the optimal block frequency instrumentation in order to perform the work profiling efficiently. The proposed work instrumentation differs from the optimal block frequency instrumentation as the latter occurs in two stages: *(i.) Before execution.* The code is instrumented with counters for each probe. *(ii.) After execution.* The information from the recorded probes is propagated in the CFGs of the program. In contrast, the work instrumentation has a single counter and it only requires the instrumentation before execution, without any post-processing of the recorded profiling.

Figure 5.1 shows a high-level overview of the complete work instrumentation algorithm. The highlighted sections are introduced or improved by our work profiling instrumentation. The instrumented code is assumed to be generated before optimising

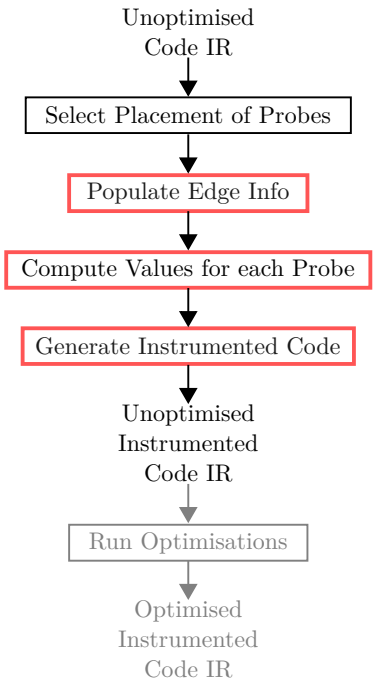
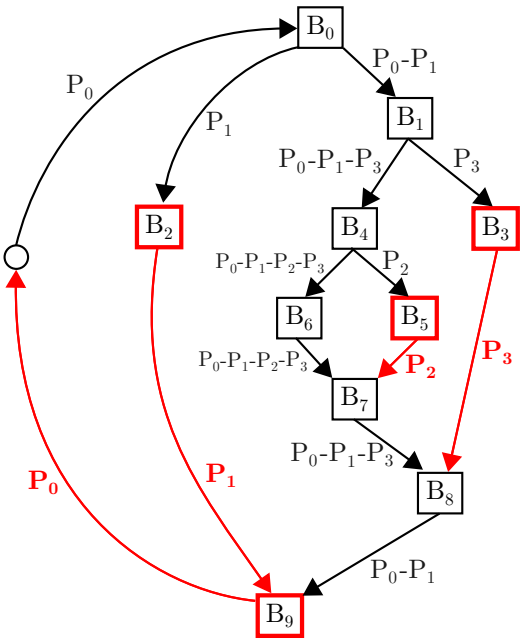


Figure 5.1: Overview of the work instrumentation algorithm.

the code. This assumption is based on three key points: *(i.)* it guarantees that the work metric is independent of optimisation, i.e., the same input is always mapped to the same amount of work, regardless of the optimisation; *(ii.)* it simplifies the code generator; *(iii.)* it leverages from the optimisations to further improve the instrumentation code. Having a work metric that is independent of optimisation is the most important reason for which we instrument the code before optimisations.

The optimal placement of the probes works exactly as previously described in Section 3.3. It first computes the maximum spanning tree based on a weighing that assigns a non-negative value to each edge in the CFG. These weights can be obtained either by empirical measurements or heuristic estimations, and their goal is to avoid probing in frequently executed edges. Once we have the maximum spanning tree, probes are placed on every edge not in the spanning tree. Figure 5.2 shows an example of a CFG with a maximum spanning tree represented by the black edges, while the edges highlighted in red represent the placement of the probes.

In contrast to the naive instrumentation where each basic block records only its own amount of work, with the optimal profiling, the instrumented basic blocks need to record an aggregated value of work that represents a path in the CFG. These values are constructed with some instrumented basic blocks speculatively assuming some paths while other probes correct when these assumptions are wrong (see for example $w(P_0)$



Instrumented value for each probe P_i:

$$\omega(P_0) = w(B_0) + w(B_1) + w(B_4) + w(B_6) + w(B_7) + w(B_8) + w(B_9)$$
$$\omega(P_1) = w(B_2) - w(B_1) - w(B_4) - w(B_6) - w(B_7) - w(B_8)$$
$$\omega(P_2) = w(B_5) - w(B_6)$$
$$\omega(P_3) = w(B_3) - w(B_4) - w(B_6) - w(B_7)$$

Figure 5.2: Example of a CFG with its minimum spanning tree in black and the basic blocks highlighted in red represent the instrumented basic blocks with the placement of the probes.

and $w(P_1)$ in Figure 5.2).

Because the algorithm for the optimal placement of the probes is proved to uniquely compute the block frequencies by propagating the probe counts, we adapt this algorithm (see Listing 3.1) in order to compose the aggregated values that will be instrumented in each probe, based on our model of computational work, ΔW, derived from the basic block frequencies (see Section 4.1). We perform a similar propagation of the probes in a symbolic fashion, as illustrated in Figure 5.2. This symbolic propagation of the probes is implemented by the data-flow analysis described in Listing 5.1, and the final aggregated values are extracted from the edge information as described by Listing 5.2.

The data-flow analysis in Listing 5.1 keeps two sets for each edge, namely, the increment and the decrement sets. We consider that both sets represent the *edge ex-*

```
1  // Inputs: CFG with the known edges flows of the chords
2  // Output: Updated CFG with all edge flows
3  populateEdgeInfo(G) {
4    //the instrumented edges are the only known edge flows
5    for e in instrumentedEdges(G):
6      B = instrumentedBlock(e)
7      inc[e] = set( {B} )
8      dec[e] = set()
9      knownInfo[e] = true
10
11   changed = true
12   while changed:
13     changed = false
14     for B in G.vertices():
15       unIN = count( G.unknownIncomingEdges(B,knownInfo) )
16       unOUT = count( G.unknownOutgoingEdges(B,knownInfo) )
17       if unIN==0 and unOUT==1:
18         //sum known incoming and outgoing edges in B
19         incSum = set()
20         decSum = set()
21         for predB in G.predecessors(B):
22           incSum = incSum union dec[G.getEdge(predB, B)]
23           decSum = decSum union inc[G.getEdge(predB, B)]
24         for succB in G.successors(B):
25           incSum = incSum union inc[G.getEdge(B, succB)]
26           decSum = decSum union dec[G.getEdge(B, succB)]
27         //update unknown outgoing edge in B with incSum and decSum
28         e = G.getUnknownOutgoingEdge(B,knownInfo)
29         inc[e] = incSum-decSum
30         dec[e] = decSum-incSum
31         knownInfo[e] = true
32         changed = true
33       if unIN==1 and unOUT==0:
34         //sum known incoming and outgoing edges in B
35         incSum = set()
36         decSum = set()
37         for predB in G.predecessors(B):
38           incSum = incSum union inc[G.getEdge(predB, B)]
39           decSum = decSum union dec[G.getEdge(predB, B)]
40         for succB in G.successors(B):
41           incSum = incSum union dec[G.getEdge(B, succB)]
42           decSum = decSum union inc[G.getEdge(B, succB)]
43         //update unknown incoming edge in B with incSum and decSum
44         e = G.getUnknownIncomingEdge(B,knownInfo)
45         inc[e] = incSum-decSum
46         dec[e] = decSum-incSum
47         knownInfo[e] = true
48         changed = true
49 }
```

Listing 5.1: Pseudocode of the data-flow analysis for assigning the values computed in each probe of the instrumentation for the profiling of the work metric.

```
1 // Inputs: 1) CFG with the known edges flows of the chords
2 //         2) Basic block targeted for probing
3 // Output: Work value to be incremented by the given probe B
4 instrValue(G, B, inc, dec) {
5   value = 0
6   for B in G.vertices():
7     incB = set()
8     decB = set()
9     for succB in G.successor(B):
10      incB = incB union inc[ G.getEdge(B, succB) ]
11      decB = decB union dec[ G.getEdge(B, succB) ]
12    if B in (incB - decB):
13      value = value + w(B)
14    if B in (decB - incB):
15      value = value - w(B)
16  return value
17 }
```

Listing 5.2: Pseudocode that describes how the edge information is used in order to extract the value that will be computed in a given instrumented basic block B_I. This algorithm could equally be implemented based on the predecessors.

pressions shown in Figure 5.2, for which we define a *symbolic sum* by computing the union of the increment and decrement sets, respectively, with the appropriate cancellation of common elements. This data-flow analysis is based on the invariant that the symbolic sum of all the incoming edges must equals the symbolic sum of the outgoing edges. For example, the symbolic sum of the incoming edges of the basic block B_8 is $P_0 - P_1$, where P_3 is cancelled out.

Listing 5.2 reads the edge information for each basic block by computing the symbolic sum of their respective incoming edges (or outgoing edges). From these *edge expressions*, we are able to compose the aggregated value of the probes. The positive terms in the edge expression of a basic block indicate that the amount of work of this basic block will be incremented in the probes represented by these positive terms, similarly, the negative terms indicate that the amount of work of this basic block will be decremented in the probes represented by these negative terms. For example, because the edge expression for the basic block B_8 is $P_0 - P_1$, the amount of work of B_8, denoted by $w(B_8)$, is incremented in probe P_0 and decremented in P_1.

5.1 Code Generation for the Instrumentation Probes

This section describes the code generator for the work instrumentation. Once we have the placement of the probes as well as the aggregated value computed for each probe, we insert the appropriate instructions for the probes of the work profiling. Because the instrumented code is assumed to be generated before optimising the code, the

code generator has a straightforward code generation process. It produces the code for the probes using a local variable as it tends to improve optimisation opportunities. This variable acts as the local accumulator that will eventually be incremented into the global work counter.

For every function, the code generator allocates memory on the stack frame for the local work variable. In LLVM, allocated memory on the stack is automatically released when the function returns, therefore there is no need to generate code for that purpose. Afterwards, the local work variable is set to zero.

```
1  %local.work = alloca i32
2  store i32 0, i32* %local.work
```

Listing 5.3: Code for the entry point of a function.

For every probe, the code generator produces memory access operations in addition to the actual increment of the local work counter. The value incremented to the local counter is the value computed by Listing 5.2.

```
1  %r1 = load i32, i32* %local.work
2  %r2 = add i32 %r1, 8
3  store i32 %r2, i32* %local.work
```

Listing 5.4: Code for a probe that increments the local work counter.

Because it produces instrumentation code using a local variable, eventually this local variable needs to be incremented into the global counter. This is performed at every exit point of the function, even if the basic block with the exit point was not selected for having a probe. For every exit point of the function, the code generator produces simple memory access operations that load the current values of both the local and the global counters, adds them together, and finally updates the global counter.

```
1  %r1 = load i32, i32* %local.work
2  %r2 = load i32, i32* @__work_counter
3  %r3 = add i32 %r1, %r2
4  store i32 %r3, i32* @__work_counter
```

Listing 5.5: Code at an exit point of a function.

For the special case where a probe belongs to a basic block which is also an exit point of the function, the code generator leverages from this scenario to produce a code for the probe that works in collaboration with the update of the global counter.

```
1  %r1 = load i32, i32* %local.work
2  %r2 = add i32 %r1, 273
3  %r3 = load i32, i32* @__work_counter
4  %r4 = add i32 %r2, %r3
5  store i32 %r4, i32* @__work_counter
```

Listing 5.6: Code for a probe that increments the local work counter and also updates the global counter at an exit point of a function.

After optimisations, the instrumented code can benefit from some of the transformations. For example, the optimisation pass for promoting memory to register (`-mem2reg`) is able to eliminate the local variable such that the local work counter is only kept in registers. The only explicit access to memory is performed when updating the global work counter.

```
1  ...
2  ;If probes from multiple paths reach a given point,
3  ;a phi operation is used for selecting the appropriate value.
4  ;Furthermore, the initial store of 0 is unnecessary.
5  %r1 = phi i32 [ 0, %entry ], [ %r2, %for.inc ]
6  ...
7  ;If there is only one value that reaches a given point,
8  ;this value can be directly used.
9  %r3 = add i32 %r1, 273
10 %r4 = load i32, i32* @__work_counter
11 %r5 = add i32 %r3, %r4
12 store i32 %r5, i32* @__work_counter
```

Listing 5.7: An illustrative example of how `-mem2reg` can optimise the instrumented code.

5.2 Relaxed Instrumentation

Although the optimal instrumentation significantly reduces the profiling overhead when compared to the naive instrumentation, from an average overhead of 79% to 13%, in some critical cases, even the optimal instrumentation can have an overhead of about 70% (see benchmark `adpcm_d` in Figure 6.5). In order to further reduce the overhead in these critical cases, we propose a relaxed instrumentation by trading off accuracy and overhead. Figure 5.3 shows an overview of the relaxed instrumentation algorithm. The highlighted section is introduced by the relaxed instrumentation on top of the previously defined optimal work instrumentation algorithm.

The relaxed instrumentation performs a post-processing on the resulting instrumentation of the optimal algorithm. The relaxation starts by extracting DAGs (directed acyclic graphs) from the CFG, as illustrated in Figure 5.4. The algorithm extracts all the subgraphs that represent a loop or the outer most region of the function. These subgraphs are transformed into DAGs by ignoring the back edge and also by considering that any loop within the subgraph is never executed, i.e., only the headers of the inner loops are actually included into the DAG. Figure 5.4 shows a CFG partitioned into two DAGs (consider only basic blocks and edges completely inside the yellow and green boundaries).

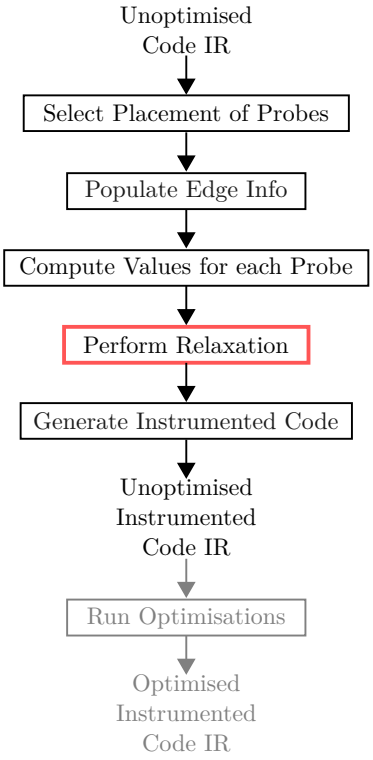


Figure 5.3: Diagram.

For every DAG with a set of probes $\{P_0, P_1, \ldots, P_k\}$, we relax the profiling accuracy by selecting a subset of the probes to be removed, subject to the maximum allowed percentage error, M.

We model the relaxation as a 0-1 Knapsack problem:

$$\text{maximise} \sum_{i=0}^{k} f(P_i)x_i$$

$$\text{subject to} \sum_{i=0}^{k} \varepsilon(P_i)x_i \leq M \text{ and } x_i \in \{0,1\}$$

where $f(P_i)$ is the execution frequency of probe P_i, x_i denotes the probes selected for removal, and $\varepsilon(P_i)$ is the percentage error of removing probe P_i relative to the minimum work value possible to compute in the DAG, i.e., if m is the minimum amount of work possible to be computed when executing the DAG, then $\varepsilon(P_i) = \frac{\omega(P_i)}{m}$. Because the percentage error is computed based on the path with the minimum amount of work, $\varepsilon(P_i)$ represents the maximum error possible that would be incurred when removing probe P_i. Furthermore, by constraining the percentage error of every DAG below a given threshold we guarantee that the final error of the relaxation will always be bounded by the threshold.

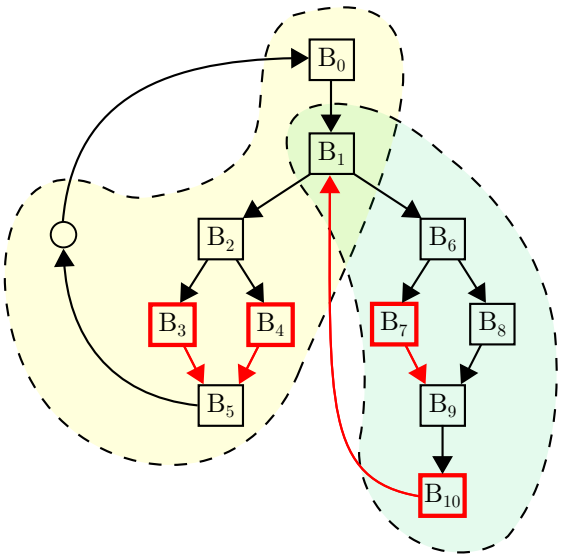


Figure 5.4: Example of a CFG containing a loop and its decomposition into DAGs when applying the relaxation. The DAGs are the subgraphs within the dashed boundaries.

Proposition 5.2.1. Let n_i be the number of times a given DAG i is executed, r_i be the total relaxation (amount of work removed) in DAG i, and m_i be its minimum amount of work. If $\frac{r_i}{m_i} \leq M$ for every i, then the final error of the relaxation will always be bounded by the same threshold.

Proof. We can model the overall error of the relaxation as:

$$1 - \frac{n_1(m_1 - r_1) + n_2(m_2 - r_2) + \ldots + n_k(m_k - r_k) + c}{n_1 m_1 + n_2 m_2 + \ldots + n_k m_k + c}$$

That is,

$$1 - \frac{n_1 m_1 + n_2 m_2 + \ldots + n_k m_k + c}{n_1 m_1 + n_2 m_2 + \ldots + n_k m_k + c} + \frac{n_1 r_1 + n_2 r_2 + \ldots + n_k r_k}{n_1 m_1 + n_2 m_2 + \ldots + n_k m_k + c} = \frac{n_1 r_1 + n_2 r_2 + \ldots + n_k r_k}{n_1 m_1 + n_2 m_2 + \ldots + n_k m_k + c}$$

If $\frac{r_j}{m_j}$ is the maximum ratio $\frac{r_i}{m_i}$ for every i, then

$$\frac{n_1 r_1 + n_2 r_2 + \ldots + n_k r_k}{n_1 m_1 + n_2 m_2 + \ldots + n_k m_k + c} \leq \frac{n_1 r_j + n_2 r_j + \ldots + n_k r_j}{n_1 m_j + n_2 m_j + \ldots + n_k m_j + c} \leq \frac{n_1 r_j + n_2 r_j + \ldots + n_k r_j}{n_1 m_j + n_2 m_j + \ldots + n_k m_j}$$

If $N = max\{n_i \text{ for every } i\}$, then

$$\frac{n_1 r_j + n_2 r_j + \ldots + n_k r_j}{n_1 m_j + n_2 m_j + \ldots + n_k m_j} \leq$$
$$\frac{N r_j + N r_j + \ldots + N r_j}{N m_j + N m_j + \ldots + N m_j} =$$
$$\frac{N k r_j}{N k m_j} = \frac{r_j}{m_j} \leq M$$

□

While the optimal placement of probes tries to place probes in edges that are less likely to be executed, the relaxation focus on removing probes that are more likely to be executed. The necessary block frequency information for optimising both the placement of probes can be acquired from profiles of previous executions of the program or by a static heuristic of the CFG during compilation.

For our experiments, we implemented two solvers for the 0-1 Knapsack problem: the optimal brute-force solver; the greedy heuristic based on sorting the items (Dantzig, 1957). We use the brute-force solver for DAGs with a small number of probes and the greedy heuristic when the number of probes is greater than a threshold. Some of the benchmarks have DAGs with several hundreds of probes, which could result in a long compilation time.

5.3 Whole Program Relaxation

In some cases, the proposed relaxation can be very conservative, because it considers the static error of removing probe P_i relative to the minimum work possible of a DAG, in order to be able to guarantee that the dynamic error will be bounded by a given threshold. This conservatism can be overly restrictive in some cases, resulting in a negligible overhead reduction but also causing only negligible dynamic error to the work profiling.

In some cases, this overly restrictive conservatism may be unecessary. For these cases, we propose an adapted version of the relaxation algorithm that operates on the whole program. Traditionally, compilers optimisations are performed on the function-level, or at best on a per module basis. Whole program optimisation (WPO) means that the compiler considers all compilation units of the program and optimises them using the combined knowledge of how they are used together.

The whole program relaxation works by using block frequency profiling from previous executions. By having this profiling information, the *whole program relaxation* is able to compute the error of removing a given probe in terms of the whole program's execution, and then use this error values for selecting a subset of all the probes to be removed.

For a program with a set of probes $\{P_0, P_1, \ldots, P_k\}$, we model the whole program the relaxation as the following 0-1 Knapsack problem:

$$\text{maximise} \sum_{i=0}^{k} f(P_i)x_i$$

$$\text{subject to} \sum_{i=0}^{k} \varepsilon(P_i)x_i \leq M \text{ and } x_i \in \{0,1\}$$

where $f(P_i)$ is the execution frequency of the instrumented basic block P_i, x_i denotes the probes selected for removal, M is the error threshold, and $\varepsilon(P_i)$ is the percentage error of removing probe P_i relative to the profiled global work, i.e., if ΔW is the work value for the whole program's execution, computed from the basic block frequencies profiled from previous executions, the error for a given probe P_i is

$$\varepsilon(P_i) = \frac{\omega(P_i)f(P_i)}{\Delta W}.$$

Contrary to the per DAG relaxation, the whole program relaxation is not guaranteed to be bounded by the error threshold M, as it is dependent on the representativity of the profiling information that was provided.

5.4 Other Possible Uses

Talk about how this instrumentation techniques (relaxation) could be applied to other profiling-based mechanism to estimate input sizes (Zaparanuks and Hauswirth, 2012; Coppa et al., 2014), namely *read memory size*

Chapter 6

Experimental Evaluation

In this section we discuss our experimental evaluation. First we describe the benchmarks with datasets used in the experiments. Afterwards, we discuss our results concerning the instrumentations for profiling our work metric. Finally we present the results of the online iterative compilation.

We implemented the instrumentations in LLVM 4.0. The target platform is a Linux-4.4.27 system with an Intel Core i7-4770 3.40GHz Skylake CPU with 16 GiB RAM.

6.1 Benchmarks

For the experimental evaluation we have used a subset of the *KDataSets* benchmark suit, which is the same benchmark and dataset suit used by Chen et al. Chen et al. (2010, 2012b). The KDataSets contains 1000 different inputs for each one of its benchmark programs. These benchmarks cover a broad spectrum of application scenarios, ranging from simple embedded signal-processing tasks to common mobile-phone and desktop tasks. The different inputs try to capture distinct characteristics in terms of workload sizes and how these workloads exercise different control flow paths. A summary of the benchmarks and dataset suit is shown in Table 6.1.

The shaded (grey) benchmarks in Table 6.1 represent the benchmarks used for training the cost model used for computing the weight of the instructions for the work metric. These same training benchmarks were also used for collecting a fixed set of optimisations for the iterative compilation. The remaining (white) benchmarks are used for the experimental evaluation.

Program	LOC	Input file size	Input description
qsort	154	32K-1.8M	3D coordinates
jpeg_d	13501	3.6K-1.5M	JPEG images
jpeg_c	14014	16K-137M	PPM images
tiff2bw	15477	9K-137M	TIFF images
tiff2rgba	15424		
tiffdither	15399		
tiffmedian	15870		
susan_c	1376	12K-46M	PGM images
susan_e	1376		
susan_s	1376		
adpcm_c	210	167K-36M	WAVE audios
adpcm_d	211	21K-8.8M	ADPCM audios
lame	14491	167K-36M	WAVE audios
rsynth	4111	0.1K-42M	Text files
sha	197	0.6K-35M	Files of any format
bitcount	460	-	Numbers: random
dijkstra	163	0.06K-4.3M	Adjacency matrices
patricia	290	0.6K-1.9M	IP and mask pairs
mad	2358	28K-27M	MP3 audios
ghostscript	99869	11K-43M	Postscript files
stringsearch	338	0.1K-42M	Text files
CRC32	130	0.6K-35M	Files of any format
bzip2e	5125	0.7K-57M	Files of any format
bzip2d	5125	0.2K-25M	Compressed files

Table 6.1: Description of the KDataSets with 1000 inputs for each benchmark (Chen et al. Chen et al. (2010, 2012b)).

6.2 Evaluation of the Instrumentation

In this section we evaluate the performance of the work instrumentation presented in Chapter 5, comparing between the optimal and the relaxed instrumentation with different thresholds.

6.2.1 Static Evaluation

We first compare static aspects of the instrumentation algorithms. The naive instrumentation always has 100% of the basic blocks instrumented, by definition. Figure 6.1 shows percentage of instrumented basic blocks for the optimal and the relaxed instrumentation with different relaxation thresholds. While Figure 6.1 compares the different

instrumentation algorithms in respect of the naive instrumentation, Figure 6.2 shows the improvement of the relaxation algorithm over the optimial instrumentation.

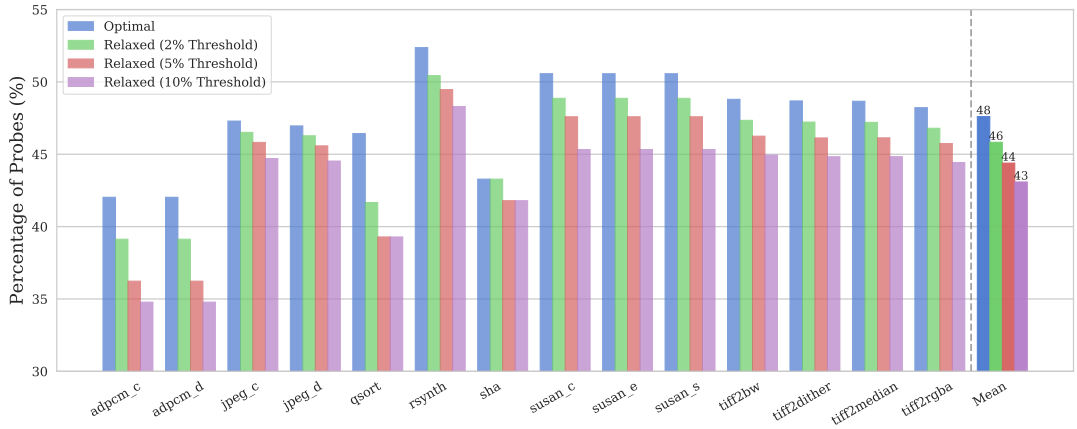


Figure 6.1: Percentage of instrumented basic blocks for the optimal and the relaxed instrumentation with different relaxation thresholds.

Even a small threshold of 2% is able to reduce the number of probes by an average of 5% compared to the optimal algorithm. The `sha` benchmark was the only benchmark for which a 2% threshold was not sufficient for further reducing the number of probes. With a 10% threshold the relaxation algorithm was able to improve over the optimal instrumentation by an average of 11%, in terms of the amount of instrumented basic blocks.

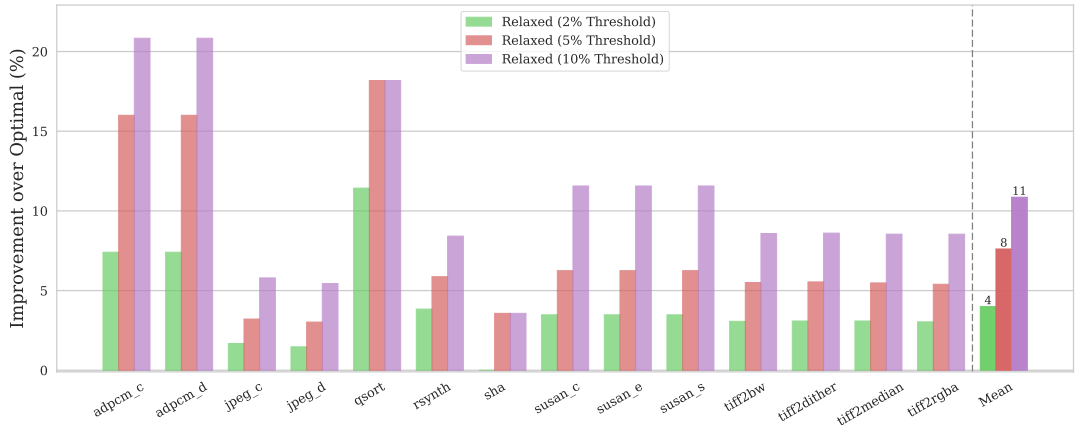


Figure 6.2: Percentage of instrumented basic blocks for the optimal and the relaxed instrumentation with different relaxation thresholds.

The static errors presented in Figure 6.3 indicate the amount of error expected by the relaxation algorithm after reducing the number of probes. This figure shows both

the maximum (bars with light colours) and the average (bars with dark colours) static errors observed when relaxing the instrumentation for each DAG of the benchmarks. Notice that, in most cases, the average static errors are considerably lower than the relaxation threshold, while the maximum static errors are usually close to the threshold. The average static error shows that in a significant number of DAGs, the relaxation was notably conservative. This conservatism will be specially evident in the dynamic evaluation.

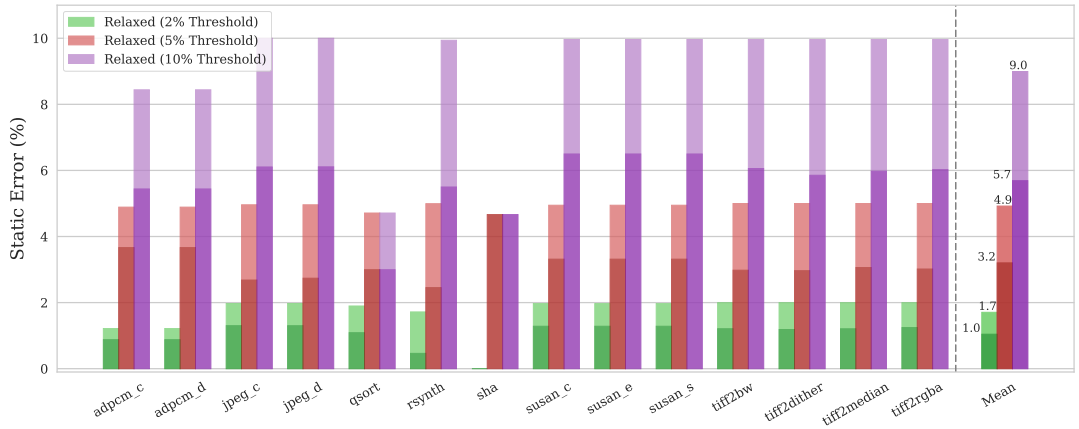


Figure 6.3: Average and maximum static error expected for the work profiling, after relaxing the number of probes.

In order to investigate the effects caused to the instrumentation when varying the relaxation threshold, we performed a full analysis of the reduction in the number of probes varying the threshold from 0% up to 100%, as presented in Figure 6.7. This figure shows the percentage of probes, instrumented basic blocks, over the total number of basic blocks. It is interesting to understand some of the aspects of these results. In some cases, when increasing the threshold, the number of probes also increased, e.g., as it is clearly noticable in the `qsort` benchmark. This happens when the larger threshold allows the 0-1 knapsack solver exchange a large probes for fewer probes with higher execution frequency value but also higher static error. Figure 6.7 also shows the It is also interesting to notice the highly conservative aspect of the relaxation algorithm, as most benchmarks remain with more than 30% of basic blocks instrumented, even with a 100% relaxation threshold. This happens as the static error is computed based on the path with the minimum amount of work possible in the DAG, which can be very conservative in most cases.

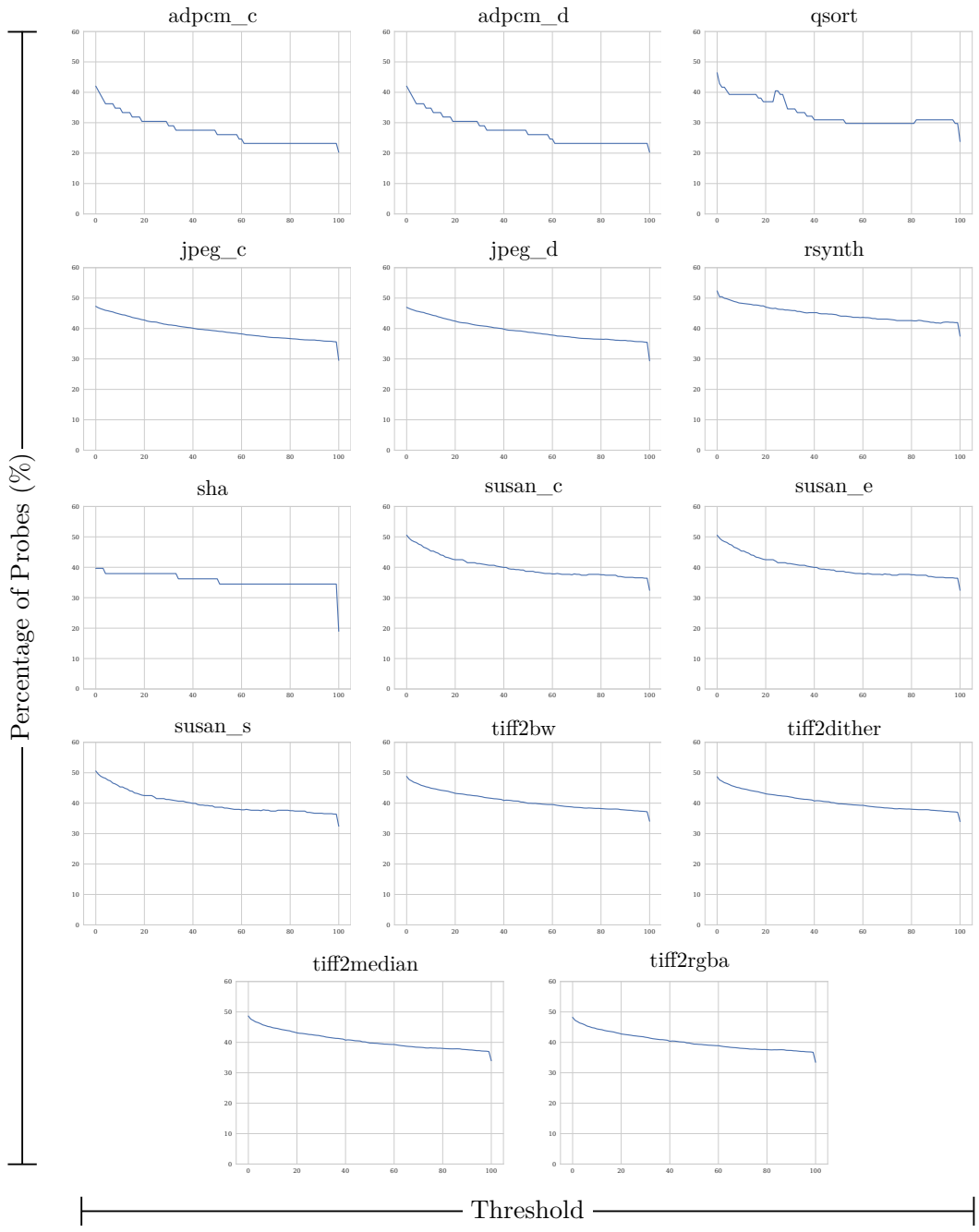


Figure 6.4: .

6.2.2 Dynamic Evaluation

For the evaluation of the performance overhead that the instrumentation incur to the benchmarks, we measure the wall-clock time of the benchmarks when compiled with the default `-O3` optimisation. For each benchmark, we compute the average overhead over all its 1000 input dataset. When measuring the wall-clock time for each input, in order to reduce noise, we execute the same input until we have a statistically sound

measurement, i.e. we execute until we have an interval no larger than 1% with 99% confidence. Figure 6.5 shows the performance overhead imposed by the work instrumentation on the benchmarks when compared to their non-instrumented counterparts. Figure 6.6 shows the improvement of the relaxation algorithm over the optimial instrumentation, regarding the reduction in overhead.

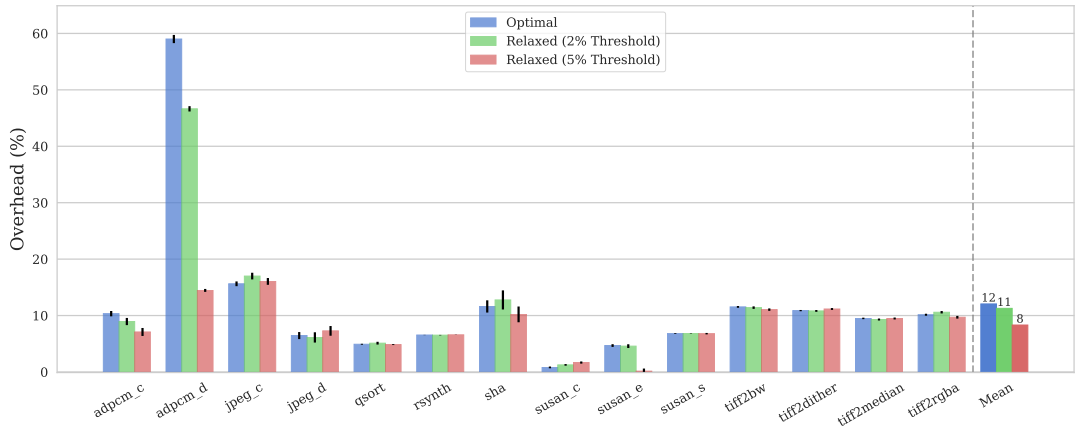


Figure 6.5: Overhead of the instrumentations compiled with `-O3`.

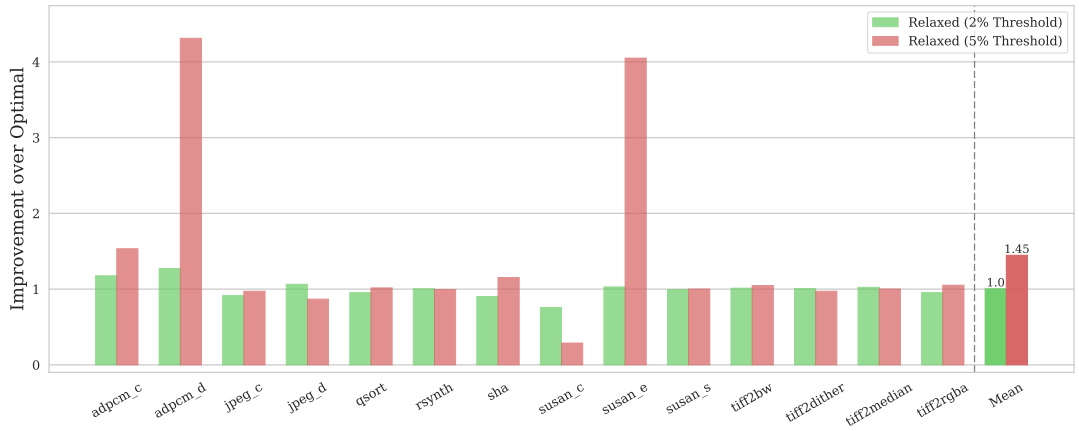


Figure 6.6: Overhead of the instrumentations compiled with `-O3`.

Figure 6.6 shows that the relaxation algorithm, with 5% threshold, is able to improve an average of 48% over the optimal instrumentation, improving about 4.5× for both the `adpcm_d` and `susan_e` benchmarks.

Profile-guided Instrumentation.

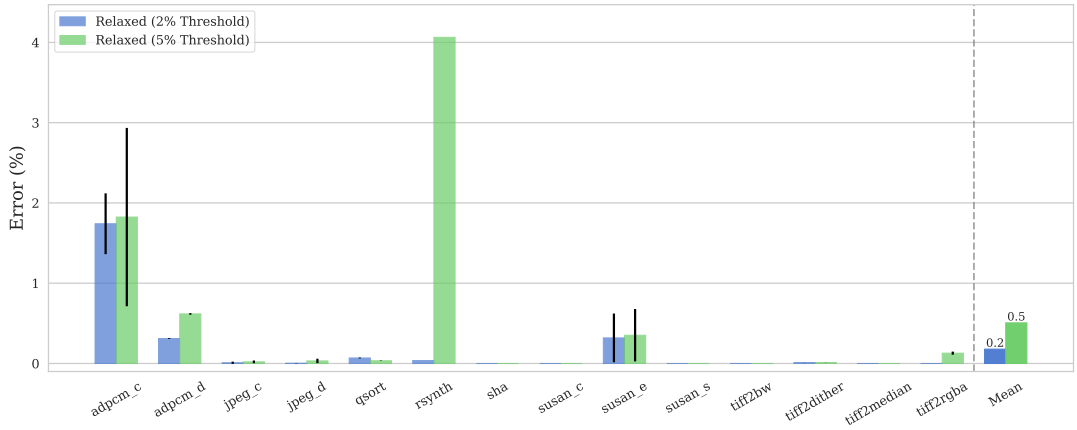


Figure 6.7: .

6.2.3 Case Studies

Analysis of the Best Improvement in Overhead: `adpcm_d`

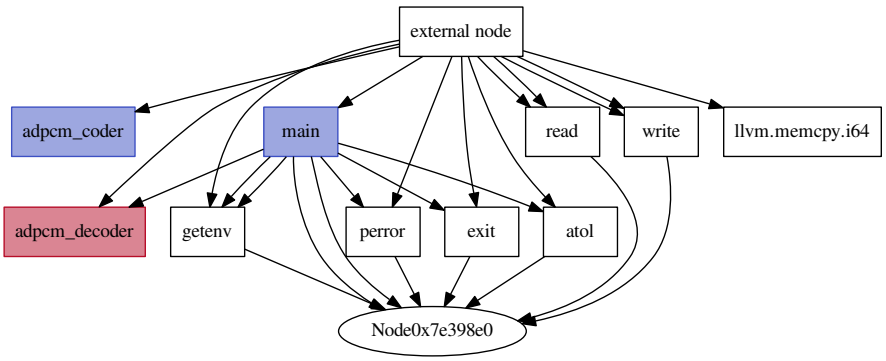


Figure 6.8: Call graph highlighting the maximum basic block frequency of each function, using a cool/warm colour map.

The `adpcm_d` benchmark is the most critical case amongst the evaluated benchmarks, with an overhead of about 59% for the optimal instrumentation. Figure 6.8 shows the *heat* call-graph of the `adpcm_d` benchmark, with each function coloured based on their maximum basic block frequency, using a cool/warm colour map. This benchmarks has a single hot function, namely the function called `adpcm_decoder`. Moreover, it consists mainly of a single hot loop with several branches inside it, as depicted by Figure 6.9. The relaxation algorithm is able to reduce this overhead down to about 47% (with a static error threshold of 2%) and 14% (with a static error threshold

of 5%) by removing only one and two probes from the hot loop, respectively. These overhead reductions represent a 20% and 4.5× improvement over the optimal algorithm, with 2% and 5% relaxation threshold respectively. The two relaxed probes were placed in branches, inside the hot loop, with a high probability of being taken, but with a small contribution to the total amount of work measured in the loop.

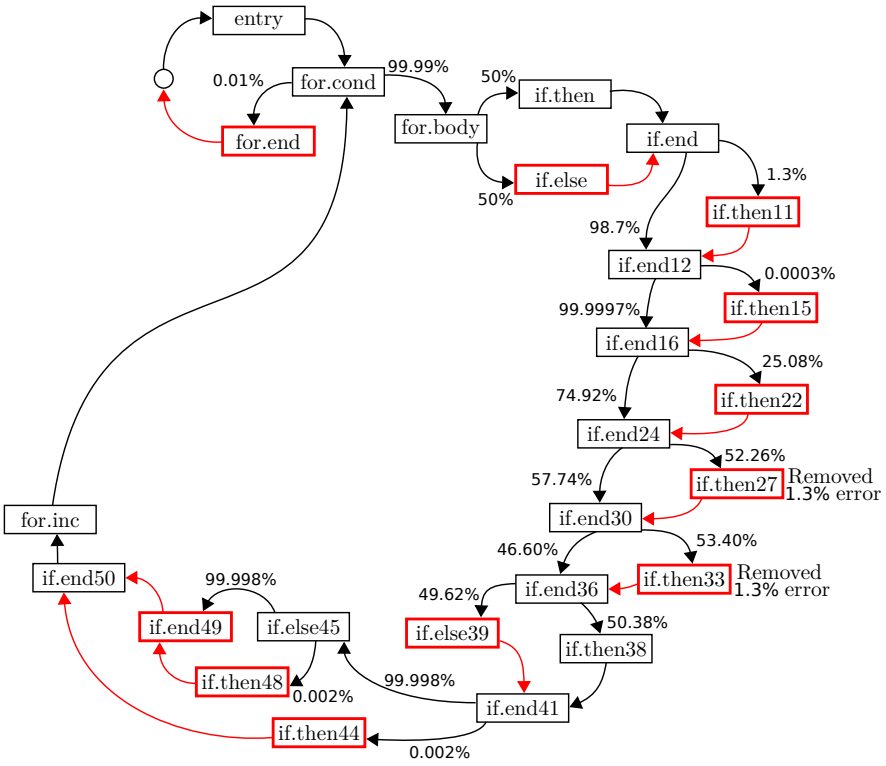


Figure 6.9: CFG of the function that contains the hot loop of the `adpcm_d` benchmark.

Figure 6.10 shows all the probes necessary for the function that contains the hot loop of the `adpcm_d` benchmark. Notice that the relaxation was able to remove all probes with an static error lower than the 5% threshold. Because these probes were placed in basic blocks with a considerable execution frequency, their removal resulted in a significant reduction in performance overhead.

Analysis of the Abnormal Regression: `susan_c`

6.3 Evaluation of the Online Iterative Compilation

In this section we evaluate the online iterative compilation guided by the work-based performance metric. For comparison, we use the following baselines and configurations:

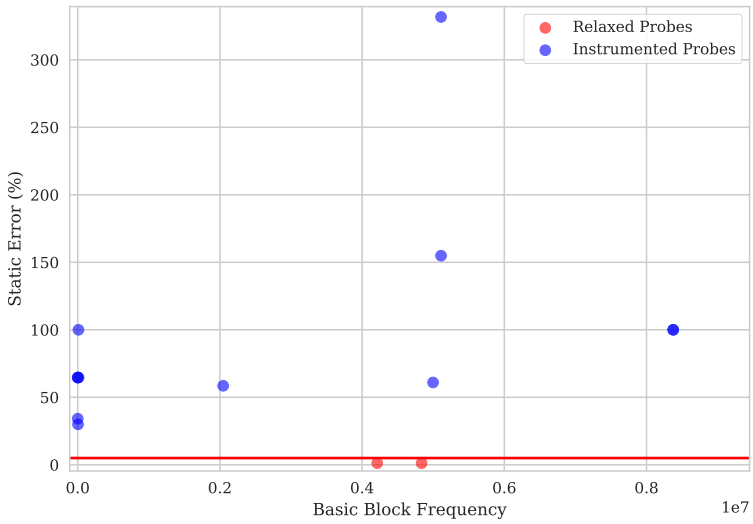


Figure 6.10: Comparison between probe frequency and static error for the `adpcm_d` benchmark. The red line marks the 5% threshold limit.

- **Oracle-RM** executes the program multiple times for each input, measuring the real speedup for each compiler optimisation, and then uses the real speedup over `-O0` for comparing the optimisations. The speedups are computed based on the wall-clock time. In order to reduce noise, the program is executed several times for the same input, until the confidence interval no larger than 1% for a 99% confidence.

- **Oracle-PP** represents an oracle with a *perfect* non-intrusive profiling. Although it uses the work-based performance metric for comparing optimisation, this oracle also avoids noise in its measurements by also executing the program multiple times for each input. The first execution is used for profiling the work metric. The remaining executes are used for measuring the wall-clock time without using the work profiling.

- **Real-OP** corresponds to the online iterative compilation as it would be applied in a real online scenario. For each optimisation, a random sample of inputs is selected, and the program is executed only once with each input. When executing each input, it uses the optimal work instrumentation for profiling the work metric. The average of the work-based performance metric, $\frac{\Delta W}{\Delta t}$, for the sample of inputs is then used for selecting the best optimisation.

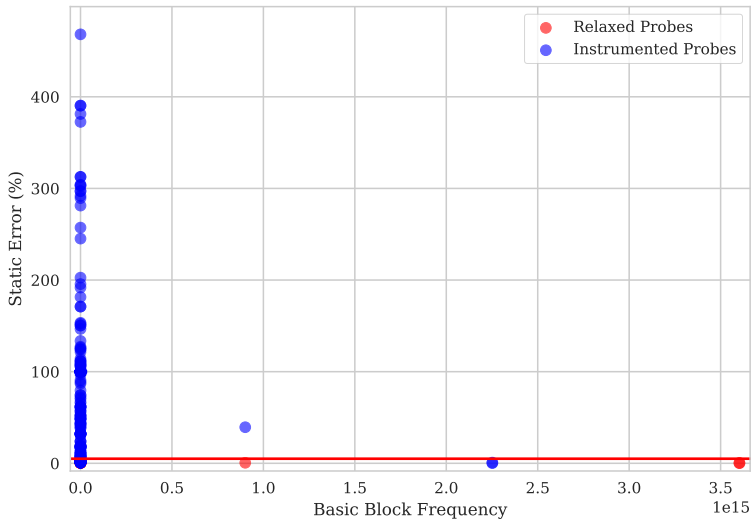


Figure 6.11: Comparison between probe frequency and static error for the `susan_c` benchmark. The red line marks the 5% threshold limit.

- **Real-R5** is similar to the Real-OP. It also corresponds to the online iterative compilation as it would be applied in a real online scenario. For each optimisation, a random sample of inputs is selected, and the program is executed only once with each input. When executing each input, it uses the relaxed work instrumentation, with a 5% threshold, for profiling the work metric. The average of the work-based performance metric for the sample of inputs is then used for selecting the best optimisation.

In all configurations, the same optimisation is used for multiple distinct inputs, using a dynamic window size, as explained in Section 4.2. This window size provides an estimate for the optimisation performance across inputs. When selecting the best optimisation, they are ranked by the average of their performance measurement, using the work-based performance metric, except for the Oracle-RM which uses the real speedup over `-O0`. Figure 6.12 shows the average window size for each benchmark and configuration.

6.3.1 The Optimisation Set

For the purpose of evaluating the use of the work-based performance metric with iterative compilation, we collected in advance a fixed set of optimisations. This set contains 500 optimisations collected in a random search using the training benchmarks. These

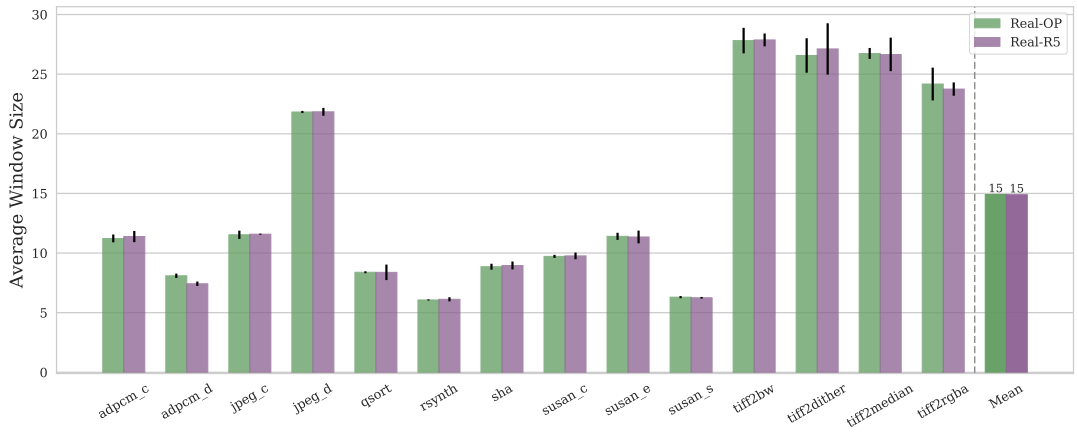


Figure 6.12: Average window size observed during the online iterative compilation.

optimisations contain an average of 40 transformation, including repetitions, with a maximum of 119 optimisation passes, but it also contains optimisations which consists of a single flag, such as the default optimisations `-O1`, `-O2`, `-O3`, `-Os`, and `-Oz`.

Example of a short optimisation sequence:

-mem2reg -simplifycfg -constprop -dce

Example of a long optimisation sequence:

-globalopt -reassociate -instcombine -loop-rotate -block-freq -deadargelim -early-cse -sroa -argpromotion -sccp -tbaa -barrier -constmerge -loop-vectorize -domtree -basicaa -memdep -basiccg -memcpyopt -constprop -adce -globaldce -mem2reg -constmerge -globaldce -constprop -instsimplify -dse -dce -simplifycfg -loop-unroll -reassociate -constprop -globaldce -instsimplify -adce -constmerge -bb-vectorize -dce -mergefunc -simplifycfg -dse -loop-unroll -globaldce

Example of an optimisation sequence which includes -O3:

-O3 -adce -globaldce -simplifycfg -memcpyopt -reassociate -mergefunc -dce -dse

Repeating the same optimisation pass can be beneficial and usually expected by other passes. For example, the -loop-simplify pass is used for transforming loops into a canonical form by inserting pre-header and exit basic blocks. Although this pass inserts jumps due to redundant basic blocks, this canonical form can be favourable to other loop optimisations. Because of the redundant basic blocks, this optimisation pass expects that the -simplifycfg will eventually be executer later on the optimisation pipeline. Another example of such inter-relation between transformations concerns the -licm and -mem2reg passes. The -licm pass is responsible for moving invariant code

out from the loop body. It usually creates new local variables, using memory access operations, for assisting with the code manipulation, which means that the executing the -mem2reg pass afterwards would be useful as a cleanup pass for removing the extra memory accesses generated. However, many of the analysis required for identifying loop invariant also benefit from the transformations performed by the -mem2reg pass. These examples illustrate the importance of repeating optimisation passes. Moreover, they illustrate the intricate relation amongst several transformations.

However, all optimisations in the set of optimisations were generated completely at random, without using any knowlege of individual transformations. Each optimisation was generated in two steps: *(1)* randomly selects the number of flags; *(2)* randomly selects the flags, allowing repetitions. Afterwards, this randomly generated optimisation would be included in the set of optimisations only if it was able to improve the performance of a training benchmark, also selected at random, in respect of the -O3 optimisation. This process was repeated until we obtained the 500 distinct optimisations.

6.3.2 Performance Evaluation

In order to evaluate the quality of the final optimisation selected by the iterative compilation search, we compare their speedup by measuring wall-clock time of the benchmarks when compiled with the default `-O3` optimisation. For each benchmark, after selecting the final optimisation, we compute the average speedup over all the 1000 input dataset. When measuring the wall-clock time for each input, in order to reduce noise, we execute the same input until we have a statistically sound measurement, i.e. we execute until we have an interval no larger than 1% with 99% confidence. Figure 6.14 shows the histogram of the speedups obtained for each benchmark with all their respective 1000 inputs. This figure shows the speedups of the best optimisation found by the oracle with real measurements of execution time (Oracle-RM).

Figure 6.14 is also important for showing that, although some of the benchmarks have a very consistent speedup for all their inputs, other benchmarks are highly sensitive to the input. For example, the optimisation with best average performance for the `adpcm_d` benchmark presents a wide range of speedups across all its inputs, varying from 0.7 up to about 1.5 of speedup. On the other hand, although the best optimisation selected for the `jpeg_d` benchmark has a much shorter range of speedups, there is a clear concentration of inputs around two different speedups. These results corroborate

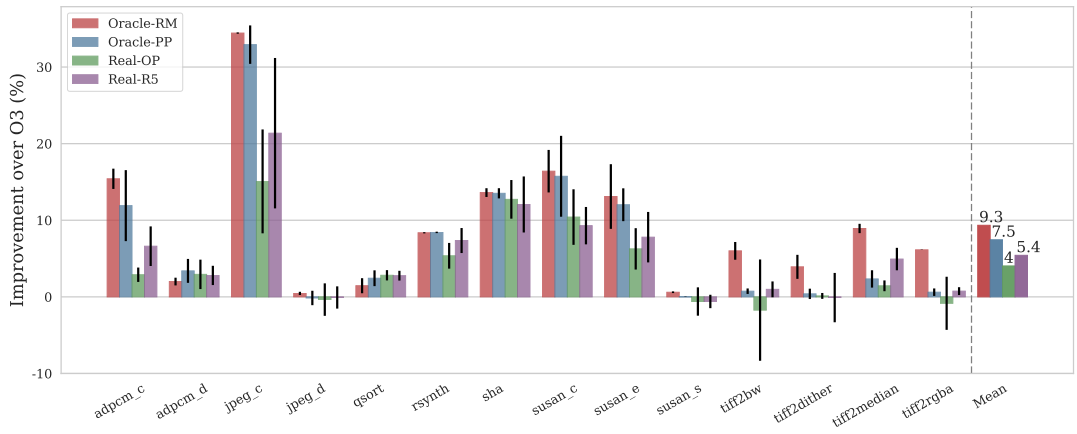


Figure 6.13: Speedups obtained from the final optimisation selected by the online iterative compilation. The speedups reported for each benchmark represents the average speedup across their complete 1000 input datasets.

the claim that performing iterative compilation on a single input can be misleading and cause the optimisation search to overfit.

6.3.3 Contribution of individual optimisation passes

Figure 6.15 shows an aggregated view of the final combination of compiler optimisations that were selected by iterative compilation search. The figure presents the individual optimisation passes with at least 1% of frequency in the selected combination of compiler optimisations that improved the performance over `-O3`.

6.3.3.1 Inter-Procedural Optimisations

Dead Global Elimination (`-globaldce`): It is an inter-procedural optimisation that eliminates unreachable internal globals from the program. It uses an aggressive algorithm that searches for global definitions that are known to be alive. After finding all live global definitions, it deletes all remaining globals.

Merge Functions `-mergefunc`: This pass looks for equivalent functions that are mergable and folds them. It introduces a function encoding which allows it to provide a total-ordering for the function set. The total-ordering allows to arrange functions into the binary tree. When iterating over the functions, it checks for an equivalent function in tree. If an equivalent function exists, both functions are merged, otherwise, the new function is added to the tree. This pass focus mainly on improving code size and memory footprint.

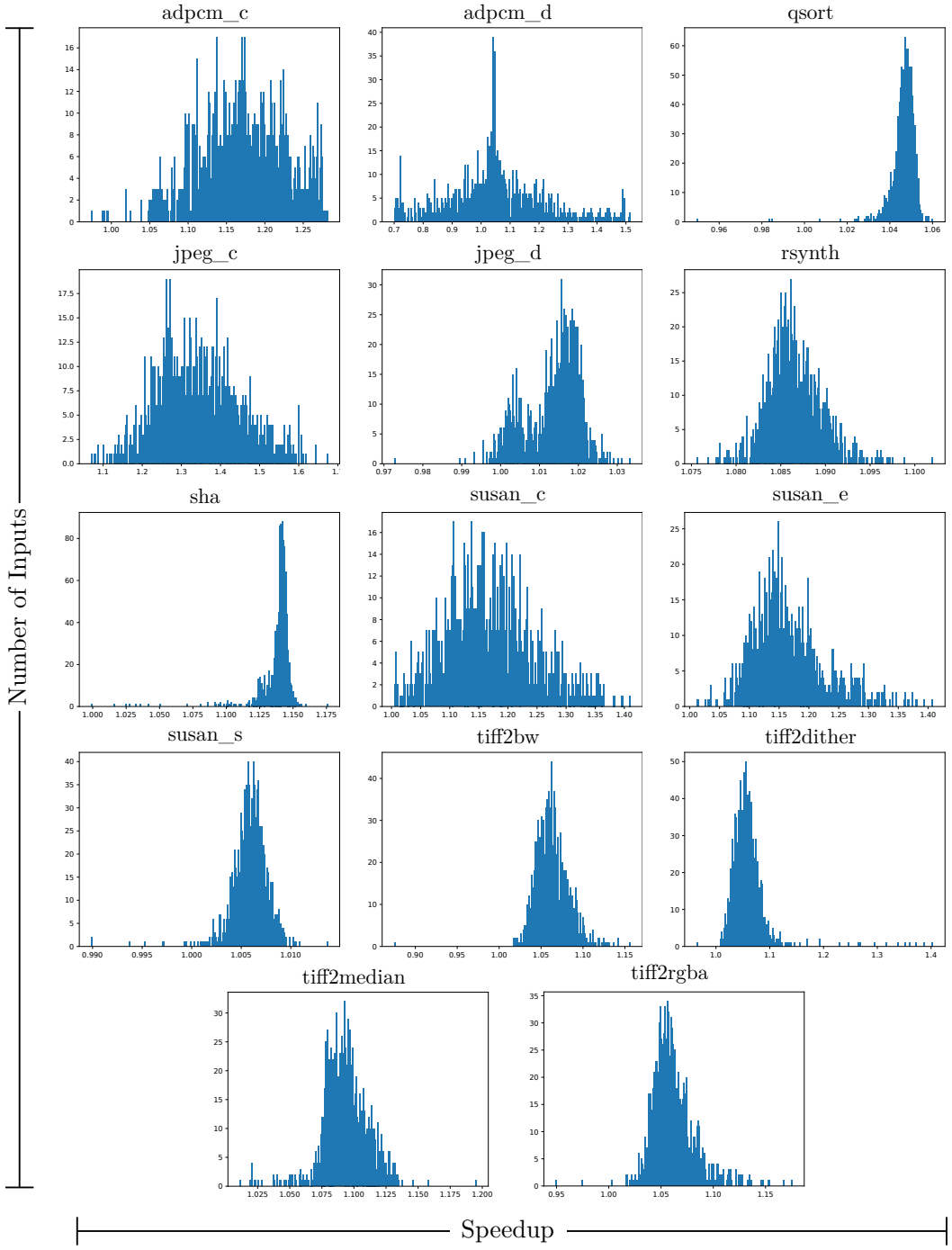


Figure 6.14: Histograms for the speedups over `-O3` with the complete dataset of 1000 inputs for each benchmark. In each case, we are using the optimisation with the best average speedup over `-O3`. This figure shows how the performance for some of the benchmarks are highly sensitive to the input.

6.3.3.2 Transformations of the CFG

Simplify the CFG `-simplifycfg`: This optimisation simplifies the CFG by basic blocks merging and dead code elimination, such as: eliminates a basic block that

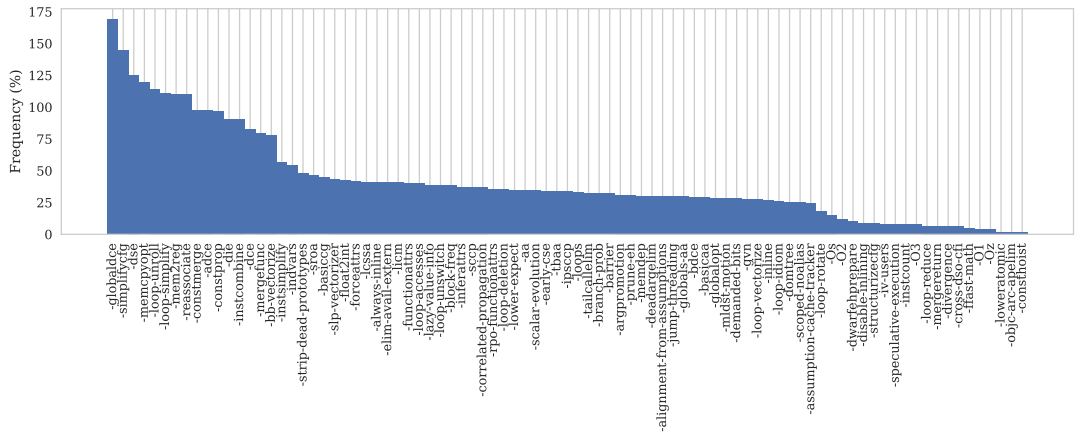


Figure 6.15: Frequency of individual optimisation passes on the final selected compiler optimisations of the iterative compilation search over all benchmarks.

only contains an unconditional branch; removes basic blocks with no predecessors; eliminates PHI nodes for basic blocks with a single predecessor; etc. This pass makes use of LLVM's built-in cost model to decide when it is beneficial for performing such transformations.

Canonicalize Natural Loops `-loop-simplify`: This pass performs several transformations to convert natural loops into a simpler form, which makes subsequent analyses and transformations simpler and more effective. Loop pre-header insertion guarantees that there is a single, non-critical entry edge from outside of the loop to the loop header. This simplifies a number of analyses and transformations, such as Loop Invariant Code Motion (LICM). Loop exit-block insertion guarantees that all exit blocks from the loop (blocks which are outside of the loop that have predecessors inside of the loop) only have predecessors from inside of the loop (and are thus dominated by the loop header). This simplifies transformations such as store-sinking that are built into LICM. This pass also guarantees that loops will have exactly one backedge. Note that the `-simplifycfg` pass will clean up blocks which are split out but end up being unnecessary, so usage of this pass should not pessimize generated code.

Unroll Loops `-loop-unroll`: This pass implements a simple loop unroller. It works best when loops have been canonicalized by the `-indvars` pass, allowing it to determine the trip counts of loops easily. This pass makes use of LLVM's built-in cost model to estimate the computational cost of both rolled and unrolled loops, which enables it to perform loop unrolling only when it is profitable.

6.3.3.3 Dead Code Elimination

Dead Instruction Elimination `-die`: Dead instruction elimination performs a single pass over the function, removing instructions that are obviously dead.

Dead Store Elimination `-dse`: A trivial dead store elimination that only considers basic-block local redundant stores.

Dead Code Elimination `-dce`: Dead code elimination is similar to dead instruction elimination, but it rechecks instructions that were used by removed instructions to see if they are newly dead. In other words, it performs dead instruction elimination until it reaches a fixed point.

Aggressive Dead Code Elimination `-adce`: ADCE aggressively tries to eliminate code. This pass is similar to DCE but it assumes that values are dead until proven otherwise. This is similar to SCCP, except applied to the liveness of values.

6.3.3.4 Simplification of Expressions

Combine redundant instructions `-instcombine`: Combine instructions to form fewer, simple instructions, by means of algebraic simplifications. This is a simple worklist driven algorithm, in a peephole fashion. Some of the simplifications and canonicalizations are: Constant operands are moved to the right-hand side of commutative binary operations; Multiplications with a constant power-of-two argument are transformed into shifts; All compare instructions on boolean values are replaced with logical operations. This pass can also simplify calls to specific well-known function calls (e.g. runtime library functions). For example, a call exit(3) that occurs within the main() function can be transformed into simply return 3.

Reassociate expressions `-reassociate`: This pass reassociates commutative expressions in an order that is designed to promote better constant propagation, Global Common Subexpression Elimination (GCSE), LICM, etc. It performs a canonical reordering of the operands of commutative binary operations.

Merge Duplicate Global Constants `-constmerge`: Merges duplicate global constants together into a single constant that is shared. This is useful because some passes (i.e., TraceValues) insert a lot of string constants into the program, regardless of whether or not an existing string is available.

Simple constant propagation `-constprop`: This pass implements constant propagation and merging. It looks for instructions involving only constant operands and replaces them with a constant value instead of an instruction This pass has a habit of

making definitions be dead. It is a good idea to run a Dead Instruction Elimination pass sometime after running this pass.

6.3.3.5 Other Frequent Transformations

MemCpy Optimization `-memcpyopt`: This pass performs various transformations related to eliminating `memcpy` calls, or transforming sets of stores into `memsets`.
Promote Memory to Register `-mem2reg`: This file promotes memory references to be register references. It promotes alloca instructions which only have loads and stores as uses. An alloca is transformed by using dominator frontiers to place phi nodes, then traversing the function in depth-first order to rewrite loads and stores as appropriate. This is just the standard SSA construction algorithm to construct pruned SSA form.
Basic-Block Vectorization `-bb-vectorize`: This pass combines instructions inside basic blocks to form vector instructions. The algorithm was inspired by that used by Franchetti et al. (2005). It iterates over each basic block, attempting to pair compatible instructions, repeating this process until no additional pairs are selected for vectorization. When the outputs of some pair of compatible instructions are used as inputs by some other pair of compatible instructions, those pairs are part of a potential vectorization chain. Instruction pairs are only fused into vector instructions when they are part of a chain longer than some threshold length. Moreover, the pass attempts to find the best possible chain for each pair of compatible instructions. These heuristics are intended to prevent vectorization in cases where it would not yield a performance increase of the resulting code.

6.4 Summary

Chapter 7

Conclusions and Future Work

Create a context-aware instrumentation in order to avoid updates to the global counter inside function calls in tight loops, for which function cloning could be employed. **Elaborate**

Merge probes in branches with similar work value. **Elaborate**

Perform a loop-aware relaxation which is able to considering inner loops where the upper bound of the trip count is known. **Elaborate**

Use the work-based metric with the work instrumentation in order to perform run-time auto-tuning or multi-versioning optimisations.

Bibliography

Adve, V., Lattner, C., Brukman, M., Shukla, A., and Gaeke, B. (2003). Llva: A low-level virtual instruction set architecture. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 36, pages 205–, Washington, DC, USA. IEEE Computer Society.

Agakov, F., Bonilla, E., Cavazos, J., Franke, B., Fursin, G., O'Boyle, M. F. P., Thomson, J., Toussaint, M., and Williams, C. K. I. (2006). Using machine learning to focus iterative optimization. In *International Symposium on Code Generation and Optimization (CGO'06)*, pages 1–11.

Alameldeen, A. R. and Wood, D. A. (2006). IPC considered harmful for multiprocessor workloads. *IEEE Micro*, 26(4):8–17.

Allen, F. E. (1970). Control flow analysis. In *Proceedings of a Symposium on Compiler Optimization*, pages 1–19, New York, NY, USA. ACM.

Alpern, B., Wegman, M. N., and Zadeck, F. K. (1988). Detecting equality of variables in programs. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, pages 1–11, New York, NY, USA. ACM.

Ball, T. and Larus, J. R. (1993). Branch prediction for free. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, PLDI '93, pages 300–313, New York, NY, USA. ACM.

Ball, T. and Larus, J. R. (1994). Optimally profiling and tracing programs. *ACM Trans. Program. Lang. Syst.*, 16(4):1319–1360.

Ball, T. and Larus, J. R. (1996). Efficient path profiling. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 29, pages 46–57, Washington, DC, USA. IEEE Computer Society.

Bodin, F., Kisuki, T., Knijnenburg, P., O' Boyle, M., and Rohou, E. (1998). Iterative compilation in a non-linear optimisation space. In *Workshop on Profile and Feedback-Directed Compilation*, Paris, France.

Brandolese, C., Corbetta, S., and Fornaciari, W. (2011). Software energy estimation based on statistical characterization of intermediate compilation code. In *Proceedings of the 17th IEEE/ACM International Symposium on Low-power Electronics and Design*, ISLPED '11, pages 333–338, Piscataway, NJ, USA. IEEE Press.

Cavazos, J., Fursin, G., Agakov, F., Bonilla, E., O'Boyle, M. F. P., and Temam, O. (2007). Rapidly selecting good compiler optimizations using performance counters. In *International Symposium on Code Generation and Optimization (CGO'07)*, pages 185–197.

Chen, Y., Fang, S., Eeckhout, L., Temam, O., and Wu, C. (2012a). Iterative optimization for the data center. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 49–60, New York, NY, USA. ACM.

Chen, Y., Fang, S., Huang, Y., Eeckhout, L., Fursin, G., Temam, O., and Wu, C. (2012b). Deconstructing iterative optimization. *ACM Trans. Archit. Code Optim.*, 9(3):21:1–21:30.

Chen, Y., Huang, Y., Eeckhout, L., Fursin, G., Peng, L., Temam, O., and Wu, C. (2010). Evaluating iterative optimization across 1000 datasets. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 448–459, New York, NY, USA. ACM.

Coppa, E., Demetrescu, C., and Finocchi, I. (2014). Input-sensitive profiling. *IEEE Transactions on Software Engineering*, 40(12):1185–1205.

Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., and Zadeck, F. K. (1989). An efficient method of computing static single assignment form. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, pages 25–35, New York, NY, USA. ACM.

Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., and Zadeck, F. K. (1991). Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490.

Dantzig, G. B. (1957). Discrete-variable extremum problems. *Operations Research*, 5(2):266–288.

Eyerman, S. and Eeckhout, L. (2008). System-level performance metrics for multiprogram workloads. *IEEE Micro*, 28(3):42–53.

Fang, S., Xu, W., Chen, Y., Eeckhout, L., Temam, O., Chen, Y., Wu, C., and Feng, X. (2015). Practical iterative optimization for the data center. *ACM Trans. Archit. Code Optim.*, 12(2):15:1–15:26.

Forman, I. R. (1981). On the time overhead of counters and traversal markers. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, pages 164–169, Piscataway, NJ, USA. IEEE Press.

Franchetti, F., Kral, S., Lorenz, J., and Ueberhuber, C. W. (2005). Efficient utilization of simd extensions. *Proceedings of the IEEE*, 93(2):409–425.

Fursin, G., Cavazos, J., O'Boyle, M., and Temam, O. (2007). MiDataSets: Creating the conditions for a more realistic evaluation of iterative optimization. In *Proceedings of the 2Nd International Conference on High Performance Embedded Architectures and Compilers*, HiPEAC'07, pages 245–260, Berlin, Heidelberg. Springer-Verlag.

Giusto, P., Martin, G., and Harcourt, E. (2001). Reliable estimation of execution time of embedded software. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '01, pages 580–589, Piscataway, NJ, USA. IEEE Press.

Goldsmith, S. F., Aiken, A. S., and Wilkerson, D. S. (2007). Measuring empirical computational complexity. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE '07, pages 395–404, New York, NY, USA. ACM.

Haneda, M., Knijnenburg, P. M. W., and Wijshoff, H. A. G. (2006). On the impact of data input sets on statistical compiler tuning. In *Proceedings 20th IEEE International Parallel Distributed Processing Symposium*, pages 1–8.

Kisuki, T., Knijnenburg, P. M. W., and O'Boyle, M. F. P. (2000). Combined selection of tile sizes and unroll factors using iterative compilation. In *Proceedings 2000 International Conference on Parallel Architectures and Compilation Techniques (Cat. No.PR00622)*, pages 237–246.

Kisuki, T., Knijnenburg, P. M. W., O'Boyle, M. F. P., Bodin, F., and Wijshoff, H. A. G. (1999). *A feasibility study in iterative compilation*, pages 121–132. Springer Berlin Heidelberg, Berlin, Heidelberg.

Knijnenburg, P. M. W., Kisuki, T., Gallivan, K., and O'Boyle, M. F. P. (2004). The effect of cache models on iterative compilation for combined tiling and unrolling. *Concurrency and Computation: Practice and Experience*, 16(2-3):247–270.

Knuth, D. E. (1971). An empirical study of FORTRAN programs. *Software: Practice and Experience*, 1(2):105–133.

Knuth, D. E. and Stevenson, F. R. (1973). Optimal measurement points for program frequency counts. *BIT Numerical Mathematics*, 13(3):313–322.

Kulkarni, P., Hines, S., Hiser, J., Whalley, D., Davidson, J., and Jones, D. (2004). Fast searches for effective optimization phase sequences. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, PLDI '04, pages 171–182, New York, NY, USA. ACM.

Kulkarni, S. and Cavazos, J. (2012). Mitigating the compiler optimization phase-ordering problem using machine learning. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 147–162, New York, NY, USA. ACM.

Lattner, C. and Adve, V. (2004). Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–, Washington, DC, USA. IEEE Computer Society.

Martello, S., Pisinger, D., and Toth, P. (2000). New trends in exact algorithms for the 01 knapsack problem. *European Journal of Operational Research*, 123(2):325 – 332.

Mpeis, P., Petoumenos, P., and Leather, H. (2016). Iterative compilation on mobile devices. In *Proceedings of the 6th International Workshop on Adaptive Self-tuning Computing Systems (ADAPT 2016)*.

Nahapetian, A. (1973). Node flows in graphs with conservative flow. *Acta Informatica*, 3(1):37–41.

Ogilvie, W. F., Petoumenos, P., Wang, Z., and Leather, H. (2017). Minimizing the cost of iterative compilation with active learning. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 245–256.

Pan, Z. and Eigenmann, R. (2006). Fast and effective orchestration of compiler optimizations for automatic performance tuning. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '06, pages 319–332, Washington, DC, USA. IEEE Computer Society.

Powell, D. C. and Franke, B. (2009). Using continuous statistical machine learning to enable high-speed performance prediction in hybrid instruction-/cycle-accurate instruction set simulators. In *Proceedings of the 7th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '09, pages 315–324, New York, NY, USA. ACM.

Shafer, G. et al. (1976). *A mathematical theory of evidence*, volume 1. Princeton university press Princeton.

Stephenson, M., Amarasinghe, S., Martin, M., and O'Reilly, U.-M. (2003). Meta optimization: Improving compiler heuristics with machine learning. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI '03, pages 77–90, New York, NY, USA. ACM.

Touati, S.-A.-A. and Barthou, D. (2006). On the decidability of phase ordering problem in optimizing compilation. In *Proceedings of the 3rd Conference on Computing Frontiers*, CF '06, pages 147–156, New York, NY, USA. ACM.

Wagner, T. A., Maverick, V., Graham, S. L., and Harrison, M. A. (1994). Accurate static estimators for program optimization. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, PLDI '94, pages 85–96, New York, NY, USA. ACM.

Wu, Y. and Larus, J. R. (1994). Static branch frequency and program profile analysis. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, MICRO 27, pages 1–11, New York, NY, USA. ACM.

Zaparanuks, D. and Hauswirth, M. (2012). Algorithmic profiling. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 67–76, New York, NY, USA. ACM.

Zhou, Y. Q. and Lin, N. W. (2012). A study on optimizing execution time and code size in iterative compilation. In *2012 Third International Conference on Innovations in Bio-Inspired Computing and Applications*, pages 104–109.