# How I Did It

*Rodrigo Caetano de Oliveira Rocha*


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

# Abstract

This doctoral thesis will present the results of my work into the reanimation of lifeless human tissues.

# Acknowledgements

Many thanks to my mummy for the numerous packed lunches; and of course to Igor, my faithful lab assistant.

# 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

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

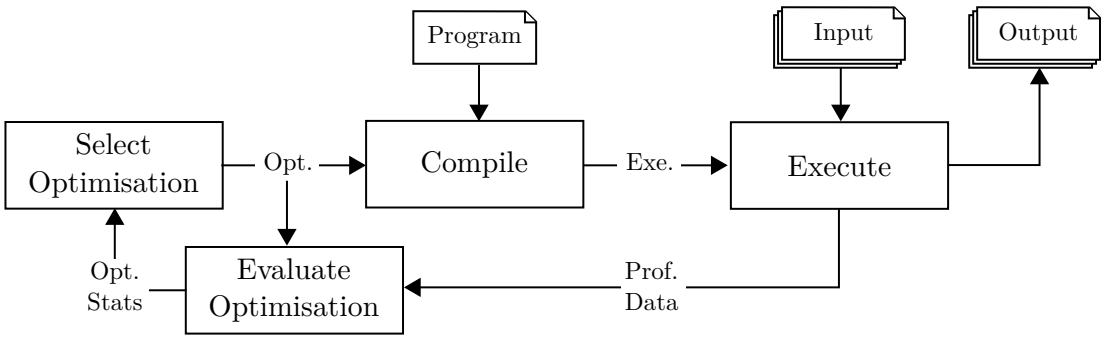



Figure 1.1: Overview of the execution engine for applying iterative compilation.

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 havea 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 Early Work in Iterative Compilation

**Original work in Iterative Compilation: single input and offline**

## 2.2 Program Profiling

**Basic background about profiling in general**

## 2.3 LLVM Compiler Infrastructure

LLVM is a compiler framework that implements the classical *three-phase compiler* infrastructure, which consists of a frontend, an optimiser, and a backend.

The frontend is responsible for parsing, validating and diagnosing errors in the source code. This parsed source code is then translated into LLVM Intermediate Representation (IR). The optimiser is responsible for doing a broad variety of transformations to try to improve the code's running time, such as eliminating redundant computations, and is usually more or less independent of language and target. The backend (also known as the code generator) then maps the code onto the target instruction set. In addition to making correct code, it is responsible for generating good code that takes advantage of unusual features of the supported architecture. Common parts of a compiler back end include instruction selection, register allocation, and instruction scheduling.

LLVM IR is a low-level RISC-like virtual instruction set. Like a real RISC instruction set, it supports linear sequences of simple instructions like add, subtract, compare,

and branch.

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.

The LLVM code representation describes a program using an abstract RISC-like instruction set but with key higherlevel information for effective analysis. This includes type information, explicit control flow graphs, and an explicit dataflow representation (using an infinite, typed register set in Static Single Assignment form [15]). There are several novel features in the LLVM code representation: (a) A lowlevel, language-independent type system that can be used to implement data types and operations from high-level languages, exposing their implementation behavior to all stages of optimization. This type system includes the type information used by sophisticated (but language-independent) techniques, such as algorithms for pointer analysis, dependence analysis, and data transformations. (b) Instructions for performing type conversions and low-level address arithmetic while preserving type information. (c) Two low-level exception-handling instructions for implementing languagespecific exception semantics, while explicitly exposing exceptional control flow to the compiler.

These instructions are in three address form, which means that they take some number of inputs and produce a result in a different register.5 LLVM IR supports labels and generally looks like a weird form of assembly language.

Unlike most RISC instruction sets, LLVM is strongly typed with a simple type system (e.g., i32 is a 32-bit integer, i32** is a pointer to pointer to 32-bit integer) and 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 doesn't use a fixed set of named registers, it uses an infinite set of temporaries named with a

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. The LLVM Project also provides tools to convert the on-disk format from text to binary: llvm-as assembles the textual .ll file into a .bc file containing the bitcode goop and llvm-dis turns a .bc file into a .ll file.

***LLVM IR is a Complete Code Representation*** In particular, LLVM IR is both well specified and the only interface to the optimizer. This property means that all you need to know to write a front end for LLVM is what LLVM IR is, how it

works, and the invariants it expects. Since LLVM IR has a first-class textual form, it is both possible and reasonable to build a front end that outputs LLVM IR as text, then uses Unix pipes to send it through the optimizer sequence and code generator of your choice.

It might be surprising, but this is actually a pretty novel property to LLVM and one of the major reasons for its success in a broad range of different applications. Even the widely successful and relatively well-architected GCC compiler does not have this property: its GIMPLE mid-level representation is not a self-contained representation. As a simple example, when the GCC code generator goes to emit DWARF debug information, it reaches back and walks the source level ”tree” form. GIMPLE itself uses a ”tuple” representation for the operations in the code, but (at least as of GCC 4.5) still represents operands as references back to the source level tree form.

The implications of this are that front-end authors need to know and produce GCC’s tree data structures as well as GIMPLE to write a GCC front end. The GCC back end has similar problems, so they also need to know bits and pieces of how the RTL back end works as well. Finally, GCC doesn’t have a way to dump out ”everything representing my code”, or a way to read and write GIMPLE (and the related data structures that form the representation of the code) in text form. The result is that it is relatively hard to experiment with GCC, and therefore it has relatively few front ends.

**IR, what is a Basic Block, SSA Form**

# Chapter 3

# Related Work

## 3.1 Iterative Compilation

**Specific work in Iterative Compilation about multiple inputs and online scenarios**

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?* Their results show that it is possible to find a combination of compiler optimisations that achieves at least 86% of the maximum speedup across all input test cases. These optimal combinations are program-specific and yield average speedups up to 3.75× over the highest optimisation level of compilers.

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.

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 eval-

uation 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 weighing 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). The optimal instrumentation was later studied by Forman (1981) and Ball and Larus (1994) in order to further reduce its overhead by placing the probes in basic blocks that are less likely to be executed.

**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.

[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
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: A data-flow analysis for populating all edge flows based on the probed edges.

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).

Algorithm 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).

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 weighing 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.

**Talk about extending to inter-procedural**

### 3.3.1 Extending to Interprocedural Instrumentation

### 3.3.2 Static Heuristic Estimations

**Talk about how Static Heuristic Estimations works** (Forman, 1981; Ball and Larus, 1994) Wu and Larus (1994): Static branch frequency and program profile analysis

# Chapter 4

# 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*, $\Delta W$, performed during a period of time, $\Delta t$.

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

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 $\Delta W$ 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.

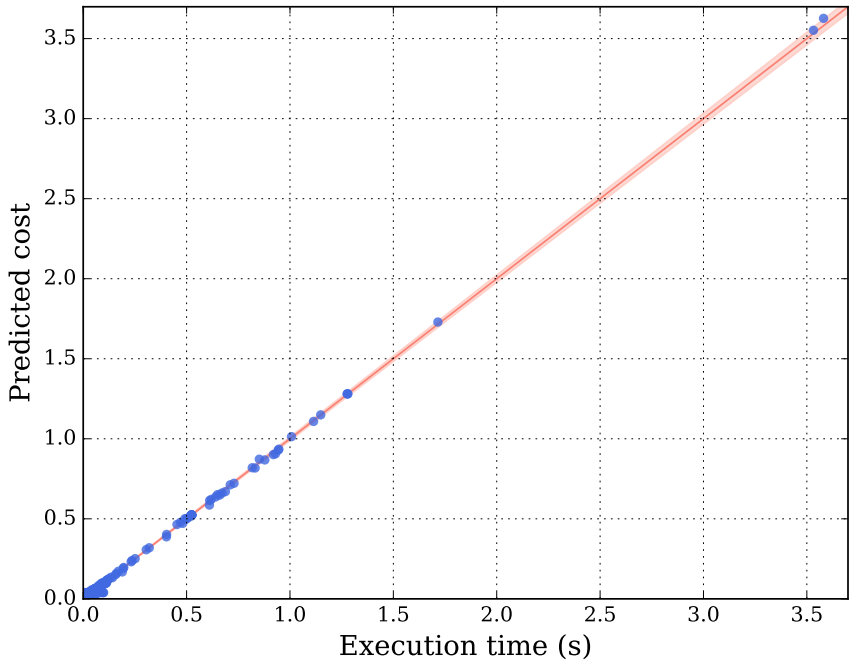



Figure 4.1: Linear model fitted from empirical data. The mean absolute error (MAE) for the fitted curve is of 7 milliseconds.

# 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, $\varepsilon$, 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. 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 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 is independent of optimisation is the most important reason for which we instrument the code before optimisations.

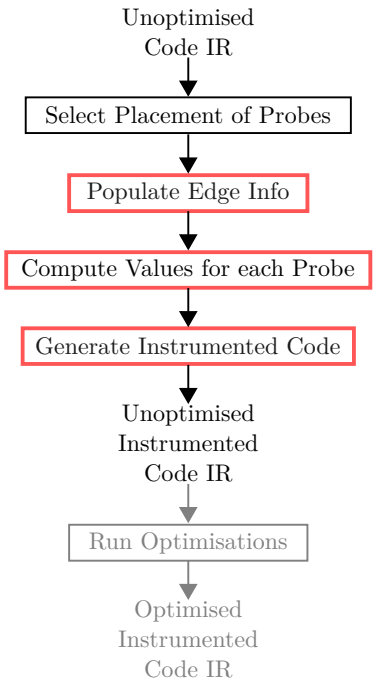



Figure 5.1: Overview of the work instrumentation algorithm.

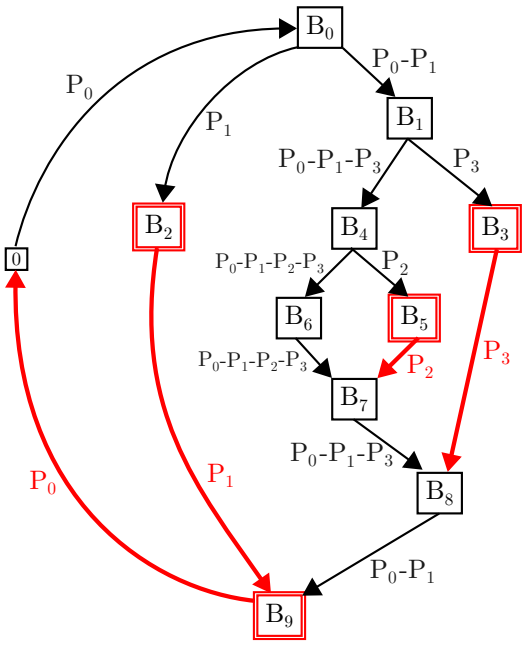
The optimal placement of the probes works exactly as previously described in Section 3.3. It first computes the maximum spanning tree based on a weighing that assigns a non-negative value to each edge in the CFG. These weights can be obtained either by empirical measurements or heuristic estimations, 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 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 $w(P_0)$ 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 in order to compose the aggregated values that will be instrumented in each probe, based on our model of computational work, $\Delta W$, derived from the basic block frequencies (see Section 4). We perform a similar propagation of the probes in a sym-

```
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.




Instrumented value for each probe $P_i$:

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

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.

bolic fashion, as illustrated in Figure 5.2. This symbolic propagation of the probes is implemented by the data-flow analysis described in Algorithm 5.1, and the final aggregated values are extracted from the edge information as described by Algorithm 5.2.

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

Algorithm 5.2 reads the edge information for each basic block by computing the

```
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.

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 Algorithm 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 6.1). 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 *extended* DAGs (directed acyclic graphs) from the CFG, as illustrated in Figure 5.4. Each loop (and the outer most region of the function) represents an extended DAG where any inner loop is considered to never execute, i.e. only the headers of the inner loops are included into the extended DAG.

For every extended DAG with a set of probes $\{P_0, P_1, \ldots, P_k\}$, we relax the instrumentation 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:

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

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

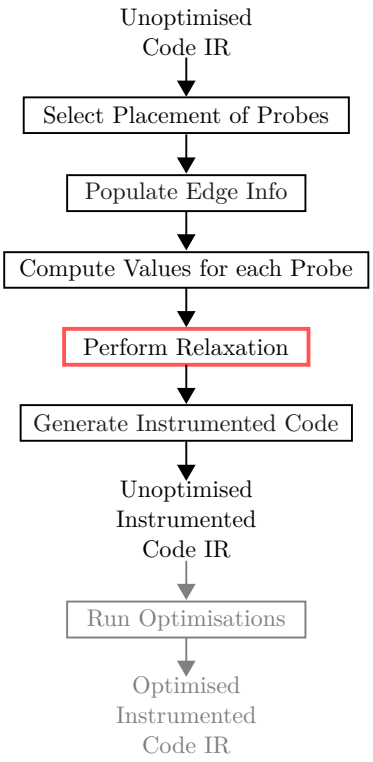



Figure 5.3: Diagram.

where $f(P_i)$ is the execution frequency of the instrumented basic block $P_i$, $\varepsilon(P_i)$ is the percentage error of removing probe $P_i$ relative to the minimum work value possible to compute in the extended DAG, and $x_i$ denotes the probes selected for removal. 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 extended DAG below a given threshold we guarantee that the final error of the relaxation will always be bounded by the threshold.

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

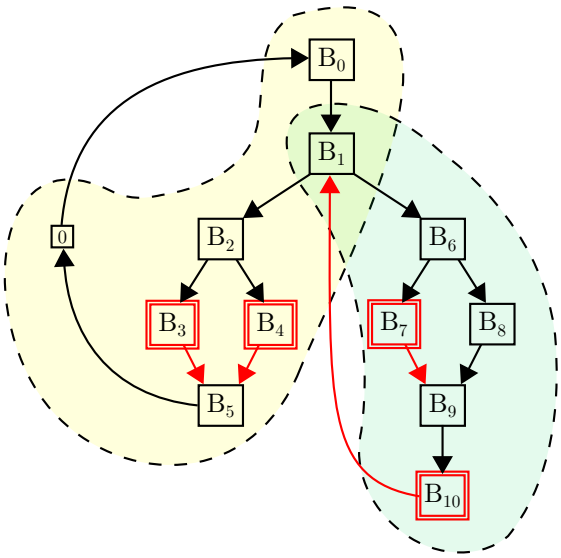



Figure 5.4: Example of a CFG containing a loop and its decomposition into extended DAGs when applying the relaxation.

compilation time.

## 5.3 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 benchmark and dataset suit is shown in Table 6.1.

## 6.2 Evaluation of the Instrumentation

Figure 6.2 shows percentage of instrumented basic blocks for the optimal and the relaxed instrumentation with different relaxation thresholds. The naive instrumentation always has 100% of the basic blocks instrumented, by definition.

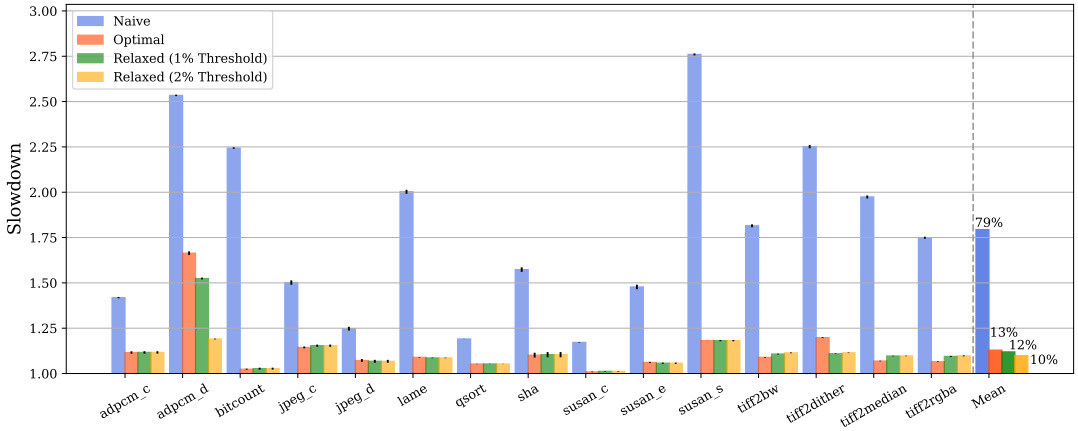



Figure 6.1: Overhead of the instrumentations compiled with `-O3`.

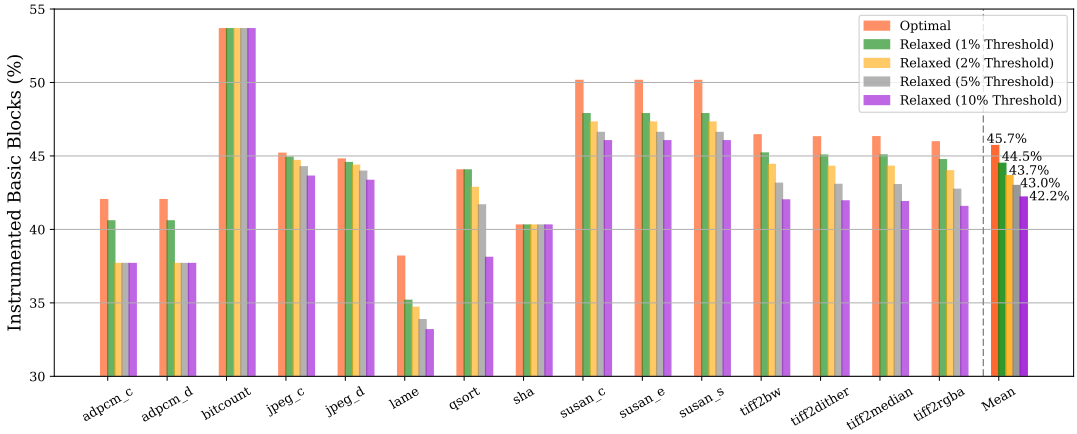



Figure 6.2: Percentage of instrumented basic blocks for the optimal and the relaxed instrumentation with different relaxation thresholds.

| 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 |
| rsynth | 4111 | | Text files |
| sha | 197 | 0.6K-35M | Files of any format |

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

The `adpcm_d` benchmark is the most critical case amongst the used benchmarks, with a overhead of about 66% with the optimal instrumentation. This benchmark consists mainly of a single hot loop with several branches inside it. The relaxation algorithm is able to reduce this overhead down to about 52% (with an error threshold of 1%) and 19% (with an error threshold of 2%) by removing only one and two probes from the hot loop, respectively. These two removed probes were placed in branches, inside the hot loop, with a high probability of being taken, but with a small contribution to the work measured in the loop.

## 6.3 Evaluation of the Online Iterative Compilation

- Oracle-RM executes the program twice, for each input, and then measuring the real speedup for each compiler optimisation and uses the real speedup for selecting the best optimisation.

- Oracle-PP represents the *perfect* non-intrusive profiling by also executing the program twice, once for estimating the amount of work and the second for measuring its execution time. This version uses the work-based metric, $\frac{\Delta W}{\Delta t}$, during the iterative compilation search.

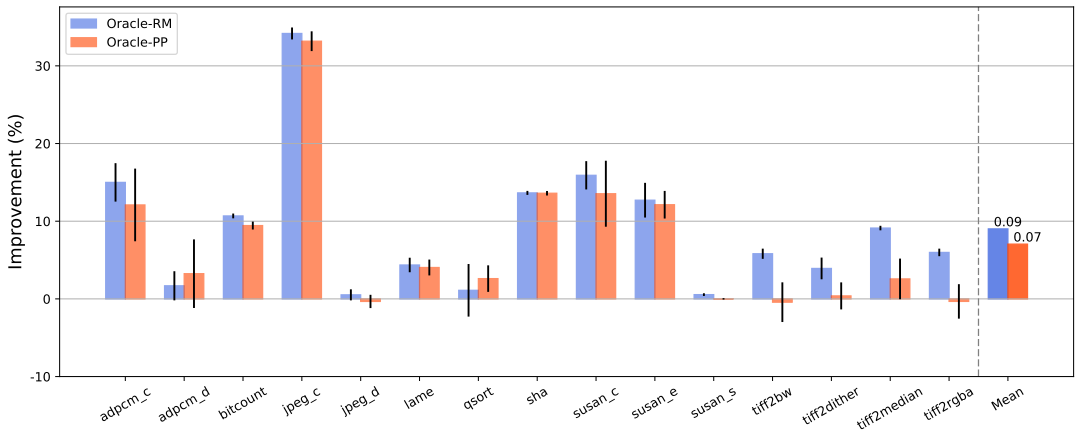



Figure 6.3: Speedups observed with the online iterative compilation.

# Chapter 7

# Conclusions

# Bibliography

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.

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.

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.

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., 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.

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.

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.

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.

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.

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.