Universiteit Leiden
The Netherlands

Opleiding Informatica

An initial exploration of the Importance of Program Instruction Order for Dynamically Scheduled Processors

Mariska IJpelaar

Supervisors:
Dr. Kristian F. D. Rietveld & Dr. A. Uta

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

August 6, 2020

Abstract

Instruction Scheduling has a major performance impact on processors that only schedule statically. As they can only execute the instructions of a program exactly in the order they are given, a good scheduler is essential. Optimal instruction scheduling is even proven NP-complete. When Dynamic Scheduling was introduced, it led to the idea that compilers do not optimally schedule code, as the hardware could do better due to the availability of runtime information. Many even believed that static scheduling had become redundant. Nowadays, many question this opinion. Literature supports arguments in both sides, but there is an absence of experimental validation.

In this work, we investigate whether program instruction order has any measurable influence on performance in runtime, when executed on dynamically scheduled superscalar processors. To answer this question we generated all valid permutations of a function within a simple program. Creating these permutations is not trivial and requires dependencies and side effects to be taken into account.

We found that program instruction order indeed has measurable influence on performance in runtime when executed on a dynamically scheduled superscalar processor. Some additional analyses have shown no direct influence of the mean and standard deviation of the permutations. However, some interesting results show promise of potential relations, for instance between the distribution in type of instructions and runtime.

Contents

1 Introduction

"80% of the execution time is usually spent executing no more than 20% of the code",
a rule of thumb derived from the Pareto Principle (by Vilfredo Pareto).
It illustrates the importance of knowing the bottleneck of a program in order to improve its performance. Some believe optimizing code without a good analysis may only make it worse. Donald Knuth [1] even called premature optimization "the root of all evil". In short, code optimization is important and definitely not trivial.

A program consists of instructions and can be executed in various orders. Instruction Scheduling is the process of constructing one such order. Usually, the aim in this task is to find the order with the highest efficiency. This is not trivial, and has even been proven NP-Complete by Henessy and Gross [2]. In conclusion, a good instruction scheduler is vital for having an efficient execution of a program.

Instruction Scheduling in general can either be done at compile-time or run-time. Scheduling at compile-time is called static scheduling and scheduling at run-time is called dynamic scheduling. On the one hand, dynamic scheduling could make static scheduling redundant, as the hardware can reorder all statically scheduled instructions. Tate et al. [3] even claim that aggressive static scheduling hinders the speedup obtained by dynamic scheduling. On the other hand, it is believed that there is always room for hand-tuning (Programmer's Wisdom). Holler [4] believes aggressive compiler optimization is key in the PA-8000, a dynamically-scheduled machine, being the most powerful microprocessor at its time. Literature seems to show support for both arguments, also stated by P.Faraboschi et al [5]: "...superscalar hardware sometimes rearranges code at runtime. This led early superscalar architects to maintain that compilers need not schedule code, since the hardware would do the job instead. Few researchers believe this today, rather most feel that the bulk of the rearrangement to be done must be done in advance in each case". However, there is an absence of experimental validation of either arguments.

The main question we try to answer in this thesis is: "Does program instruction order have any measurable influence on performance in runtime when executed on dynamically scheduled superscalar processors?" We will answer this question by generating and executing all valid permutations of a function within a simple program, run on a *common* desktop processor (Intel Core i7). Through experiments and analyses we try to find significant differences. The main contributions of this work are:

1. Two extensions of the Egalito [6] framework named 'Etreorder' and the 'Reorder Pass', to generate all valid permutations of a function within a program.

2. An initial exploration of the influence of program instruction order on performance in runtime.

In the next section we discuss the complexity of permutation generation and provide some background information and terminology. In Section 3 we discuss related work. The design and implementation of our extensions is covered in Sections 4 and 5. In Section 6 we describe our experiments, followed by their results in Section 7. Finally, we provide our conclusions in the last Section.

2 Background

In this work we aim to measure the influence of program instruction order on program runtime when executed on dynamically scheduled superscalar processors. In this section we provide some background information on instruction scheduling, program dependencies and compiler code generation.

2.1 Instruction Scheduling

Instruction Scheduling is the process of constructing an order for the instructions of a program. In general, it can either be done at compile-time or run-time. Scheduling at compile-time is called static scheduling, and aims to prevent stalls at in-order pipelines. Scheduling at runtime is called dynamic scheduling, and aims to eliminate Write after Read and Write after Write hazards. Because of the available runtime information, dynamic scheduling is also able to lighten the effects of unpredictable events, such as L1 cache misses, and is able to work ahead after a branch prediction. As the compiler cannot be certain about such events, this ability grants a significant advantage over static scheduling. Many of these dynamically scheduled processors are superscalar, which means they are capable of simultaneously dispatching multiple instructions to different execution units.

2.2 Program Dependencies

In order to measure the effect of instruction order on program runtime, we generate a variety of permutations of a program. We ensure the correctness of these permutations by analysing the explicit and implicit dependencies between instructions before permuting.

We now cover the examples of assembly code (Intel x86_64) from Figure 1 to consider the complexity of the dependency analysis.

```
addq  %rax, %rsi
movq  %rcx, %rdx
```

(a) No dependencies

```
addq  %rax, %rsi
movq  %rsi, %rcx
```

(b) An explicit Read After Write dependency

```
addq  %rax, %rsi
movq  %rax, %rcx
```

(c) An explicit Read After Read dependency

```
addq  %rax, %rdx
movq  %rcx, %rdx
```

(d) An explicit Write After Write dependency

```
addq  %rax, %rbx
addq  %rcx, %rdx
```

(e) An implicit dependency

Figure 1: Examples of Pairs of Assembly Code Instructions (Intel x86_64) with varying Dependencies.

We can easily see that there are no dependencies between the two instructions of the first example in Figure 1a. The execution order of the program does not matter. In the example of Figure 1b, however, the order is important because of an explicit dependency on the used registers. This is a so called *Read After Write (RAW)* dependency; the value of `%rax` is read by `%addq` after the write from `%movq`. Other types of dependencies are: *Read after Read (RAR)*, *Write after Write (WAW)*, and *Write after Read (WAR)*.
It is easy to see *Read after Read* is not a real dependency. This is evident in the example in Figure 1c. Changing the order of the two instructions does not change the value in the registers after execution. Thus, this dependency does not give a restriction on the order. The Write after Write and Write after Read dependencies however, are only dependencies in name. Changing one of the registers solves the dependency. Figure 1d contains an example of this. Both instructions write to `%rdx`, thus we have a *WAW* dependency. However, by changing the first instruction to `addq %rax, %rbx`, this dependency disappears. Of course, this only applies if it creates no other dependencies on `%rbx`.
Until this point, we have only covered explicit dependencies, but we must be careful about implicit dependencies too. For the example in Figure 1e, it seems there are no dependencies. However, it is possible for one or both of these instructions to overflow. When an overflow happens, a bit is set in the register `eflags`. Even though this happens implicitly, the value of the register can be read by a subsequent instruction, in which case a dependency does exist and the order of the two instructions matters.

2.3 Code Generation

Instruction scheduling can be done at compile-time, by the compiler. To achieve this, the compiler uses many techniques to analyse and optimize code as explained in the Red Dragon Book [7]. In this part of the Section, we cover the process of code generation.

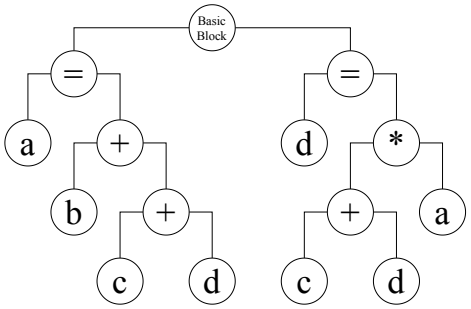
We now give a rough description of this process. Figure 2 can be referenced for an example of several phases. The first phase starts with the code of the programming language. After the lexical analysis and token parsing, an Abstract Syntax Tree (AST) is built. This tree is an abstract representation of the hierarchical syntactic structure of the code. It is abstract in the sense that this tree is architecturally independent. An example is shown in Figure 2b. From the Abstract Syntax Tree, an Intermediate Representation is constructed in the form of "three-address code". For this construction, the compiler uses Basic Blocks, sequences of statements always executed consecutively, without any branching. Figure 2a and Figure 2c show an example of the Basic Block and Intermediate Representation respectively.

The compiler optimizes code using the Abstract Syntax Tree and Intermediate Representation. As an example we can look at the Abstract Syntax Tree in Figure 2c and see the appearance of the *common subexpression* `c + d`, as we have two equal subtrees. Uncovering and removing these duplicate subexpressions is called *common subexpression elimination*.
The Intermediate Representation is commonly used to perform peephole optimizations, a process in which a window of instructions, called the peephole, is optimized.

Instruction selection and scheduling takes place in the last phase of the process, in which the intermediate representation is transformed into executable code. The compiler has a certain degree of freedom in choosing instructions and encodings, dependent on the architecture, that lead to the same result. However, the selection and scheduling must conform to the dependencies between the instructions. The compiler discovers and analyses these by using a Directed Acyclic Graph (DAG), as illustrated in Figure 2d. The DAG can be created at either the Abstract Syntax Tree (AST) or Intermediate Representation (IR) stage.
In this work, we are interested in the scheduling part of the last step. Thus, we are going to look into changing the order of instructions, but not the instructions themselves.

```
a = b + c + d
d = (c + d) * a
```

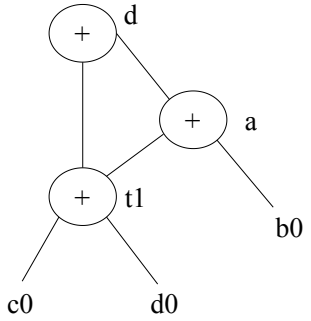
(a) Sample Code in the form of a Basic Block



(b) Abstract Syntax Tree of Sample Code in Figure 2a. Each node is an operator, and its children are the operands. We have named the root 'Basic Block' to indicate this tree is about the whole Basic Block. The leaves are the variables of the code.

```
t1 = c + d
a = b + t1
d = t1 * a
```

(c) Intermediate Representation (IR) in the form of Three-Address Code constructed from the Abstract Syntax Tree in Figure 2b



(d) Directed Acyclic Graph (DAG) of the IR in Figure 2c. The leaves denote the initial values of the variables. Every node contains an operator, and the variable to which it writes

Figure 2: Toy Example of several phases in the Code Generation Process.

3 Related Work

There is a large body of literature on instruction scheduling and performance characteristics of different schedule methodologies. However, as far as we know, the specific question of this thesis, whether instruction scheduling has influence on dynamic scheduling, is not addressed in the literature. In this section, we discuss work on instruction scheduling, out-of-order processors and end with work on register pressure in the instruction selection phase.

3.1 Scheduling Heuristics

Hennessy and Gross [2] proved that instruction scheduling is NP-Complete. Therefore, much is done in finding good heuristics, such as done by Russell [8], who developed a method for automate heuristic design. However, not all articles try to find heuristics. For example, Shobaki [9] presented the first set of algorithms to optimally schedule traces and superblocks. A trace is a sequence of basic blocks and a superblock is a trace with a single entry, but possibly multiple exists.

Good scheduling is especially important for processors which benefit from Instruction-Level-Parallelism (ILP) such as statically scheduling Superscalar or Very Long Instruction Word (VLIW) architectures. Without good scheduling, these two cannot fully utilize their capabilities, as explained by Faraboschi et al. [5], whose focus is on VLIW-type instructions. Chou and Chung [10], however, focused on superscalar processors in general. In their more theoretical paper, they give two models and study the interrelations between instructions and partial schedules.

3.2 Out-Of-Order Processors

McFarlin et al. [11], studied the Out-of-Order processors relative to In-Order processors. They explain that the out-of-order processors plays a dominant role with respect to the in-order processors. However, they did not study whether permuting instructions statically has any influence on the runtime performance of the out-of-order processors. On the contrary, they believe an in-order architecture cannot match the performance of the out-of-order architecture, without utilising some features of an out-of-order machine.

Holler [4] discusses a number of optimization techniques for the PA-8000, a dynamically scheduled machine. Although this paper is focused on one very specific processor, it gives some promising results. One example is the effectiveness of loop unrolling for this machine. Holler also connects hardware innovation with compile-time scheduling. She explains some may expect that the lengthy reorder queue and wide issue of the PA-8000 subsume instruction scheduling. However, the PA-8000 is only the most powerful microprocessor at the time due to both innovative hardware and aggressive optimization, such as static scheduling. So, even though this paper is about a specific out-of-order processor, it showed that instruction scheduling has influence in the runtime of programs executed on dynamically scheduled superscalar processors.

Tate et al. [3] also focus more on the influence of static scheduling for the out-of-order processors. They examined how the scheduling process hinders the performance for out-of-order instruction issue. Their experiments were run on an extension of an instruction level simulator. They state that static instruction scheduling hinders the speedup achieved by an out-of-order microprocessor. However, they do suggest that the compiler should aim at minimizing the number of true dependencies to improve performance. There is however, one major downside to this paper; they assume an ideal environment, such as perfect caches with single cycle latencies. This is not a true representative for an Out-of-Order Instruction Issue Processor. In this work, we execute all experiments on a physical out-of-order processor, to get realistic results.

3.3 Register Pressure

Rawat et al. [12], show that register spills are a performance bottleneck for scientific applications executed on out-of-order processors. Register spill occurs when a value is stored in memory while it will be used again later on, resulting in an unnecessary load. As loads and stores are expensive operations, work in preventing these spills is important. For this reason, looking into *register allocation* can be very useful.
The scope of this work is limited to instruction scheduling, mainly because the programs we worked with were too simple to cause register spills. However, even in this scope, we noticed the influence registers can have, especially when permuting instructions for fixed registers. Valluri et al. [13] discuss this order of the phases of register allocation and instruction ordering, as they have conflicting interests. The aim in Register allocation is reducing the life range of a variable to prevent register spill. However, in the instruction scheduling phase, this aim shifts to minimizing the amount of nearby dependencies and extending the life ranges of a variable. If register allocation precedes instruction scheduling, it is called *Postpass* method, otherwise it is called *Prepass* method.

4 Design

The aim of this work is to generate all valid permutations to measure the effect of instruction reordering. Although the idea of simply permuting the instructions may seem trivial, we should ensure that the resulting permutations are valid programs generating correct results. This implies we need to take dependencies into account. Often, this can be achieved using a Directed Acyclic Graph (DAG). This generation can be accomplished by, for instance, the method described by Chou and Chong [10], resulting in the DAG from Figure 3. The complexity of this DAG is a good demonstration for the difficulty of the instruction scheduling problem.

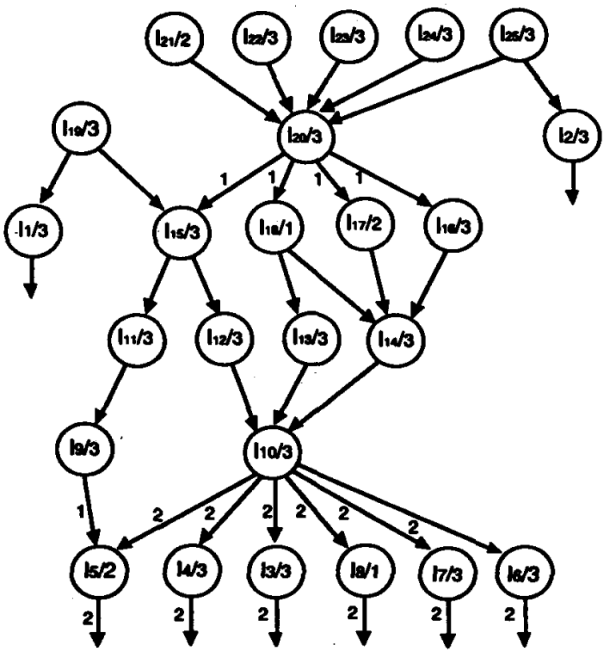


Figure 3: Sample Instruction DAG taken from Chou and Chung [10].

Rather than writing our own implementation, we opted to investigate whether compiler frameworks contain a suitable implementation that we could reuse for our experiments. For this we looked at LLVM and Fracture. LLVM is a set of compiler and toolchain technologies which can be used as a compiler framework. It was introduced in a paper by Chris Lattner and Vikram Adve [14]. Fracture is an open source architecture-independent decompiler to the LLVM Intermediate Representation by the Charles Stark Draper Laboratory [15].
The first part of this section discusses the problems these two tools posed. In the second part of this section, we discuss Egalito: a binary recompiler, introduced in the paper by Williams-King et al [6].

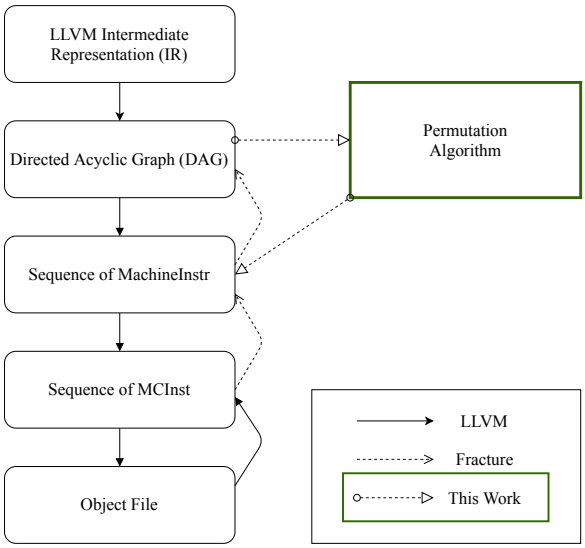


Figure 4: Fracture and LLVM
The filled arrows represent the steps supported by LLVM itself. The dashed arrows represent the steps provided by Fracture. The open arrows are the steps for our extension, the idea outlined in green.

4.1 LLVM and Fracture

LLVM implements all steps to go from its Intermediate Representation (IR) to executable code stored in an object file. The diagram in Figure 4 shows this process. As a first step, the Intermediate Representation is transformed to a Directed Acyclic Graph. Next, the nodes of the DAG are turned into a valid sequence of instructions, stored as instances of the `MachineInstr` class. This step is in fact a form of instruction scheduling. The target-dependent `MachineInstr` class abstractly represents machine instructions. A number of further optimizations are performed at this level, but these are out of scope of this work. Finally, the sequence of `MachineInstr` is converted into a sequence of `MCInst`, a simpler class that still represents the target machine code, and subsequently into the binary encoding of the instructions stored in the final object file.
We want to create the permutations of programs as stored in the object files. So, we need the DAG corresponding to the final program. As can be seen in Figure 4, there are multiple steps between the DAG and the final object file. In these steps, the DAG is no longer updated. As such, this DAG does not correspond to the final code emitted to the object file. To overcome this, there are two possibilities:

1. We implement the code necessary to keep the DAG up-to-date in the steps towards the final object file. We perceived this to be very hard: LLVM code generation is a vast code base as it deals with many corner cases, also the DAG datastructure that is used is complicated. An example DAG can be seen in Figure 5. It represents C code that performs a multiplication of two `long` variables.

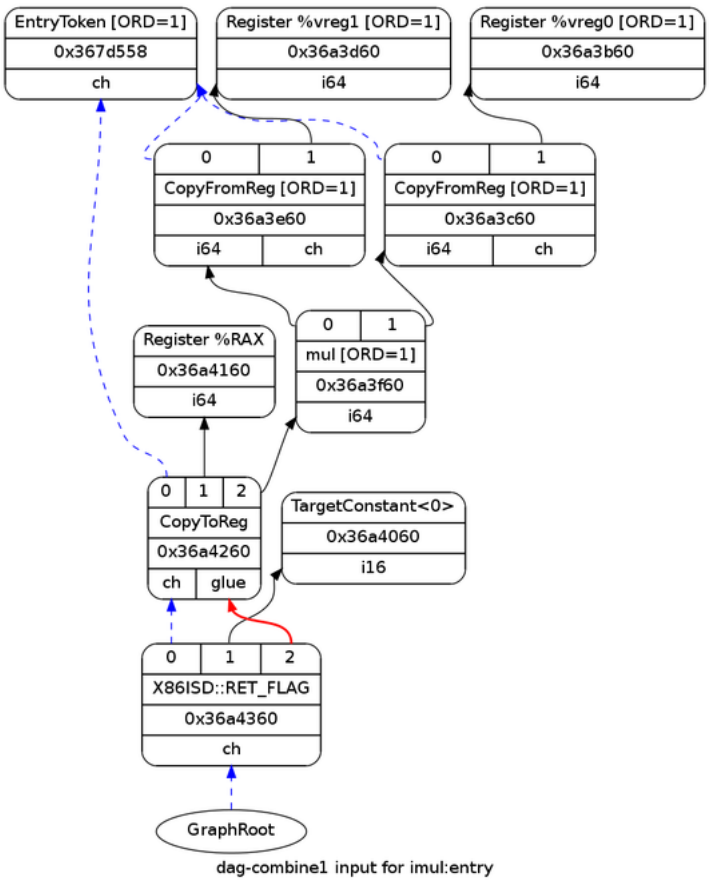


Figure 5: Sample LLVM DAG for a multiplication of two *long* variables. A node can be an operation, a register, a 'CopyToReg' or a 'CopyFromReg'. The arrows indicate the dependencies, as explained by Thuy and Berg [16]. The black arrows represent the natural arrangement between two nodes, often operand and operator. The dashed blue arrows represent chains to enforce relationships between nodes, to handle for example accesses to the same memory location. The red arrows *glue* two nodes together, to ensure they are executed directly after each other. Figure Source: [17]

2. We considered it more productive to explore the reverse direction and construct a DAG based on the final object code. LLVM itself supports the step back from object file to `MCInst`, but no further. For the steps from `MCInst` to DAG we have the tool Fracture. The idea, as illustrated in Figure 4, is to perform a Permutation Algorithm on the generated DAG, and recompile all permutations using the 'usual' LLVM way.
   However, Fracture was still a work in progress and had not received any modifications in 5 years. Nevertheless, enough was implemented to work with it and Fracture did seem to provide correct output. Yet, because Fracture was unfinished, the dependencies were too strict to allow the amount of permutations we wanted. The DAG (re)created by Fracture was somewhat simpler, but was still hard to interpret. To get an idea about the complexity, Figure 6 shows a simplified DAG from fracture for a toy example of only two instructions.

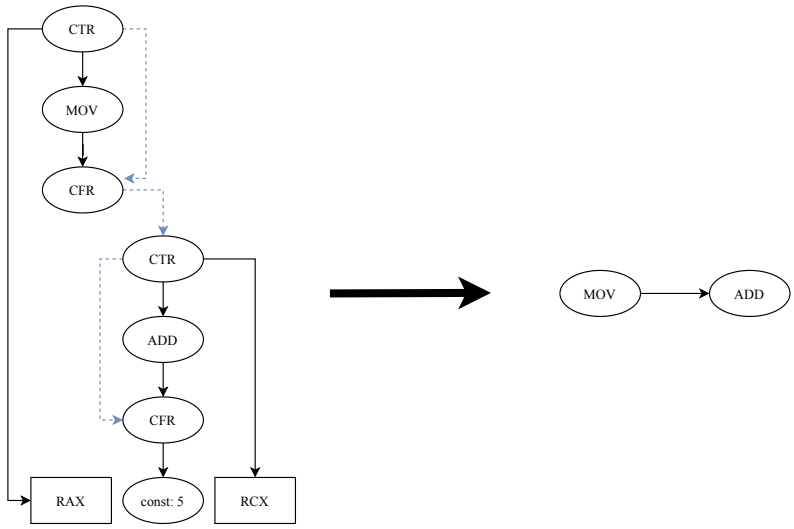


Figure 6: Toy example conversion for a DAG created by Fracture (simplified) of the two instructions *MOV* and *ADD*. *const* 5 represents the constant integer 5. *CTR* and *CFR* are 'CopyToRegister' and 'CopyFromRegister' respectively. The arrows represent the same as explained in Figure 5.

4.2 Egalito

Egalito was designed specifically for binary hardening and security mechanisms. To this end, Egalito needs to parse existing Linux binaries and implement various analyses to enable transformations. An overview of the Egalito framework can be seen in Figure 7. In fact, Egalito implements the necessary reverse step that we were trying to achieve with LLVM and Fracture. Egalito includes many features implemented in so-called *passes* and is designed so new passes can be added easily. One of these passes is called *ReorderPush*, which generates a random instruction reordering. We used this pass as an inspiration to create all instruction reorderings. We used the class *ReachingDef* to generate and keep track of all instruction dependencies.
In conclusion, Egalito makes working with DAGs redundant as it handles dependencies for us.

4.3 Extending Egalito

The code of the Egalito framework is subdivided into different sections. For this work, we only focus on the EgalitoInterface and the Passes. Figure 8 shows an overview of the relation between Egalito and our extension. The EgalitoInterface is an interface for working with Egalito outside the tool. The Passes of Egalito each provide their own functionality for a certain goal. For the Pass ReachingDef this is finding and keeping track of all dependencies. For the Permute Pass, our extension pass, this is generating all valid permutations of a function. We added Permute to the Passes of Egalito, which uses the Pass ReachingDef. Etreorder is not part of the Egalito tool, but uses its functionality through the EgalitoInterface and our Permute Pass.
We faced one obstacle during the implementation of our extension, however; Egalito was not designed to repeatedly transform the same program within a single invocation. As a result of this, Egalito does not explicitly release memory. A number of modifications were made to reduce the memory consumption across resets of the internal state of Egalito.

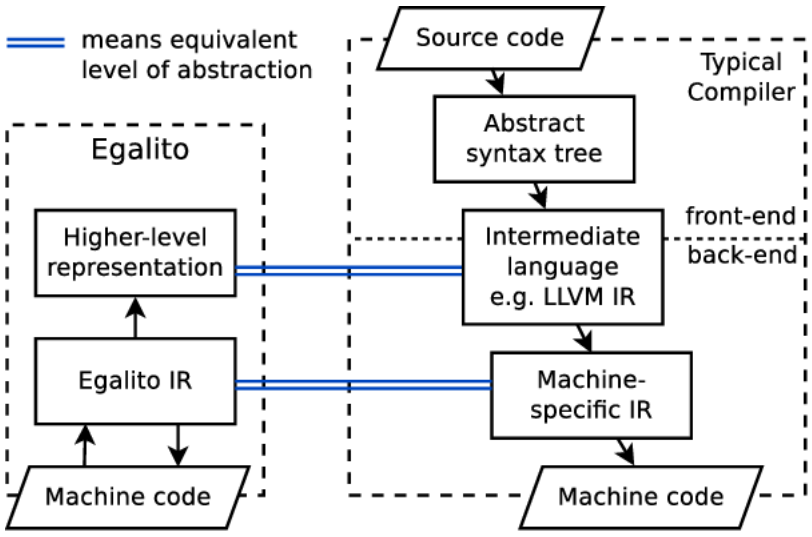


Figure 7: Egalito IR (EIR) design. Egalito reverses a compiler backend to obtain an IR similar to a machine-specific IR, and augments it with higher-level data structures [6].

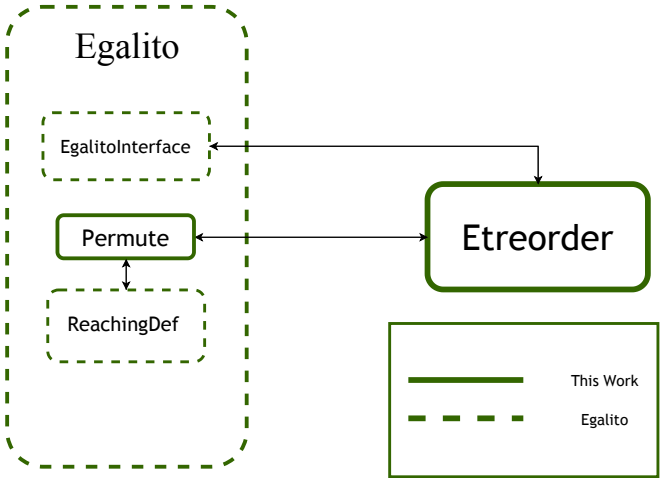


Figure 8: An overview of the relation between Egalito and our extension. The thick (green) lines indicate our extension and the dashed (green) lines indicate the Egalito framework.

5 Implementation

In this section, we elaborate on the implementation of our extension of the framework Egalito, as explained in Section 4.3. We start with going into detail about the different phases of Etreorder. This is followed by a small discussion on the criteria for our extension to be considered correct. In the end, we combine this with the actual implementation of the Permute Pass.

5.1 Etreorder

Etreorder generates all valid permutations of a function of a program. Its name is based on the name-convention of the Egalito applications: `et + application-name`. Etreorder can be subdivided into three phases: parsing, generating, transform and write.

In the **parsing** phase, the program is parsed by Egalito using the EgalitoInterface. This is required to analyse, transform and recompile the program. In the **generating** phase, the permutations of one function are generated using the Permute Pass. In the **transform and write** phase the function of the program is permuted and directly recompiled n times, where n is the amount of valid permutations. To have a baseline for the experiments we also recompile the original function. To ensure valid permutations, the transformations have to be performed on the original function. Because Egalito keeps track of an intricate datastructure on this function and due to its complex code structure, we have to reset Egalito for every transformation.

Etreorder requires a few command-line options to work:
`[options] input-file function output-file`.
Here `[options]` can be either `-v` or `-q` (or neither), which are verbose mode and quiet mode, respectively. These options either print or suppress the logging messages. The `input-file` is the filename of the program. The `function` is the (symbol)name of the function to be permuted. The `output-file` is the filenamebase for the generated executables. The structure for these executables are: `filenamebase + i`, with i the i-th permutation. Note that a special case is `filenamebase-o`, for the original program.

5.2 Validness and Completeness

It is important to determine when the generation of permutations is valid and complete. A few questions to consider are: Do we have all permutations? Are all permutations valid? Important to note is that the first question is not trivially answered and is dependent on the thoroughness of the used tool. To answer this question, we must first decide to what extend we consider instructions independent of one another. Note, that to guarantee correctness, we rather have the dependencies too strict than too lenient. A few examples:

- One difficult dependency to keep track of is *Memory Dependencies*. Which variable writes to which place in memory? This may seem easy for integer variables, but gets complicated when considering pointers.

- We noticed that the tool Egalito did not allow interchanging `ADD` instructions. This can have several reasons, for example the implicit dependency on the register `eflags`; interchanging them may give a different result for this register. However, for this particular situation, neither results of these overflows would have been used, indicating that the dependency could be *false*.

- How do you approach two operations writing to different parts of the same register? For example, a write to `%eax` could overwrite a previous write to `%rax`, but `%ah` and `%al` do not share this problem.

In conclusion, knowing the exact dependencies is extensive work and probably requires hand-tuning on top of automated work. Therefore, we consider this out of the scope of this work.
To address this problem, we apply the following restrictions to permutations, which must hold in order to be considered valid and complete:

1. Uniqueness: All generated permutations should be unique

2. Validness: With the assumption that the tool Egalito can provide not too strict nor too lenient dependencies, the permutations should in no way violate dependencies or any other criteria that would result in incorrect behaviour.

3. Completeness: With the assumption that the tool Egalito can provide not too strict nor too lenient dependencies, all valid permutations have been created.

5.3 Permute Pass

The Permute Pass itself consists of two phases: initialization, in which the permutations are generated, and permute, in which the actual transformation takes place.

For the initialization phase we ensure the restrictions of the previous subsection by the following measures:

1. We ensure Uniqueness by simply checking this restriction at the end of the initialization.

2. We ensure Validness by assuming the correctness of the dependencies generated by the tool Egalito. As the pass ReachingDef provides us with a list of available instructions at every step of this process, simply choosing only instruction from this list ensures our validness.

3. Completeness is ensured by going over all possible combinations of the provided available instructions. Suppose all possible sets of available instructions were structured in a tree (graph). Then our approach is a depth-first search on this tree, providing all combinations of instruction choices.

We decided to implement the depth-first search in the ReachingDef Pass, as the possibilities were limited by the code structure of Egalito. Figure 9 illustrates the depth-first search for a simple example.

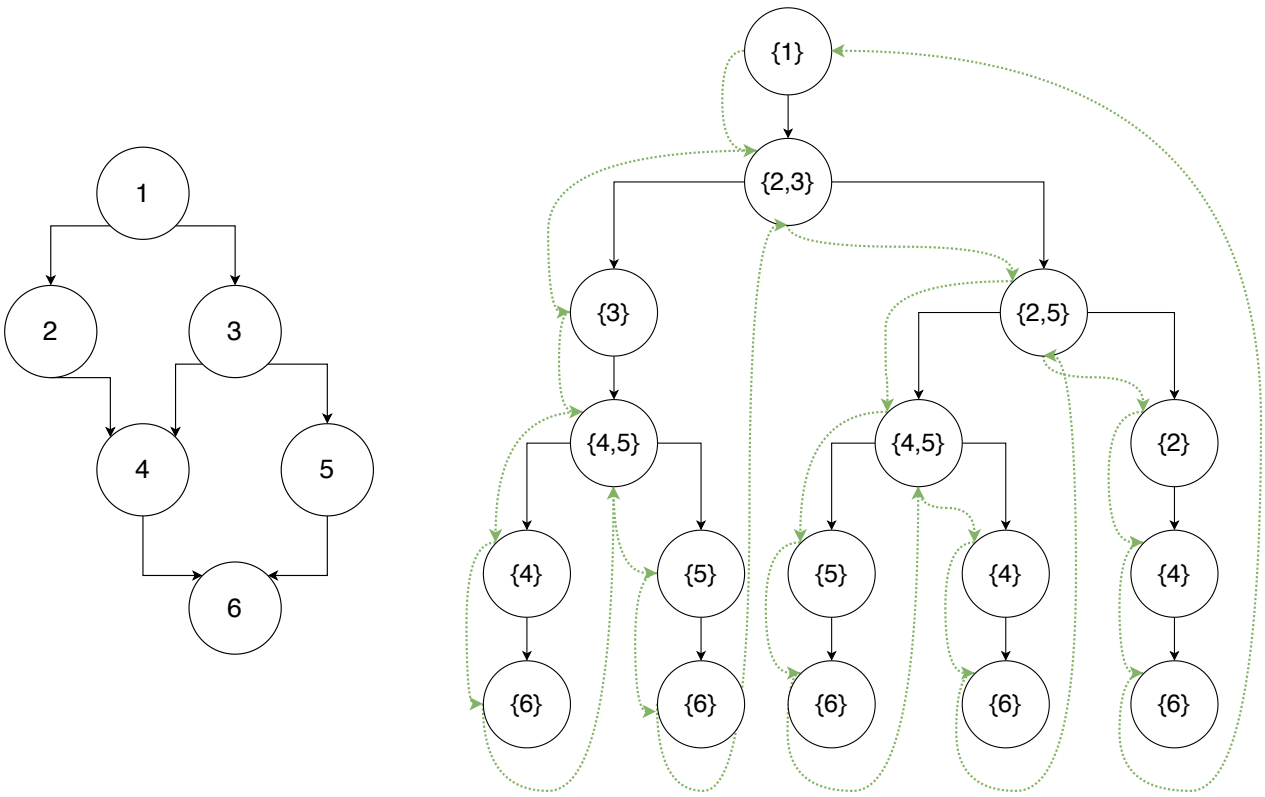


Figure 9: On the left we have a toy example of the Directed Acyclic Graph (DAG) of six instructions. The arrows indicate the dependencies. On the right, we have the corresponding tree of available instructions. A path through this tree is a permutation, the nodes are the available instructions. The dashed (green) arrows represent the depth-first search. We did not merge equivalent subtrees to illustrate the whole search.

In the permute phase the function is transformed by the generated permutation. This phase can be called n times, with n the amount of permutations. The Permute Pass knows the permutation it should use by keeping track of a state. This state is an integer and can be passed to the function $f_i(x)$ to find the permutation number. After every call of this function, the integer is increased. Before we declare this function further, we first go into detail about the structure of a function and its permutations. A function is divided into blocks, which are in fact the Basic Blocks of the function. The provided list of available instructions are in scope of these blocks. Because of this structure, the permutations are not saved for the whole function, but for the Basic Blocks of the function. A permutation of a function is thus a combination of permutations of its Basic Blocks. Now we can define the function $f_i(x)$ as follows:

$$f_i(x) = \frac{x}{y_i} \ \% \ n_i, \text{ with} \qquad y_i = n \cdot \left(\prod_{j=0}^{i} n_j \right)^{-1}$$

with n the total amount of permutations, i the Basic Block number and n_i the number of permutations for Basic Block i.

In order to preserve the uniqueness and completeness restrictions, the union of these Basic Block functions $f(x) = \bigcup_{i=0}^{k} f_i(x)$, with k the amount of Basic Blocks, should be a bijection from $\{0, 1, \ldots, n-1\}$ to the set of (sets of) Basic Block permutations. Note that $n = \prod_{j=0}^{k} n_j$ and the result of the function will be as follows:

$$f(0) = \{0, 0, \ldots, 0\}$$
$$f(1) = \{0, 0, \ldots, 1\}$$
$$\vdots$$
$$f(n_k - 1) = \{0, 0, \ldots, n_k - 1\}$$
$$f(n_k) = \{0, 0, \ldots, 1, 0\}$$
$$f(n_k + 1) = \{0, 0, \ldots, 1, 1\}$$
$$\vdots$$
$$f(n-1) = \{n_0 - 1, n_1 - 1, \ldots, n_{k-1} - 1, n_k - 1\}$$

It is now clear that $f(x)$ maps $\{0, 1, \ldots, n-1\}$ to exactly all generated permutations. For a complete (small) example see Table 1.

$f(x)$	$f_1(x)$	$f_2(x)$	$f_3(x)$
$f(0)$	0	0	0
$f(1)$	0	0	1
$f(2)$	0	0	2
$f(3)$	1	0	0
$f(4)$	1	0	1
$f(5)$	1	0	2

Table 1: Small example of the function $f(x)$ for 3 basic blocks, where the first block has 2 permutations, the second block 1 permutation and the last block 3 permutations. Total amount of permutations is 6.

The actual transformation of the function is inspired by the implemententation of the pass ReachingDef; we iterate through all the Basic Blocks and generate a new order of instructions according to the retrieved permutation. For each instruction in the original order, we determine if it has changed according to the new order. If so, dependencies and instruction semantics will be changed accordingly through the datastructure of Egalito.

6 Experimental Setup

To investigate the influence of program instruction order on runtime, we have set up an empirical evaluation experimental plan. In this section we elaborate on the setup of these experiments, starting with the program we used, followed by the machine and its settings. The last part of this section is dedicated to the configuration of the parameters of this experiment.

6.1 Test Program

The program we constructed, called `Simpleunroll4`, is based on the fact that loop unrolling provides more permutable instructions. This allows us to generate more permutations and increases the chance to reveal significant differences. Within this program, we make use of an array. We chose to work with an array size of 4096, as this is large enough to allow longer runtimes, and thus more accurate measures, but still is small enough to fit in the L1 data cache. The last property is important to avoid introducing performance variation due to memory hierarchy effects. The program `Simpleunroll4`, contains the function `compute`, for which the permutations are generated. Outside of this function, we monitor performance and assure that the compiler does not optimize the function call away. We tested the correctness of the sum of this function, before we started our experiments. Figure 10 shows the function `compute` for an unroll-factor of 4 and Figure 11 shows its corresponding assembly. Important to note is that the original assembly of the loop body of the function `compute` contains a few *name dependencies*; the array elements are saved only in the registers `%rsi` and `%rax`, resulting in a fixed static scheduling order. To solve this, we manually renamed the registers. This allows for the amount of reordering possibilities one might expect from an unrolled loop.
Indeed, we find the number of permutations increase for a higher unroll amount. Table 2 shows the generated permutations for the different unroll-factors. Experiments have been done on the program with loop-unroll factor 4.

Unroll-factor	**Permutations**
0 (no unroll)	3
2	30
4	138600

Table 2: The amount of permutations generated by Etreorder for multiple loop-unroll factors.

```
long compute(void)
{
    long sum = 0;
    for (int i = 0; i < 4096; i += 4)
    {
        sum += array[i] * 3;
        sum += array[i+1] * 3;
        sum += array[i+2] * 3;
        sum += array[i+3] * 3;
    }
    return sum;
}
```

Figure 10: The function `compute` from the c-program `Simpleunroll4`. It computes the following sum over the array `array[]` : $\sum_{i=0}^{4095} \texttt{array[}i\texttt{]} \cdot 3$.

```
    xorl    %ecx, %ecx              # rcx = i = 0
    leaq    array(%rip), %rdx       # rdx = array[0]
    xorl    %eax, %eax              # rax = sum = 0
    .p2align 4, 0x90                # NOPs for alignment
.LBB1_1:
    movslq  (%rdx,%rcx,4), %rsi     # rsi = array[i] (array[0] + 4*i)
    leaq    (%rsi,%rsi,2), %rsi     # rsi = array[i] * 3
    addq    %rax, %rsi              # rsi = sum + array[i] * 3
    movslq  4(%rdx,%rcx,4), %r8     # r8 = array[i+1] (4+(array[0] + 4*i))
    leaq    (%8,%r8,2), %8          # r8 = array[i+1] * 3
    addq    %rsi, %r8               # r8 = sum + array[i+1] * 3
    movslq  8(%rdx,%rcx,4), %r9
    leaq    (%r9,%r9,2), %r9
    addq    %r8, %r9                # ... etc. ...
    movslq  12(%rdx,%rcx,4), %r10
    leaq    (%r10,%r10,2), %r10
    addq    %r9, %r10               # r10 = sum + array[i+3] * 3
    movq    %r10, %rax              # rax = sum + array[i+3] * 3
    addq    $4, %rcx                # rcx = i += 4
    cmpq    $4096, %rcx             # i < 4096
    jb  .LBB1_1
    retq                            # return sum (rax)
```

Figure 11: Assembly corresponding to the c-program `Simpleunroll4`. The lines have been annotated, to give a better understanding of the generated assembly.

6.2 Test System

The experiments were performed on a stand-alone system configured to minimize variation across measurements. We used an *Intel Core i7*-3770 system, which implements the *Sandy Bridge* x86_64 microarchitecture, with $3.4GHz$ clock frequency, 4 cores (4 threads), $64k$ L1 cache, $256k$ L2 cache and $8192k$ L3 cache. To create a stable and controlled environment we configured this system as follows: Fixed CPU clock frequency, disabled Turbo Boost, disabled Idle modes, no HyperThreading. Through experimental validation, we chose to use at most 2 of the 4 available cores, to run multiple generated programs in parallel. Using only one core limits the speed in which the experiments can be done. Using more cores would cause the operating system to interrupt experiments more frequently, which results in less accurate measurements.

6.3 Parameters Configuration

To investigate the influence of permutations on runtime, we measured the *runtime* and a few other metrics. Some direct influences on runtime are the *number of instructions* and the *number of cycles*. In addition we measured the amount of *stalls* as these could be responsible for longer runtimes. These metrics are measured with PAPI [18], a tool to monitor hardware performance counters on Linux systems.

We tested the reliability of our measurements on the smaller unroll-version (factor 2). In this phase we decided on the following configurations:

1. **leave the minimum and maximum out**
   We do not control the scheduler of the system we are running experiments on. This may result in noise, as some executions may be interrupted, resulting in longer runtimes. Since this type of noise cannot be prevented, we try to exclude the outliers by removing both the longest and shortest runtime of each executable.

2. **Inner Repeats** = 10000
   The runtime of one execution of the function `compute` (Figure 10) is too short to measure accurately. Therefore we repeat this computation until a point where measurements can be taken accurately. Table 3 shows the results for a few Inner Repeat values. An Inner Repeats value of 1 is definitely too small. A value of 1000 is a lot better already, but we still considered this not accurate enough. Therefore, we chose an *Inner Repeats* value of 10000. Of course, higher values would gain more accurate results, but this also drastically increases the total runtime for the experiments. Our chosen value, represents a balance between accurate results and manageable runtimes.

3. **Outer Repeats** = 12
   Since one data point for each executable is not nearly enough to counter the natural variance in runtime, we have to repeat the run of each executable. Through experimental validation we tried to find the number of repeats for which the permutations have stable means and standard deviations, meaning that repeating the experiments would give approximately the same results. We found 10 to be enough, but to counter the first measure, we incremented this by 2, resulting in the *Outer Repeats* value of 12.

Inner Repeats	Mean	Max	Min	Dev
1	0.0000504	0.000051	0.000050	0.000001
1000	0.0432597	0.043471	0.043234	0.000237
10000	0.4321927	0.432227	0.432063	0.000164
50000	2.1609273	2.160983	2.160849	0.000134

Table 3: Differences in runtime (in seconds) for different Inner Repeats values. Dev is the difference between the highest (Max) and lowest (Min) runtime value.

4. **Order**
   The order of the executables may affect the distribution of increase in deviation. Table 4 shows an example of this concept. The effect of this ordering on deviation is as follows: Assume the system gets a new job to execute during the experiments. The first type of order has the highest chance this will affect the same executable twice. The third type of order has a lower chance this will affect the same executable twice, but it still might happen. The second type of order has the lowest chance this will affect the same executable twice. Let us assume, for example, that the scheduler decides to plan the new job in timeslots E and F, marked in Table 4 (as red). Only the second type of order gives a more uniform distribution of the increase in deviation. We aim for a uniform distribution by ordering the executables by Outer Repeats.

Timeslot	A	B	C	D	E	F	G	H	I
Order by Executable	E1	E1	E1	E2	E2	E2	E3	E3	E3
Order by Outer Repeats	E1	E2	E3	E1	E2	E3	E1	E2	E3
Random Order	E1	E3	E2	E1	E2	E2	E3	E1	E3

Table 4: Toy example of the influence of execution order on deviations. Each i-th executable is represented as Ei. The Order by Executable first repeats $E1$ x times, then $E2$ x times, etcetera. The Order by Outer Repeats runs all executables once and repeats this process x times. The Random Order runs each executable x times. The Timeslots can be used as a reference to the order and give no guarantee in the duration of the slot.

7 Results

In this section, we are going to analyse the results from the experiments and draw conclusions. We start the section by analysing the results from the smaller experiment on the program `Simpleunroll2`, with loop-unroll factor 2.
This is followed by the results from the larger experiment on the program `Simpleunroll4`, with loop-unroll factor 4. We start this second part with comparing the original executable with the generated permutations. Next, we analyse a few metrics known to be direct influencers of runtime, with the results from the larger experiment. The last part of this section is dedicated to an analysis in the type of permutations.

7.1 Analysis on Permutation Runtimes

The main question we try to answer is if program instruction order has any measurable influence on performance in runtime when executed on dynamically scheduled superscalar processors. To answer this question, we examine the runtimes of the original executable and all generated permutations.

We start with `Simpleunroll2`, which yielded the mean and standard-deviation for 30 permutations, the results of which are shown in Figure 12. Even with this low amount of permutations, we can already draw an interesting conclusion; the majority of the permutations are significantly slower than the original executable. None seem to be significantly faster.

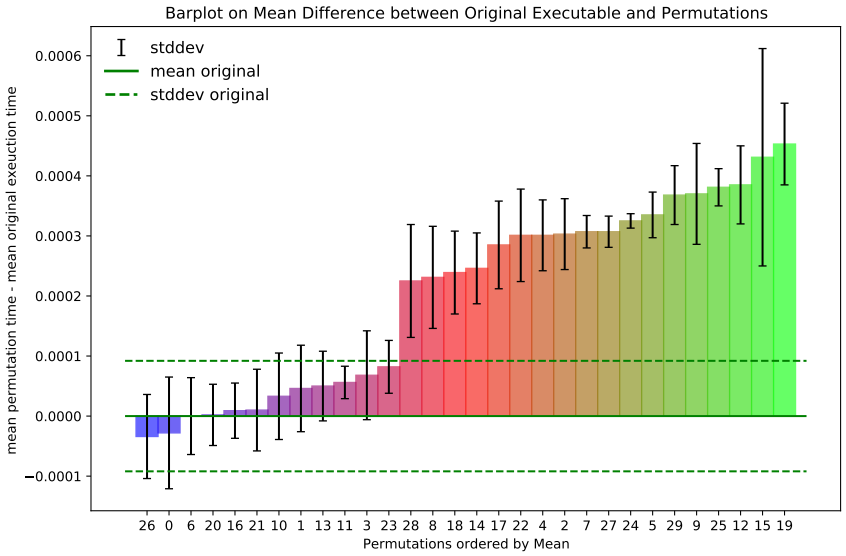


Figure 12: Barplot on Mean Difference between Original Executable and its Permutations, for small experiments. Permutations are ordered by their mean.

Now, we continue to `Simpleunroll4`. The experiments yielded the mean and standard-deviation for 138600 permutations. This is hard to visualise and analyse, while results will often be very similar. Therefore, we decided to group the permutations into different clusters. Since we cannot draw any conclusions for permutations with a standard-deviation of 0.0004 or higher, we decided to exclude these from the clustering. This allows us to draw more precise conclusions about the rest of the permutations. For permutations with a standard-deviation lower than 0.0004 we used the clustermethod `kMeans`, with $k = 15$. As the initial cluster centers of this method are random, we decided to repeat it 100 times, as this ensures more stable clusters.
The results are shown in Figure 13, a clusterplot on the difference between the original executable and its permutations. Plotted standard-deviations are the maximum standard-deviations of the clusters, as this guarantees reliable results. The permutations with high standard-deviations are plotted in a scatterplot to the right of the clusterplot. The first thing to notice is that, as opposed to the smaller experiment, there are clusters which are actually significantly faster than the original executable. This time, none seem to be significantly slower. This is an interesting result, not only did we find that the instruction order matters, but this influence is also different for a small alteration in the program. There are a number of clusters, however, which still have a too high standard-deviation. For these clusters, no conclusions can be drawn.

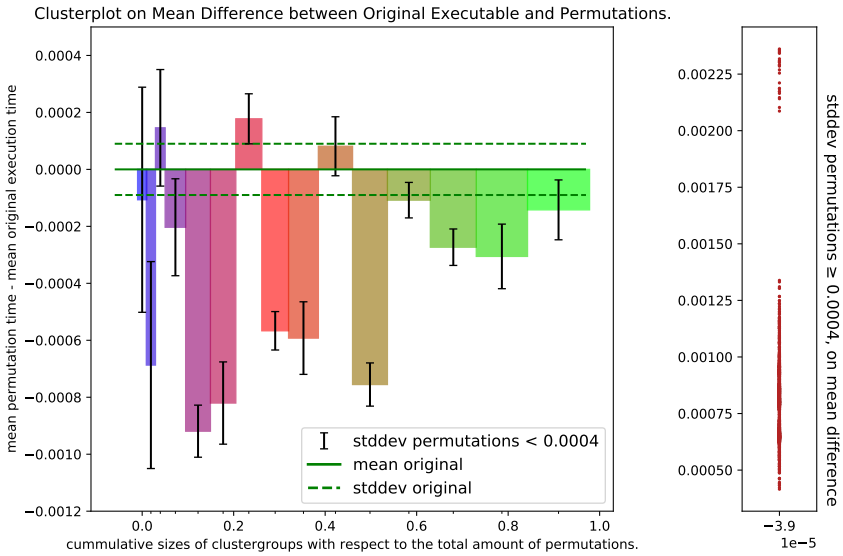


Figure 13: Clusterplot on Mean Difference between Original Executable and its Permutations. Bars are ordered and scaled by their size, with a minimum of 0.02. The y-axis has been limited to the ranges $[-0.0012, 0.0005]$. Plotted standard-deviations are the maximum standard-deviations of clusters. The highest standard-deviations are not plotted in the clusterplot, but in the scatterplot on the right.

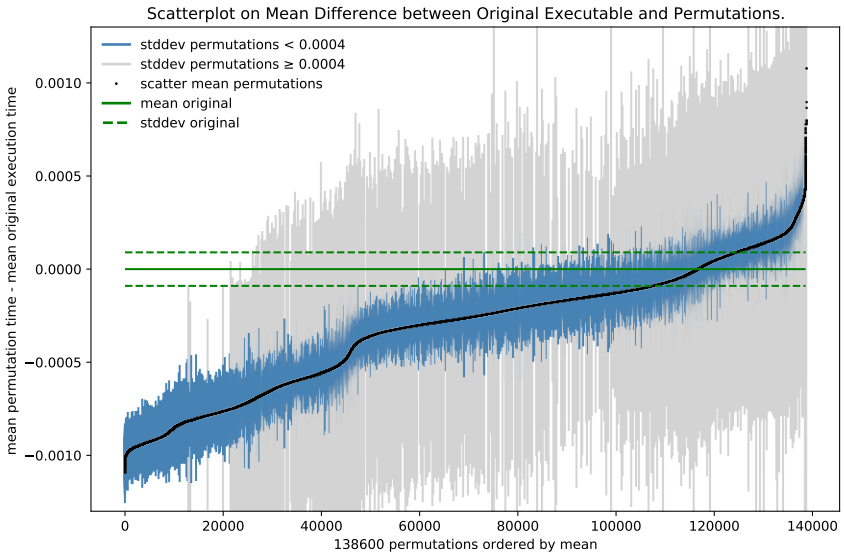


Figure 14: Scatterplot on Mean Difference between Original Executable and its Permuations. Permutations are ordered by their mean. The y-axis has been limited to the ranges $[-0.0013, 0.0013]$.

The use of the maximum standard-deviation in the clusterplot, was necessary to have reliable results. However, when considering all permutations, this may be too strict. Therefore, we created a detailed overview of all results, as can be seen in Figure 14. Here the mean and standard-deviation of all permutations are depicted. The number of standard-deviations larger than 0.0004 is small compared to the total number of permutations, but is still large when plotted. We do not want to leave these permutations out, but neither do we want them to overshadow the important results. Therefore, permutations with a standard-deviation ≥ 0.0004 are plotted as background information, so we can focus on the reliable data. From this Figure, we can also conclude that there is a large number of permutations that are faster than the original program, even when the standard deviations are considered.
The differences in runtime between the original and all permutations is an important result, as it confirms the significance of instruction order. However, we would like to find a cause for these differences. The rest of this section is dedicated to an investigation into this question.

7.2 Analysing Metrics

As a first step in finding an explanation for the found differences in the previous subsection, we look into metrics that are known for their direct influence on runtime. We investigate three metrics; the number of retired instructions, the number of cycles per execution, and the total number of stalls. The first two are very direct influences of runtime. The total number of stalls could variate due to the permutations.

First, on the number of retired instructions. This event occurs when an instruction is completely executed, that is, when its state is written to registers as would have been done for an in-order execution, as stated by the Intel VTune Profiler User Guide [19]. This is different from the number of issued or executed instructions, as some instructions, and their results, are flushed in case of a

misprediction.
Logically, the number of instructions should remain constant per permutation. However, this is not true in reality. In our experiment we found the difference in retired instructions to be 444, with a mean of 2686290453. The size of this number is of course negligible, but the effect is undesirable. Weaver et al. [20], found two primary causes, *nondeterminism* and *overcount*, for the deviations in this event, and others. These causes find their sources in events like Hardware Interrupts and Page Faults.

Figure 15 shows the metric cycles versus runtime. It is not surprising that there appears to be a linear relation between the number of cycles and the runtime. The permutations themselves have been grouped by their standard-deviations by plotting them in different colors. This allows us to investigate if there is any correlation between the number of cycles and the standard-deviation of a permutation. Unfortunately, this cannot be concluded from the Figure, as the groups are too close together and the standard-deviations too large. It does mean, however, that the standard-deviations are influenced by something else.

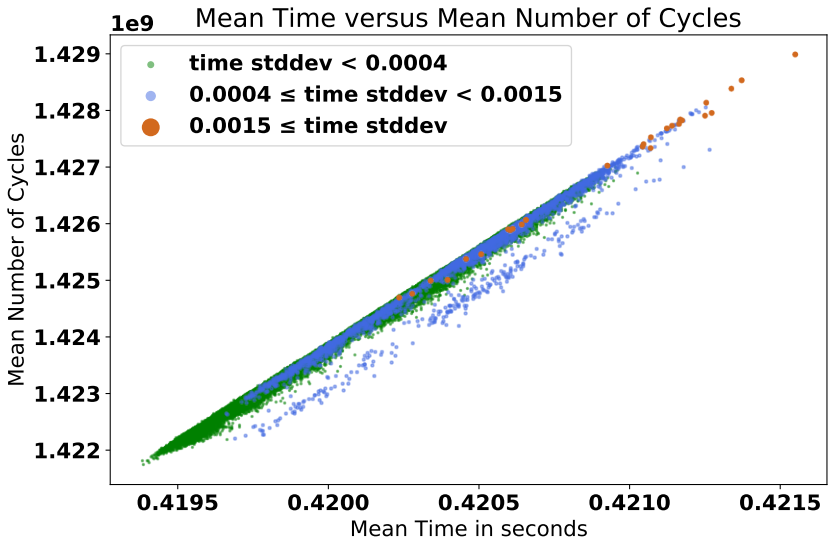


Figure 15: Mean Runtime versus Mean amount of Cycles for each permutation, grouped by the runtime standard-deviation.

Figure 16 shows the metric total stalls versus runtime. The permutations in this Figure are grouped by the clusters from Figure 13. Figure 16 does seem to show a sort of group forming. However, the groups are 'vertical', and thus influenced by time and not stalls. Since the clusters are based on time (and standard-deviation), this does not yield new results. Also, a clear correlation between stalls and runtime cannot be found. This result is not entirely surprising, as the idea behind dynamic schedulers is to reduce stalls caused by dependencies.
So unfortunately, neither metrics seemed to give any explanation for the difference in runtime and standard-deviations. However, these results are important, as we now know for certain that we do not need to investigate these metrics further. Furthermore, they confirm some properties that could be expected from a processor with a dynamic scheduler.

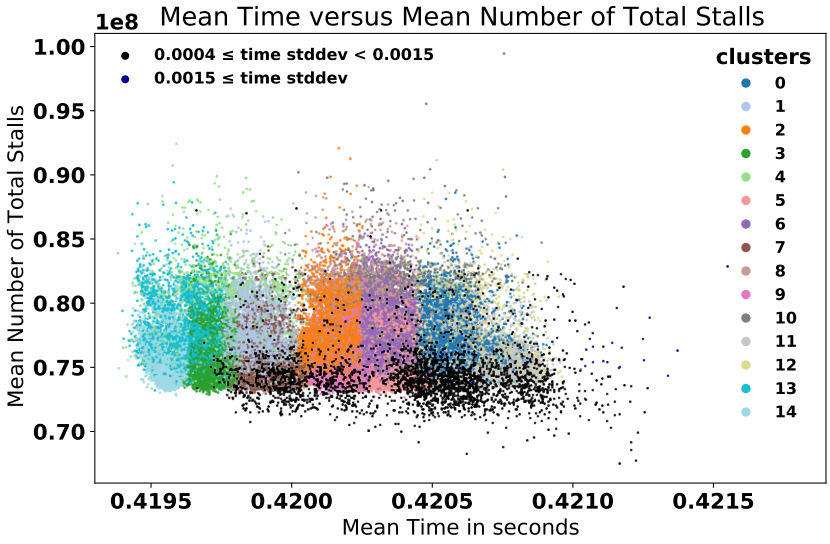


Figure 16: Mean Time versus Mean amount of Total Stalls for each permutation, grouped by the clusters from Figure 13. The group with high standard-deviation has been split further into two groups.

7.3 Characterizing Permutations

We attempt to find characteristics of permutations that influence the runtime. We start with the permutations from the small experiment and end with trying to identify a relation between these characteristics and the generated clusters from Figure 13. In order to do so, we identified the ordering of instructions for each permutation with respect to the original executable.

There are various types of instructions. Often, identical type of instructions are scheduled distantly from each other to avoid stalls caused by dependencies. As dynamic schedulers do not have to conform to this order, these dependencies may cause less of an issue, since the instructions do not have to be executed strictly consecutively. The question remains if this issue is solved completely, or if there is still a negative influence. Therefore, we investigated whether a high frequency of these instructions in vicinity of each other, influences the mean runtimes.

We start with the permutations from the program `Simpleunroll2`. Since only 30 permutations were generated from this program, we were able to manually distinguish the different orderings of the permutations and draw conclusions accordingly. We split the permutations into two groups according the results shown in Figure 12. The first group, called `slow`, are the instructions that ran significantly slower than the original executable. The second group, called `equal`, are the instructions that were neither significantly faster nor slower. Analysing the orderings of these groups gave some useful insights. We focused on the instructions from the loop body, as most time is spent in executing those. Within this set of instructions, we only considered those whose positions were not fixed. This gave the final subset of investigated instructions:

```
0:   movslq   0(%rdx, %rcx, 4), %rsi
1:   leaq     0(%rsi, %rsi, 2), %rsi
2:   addq     %rsi, %rax
3:   movslq   4(%rdx, %rcx, 4), %r9
4:   leaq     0(%r9, %r9, 2), %r9
```

The results are illustrated in Figure 17. Note, that the first `movslq` instruction can be either number 0 or 3, and is thus not fixed.
There are a few things to conclude from this Figure. First of all, the two groups have their own distinctive orderings, they share none. This alone, gives a very clear indication that the permutations can be grouped according to their orderings. Looking closer into these distinctive orderings, we find that the ordering of the `slow` group always end with `addq`. As the first instruction after this sequence is also an `addq`, this results in two `addq`'s consecutively at the end. It is difficult to determine an exact reason as to why this seems to have such a great influence, but we can consider the fact that this probably introduces a delay at the end of a critical path, since the two `addq` instructions have dependencies on each other.
Therefore we can conclude from this experiment that the distribution of the type of instructions, indeed causes variations in runtime.

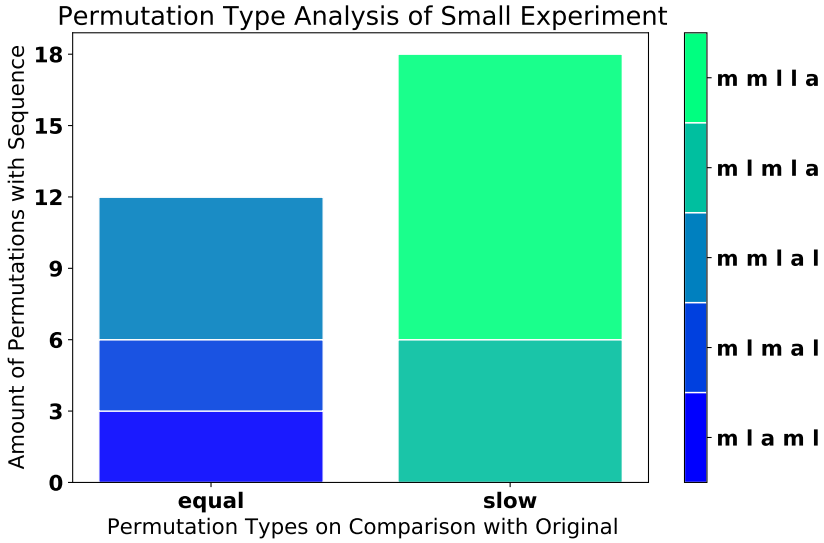


Figure 17: Permutation Type Analysis of the Small Experiment. The sequences are depicted at the colorbar on the right, with **m** the `movslq`, **l** the `leaq` and **a** the `addq` instructions.

Of course, we cannot draw final conclusions, without looking at the big experiment too. However, 138600 permutations from the program `Simpleunroll4` are far too many to consider separately. For this reason, we looked for a relation between the type of permutations and the generated clusters from Figure 13.
After identifying the ordering, we grouped the permutations by equivalency. This yielded 463 non-equivalent groups. The equivalency is measured on the type of instructions. For example, instruction `0: movslq 0(%rdx, %rcx, 4), %rsi` is categorized as `movslq`, or simply `m`. Again, the amount of groups hinder extensive analysis into these orderings, as they are out of

the scope of this work. Therefore, we considered sequences with 3 consecutive instructions of the same type, grouped by `movslq`, `leaq` and `addq`. On top of that, a sequence of 3 consecutive instructions of all different types, were also investigated. The results are shown in Figure 18. Important to note is that the clusters do not have the same sizes.
Unfortunately, the clear distinction we found in the small experiment, is not presented in this Figure. However, we can point to a few interesting occurrences that future work may fully exploit. When comparing the `mla` sequence with the others, we see an opposed trend in its frequency. This is not completely surprising, as these are opposite in distribution in the type of instructions. Also easy to notice, is the difference in frequencies between the types: `mla` and `mmm` are much more frequent than `lll` and `aaa`. This too, is not completely surprising, when considering hindrance by dependencies when permuting instructions. Especially the `aaa` sequences differ greatly in frequencies. This is interesting, as the `addq` instruction was decisive in the small experiment. Since the Figure does not show a single clear trend, and the clustersizes are not shown in this Figure, we cannot draw any decisive conclusions. However, this appearance is interesting and further research may uncover an explanation.
In general, no clear conclusions can be drawn. However, by this Figure and the results of the small experiment, characterizing permutations can be exploited more thoroughly in the search for finding causes of the differences in runtimes between the original executable and the permutations.

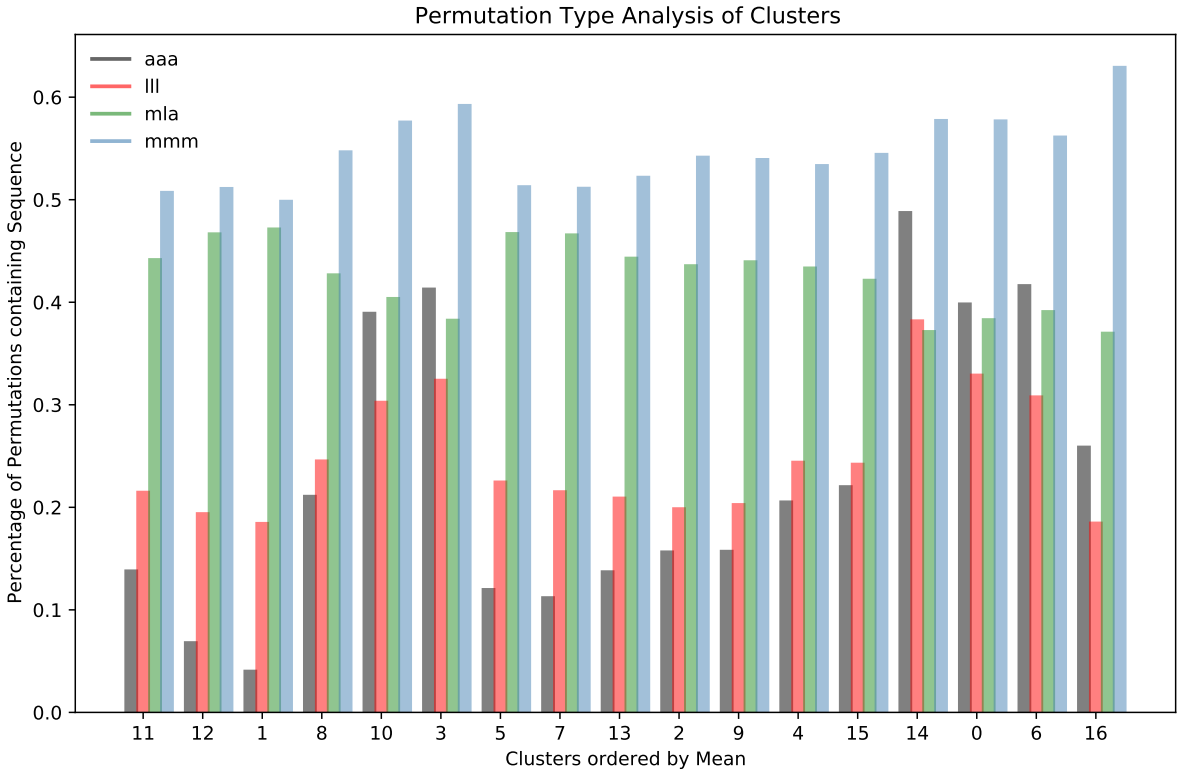


Figure 18: Permutation type analysis based on the clusters from Figure 13. We added two clusters to the existing ones in Figure 13; cluster 15 is the permutations with $0.0004 \leq$ standard deviation ≤ 0.0015 and cluster 16 is the permutations with $0.0015 \leq$ standard deviation.

8 Conclusion

We investigated if program instruction order has any measurable influence on performance in runtime when executed on dynamically scheduled superscalar processors. In order to do so, we developed a tool to generate all valid permutations of a program. This required extensive dependency-analysis, for which we used the Egalito framework. By extending this framework with the `Permute Pass`, we were able to keep track of all dependencies. We created the application `Etreorder` to generate all valid permutations of a function within a simple program.

In order to conduct an initial exploration of the influence of program instruction order on program runtime, we performed a series of experiments using a simple testprogram and a specifically configured test system. From the results followed that instruction order indeed has measurable influence on performance in runtime when executed on a dynamically scheduled superscalar processor. We even found that a small alteration in the type of program, creates a different type of influence on this performance in runtime.
To explain the observed differences in runtime, we investigated the relation of a number of metrics and the runtime as well as the influence of different sequences of `movslq` and `addq` instructions. No clear correlations were found and further research is required to uncover the cause of these differences. However, we did find an indication to where to start in this search; an exploration into the influence of instruction type distribution on differences in runtime between an executable and its permutations.
To this aim, we propose the following interesting extensions to our research:

1. *Differences Analysis:* An in-depth analysis into the causes for the differences in runtime between the original program and all generated permutations. With methods such as profiling, characteristics can be found for the different types of permutations.

2. *Other Microarchitectures:* Our test system uses an aggressive dynamic scheduler. Will the differences be greater for microarchitectures with a more moderate dynamic scheduler?

3. *Other Kinds of Programs:* Do other types of programs yield the same results as we found? Can we find any patterns in these types? Do any of these patterns lead to a different performance?

References

[1] Donald E Knuth. "Structured programming with go to statements". In: *ACM Computing Surveys (CSUR)* 6.4 (1974), pp. 261–301.

[2] John L Hennessy and Thomas R Gross. "Code generation and reorganization in the presence of pipeline constraints". In: *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 1982, pp. 120–127.

[3] Daniel Tate, Gordon Steven, and Fleur Steven. "Static scheduling for out-of-order instruction issue processors". In: *Proceedings 5th Australasian Computer Architecture Conference. ACAC 2000 (Cat. No. PR00512)*. IEEE. 2000, pp. 90–96.

[4] Anne M Holler. "Optimization for a superscalar out-of-order machine". In: *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture. MICRO 29*. IEEE. 1996, pp. 336–348.

[5] Paolo Faraboschi, Joseph A Fisher, and Cliff Young. "Instruction scheduling for instruction level parallel processors". In: *Proceedings of the IEEE* 89.11 (2001), pp. 1638–1659.

[6] David Williams-King, Hidenori Kobayashi, Kent Williams-King, Graham Patterson, Frank Spano, Yu Jian Wu, Junfeng Yang, and Vasileios P Kemerlis. "Egalito: Layout-Agnostic Binary Recompilation". In: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 2020, pp. 133–147.

[7] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. *Compilers: Principles, Techniques and Tools (also known as The Red Dragon Book)*. 1986.

[8] Tyrel Russell. "Learning Instruction Scheduling Heuristics from Optimal Data". MA thesis. University of Waterloo, 2006.

[9] Ghassan Omar Shobaki and Kent Wilken. *Optimal global instruction scheduling using enumeration*. University of California, Davis, 2006.

[10] Hong-Chich Chou and Chung-Ping Chung. "An optimal instruction scheduler for superscalar processor". In: *IEEE Transactions on Parallel and Distributed Systems* 6.3 (1995), pp. 303–313.

[11] Daniel S McFarlin, Charles Tucker, and Craig Zilles. "Discerning the dominant out-of-order performance advantage: Is it speculation or dynamism?" In: *ACM SIGARCH Computer Architecture News* 41.1 (2013), pp. 241–252.

[12] Prashant Singh Rawat, Aravind Sukumaran-Rajam, Atanas Rountev, Fabrice Rastello, Louis-Noël Pouchet, and P Sadayappan. "Associative instruction reordering to alleviate register pressure". In: *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE. 2018, pp. 590–602.

[13] Madhavi Gopal Valluri and R Govindarajan. "Evaluating register allocation and instruction scheduling techniques in out-of-order issue processors". In: *1999 International Conference on Parallel Architectures and Compilation Techniques (Cat. No. PR00425)*. IEEE. 1999, pp. 78–83.

[14] Chris Lattner and Vikram Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation". In: San Jose, CA, USA, Mar. 2004, pp. 75–88.

[15] Charles Stark Draper Laboratory Inc. *Fracture*. https://github.com/draperlaboratory/fracture. 2014–2015. (Visited on 05/24/2020).

[16] Trang Tran Thi Thuy and Jonas Berg. *LLVM Legalization and Lowering*. https://wiki.aalto.fi/display/t1065450/LLVM+Legalization+and+Lowering. (Visited on 06/12/2020).

[17] Eli Bendersky. *A deeper look into the LLVM code generator part 1*. https://eli.thegreenplace.net/2013/02/25/a-deeper-look-into-the-llvm-code-generator-part-1. (Visited on 06/12/2020).

[18] Dan Terpstra, Heike Jagode, Haihang You, and Jack Dongarra. "Collecting performance data with PAPI-C". In: *Tools for High Performance Computing 2009*. Springer, 2010, pp. 157–173.

[19] *Intel® VTune™ Profiler User Guide - Instructions Retired Event*. https://software.intel.com/content/www/us/en/develop/documentation/vtune-help/top/analyze-performance/custom-analysis/custom-analysis-options/hardware-event-list/instructions-retired-event.html. (Visited on 07/17/2020).

[20] Vincent M Weaver, Dan Terpstra, and Shirley Moore. "Non-determinism and overcount on modern hardware performance counter implementations". In: *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE. 2013, pp. 215–224.