University of Exeter
Department of Computer Science

Performance Counter Measurements of Data Structures: Implementations for Multi-Objective Optimisation

Silvia I. Candia

February 5th 2020

Supervised by Prof. J. Fieldsend & Dr. D. Acreman

Submitted by Silvia I. Candia, to the University of Exeter as a thesis for the degree of Master by Research in Computer Science, February 5th 2020.

I certify that all material in this thesis which is not my own work has been identified and that no material has previously been submitted and approved for the award of a degree by this or any other University.

(signature) .....

Abstract

Solving multi-objective optimisation problems using evolutionary computation methods involve the implementation of algorithms and data structures for the storage of temporary solutions. Computational efficiency of these systems becomes important as problems increase in complexity and the number of solutions maintained becomes large.

Many data structures and algorithms have been proposed looking to decrease computational times. The effectiveness of a data structure/algorithm can be characterised using wall-clock time. This is a widely used parameter in the literature, however it is strongly dependent on the underlying computer architecture and hence not a reliable measure of absolute performance. A commonly used approach to avoid architectural dependencies is to compare the performance of the data structure being evaluated to the equivalent implementation using a linked list.

Modern processors offer built-in hardware performance counters, giving access to a wide set of parameters that can be used to explore performance. In this dissertation we study the efficiency of a non-dominated quad-tree data structure in combination with different evolutionary algorithms using hardware performance counters. We also compare the results for the quad-tree data structure to a linked list as it is the standard practice, however we find non-scalable hardware dependencies might appear.

to Lucas and Emilia

Acknowledgements

I would like to start by thanking my supervisors Prof. J. Fieldsend, Dr. M. Schreiber during the first half of this project and Dr. D. Acreman during the second, for their guidance and support along this project. I am particularly thankful to Prof. J. Fieldsend, my main supervisor, who without detriment to academic excellence does not leave the human factor out of research, bringing the best out of the people who work with him as was my case. It has been a pleasure to work with you.

Thanks also for the interesting discussions and friendship to the rest of the research group: Abdulaziz, Amjad, Khulood, Kevin, Carina, Tinkle and George.

The support I have received from my family, extended family and friends has been amazing. It was not easy to go back to university after so many years of inactivity and with a family system put in place that had to change to accommodate my new career. First I would like to thank Lucas and Emilia, my children, who were so understanding, loving, supportive and excited to see me retake a career. Matthew: you have been incredible, a great friend and a source of endless encouragement and patience on the many occasions things got really tricky. I will be forever grateful. Janet and Pip, for rescuing me many times when kids needed to be looked after and always being ready with a cup of tea to have a chat. And Elyse: you are not physically with us any more but you will remain with me always. You saw me through more than half of this, and your unconditional love, support and strong belief in me have made all the difference.

Contents

Nomenclature and Abbreviations

Acronyms and abbreviations

μOP	Micro-Operation *(p. 8)*
ALU	Arithmetic Logic Unit *(p. 6)*
BOB	Branch Order Buffer *(p. 10)*
BPU	Branch Predictor Unit *(p. 8)*
CPI	Cycles per Instruction *(p. 12)*
CPU	Central Processing Unit *(p. 6)*
CPUID	serialising instruction (returns CPU details) *(p. 32)*
IBR	Instruction Buffer Register *(p. 6)*
IC	Instruction Cycle *(p. 5)*
IDQ	Allocation Queue Unit *(p. 8)*
IHS	Improved Harmony Search *(p. 68)*
IR	Instruction Register *(p. 6)*
ISA	Instruction Set Architecture *(p. 6)*
L1D	Level 1 Data *(p. 11)*
L1I	Level 1 Instructions *(p. 11)*
L1IC	Level 1 Instruction Cache *(p. 8)*
LSD	Loop Stream Detector *(p. 10)*
M	Main Memory *(p. 6)*
MAR	Memory Address Register *(p. 6)*
MBR	Memory Buffer Register *(p. 6)*
EA	Evolutionary Algorithm *(p. 49)*
MITE	Micro-Instruction Translation Engine *(p. 8)*
MOEA	Multi-Objective Optimization Evolutionary Algorithm *(p. 49)*
MOP	Multi-Objective Optimisation Problem *(p. 48)*
MOEA/D	Multiobjective Evolutionary Algorithm Based on Decomposition *(p. 59)*
MSR	Model Specific Register *(p. 23)*
NSGA-II	Non Dominated Sorting Genetic Algorithm II *(p. 68)*
OS	Operating System *(p. 4)*
PAPI	Performance Application Programming Interface: a program to access hardware counters *(p. 29)*
PMC	Performance Monitoring Counter *(p. 23)*
PMU	Performance Monitoring Unit *(p. 23)*

RAT	Register Alias Table *(p. 10)*
RDMSR	instruction to read the MSR *(p. 24)*
RDPMC	instruction to read the PMC *(p. 24)*
RDTSC	instruction to read the TSC *(p. 31)*
ROB	Reorder Buffer *(p. 10)*
WRMSR	instruction to write the MSR *(p. 24)*
TLB	Translation Lookaside Buffer *(p. 11)*
TSC	Time Stamp Counter *(p. 31)*

Symbols

A	Pareto archive *(p. 50)*
CPI	average number of cycles per instruction *(p. 12)*
$\boldsymbol{F}$	objective function *(p. 48)*
$\boldsymbol{F}(S)$	feasible objective region *(p. 48)*
f_i	each of the k conflictive objective functions to be minimised (or maximised) simoultaneously *(p. 48)*
I_c	total number of instructions for a program *(p. 12)*
m	number of conflictive objectives of an MOP *(p. 53)*
$MIPS$	millions of instructions per second *(p. 12)*
P	solution population *(p. 50)*
S	region of the decision space determined by the constraints of the problem *(p. 48)*
T	total time a processor takes to execute a program *(p. 12)*
t	generation produced by evolving a population P *(p. 50)*
$\boldsymbol{u}$,$\boldsymbol{v}$	objective vectors *(p. 48)*
$\boldsymbol{x}$	decision vector *(p. 48)*

1 Overview

Multi-objective optimisation problems have been the subject of extensive research over the last few years. The need to find a solution to problems involving many variables while optimising two or more conflictive objectives is found across a broad range of study areas such as social sciences, engineering, industry and science [Coello Coello, 2006]. The wide reach of possible applications include topics such as crop planning, qualitative and quantitative control of urban flooding and water pollution, trajectory planning in robotics and resource allocation problems in management, just to mention a few [Deb, 1999; Oraei Zare et al., 2012; Márquez et al., 2011; Coello Coello, 2006].

Despite the ongoing advance towards more efficient computing machinery, the size and complexity of these problems still require a careful choice of the data structures and algorithms used in their calculation and storage of non-dominated sets of solutions [Sun and Steuer, 1996; Altwaijry and El Bachir Menai, 2012]. Many data structures and algorithms have been proposed over the last few decades in the quest for better performance [Sun and Steuer, 1996; Zitzler et al., 2000; Mostaghim et al., 2002; Zhang and Li, 2007; Drozdik et al., 2014; Dementiev, 2016].

The standard approach for evaluation consists in measuring the wall-clock time of the proposed data structure/algorithm and comparing it to an equivalent implementation using a list (see, for example, Mostaghim and Teich [2005]). This method is reasonable when establishing the areas of efficiency for the particular machine used to run the experiment, however a generalisation of these results to other computer systems requires a more careful analysis. It has been found, for example, that the computational time taken for a given data structure depends on the amount of data stored and the problem size: given the particular configuration a list might be preferred to a more complex data structure and vice-versa [Mostaghim and Teich, 2005; Drozdik et al., 2014]. The particular architecture used to perform these experiments might have an important effect on the identification of these performance areas, however little attention has been paid to this aspect. This dissertation is a preliminary attempt to address this issue.

The main objective of this research is to establish if the measured performance of a multi-objective data structure, using a particular computer system can be affected by the underlying architecture of the system itself. Knowing this is of fundamental importance when looking to generalise the results obtained with one computer system to any possible architecture, and assign merits of better performance to the algorithm under test rather than the system in which the program is being run. Even relative performance between programs might be affected: while in a system program A might run more efficiently than

program B, unless the effect of the architecture in each of these programs is well known, it is not obvious that program A will be faster than program B under a different architecture. Notice that time complexity analysis of an algorithm abstracts the program from the system being run on, while we would like to be able to identify in terms of architectural components how the experimentally measured performance changes for different systems.

In order to address these questions, we intend to identify and single out the main architectural components contributing to overall performance. Once identified, we can measure their individual contributions through performance counters, registers in the processing unit built to monitor the functioning of the micro-architectural units processing the instructions and data of a program.

Our procedure will also put in evidence that wall-clock time as an experimental measure of program efficiency is insufficient, and instead a set of performance parameters is a more accurate representation of a program's performance.

We outline here the main subjects covered in this dissertation. We start by introducing the micro-architectural components and technologies that affect the performance of a particular computer system and we discuss several models that have been developed to analyse computing performance. We then introduce some commonly used performance analysis methods and tools, and we implement code using architectural performance counters to micro-benchmark a small piece of code. We then compare the different approaches we used, and discuss the results obtained with each of them. In Chapter 5 we introduce some data structures and algorithms used to solve multi-objective optimisation problems, and we concentrate on the prototypical quad-tree data structure and algorithm proposed by Mostaghim and Teich [2005]. We implement these with a program written in C++, and using the performance counters discussed earlier we measure the performance for different combinations of parameters. We end by summarising our findings and future lines of work.

2 Micro-architecture, Performance Analysis and Profiling

The analysis of algorithms is very relevant to computer programming. In most cases, there might be several methods to resolve a problem, involving different steps in the calculation of the solution. The running time of a program dealing with a given set of data depends on the algorithms chosen to perform the different operations applied to the dataset. The best choice will be the one that takes the smallest time to perform the same task.

It is possible to calculate the total number of steps M of an algorithm A_M analytically as

$$M = \sum_i M_i n_i, \tag{2.1}$$

where M_i is the number of steps a particular step i requires, n_i the number of times the step i is executed and the subscript i represents an index running over all the steps [Knuth, 1997].

An exact solution for the time it takes to execute an algorithm A_M in a given processor is complex. While Equation (2.1) can be isolated from the type of processor used, its computation time T_c is strongly correlated to the choice of processor. As a first approach we could consider a simple serial processor which gives

$$T_c = \sum_i t_i M_i n_i, \tag{2.2}$$

where t_i is the time it takes to run the step M_i. Equation (2.2) assumes the steps M_i execute one at a time.

As we will see later, there are many important considerations about modern processors and programming languages not reflected in the equations above: M_i and n_i in Equation (2.2) are processor dependent as a result of technologies such as micro and macro-fusion of operations, instruction level parallelism, loop stream detection and multiple stage pipelines.

Similarly, in the case of high-level languages additional considerations are required. The translation of an algorithm A_M into machine instructions depends not only on the programming language of choice but also on the choice of compiler with its specific optimisation options. In addition, the program is run in an environment managed by an Operating System (OS).

Experimentally, the measurement of performance is done through profiling software. Due to the considerations above, the time a program takes to run is not necessarily a suitable metric for comparison. By developing performance models that take into account the underlying architecture, we can define metrics that allow a better characterisation of performance.

Benchmarking and Program Similarity

The measurement of metrics through profiling software serves two main purposes. On one hand, they are useful to identify bottlenecks within a program and hint to potential improvements in performance through modification of the source code. On the other hand, they provide experimental measurements of quantities that can be use to characterise a program and compare it to others. Studies on program similarity and benchmarking concentrate on this second aspect.

It is not possible to infer directly from the vendors' specifications how a piece of software will perform a particular machine. Not only the same processor might perform differently than the expected from the specifications, but also the choice of compiler, OS and how the application itself was written might affect user experience. Benchmarking of processor performance aims to produce a meaningful comparison among commercial computers. Benchmarks are set of programs created with the aim to compare different systems to decide which one performs better [Stallings, 2000].

An important concept in benchmarking is program similarity [Joshi et al., 2006]. A good benchmark suite will be composed of programs which each stress different bottlenecks, i.e., the values for the metrics that characterise them are very different [Vandierendonck and De Bosschere, 2004].

While initially these tools and methods were developed to determine the true performance of different commercial processors, they can be also used to classify programs and algorithms in function of their similarity in a more general fashion. In particular they can be used to explore the Evolutionary Algorithms (EAs, see Section 5.2) and data structures used as benchmarks in the computation of Multi-objective Optimisation Problems (MOPs, see Section 5.1) [Tanabe et al., 2017].

2.1 Architecture of a Processor

The fetch-decode-execute cycle or instruction cycle (IC) is the principle of operation of a computer. Instructions are fetched from memory, decoded into operations and then executed by the processor [Stallings, 2000; Tanenbaum and Austin, 2013].

Figure 2.1 shows a diagram for the structure of the IAS computer, the first machine following a von Neumann architectural model [Stallings, 2000]. A cycle starts by fetching a word

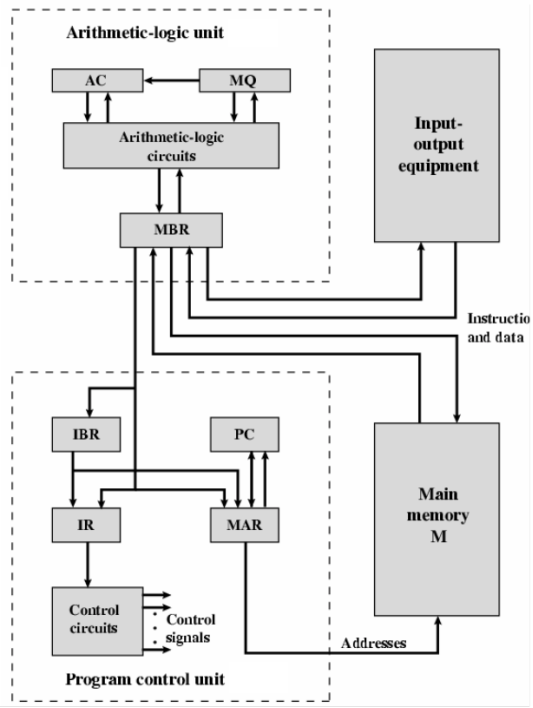

Figure 2.1 Structure of the IAS computer, the first machine following a von Neumann architectural model (from Stallings [2000])

for the next instruction to be executed, loading the opcode part into the Instruction Register (IR) and the address part into the Memory Address Register (MAR). This instruction can be fetched either from the Instruction Buffer Register (IBR) or from the Memory Buffer Register (MBR). The execute cycle begins by the processing the opcode through the control circuits. The resulting control signals are then sent to produce an event in the circuit, for example, make the Arithmetic Logic Unit (ALU) perform an operation. Instructions can be grouped by type according to the kind of event they produce. The set of all the instructions used to operate the circuits in a machine is called the Instruction Set Architecture (ISA).

Modern computers present a variety of highly sophisticated architectures for the Central Processing Unit (CPU). Advances in microchip technology have allowed bigger memories and faster processing speeds while at the same time developing techniques for improving efficiency of use. One of these important advances is the introduction of a memory buffer known as the cache. Originally the speeds of the Program Control and the Arithmetic-logic Units with respect to the filling of the Main Memory (M) were such that the processor was busy working at every cycle, with data and instructions to be processed always waiting in memory. However with advances in technology processor speeds became faster, eventually surpassing the memory speed to gather more data and instructions. This resulted in inefficiencies, as the processor had to stall few cycles waiting for the instructions or data to be loaded into memory. While faster main memories are possible they would be too costly to replace in their totality. The introduction of cache systems offers the benefits of a fast memory at an affordable cost. The cache takes advantage of the locality of reference of a computer program (execution memory references to data and instructions tend to cluster). The cache works as a high-speed memory buffer between the processing unit and

the main memory, loading the most likely addresses to be requested. Multiple-level caches are common elements of efficient memory subsystems [Stallings, 2000].

Superscalar Out-of-order Processors

In modern processors, the fetch-decode-execute instruction cycle can be divided into many more parts which are individually handled by dedicated pieces of hardware processing in parallel. The units of execution are called stages. Multiple-stage pipelines improve the performance of a computer by increasing the processing speed of instructions. As soon as the first part of the first instruction is processed at the first stage, the unit is free and ready to received the first part of the second instruction. In the following cycle, the second stage of the first instruction and the first stage of the second are processed simultaneously [Tanenbaum and Austin, 2013].

A superscalar processor implements one multiple-stage pipeline which has more than one functional unit at the execution stage. In general the term superscalar refers to a processor that issues multiple instructions in a clock cycle.

These advances are responsible for instruction-level parallelism. Processor level parallelism is achieve by the simultaneous use of multiple CPUs. A multiprocessor is a system with more than one CPU sharing a common memory. Each of these CPUs are commonly referred to as 'cores'.

Dependence among instructions create stalls in the pipeline, as it is put to a halt until the output on which the next operation depends on is produced. Out-of-Order execution improves performance by altering the order in which instructions are delivered to the pipeline. Instructions that are dependent on each other are grouped together and instructions that would appear after them if the execution was in-order are executed before.

Other technological advances that have contributed to a more efficient use of resources include branch prediction, speculative execution, extension of the instruction set, hyper-threading technology, macro-fusion and micro-fusion of operations [Patt, 2001]. As a result, the instruction cycle in a modern processor is highly complex.

2.2 The Instruction Cycle in a Modern Superscalar Out-of-Order Processor - Intel Kaby Lake

Among the most popular processor families is the Intel x86 architecture. The processor we used for our measurements belongs to the 7th generation Intel® Core™ family. This supports Intel 64 architecture and are based on the Intel® microarchitecture Kaby Lake, derived from the microarchitecture **Nehalem** using 45nm process technology [Intel, 2018d]. The name of the microarchitecture (commonly known as 'code name') is important to identify the technologies used and the structure of the pipeline. When looking through manuals and specifications with the objective to understand the instruction cycle, we always refer to Kaby Lake, Skylake (Kaby Lake inherits the microarchitecture from Skylake with a number of enhancements) or Nehalem.

In the Kaby Lake microprocessor the pipeline can be broken down into three major areas: the front-end, back-end (or execution engine), and the memory subsystem. Figure 2.2 shows a diagram for the instruction cycle in the Skylake microprocessor (which is the same in Kaby Lake) [Intel, 2016; WikiChip].

2.2.1 The Front-End

The front-end is where instructions coming from memory go through the decoding process [Intel, 2018d]. Instructions are fetched from the Level 1 Instruction Cache (L1IC) into the instruction fetch and pre-decode unit at a rate of up to 16 bytes/cycle (every other cycle if using two threads as the unit is shared between the threads). The pre-decoded instructions are then sent to the Instruction Queue Unit at a rate of up to 6 macro-operations (macro-ops) per cycle (instructions are not yet decoded and they are still x86 architectural instructions). An optimisation introduced at this stage is Macro-fusion, where two instructions can be combined into one. Effectively, a new (more complex) instruction with the same combined function replaces the previous two. Only one macro-fusion per cycle is possible. These instructions are then fed into the decoder, at a rate of 5 (pre-decoded)instructions/cycle and are sent through different decoders depending on their complexity (Microcode Sequencer, Complex Decoder or Simple Decoder). With outputs up to 5 micro-operations (μOPs)/cycle the now decoded instructions go into the Allocation Queue Unit (IDQ).

The above path through the front end going from Instruction Cache to Allocation Queue IDQ is referred to as the traditional (or 'Legacy') path, and constitutes the Micro-Instruction Translation Engine (MITE). An improvement is the introduction of a micro-operations cache holding already decoded instructions which are fed directly to the IDQ in when guessed correctly by the Branch Predictor Unit (BPU), resulting in a bypass of the MITE (the micro-operations cache stores actual decoded instructions but it is still a subset of the L1 instruction cache). The rate of transfer in the event of a hit (correct guess) is 4 μOPs/cycle.

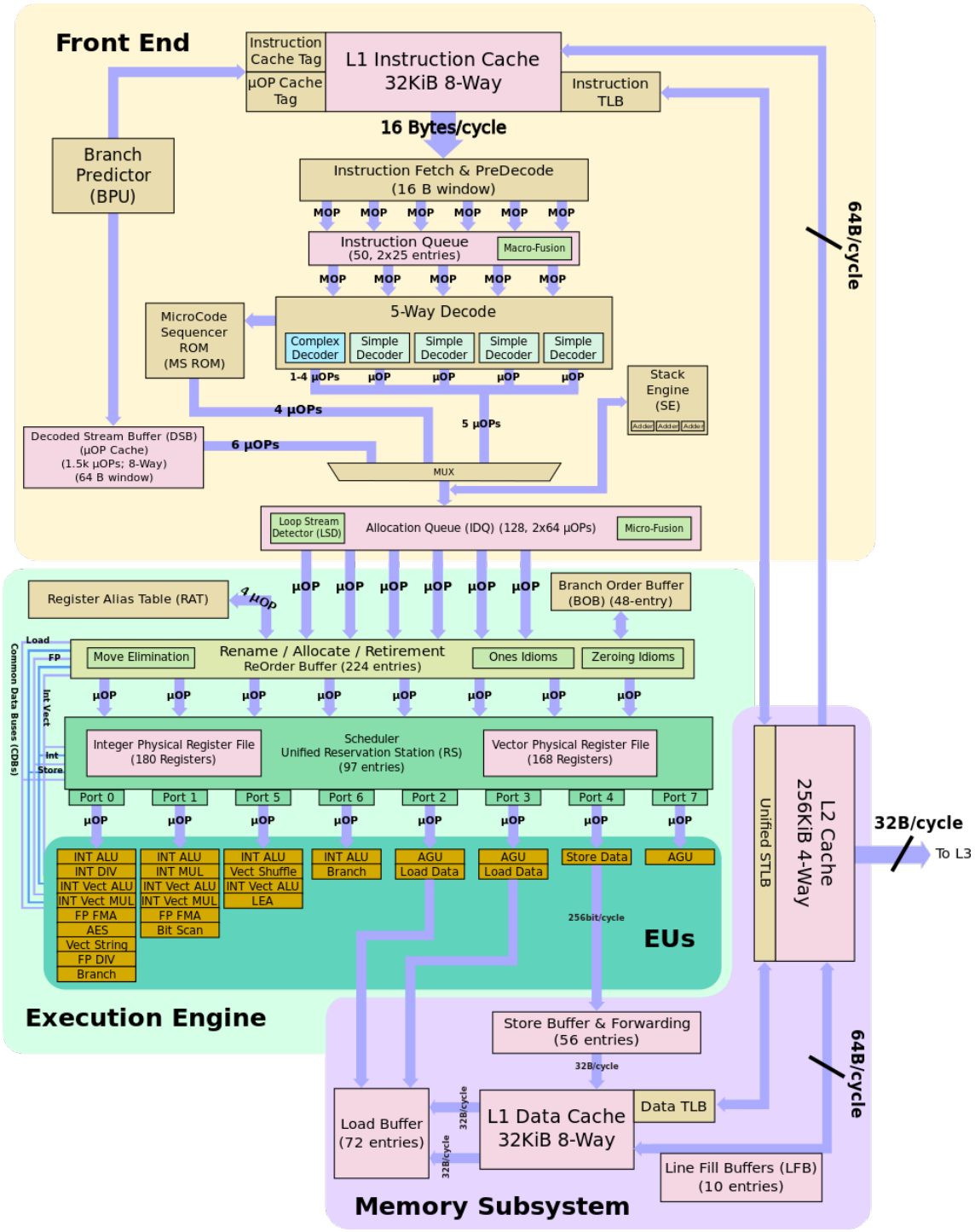

Figure 2.2 Block diagram for the Front-End, Back-End and Memory Subsystem for Skylake. This is the same micro-architecture as Kaby-lake (diagram from WikiChip)

A separate Stack Engine dedicated to stack operations sits after the decoders. Incoming stack-modifying operations (PUSH, POP, for example) are caught and processed by the Stack Engine, which has its own ALU (a more efficient alternative to processing them via the back-end).

The IDQ is the last stage of the pipeline in the front-end section. If possible, micro-fusion is applied to the incoming micro-operations. Micro-fusion fuses multiple micro-ops from the same instruction into a single complex micro-op, improving bandwidth. It also increases the bit density of a micro-fused operation (empty bits sections are eliminated and non-empty bits shifted), increasing the storage capacity of the IDQ.

Also part of the IDQ, a Loop Stream Detector (LSD) detects small loops that fit in the micro-op queue and locks them down. The loop streams directly from the micro-op queue with no more fetching, decoding, or reading micro-ops from any of the caches, until a branch misprediction signals its end.

The IDQ links the front end with the out-of order execution engine, delivering four micro-ops for execution each cycle, acting as the interface between the front-end and the back-end.

2.2.2 The Back-End (Execution Engine)

Across the different front-end pipeline stages instructions are kept in-order. During the first back-end pipeline stage this order is lost [Intel, 2018d].

From the allocation queue instructions are sent to the Reorder Buffer (ROB) at the rate of up to 6 fused-μOPs each cycle. Registers are mapped onto physical registers (register renaming), controlled via a Register Alias Table (RAT).

A Branch Order Buffer (BOB) keeps track of the architectural states after execution: in the case of an incorrect speculative execution the BOB helps rolling back to the last known state.

Some additional optimisations occur at this stage (move elimination, ones idioms and zeroing idioms) resulting in a reduction of μOPs in the ROB.

From the reorder buffer, μOPs are sent to the scheduler, storing them while they wait to be executed. There is also an integer and a vector register file which store the output operand data.

From the scheduler, μOPs are dispatched to the execution unit through one of seven issue ports. The scheduler queues the μOPs to the appropiate port. The ports are connected to execution clusters, a collection of execution units performing several operations. μOPs dealing with memory access (e.g. load & store) are sent on the dedicated scheduler ports 2, 3 and 4. Store operations go to the store buffer which is also capable of performing forwarding when needed. Likewise, load operations come from the load buffer. The scheduler can dispatch up to six μOPs every cycle, one on each port.

After a μOP passes through the execution unit, it is either written back to the register file or forwarded through a bypass network to a μOP in-flight that needs the result. Once a μOP leaves the execution engine, it reaches the last stage of the pipeline and it is retired (in-order), releasing any used resources (such as those used to keep track in the reorder buffer, for example).

2.2.3 The Memory Subsystem

The memory subsystem is a 3-level cache (L1, L2 and L3) [Intel, 2018d].

The L1 cache is divided into two sections: one dedicated to caching (pre-decoded) instructions (L1I) and another caching data (L1D). The L2 cache is a unified data and instruction cache. Each processor core has its own L1 and L2. The L3 cache is an inclusive, unified data and instruction cache, shared by all processor cores inside a physical package. The cache lines are 64 bytes wide.

Two level Translation Lookaside Buffers (TLBs) handle address translation and improve memory access by reducing the accesses required. The TLBs store the most recently used page-directory and page-table entries. The first level TLBs consists of two separate buffers that handle instructions (connected to the L1I cache) and data (connected to the L1D cache). The second level TLB is unified for instructions and data and handles page translation operations missed by the first level TLBs.

As mentioned in Section 2.2.2, three additional buffers (one for store, two for loads) are associated with instruction execution units. The store buffer allows writes to system memory and/or the internal caches to be saved and in some cases combined to optimise performance.

Notice that knowledge of the behaviour of these caches together with their dimensions can be used in optimizing software performance as they would determine, for example, how large a data structure can be operated on at once without causing cache thrashing [Intel, 2016]. In the case of Kaby Lake, the L1D cache and L1I cache have a capacity of 32 KiB. The capacity of the core-private L2 cache is 256 KiB. Each of the cores share the L3 cache, with a slice of 2MiB per core.

2.3 Processor Performance Modelling and Analysis

Performance can be measured in terms of the instruction execution rate, i.e., how many instructions can be executed in a single cycle, or its reciprocal Cycles per Instruction (CPI) [Stallings, 2000].

Cycles Per Instruction (CPI)

We define the (average) number of cycles per instruction CPI of a program as

$$CPI = \sum_{i}^{n} \frac{CPI_i \times I_i}{I_c}, \tag{2.3}$$

where CPI_i is the average cycle per instruction i, I_i the number of executed instructions of type i for a given program and I_c the total number of instructions for the given program. Notice that if all the basic instructions I_i required the same number of clock cycles, then $CPI = CPI_i$ would be a constant value for a processor.

If $\tau = 1/f$ is the cycle time of the processor clock running at frequency f (a constant), then the time T the processor takes to execute the given program is

$$T = I_c \times CPI \times \tau. \tag{2.4}$$

A common measure of performance when comparing different processors is the rate at which instructions are executed, expressed in millions of instructions per second $MIPS$. In terms of CPI:

$$MIPS = \frac{I_c}{T \times 10^6} = \frac{f}{CPI \times 10^6}. \tag{2.5}$$

From Equations (2.4) and (2.5) we see the factors affecting performance. Given a fixed number of instructions I_c and a set $f = 1/\tau$, we can see that it is the factor CPI which inherits the complexity of the instruction cycle of the particular machine.

Various models have been developed for the performance of a processor, based on the instruction cycle and pipeline stages described in Section 2.1 (see [Van Den Steen et al., 2016] for a brief account of different kind of approaches and models to date). Some of these include analytical models (where mathematical forms are used), empirical models (which build from a training set and using machine learning techniques), mechanistic models (which model the flow of instructions through the pipeline), Hybrid empirical/mechanistic models.

Among these, empirical models give the most accurate results [Van Den Steen et al., 2016] but they lose insight in the processor mechanisms, making it difficult to understand

the reasons for the performance results. On the other hand, mechanistic models are not that accurate but they provide understanding of the underlying mechanisms affecting performance. We will chose this approach, in particular the Interval Processor Model developed by Eyerman et al. [2009].

In general a performance model approaches the problem identifying two main components: the performance element under ideal conditions (i.e., instructions and data being processed through the pipeline without any stalls); and the corrections due to inefficiencies in the use of the optimisation technologies in the processor (performance penalties due to miss events) [Ailamaki et al., 1999; Karkhanis and Smith, 2004; Eyerman et al., 2006a; Yasin, 2014]. If we define $CPI_{steadystate}$ as the total cycles per instruction for the first of these contributions (total cycles in absence of any stalls), and $CPI_{inefficiencies}$ for the additional cycles per instruction that appear when missed events occur, we can express a the total CPI of a given performance model as as the sum of both these terms:

$$CPI = CPI_{steadystate} + CPI_{inefficiencies}. \tag{2.6}$$

2.3.1 A Simple Processor Model

To begin we will consider a simple model [Stallings, 2000]. We assume the only delays occur when instructions are transferred between memory and execution unit. This is the equivalent of a two stage pipeline (fetch-execute) but where no superposition is possible (i.e., for the processor to fetch instructions from memory it must have finished execution of the previous one). If we define the memory cycle time as τ_m, m the number of memory references needed, and $k = \tau_m/\tau$ then

$$CPI = p + (m \times k)\,, \tag{2.7}$$

where p is the number of cycles needed to process and decode the instruction. Comparing Equation (2.7) with Equation (2.6) we can attribute the correspondences $CPI_{steadystate} = p$ and $CPI_{inefficiencies} = m \times k$.

Superscalar out-of-order processor models provide expressions to account for the lost cycles in the pipeline introduced in Section 2.2 [Ailamaki et al., 1999; Karkhanis and Smith, 2004; Eyerman et al., 2009; Chen et al., 2012; Van Den Steen et al., 2016]. The many modifications to the basic pipeline introduced by optimisation technologies result in a lower $CPI_{steadystate}$ and add complexity to $CPI_{inefficiencies}$ which receives contributions from the stages where the pipeline can stall.

One of the first models to consider out-of-order execution and branch prediction with speculative execution was developed by Ailamaki et al. [1999], studying the case of commercial database management systems run in a Pentium II architecture. They propose a decomposition of the time spent in a (query) instruction as

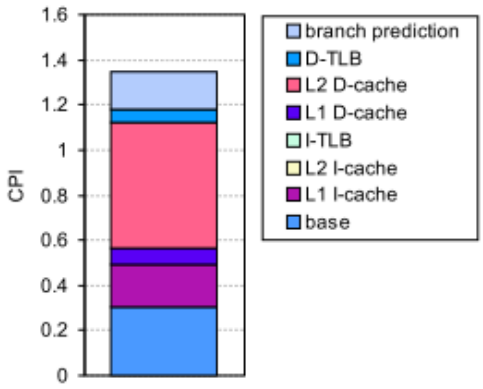

Figure 2.3 Example of a CPI stack (From Eyerman et al. [2006a])

$$T_Q = T_C + T_M + T_B + T_R - T_{OVL}. \tag{2.8}$$

In Equation (2.8) T_C is the computation time, T_M the time taken by memory stalls, T_B the time incurred by branch misprediction, T_R the time taken by resource related stalls (execution units not being available such as registers, for example). The term T_{OVL} accounts for the 'overlap' of stalls due to processor optimisation techniques that allow parallelism such as out-of-order execution, for example. In this case when an instruction is stalled at a given stage in the pipeline, unrelated instructions coming after the stalled one can be advanced in their execution while waiting for the earlier one to continue down the pipeline (see 2.1). Stalls for these instructions are said to 'overlap'.

Later work in [Karkhanis and Smith, 2004] develops a first order performance model for superscalar processors separating the main contributions to the inefficiency term in Equation (2.6) as

$$CPI_{inefficiencies} = CPI_{brmisp} + CPI_{ichachemiss} + CPI_{dcachemiss}, \tag{2.9}$$

where *brmisp* accounts for branch mispredictions, *ichachemiss* for instruction cache misses and *dcachemiss* for data cache misses.

CPI Stacks

The breakdown of CPI in Equation (2.6) as in Equation (2.8) or Equation (2.9) can be visualised as a 'CPI stack', a stacked histogram weighed by the different components [Yasin, 2014; Eyerman et al., 2006a] (see Figure 2.3).

A CPI stack is a useful tool to visualise the performance of a program in a given processor as whole, while raw miss rates and metrics on their own might not be as clear.

2.3.2 Events and Metrics

Events are defined are occurrences in the hardware. For example, a L1 cache miss is an event that counts each time the data or instruction requested is not in the L1 cache. Definition of events are arbitrary: any occurrences (in any combination) in the pipeline can be defined as events.

The measurement of CPI components in a real system can be done using hardware counters built in the processor. Different performance models express the CPI breakdown in terms of their own defined events. Feedback between researchers and hardware architects has resulted in a variety of event counters integrated on chip with the aim of obtaining accurate performance evaluations of those events. Different architectures provide counters following different approaches [Eyerman et al., 2006a]. Some events are common to most models (as L1 cache misses for example) and are available on most modern microprocessors. Not all the events proposed in a model can be measured with the available counters, however sometimes it is possible to deduce or estimate their value from measurable events.

Modelling of Memory Performance

We present here a simple model to approach memory performance with its related events (as by Stallings [2000]).

We will only consider random access memory. Usual parameters that describe the performance of memory are the access time or latency, the memory cycle time and the transfer rate of data from memory to processor.

We define latency as the time it takes to perform a read or write operation in memory. Before a second access, additional access time might be required due to transient signals and other effects in the system bus which is now active. We define the memory cycle time as the total time resulting from the addition of the access time plus any system bus surplus time (τ_m in Section 2.3.1). Finally, we define the transfer rate as the rate at which data can be moved in or out of a memory unit $f_m = 1/\tau_m$.

Based on Section 2.2.3 we assume a 3-level cache system. The principle of operation behind cache memory attempts to find a balance between the advantages of the main memory (large capacity but low access time) and a buffer memory (low capacity but fast). The cache memory contains copies of portions of main memory. When a program is running in the processor and access to a word is needed, the first level L1 of the cache is checked. If the word is found it is transferred to the processor at a rate t_1. The other cache levels are checked successively when the word is not found in the lower levels. The size of the caches going up the levels increases while the transfer time decreases. If the word is not found at any of the cache levels the main memory is accessed and a memory block containing the requested word is copied into the cache at a rate t_m. If the word is found in the faster cache memory we called it a hit, if not found, a miss.

We define the hit ratio h as

$$\text{hit ratio} = h = \frac{\text{Number of times required word is found in cache}}{\text{Total number of references}} \tag{2.10}$$

and conversely the miss rate as $m = 1 - h$.

We can calculate the average access time t_a in terms of misses as

$$t_a = t_c + mt_m, \tag{2.11}$$

where t_c is the cache access time. For a two-level cache system Equation (2.11) becomes

$$t_a = t_{c1} + m_1 t_{c2} + m_2 t_m, \tag{2.12}$$

where t_{c1} and t_{c2} are the access times for the caches $L1$ and $L2$ respectively, with m_1 the miss ratio for the L1 cache and m_2 the combined miss ratio of L1 and L2. (We can make a similar analysis to arrive to an equation for a 3-level cache but we restrict the equation to 2 levels for simplicity.)

2.3.3 Naive Approach

This is the approach behind the model of Ailamaki et al. [1999] (Equation (2.8)) and it is known as the 'naive' approach and is a traditional method to calculate memory related performance events. In brief, this method assigns penalties to the events responsible for the stalls [Yasin, 2014], so the totality of Stall Cycles is the sum over all the events incurring penalties (i the index denoting the event):

$$StallCycles = \sum_i Penalty_i \times Event_i. \tag{2.13}$$

Following Equation 2.13 disregards many of the effects found in real processors that add complexity to the calculation of the stall cycles due to memory events (overlap of stalls, for example). We will complete here the model and then consider more accurate ones.

The Pentium II processor has a 2-level cache memory system with independent data and instructions cache and translation lookaside buffer, the time in terms of cycles spent on stalls

$$T_M = T_{L1D} + T_{L1I} + T_{L2} + T_{DTLB} + T_{ITLB}, \tag{2.14}$$

where $L1D, L1I$ refer to the data and instruction level 1 caches respectively, $L2$ the level 2 cache and $DTLB, ITLB$ the data and instruction translation lookaside buffers.

To include the additional factors affecting the performance of the instruction cycle they consider the resource stall time T_R. This is the sum of contributions coming from the functional unit (FU) unavailability, the dependencies among instructions (DEP) and other stalls due to platform-specific characteristics (MISC):

$$T_R = T_{FU} + T_{DEP} + T_{MISC}. \tag{2.15}$$

Using the available event hardware counters in Pentium II, Ailamaki et al. [1999] use the following measurement methods for the stall cycles of the different components in Equation (2.8):

- C : estimated minimum based on the counter for the event μops retired
- Stall Cycles in Memory (generating T_M):
  - $L1D$ number of misses * 4 cycles (penalty)
  - $L1I$ actual stalls
  - $L2$ number of misses * measured memory latency (penalty)
  - $DTLB$ not measured
  - $ITLB$ number of misses * 32 cycles (penalty)
- B branch misprediction retired * 17 cycles (penalty)
- R actual stalls
- OVL not measured

For the events where hardware counters were not available, the authors estimated the total stall cycles for that even by assigning them penalty cycles. Unfortunately in their work they do not clarify how they arrive to these particular penalty terms, but they claim to have made use of the same formulae used by Kim Keeton, who published their research some years later [Keeton et al., 2004]. However, while still in this work it is not clear where those penalties come from and they are only mentioned in two table captions, it can be inferred they are extracted from experimental observations. We will limit ourselves here to accept the penalty terms as suggested. We will see later that the penalty approach is not our preferred method.

2.3.4 Interval Analysis

As previously mentioned, the naive approach outlined in Equation (2.13) ignores some important aspects of a real processor. The average penalty for a given event can vary across programs and stalls events can overlap among other effects. Eyerman et al. [2006b] model for branch misses (B), for example, identifies five contributions to the penalty some of which are program dependent: the front-end length of the pipeline, the number of instructions since the last miss event, the level of instruction parallelism of the program, the functional unit latencies and the number of L1D cache misses. Notice also that the term T_{OVL} in Equation (2.8) cannot be measured but it could be an important contribution when working with out-of-order processors. These and other effects make a real evaluation of the CPI more complex than just a sum of individual non-interacting components.

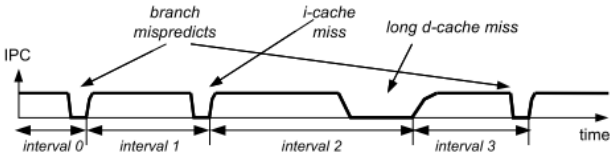

Figure 2.4 Decomposition of the IPC in function of time using Interval Analysis (from Eyerman et al. [2006a])

The Interval Analysis method attempts to address these complexities. The computation of the miss events is done through the division of time intervals between the sources for the different stalls [Karkhanis and Smith, 2004; Eyerman et al., 2006a].

The Interval Analysis model assumes there is a smooth flow of instructions which is often interrupted by miss events. When a miss event occurs the issuing of instructions is disrupted until the event is resolved. When the flow of instructions starts again performance recovers. Figure 2.4 illustrates this basic idea, CPI is shown as a function of time. As different miss events happen, the performance is divided in intervals. Notice that the profile of each interval depends on the type of event that generates it, reflecting their different behaviour. As an example, Figure 2.5 shows a model for an I-cache interval.

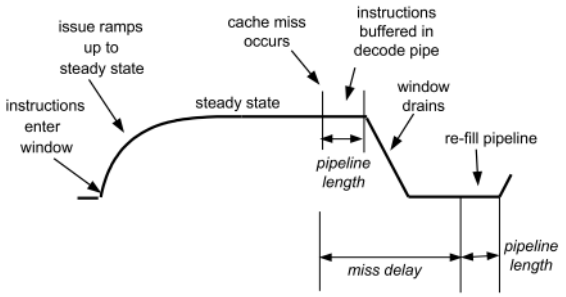

Figure 2.5 Example of Analysis of an Interval representing an Instruction Miss Event (from Eyerman et al. [2006a])

Eyerman et al. [2006a] classifies events according to their underlying characteristics and behaviour into Front End and Back End. The model accounts for the individual events and for the interactions between miss events as well. The penalties for the different events are calculated by analysing their particular performance profile.

Front End Events

Possible events in the front end stage of the pipeline are Instruction Cache misses, TLB mises and Branch Mispredictions.

As an example of how interval analysis works we will analyse Figure 2.5. When a cache miss occurs there is an initial time equal to the frontend pipeline length before the window (containing the instructions going into the back end, the IDQ in Figure 2.2) starts to drain. This is the time it takes for the instructions already in the pipeline to be dispatched. After the missed instruction is gathered from the L2 cache, there is a time required to re-fill the pipeline equal to its length. The overall penalty is then equal to the number of cycles between the time the instruction cache miss occurs and the time newly fetched instructions start filling the frontend pipeline. These cycles are then assigned to either and I-cache miss or an I-TLB miss. In absence of an specific architectural counter for the event, the L2 access latency is a good approximation for these events.

The branch misprediction penalty calculated with this model equals the branch resolution time (time between the branch entering the window and the branch being resolved) plus the time it takes to fill the pipeline again with the new instructions. Characterisation of the branch misprediction penalty in this model is complex, and is calculated in Eyerman et al. [2006b]. One of many time dependencies for branch resolution is the instruction level parallelism of a program, for example. Notice that in the naive model, the branch misprediction penalty is considered to be just the pipeline length, resulting in underestimations of the penalty compared to the Interval Analysis model.

Back End Events

The events belonging to the back end (Execution engine plus memory subsystem in Figure 2.2) are memory related and consists of L1 data cache misses, L2 data cache misses and Data TLB misses. The calculated penalty for an isolated miss as well as for overlapping long data cache misses, equals the time between the Re-Order buffer ROB (Figure 2.2) filling up and the data returning from main memory. Notice that L1 cache misses are short events, and in the case of a well balanced processor, we shouldn't expect this to be a big effect as the nature of out of order execution means the ROB is full of instructions to be executed. L2 cache misses are much longer however, and will have a much bigger impact in the calculation of the penalties, but it is important to account for the overlapping of these misses in the case a miss follows the previous one within certain number of instructions. In the naive model, overlapping of long miss events are not taken into account, resulting in important overestimation of the penalties [Eyerman et al., 2006a].

A later development of the above model to deal with the superposition of events at the different stages of the pipeline is its extension to a multi-stage analysis [Eyerman et al., 2018]. CPI stacks are built along each stage of the processor pipeline (at issue, dispatch and commit) and the performance is analysed looking at all the stages as an ensemble.

We chose to discuss the interval analysis model for the insight it provides in the understanding of events, however this is not necessarily the most suitable method to approach experimental measurements. While through the implementation of simulations any event counter architecture is possible, the available counters in a real micro-processor do not necessarily provide measurements for the penalties for the events as defined by this model. An approach that is more suitable to experimental results is the top-down analysis method.

2.3.5 Top-Down Analysis Method

The top-down method of analysis proposed by Yasin [2014] decomposes the CPI stack in a different manner and instead of looking at computing penalties, uses a top-down analysis hierarchy categorizing execution at high level first. The events are divided in the categories Front end bound, bad speculation, retiring, back end bound. Each of these are subdivided in their different components, which are then in turn divided in their components and so on until reaching the bottom events.

Top Level Breakdown

The issue point for instructions that divides the front end from the back end is chosen as the point where instructions leave the allocation queue (IDQ in Figure 2.2) to the first stage of the execution engine (Rename/allocate/retirement). Events before this point are assigned to the front end category (front-end bound) and beyond this point to the back end category (back-end bound).

The idea behind this model is to provide an accurate characterisation of the events involved in a particular bottleneck. This is different than the aim of the Interval Analysis model, which aims to provide a characterisation for an overall CPI. The Top-Down model avoids the problem of overlapping and over-counting events by focusing in a particular observed bottleneck and looking only at the events involved in that path. So if an event count is high but it is not within the branch of events we are looking at, the count is considered irrelevant. The Top-Down method is then useful when analysing bottlenecks and what causes them but does not necessarily provide a way to characterise the performance of a program as a whole.

Here we will just mention the four main event categories that constitute the top level of the hierarchy:

- Front-end Bound
- Bad Speculation
- Retiring
- Back-end Bound

For a detailed description of the list of events and derived metrics from this model, see Yasin [2014].

2.4 Summary

In this chapter we introduced the concept of performance of a computer program. In order to analyse and understand the time it takes a program to run in a processor, we looked at the processor architecture and how the tasks are organised through a pipeline. We introduced the metric CPI and three performance models to calculate it.

The common starting point for a model is the identification of events responsible for the lost cycles that modify CPI from an optimum performance base value. The way these contributions are calculated depend on the model. The traditional ('naive') models simply consider the number of miss events and assign them a penalty. This kind of calculation accounts for overlapping of miss events that might occur in the presence of optimisation technologies (out-of-order execution and parallelism, for example) in a term 'overlap' which is not calculated. As a result the CPI is a serial decomposition of events and disregards effects in modern not serial processors, losing accuracy.

A different model that attempts to account for these complexities is the interval analysis model where the miss events are modelled as producing CPI changes along the process timeline. Each of these intervals is modelled and analysed in terms of the miss events that contribute to them resulting in a more accurate calculation of the CPI components.

The results of the analysis using both models can be visualised using a CPI stack, where the contributions of the different events are plotted as a stacked histogram. This mode of visualisation gives more insight that raw cycles measurements for the individual metrics.

The models above might over-count events or disregard sources of overlapping of stalls.

A top-down method of analysis proposed by Yasin decomposes the stack in a different manner and instead of looking at the decomposition into miss events, uses a top-down analysis hierarchy. The events are divided into categories, each of subdivided in their different components and so on until reaching the bottom events defined in this model, and forming a hierarchy of events. Bottlenecks can then be investigated by following down a particular branch of the hierarchy. While this model does not provide an overall picture of performance, by concentrating in a particular bottleneck and the events that affect it provides an accurate picture that doesn't miscount events that might overlap or interact.

3 Performance Analysis Methods and Tools

Profiling is the measurement of the performance metrics (Chapter 2) of a computer program running in a processor.

There are two main profiling methods. One is through simulation, where a performance model is used and the program run on a synthetic CPU, measurements of the metrics taken from the results. The other method is through the use of performance counters built in the micro-architecture.

Both methods have their advantages and disadvantages. The simulation method allows the definition of any type of event and is not limited to the availability of built-in counters. It also allows control over the running environment which helps avoid spurious measurements (which could be introduced through background running software or hardware).

In the case of performance counters, measurements reflect the real performance of a program running in a real environment. However, there are many sources of noise which might result in inaccurate results. Also, the number of counters available might be limited (the trend in the recent years is to address this problem). In this dissertation we concentrate on measurements of performance through hardware counters.

In this chapter we will look at hardware perfomance counters and we will discuss some of the tools available for their implementation.

3.1 Performance Analysis Tools

We start this section with a brief overview of the performance analysis landscape.

Whether analysing performance through simulation or hardware measurements, there are two main categories for the type of analysis that can be done (Nethercote [2004]; Thiel [2006]) depending if the analysis is performed before run-time (Static) or during run-time (Dynamic).

Examples of static analysis are implemented by compilers. This kind of analysis involves reading the source code to find bugs and mistakes as well as optimisations of the source code.

Dynamic analysis, on the other hand, is performed during runtime and explores the program's execution. A profiler is an example of a dynamic analysis tool. Some tools for dynamic analysis require instrumentation of the source code, i.e., inserting pieces of code or probes to record and measure performance of sections while running.

As a side note, we would like to point out that authors might differ slightly in their classification of tools and show inconsistency in what they call dynamic or static. If we follow the classification by Nethercote [2004] (as above) we would consider the reading of performance counters by instrumentation of the source code as dynamic. However, if we follow Thiel [2006] it would be considered static.

We think the most important distinctions for the subject of this dissertation is whether the measurements are done through simulation or real hardware counter reads and which measurement method we use.

Either dynamic or static, there are two levels at which we can collect information about the performance of a program. At program level, we instrument the original code by introducing calls to routines that manage the counters. The original code is modified and the effect of the measurement overhead can be significant. One of this methods is self-monitoring, where performance data is collected for a precise block of code by introducing calls to functions implementing counters (Weaver [2015]).

If measuring only at hardware level the program is not modified and we only look at the collected information by the performance counters. One of the methods that use this approach is statistical sampling where counters can be programmed to collect reads at regular intervals or through overflow interrupts. The results then are obtained though statistical averages. The overheads are usually small, however the results are imprecise, and not reliable in the case of small blocks of code (Eranian [2006]; Weaver [2015]).

In the sections that follow we will review some of the tools available that implement performance counters. We start by explaining how performance counters work and how can to access them at lowest level.

3.2 Performance Monitoring Counters

Performance monitoring through hardware counters was initially introduced in the Pentium processors. Since then they have become important tools for application tuning and performance measurements and they exist in all modern processors. The specific events and architecture of performance counters varies across the industry [Intel, 2017, 2018d; Levinthal, 2009].

Performance Monitoring Counters (PMCs) are registers dedicated to the counting of hardware events within the microchip. In Intel architectures they consists of registers called MSRs (Model Specific Registers) which are grouped into a performance monitoring unit (PMU). These registers are divided between performance monitoring coun-

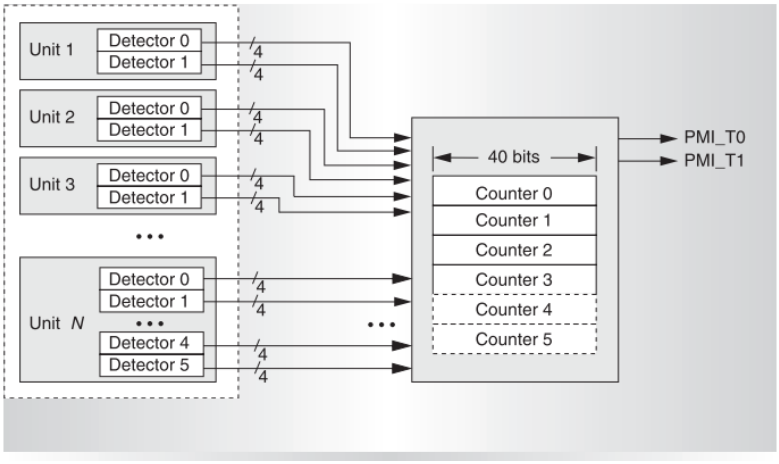

Figure 3.1 General structure of the Pentium 4 event counters and detectors (from Sprunt [2002a])

ters (IA32_PMCx MSRs) and programmable event select registers (IA32_PERFEVTSELx MSRs). The configuration of a particular architectural performance monitoring event requires the programming of its event select register, with the result reported in its corresponding (paired) performance monitoring counter.

Later versions of performance monitoring added fixed-function PMCs (IA32_FIXED_CTRx) and an associated control register IA32_FIXED_CTR_CTRL for their configuration. Each PMC can count only one architectural performance event. The counters and detectors are distributed through the microchip, with event detectors close to the counter blocks. In the KabyLake architecture there are 8 programmable and 3 fixed counters per core (shared among the core's logical processors).

3.2.1 Accessing Counters

The Intel processor architecture provides a protection system that controls the access of programs running on the processor to its resources (memory regions, registers, ports, etc.), implemented through privilege levels which provide different levels of access to the resources. The IA-32 architecture has four privilege levels (0 to 3). The highest privilege is at level zero, the lowest at level 3. Modern operating systems use level 0 for the operating system kernel, making level 0 also known as Kernel Mode in Linux (or Ring 0 in Windows OS). The level 3 is used for less critical programs. This is the level at which user space operates and where most application programs execute [Intel, 2018d].

Depending on the configuration of both programmable and fixed counters, events can be accessed at the different privilege levels of user, kernel, or user+kernel. This is particularly useful to isolate measurements of a given piece of code from procedures run by the kernel.

In the case of performance counters, there are two available modes to access them: user mode or kernel mode. Depending on the right of access, two instructions allow the reading of counters, RDPMC and RDMSR. The counters are configured by writing to the corresponding MSR with the instruction WRMSR.

WRMSR is a serialising instruction. This means no more instructions are sent to the pipeline until the last one waiting to be executed is retired. This instruction can only be executed at privilege level 0.

There are some important differences worth of mention between the instructions RDPMC and RDMSR.

RDMSR can only be executed at privilege level 0, so users need an interface if they want to read counters. RDPMC can be set to execute at any privilege level without the need to enter the kernel (however setting it up requires privilege level 0 and its setup can be complex). Note that this refers to the access to counters and not to what the counters are reading, which is set-up during configuration of the counters to read events that occur at user or kernel (or both) levels.

Also these two instructions also interpret their input argument differently: RDMSR takes the MSR to be read as input, while RDPMC takes its argument as a performance counter number.

For example, the performance counters 0,1,2,3 are programmed using MSRs 0x186,0x187,0x189 and their corresponding counts are available from MSRs 0xc1, 0xc2, 0xc3, 0xc4. Reading the count for PMC0 with RDMSR will require the argument 0xc1 while if using RDPMC the argument would be 0x0.

3.2.2 Performance Events Available

There are five main categories of performance events available on modern microprocessors [Sprunt, 2002b]:

- program characterisation: these events help measure those attributes of a program independent of the processor's implementation (the number and type of instructions, for example)
- memory accesses: to help analyse performance of the memory subsystem (number of memory references, cache misses, for example)
- pipeline stalls: to help analyse the flow of instructions through the pipeline (the number of times the pipeline stalled due to a particular operation, number of clocks the pipeline stalled waiting for memory reads, for example)
- branch prediction: to help analyse the performance of branch prediction hardware (counts of mispredicted branches, for example)
- resource utilisation: to count the portion of time a processor uses a given resource (number of cycles spent using a floating point divider, for example)

Hardware performance counters were mostly introduced independently of the processor performance models like the ones described in Section 2.3. Feedback among researchers

has led to the implementation of counters for events of some of these models, as in Yasin [2014]. A list of all the available events for KabyLake can be found in Intel [2018c].

3.2.3 Challenges Using Counters for Performance Measurements

While it might seem straightforward to use counters once they have been configured, there are complications arising when using them to measure performance.

Measurement bias in performance analysis using counters can be significant [Mytkowicz et al., 2009]. The sources of bias that have been identified are varied. Among these are the link order of the program being tested, the set-up of the environment (operating system) used [Mytkowicz et al., 2009], the number of enabled counter registers, the duration of the measurement [Zaparanuks et al., 2009], the overhead introduced by the software interface being used to read the counters [Molka et al., 2017] and the difficulty in isolating signals running in a real system, over-counting or under-counting of events in the case of certain counters [Weaver and McKee, 2008; Das et al., 2019], the techniques involved in acquiring the reads [Das et al., 2019].

As a result of these findings, there are important considerations to take when using performance counters depending on what they are being used for.

Any general findings based on results obtained by measuring counters need to be verified across different architectures to test their validity. The complexity of the underlying micro-architectural events can result in measurement bias leading to wrong conclusions.

In the case of comparison among different profiling tools and programs (as when analysing benchmarks and program similarity, for example) and following the recommendations outlined by Das et al. [2019], the use of performance counters is not recommended and other alternatives need to be considered.

For the profiling and optimisation of an specific application, it is important to isolate the processes being measured during the test. Context switches are important sources of bias. Their effect can be minimised by saving and restoring the value of the counters before and after a context switch. The identification of deterministic and non-deterministic events is also fundamental in order to know which counters measurements are to be considered reliable: not all available hardware events are useful.

The original aim of this part of the dissertation was to characterise in an accurate manner the comparative performance of different algorithms in a general fashion through hardware. While performance analysis can still be valid in the case of optimising a particular application, we think a more careful analysis of how to determine the performance of a particular benchmark is necessary for comparison purposes.

3.3 Software Tools for Performance Monitoring Counters

Here we will discuss the different options to measure counters in order of decreasing complexity of implementation by the user: direct reading, through implementation of a kernel interface within the program, through a performance counter monitoring tool. This order increments the complexity of the low-level access to a counter, while decreasing the complexity of implementation by a user wanting to measure a given event accurately.

In the section that follows we will assume a performance counter which has already been set to count a given event. We will start by considering the case of micro-benchmarking a small piece of code, i.e., code that takes only few instructions as opposed to a complex application with many functions, variables and subroutines. As an example, Listing 3.1 shows one of the small pieces of code we have measured in our tests (presented later in Section 4.2.2). Using an inline assembly instruction, we add the two input variables 'var' and 'i'.

```
asm volatile("addl %%ebx, %%eax\n\t"
      : "=a" (var)
      : "a" (var), "b" (i) );
```

Listing 3.1 Example of a small piece of code

We also assume a BIOS setup such that there is only one core active with only one thread. This simplifies the event counting implementation as it ensures we all possible activity in the processor is going through the channels we are measuring.

3.3.1 Low-level Performance Monitoring Implementation

RDMSR Instruction Through a Kernel Module

The first approach to measure a performance counter is through an assembly instruction at kernel level. This is, having the privilege level to access directly a counter we can read the contents in its register by using the RDMSR (RDPMC) instruction.

There is a caveat to the apparent simplicity of this approach: counters need to be read through a kernel module, even if the environment settings allow the instruction RDPMC to be used at user level. As mentioned in Section 3.2.3 there are many factors which add noise to the event counts. An implementation through a kernel module allows control over context switches which are an important source of bias. Spurious counts can be minimised by disabling interruptions and pre-emption scheduling, which is only possible at kernel level.

An additional complication is the recollection of measurements. Management of files for input/output is a delicate matter at kernel level which needs to be implemented correctly to avoid system failure and requires expert knowledge of the system kernel programming.

A safe (but limiting) alternative is to output the results as an error to the kernels log file. The measurements can then be extracted processing the text file of the kernel system log.

Another important step for accurate micro-benchmarking is the flushing of the instruction pipeline. This is achieved by implementing a serialising instruction before and after the measurement.

3.3.2 Accessing Counters Indirectly Through the Kernel

Through a Kernel Driver Accessed from the Program

A second approach improves on the practical aspect being more user friendly. Counters are read at user level and the output can be sent to a file for later analysis or manipulated by the user in whichever way he chooses. Implementations of this kind consist of a driver and a program to handle the measurements. The driver acts as an interface with the counters, giving the user access to them.

Interruptions and context switches still need to be handled by the user. This is achieved by a careful set-up management of threads. By creating threads for the events to be measured and setting their priority high, we hope to avoid context switches. This will be mostly an effective solution for for small pieces of code, but as the number of instructions to execute grows, so does the number of context switches during the running time and the probability of meeting one.

While this approach is friendlier to the user, it still involves the setting up of the counters to be measured and a high level of involvement with lower level programming considerations. It also is architecture dependent, as the instructions to interact with the counters, available events and register addresses are architecture dependent.

Through a Performance Monitoring Interface (Kernel Level Implementation)

A kernel driver to access the counters is a performance monitoring interface, however we refer here to implementations that extend their functionalities beyond the simple read and write of counters. Many of these interfaces build on a software abstraction layer over the specitifc instruction architecture layer to increase portability. Three of the most widely used are perfmon2, perfctr and perf_events. All these set up a kernel interface with a user level part, but with different levels of operating system contribution.

Perf_events is integrated in the kernel directly, allowing the Linux kernel to provide full access to the hardware performance counters [Weaver, 2011; Dimakopoulou et al., 2017]. Most operations are performed at kernel level. The interface is built around file descriptors in the filesystem to simplify event naming and the configuration of tools. An event is allocated with a new system call perf_event_open(). At the lowest level Perf_events make use of the low-overhead instruction to read counters RDPMC [Röhl et al., 2015].

perf_events is more user friendly than raw counter reads: users manage events instead of performance counter register addresses and values. The kernel is responsible for programming these events on the correct counters. This is an interface which is under constant maintenance and improvement, making it a very valuable and reliable tool [Dimakopoulou et al., 2017].

Other similar interface is Perfmon2 [Eranian, 2006; Perfmon2, 2018], which is implemented with multiple systems calls rather than a device driver. Perfmon2 provides support for per-thread monitoring where information is collected on a kernel thread basis. The performance counter contents are saved and restored during context switches. The functionalities can be accesses either through instrumentation of the program (requiring the implementation of a library and compilation of the original code), or dynamically by attaching the monitoring session to a running thread (in this case no modifications are needed).

Access to the performance counters registers is done through a register naming scheme which is the same independent of the architecture. This is achieved by operating on a logically PMU which is mapped to the hardware PMU by the interface. This is done in user space (at difference with perf_events which operates totally in kernel space). The PerfMon2 interface uses mainly system calls to interact with the hardware.

Perfctr developed by M. Pettersson of Upsala University [Kufrin, 2005] is distributed as a standalone kernel patch for Linux. It has been designed with self-monitoring in mind (sampling support is limited) through the implementation of a library libperfctr [Eranian, 2006; Weaver, 2015].

3.3.3 High Level Performance Monitoring Tools (PAPI and VTune)

There are many high level performance tools, among the most widely used are PAPI and VTune.

PAPI [Terpstra et al., 2010b; Mucci et al., 1999] is an open source performance application programming interface to access hardware counters which is highly portable and it provides two interfaces to the underlying counter hardware: a high level and a low level interface.

The low level PAPI interface is fully programmable. Groups of events called EventSets that deal with the counters guaranteeing thread safety and proper implementation of registers writes and reads and providing a wide range of features.

The high level interface provides the ability to perform simple measurements of counters one at a time and it can start, stop and read specific events.

Depending on the architecture, PAPI can be built on different substrates (the operating system, a kernel driver or the low-level assembly instructions) depending on the architecture and system. In a Linux OS, PAPI can be built on the kernel, on perfctr or on PerfMon2, giving different overheads for the implementation [Zaparanuks et al., 2009]. Through the implementation of their libraries, PAPI inherits important functionalities

such as its per-process monitoring capability. In the case of PerfMon2, for example, this means that the principle of operation of a monitoring session then consists in saving and restoring values of counters under context switches [Eranian, 2006]. It has been found that the substrate chosen affects the overhead of the measurements [Zaparanuks et al., 2009]. As a result of continuous development not only of PAPI but also of the substrates and drivers, the newest versions with the lowest overhead for Linux systems on x86 architectures build on perf_events [PAPI Software].

VTune is a commercial application for performance analysis of x86 architectures (both Intel and AMD), only available for Linux and Windows operating systems. The advance hardware profiling event features of the profiler are only available for Intel systems. VTune uses statistical binary instrumentation, with a usual overhead of 8%. The events defined in the Top-Down analysis method (see Section 2.3.5) have been incorporated in newest versions. VTune has a graphical user interface making it very user friendly and easy to implement in comparison with other tools.

Some other widely used performance tools worth of mention due to their popularity are LIKWID-perfctr [Röhl et al., 2015; Rohl et al., 2017], PerfSuite [Kufrin, 2005] and HPCToolkit [Adhianto et al., 2010]. Likwid-perfctr builds on msr-driver, a Linux kernel module which allows access to the MSRs by bypassing the kernel (using an alternative device file interfance to perf_event). PerfSuite is another set of tools including graphical tools to visualise performance data and software libraries to access performance counters in threaded applications. PerfSuite is built on PAPI, PerfMon2 and Perfctr. HPCToolkit is a suite of tools that uses statistical sampling of timers and hardware performance counters to collect its data. Among the advantages of sampling instead of instrumentation for measuring performance, there is no need to modify the source code and also it has lower overhead. HPCToolkit overhead is estimated between 1% to 5% [Adhianto et al., 2010].

3.4 Summary

We introduced hardware performance counters and explain how they can be accessed. We discussed the advantages and disadvantages of using performance counters.

We discussed different methods to monitor performance of a computer program. We review a variety of software implementations based in the use of counters, ranging from a basic low-level implementation to some of the most complete high-level performance tools available. We briefly pointed out their advantages and drawbacks.

4 Measuring Counters: Implementations and Results

We start this chapter by showing how to implement different tools for measuring counters. We obtain measurements for a small piece of code (micro-benchmarking) using these tools, and we discuss and analyse these results.

We start by implementing a simple kernel module to measure cycles and instructions. We then move into a more portable solution by using an example implementation of a hardware counter interface and finally move into PAPI.

4.1 Program Implementations for Micro-benchmarking

In this part of the dissertation we are interested in benchmarking of small pieces of code. This is known as Micro-benchmarking, and it is only applicable to short pieces of code. As the amount of instructions to benchmark increases, the overhead of the measurement process with respect to the results becomes less important, however the contribution of other factors becomes more important (such as context switches, per-thread measurements, etc., see Section 3.2.3). On the other hand, using performance measurement tools that incur higher overhead are suitable for larger pieces of code but might be unsuitable for micro-benchmarking [Jarp et al., 2008; Lee, 2006; Bakhvalov, 2018]. We start by showing a low-level implementation to measure counters (see Section 3.3.1).

4.2 Probing Counters through a Kernel Driver

One of the most meaningful quantities when micro-benchmarking is the number of cycles required to execute an instruction. Traditionally this was returned by the assembly instruction RDTSC in Intel architectures, which reads the time stamp counter (TSC) [Intel, 1997]. However, over the last ten years the development of precise timing counters has meant that the function of RDTSC migrated from counting precise cycles to providing precise time [Intel, 2018b]. As a result of being designed to provide accurate time, the disadvantage of using the TSC to count cycles is its frequency dependence. While the reference frequency of TSC has been designed to be constant, the CPU frequency might change during the measurement process, introducing a divergence between TSC counts and CPU cycles. Precise cycles can instead be accessed through an architectural counter designed with that purpose and which we chose to use in this work.

4.2.1 Implementation (Kernel Module Program)

To implement our kernel module program we have followed the considerations and findings for precise benchmarking of code execution times by Paoloni [2010] and their proposed implementation with the following main modification: in their work they use a combination of RDTSC and RDTSCP instructions to read the time stamp register and calculate cycles, we make use instead the fixed architectural counter CPU_CLK_UNHALTED [Intel, 2018a] accessing it with the instruction RDMSR (see Section 3.2) allowing us to count cycles directly (see beginning of Section 4.2 above).

The implementation consists of a kernel driver which must loaded for execution with kernel privilege level which provides control over CPU usage. The program guarantees total ownership of the CPU by disabling scheduling preemption and hard interrupts (only possible because of privilege level).

Additionally we need to deal with out-of-order execution. The pipeline can be cleared out by using a serialising instruction inserted before the block of code to benchmark (for a list of all serialising instructions see [Intel, 2018c] section 8.2). We serialise our code using the CPUID instruction.

These considerations are implemented in our code as:

```
#include <linux/hardirq.h>
#include <linux/preempt.h>
#include <linux/sched.h>

......

   //Minimising the effect of interrupts and context switching
   preempt_disable();
   raw_local_irq_save(flags);

   //Serialising instruction to flush the pipeline:
   asm volatile("CPUID\n\t"
            ::: "%rax", "%rdx", "%rcx", "%rbx");


   //Measurements here

   //Flush the pipeline again
   asm volatile("CPUID\n\t"
           ::: "%rax", "%rdx", "%rcx", "%rbx");


   //restore flags and re-enable scheduling
   raw_local_irq_restore(flags);
   preempt_enable();
```

where flags is a variable to store system flags to restore the system to its original estate before the disabling of scheduling.

To measure a MSR we need to:

- load the register ECX with the address of the MSR we want to read
- execute the instruction RDMSR
- read the output in registers EAX, EDX

The MSR address to read counts of instructions retired is 0x309, to read cpu clocks 0x30A. The following line of code shows how to read instructions retired using inline assembly:

```
asm volatile("mov $0x309, %%ecx\n\t"
   "rdmsr\n\t"
   "mov %%edx, %0\n\t"
   "mov %%eax, %1\n\t": "=r" (cycles_high1), "=r" (cycles_low1)
   :: "%rbx");
```

Listing 4.1 Reading instruction for the km program

The output of the measurements is printed to the kernel message error interface which in turn is logged into a system file. We extract the measurements after the driver finished its run by reading the text file.

The application is written in C and was compiled with the optimisation level -O3, using inline assembly for the implementation of instructions. The full code can be found at https://github.com/Silvia-prog/Perf-Counters.

BIOS Settings - Experimental Setup

In order to minimise the factors that can introduce noise and to simplify the implementation we run the system with only one core with Hyper-threading disabled (to ensure the only thread being measured is the one we are running). We also disabled Turbo mode and Frequency Scaling to run the CPU at a uniform rate.

The setting up of the counters was performed by using msr-tools, a command line utility for linux that allows the setting up of the hardware performance counters[1]. To enable the 3 fixed counters and the 4 programmable counters we write 0x70000000f to the 0x38F register (IA32_PERF_GLOBAL_CTRL). The MSRs registers used in this section are the fixed counters at addresses 0x309 (instructions retired) and 0x30A (unhalted clock cycles). It is important to verify before running the measurements that these registers are correctly set as sometimes the use of other functionalities (as perf, for example) might disable them. Readings on counters which are disabled will not result in errors, but in incorrect readings.

[1] man msr will provide information on how to load and use the msr-tool

4.2.2 Results

We first measured the overhead of the benchmarking code but leaving an empty space in the place where the code to benchmark will go, giving 103 cycles and 241 instructions over 1000 samples, with no variance from these values.

We sampled three different codes. They all consist of instructions inside a loop of variable size. We tested loop sizes of 1, 5, 10, 50, 100, 1000, 5000 and 10000.

The first group (sample a) consist of 38 assembly instructions combined randomly to introduce chain dependencies and then avoid out of order execution as much as possible.

The second group (sample b) adds two variables:

```
asm volatile("addl %%ebx, %%eax\n\t"
             : "=a" (var)
             : "a" (var), "b" (i) );
```

Listing 4.2 Sample b

where var is defined as volatile (to disallow optimisations by the compiler) and i is the index variable for the current sample.

The last group consists of loops with CPUID instructions, we tested two samples. Sample a:

```
for(k=0;k<100;k++)
   asm volatile("CPUID\n\t"
      ::: "%rax", "%rdx", "%rcx", "%rbx");
```

and Sample B:

```
for(k=0;k<100;k++)
asm volatile("CPUID\n\t"
   :: "a" (0) , "c" (0) : "%rdx", "%rbx");
```

Depending on the optimisation flag, the code produced at instruction level varies. Compiling with the flag -O0 most optimisations are disabled, while using the flag -O3 might introduce changes to the original code beyond our control.

We will first discuss the results for measurements of cycles in the case of the different loops using the kernel module as the benchmarking tool. We present the results as a series of plots for different loop sizes. The first plot of the group titled 'Overhead' is the cycles measured when the loop size is set to 0.

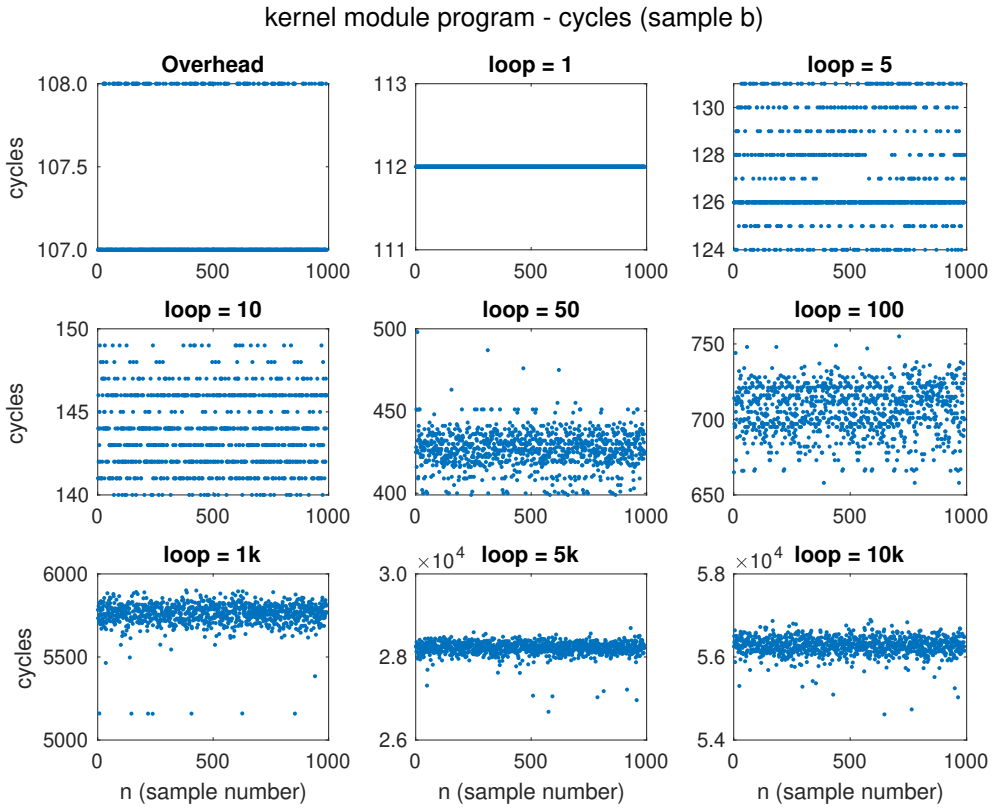

Figure 4.1 Cycles for different loop sizes (sample b)

Sample b

Figure 4.1 shows the results for sample b (see Listing 4.2).

Notice that the value of the overhead takes the two values 107-108. As the cycles increase (through bigger loops) so do the number of distinct measured values, which seem to stay in clear 'channels', with practically no noise. As the loop sizes keep increasing, the noise becomes more prevalent and begins to blur the definition of the channels. These transition becomes clearer in Figure 4.2, where we show the same measurements as in Figure 4.1 without outliers, which we have defined as points lying outside the 10th and 90th percentile range.

Figure 4.3 shows instructions retired (these are measured together with cycles). Notice that the number of instructions per loop remains constant through all measurements. We can calculate what is the number of instructions retired per loop pass by subtracting the overhead and dividing for the number of passes. The result of this calculation is 8 instructions per loop pass, consistent along all loop sizes.

Figure 4.4 shows the CPI calculated from these measurements, from where we subtracted the overhead from both cycles and instructions beforehand. Up to loop size 10 cycles per instruction decrease with the loop sizes, indicating lost efficiencies for lower loop sizes. Then an increment is observed for a loop size of 50, decreasing again from this value as loop sizes increase, reaching a constant value. Decreases in the amount of cycles as the

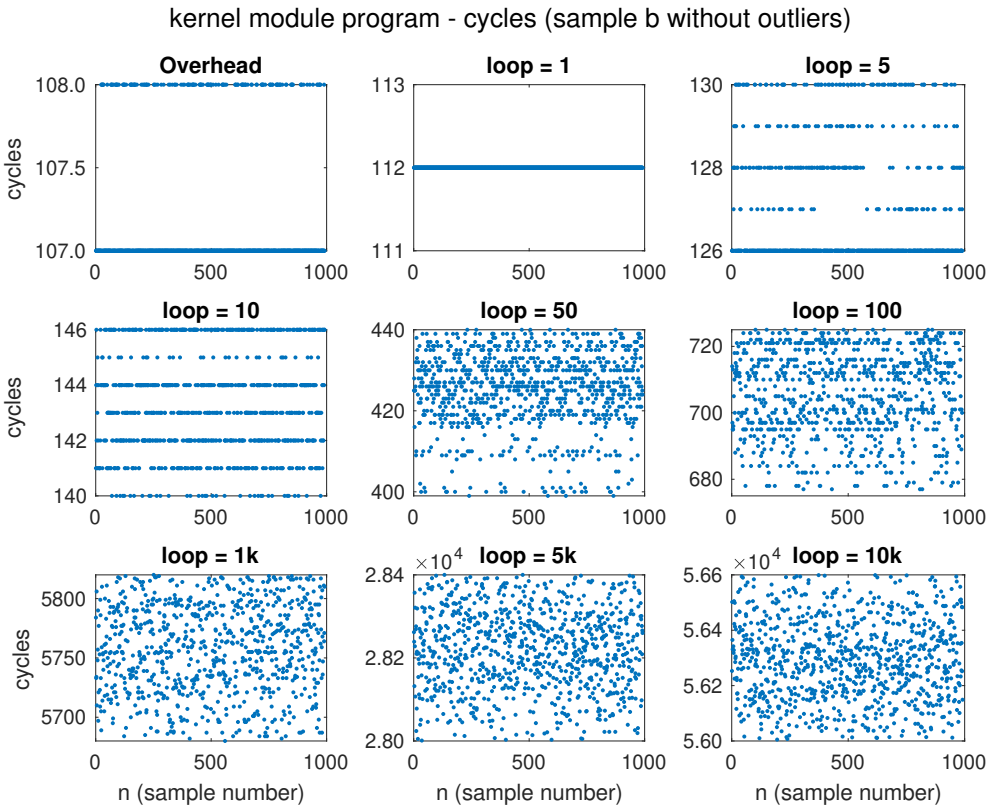

Figure 4.2 Cycles for different loop sizes (sample b) without outliers

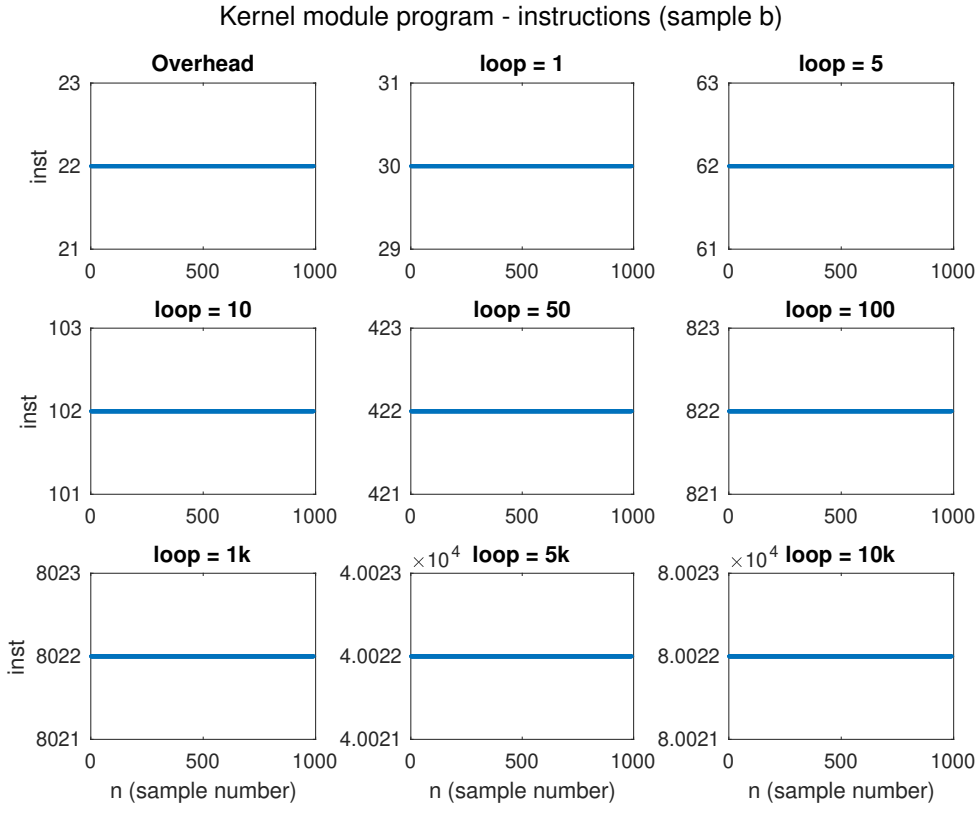

Figure 4.3 Cycles for different loop sizes (sample b)

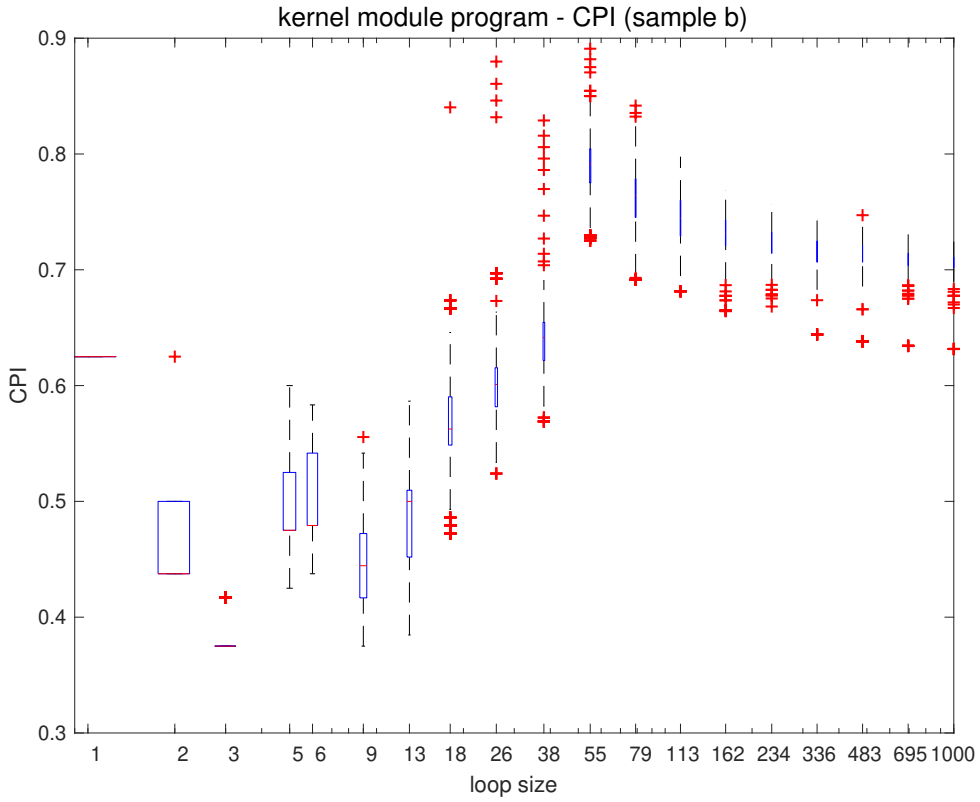

Figure 4.4 CPI for sample b)

loops increase are expected, as running the same code for a longer time would incur in less mis-speculations and misses of the different buffers. The sudden jump observed might be due to effects of the LSD (Intel [2016], section 3.4.2.5).

CPUID

Figure 4.5 shows the results when using a CPUID instruction in the body of the loop. In some cases we observe a jump in the cycles measured as the amount of samples taken increases. In the case of loop size = 5 this effect seems to be observe momentarily between samples around 500 and 800 (in the case of loop size = 10 few initial measurements show higher counts). The CPUID instruction uses the registers EAX, EBX, ECX and EDX and can return the processor identification and information about its features, depending on the value set on the EAX and ECX registers when the CPUID instruction is executed. In these first set of measurements we overlooked initialising these registers. A further set of measurements with the registers initialised to a value of 0 for both is shown in Figure 4.6. We think the cause of the jumps in the first set of measurements was due different initial values in the registers when the CPUID instruction was executed. This is also consistent with the observation of the high number of cycles incurred per instruction in the first set of measurements compared to the second one: for a single pass (loop=1) set A averages 1040 cycles, while set B takes around 210 cycles.

The appearance of channels in set B is very clear. We think these might be related to

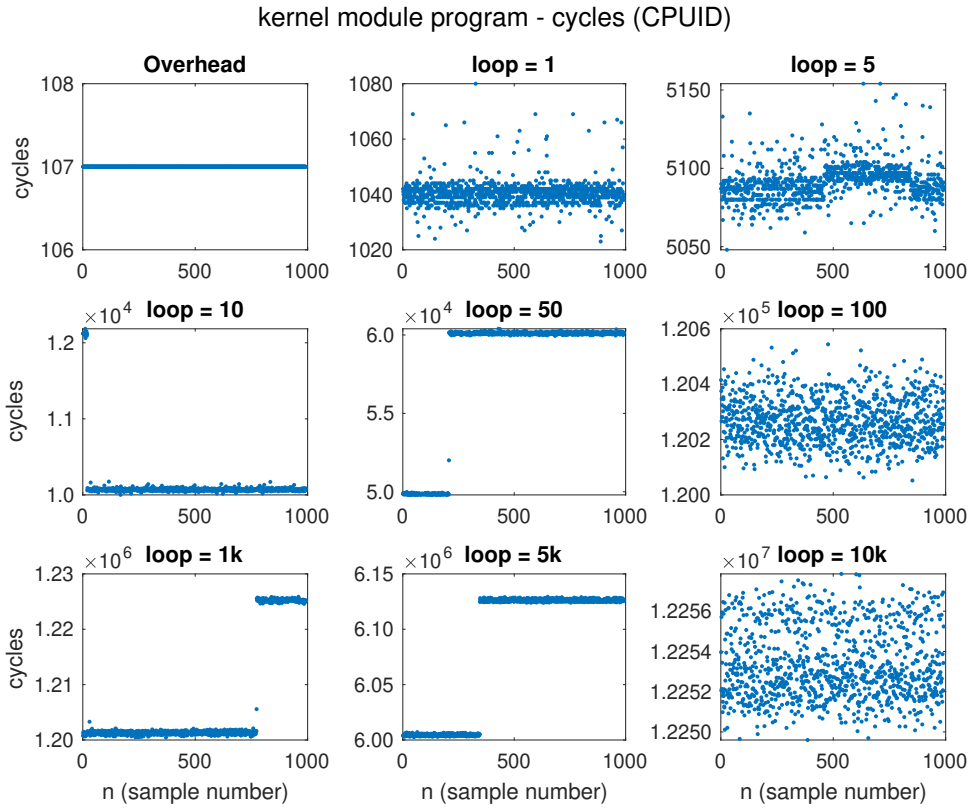

Figure 4.5 Cycles for different loop sizes (CPUID)

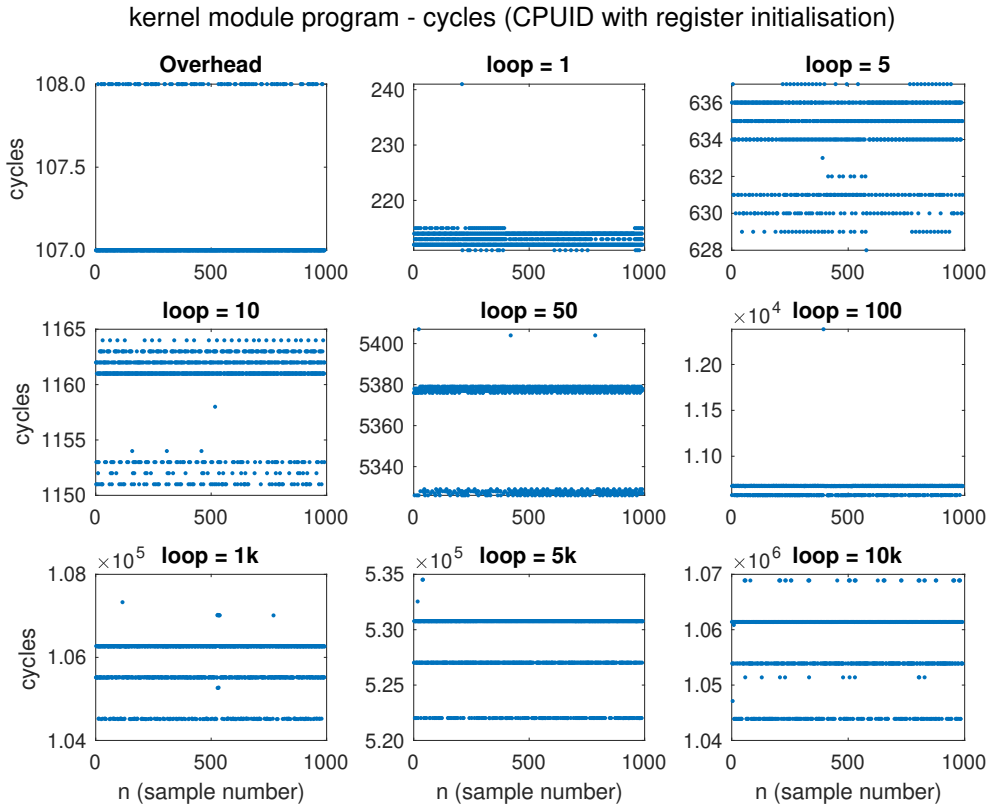

Figure 4.6 Cycles for different loop sizes (CPUID) with EAX and ECX registers initialised to same values at the time of execution

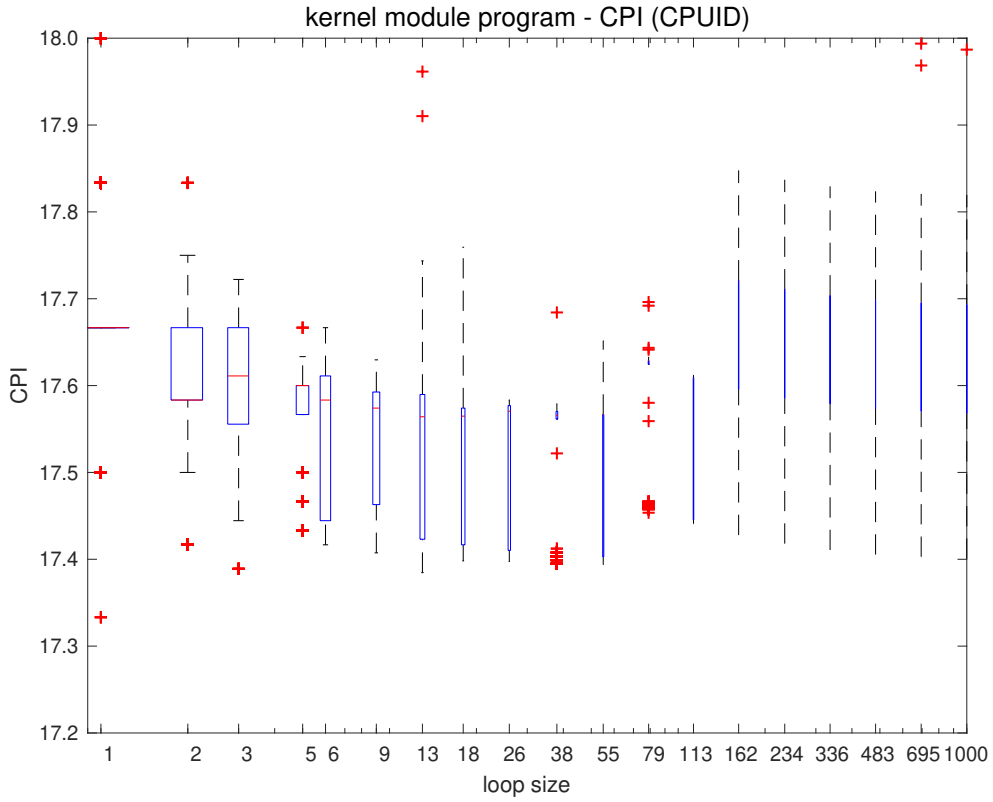

Figure 4.7 CPI for different loop sizes (CPUID). The subtracted overhead is 107 cycles.

technologies for predictions and pre-buffering of instructions and data.

Instructions are consistent across loop sizes and give 6 instructions per loop pass (4 for set A), results we used together with those of Figure 4.6 to calculate the CPI shown in Figure 4.7. The overhead from both cycles and instructions was subtracted beforehand.

Sample a

Figure 4.8 show the results for sample a. This sample was designed with the idea of forcing chain dependencies to ensure the loop does not qualify for the LSD (loop stream detector), with a total number of instructions bigger than 64 m-ops (the limit to qualify for LSD). We measured 119 instructions per loop pass. While we initially considered the possibility that some of the observed features in sample b and CPUID (as the channels) might be introduced by the LSD we can see, however, the same effect present already for a loop size of one (i.e., apart from the body loop size too big to qualify, we can expect the loop stream not working in this case as it is only one set of instructions). Additionally we observe some periodic patterns for the lower loop sizes. One reason might be instructions/data blocks periodically going in and out of the lower caches, or an artefact of compiled optimisation; however this is purely conjecture and certainly the root cause is an area that could be explored further in the future.

For the CPI see Figure 4.9.

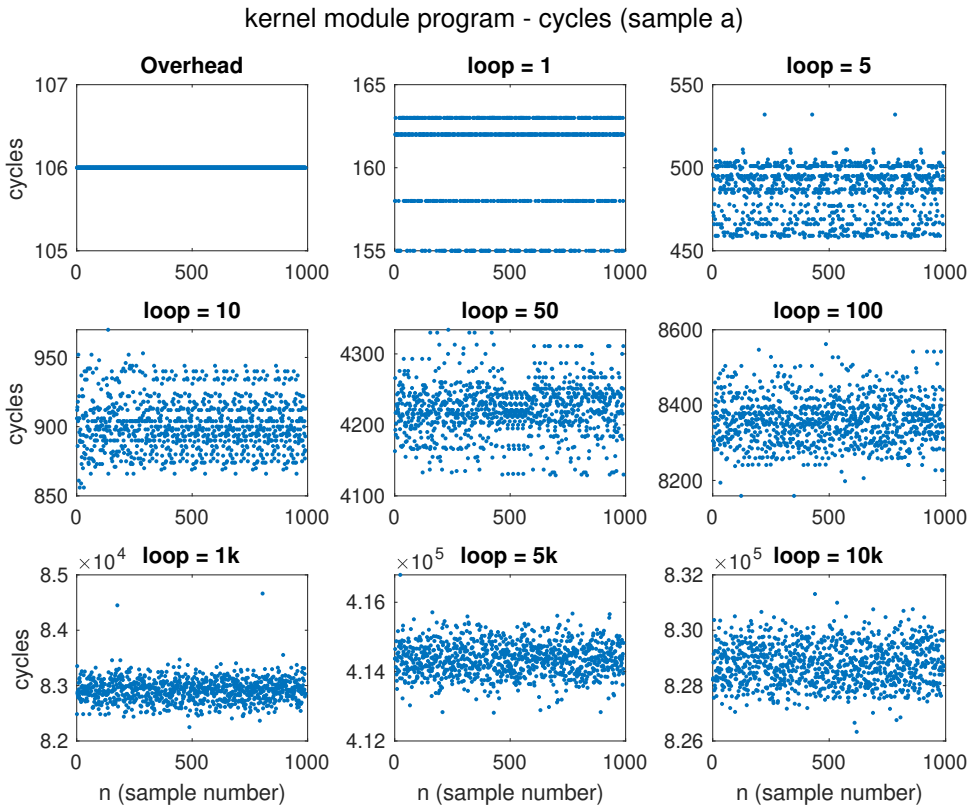

Figure 4.8 Cycles for different loop sizes (sample a)

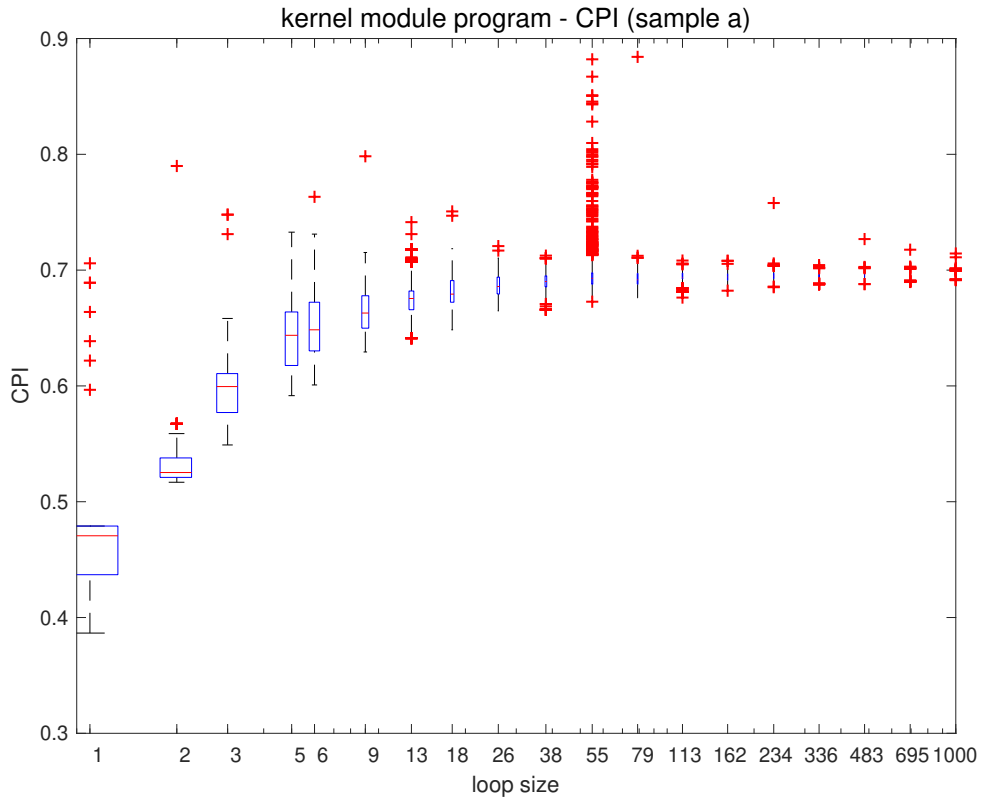

Figure 4.9 Cycles for different loop sizes (sample a)

4.3 PMC Program

The second approach we used to access performance counters provides more capabilities for the user. This program was developed by Agner Fog. It implements the linux driver MSRdrv to get privileged access to set up the counters and to read them from user space. The driver must be loaded before running the test program.

4.3.1 Implementation

The PMCTest package allows multi-threaded testing. The advantage of this over the previous method is that is closer to the real environment under which a program is run. It is not necessary to modify the BIOS settings to disable multiple cores and hyper-threading. Also the subtraction of the overhead is included if using this program to benchmark, running a reference measurement before measuring the code to test.

The program provides capabilities to easily program counters. A set of the most common events are already set up, but more can be added to the source code. Additionally counters can be selected and started from the command line (with the same functionalities as msr-tools), and accessed from a program by using the functions provided by a 'timingtest.h' library.

The reading of counters is performed using the rdpmc instruction (instead of rdmsr).

We adopted the option of including the library 'timingtest.h' in our testing program. In this case we still have an overhead that we need to take into account in later calculations. Also there is the option to perform a serialisation of the instructions previous to any readings. The serialize() function was included prior to the reading of counters with readpmc() function before the start of the loop code. It is recommended to serialise again before the readpmc() after the loop end, however measurements with and without it only showed an increase in the overhead with no qualitative differences in the results. To allow for better comparisons with the kernel module measurements (were no serialisation was present prior to the last read) we omitted the suggested serialisation and performed all measurements serialising only before the first counter read and after the end counter read.

4.3.2 Results

Figure 4.10 shows results (after eliminating outliers) for sample b using the library provided by the PMCTest package. The overhead for instructions measured was 55, and instructions per loop pass was 10. Recalling the results when measuring with the km program (Figure 4.1) the overhead in that case was 107-108 cycles and 22 instructions, with instructions per loop pass of 8. The differences in the cycles overheads are expected, accounted for by the differences between the rdmsr and the rdpmc instructions (see, for example, Weaver [2012]) and overall differences in the programs. The difference of 2 cycles in the instructions overhead are also due to differences in implementation of the reading instruction: the

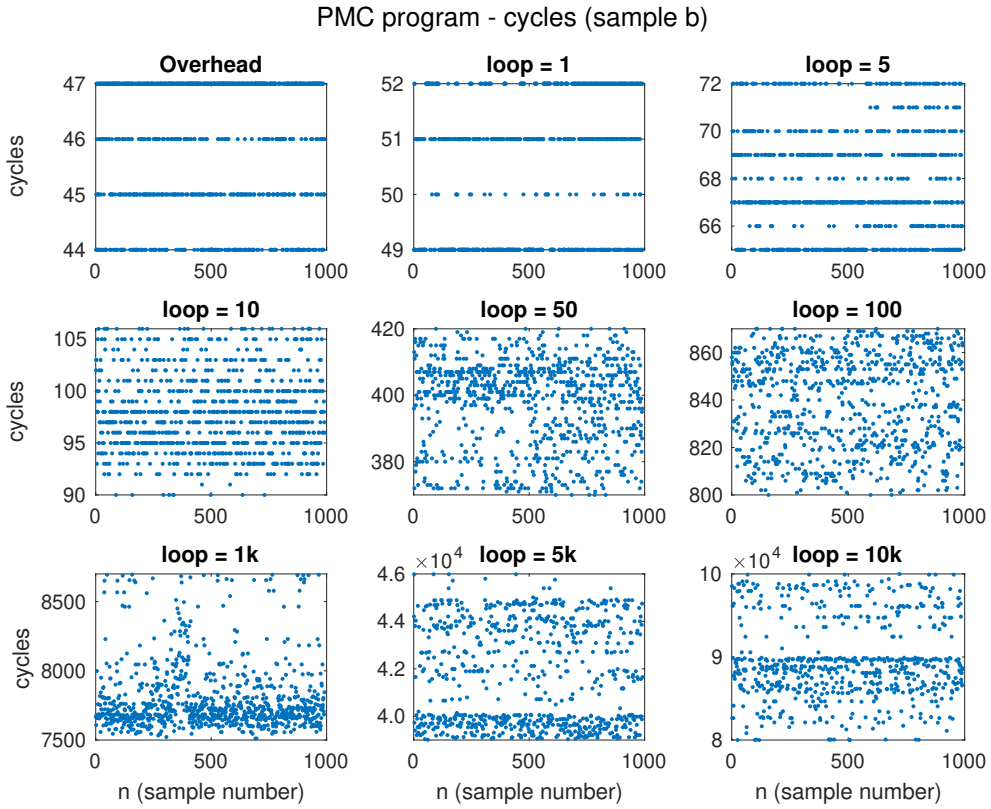

Figure 4.10 Cycles for different loop sizes (sample b), no outliers

function rdpmc() shifts the measured readings within the function body (see Listing 4.3), while in km this operation is performed outside of the measuring loop (Listing 4.1).

Figure 4.11 shows results for cpuid, which are consistent with the measurements with km (Figure 4.6) but show more noise, which is expected as the environment is noisier. As the measurements become longer the possibility of interruptions and other spurious signals increases, evidenced in loops 1000 to 10000. The instructions per loop pass were measure at 8, again consistent with km measurements at 6, the difference in cycles accounted for as in the analysis for the sample b results above.

```
static inline uint64_t readpmc(int32_t n) {
   uint32_t lo, hi;
   asm volatile("rdpmc : "=a" (lo), "=d" (hi) : "c" (n) : );
   return lo | (uint64_t)hi << 32;
}
```

Listing 4.3 rdpmc() function as implemented by the PMCTest program

Finally, Figure 4.12 shows measurements for sample a. Instructions per loop pass were measured at 121 (119 when using the km program). The fine periodic detail present when using km is lost in these measurements, which again is expected as the environmental noise has increased.

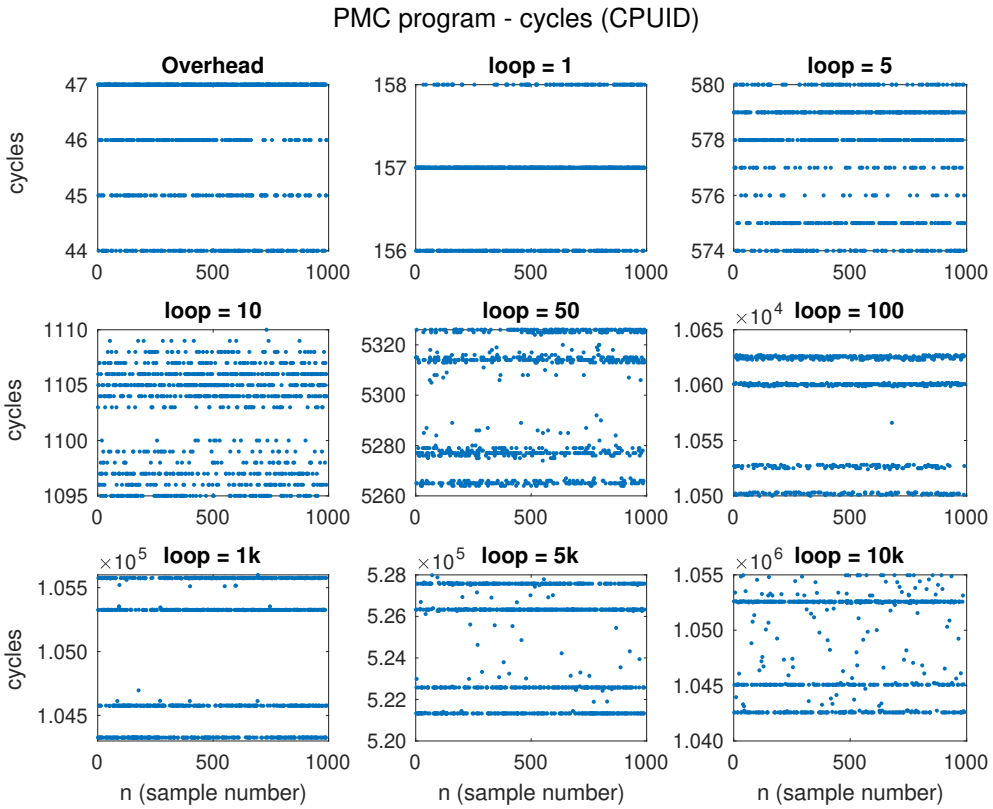

Figure 4.11 Cycles for different loop sizes (sample b), no outliers

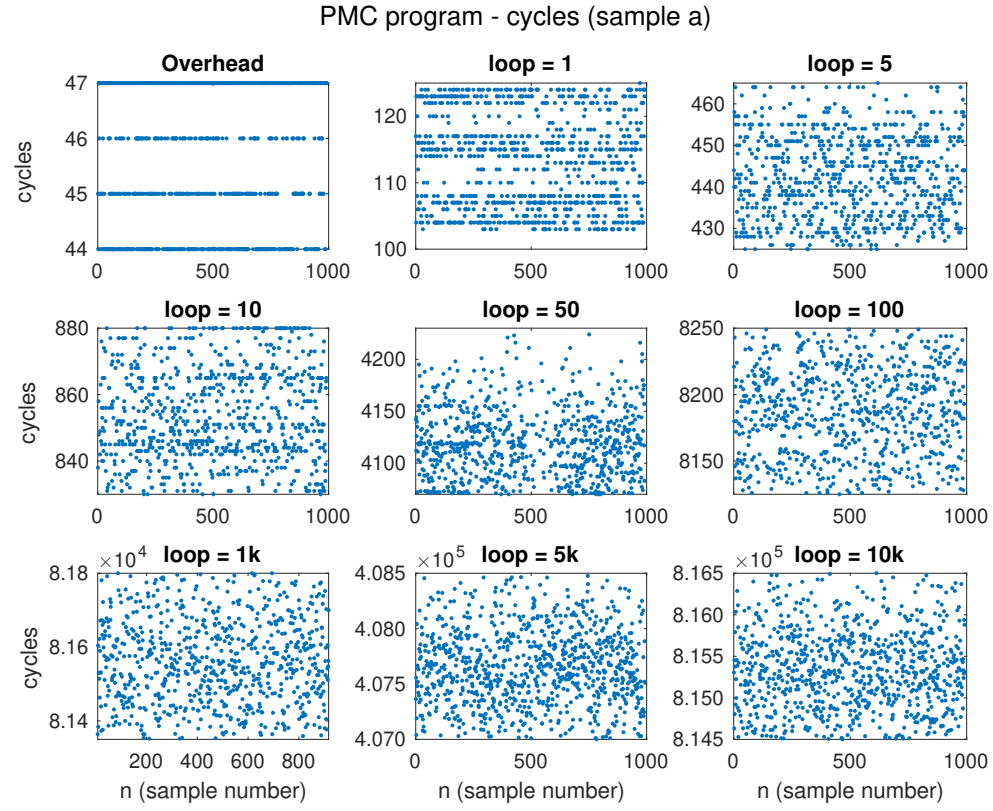

Figure 4.12 Cycles for different loop sizes (sample b), no outliers

4.4 PAPI

The third implementation uses the low level API from the PAPI library [Terpstra et al., 2010a]. PAPI is written in C, with the function calls in the C interface defined in the header file papi.h. PAPI provides both preset and native events. Preset events (predefined events) are commonly found in most CPUs and can be accessed directly through common symbolic names (PAPI preset name), which are mappings to the machine specific definitions for a particular event allowing portability of the interface. Native events constitute all the possible available events in a machine, including events not mapped into preset names and can be configured according to the particular architecture. The events we measured (cycles and instructions) are available as preset events so we do not need to worry about the specific configuration and are accessed by their event codes (preset names) PAPI_TOT_CYC and PAPI_TOT_INST.

4.4.1 Implementation

To store the counter measurements we create Event Sets, collections of hardware events (preset or native) which are measured together. We created an Event Set with two component events, PAPI_TOT_CYC and PAPI_TOT_INST. Once the event set is created and before the measurement the set need to be started, which begins recording the counter reads. A function to read the event set outputs the readings into a vector of values that then can be accessed by the user. Listing 4.4 shows the reading procedure (for the full implementation see https://github.com/Silvia-prog/Perf-Counters). The static library libpapi.a must be linked at compilation time.

```
/* Start counting */
if ( (retval = PAPI_start(EventSet)) != PAPI_OK)
   ERROR_RETURN(retval);

/* read the counter values and store them in the values array */
if ( (retval=PAPI_read(EventSet, values0)) != PAPI_OK)
   ERROR_RETURN(retval);

/***********************
* code to measure here *
***********************/

/* read the counter values and store them in the values array */
if ( (retval=PAPI_read(EventSet, values1)) != PAPI_OK)
   ERROR_RETURN(retval);
```

Listing 4.4 Readings with PAPI event sets

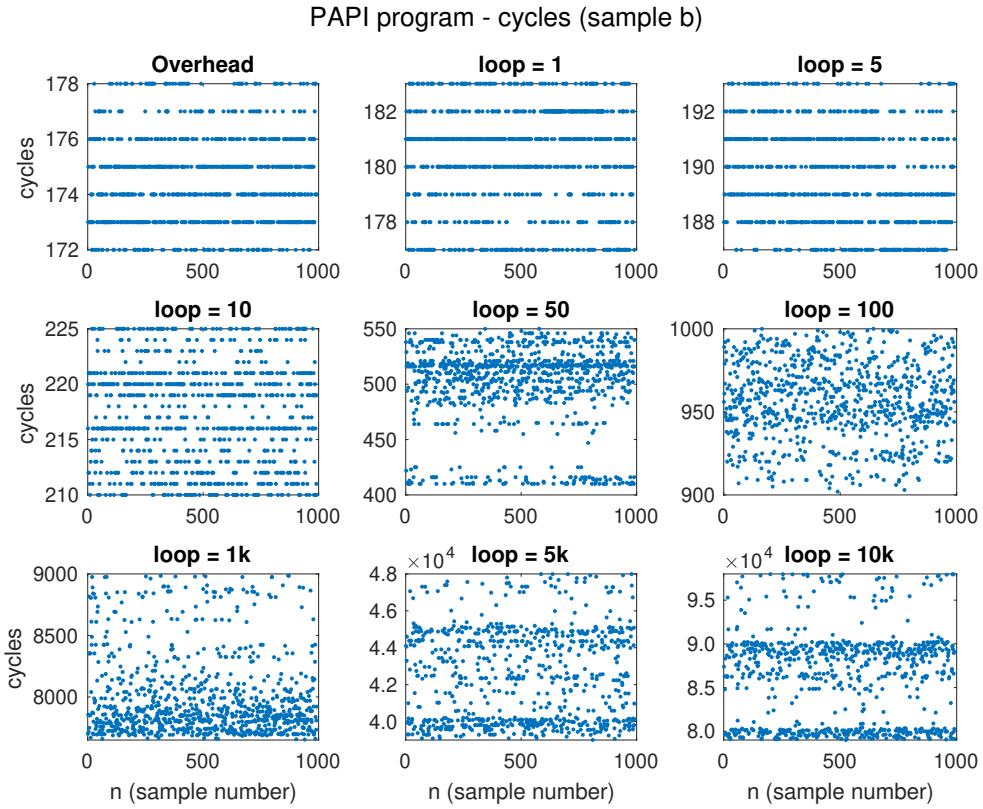

Figure 4.13 Cycles for different loop sizes (sample b), no outliers

4.4.2 Results

Figure 4.13 shows the cycles measured for sample b. While the overhead is higher than both km (107 cycles) and pmc (47 cycles) at 180 cycles, there is a strong noise contribution absent in the other implementations. Instructions, however, are very clean with a base overhead measured of 295 and 10 instructions per loop pass.

For we observe a couple more spurious measurements, the instructions per loop pass were measured at 8. The features observed for loops with a CPUID instructions still remain and appear visible once outliers are removed so we are showing the results after removing them, Figure 4.14.

For sample a, again the noise is prevalent compared to previous measurements, and instructions per loop pass were measured at 121.

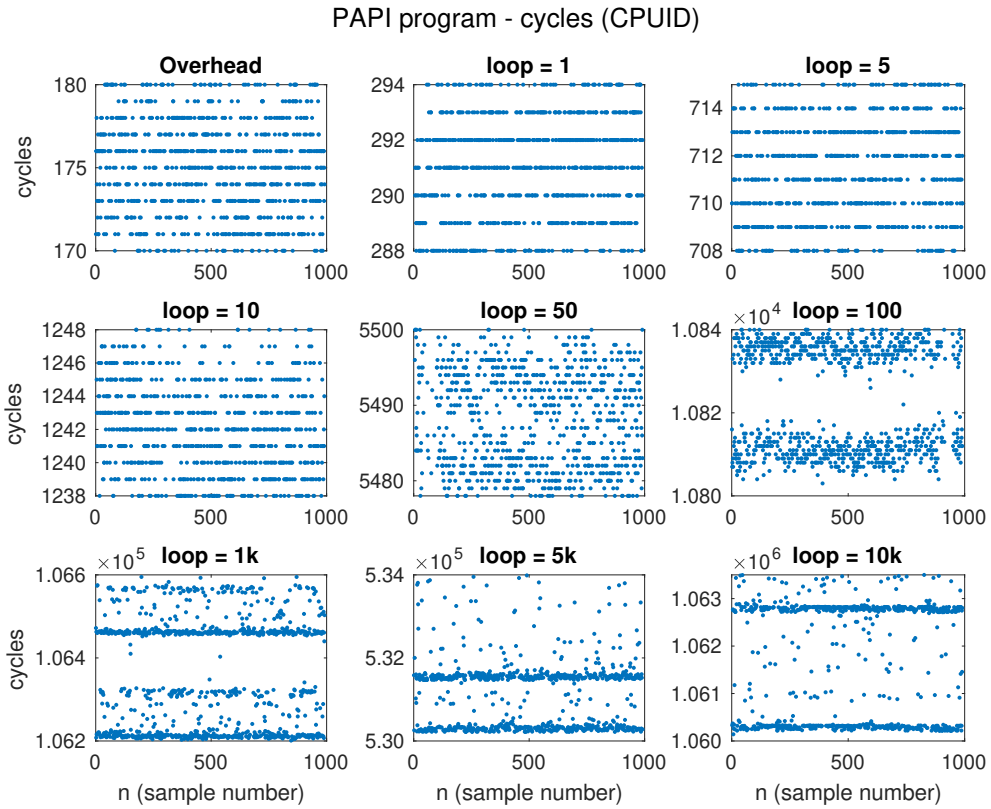

Figure 4.14 Cycles for different loop sizes (cpuid), no outliers

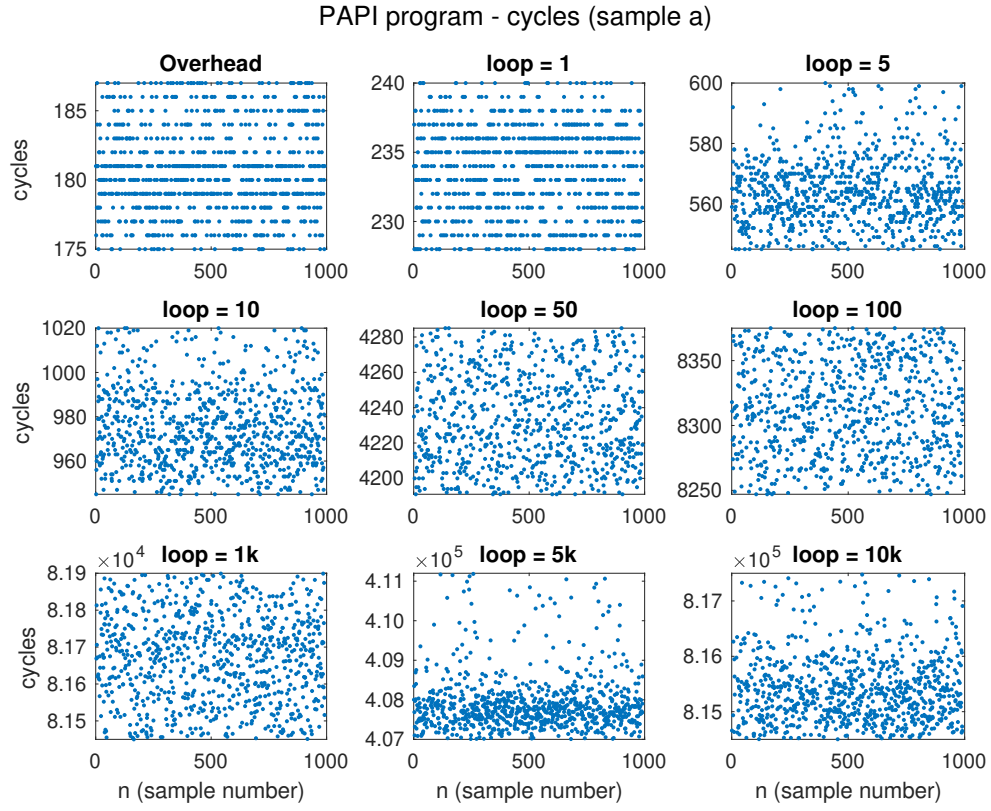

Figure 4.15 Cycles for different loop sizes (samplea), no outliers

4.5 Summary

We measured using performance counters the amount of cycles it takes a CPU to process three small pieces of code using three different approaches of varying complexity. Table 4.1 summarises the results obtained using the three methods, which are consistent and show slight variations in the readings (not taking into account the overhead) which are expected due to differences in the implementations for reading the counters.

	km	**PMC**	**PAPI**
Overhead (cycles)	107	44-47	$\sim$ 180
Overhead (inst)	22	55	295
inst/pass sample b	8	10	10
inst/pass cpuid	6	8	8
inst/pass sample a	119	121	121

Table 4.1 Comparison of results for the three different methods

Overall, the most difficult implementation and limited in its capabilities from the three we have tested is km, while PMC and PAPI do not have the limitations and are more user friendly. Results obtained with PMC and PAPI are comparable and have similar noise levels, and while PAPI has a higher overhead it is highly portable, with most performance counters already preset. We would use PMC for the micro-benchmarking of small pieces of code where low readings are expected and the overhead in using PAPI would affect the results, as the difficulty in implementation is balanced by the lower noise levels. For bigger pieces of code and general profiling (benchmarking of a whole program) PAPI is a better choice as portability and easy of use outweighs the impact of a higher overhead which becomes less relevant as the size of the program (i.e., total amount of instructions to measure) increases.

5 Data Structures and Algorithms in Evolutionary Computation for Multi-Objective Optimisation Problems

In this chapter we present the concepts of data structures with a focus on trees, and the operations that are used to manipulate structured data. We discuss the performance and analysis of algorithms used to perform these operations. We end the chapter by introducing the Multi-Objective Optimisation problem and its solution using evolutionary computation, and we discuss the different data structures used through this method.

5.1 The Multi-Objective Optimisation Problem

Many engineering, economics and logistics problems involve choosing an optimal configuration of parameters or taking an optimal decision in the presence of trade-offs involving two or more conflicting objectives. In contrast with problems where the best solution can be determined by finding a global maximum or minimum, a multi-objective optimisation problem does not have a single solution satisfying all objectives simultaneously but instead a set of alternative optimal solutions known as the Pareto set.

Mathematically we define a Multi-Objective Optimisation Problem (MOP) as minimising (or maximising) the objective function $\boldsymbol{F}$

$$\boldsymbol{F}(\boldsymbol{x}) = (f_1(\boldsymbol{x}), ..., f_k(\boldsymbol{x})),$$

where $\boldsymbol{x} = (x_1, ..., x_n)$ is a decision vector that belongs to a feasible region S of the decision space, determined by the constraint functions of the problem; $f_i : \mathfrak{R}^n \to \mathfrak{R}$ the conflictive objective k functions we want to minimise (or maximise) simultaneously. The image of $\boldsymbol{F}(S)$ is called the feasible objective region.

An objective vector $\boldsymbol{u} = (u_1, ..., u_k)$ is said to dominate another vector $\boldsymbol{v} = (v_1, ..., v_k)$ if and only if $u_i \leq v_i$ for all i, and $u_i < v_i$ at least for one i, where $i \in \{1, ..., k\}$. This condition is known as Pareto dominance, and it is used to compare solutions and identify those that are Pareto optimal: $\boldsymbol{x} \in S$ is Pareto optimal when $\boldsymbol{F}(\boldsymbol{x})$ is not dominated by the feasible objective region. The set of all Pareto optimal solutions is called the Pareto

optimal set. The image of the Pareto optimal set is defined as the Pareto front, i.e., all the $\boldsymbol{F}(\boldsymbol{x})$ such that $\boldsymbol{x}$ is part of the Pareto optimal set. Finding the solution to a MOP means finding its (complete or partial) Pareto optimal set.

Several mathematical programming techniques have been developed to solve some of multi-objective optimization problems. The limitations of these techniques, such as requiring convexity of the Pareto front, continuity of the objective functions, etc. Metaheuristics such as genetic and evolutionary algorithms, tabu search, simulated annealing, ant colony optimisation and particle swarm optimisation (among others) have been developed to overcome such limitations. Within this group, Evolutionary Algorithms (EAs) offer advantages over other methods that have made them one of the most popular techniques. EAs can deal simultaneously with a set of possible solutions, combining efficiency with simplicity of implementation [Coello et al., 2007; Eiben and Smith, 2015; Coello, 2018].

5.2 Evolutionary Algorithms for Multi-objective Optimisation

Multi-Objective Optimization Evolutionary Algorithms (MOEAs) have become standard methods to find solutions approximating the Pareto front of MOPs [Deb, 2001; Coello Coello, 2006; Zhou et al., 2011]. We will start by briefly introducing and outlining the principles of operation behind EAs. We will then explain how they are used for multi-objective optimisation.

An EA operates over a set of possible solutions to the problem, called the population. Each solution is called an individual and contains all the decisions variables of the problem. In order to determine how good a particular solution is respect to others, a function called fitness function is defined (normally a variation of the objective function). When this function is evaluated with a solution, it provides a measure of that solution's fitness. To begin, an initial population is randomly generated. Individuals are then chosen from the population through a selection process, usually a random selection from the population, with a probability of selection based in their fitness such that the fittest individuals have a higher probability of being chosen. This constitutes the mating pool from which a new set of candidate solutions is produced, usually by methods that involve crossover (recombination of two solutions) and mutation (small random changes on this generated individual). This newly generated set is known as the offspring (or children), and constitutes the new population to be evaluated on the next iteration. Each one of this iterations (fitness evaluation, parent selection based on fitness, new population using the offspring generation) is called a generation, and it is repeated until termination when a stopping condition is reached (a given number of generations, for example).

Due to their population operation principle, EAs are a good choice for MOPs as they can generate many elements of the Pareto optimal set in a single run while mathematical programming techniques usually generate one candidate solution per run. In what follows we concentrate on the use of EAs as a method to find the solutions for MOPs.

Many of the MOEAs used to generate the Pareto Front for a MOP implement an on-line Pareto archive which stores the mutually non-dominated solutions produced by the algorithm in a particular generation. Here we concentrate on this kind of MOEA. The pseudo-code in Algorithm 5.1 is an example of a MOEA. A Pareto archive A stores the non-dominated solutions selected from a solution population P produced through an EA for each generation t.

Algorithm 5.1 Archive-based MOEA

1: $t = 0$
2: Generate the initial population P_0 and initial archive A_0
3: $Evaluate(P_t)$
4: $A_{t+1} := Update(P_t, A_t)$
5: $P_{t+1} := Generate(P_t, A_t)$
6: $t = t + 1$
7: If no termination criterion is met, go to step 3

The function $Generate(P_t, A_t)$ produces new solutions from a generation t. This new population P_{t+1} is found through selection, recombination and mutation of the solutions in P_t and the ones archived in A_t. Once P_t is generated, $Evaluate(P_t)$ calculates the fitness value of each solution in P_t (in Evolutionary Algorithms, fitness is a measure of the quality of a solution). The function $Update(P_t, A_t)$ refreshes the Pareto archive estimate A_t with the new solutions, using comparison methods. If the new solution is dominated by a member of A_t, it is discarded. On the other hand, if the new solution dominates one or more solutions in A_t, the dominated solutions are removed form A_t and the new solution is added to A_t.

The updating process of the Pareto archive population with the new best solutions found by the MOEA involves two main operations:

> *Insert*. A new point $\boldsymbol{x}$ is added to the archive. The number of comparison operations is a function of the size $|A|$ of the current archive.
>
> *Delete*. After a new point $\boldsymbol{x}$ is inserted in the current archive which dominates existing solutions, the dominated solutions need to be deleted from the archive. Depending on the data structure chosen to store the archive, a re-organisation of the retained solutions might follow, adding additional operations to *Delete* and a dependency on the number of elements in A.

The operations of comparison, insertion and deletion have to be performed for each element in P_t that is being considered as candidate to be added to the archive A_t, introducing also a dependency on $|P_t|$ of the number of total operations involved in resolving a given MOP through this method.

Algorithm 5.1 is used in many MOEAs with elitism, such as SPEA and NSGA just to name two examples among many [Zitzler et al., 2000].

5.3 Data Structures for MOPs

Data can be defined as the elements on which a computer program operates in order to resolve a problem. Data is input to a program and data is outputted by the program with (hopefully) the solution. Knuth defines data as 'representation in a precise, formalize language of some facts or concepts, often numeric or alphabetic values, to facilitate manipulation by a computational method' [Knuth, 1997].

In a very simple program as the addition of two numbers, for example, the program complexity revolves around the execution of an operation. The treatment of the input data on which the addition operation is to be performed is relatively simple and it is reduced to the problem of determining the type of the (single) data we are dealing with. However as we move beyond a couple of inputs to both a larger and more (type wise) complex set, a new programmatic problem arises.

A data structure is a set of data that includes relations among its objects. Manipulating sets of data with algorithms can result in the reduction, increment or value changes of the original set, reasons for which such sets are called dynamic [Cormen et al., 2001]. In this context, a data structure is a representation of a dynamic set. A typical implementation involves pointers to the elements, allowing their identification, access and manipulation.

The two main group of operations on a dynamic set are queries (return of information about the set) and modifying operations (add, delete or change in any way the set). Typical query operations return pointers to elements that verify the query condition. Using a standard notation we call S the set and k a key value identifying an object of S, common queries are [Cormen et al., 2001]:

- $Search(S, k)$: returns a pointer x to and element of S such that $key[x] = k$ (or NIL if no key found).
- $Minimum(S)/Maximum(S)$: on a totally ordered set, returns a pointer to the element with the smallest/largest key.
- $Successor(S, \boldsymbol{x})/Predecessor(S, \boldsymbol{x})$: on a totally ordered set, returns a pointer to the next larger/smallest element in S (NIL if x is pointing to the maximum/minimum).

Typical modifying operations are $Insert(S, \boldsymbol{x})$ and $Delete(S, \boldsymbol{x})$, which adds or removes the element pointed by x to/from the set.

The time taken for an algorithm to execute an operation on a set is known as time complexity, and it can be calculated by counting the number of operations involved in the algorithm[1]. The time complexity is in general expressed as a function of the size n of the set, and we are particularly interested in its asymptotic behaviour as n increases towards large datasets. Mathematically, the behaviour of a function as it tends towards a limit is described by using the Big O notation, which is used to qualify the performance of a

[1] We briefly introduced this concept in Chapter 2 (Equation (2.1)) where we were concerned with the time to run a program in a particular processor. Here we adopt a higher level of abstraction independent of a particular machine.

particular data structure with its operations [Knuth, 1997]. A red-black tree, for example, has the characteristic that for the worst case scenario operations take $O(\log n)$.

In order of increasing complexity, some elementary data structures used to implement dynamic sets include Stacks and Queues, Linked Lists and Trees. In this dissertation we are particularly interested in Trees and we have implemented a program that maintains a dataset using a quad-tree data structure.

Dynamic Data Structures are a fundamental part of the computational problem solving of an MOP. To find an approximation to the Pareto optimal set we can approximate the Pareto front by comparing the obtained solutions among themselves and choosing those that are non-dominated. We use data structures to store the solutions found for the feasible objective region and by performing operations of query and modification we arrive to the set of solutions that form the Pareto front (or an estimate of it). The dynamic data structure that stores the Pareto front at any given moment in this process is commonly known as a Pareto archive A and the feasible solutions being considered form the population P. As described in Section 5.2, MOEAs are efficient stochastic methods to find solutions for an MOP that might rely heavily on these data structures. Given an initial population, a MOEA generates new solutions through selection, recombination and mutation of the original population, producing a new generation of solutions that forms the population P. The Pareto archive A is formed by storing the initial population and updating it with the new solutions each time a new generation is produced. To decide if a new solution is added to the archive, it needs to be tested for non-dominance, which can result in the solution being discarded or added to the archive. If added, the archive in itself might need to be updated as the new solution might dominate more than one other element of the archive.

There are several factors to consider when looking for an efficient data structure within this context: size of the population, size of the archive, efficiency of queries and insertion operations. Some of the structures used by MOEAs for archive storage include lists, trees [Sun and Steuer, 1996] and more complicated constructions resulting from their combination such as trees for which nodes are lists, for example [Drozdik et al., 2014].

Domination-Free Linear Lists

A linear list is a data structure in which its elements are stored sequentially. Arrays and linked lists are linear lists. Linked lists are useful for cases where the number of elements is not known in advance. Each element of the population is considered for insertion by comparison for non-dominance with all the elements in the archive resulting in either modification of the list by discarding the existing solutions dominated by the candidate and inserting the candidate, or the discard of the candidate if dominated by an element in the archive.

The operations that affect the time complexity of the implementation of a linear list as data structure to maintain the Pareto archive are $Insert$ and $Delete$, which are $O(m \times |P| \times |A|)$

in the case of a problem with m objectives, where $|P|$ and $|A|$ are the sizes of the population P and the archive A respectively. Notice that this time complexity arises from the non-dominance condition that must be verified for each of these operations, not from the operations in themselves, which are $O(1)$ as they only involve the elimination/addition of an element and the re-arrangement of a pointer and do not depend on the size of the list.

5.4 Domination-Free Quad-Trees

A quad-tree is a data structure introduced by Finkel and Bentley [1974] as an efficient solution to store information to be retrieved on composite keys. We can view the solution vectors as keys where every attribute corresponds to each dimension. The solution vectors are inserted in the quad-tree following the order dictated by a successorship relation, where a child is called a j-successor of its parent node and in the case of a problem with m objectives follows

$$j = \sum_{i=1}^{m} j_i 2^{m-i}, \qquad j_i = \begin{cases} 1 & \text{if} \quad x_i \geq y_i \\ 0 & \text{if} \quad x_i < y_i. \end{cases} \tag{5.1}$$

The succersorship relation can be also expressed as a binary vector constructed by comparing the elements according to the rule for j_i in Equation (5.1), and we say $\boldsymbol{x}$ is a j-son of $\boldsymbol{y}$. We also define the following two sets of j-successors:

$$\begin{aligned} S_0(j) &= \{i | j_i = 0, j = (j_1, j_2, ..., j_m)_2\}, \\ S_1(j) &= \{i | j_i = 1, j = (j_1, j_2, ..., j_m)_2\}. \end{aligned} \tag{5.2}$$

When using a quad-tree as an archive to store the estimated Pareto front, successorship is determined according to the non-dominance relation. We can obtain a non-dominated quad-tree by excluding the 0 and 2^m-successors (which relate dominated solutions). Figure 5.1 shows an example for a 3-dimensional objective space, showing the root and its successors. Notice that each son in a quad-tree can be considered the root of its own subtree, with its successors determined respect to its values.

To insert a new element in a (standard) quad-tree, we determine first its succerssorship respect to the root. If a successor for that j-value already exists, then we set the existing j-son as root and we repeat the process. In the case of non-dominated quad-trees, the algorithms for insertion and deletion become more complex. Habenicht has introduced an efficient method to process new solutions for inclusion in the Pareto archive maintaining the quad-tree domination-free, reducing the number of comparisons needed to determine domination providing a criteria to avoid testing all branches [Sun and Steuer, 1996]. We will assume a MOP where optimal solutions are minimums in the objective space.

Let $\boldsymbol{x}$ be a vector we are considering for insertion in a non-dominated quad-tree with root

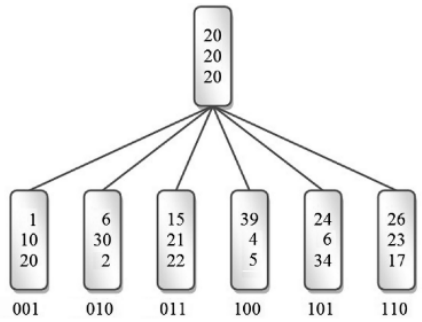

Figure 5.1 An example of a non-dominated quad-tree

$\boldsymbol{y}$, with $\boldsymbol{x}$ a j-son of $\boldsymbol{y}$ and $S_0(j)$, $S_1(j)$ as defined by Equation (5.2), and l the variable to identify the l-son $\boldsymbol{x'}$ in the non-dominated quad-tree with which we compare $\boldsymbol{x}$:

- If there are any vectors in the quad-tree that dominate $\boldsymbol{x}$, they belong to subtrees rooted at l-sons of $\boldsymbol{y}$ such that l have 0s in at least all locations that j does, i.e., $l \leq j$ and $S_0(j) \subset S_0(l)$. For example if $j = 100101 = 37$ (according to Equation (5.1)), any j-son $\boldsymbol{x'}$ with a 1 where j has 0s would mean that the corresponding objective component of $\boldsymbol{x}$ is preferred to $\boldsymbol{x'}$, i.e., it cannot dominate $\boldsymbol{x}$.

- If there are any vectors in the quad-tree dominated by $\boldsymbol{x}$, they belong to subtrees rooted at l-sons of $\boldsymbol{y}$ such that l have 1s in at least all locations that j does, i.e., $l > j$ and $S_1(j) \subset S_1(l)$. Taking the same example as above where $j = 100101 = 37$, any vector with 0 where j has 1s would mean that the corresponding objective component of $\boldsymbol{x'}$ is preferred to the component in $\boldsymbol{x}$, i.e., $\boldsymbol{x}$ cannot dominate $\boldsymbol{x'}$.

(The above considerations will become clearer through the algorithm implementations and examples that follow.)

Sun and Steuer [1996] and later Mostaghim et al. [2002] developed various algorithms to implement the operations needed to maintain non-dominated quad-trees as structures to archive and update the estimated Pareto front when using MOEAs to solve MOPs. Algorithm 5.2 shows Quadtree1 developed by Mostaghim et al. [2002].

To illustrate the operations related to the inclusion of a new element in a non-dominated quad-tree using Algorithm 5.2 we present here an example from Mostaghim and Teich [2005]. Consider the tree in Figure 5.2(a) where we are checking for insertion the vector (4 8 12). According to Equation (5.1), (4 8 12) with respect to the root (10 10 10) has successorship $j = 001$. The nodes that might be dominated by (4 8 12) will belong to the branches rooted at successors 011 and 101. The node (6 16 22) is dominated by (4 8 12) so we delete it, while (3 25 16) is not dominated so we re-insert it, becoming the 011-successor of (10 10 10). Now we check for nodes that might dominate (4 8 12), but in this case there are no successors that belong to $S_0(001)$ such that $l < 001$. We then proceed to insert (4 8 12) as the 001-son. As (10 10 10) has already a 001-son, (5 5 23) becomes the new root with (4 8 12) a 101-successor of this new root. We now check the successors 011 and 110 for domination and we find that (9 8 18) is dominated by (4 8 12) so we delete

Algorithm 5.2 Quad-tree1

Input: $\boldsymbol{x}$ to be inserted into a quad-tree with root $\boldsymbol{y}$
Output: Updated domination-free quad-tree
Step 1. Let $\boldsymbol{y}$ be the root of the tree
Step 2. Calculate j such that $\boldsymbol{x}$ is the j-successor of $\boldsymbol{y}$
if $j = 2^k$ or $x_i = y_i; \forall i \in S_0(j)$ **then**
    /* $\boldsymbol{x}$ is dominated by $\boldsymbol{y}$ */
    Stop
if $j = 0$ **then**
    /* $\boldsymbol{x}$ dominates $\boldsymbol{y}$ */
    Save subtree of $\boldsymbol{y}$
    Delete $\boldsymbol{y}$ and its subtree
    Insert $\boldsymbol{x}$ in the position of $\boldsymbol{y}$
    Stop
Step 3.
for all $\boldsymbol{z}$ such that $\boldsymbol{z}$ is l-son of $\boldsymbol{y}$, $l < j$ and $S_0(j) \subset S_0(l)$ **do**
    TEST1($\boldsymbol{x}$,$\boldsymbol{z}$):
    /* Check if $\boldsymbol{x}$ is dominated by $\boldsymbol{z}$ or son of $\boldsymbol{z}$ */
    Calculate j such that $\boldsymbol{x}$ is the j-successor of $\boldsymbol{z}$
    if $x_i = z_i; \forall i \in S_0(j)$ **then**
        Stop
    for all $\boldsymbol{v}$ such that $\boldsymbol{v}$ is l-son of $\boldsymbol{z}$, $l < j$ and $S_0(j) \subset S_0(l)$ **do**
        TEST1($\boldsymbol{x}$,$\boldsymbol{z}$)
Step 4.
for all $\boldsymbol{z}$ such that $\boldsymbol{z}$ is l-son of $\boldsymbol{y}$, $j < l$ and $S_1(j) \subset S_1(l)$ **do**
    TEST2($\boldsymbol{x}$,$\boldsymbol{z}$):
    /* Check if $\boldsymbol{x}$ dominates $\boldsymbol{z}$ or a son of $\boldsymbol{z}$ */
    Calculate j such that $\boldsymbol{x}$ is the j-successor of $\boldsymbol{z}$
    if $j = 0$ **then**
        Delete $\boldsymbol{z}$ and reinsert all its subtrees from the global root
    for all $\boldsymbol{v}$ such that $\boldsymbol{v}$ is l-son of $\boldsymbol{z}$, $l < j$ and $S_1(j) \subset S_1(l)$ **do**
        TEST2($\boldsymbol{x}$,$\boldsymbol{v}$)
Step 5.
if a j-son of $\boldsymbol{y}$ already exists **then**
    Replace $\boldsymbol{y}$ with the j-son
    Goto Step2
else $\boldsymbol{x}$ is the j-son of $\boldsymbol{y}$
    Stop

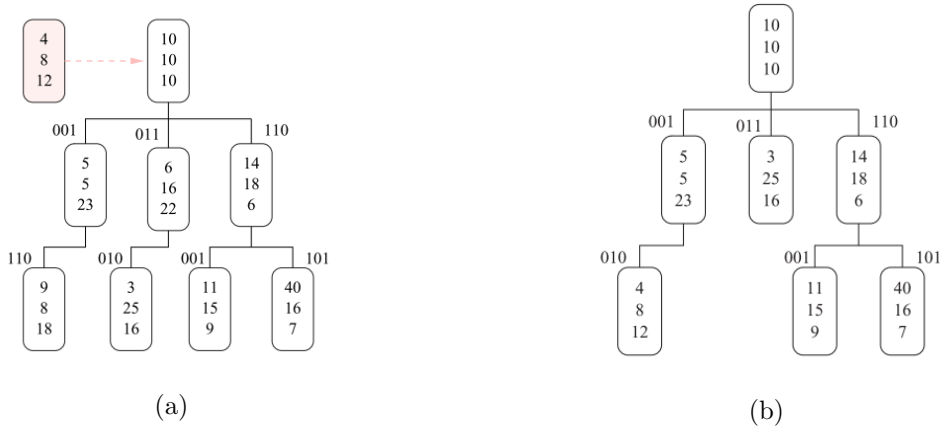

Figure 5.2 (a) Example of insertion of a new solution on a quad-tree (see text) (b) Quad-tree after insertion and deletion of nodes (see text) Mostaghim and Teich [2005]

it, and as there are no other sons, we insert (4 8 12) as the 010-successor, resulting in the tree of Figure 5.2(b).

Mostaghim et al. [2002] also present two improved versions of Algorithm 5.2. Algorithm Quadtree2 uses a more efficient method for the deletion process that involves marking nodes for deletion before re-insertion. Once a node has been identified as dominated by a new vector, instead of deleting and re-inserting its subtree immediately, the node is marked for deletion and the checks continue for the subtree as well as any other nodes found to be dominated. Once the checking procedure has finished and all possible checks have been done, the marked nodes are deleted and the non-marked nodes re-inserted. This saves unnecessary re-insertions of nodes that have not yet been checked against the new element. Another alternative they propose is the algorithm Quadtree3 that solves the deletion problem differently. We developed our quad-tree implementation using the QuadTree1 version.

5.5 Summary

We introduced the multi-objective optimisation problem and the use of data structures for storing and updating the estimated Pareto archive.

The quad-tree is a useful data structure to integrate with evolutionary algorithms to solve multi-objective optimisation problems. We described the algorithm QuadTree1, which we used in our implementation of a Pareto archive.

6 Profiling the Quad-Tree Data Structure with MOEAs: Implementations and Results

The standard procedure to evaluate the performance of a data structure being used as non-dominated archive for solving a MOP (and its archive maintenance algorithms) consists in running a set of appropriately designed test problems, e.g. those of Deb et al. [2002b], over a fixed initial population. Using a MOEA the population is evolved over few generations. To find the most efficient combination of parameters, the time it takes to run the set-up under a set of conditions is measured and then compared to the equivalent experiment using other data structures (or the same data structure with different implementation algorithms). This is the approach taken in Mostaghim et al. [2002], for example, where the implementation of a quad-tree structure with variations of a base algorithm for its operations is compared against a linear list.

We think a more transparent description of the performance of a data structure needs to include additional factors which might be disregarded in the standard and more simplistic approach of comparing computational times.

Following the methodology proposed by Ailamaki et al. [1999] discussed in Section 2.3, updated with the newer micro-architectural models used in performance evaluation described in Section 2.3.4 we implement an analysis to include those broader aspects.

We can classify the two main factors contributing to the performance of a given program running in a particular computer in architecture-independent (analytical) and architecture-dependent. These two classifications can also be seen as levels: the higher analytical level and the underlying architectural (low) level.

We will first identify the various analytical contributions to a particular set up of data structure, maintenance algorithm and MOEA problem. In a similar fashion to CPI stacks we construct a profile of operations and components that affect the implementation of a problem.

6.1 Generalised Performance Stacks for a MOP

We can view the architecturally independent contribution to the performance of a MOP as the aggregate effect of three components:

- Underlying Data Structure
- Algorithms
- Test problem (complexity, size, etc)

Two examples of previous research analysing these contribution factors are Drozdik et al. [2014], who studied their proposed M-Front structure with the generalised differential evolution algorithm GED3, using DTLZ1 as test function; and Mostaghim et al. [2002] who studied the quad-tree structure using the strength pareto evolutionary algorithm SPEA with different test functions. Both look to quantify the efficiency of the data structure when used to store the estimated Pareto archive of the problem in terms of CPU time. In Mostaghim and Teich [2005] the measurements are taken over the problem system all together, after 400 generations and compared to the same implementation with a linear list. The variables measured are population size $|P|$, archive size $|A|$ and number of objectives m. Most of their measurements use variations on the population $|P|$. The total number of incurred operations are also measured. Their main results show CPU time v. archive size and v. population size, for different number of objectives and test functions. Their methodology integrates the structure and algorithms with the evolutionary algorithm. After the evolution in each generation is finished and the archive with the Pareto front is updated, a new population is formed by selecting the points in the archive. Notice that the efficiency of this stage will be dependent on the data structure used to implement the archive. The results of measurements after letting the problem evolve for many generations will include the speed of the MOEA and the speed of access to obtain the new populations at the beginning of each generation.

Drozdik et al. [2014] propose the data structure M-Front and they compare the CPU time to an equivalent implementation using Jensen-Fortin's algorithm [Jensen, 2003]. In this case, there is not a direct comparison between data structures: while Jensen-Fortin's method is a procedure that does not involve the maintenance of an archive, M-Front is a data structure with its maintenance algorithms which is integrated to a MOEA to solve the problem. Notice that from the point of view of possible architectural bottlenecks, the components contributing to lost cycles for both methods might have very different landscapes as result of a utilisation of the architecture geared towards different components. It would be interesting to see the detailed performance profiles of these two approaches and they might shed light into the underlying reasons for their difference in efficiency for different parameters of the problem. A disadvantage of this method as an efficiency evaluation is that the results shown might change significantly when used with different architectures.

In our work we look exclusively at the performance of the quad-tree data structure and the version of the algorithm chosen to maintain the non-dominated archive. Once the new population is generated, we start the performance counters and we stop them after the last individual has been considered for insertion. We also keep a count of the operations performed during that generation and the final size of the archive.

We performed the measurements in two stages. The first stage records the archive sizes for different populations and algorithms during each generation, for 100 runs. The number of operations at each generation are also recorded. The seeds for each run are set to allow reproducibility later.

In the second stage we selected a seed and we measured the time and some front-end and back-end performance counters when using a quad-tree and a single linked list, with different algorithms.

6.2 Results

We set a quad-tree structure with the QuadTree1 algorithm proposed in Mostaghim et al. [2002] (see Section 5.4), and a list structure for the last part of the experiment. Having limited time available we decided to use only the test function DTLZ1 [Deb et al., 2002b] as test problem, as it is widely used in this area of research. Our aim is to better understand the effects of the processor micro-architecture on performance, so we have sacrificed test function diversity in favour of a higher number of efficiency variables and different conditions. We implemented the DTLZ1 problem setting the number of variables to 15 and the number of objectives to 3, settings we maintained for the whole of the experiment.

For each set of measurements we varied the population size (which is also the size of the new population for each generation). Once the initial population is generated, each individual is considered for insertion in the data structure (quad-tree or list). When all the individuals have been either inserted or rejected, we evolve the population using one of three different EAs: a Multiobjective Evolutionary Algorithm Based on Decomposition (MOEA/D), the Non-dominated Sorting Genetic Algorithm II (NSGA-II) and the Improved Harmonic Search Algorithm (IHS) [Biscani and Izzo, 2019]. This process constitutes one generation. The resulting evolved population is then considered for insertion in the existing archive. We set the number of generations to 300.

The program was written in C++. We used the PAGMO library [Biscani and Izzo, 2019] to generate the problem with the test function DTLZ1 and we evolved the population through evolutionary algorithms provided by the interface. We used the PAPI library to measure a group preset events (described later in Section 6.3). The implementation with the data structure quad-tree and the programs used to measure the tree sizes and performance counters can be found at https://github.com/Silvia-prog.

We will start by discussing the results for the quad-tree implemented with MOEA/D for different population sizes, concentrating on the growth of the archive with generations and

how the number of operations involved in maintaining its non-dominance might be affected by its size. We then reproduce these measurements for NSGA-II and IHS. Finally, we present performance counter measurements for the quad-tree with MOEA/D and NSGA-II and we compare them with the equivalent implementation using a non-dominated linear list.

6.2.1 Quad-Tree Growth

For our first set of results and initial examination of the quad-tree we have chosen MOEA/D as our EA. MOEA/D [Zhang and Li, 2007] is an algorithm framework implementing a decomposing of the MOP into scalar optimisation sub-problems which are optimised simultaneously. The objective of each sub-problem is constructed as an aggregation of the individual objectives. The aggregation vectors (which elements are the coefficients of the decomposition) are used to determine the distance between two sub-problems. The set of possible solutions for the sub-problems are evolved simultaneously towards their solutions by using the best solution found so far (for each sub-problem) with its neighbouring sub-problems. [Zhou et al., 2011; Eiben and Smith, 2015].

Figure 6.1 shows the changes in archive size $|A|$ over 300 generations for three different runs, using MOEA/D as the evolutionary algorithm. We performed the measurements for three population sizes $|P|$ of 55, 210 and 1035. Each column corresponds to a different seed. Notice that having a fixed seed does not guarantee that a given population after evolving the first generation will be a subset of an initial population of bigger size started and evolved with the same seed. As new individuals will be generated by combination of the individuals in the previous generation, after the first generation the pool of original individuals will be different for different $|P|$. Comparisons in behaviour depending on $|P|$ need to consider other factors as we will see later. (We do maintain reproducibility of the experiment however for a given $|P|$ as seeds are set for both the initial population and the evolutionary algorithm.)

We observe a clear variation in the growth of the trees across different runs. Figure 6.2 shows the distribution of tree sizes at the end of the 1st, 100th and 300th generations for the same populations as in Figure 6.1, which we constructed by taking 100 different runs in each case. From these measurements we extracted the average tree sizes we show in Figure 6.3. We also show the median (cyan line) and the the first and third quantiles q1 and q2 (dotted lines).

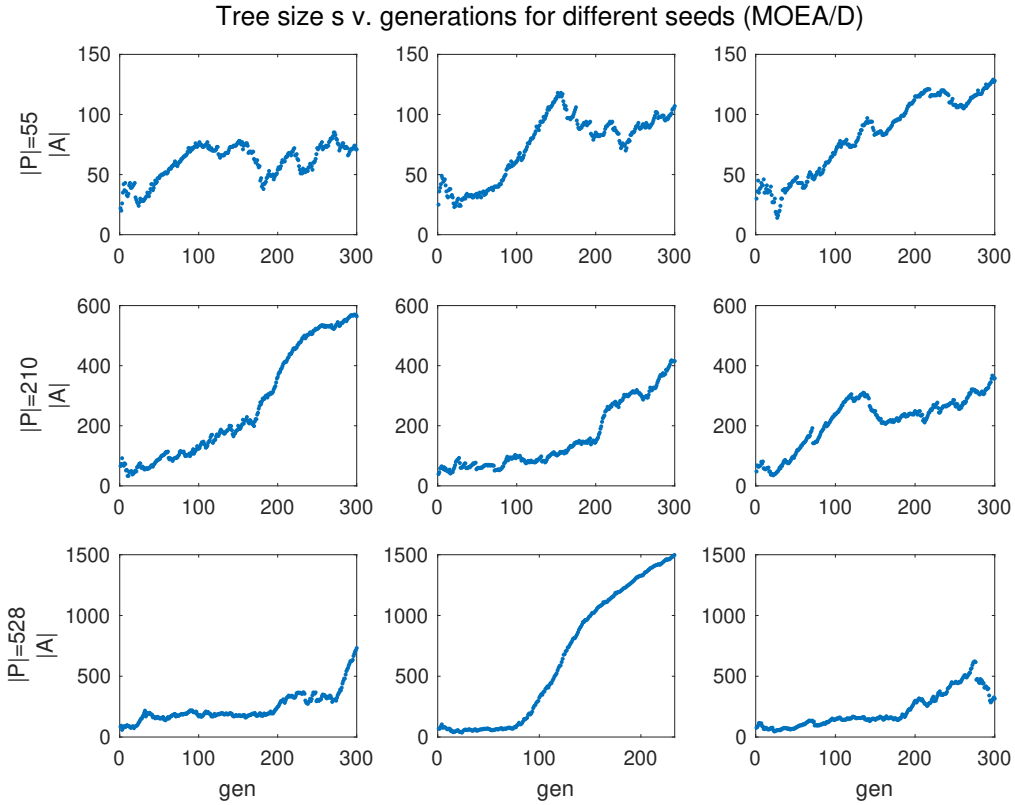

Figure 6.1 Distribution of tree sizes $|A|$ vs. generation, for populations 55, 210 and 1035 (three runs) using MOEA/D to evolve the population. Three different seeds were used, each column corresponds to a different seed

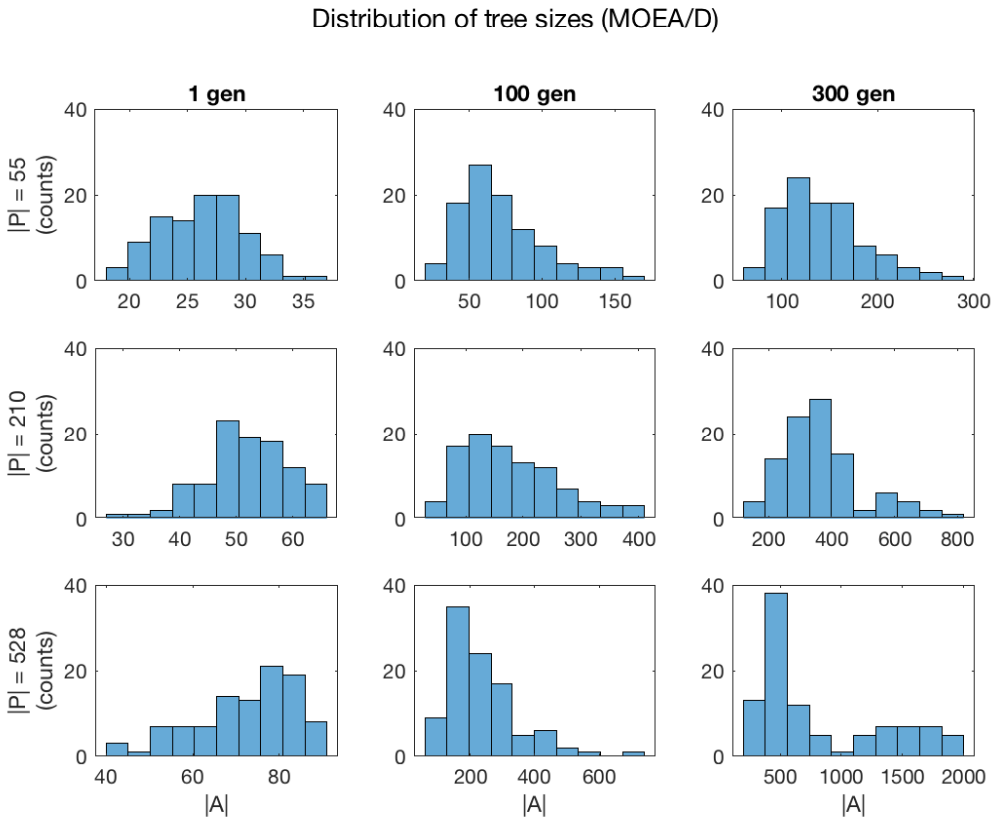

Figure 6.2 Distribution of tree sizes $|A|$ for three different generations measured over 100 seeds using MOEA/D, for three different populations

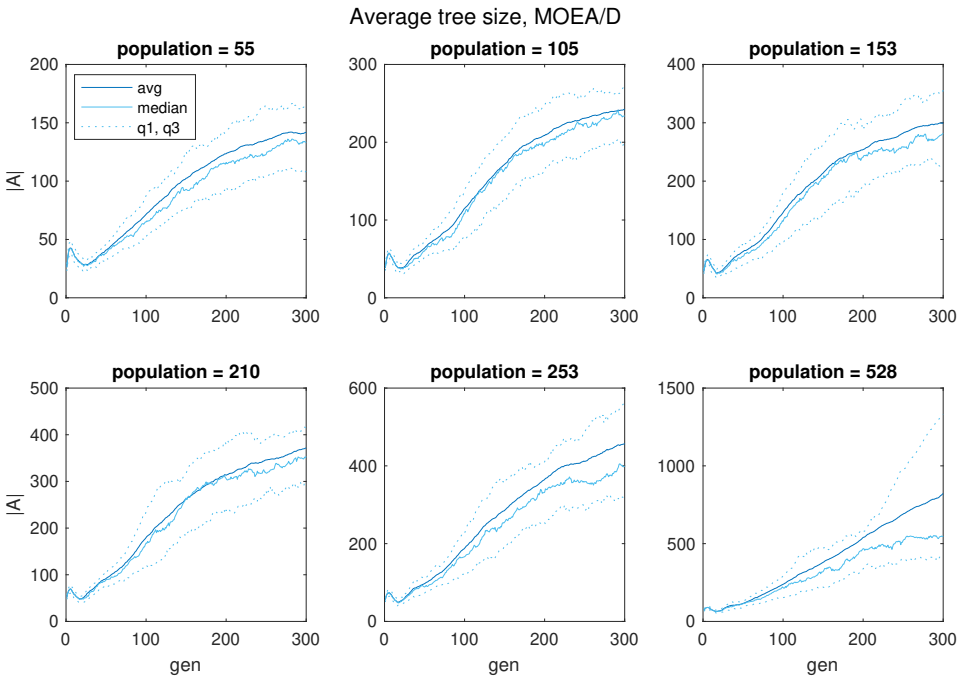

Figure 6.3 Average tree size $|A|$ vs. generation extracted histograms as the ones in Figure 6.2 (100 seeds), for different populations, with MOEA/D. Shown are the median, the first and the third quantiles

For all populations the average tree size shows a slight increase in the first generations, decreasing to a minimum at generation ~ 20 and then continuously increasing for the rest of the generations. At low populations the average rate of growth starts decreasing after ~ 150 generations, and it seems to converge towards a constant. For higher populations the average continues to increase steadily. Notice, however, the median separating from the mean with the median converging towards a constant. This indicates the appearance of trees that reach a fast rate of growth after certain generation. We observed one of these tree profiles for the second seed at $|P| = 528$ in the Figure 6.1. With increasing population and for higher generations, more tree profiles like these one appear, separating the mean from the median (also observed in the distribution of tree sizes after 300 generations in Figure 6.2).

6.2.2 Quad-Tree Operations

In every generation the efficiency of tree update will depend on the population size (the number of times an insertion is considered), the tree size (the number of comparisons that will have to be made) and the amount of dominated and non-dominated solutions generated for that given population. This dependency is reflected through the type and number of operations involved at each generation, which is determined by the choice of algorithm used to maintain the archive.

The implementation of the QuadTree1 algorithm we used in our experiments (described in Algorithm 5.2) involves three main operations. When considering a new individual to add to the archive, the possible outcomes are

- Rejection
- Clean Insertion
- Insertion with deletion of existing nodes.

Rejection

In the case of rejection, the new individual $\mathbf{x}$ is found to be dominated by an existing node. The best case scenario is when $\mathbf{x}$ is dominated by the root of the tree, involving just one comparison. The worst case scenario is when the dominating node is the last one in the check list. Notice that (as explained in Section 5.4) not all branches need to be checked, which could result in an advantage when compared to a list for sufficiently large archive sizes.

Clean Insertion

In the case of clean insertion, the nodes in the tree and x are mutually non-dominated. This operation is more expensive than a rejection: after checking all possible nodes to see if x is dominated (equivalent to the worst case scenario for the rejection operation), nodes in the tree that might be dominated by x need to be checked too.

Insertion with Deletion of Existing Nodes

In this case, one or more nodes in the tree are dominated by x. The worst case scenario is when branches are found with only one dominated node. After cutting off the branch, the dominated node is removed and the rest of the branch is re-inserted. The re-insertion is not as bad as a clean insertion because nodes do not need to be checked for dominance.

If $Nrej$ is the number of rejections, ci the number of clean insertions and id the insertion with deletion, then

$$|P| = Nrej + ci + id. \tag{6.1}$$

Taking into account Equation (6.1), inspection of $ci + id$ for a given population $|P|$ is sufficient to characterise the number of operations for each generation. Figure 6.4 shows clean insertions and insertions with deletion (normalized by population size) as a stacked bar plot for three populations, three runs each, using the MOEA/D. The proportion of rejections respect to the size of the tree is the lowest at earlier generations, around 0.6 for

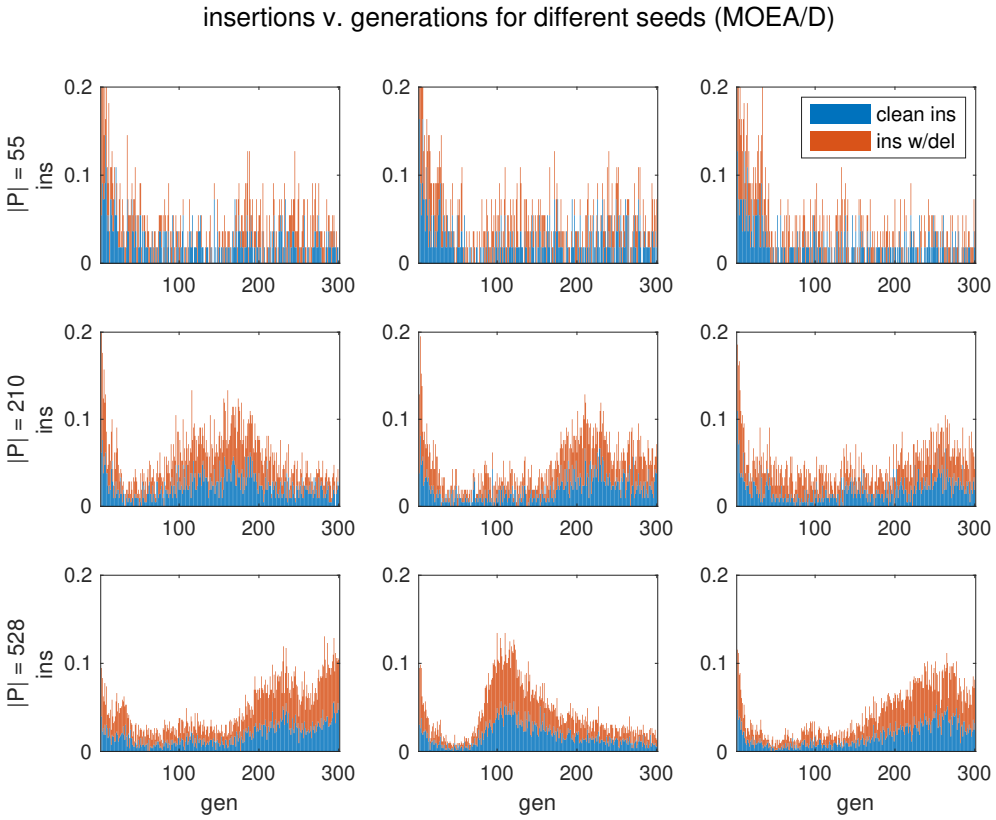

Figure 6.4 Number of insertion operations v. generation for the quad-tree (with MOEA/D) and for populations 55, 210 and 1035, normalised by population size. Rows indicate different populations, columns different seeds. The bars are stacked, the sum of the bars gives the total number of rejections for that generation (normalised)

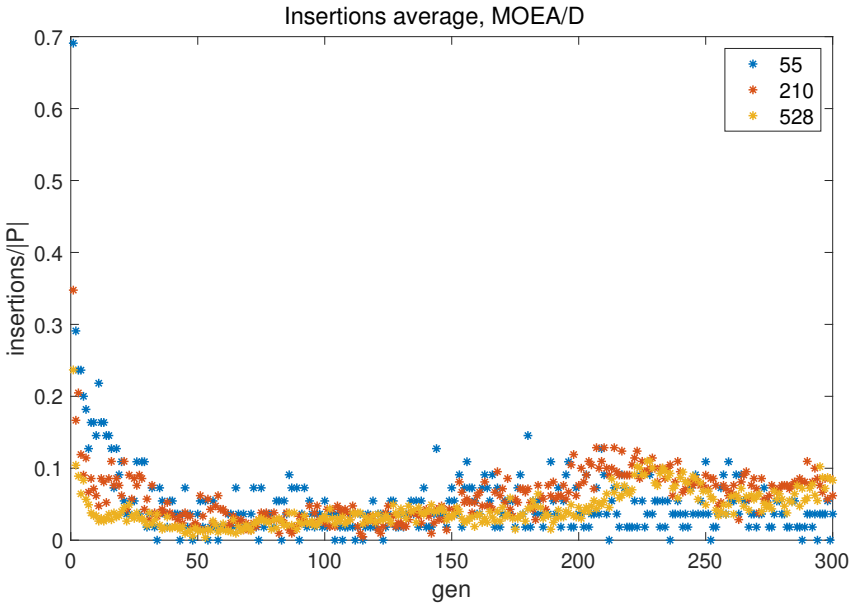

Figure 6.5 Number of insertion operations for the quad-tree (with MOEA/D) normalised by population size, for three different populations 55, 210 and 528

$|P| = 55$, 0.4 for $|P| = 210$ and 0.2 for $|P| = 1035$. For the rest of the generations the level of operations are similar among all runs, staying below 0.2.

The difference in level of operations for low generations makes it difficult to compare among graphs in Figure 6.4. Figure 6.5 shows a scatter plot for the number of insertions (normalised) for the three populations as in Figure 6.4 to illustrate this point. We have separated these two regions in the measurements that follow. We will first discuss the results excluding lower generations.

Average Operations for Generations > 10

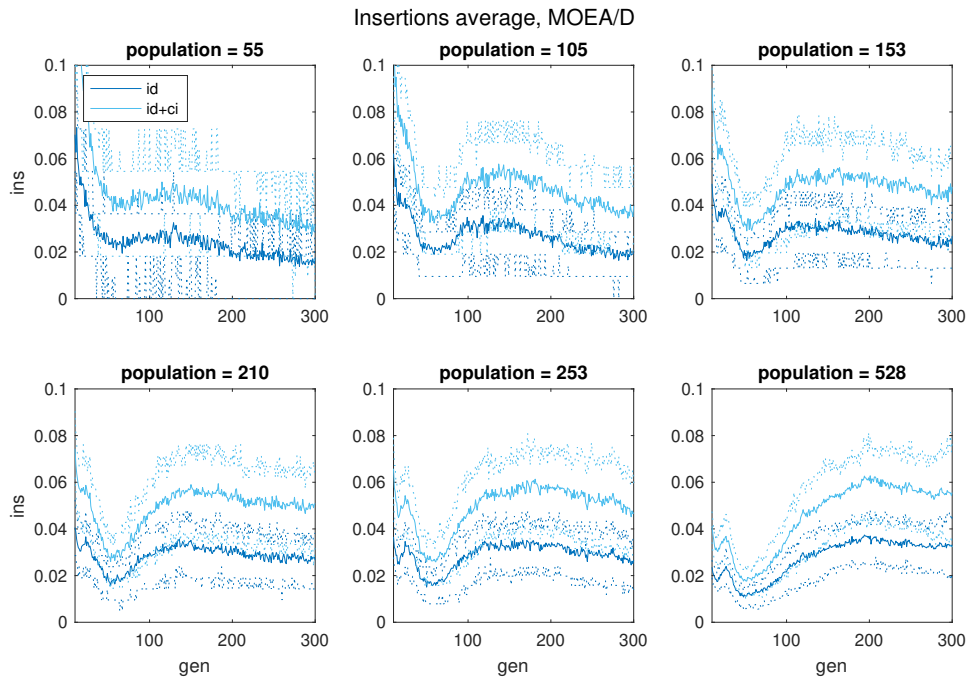

Figure 6.6 Averages for the insertion operations taken over 100 seeds for different populations. The results for the first 10 generations are presented elsewhere, Figure 6.7. The cyan lines correspond to the first and third quantiles for $id + ci$

Figure 6.6 shows the operation averages over 100 runs for different populations after the 10th generation. The dark blue line shows the mean number of instructions with deletions and the cyan line the total inclusions $id + ic$, dotted lines mark their quantiles. All results have been normalised by population size. For low populations the average insertions are maximum for the low generations, with lower values the end of the generation measuring range. For larger populations the insertions at low generation decrease, and the insertions for higher generations increase with its values becoming maximum in this zone. At generation ~ 50 a minimum appears for $|P| \simeq 105$, its value decreasing slightly with $|P|$.

Average Operations for Generations ≤ 10

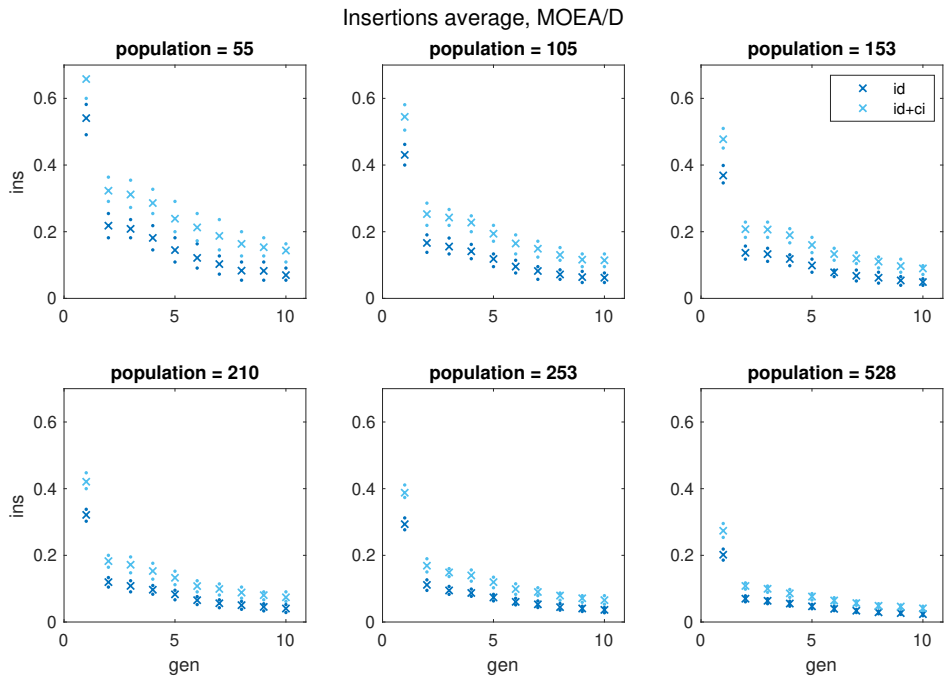

Figure 6.7 Averages for the insertion operations taken over 100 seeds for different populations during the first 10 generations

Figure 6.7 shows the first 10 generations that were excluded in the previous Figure 6.6. The yellow stars are the mean for insertion with deletion and the blue starts the mean for the total insertion number. The small cyan dots indicate the first and third quantiles. The number of total insertions in the first generation decreases with $|P|$ from ~ 0.6 for $|P| = 55$ to ~ 0.3 for $|P| = 528$, with a relative decrease with population size for the rest of the generations range. We will discuss the results in terms of the number of rejections (i.e., $1 - (ci + id)/|P|$) for clarity, i.e., for a given generation we observe a higher number of rejections as population increases. This would be expected if the algorithm generating the population tends to produce points within the current Pareto front such that more generated points in one generation will result in a higher precision of the front. For a given population, the observed average increase in the proportion of rejected individuals (or decrease of total insertions) will indicate a slow advancement of the Pareto front in favour of an increase of individuals belonging to the front stored in the archive.

The characteristics of the advancement of the Pareto front might depend on the test function (DTLZ1) and also on the chosen evolutionary algorithm (MOEA/D). Figure 6.8 shows the mean $|A|$ as a function of the total number of function evaluations for each population and generation (in linear and logarithmic scales). At each generation *gen* the test function is evaluated $|P|$ times, giving a total number of evaluations $|P| \times gen$. The average tree size in Figure 6.8 seems to scale well with the number of evaluations for populations up to $|P| = 253$, indicating that the front first grows in precision and advances slowly. A bigger population will grow a larger tree, with lower populations needing more generations to get to the same tree size.

The case for $|P| = 528$ separates from the rest, but seems to converge towards the expected growth behaviour extrapolated from the graph after number of evaluations reaches $10e4$. This could indicate a threshold for $|P|$ above which there is a higher proportion of individuals generated that belong to a new front, growing the tree at a slower rate..

Notice we are taking averages over 100 seeds, a higher number of samples are needed to confirm these conclusions and explore these features further, and will be considered in future work.

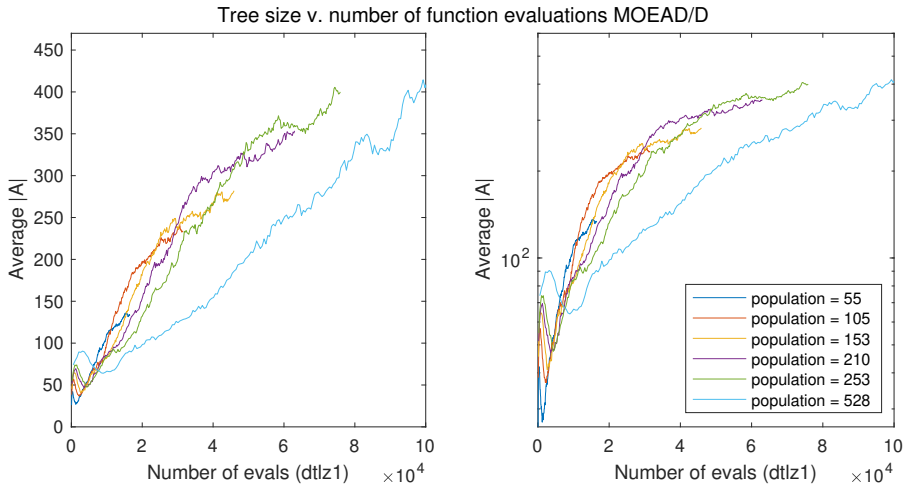

Figure 6.8 Average tree size for different populations as a function of the number of function evaluations in linear and logarithmic scales

6.2.3 Results Using Different Algorithms to Evolve the Population

The results of Section 6.2.2 indicate a possible dependency on the choice of test function and the algorithm used to produce and evolve the population. In this section we show results obtained using DTLZ1 with two more algorithms: the Non Dominated Sorting Genetic Algorithm II (NSGA-II) and Improved Harmony Search (IHS).

NSGA-II [Deb et al., 2002a] is a popular MOEA that uses an elitist principle for the selection of individuals in the population to be evolved, preserves diversity by an explicit mechanism and emphasises non-dominated solutions [Deb, 2001]. For a generation, the offspring population is created from the N individuals of the parent population. A new population (of size $2N$) is formed from these two groups and classified into non-domination fronts (non-dominated sorting). The N available slots for the new population are filled by taking points of the different fronts, one at a time until the population size has been reached. At this point the not considered fronts are deleted. In the case of the last considered front, if it has more points than slots available, members with the higher diversity are chosen. Depending on the size of the solution sets and their distance to the solution boundaries a crowding distance is introduced. This tournament selection is based on ranking and distance such that solutions with a better rank are chosen first, if the rank is the same then the one with highest crowding distance is chosen [Yang, 2014].

Li and Qingfu Zhang [2009] have shown that in the case of complicated shapes of the Pareto set, MOEA/D outperforms NSGA-II. As a possible explanation they conjecture that one of the reasons could be that the differences in selection mechanisms of the two frameworks. The selection mechanism in NSGA-II lacks control of the distribution of the computational effort over different ranges of the PF, while the MOEA/D mechanism allows to distribute the computational effort evenly among the single objective optimisation sub-problems. While a more thorough analysis of the particular mechanisms of each algorithm and their effects in performance are beyond the scope of our work, we would like to point out that in general, when comparing these two frameworks, we expect MOEA/D to perform better.

IHS [Mahdavi et al., 2007] is an improved version of the Harmony Search (HS) algorithm [Zong Woo Geem et al., 2001]. IHS starts by randomly generating a group of N solutions, this is the initial population. The population at the next generation is replaced by N new solutions which are each formed by using the totality of the parent population (as opposed to 2 parents as in other EAs). To generate a new solution, a member from the parent set is chosen randomly with a probability P_{HM}. Once the solution is chosen it might be modified by varying its original value by a random amount. The rate at which a member will be modified is set by the probability P_{PA}. The probabilities P_{HM} and P_{PA} are parameters that help the algorithm improved solutions, and vary at each generation (P_{HM} linearly and P_{PA} exponentially). The new individual replaces the current worst individual if it has better fitness.

Figure 6.9 shows tree sizes v. generations for three populations each with three runs, using MOEA/D, NSGA-II and IHS. In the case of the implementation with NSGA-II the population number must be a multiple of 4 (see Biscani et al. [2010]). At the time of

measuring with NSGA-II the measurements and their processing for the implementation with MOEA/D had finalised. We decided to use for NSGA-II populations multiples of 4 closest to the ones we had chosen for MOEA/D. For clarity we have grouped the results for NSGA-II with their closest corresponding populations: a NSGA-II populations of 56, 208 and 528 have been grouped with the MOEA/D-IHS populations of 55, 210 and 528. When analysing NSGA-II results on their own we will use the exact population numbers (see later subsections NSGA-II in this and following sections).

NSGA-II shows a similar behaviour to MOEA/D for generations up to 100 $\sim$ 200, then deviating from each other, NSGA-II showing stronger growth. IHS, on the other hand, either remains stable or decreases with generations. We first examine the case for NSGA-II. We only present preliminary results for IHS.

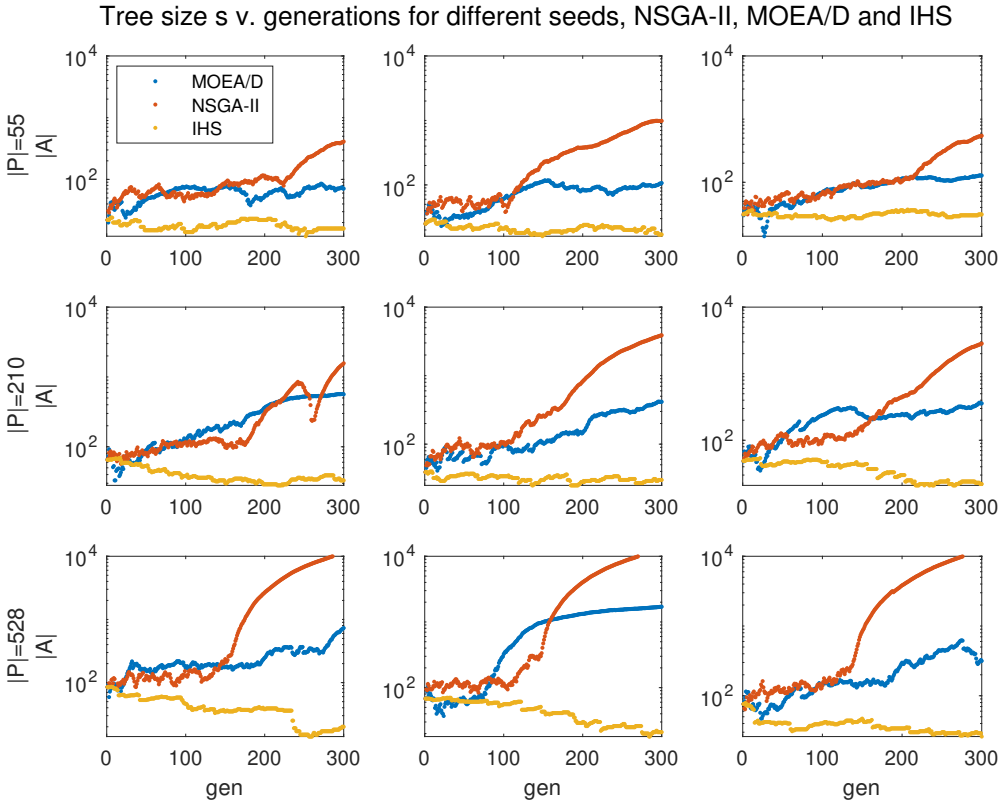

Figure 6.9 Tree size $|A|$ vs. generations, three different seeds and three populations, when using different algorithms to evolve the population. Measurements for NSGA-II were performed for populations 56, 208 and 528; results have been grouped with their closest population sizes as listed in the y-axis (see text)

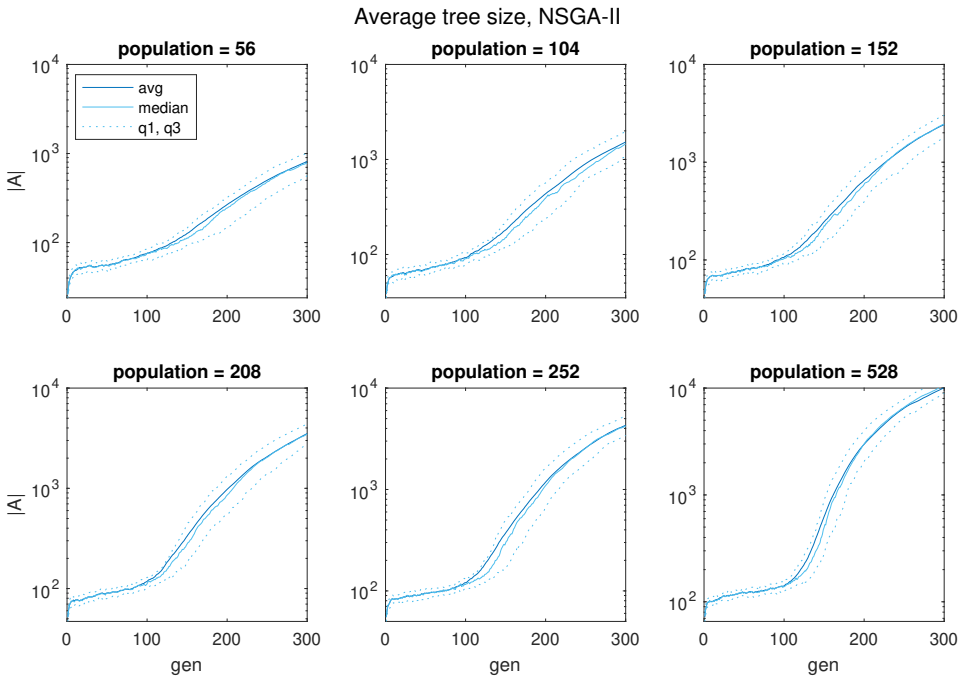

Figure 6.10 Tree size $|A|$ vs. generation averaged over 100 seeds, for different populations, with NSGA-II. Shown are the average, median and first and the third quantiles

NSGA-II

Figure 6.10 shows the mean the tree size $|A|$ v. generation using NSGA-II to evolve the population, in logarithmic scale for clarity. We see two regions with different rates of growth. For lower generations the rate of growth seems constant up to generation ~ 120 when it increases. As the population size grows an area for higher generations with a lower rate of growth starts to appear.

Figure 6.11 shows the mean insertions id and $id + ci$ with quantiles for the latter, for generations > 10. The behaviour of the mean is similar for all population sizes. At around generation 50 a minimum is reached for all populations, with its value decreasing and broadening with $|P|$. For $|P| = 55$ the minimum is ~ 0.2 and it extends over a range of around 10 generations centred at generation 50. For $|P| = 528$ the minimum has decreased to ~ 0.1, extending over the generation range 10-100. After the minimum is traversed, the average number of total insertions increases tending to a steady level towards the end of the evolution period (300th generation), reaching it sooner for higher populations.

Figure 6.12 shows the average insertions for the first 10 generations. As observed with MOEA/D (Figure 6.7) the value for the first generation is highly dependent on $|P|$, decreasing from ~ 0.7 for $|P| = 55$ to ~ 0.3 for $|P| = 528$ and decreasing with slower rate for the rest of the generational range. The relative average insertions lowers with increasing population.

Figure 6.13 shows the average size of the tree as a function of the total number of function evaluations for each choice of population, in linear and logarithmic scales. At difference from the same plot for MOEA/D (Figure 6.8) the populations do not scale. For a fixed

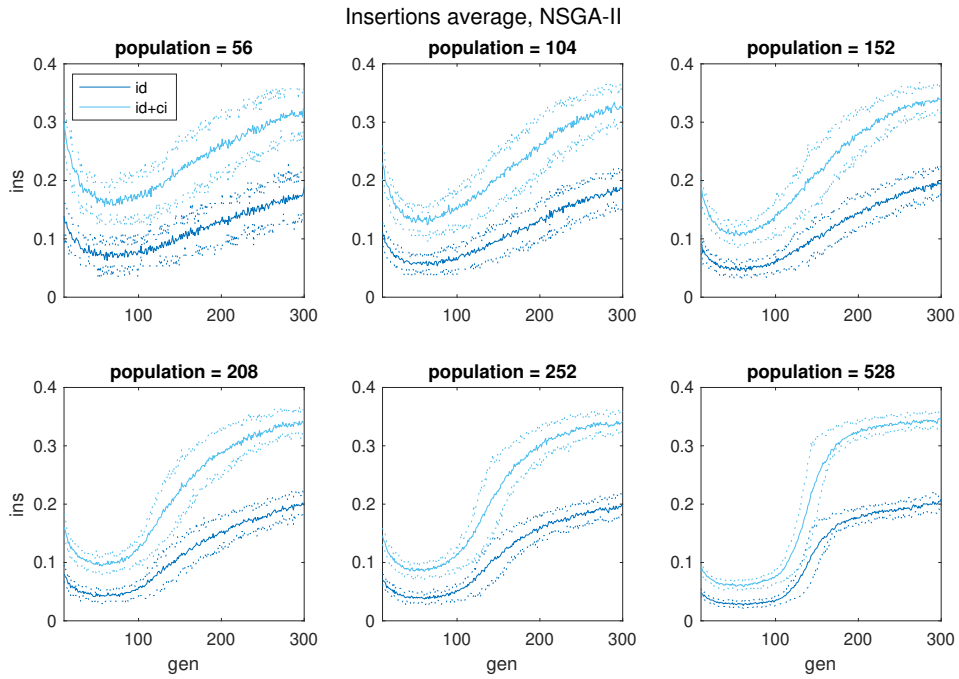

Figure 6.11 Average number of insertion operations v. generation for different populations, over 100 runs, using NSGA-II. The cyan lines correspond to the first and third quantiles for $id + ci$

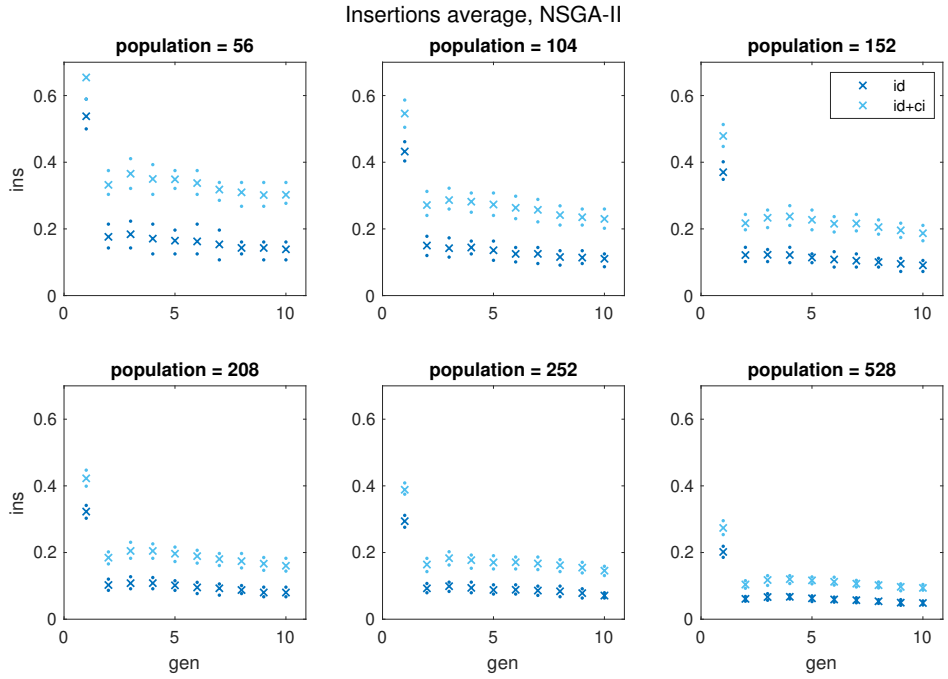

Figure 6.12 Average number of insertion operations v. generation for different populations, over 100 runs, using NSGA-II. The cyan lines correspond to the first and third quantiles for $id + ci$

number of function evaluations at the low end of the scale the average tree size is larger for growing $|P|$. This dependency is inverted as function evaluations increase and the curves cross each other. After the point of crossing for a fixed number of evaluations a larger average tree size is reached with lower populations. This behaviour could indicate that as the average tree size reached by lower populations for the same number of evaluations is larger, each new evolved generation adds to the precision of the front rather than to its advancement. The opposite behaviour might occur in the region of low number of function evaluations.

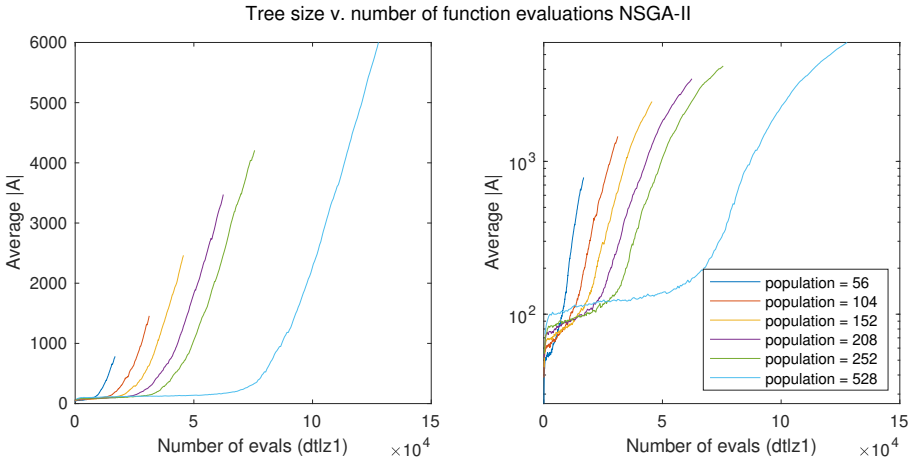

Figure 6.13 Average tree size for different populations as a function of the number of function evaluations in linear and logarithmic scales

Measurements for higher number of evaluations are needed to study the behaviour for large $|A|$ where the plots for the different populations might join as indicated from their extrapolation.

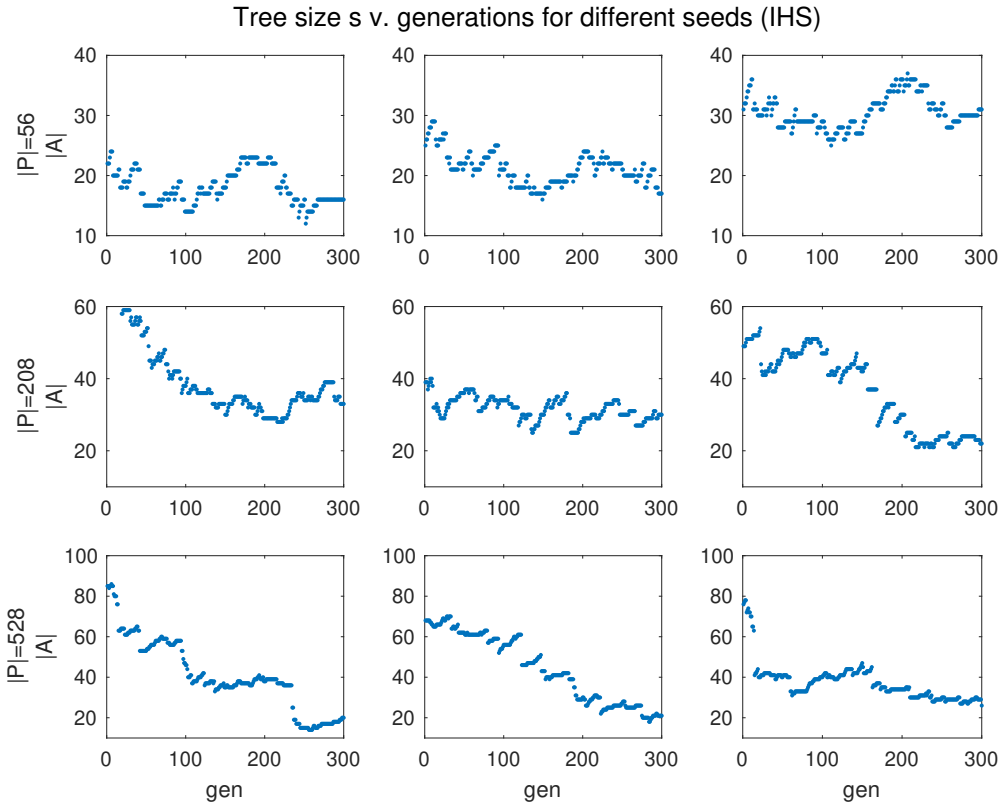

Figure 6.14 Distribution of tree sizes $|A|$ vs. generations when using different seeds with using IHS to evolve the population

IHS

Unfortunately, due to time constraints we only present initial results for the implementation using DTLZ1 and algorithm IHS. Figure 6.14 shows the tree sizes for three runs with different populations. For lower populations the tree sizes seem to remain within the observed range. For higher $|P|$ the tree sizes tend to decrease as the population evolves, indicating possibly a fast advancement of the Pareto front.

A possible explanation for the decreasing tree sizes might be that for large populations a small percentage of points are generated which slight increase in precision towards the Pareto front. This increase in resolution is in detriment of a wider coverage of the front. A more careful analysis of this algorithm together with more measurements is needed to gain a better understanding of these results.

6.3 Performance Counter Measurements

We measured the following performance counters using PAPI :

- TIME_US: total real time in microseconds (wall clock time)
- TOT_INS: Total instructions executed
- TOT_CYC: Total cycles
- L1_ICM: Level 1 instruction cache misses
- L2_ICM: Level 2 instruction cache misses
- TLB_IM: Instruction translation lookaside buffer misses
- BR_MSP: Conditional branch instructions mispredicted
- L1_DCM: Level 1 data cache misses
- L2_DCM: Level 2 data cache misses
- TLB_DM: Data translation lookaside buffer misses

These correspond to the identified metrics in Section 2.3.1. Having the limitation of 8 performance counters available in our system, we performed the measurements in two sets. In the first set we measured L1_ICM, L2_ICM, TLB_IM and BR_MSP, in the second set L1_DCM, L2_DCM and TLB_DM. We measured TIME_US, TOT_INS and TOT_CYC for both sets to cross reference the measurements for each and to ensure we are within the same performance threshold.

Ideally we would measure the metrics using the Top-Down analysis method proposed by Yasin [2014] (see Section 2.3.5), which gives a complete account of events and helps identify the most important contributions to performance. However due to the high amount of performance counters that need to be measured under this model, and the limited time we had to perform the measurements we decided to only use the counters discussed under the naive approach. We consider these results preliminary, future work will include all counters and will use a top-down analysis approach.

To measure performance counters accurately it is necessary to take the average readings over various samples. Due to limitations in time we present here preliminary results where we restricted ourselves to just one run for 3 populations (i.e, one of the columns in Figure 6.9) and two algorithms. To ensure reproducibility we kept the seeds used to generate the populations and evolve them fixed. One sample involves running the program for 300 generations and reading the performance counters at the end of each one 100 times. We did this for the three populations measured in Section 6.2.1 and the two evolutionary algorithms MOEA/D and NSGA-II. We regret we lacked the time to run the same experiments for IHS, which is very unfortunate as IHS presents differentiable behaviour in the three size growth pattern compared to the other two algorithms.

6.3.1 Overhead Measurements

To measure the overheads, we run the program as set for the different measurements to be taken and eliminating the insertion operation. We are still declaring an object Qtree that will be allocated in memory but we are not growing it. We are also generating and evolving the population (but not adding the individuals to the tree). We expect the overhead to be generation and seed independent. The population size might affect the overhead as it will take more instructions to generate the the population and the amount of space in memory taken to store it will be different. While there are no instructions being measured between the start and end of the counters, when measuring with PAPI we don't serialise the instructions nor fence the memory, and instructions coming from surrounding code will be still in the pipeline when taking the measurements.

In the figures that follow we group the results for performance counters in two sets. The first set contains the counters TIME_US, TOT_INS, L1_ICM, L2_ICM and TLB_IM; the second set TOT_CYC, BR_MSP, L1_DCM, L2_DCM and TLB_DM. Presenting groups in this way will allow us to show the performance counter results for each of the two algorithms side by side.

Regarding the sizes we have chosen for the populations the labels in the figures indicate sizes for MOEA/D, and in brackets for NSGA-II. The discrepancies in population sizes for the first two are due to the constrains in the implementation of the algorithms (see Section 6.2.3).

Figure 6.15 and Figure 6.16 show the measured performance counter overheads for ten runs using MOEA/D and populations of 55, 210 and 528 (first column) and NSGA-II with populations of 56, 208 and 528 (second column), as a function of generation. As expected there is no dependency in generations and a dependency on population size is clearly present.

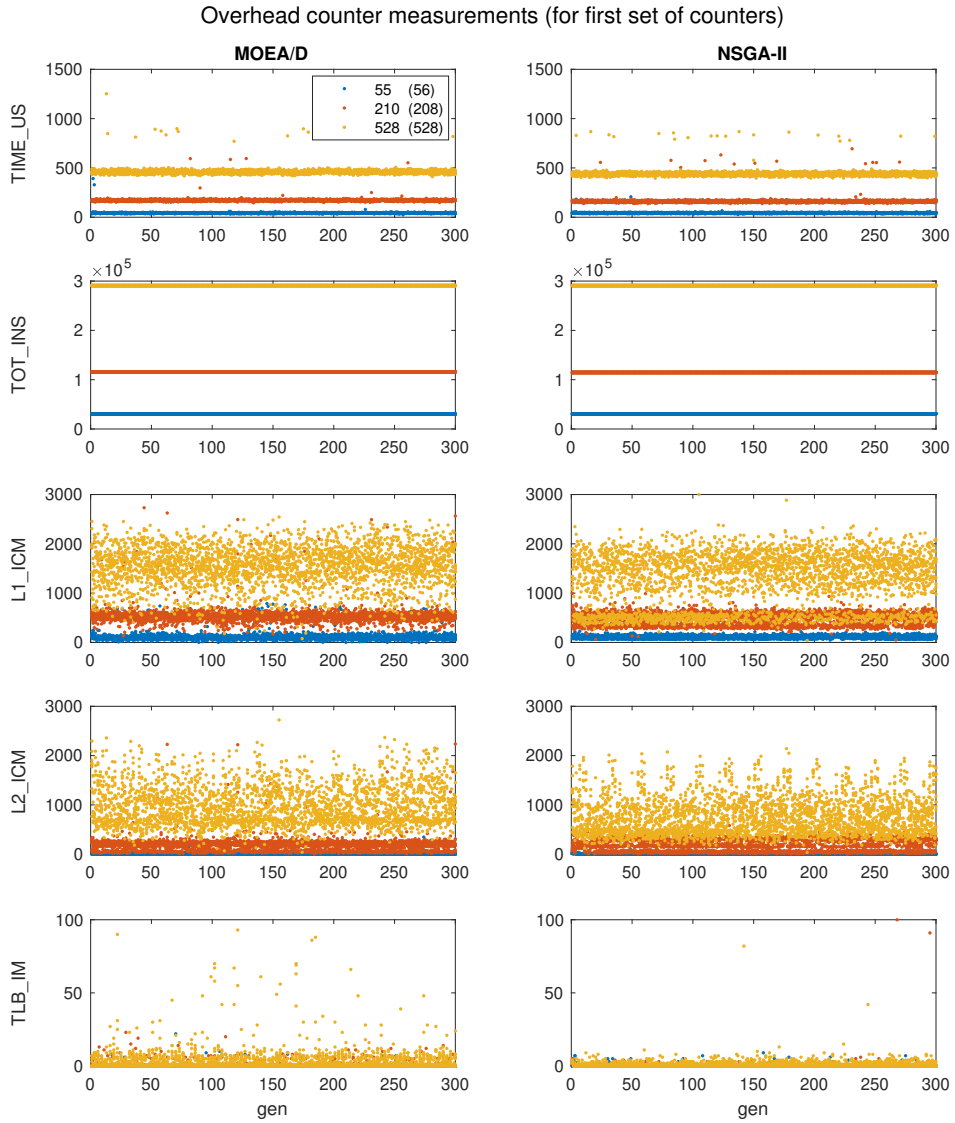

Figure 6.15 Overhead measurement for a first set of counters, taken over many samples and three populations. The first column shows measurements for MOEA/D, the second for NSGA-II. The legend indicates the populations for MOEA/D and (in brackets) for NSGA-II

For NSGA-II, notice the un-normalised L2 misses for instructions and data show a regular feature appearing with generations not observed when using MOEA/D. While more investigations into the specifics of L2 cache misses is needed, this could be due to the lower computational complexity at each generation of MOEA/D respect to NSGA-II [Zhang and Li, 2007]. This could be producing differences in the memory maps for each algorithm to the point of introducing generational dependencies in the L2 cache misses. By showing

a stronger cyclical dependency with growing generations, NSGA-II might be triggering optimisations not evidenced in the results for MOEA/D.

The absolute levels for the different counter reads are similar. From the number of instructions we see the differences in the implementation of the two algorithms are not introducing dependencies when producing and evolving the population.

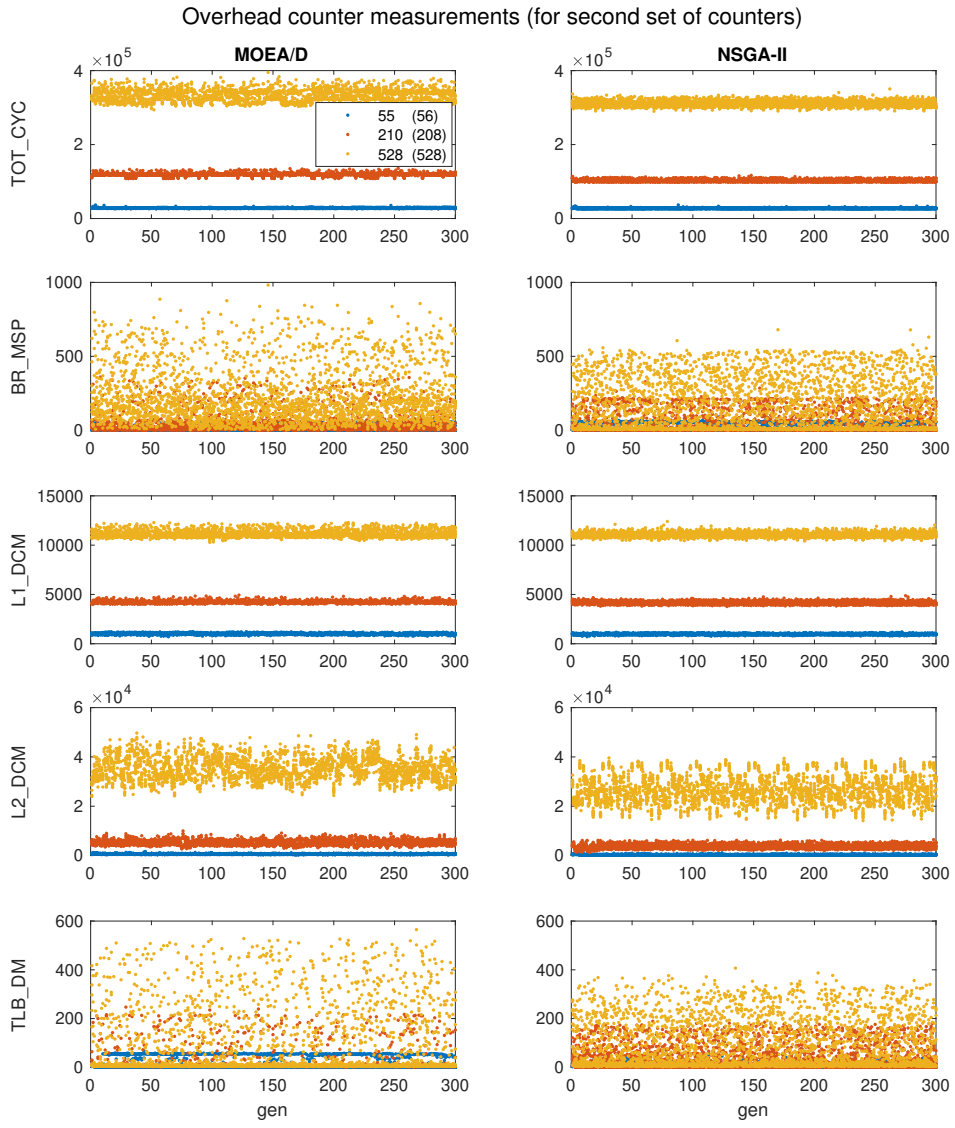

Figure 6.16 Overhead measurement for a second set of counters, taken over many samples and three populations. The first column shows measurements for MOEA/D, the second for NSGA-II. The legend indicates the populations for MOEA/D and (in brackets) for NSGA-II

Figure 6.17 and Figure 6.18 show the results for the two sets of performance counters normalised by population size $|P|$ and in the case of L2_DCM and L2_ICM by $|P|^2$, for MOEA/D (first column) and NSGA-II (second column). The overheads seem to scale well with population size, with some counters showing a better scaling than others. The number of total instructions TOT_INS take the same average value after scaling, giving a clear linear dependency on the population size. TOT_CYC still shows differences after scaling, suggesting lost cycles coming from other sources.

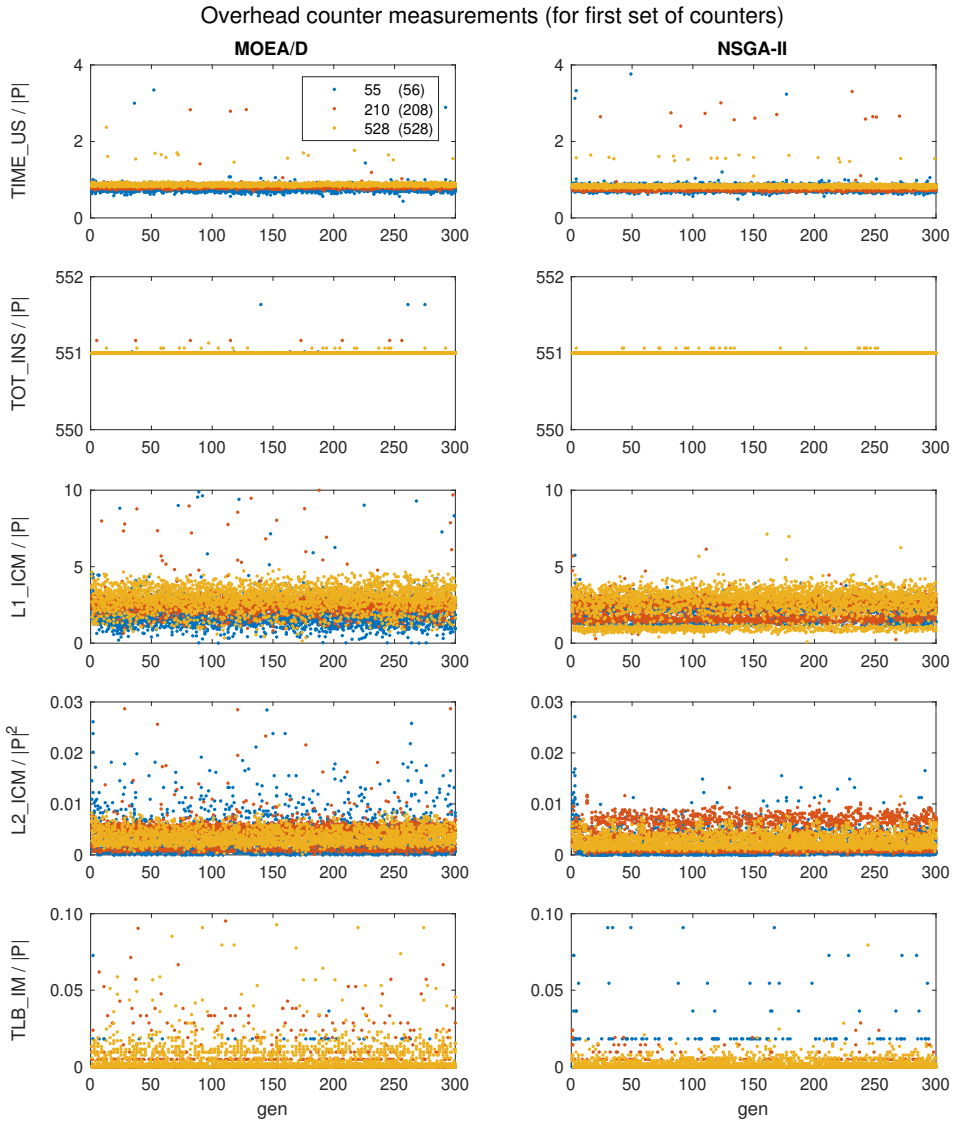

Figure 6.17 Overheads normalised by population size, fist set of counters. The first column shows results for MOEA/D, the second for NSGA-II. The legend indicates the populations for MOEA/D and (in brackets) for NSGA-II

Notice the readings for L2_DCM and L1_DCM scale with $|P|^2$, i.e., for each cache miss in L1 we have $|P|$ cache misses in L2. This could be the result of the mechanisms used to evolve the population by PAGMO together with the traversal of the population by the program accessing the individual to insert them into the tree (while we are not performing

the insertion here, we are still reading each data point). Independently of the particular mechanism causing this dependency, notice that when present these events will introduce a non-linear population dependent contribution to the lost cycles with the important implication that measurements for the population dependent performance of data structures might have components that change across implementations that use the computer resources differently.

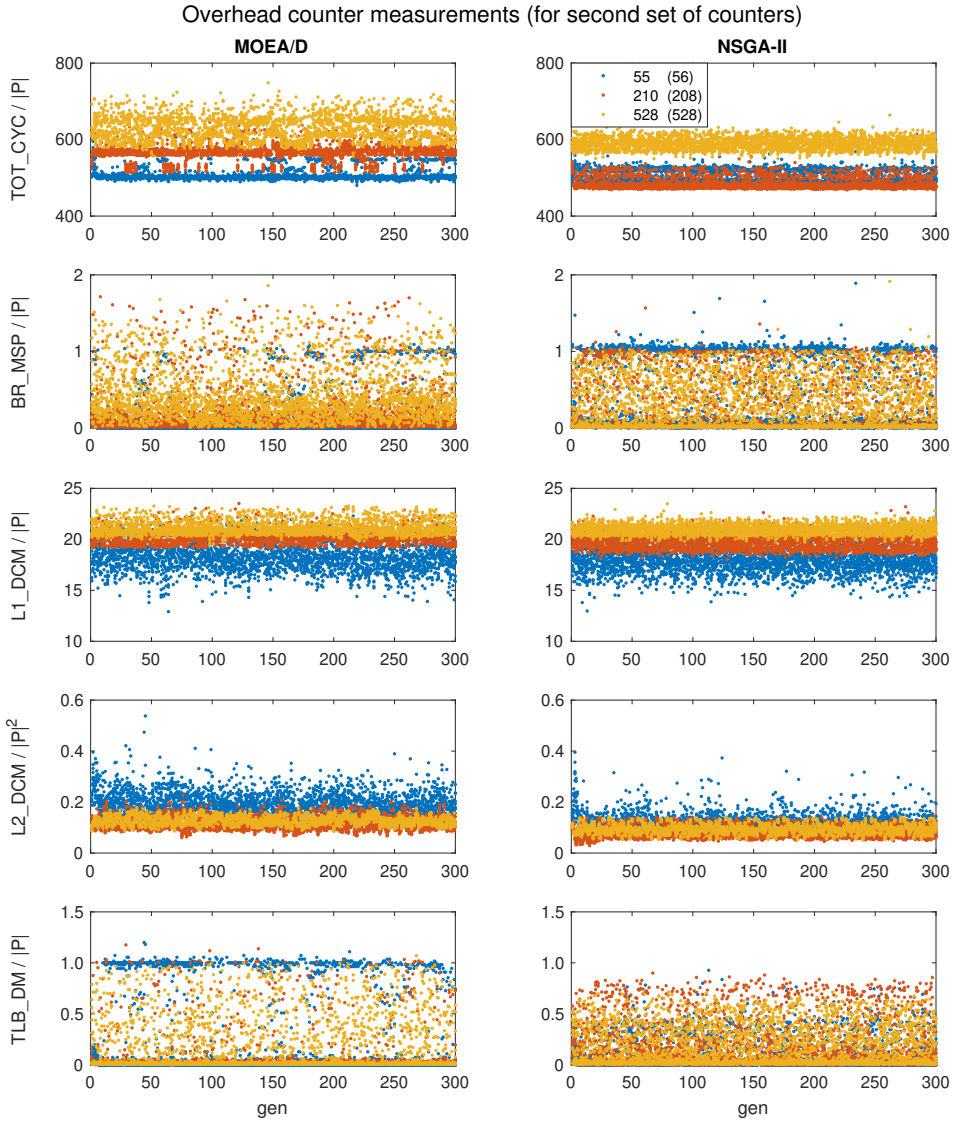

Figure 6.18 Overheads normalised by population size, second set of counters. The first column shows results for MOEA/D, the second for NSGA-II. The legend indicates the populations for MOEA/D and (in brackets) for NSGA-II

6.3.2 Counter Measurements

We measure our chosen set of performance counters (Section 6.3) for three populations, two algorithms and two data structures to store the Pareto archive (quad-tree and list). Due to limitations in time we present here the results obtained from averaging 100 runs for one choice of seed for each experimental configuration. Future work will include more seeds.

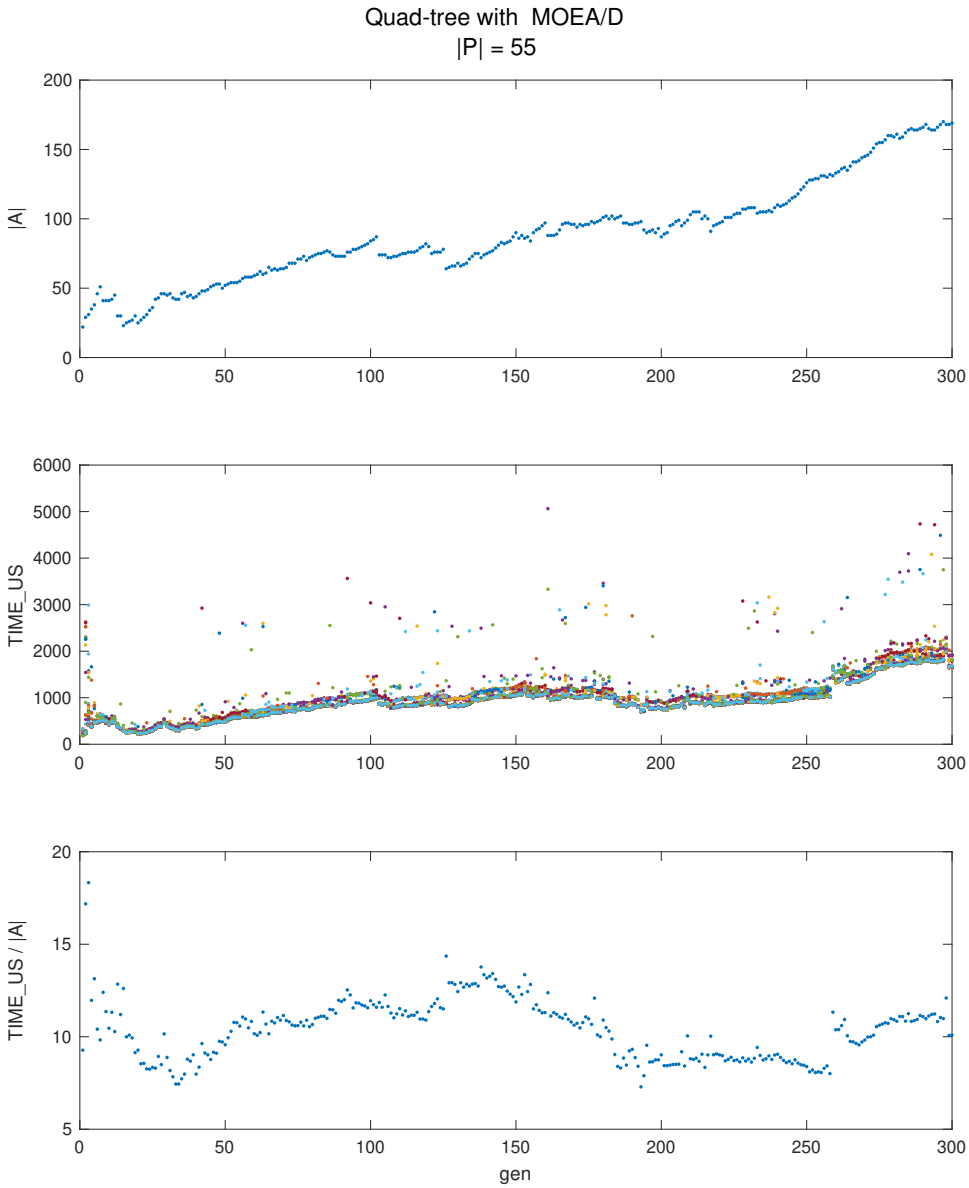

Figure 6.19 Measurements used to calculate the average performance counter readings normalised by $|A|$ for a quad-tree data structure (with MOEA/D) and $|P| = 55$, as a function of generation. With one seed set, we measure $|A|$ over 300 generations (top). We then take 100 samples of TIME_US for all the generations (middle, each sample is represented by a different colour, overhead has been subtracted). Finally we obtain the normalised average (bottom) by dividing the mean (over the 100 samples) of TIME_US by $|A|$

Figure 6.19 shows how we process the performance counter measurements for one of the

experimental set-ups. We expect the readings to be dependent on the archive size $|A|$ as the insertion algorithm for the tree and list will be dependent on tree size (see Section 6.2.2 and Section 5.3). Each of the performance counter samples is divided by the corresponding tree size for that generation. The mean over all the samples (different colours in Figure 6.19, middle) is then calculated, obtaining the average performance count normalised by $|A|$. In the initial processing we noticed the subtraction of overheads do not add perceptible differences to the raw measurements. We present here results without overhead correction.

Figure 6.20 shows the tree sizes v. generation for the seeds we chose at each population and each algorithm. These were used to normalised the corresponding counter measurements.

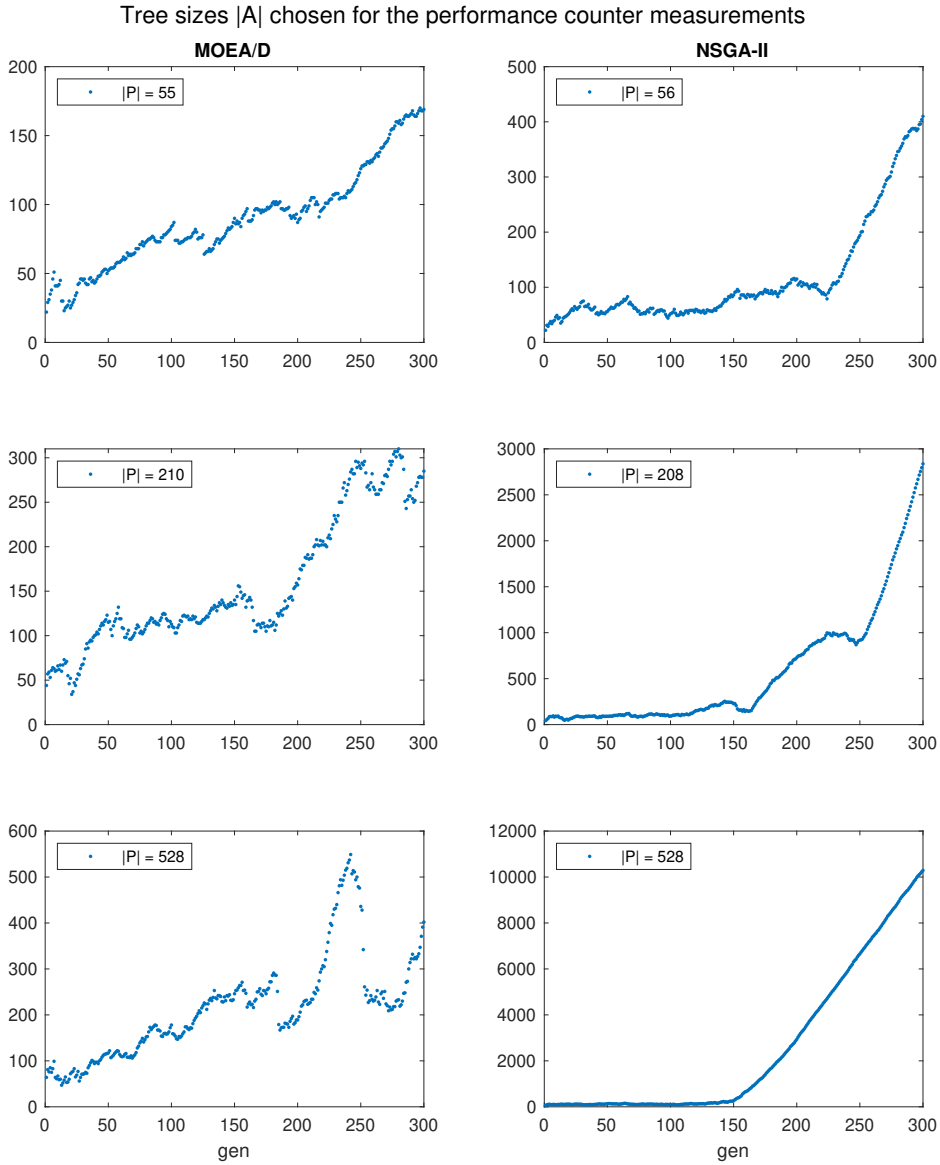

Figure 6.20 Tree sizes v. gen for the runs used to measure performance counters (quad-tree with MOEA/D and quad-tree with NSGA-II)

Figure 6.21 and Figure 6.22 show the means normalised by $|A|.|P|$ for the two sets of performance counter for three populations using MOEA/D, and three similar populations (in brackets) using NSGA-II. We see most counters scale well with $|A|$ (the dependency on generation is eliminated) and $|P|$ (the normalised count levels lie within the same range).

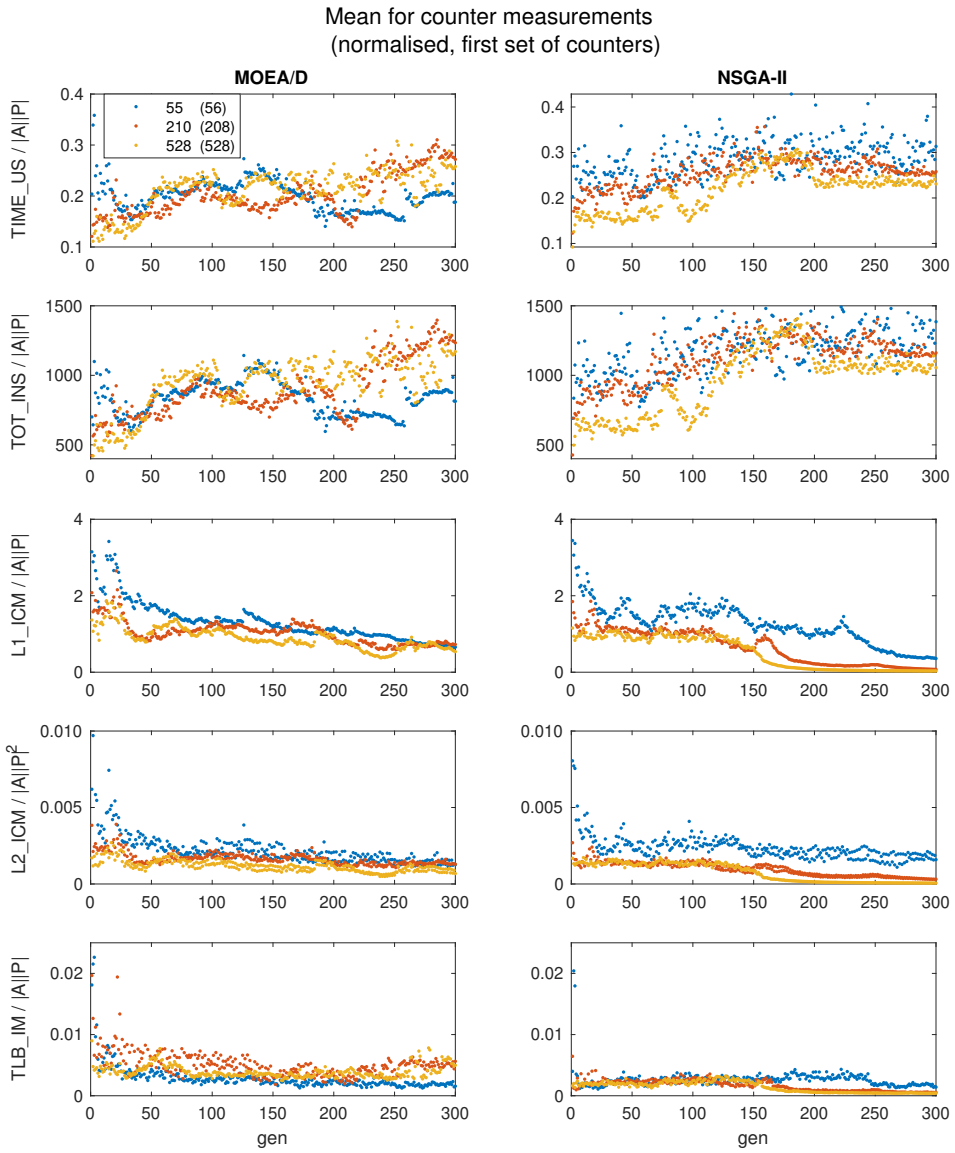

Figure 6.21 Counter measurements for the first set of counters and their means for three populations, normalised by $|A|.|P|$, using quad-tree with MOEA/D (first column) and NSGA-II (second column). The legend indicates the populations for MOEA/D and (in brackets) for NSGA-II

Notice the readings for L1_DCM (Figure 6.22) show a higher level of noise and deviate from the the other populations after the 50th generation for $|P| = 528$ and after the 220th generation for $|P| = 210$. This could be indicate a data access dependency on tree size and the population, however further measurements for other $|A|$ profiles are necessary to clarify these preliminary observations.

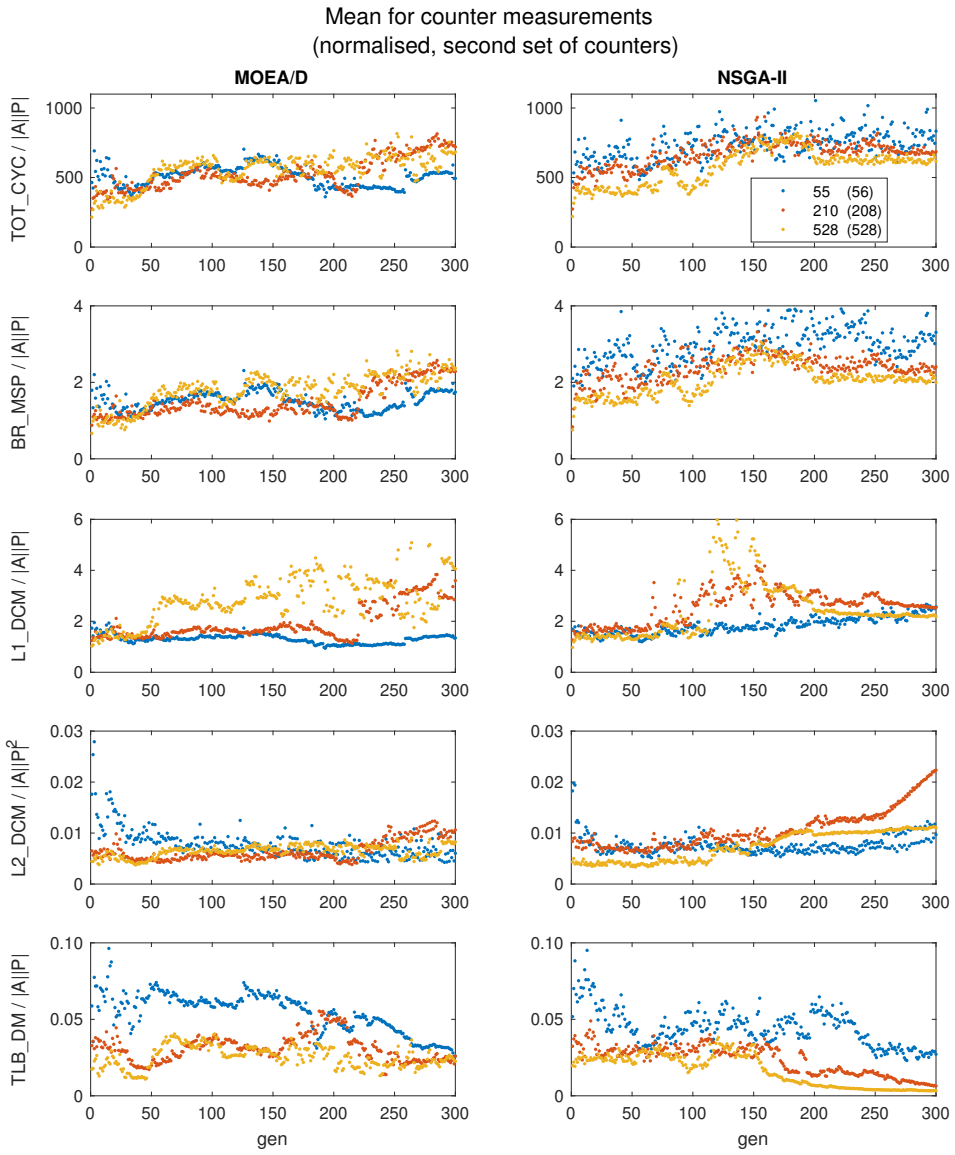

Figure 6.22 Counter measurements for the first set of counters and their means for three populations, normalised by $|A|.|P|$, using quad-tree with MOEA/D (first column) and NSGA-II (second column). The legend indicates the populations for MOEA/D and (in brackets) for NSGA-II

For NSGA-II the count levels are comparable to the results with MOEA/D, however we observe clear qualitative differences. In particular, counts for NSGA-II show a different behaviour below and above the 150th generation. Recalling the growth profile for $|A|$ (Figure 6.20) we notice these two areas correspond with two clearly distinguishable tree growth rates. The effects of tree growth rate are most evident for L1_ICM counts (Figure 6.21), showing clear decrease in the normalised count level meaning a dependency on $|A|$ is still present for large values of $|A|$. In the case of L1_DCM (Figure 6.22), the counts scale well for both regimes, showing only a temporary increase in the generation range 100-200 for $|P| = 208$ and $|P| = 528$, which corresponds to the area before the growth rate of the tree changes.

Another important variable to consider regarding the performance counter readings is the data structure being used to store the Pareto archive. Figure 6.23 shows the results using a quad-tree and a list with the NSGA-II algorithm. The light shade colours correspond to the list measurements for the same populations as the dark shade (for the quad-tree results). Notice the counter for L2_ICM (Figure 6.21) shows the counts transitioning between two levels. This is an effect related to the measuring counter in itself and not exclusive to lists: we observed it for all the readings we measured using this counter however in other cases the difference between the two channels is small compared to the relative count levels.

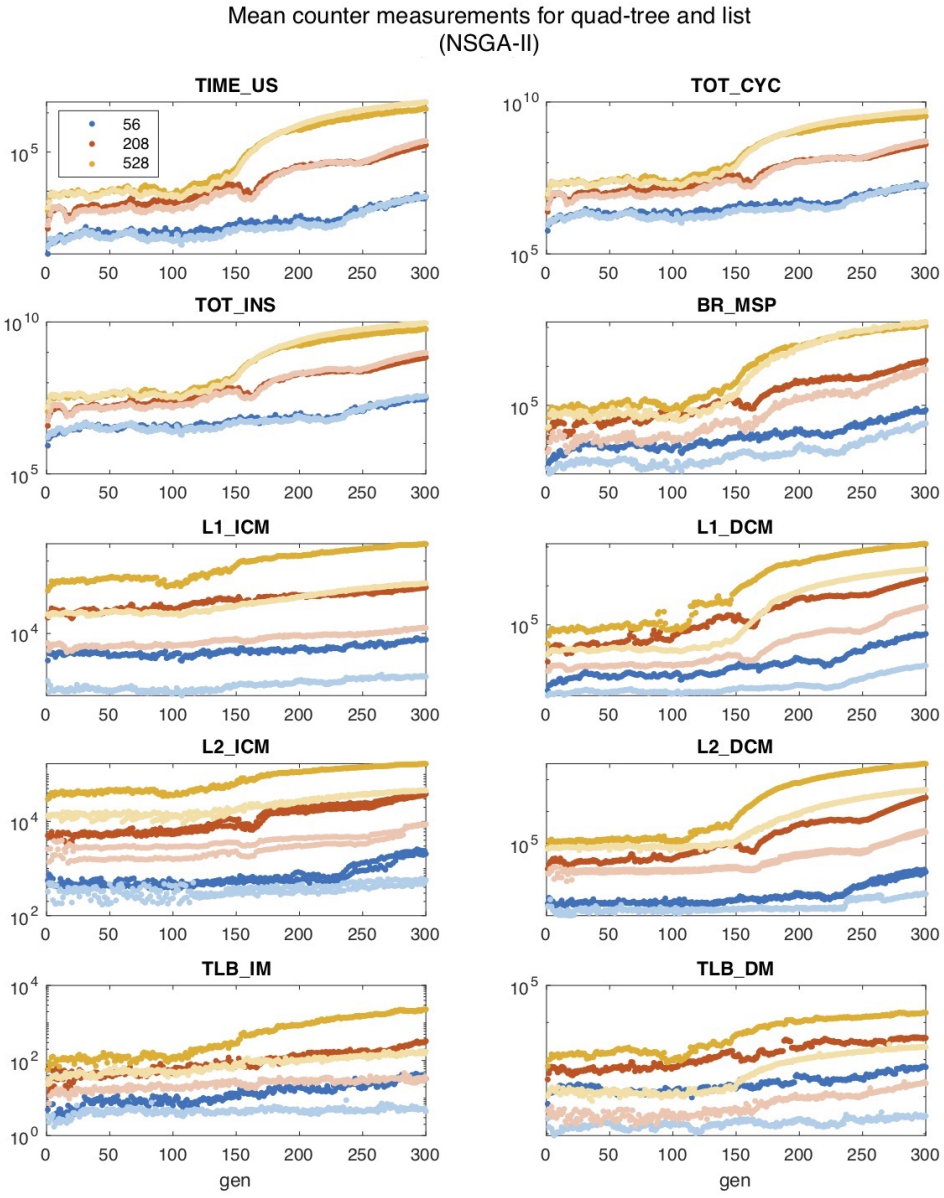

Figure 6.23 Mean counter measurements for the quad-tree and the non-dominated list for three populations (NSGA-II). The dark shaded dots are the results for the quad-tree, the light shaded dots for the list (same populations as the corresponding colour in legend)

Both the list and quad-tree show similar results for time, cycle and instruction counters, with the list showing lower readings than the quad-tree up to generation $\sim$ 170 and

switching to higher reads for generations > 170. These observations are compatible with those reported in Mostaghim and Teich [2005], who found the relative efficiency of a quad-tree respect to a list depends on $|P|$ and $|A|$, with quad-trees more efficient than linear lists when the archive sizes are small and the population sizes are large. In our measurements this region might be found at very low generations where $|A|$ is still small compared to the population size and where time measurements for the list and quad-tree are within the same range. For large $|A|$ (generations > 150) the list becomes slower than the quad-tree for $|P| = 208$ and $|P| = 528$ while for $|P| = 56$ there seem to be no difference, which coincides with Mostaghim and Teich [2005] observations for smaller population sizes and larger archive sizes, where they found linear lists take less time than quad-trees.

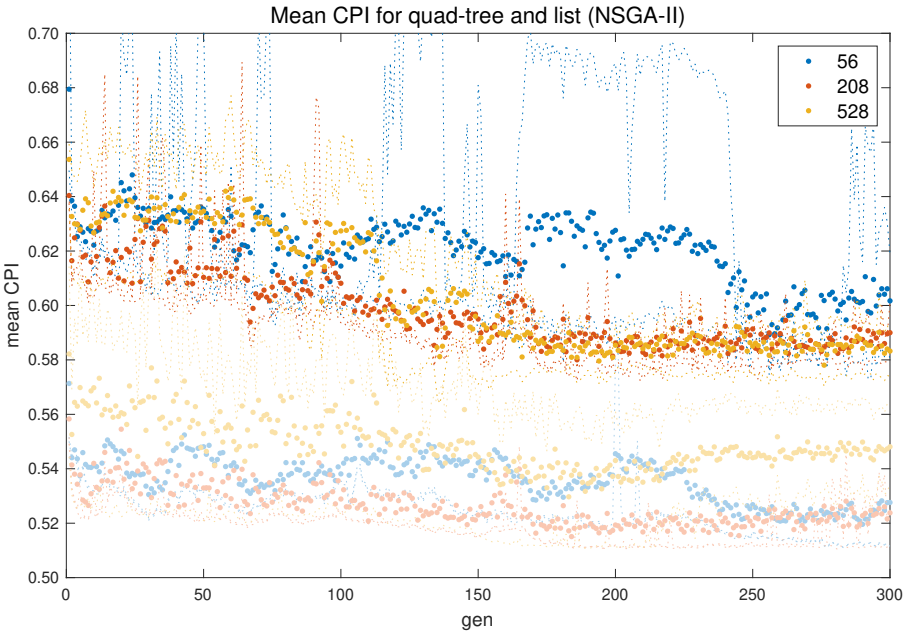

Figure 6.24 CPI mean extracted from the results in Figure 6.23, for quad-tree and list and populations 56, 208 and 528 (NSGA-II). The dotted lines are the quantiles

Figure 6.24 shows the cycles per instructions mean calculated from the measurements for cycles and instructions extracted from the un-normalised results of Figure 6.23, for quad-tree and list. The quantiles are shown as dotted lines. Notice that apart from $|P| = 56$, the quantiles remain close to their means. The mean CPI for quad-tree varies between ~ 0.64 for generations < 150 to 0.58 for generations > 150, suggesting a higher efficiency for high values of $|A|$. In the case of the list, we observe lower variations within the same population, and a higher average CPI over the whole range for $|P| = 528$.

Figure 6.25 shows results normalised by $|A|.|P|$. In the case of time, cycles and instructions, measurements are within the same range after scaling. There is however a clear crossing of the counts for the quad-tree and list at generation ~ 150, and also a separation between $|P| = 528$ from the other two populations for the list. We also notice differences in level for the other counters, dependent on population size and with distinguishable regions below and above generation ~ 150. For L1_DCM it is particularly noticeable the separation of the quad-tree and list groups, pointing to a data structure dependent profile for data cache misses.

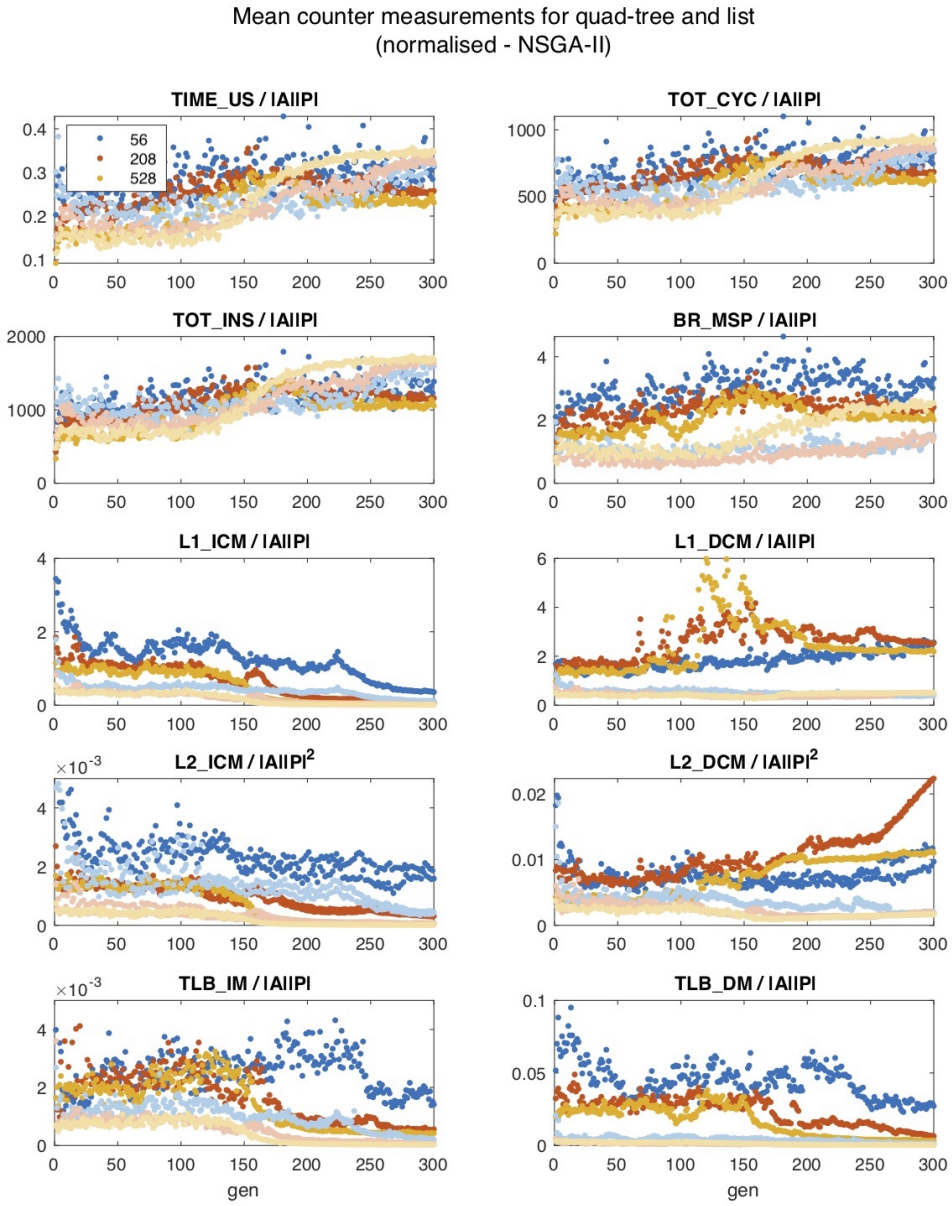

Figure 6.25 Mean counter measurements for quad-tree and list for three populations, normalised by $|A|.|P|$ (NSGA-II). The dark shaded dots are the results for the quad-tree, the light shaded dots for the list (same populations as the corresponding colour in legend)

These preliminary findings suggest the importance of including a variety of events when studying the performance of a data structure, even when presenting comparative results. The difference in counts for the list and the quad-tree for the cache miss events would be different using another architecture running the same experiment, and changing the effective performance profile and possibly changing the points of transition of the areas where one of the structure seems more efficient than the other.

6.4 Summary

We first introduced the concept of a generalised performance stack applied to the implementation to solve a MOP. We developed a software implementation of a quad-tree and we measured the average tree size over the evolution period, evolving the population with MOEA/D and NSGA-II. We found the estimated Pareto front advances differently depending on the algorithm being used to evolve the population, and we found some differences also as a function of the search population size.

We set up the performance counters corresponding to the events used in the naive approach, and we starting by measuring the overheads for our quad-tree program. We found background counts depending on population, indicating the presence of out-of-order execution within the pipeline. This dependency also hints to the non-trivial effect of the environment variables in performance measurements that might be independent of the data structure being measured.

We chose one of the runs for each of the experimental set-ups and we measured the performance counter set-up. We found dependencies of the counts with the population size, archive size and the algorithm used to evolve the population.

While more measurements are needed, our results highlight the need of an inclusion of architectural effects to arrive to an accurate characterisation of the performance of a particular MOP data structure implementation. The choice of evolutionary algorithm for the problem is also a variable that might affect the performance characteristics of a particular structure.

7 Conclusion

We started by studying the architecture of a computer machine. Using performance counters we measured cycles and instructions of a small piece of code at different implementation levels, experimenting with approaches of varying complexity. We found the most accurate results using a low level kernel module, however precision comes to the detriment of user friendliness, complexity of implementation and portability. Results obtained with higher level implementations are comparable and have similar noise levels. We chose PAPI as the preferred software interface to measure performance counters. PAPI has a higher overhead than the other methods we tried, but it is highly portable and very user friendly, having a large set of preset performance counters. We concluded that it is preferable to use low level implementations for micro-benchmarking small pieces of code where low readings are expected and the overhead in using PAPI would affect the results. For larger pieces of code and general profiling (benchmarking of a whole program) PAPI is a better choice.

We then introduced the use of data structures as an archive for solutions when using evolutionary algorithms to solve multi-objective optimisation problems. We discussed the quad-tree data structure and we implemented it with a program in C++. Using the PAPI library we measured a set of 9 performance counters to explore the behaviour of events throughout the running of a program. We analysed the effect of the population size and archive size on the chosen metrics and compared results using two different algorithms to evolve the population.

While the results are preliminary and more measurements are needed to establish more conclusive findings, our initial explorations highlight the importance of processor underlying architecture in data structure performance tests. Even in comparative performance measurements using a benchmark data structure (such as a list), the results might be highly dependent on the particular architecture of the system used. Moreover, there might be other factors affecting the relative performance such as the algorithm used to evolve the population, the particular choice of benchmark function and the technical details of the software implementation written to perform the tests. The individual contribution of all these software components might be directly affected by the computer architecture, resulting in a total effect that could give place to results dependent on implementation and system.

Future Work

Future work will include a broader set of counters. The events measured through the 9 counters chosen in this project while being illustrative are not sufficient. A more thorough approach should include the events and follow the methodology suggested by Yasin [2014]. The experiment will be repeated over a broader range of archive sizes and populations, using different algorithms to evolve the population (in particular completion of measurements for IHS) and other benchmark functions with varying number of objectives to test dimensionality effects. Ideally, these results will help building a phase diagram (if possible) where we could identify a relationship between measured variables and events, with their areas of performance, allowing to locate the ideal combination of software elements for the system where the experiment will be run.

Analysis of the performance for a particular MOEA under different test problem suites could also be used to determine program similarity. The features chosen in the construction of test problem suites for MOEAs do not necessarily produce a broad workload distribution Eeckhout et al. [2003]; Vandierendonck and De Bosschere [2004]. This could result in the selection of test problems under which the performance of an MOEA is biased by the kind of operations being tested. Understanding how these features influence performance can be used to explore the sampling range of commonly used test suites. Future work will be undertaken towards this direction.

Bibliography

Adhianto, L., Banerjee, S., Fagan, M., Krentel, M., Marin, G., Mellor-Crummey, J., and Tallent, N. R. (2010). HPCTOOLKIT: Tools for performance analysis of optimized parallel programs. *Concurr. Comput. Pract. Exp.*, 22(6):685–701.

Ailamaki, A., DeWitt, D. J., Hill, M. D., and Wood, D. A. (1999). DBMSs on a modern processor: Where does time go? *VLDB" 99, Proc. 25th Int. Conf. Very Large Data Bases, Sept. 7-10, 1999, Edinburgh, Scotland, UK*, pages 266–277.

Altwaijry, N. and El Bachir Menai, M. (2012). Data Structures in Multi-Objective Evolutionary Algorithms. *J. Comput. Sci. Technol.*, 27(6):1197–1210.

Bakhvalov, D. (2018). PMU counters and profiling basics .

Biscani, F., Izzo, D., and Yam, C. H. (2010). A Global Optimisation Toolbox for Massively Parallel Engineering Optimisation.

Biscani, F. M. P. I. f. A. and Izzo, D. E. S. A. (2019). PAGMO.

Chen, W.-K., Chau, J., Bhansali, S., de Jong, S., Edwards, A., Drinić, M., Murray, R., and Mihočka, D. (2012). Framework for instruction-level tracing and analysis of program executions. page 154.

Coello, C. A. C. (2018). Multi-objective Optimization. *Handb. Heuristics*, pages 1–28.

Coello, C. C., Lamont, G. B., and Veldhuizen, D. A. V. (2007). *Evolutionary algorithms for solving multi-objective problems.*

Coello Coello, C. A. (2006). Evolutionary Multi-Objective Optimization: A Historical View of the Field. *IEEE Comput. Intell. Mag.*, 1(1):28–36.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2001). *Introduction to Algorithms 2nd Edition.* MIT Press.

Das, S., Werner, J., Antonakakis, M., Polychronakis, M., and Monrose, F. (2019). SoK: The Challenges, Pitfalls, and Perils of Using Hardware Performance Counters for Security. *Proc. - IEEE Symp. Secur. Priv.*

Deb, K. (1999). Evolutionary algorithms for multi-criterion optimization in engineering design. *Proc. Evol. Algorithms Eng. Comput. Sci.*, (ii):135–161.

Deb, K. (2001). *Multi-Objective Optimization Using Evolutionary Algorithms*. John Wiley \& Sons, USA.

Deb, K., Pratap, A., Agarwal, S., and Meyarivan, T. (2002a). A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Trans. Evol. Comput.*, 6(2):182–197.

Deb, K., Thiele, L., Laumanns, M., and Zitzler, E. (2002b). Scalable multi-objective optimization test problems. In *Proc. 2002 Congr. Evol. Comput. CEC 2002*, volume 1, pages 825–830.

Dementiev, R. I. (2016). Monitoring Integrated Memory Controller Requests in the 2nd, 3rd, 4th, 5th, 6th generation Intel® Core™ processors.

Dimakopoulou, M., Eranian, S., Koziris, N., and Bambos, N. (2017). Reliable and Efficient Performance Monitoring in Linux. *Int. Conf. High Perform. Comput. Networking, Storage Anal. SC*, (November):396–408.

Drozdik, M., Akimoto, Y., Aguirre, H., and Tanaka, K. (2014). Computational Cost Reduction of Non-dominated Sorting Using M-front. *IEEE Trans. Evol. Comput.*, 19(c):1–1.

Eeckhout, L., Vandierendonck, H., and De Bosschere, K. (2003). Quantifying the impact of input data sets on program behavior and its applications. *J. Instr. Parallelism*, 5(1):1–33.

Eiben, A. E. and Smith, J. E. (2015). *Introduction to Evolutionary Computing*. Springer Berlin Heidelberg.

Eranian, S. (2006). Perfmon2: a flexible performance monitoring interface for Linux. In *Proc. 2006 Ottawa Linux Symp.*, pages 269—-288.

Eyerman, S., Eeckhout, L., Karkhanis, T., and Smith, J. E. (2006a). A performance counter architecture for computing accurate CPI components. In *Proc. 12th Int. Conf. Archit. Support Program. Lang. Oper. Syst. - ASPLOS-XII*, number August, page 175.

Eyerman, S., Eeckhout, L., Karkhanis, T., and Smith, J. E. (2009). A mechanistic performance model for superscalar out-of-order processors. *ACM Trans. Comput. Syst.*, 27(2):1–37.

Eyerman, S., Heirman, W., Du Bois, K., and Hur, I. (2018). Multi-Stage CPI Stacks. *IEEE Comput. Archit. Lett.*, 17(1):55–58.

Eyerman, S., Smith, J., and Eeckhout, L. (2006b). Characterizing the branch misprediction penalty. *2006 IEEE Int. Symp. Perform. Anal. Syst. Softw.*, pages 48–58.

Finkel, R. A. and Bentley, J. L. (1974). Quad trees a data structure for retrieval on composite keys. *Acta Inform.*, 4(1):1–9.

Intel (1997). Using the RDTSC Instruction for Performance Monitoring.

Intel (2016). Intel ® 64 and IA-32 Architectures Optimization Reference Manual. (June).

Intel (2017). Intel 64 and IA-32 Architectures Performance Monitoring Events.

Intel (2018a). Intel ® 64 and IA-32 Architectures Software Developer ' s Manual Model-Specific Registers. 4(334569).

Intel (2018b). Intel ® 64 and IA-32 Architectures Software Developer ' s Manual V.2. 2(253665).

Intel (2018c). Intel ® 64 and IA-32 Architectures Software Developer ' s Manual V.3. 3(253665).

Intel (2018d). *Intel® 64 and IA-32 Architectures Software Developer's Manual V.1*, volume 1.

Jarp, S., Jurga, R., and Nowak, A. (2008). Perfmon2: A leap forward in performance monitoring. In *J. Phys. Conf. Ser.*, volume 119.

Jensen, M. T. (2003). Reducing the Run-Time Complexity of Multiobjective EAs : The NSGA-II and Other Algorithms. *IEEE Trans. Evol. Comput.*, 7(5):503–515.

Joshi, A., Phansalkar, A., Eeckhout, L., and John, L. K. (2006). Measuring benchmark similarity using inherent program characteristics. *IEEE Trans. Comput.*, 55(6):769–782.

Karkhanis, T. and Smith, J. (2004). A first-order superscalar processor model. In *Proceedings. 31st Annu. Int. Symp. Comput. Archit. 2004.*, pages 338–349. IEEE.

Keeton, K., Raphael, R. C., He, Y. Q., Baker, W. E., and Patterson, D. A. (2004). Performance characterization of a Quad Pentium Pro SMP using OLTP workloads. *ACM SIGARCH Comput. Archit. News*, 26(3):15–26.

Knuth, D. E. (1997). *The Art of Computer Programming: Fundamental Algorithms*, volume 1. Pearson Education.

Kufrin, R. (2005). PerfSuite An accessible Open source performance analysis.PDF. 2005(April).

Lee, E. A. (2006). The problem with threads. *Computer (Long. Beach. Calif).*, 39(5):33–42.

Levinthal, D. (2009). Performance Analysis Guide for Intel® CoreTM i7 Processor and Intel® XeonTM 5500 processors.

Li, H. and Qingfu Zhang (2009). Multiobjective Optimization Problems With Complicated Pareto Sets, MOEA/D and NSGA-II. *IEEE Trans. Evol. Comput.*, 13(2):284–302.

Mahdavi, M., Fesanghary, M., and Damangir, E. (2007). An improved harmony search algorithm for solving optimization problems. *Appl. Math. Comput.*, 188(2):1567–1579.

Márquez, A. L., Baños, R., Gil, C., Montoya, M. G., Manzano-Agugliaro, F., and Montoya, F. G. (2011). Multi-objective crop planning using pareto-based evolutionary algorithms. *Agric. Econ.*, 42(6):649–656.

Mazor, S. (2010). Intel's 8086. *IEEE Ann. Hist. Comput.*, 32(1):75–79.

Molka, D., Schöne, R., Hackenberg, D., and Nagel, W. E. (2017). Detecting Memory-Boundedness with Hardware Performance Counters. In *Proc. 8th ACM/SPEC Int. Conf. Perform. Eng. - ICPE '17*, pages 27–38, New York, New York, USA. ACM Press.

Mostaghim, S. and Teich, J. (2005). Quad-trees: A data structure for storing pareto sets in multiobjective evolutionary algorithms with elitism. In *Evol. multiobjective Optim.*, pages 81–104. Springer.

Mostaghim, S., Teich, J., and Tyagi, A. (2002). Comparison of data structures for storing Pareto-sets in MOEAs. In *Proc. 2002 Congr. Evol. Comput. CEC'02 (Cat. No.02TH8600)*, volume 1, pages 843–848. IEEE.

Mucci, P., Browne, S., Deane, C., and Ho, G. (1999). PAPI: A portable interface to hardware performance counters. *Proc. Dept. Def. HPCMP Users Gr. Conf.*, 32:7–10.

Mytkowicz, T., Diwan, A., Hauswirth, M., and Sweeney, P. F. (2009). Producing wrong data without doing anything obviously wrong! *ACM SIGPLAN Not.*, 44(3):265.

Nethercote, N. (2004). Dynamic binary analysis and instrumentation. Technical Report UCAM-CL-TR-606, University of Cambridge, Computer Laboratory.

Oraei Zare, S., Saghafian, B., Shamsai, a., and Nazif, S. (2012). Multi-objective optimization using evolutionary algorithms for qualitative and quantitative control of urban runoff. *Hydrol. Earth Syst. Sci. Discuss.*, 9(1):777–817.

Paoloni, G. (2010). How to Benchmark Code Execution Times on Intel ® IA-32 and IA-64 Instruction Set Architectures. *Slides*, (September):1–37.

PAPI Software. Currently Supported Platforms - PAPI website.

Patt, Y. (2001). Requirements, bottlenecks, and good fortune: agents for microprocessor evolution. *Proc. IEEE*, 89(11):1553–1559.

Perfmon2 (2018). Peformance Monitoring Software.

Rohl, T., Eitzinger, J., Hager, G., and Wellein, G. (2017). LIKWID monitoring stack: A flexible framework enabling job specific performance monitoring for the masses. In *Proc. - IEEE Int. Conf. Clust. Comput. ICCC*, volume 2017-Septe, pages 781–784.

Röhl, T., Treibig, J., Hager, G., and Wellein, G. (2015). Overhead Analysis of Performance Counter Measurements. In *Proc. Int. Conf. Parallel Process. Work.*, volume 2015-May, pages 176–185.

Sprunt, B. (2002a). Pentium 4 performance-monitoring features. *IEEE Micro*, 22(4):72–82.

Sprunt, B. (2002b). The basics of performance-monitoring hardware. *IEEE Micro*, 22(4):64–71.

Stallings, W. (2000). *Computer organization and architecture: designing for performance*. Pearson Education India.

Sun, M. and Steuer, R. E. (1996). Quad-trees and linear lists for identifying nondominated criterion vectors. *INFORMS J. Comput.*, 8(4):367–375.

Tanabe, R., Ishibuchi, H., and Oyama, A. (2017). Benchmarking Multi- and Many-Objective Evolutionary Algorithms under Two Optimization Scenarios. *IEEE Access*, 5:19597–19619.

Tanenbaum, A. S. and Austin, T. (2013). *Structured computer organization, Sixth Edition*. Pearson.

Terpstra, D., Jagode, H., You, H., and Dongarra, J. (2010a). Collecting Performance Data with PAPI-C. *Tools High Perform. Comput. 2009*, pages 157–173.

Terpstra, D., Jagode, H., You, H., and Dongarra, J. (2010b). *Tools for High Performance Computing 2009*. Number May. Springer Berlin Heidelberg, Berlin, Heidelberg.

Thiel, J. (2006). An overview of software performance analysis tools and techniques: From gprof to dtrace. Technical report.

Van Den Steen, S., Eyerman, S., De Pestel, S., Mechri, M., Carlson, T. E., Black-Schaffer, D., Hagersten, E., and Eeckhout, L. (2016). Analytical Processor Performance and Power Modeling Using Micro-Architecture Independent Characteristics. *IEEE Trans. Comput.*, 65(12):3537–3551.

Vandierendonck, H. and De Bosschere, K. (2004). Many benchmarks stress the same bottlenecks. *Work. Comput. Archit. Eval. Using Commer. Workload.*, pages 57–64.

Weaver, V. M. (2011). The Unofficial Linux Perf Events Web-Page.

Weaver, V. M. (2012). Perf Event Overhead Measurements.

Weaver, V. M. (2015). Self-monitoring overhead of the Linux perf- event performance counter interface. *ISPASS 2015 - IEEE Int. Symp. Perform. Anal. Syst. Softw.*, pages 102–111.

Weaver, V. M. and McKee, S. A. (2008). Can hardware performance counters be trusted? *2008 IEEE Int. Symp. Workload Charact. IISWC'08*, pages 141–150.

WikiChip. Skylake (client) - Microarchitectures - Intel.

Yang, X.-S. (2014). *Nature-Inspired Optimization Algorithms*. Elsevier.

Yasin, A. (2014). A Top-Down method for performance analysis and counters architecture. *ISPASS 2014 - IEEE Int. Symp. Perform. Anal. Syst. Softw.*, (October):35–44.

Zaparanuks, D., Jovic, M., and Hauswirth, M. (2009). Accuracy of performance counter measurements. In *ISPASS 2009 - Int. Symp. Perform. Anal. Syst. Softw.*, pages 23–32. IEEE.

Zhang, Q. and Li, H. (2007). MOEA/D: A multiobjective evolutionary algorithm based on decomposition. *IEEE Trans. Evol. Comput.*, 11(6):712–731.

Zhou, A., Qu, B.-Y., Li, H., Zhao, S.-Z. S.-Z., Suganthan, P. N., and Zhang, Q. (2011). Multiobjective evolutionary algorithms: A survey of the state of the art. *Swarm Evol. Comput.*, 1(1):32–49.

Zitzler, E., Deb, K., and Thiele, L. (2000). Comparison of multiobjective evolutionary algorithms: Empirical results. *Evol. Comput.*, 8(2):173–195.

Zong Woo Geem, Joong Hoon Kim, and Loganathan, G. (2001). A New Heuristic Optimization Algorithm: Harmony Search. *Simulation*, 76(2):60–68.