Performance Engineering for Graduate Students: a View from Amsterdam

Ana-Lucia Varbanescu
a.l.varbanescu@utwente.nl
University of Twente
Enschede, The Netherlands

Stephen Nicholas Swatman
s.n.swatman@uva.nl
University of Amsterdam
Amsterdam, The Netherlands

ABSTRACT

HPC relies on experts to design, implement, and tune (computational science) applications that can efficiently use current (super)computing systems. As such, we strongly believe we must educate our students to ensure their ability to drive this activities, together with the domain experts. To this end, in 2018, we have designed a performance engineering course that, inspired by several conference-like tutorials, covers the principles and practice of performance engineering: benchmarking, performance modeling, and performance improvement. In this paper, we describe the goals, learning objectives, and structure of the course, share students feedback and evaluation data, and discuss the lessons learned. After teaching the course five times, our results show that the course is tough (as expected) but very well received, with high-scores and several students continuing on the path of performance engineering during and after their master studies.

1 INTRODUCTION

As high-performance computing (HPC) focuses on the design of applications (and systems) that maximize performance and efficiency. HPC is often defined in the context of big science, large-scale applications, but has been slowly expanding - sometimes wiht different names, such as HPDA (high-performance data analytics) or HTC (high-throughput computing) - towards medium-size applications and simulation, and even non-scientific applications - like, for example, traditional data science or machine learning applications.

Maximizing performance was very often the task of library developers and/or expert developers, who focused specifically on a couple of codes and, with inside knowledge of the implementation and problem domain, managed to steadily improve these applications for the specific target machines - often supercomputers with somewhat custom architectures. However, the rapid development of multi- and many-core processors and accelerators, and their creative combinations in large-scale clusters and supercomputers, has significantly increased the demand for HPC, to the point where it has far exceeded the ability of experts to keep up. To cope with this massive demand, better tools and more systematic approaches are needed to provide highly-optimized, very efficient versions of HPC applications for the broad range of computing systems. We call the discipline that focuses on this systematic approach **Perforance Engineering**. In this paper, we discuss the design and implementation of a graduate (i.e., MSc) level course on performance engineering. This is the first and only such course in The Netherlands to this date.

Our approach to performance engineering covers application and system characterization, performance modeling, and optimizations. We target multi-node heterogeneous platforms combining CPUs and GPUs (as accelerators), and use existing methods and tools - often used by experts - to demonstrate how the performance engineering process can be effectively conducted regardless of the specifics of the application and/or machine. To this end, the course covers both theoretical and practical aspects of the process: the lectures describe the concepts and methods, while the practical assignments enable students to actually test these methods and tools, and experience their strengths and limitations. The course further includes a project where, using an application of their choice, the students demonstrate all the stages of the performance engineering process, and ultimately provide a *better* version of the chosen application. In this paper, we describe in detail how these three components are combined.

We have taught the performance engineering course for 5 years (in-person and online), and we have collected valuable feedback from around 80 students who participated in the course. The feedback - from which we extract the data we use in this paper as evaluation data - indicates that students highly appreciate the course and its topics, even though they find the course difficult in terms of workload. The feedback, combined with the assessment results, have also provided us valuable insights into how to further improve the course; we also shared these lessons learned in this paper.

In summary, the main contribution of this work is to describe the first (and only) performance engineering course for HPC in The Netherlands, from design to implementation and evaluation, including lessons learned and future work suggestions. To this end, the remainder of this paper is structured as follows. In Section 2 we introduce the main terminology we use, and discuss related courses/tutorials that have inspired us in our design. Next, sections 3 and 4 introduce the design and implementation of the course. In section 5 we present the analysis of the students results (and feedback). Finally, we discuss several lessons learned in section 6, and conclude the paper in section 7.

2 BACKGROUND AND RELATED WORK

In this section we provide a short overview of the technologies and definitions we use in the course, as well as a brief discussion on related work.

2.1 Computing systems

The course covers heterogeneous, potentially multi-node systems built from CPUs and GPUs. *CPUs* are representative multi-cores, with multiple caches and shared main memory. *GPUs* are representative many-cores, and are used as accelerators for the main CPU - i.e., the GPU is the accelerator *device* to the CPU *host*. *Multiple nodes* - potentially heterogeneous both intra- and inter-node - are

representative for scaled-out systems, like clusters and supercomputers.

2.2 Programming languages

Programming HPC has not yet reach consensus in terms of a standardized (set of) programming model(s) for its supercomputing systems. Therefore, based on popularity and accessibility, our course uses C/C++ as main programming language, combined with OpenMP (for shared-memory and accelerator programming), CUDA and OpenACC (for NVIDIA GPUs), and MPI for distributed computing. While our approach to performance engineering, and the methods we propose to support it, are agnostic to the programming models and languages being used, we find that most open-source tools are built to work with this mix of models. We further rely on assembly for several of the detailed, low-level analyses we teach in the course.

2.3 Performance Engineering

For the purpose of our course, we adapted the definition of performance engineering from the discipline of software engineering [6] as follows: *Software Performance Engineering (SPE) is a systematic, quantitative approach to the cost-effective development of software systems to meet performance requirements. SPE is a software-oriented approach that focuses on architecture, design, and implementation choices. SPE provides the information needed to build software that meets performance requirements within a given budget (defined in terms of time, cost, or efficiency).*

Further inspired by [2, 6], we define the process of performance engineering as a seven-stage, iterative process, as follows:

(1) Collect and analyse (user) performance requirements.
(2) Understand current performance.
(3) Assess feasibility of the requirements.
(4) Assess suitable approaches to meet the requirements (including algorithm and/or system (co-)design).
(5) Apply tuning and optimization.
(6) Assess progress and iterate back to steps 3-5.
(7) Analyse and document the process and the final result.

The lectures cover all stages, while the practical aspects of the course specifically target stages 2-6, allowing students to learn how to use (and further become proficient using) different tools for each of these stages.

2.4 Related Work

We have started our course from the concepts of "performance engineering" found in the software engineering discipline [6]. We combined these aspects with methods and tools presented in scientific articles - for example, the Roofline model [10], the ECM model [5], microbenchmarking [11] and benchmarking [3, 4]. We further incorporate the technical and practical aspects described in tutorials presented in the relevant HPC venues - for example: single-core performance engineering (at SC and ICPE), performance modeling using the Roofline model for CPUs and GPUs (at SC, NVIDIA TechDays), performance modeling for distributed applications (at SC, ISC), or the polyhedral model (at HiPEAC). As such, we have designed a course that provides a unique, novel combination of theoretical concepts and methodological aspects for performance engineering.

While ours is the first and only course on performance engineering in The Netherlands, we have also drawn inspiration from courses like Computer Systems from CMU (http://15418.courses.cs.cmu.edu/spring2016/lectures), Performance Engineering from MIT[1], Performance Modeling from TUDelft[2] and University of Edinborough[3], and Queuing Theory from MIT [4]. These courses provided inspiration in terms of matching theoretical topics with tools and assignments. However, to the best of our knowledge, no other academic course currently taught provides the same range and mix of topics as our course.

3 COURSE DESIGN

In this section we describe the goals of our course, the learning objectives, and the structure we propose. We provide further details on the actual implementation in section 4.

3.1 Goals and Learning Objectives

The course focuses on the modern aspects of performance engineering in the context of parallel applications and distributed heterogeneous (super)computing systems. To this end, the course introduces performance metrics and measurement techniques, performance analysis, benchmarking and microbenchmarking, performance models (analytical and statistical), and performance prediction. The course further demonstrates how to apply these techniques for several applications running on multi-node computing systems, featuring combinations of multi-core CPUs and many-core GPU accelerators. The ultimate goal of the course is to bundle these techniques together and provide students the opportunity to *create their own performance engineering toolbox, which they can successfully use to deploy a systematic approach for performance engineering on any application.*

The course is designed to meet seven learning objectives [8]. Specifically, at the end of the course, students will be able to:

LO1 quantify (using the appropriate tools and methods) the performance of an application running on a computing system using the appropriate metric;
LO2 demonstrate and compare several performance modeling methods, and assess their usefulness for practical problems;
LO3 classify and use several performance prediction methods, and compare their applicability in practice;
LO4 design an empirical performance analysis process for any application, interpret its results, and recommend solutions for performance improvement;
LO5 design and use a suitable model for accurate performance prediction for a given application;
LO6 apply and assess different (existing) optimization techniques to paralell and distributed codes;

[1] https://ocw.mit.edu/courses/6-172-performance-engineering-of-software-systems-fall-2018/
[2] https://repository.tudelft.nl/islandora/object/uuid%3Aea4a3f85-597c-479c-b045-15da8dc4850b
[3] https://www.inf.ed.ac.uk/teaching/courses/pm/
[4] https://ocw.mit.edu/courses/15-072j-queues-theory-and-applications-spring-2006/

LO7 design and develop a complete performance engineering process, apply it successfully on any given application, and assess its outcome in terms of performance gain;
LO8 Use different performance engineering tools (e.g., profilers, microbenchmarks and benchmarks, performance counters libraries, etc.).

3.2 Prerequisites

Performance engineering is a multi-disciplinary topic, combining computer systems and computer architecture, algorithm design, parallel and distributed algorithms, and parallel programming. For students to succeed in following thie course, we formulated the following prerequisites:

P01 computer organization and architecture basics, including data representation, CPU architecture and functionality, assembly language literacy, memory hierarchy and caching, multi- and many-core architectures;
P02 computer systems fundamentals, including shared-memory systems, distributed systems, and heterogeneous systems design and implementation;
P03 Parallel algorithms design and development, and basic skills in C/C++ as support language.
P04 Parallel and distributed programming basics, including basic programming models for such programming: OpenMP, CUDA/OpenCL, MPI.
P05 Basic statistics and data analysis methods, including regression techniques.

The performance engineering course is embedded in a computer science master program where students follow all these topics to a certain extent. Although we provide quick refreshers for all the topics discussed above, we oobserve that students who have prior knowledge in these topics (in the sense of having actively taken courses, solved assignments, or developed small projects in these topics) perform better, because they can build their own performance engineering toolbox and expertise on top of assimilated knowledge.

Overall, the most successful students in the course are computer science students, but we have had several successful graduates from computational science (where modeling and numerical methods are focused much more than systems and performance aspects) and exchange students with diverse backgrounds. These successes indicate that the course is self-contained and comprehensive.

3.3 Course Structure

To reach the course goal and achieve its learning objectives, we have structured the course along three types of activities: lectures, assignments, and project.

Lectures. We designed the theoretical part of the course to cover the principle and methods of performance engineering. To this end, we teach *how* to correctly measure and communicate performace data, design microbenchmarks and benchmarking suites, design and validate different performance models, and design and optimize parallel applications for modern computing systems. Lectures combine fundamental systems and applications knowledge with modern, state-of-the-art methods from literature.

Assignments. The course relies on assignments to encourage students to link the theoretical aspects presented in the course with existing processing and tools that facilitate the application of these theoretical aspects in practice. The assignments formulate a general problem, provide a starting point in the form of code templates, and require students to follow specific performance engineering steps to analyse and/or improve the code. All assignments rely heavily on empirical research - from experimental design to data analysis -, but use simple applications to limit the coding effort required for completion. All assignments also require detailed documentation to allow students to learn not only how to execute performance engineering tasks, but also how to document them for non-expert stakeholders (e.g., domain experts, application owners, system designers).

Project. While assignments are self-contained and designed to focus on specific aspects of the performance engineering process, the project enables students to experience the deployment of performance engineering for a larger, real case-study application. This enables them to better understand the limitations and challenges of the provided methods and tools, and the manual effort still required when actually aiming to systematically improve application performance, aiming at efficient HPC applications.

For the project, the students must achieve four important milestones:

(1) Define an application of interest and formulate a performance problem to be solved.
(2) Formulate a plan to deploy performance engineering methods to solve the proposed performance problem. The required elements of the plan are: benchmarking and starting performance analysis, requirements analysis, performance modeling and prediction, performance optimization, reflection.
(3) Document the performance engineering process.
(4) Present their intermediate and final results to an audience of their peers.

3.4 Assessment

The course uses continuous assessment - based on grading in-class quizzes and assignments, a final exam, and two project presentations.

The final exam assesses the theoretical knowledge of the students, targeting mostly the knowledge they have acquired during the lectures. The exam is included to ensure that students do understand the fundamentals of the tools they employ, which in turn demonstrates they can think critically about the output of the tools they use, as well as they can port their knowledge to new systems, tools, and applications. We find that such an exam is a real discriminating factor between students who only master the usage of tools and those who truly understand the methodological aspects of performance engineering. We use in-class quizzes to stimulate students to acquire such knowledge. The exam is, by design, individual - compared with the rest of the course components, which can be solved individually or, ideally, in teams of 2-4 students.

Assignments are provided to encourage students to practice with all tools we believe to be important for a performance engineer's arsenal. We find such assignments work when they are focused

and targeted at specific elements in the performance engineering process. The challenging aspect of these assignments is to find he right balance between workload and learning benefits.

Finally, we assess the completion of the chosen project, including the presentation and report the students provide. We assess the project based on the feasibility of the proposed plan to address the identified performance problem, the suitability of the employed methods and tools, and the communication aspect (including the reports and the presentation). We do not factor in the success (or lack thereof) in achieving a specific performance goal, but rather focus on the analysis and reflection aspects - i.e., how and why certain steps succeed or fail.

Overall, the final students grades are calculated as a weighted average of these three grades. We chose to give the largest weight to the project, as many of our students lack experience with working on real applications and systems, while being mostly exposed to simple, "in-lab" assignments that are dedicated to show how methods and tools *do* work. In the case of the projects, students mostly learn how and why many such methods and tools are limited and/or fail in a real-life scenario, and how they can use their theoretical knowledge to compensate for these limitations. We further choose equal weights for the assignments and the theoretical exam, but allow for slack for the students to slightly compensate one with the other, if/when needed (mode details in section 4.

4 COURSE IMPLEMENTATION

In this section we present the actual implementation of the proposed course. Specifically, we (briefly) discuss the alignment of learning objectives to the specific lectures and assignments, and the actual topics we cover with each of them. The course has been running in a block of 8 weeks, with three weekly scheduled activities: lectures, labs (dedicated to solve the practical assignments), and seminars (dedicated to discussing the projects). The final week is dedicated to the exam and project final presentation.

4.1 Lectures and topics

The course covers the following topics, aligned with both the performance engineering process (see section 2.3) and the learning objectives (see section 3.1):

- Basics of performance, basics of HPC applications and systems - recap of prerequisite information, definition of metrics and requirements (stage PE1, LO1)
- Code tuning and code optimization techniques (stage PE4, LO6, LO7)
- The Roofline model and extensions (stage PE2, LO1-3, LO7)
- Analytical modeling (stages PE2, PE3, LO1-4)
- Benchmarking and microbenchmarking (stages PE1, PE3, LO4, LO6-8)
- Data-driven and statistical modeling (stages PE1, PE3, LO1-3, LO6-7)
- Simulation and simulators (stages PE2, PE4, PE6, LO4, LO6-8)
- Performance counters and performance patterns (stages PE2, PE5-6, LO6-8)
- Scale-out to distributed systems (stages PE2-7, LO1, LO4-8)
- Queuing theory (stages PE3-4, LO2-3)
- The polyhedral model (stage PE5, LO6-8)

The course has been taught with different numbers of lectures: we started from 7 lectures (one per week), but, given the growing list of topics, we eventually extended the number of lectures to 10 - where we give two lectures per week in the first three weeks, where more theoretical knowledge is needed. The lectures are provided in the performance engineering repository [9].

4.2 Labs and Assignments

The course features four practical assignments, which are briefly introduced in the following paragraphs. The actual assignments, including code, data, and templates for reporting (where needed), are available in the course repository [9]. For all assignments we allow students to use their own machines and we further provide access to a local HPC cluster (featuring job isolation and dedicated hardware resources via a SLURM-based scheduler).

Assignment 1: the Roofline model. The goal of this assignment is to allow students to use one of the simplest performance modeling tools available to performance engineering: the Roofline model [10]. However, the model relies on a deep understanding of computer systems and their main performance indicators, the balance between compute- and memory-bound in systems and applications, and a characterization of the application itself. In this assignment, the students are provided a basic matrix multiplication code, and are asked to provide a Roofline model for the sequential code, then further optimize the code (based on the information from the Roofline model) - thus addressing the identified bottleneck(s) - and reapply the same modeling technique. This exercise aims to demonstrate that the model is able to capture different versions of the same code. Finally, the assignment also requires the students to implement and Roofline-model a parallel version of matrix multiplication. Again, the goal is to demonstrate how the model of both the system and the application change when parallelism is added.

To bootstrap this assignment, we provide sequential code for matrix multiplication, and suggest optimizations like loop reordering and loop tiling. We do not aim to achieve the best possible performing matrix multiplication, but rather to have different versions with different performance envelopes, to demonstrate the sensitivity of the model.

Finally, the assignment requires the students to practice their experimental design skills, as we request to provide models and evaluations of the model using different input datasets. We also suggest tools that can calculate and plot the model automatically, but want the students to reflect on the difference between modeling by hand and by tool. Please note that very little coding is required for this assignment in terms of adapting matrix multiplication (somewhere around 100 lines of code), but a lot of effort is required for systematic evaluation and model validation (for which we recommend and eventually provide scripts and templates).

Assignment 2: Analytical modeling and microbenchmarking. In this assignment, students need to provide an analytical model of their matrix multiplication versions - sequential and parallel, calibrate these models using microbenchmarking, and evaluate the models against measured performance data. We further add a second kernel to the mix, where we ask students to also model basic histogram

calculation, aiming to add data-dependent behavior as additional modeling challenge. The students can reuse their code and/or provided code for both kernels.

The goal for this assignment is threefold: we want students (1) to observe and understand the levels of granularity in analytical models, and the additional calibration challenges that come with those, (2) to get familiar with microbenchmarking as a model calibration tool, and (3) expose students to models that, although potentially very inaccurate, can still provide important insight into the performance of a given application. The students learn by trial and error to find the right level of granularity (ranging from coarse, at function level, to very fine, at ASM instruction level) where the kernels can be accurately model, and understand the difference in the details provided by these different models.

This assignment also provides students the opportunity to use the tabulated performance data for different processors [1], or different microbenchmarking tools, like STREAM or uops. They can further get familiar with detailed performance profilers like perf, NVIDIA's nvprof/nsight, as well as with instruction scheduler simulators like IACA, OSACA, or LLVM-MCA.

Assignment 3: Statistical modeling. In this assignment, students must demonstrate they can work around the limitations of analytical modeling by using machine-learning models. To this end, they are required to collect performance data from a relevant set of inputs, and model - using statistical methods - the expected performance. Finally, they must evaluate the prediction accuracy of the proposed model. To do so, they revisit matrix multiplication, and are provided with a new kernel: sparse matrix-vector multiplication (SPMV). We provide three versions of the kernel - based on the three classical storage models, CSR, CSC, and COO - and request them to model the sequential version, one parallel version of their choice, and compare the results against an analytical model.

With this assignments, we aim to showcase the challenges of defining and collecting training data, of feature engineering, and of empirical validation for such models. We further showcase the interpretability of the models by comparison, by exposing students to two extremes: the highly-explainable analytical model vs. the black-box statistical models.

Depending on the types of data they choose to collect, this assignment can further expose students to tools for collecting performance counter data and/or other detailed performance indicators. However, these are separately introduced and demonstrated in assignment 4.

Assignment 4: Performance counters and performance patterns. This final assignment goes into further detail into collecting detailed performance data for one of the provided kernels. We opted for SpMV, but the same exercise can be applied for matrix multiplication. Furthermore, to better illustrate the use of performance counters in the context of performance anomalies hypotheses, we introduce the concept of performance patterns (inspired by [7]) and encourage students to understand the correlation of performance patterns and observed counters values. For the latter part, we ask students to develop a simple (synthetic) kernel to demonstrate some of this performance patterns, and show they can be identified and fixed using performance counters data.

One important aspect of this assignment is to expose students to clear peformance patterns that often appear in HPC applications and algorithms, and teach them how they can identify them with simple empirical tools. Furthermore, in this assignment, students also get familiar with tools like Perf, PAPI, and LIKWID, Intel VTune, as well as nvprof amd nsight.

4.2.1 Coverage and timeline. The current assignments cover all technical skills for performance analysis, modeling, and prediction. We note that we do not cover well the modeling and analysis of distributed systems (where we discuss and demonstrate tools like Vampir, Score-P, and the Scalasca approach), but we found that we have insufficient time in this format to also cover these tools/approaches into an actual assignment.

We provide assignments somewhat sequentially: first assignment 1 (2 weeks deadline), followed by assignment 2 (2 weeks deadline, some days overlapping with assignment 1), and followed by both assignments 3 and 4, which we release at the same time, and the deadline is the end of the 7-weeks course period (effectively, students have 3 weeks for both assignments). This approach allows students to mix and match different parts of the assignments, often guided by what they need for their project.

We also provide 2-3 in-person lab sessions (with TAs support) to help students complete each of these assignments. Lab sessions focus on methods and tools demonstration, using simpler/different kernels than those from the assignments, but allowing students to observe how the tools are being used in practice.

4.3 Seminars and the Project

To support the development of the project, we kick-off the project in week 1, and provide the following timeline for students to plan their work:

- Week 1 Brief introduction of the project goals and several examples at high-level (dedicated seminar). We focus on describing the goals of the project, and helping students understand what a performance challenge could be for a different application.
- Week 2 Students work on providing a prototype of the sequential/reference version. We note that this version can be based on their own code or any open-source code they find useful as reference implementation.
- Week 3 Students must define their evaluation strategy and define an experimental setup (dedicated seminar). We focus here on the right infrastructure, metrics, and measurement methods needed for the assessment of current and improved performance of the target application.
- Week 4 Students should provide a first performance model of the code, and potentially apply the first optimizations. This leads to creating more prototypes starting from the reference code.
- Week 5 Students have a skeleton of their report and provide a short (5-minute) talk to introduce their application, status and goals, and first model.
- Weeks 6-7 Students provide more prototypes and a full performance engineering process, evaluating each prototype and motivating (through models) the following prototype(s). These

weeks focus on performace analysis, modeling, and improvement.

Week 8 : Students finalize project report and presentation, and reflect on whether and how they managed to meet the performance requirements.

For weeks 2, 4-6 we schedule invited talks from previous students and/or performance engineering experts that discuss their approach (and successes) in applying performance engineering. The remainder of the seminar time is allocated to consultation and discussion about the challenges of different projects.

4.4 Grading

The student grades are calculated as a weighted average using the grade for the exam, the assignments, and the project, and using in-class quizzes as bonus points. Currently, we use the formula from equation 1, where G_P is the project grade, G_A is the assignments grade, G_E is the exam grade, and S_Q is the cumulated score for in-class assignments.

$$G = max(10, 0.5 \times G_P + 0.3 \times G_A + 0.3 \times (G_E + S_Q/70)) \quad (1)$$

Note that our grading scheme clearly rewards the most important part of the course, the project, but it also provides some slack to enable students to excel in either the theoretical or the practical aspects of the course (i.e., exam or assignments, respectively), and rewards them for in-class participation through quizzes.

The exam grade is a simple sum of the points achieved for correct answers in the exam. This grade is augmented

To calculate the project grade, we combine the grade for the project itself (i.e., the application of performance engineering methods and tools to the application at hand), G_P^p, the report, G_P^r, and the presentations (midterm and final, averaged), G_P^t, as follows:

$$G_P = 0.4 \times G_P^p + 0.3 \times G_P^r + 0.3 \times G_P^t \quad (2)$$

For the assignments, we use a points system that enables students to work in different groups. Each assignment has an allocated number of points (i.e., 10p, 9p, 11p, 12p for assignments 1,2,3,4, respectively) and the final grade is calculated as the sum of the assignment points, G_A^i, $1 \leq i \leq 4$, divided by a factor that accounts for the number of students per team (we remind the reader the assignments and project are to be executed in teams of 1-4 students). Specifically, the calculation is described in equation 3.

$$.G_A = \sum_{1 \leq i \leq 4} G_A^i / N \quad (3)$$

$$N = \begin{cases} 32 & \text{for 1 student} \\ 36 & \text{for 2 students} \\ 40 & \text{for 3-4 students} \end{cases} \quad (4)$$

5 EVALUATION

In this section we reflect on the student performance and the feedback we have received for this course [5]

[5] A more detailed analysys of the students evaluation will be included in the final camera-ready version of the paper, once permission is granted by the university (due to potential GDPR issues).

5.1 Students' performance

The total number of students who followed the course in five years is around 85. We started with about 100-110, but, due to the high workload and advanced content, about 15-25% drop out in every edition of the course. In the following paragraphs, we present a few insights on the progress of the students in this course, based on their grades.

The average grade for the students passing the course is 8. This is higher than other courses due to the many different ways to achieve points. In fact, we find that students that finalize the course also get very good results, while those who would usually struggle fail and drop out early on. This is also a side-effect of the relative low impact of the final exam grade, which is often used as a discriminating factor between good and very good student performance, and rarely as a pass/fail discriminator.

The average assignments grade is around 8. Given the clear nature of the assignments, and the extensive support we provide in terms of tools, demonstration, and feedback, students provide very good solutions for the assignments. Traditionally, reports are of somewhat lower quality (often due to time pressure). To imrpove this aspect, we provided reporting templates. which have helped raising the average grade in the last couple of years.

The average exam grade for the students is around 7.5 - this is due to the difficult nature of the exam, and the fact that we cover more theoretical aspects of performance engineering. These are also aspects that are not practiced extensively during the labs. We find that such exams are very good to discriminate individual talent/preparation within the teams which, otherwise, will receive the same grade. We observe that in-class quizzes clearly help with good performance in the exam, as students often understand where and how to emphasize on specific topics. However, we do admit these quizzes may take a long time to create and grade, and more detailed feedback would help even further with the comprehension of the more theoretical aspects of the course.

For projects, the average grade is 8. This somewhat expected, given that we follow closely their progress, and we provide many feedback and consultation moments. The grades are not higher due to time pressure and, often, due to challenging priorities and project management within the team(s). However, all projects we have seen have been successful to a certain degree, with grades ranging from 6.5 (where parallelism was not correctly applied) to 10 (for work that we still use as example when we kick-off the project every year).

On a side note, we find it interesting to provide insight into the projects students have chosen to complete in this course. Recurring projects are, in the decreasing order of popularity: 2D stencil code optimization (due to an assignment from a previous course), game of life (also an assignment in a previous course), graph processing (due to one of the recurring invited lectures). However, we have also seen exotic applications, like content generation for games (RushHour), game optimization (EveOnline), solving Wordle, or FFT optimizations.

5.2 Students' feedback

Overall, students grade the course between 8 and 9 (out of 10) every year, including during the pandemic. This is exceptionally

high for courses in the Dutch academic system. They consistently grade the acquired knowledge, applicability, and practical relevance above 90%. Students appreciate the connection between the course, assignments, and project, and appreciate the ability to work on their own application. They further enjoy the final project presentations, where they not only get to brag about their own results, but also learn about new applications and new methods/tools to analyse and improve these applications.

On the downside, students are more critical of the high workload of the course: they claim to spend 20-50% more time than officially allocate for the course. To alleviate this problem, we created lab templates (to allow the reporting to be more efficient) and provided more flexible deadlines for the different assignments. We also further emphasized the importance of automation in their empirical analysis (from data collection to plotting), which seems to be the most time consuming aspect of the assignments and/or project.

6 LESSONS LEARNED

Before concluding this paper, we summarize our insights after these five years of teaching in a set of six lessons learned. They are:

Lesson 1 Performance Engineering is appealing when treated like a puzzle. We appeal to students' curiosity to understand why applications behave weirdly on different systems. To allow for this, one needs to prepare topics and assignments that showcase such behavior.

Lesson 2 Provide both methods and tools for each part of the course. Students appreciate the theory much better when they can link it to concrete examples. Furthermore, this mix enables them to build an intuition about how to navigate the performance engineering process and, in the long term, how to further improve it.

Lesson 3 Do not underestimate empirical analysis efforts. We often notice that students spend a lot of time in empirical analysis. This is often the case when experimental design is missing, and/or automation is not properly defined. We spend time and provide many examples on how this should be done, to allow them to build such automated setups correctly and efficiently.

Lesson 4 Projects stimulate creativity, and students should be allowed exploration time and space. As such, we provide no end-line for our projects. Instead, we want them to try different things and report, after critical reflection, on their findings.

Lesson 5 Stimulate critical thinking to reporting on both positive and negative results. There are no negative results for our projects/assignments: we grade the process and the actual insights, and not the ultimate speed-up or high-accuracy models students achieved. Understanding why and how methods and tools work and fail is fundamental to this course.

Lesson 6 This is an intensive course for both teachers and students. Keeping the material up-to-date, adding new tools and infrastructure, eliminated deprecated content are difficult when preparing the course. However, we are proud to offer something that students can rely on and immediately apply for their next performance engineering project in real-life.

7 CONCLUSION

As HPC and (super)computing systems increase in diversity, more effort is needed to keep improving the performance and efficiency of applications for newer systems. In turn, this means more specialists are needed to directly work with domain experts for HPC applications, but also to design and build better tools to facilitate this process. To create these specialists, we need our students to be aware, able, and willing to contribute to performance engineering. We strongly believe this can only happen if students are properly trained in the systematic nature of the performance engineering process.

To facilitate this education, we designed and implemented the first (and only) graduate-level course on performance engineering. Our course combines lectures, assignments, and a project to combine theoretical, methodological, and practical aspects (and tools) to demonstrate and practice the full performance engineering process. After teaching the course for five years, we can observe that such a course (1) is feasible for graduate-level students with basic knowledge of computer architecture and programming, (2) provides them with a deeper understanding and practical, applicable knowledge of efficient systems and application design, and (3) improves their ability to communicate with different stakeholders, including applications' end-users and system designers. Students' feedback indicates the course is challenging, workload-wise, but interesting in its open approach to assignments and project. The course has been evaluated among the top courses in the program, and it has lead to 10-25% of the students in each year's cohort to be interested in taking on a performance engineering project for their MSc degree.

To further improve the course, we identified one main challenge to be tackled and three topics to be further developed. The main challenge facing this course is the continuous updating of the material: students appreciate it because it is up-to-date with the current tools and practices from the field, yet this often requires yearly efforts to test and update software, libraries, and tools. This can be extra-challenging when the course relies on shared machines, with very limited user rights. In terms of topics to be improved, we list the three critical ones: (1) supporting various vendors hardware, (2) including additional metrics - such as energy-efficiency - more prominently, and (3) expanding the distributed computing aspect of the course, potentially extending towards shared systems like cloud computing/the computing continuum and the use of VMs and/or containers.

REFERENCES

[1] Fog, A., et al. Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs. *Copenhagen University College of Engineering 93* (2011), 110.

[2] John, L. K., and (Eds.), L. E. *Performance Evaluation and Benchmarking (1st ed.)*. CRC Press, 2006.

[3] McCalpin, J. D. Stream: Sustainable memory bandwidth in high performance computers. Tech. rep., University of Virginia, Charlottesville, Virginia, 1991-2007. A continually updated technical report. http://www.cs.virginia.edu/stream/.

[4] McCalpin, J. D. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter* (Dec. 1995), 19–25.

[5] Seiferth, J., Alappat, C., Korch, M., and Rauber, T. Applicability of the ecm performance model to explicit ode methods on current multi-core processors. In *High Performance Computing* (Cham, 2018), R. Yokota, M. Weiland, D. Keyes, and C. Trinitis, Eds., Springer International Publishing, pp. 163–183.

[6] Smith, C. *Performance Evaluation – Stories and Perspectives - Chapter 16: Software performance engineering*. Austrian Computer Society, 2003.

[7] Treibig, J., Hager, G., and Wellein, G. Performance patterns and hardware metrics on modern multicore processors: Best practices for performance engineering. In *Euro-Par Workshops* (2012).

[8] University of Amsterdam. Course catalogue (2022-2023). https://studiegids.uva.nl/xmlpages/page/2022-2023-en/search-programme/programme/7283/251111.

[9] Varbanescu, A.-L., and Swatman, S. Performance engineering course repository. https://github.com/alvarbanescu/PerfEngCourse.

[10] Williams, S., Waterman, A., and Patterson, D. Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM 52*, 4 (apr 2009), 65–76.

[11] Wong, H., Papadopoulou, M.-M., Sadooghi-Alvandi, M., and Moshovos, A. Demystifying gpu microarchitecture through microbenchmarking. In *2010 IEEE International Symposium on Performance Analysis of Systems Software (ISPASS)* (2010), pp. 235–246.

8 ARTIFACT INFORMATION

In accordance with the Supercomputing Transparency and Reproducibility Initiative, this section provides additional detail about the artifacts relevant for this manuscript.

8.1 Summary

Our paper presents our experience with teaching performance engineering to graduate students. To this end, all the proposed assignments are available for assessment as provided to the students.

8.2 Artifact Description

We propose four assignments for this course.

8.3 Artifact Availability

All data and materials for the proposed assignments are available on GitHub for our students, and can be accessed by the evaluators [9].

8.4 Experimental Setup

Our assignments require a heterogeneous system, using a CPU and NVIDIA GPU. We have used both Intel and AMD CPUs, although the tools we recommend are Intel-specific. We have use GPUs of compute capability between 3.0 and 7.2. We have not tried AMD GPUs, due to limited hardware access and available TA support.

The aforementioned software and hardware were not altered in any way in our work: our code focuses solely on providing the seed the students need to use for their development.

8.5 Artifact Evaluation

The goal of the provided software is to provide a starting point for the assignments. To this end, all our code can be compiler with off-the-shelf C/C++ compilers. All the provided code is developed by teachers and/or TAs, or uses open-source code available online (e.g., for reading matrices in the matrix market format).