Pattern-Driven Node-Level Performance Engineering

Jan Treibig, Georg Hager and Gerhard Wellein
Erlangen Regional Computing Center (RRZE)
Friedrich-Alexander-Universität Erlangen-Nürnberg
Martensstrasse 1, 91058 Erlangen, Germany
Email: jan.treibig@rrze.fau.de

I. Introduction

Performance engineering has been established as an integral part of software engineering. However, in many cases the often indeterministic and chaotic approaches used in what was formerly called "performance optimization" are transfered without change. There is a strong belief that performance analysis and optimization is too complex to be put into a systematic process.

Therefore highly skilled performance experts using available profiling tools are a common solutions to this problem. To ease this situation there are also efforts to provide complex automatic performance tools. We previously proposed a systematic performance engineering process constructed around the formulation of a diagnostic performance model. The performance model is a tool to force the developer to acquire knowledge about the interaction of his code with a specific system architecture. In the present work we want to further improve this process and combine it with the notion of performance patterns introduced in [1]. Performance patterns give the developer a concrete plan how to proceed to understand the observed performance, how to formulate a suitable model, and decide on optimizations. Patterns provide common knowledge and condense experience in an accessible way. They help in understanding what the limiting factors of a code are, which quantitative model is suitable to describe the performance, and finally which optimization techniques are available to improve performance. In the following we introduce a resource driven view on computer architecture which provides the right level of abstraction in order to get an intuitive access to understand observed performance and allows to formulate an adequate performance model. We present examples for common performance patterns and how they can be detected. Finally we show the application of our performance engineering process on several examples.

II. A resource driven view on computer architecture

We start with a simplified view on which relevant aspects of the hardware are necessary to describe the performance of an application code running on a computer. The two elementary resources offered by any compute device are:

- Instruction execution as the primary resource with "instructions" as the unit of work
- Data transfer as the secondary resource induced by instruction execution, with data volume as the unit of work

There are two basic ways to look at performance in a processor design. In computer architecture the common case is the "core-centric" view, which sees everything in terms of waiting times. While this may make sense in order to understand timing issues and carry out a critical path analysis, for our purpose such a view has several weaknesses. What is "good enough" or which performance is optimal is difficult to answer in an intuitive way. Hence, we propose a view which is centered on resource utilization. By connecting the amount of useful work to the runtime of the code we get a comparable performance unit which allows to easily see the efficiency of a code in terms of resource utilization. In a data transfer limited setting the *bandwidth* is the resource used by the code, i.e., the ability to transfer data in a certain amount of time. Consequently, even latency can be seen as a technical limitation or hazard which as a consequence leads to a reduced bandwidth.

While the above view already covers the basic correlations in code-hardware interaction, processors are highly complex machines exposing many other technical limitations and pitfalls. Still those all can be seen as *hazards* preventing full use of the basic resources, instruction throughput and data transfer. In our opinion it is important to always put the utilization of the basic resources in the middle of an analysis and ensure efficient utilization by identifying and avoiding hazards. This can be also seen in a hardware-software co-design way, giving an indication how feasible it is for software to make full use of resources offered by a processor design and what the optimal sustained performance is.

III. Performance Patterns and Signatures

In [1] we introduced the term "performance pattern" as a common performance-limiting setting. The diagnosis of a specific performance pattern is supported by a so-called "signature", which is a set of indicators that accompany a pattern. A signature can be, e.g., a characteristic performance behavior, a specific manifestation of hardware performance monitoring measurements, or the comparison to micro-benchmarking experiments. The advantage of using patterns is that previous experience is categorized to provide common approaches for understanding and improving the performance of a code matching a pattern.

The patterns any optimization effort aims for are patterns which are directly related to the full utilization of available resources. The most prominent example here is bandwidth saturation of a shared data path. The second class of patterns are governed by technical limitations of the hardware. This may be issues as false cache line sharing or low spatial data access locality. Finally a third class of patterns is related to the notion of work in our architecture view. This covers the total amount of work in terms of necessary instructions. A common manifestation of this pattern is a bad ratio between useful versus overhead instructions, a situation common in applications implemented in object oriented languages like C++. A further example is the frequent use of long-latency instructions like divide or exponential. Another class of patterns is rooted in the limitations of concurrent execution. The influence of a serial fraction in parallel execution described by Amdahls' law is an elementary property governing the scaling behavior of a given code. Other patterns here are synchronization overhead and load imbalance as elementary issues preventing good scaling. Also the improvement of temporal data access locality with a better benefit of the cache hierarchy can be seen as a reduction of work for the processor in terms of reduced data volume over a slow data path.

A pattern is therefore a specific performance-limiting manifestation which is connected to a phenomenological performance behavior and can be detected by a set of indicators (its signature).

IV. Performance Models

A pattern is a *qualitative* classification. To evaluate the performance of a specific code it is necessary to get a *quantitative* view. This is achieved by using the pattern as a motivation to formulate a diagnostic performance model. The model makes it possible to compare a certain view of limitations and interaction between code and machine with real benchmarking results. Constructing and refining a model improves the confidence in the qualitative view. A model also allows a-priori estimates for the effect of optimizations or porting to different compute devices and thereby enables a thorough decision process based on knowledge instead of guessing and common myths. The most popular model in this context is the roofline model [2], which is a direct representation of the resource-driven view introduced in Section II. The main limitation of the roofline model is its focus on one data path ("the bottleneck"). Moreover, the model only works correctly if perfect overlap of execution and data transfer can be assumed and the data path bandwidth is fully saturated. A more general approach related to the roofline model is the Execution-Cache-Memory (ECM) model introduced in [3]. The ECM model can also be used if there is no well-defined "slowest" data path, or if the available bandwidth is not fully saturated. The model is also capable of predicting the saturation point in a multicore scaling context.

V. Performance Engineering Process

Our proposed performance engineering process starts with a runtime profiling of the application in order to find hot spots in the code. The rest of the process is carried out for each homogeneous hot spot. This may be, e.g., a nested loop. After identifying the basic blocks for optimization a thorough code review is performed in order to get a basic understanding of what the code is doing and already get insights such as consumed data volume, data structure requirements, and many more things which are hard to detect for an automated black-box-like tool. Other optimization strategies applicable in this stage are work and memory usage reduction. At this stage a performance model describing the code hardware interaction is formulated. The model is validated against performance measurements. If there is a discrepancy between model and measurement applying patterns help to refine and improve the model. If the model is correct patterns help to find possible optimization opportunities. The next step is to determine which patterns apply for the code block under consideration. The basic procedure is very similar to a criminal investigation: The analysis starts with suspicions that a certain pattern applies based on the static code review. To support the suspicion, indicators or proofs are searched and collected, i.e., the developer tries to collect evidence in order to confirm a signature. Such a signature can consist of parallel scaling runs, hardware performance counter measurements, comparing performance to microbenchmark data, or scaling the input data size. By collecting data new facts might influence the matching patterns. Due to the overall complexity, the process of getting confidence in a judgment about the code performance is a crucial element. The patterns give the developer a frame of reference they can start with. Still during execution of the process the developer gains their own domain specific knowledge which they can transfer to other optimization tasks. This makes such a "human-driven" approach much more powerful, flexible, portable, and sustainable than a tooling-only solution.

VI. Practical Examples

We first introduce the ECM model on the example of a simple streaming code, and then elaborate on how erratic access patterns can be accommodated in sparse matrix-vector multiply (spMVM). A more complex example (3D 15-point stencil from a MG solver) shows how naive assumptions about code execution can lead to false conclusions. Finally, a medical imaging application is used to demonstrate how the performance engineering process is iteratively applied and how different patterns are tried and partly discarded in its course, thereby learning more and more about the limitations of the code.

References

[1] J. Treibig, G. Hager, and G. Wellein, "Performance patterns and hardware metrics on modern multicore processors: Best practices for performance engineering." in *Euro-Par Workshops*, ser. Lecture Notes in Computer Science, I. Caragiannis, M. Alexander, R. M. Badia, M. Cannataro, A. Costan, M. Danelutto, F. Desprez, B. Krammer, J. Sahuquillo, S. L. Scott, and J. Weidendorfer, Eds., vol. 7640. Springer, 2012, pp. 451–460.

[2] S. W. Williams, A. Waterman, and D. A. Patterson, "Roofline: An insightful visual performance model for floating-point programs and multicore architectures," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2008-134, Oct 2008. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-134.html

[3] J. Treibig and G. Hager, "Introducing a performance model for bandwidth-limited loop kernels," in *Parallel Processing and Applied Mathematics*, ser. Lecture Notes in Computer Science, R. Wyrzykowski, J. Dongarra, K. Karczewski, and J. Wasniewski, Eds. Springer Berlin / Heidelberg, 2010, vol. 6067, pp. 615–624.