A Generic Profiling Infrastructure for the Hyperbolic PDE Solver Engine ExaHyPE

Master's Thesis in Computational Science and Engineering

Fabian Güra
Department of Informatics
Technische Universität München

September 26, 2016

Examiners: Univ.-Prof. Dr. Michael Bader
Dr. Tobias Weinzierl, Lecturer
Supervisor: Dr. Vasco Varduhn

I hereby declare that this thesis is entirely the result of my own work except where otherwise indicated. I have only used the resources given in the list of references.

Place: _____ Date: _____ Signature: _____

Abstract

The ExaHyPE project is tasked with tackling the software side challenges of exascale high performance computing. In order to optimize and evaluate the novel algorithmic and numeric approaches developed as part of the project, metrics correlated with performance and energy efficiency need to be obtained. In this thesis we therefore discuss design goals, architecture and exemplary implementations of a generic profiling infrastructure for the hyperbolic PDE solver engine. Against the background of hyperbolic balance laws and a state of the art limited ADER-DG scheme we present preliminary profiling results to illustrate the employed techniques. In two case studies we demonstrate how the generic profiling infrastructure can be used to quickly track down the source of possible issues in a simulation without changing a single line of code and how it provides guidance for optimization efforts within the scope of metrics-driven performance engineering.

Keywords: Arbitrary high-order Discontinuous Galerkin scheme, a posteriori subcell finite volume limiting, ADER-DG, MUSCL-Hancock, hyperbolic balance laws, high performance computing, HPC, exascale, ExaHyPE, profiling, performance, energy, x86, performance counters, RAPL

Contents

Chapter 1

Introduction

Over the years scientific computation has established itself as a major driver for progress in numerous disciplines and has become an equal constituent next to traditional approaches based on theory and real-world experiments. The increasing capabilities of scientific simulations are inherently linked to incremental and disruptive innovations in high performance computing. In many scenarios HPC can be used to augment or replace time-consuming manual derivations and prohibitively expensive or even hazardous experiments [1]. It is well-documented that applications made possible by HPC have furthermore generated insight that could not have been obtained otherwise. Future HPC systems will enable even larger and more detailed simulations; in areas such as astrophysics, particle physics or neuroscience results obtained in this way may simply not be observable in traditional experiments [2]. Figure 1.1 visualizes the exponential growth in performance of the 500 fastest supercomputers listed on the accordingly named Top500 list [3]. Even though a trend towards slight prolongation of the time intervals after which systems at the very top are replaced by a new performance leaders might be observed, it gives rise to the commonly stated prediction that there will be an exascale supercomputer by 2020, i.e. a single machine will be able to perform 10^{18} floating-point arithmetic operations per second. Without doubt one of the greatest challenges on the way towards this "new era of computing" [4] is energy consumption. Already today it is common practice to underprovision power supply and cooling for data centers and supercomputing facilities [5], i.e. to design them with respect to a worst-case average rather than absolute peak power consumption.

While the advancement of performance in HPC systems has traditionally been considered to be a challenge mostly with respect to hardware, in recent years it became obvious that also software for current and to an even greater degree for future supercomputers needs to be developed with architecture-specific performance optimization and energy efficiency in mind [6]. Against the background of the promises and challenges of exascale computing appearing on the horizon as well as in view of the great importance of hyperbolic balance laws in almost all areas of science and engineering, the ExaHyPE project in essence is concerned with creating an easy to use engine that will allow application domain experts to run their simulations at sublime performance and efficiency to enable scientific progress at a scale at which it will be of significant benefit for the whole of society. In order to allow for systematic optimization of the novel algorithmic and mathematical approaches developed as part of the project it is necessary to ob-

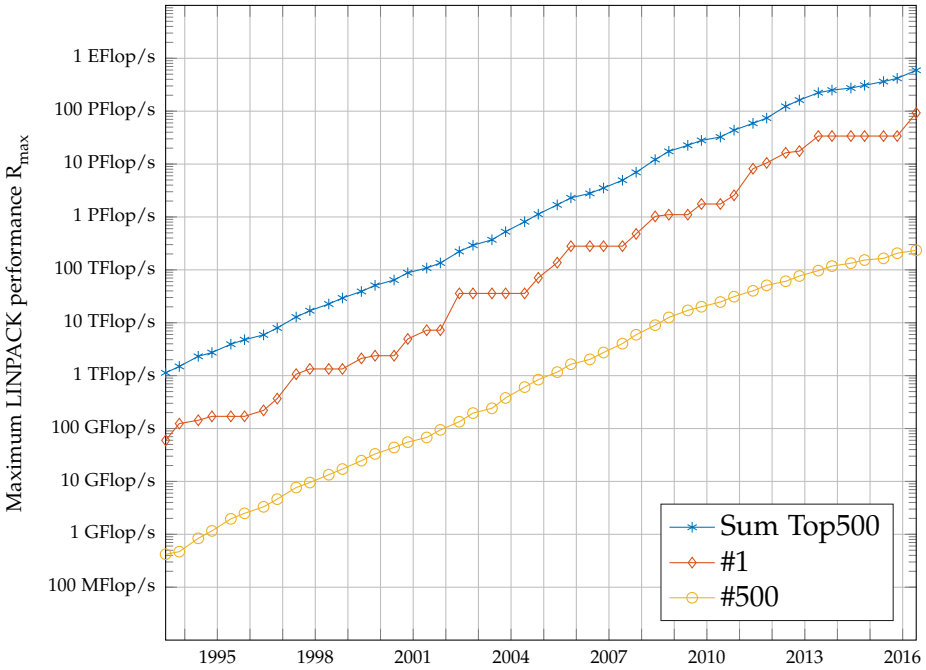


Figure 1.1: Performance development over time of the supercomputing systems on the TOP500 list [7] with respect to peak performance achieved in the LINPACK benchmark [8]. The figure clearly indicates exponential growth of the accumulated performance of all listed systems, but also illustrates that the time intervals between the emergence of a new performance leader tend to prolongate.

tain metrics correlated with performance and energy efficiency. In this thesis we therefore present a generic profiling infrastructure for ExaHyPE that can besides the more traditional metrics for performance in HPC such as floating-point operations per second (FLOP/s) and time to solution also provide metrics indicating the overall energy consumption of the simulation. We present implementations based on on-chip measurement units ("performance counters") to illustrate the applied techniques and their usefulness in practice; the generic nature of the proposed interface, however, allows for flexible extension in the future.

In summary the thesis is structured as follows: The first two chapters are dedicated to considerations on the theoretical context of the presented work: In chapter 2 the complete state of the art scheme ("Arbitrary High Order Derivatives Discontinuous Galerkin (ADER-DG) method with a posteriori MUSCL-Hancock subcell limiting") for solving hyperbolic balance laws embedded into ExaHyPE is derived for an arbitrary number of spatial dimensions and simplified to a point where it can directly be mapped to common programming languages. Appendix A lists code that can be used to precompute the necessary discrete operator matrices. In chapter 3 we then focus on theoretical aspects of the x86 instruction set architecture, its current prevalence in HPC and – most prominently – ways to obtain performance and energy metrics using performance monitoring units available on modern x86 microprocessors. An extensive literature review on the different approaches, techniques and libraries with respect to availability, ease of use, accuracy, potential pitfalls and limitations is given. Moving on to chapter 4 we discuss in great detail the ExaHyPE project (chapter 4.1) and the generic profiling

infrastructure for the engine as the major contribution covered by this thesis (chapter 4.2). We present four exemplary implementations of the latter and give ideas and guidance for future work. The subsequent chapter 5 consists of two parts: We first give some preliminary profiling results for the unoptimized ExaHyPE ADER-DG kernels to illustrate the implemented techniques and then demonstrate in two case studies how they can be used in practice to a), track down the source of potential issues in a simulation without changing a single line of code, and b), guide optimization efforts in the context of metrics-driven performance engineering. We conclude the thesis in chapter 6 with remarks on where the ExaHyPE engine and the presented profiling infrastructure stand with respect to the challenges of exascale computing and give an outlook on promising topics for future work.

Chapter 2

A D-dimensional ADER-DG Scheme With MUSCL-Hancock a Posteriori Subcell Limiting for Hyperbolic Balance Laws

2.1 Introduction

As stated already in the introduction, the ExaHyPE project is concerned with building an engine for simulating problems that can be formulated in terms of a hyperbolic balance law (HBL). Leaving all of the high performance computing (HPC) components aside, the very heart of the engine therefore comprises of an embedded numerical scheme for solving partial differential equations (PDEs) of HBL type.

Solving PDEs of this kind is of great interest in practice and has been a topic of active research for about a century[1]. A comprehensive overview on schemes that have been proposed over the years can be found in [10]. Two particularly challenging aspects of such simulations are

1. the accurate simulation of HBL problems over long periods of time, especially when facing (prescribed or spontaneously arising) discontinuities ("shocks") or stiff source terms leading to stability issues in space and time, respectively, and,
2. the inherent data access patterns the numerical schemes inherit from the PDE that make it challenging to avoid excessive communication and to achieve high arithmetic density, two key drivers for good performance on modern distributed HPC systems.

In ExaHyPE a state of the art method called Arbitrary High Order Derivatives Discontinuous Galerkin (ADER-DG) is employed together with a posteriori subcell limiting based on the robust, second-order MUSCL-Hancock finite volume method (FVM) scheme (see [11] for details). As the name implies, the Discontinuous Galerkin framework with high-order local polynomials ansatz functions acts

[1]The British scientist and Durham University alumnus Lewis Fry Richardson is considered to be one of the founding fathers of computational fluid dynamics (CFD). In 1922 he published a book in which he presented a method for weather forecasting based on the solution of differential equations, in a time when most computations were still done by human "computers." See [9] for a second edition of the book published in 2007 which puts the work into a contemporary context and emphasizes how modern weather forecasting is still based on Richardson's ideas.

as the theoretical foundation of the method (see [12, 13] for more details on the underlying theory). In the following sections we will step by step derive the complete scheme and emphasize how it addresses the challenges stated above. The first part on unlimited ADER-DG is based to varying degree on work presented in [14] with three important additional contributions:

1. The scheme is presented in a more general form for systems involving V quantities in D spatial dimensions.
2. We employ index notation and simplify the equations up to a point where the resulting mathematical formulae can easily be mapped to a computer programming language, in particular it is in direct agreement with FORTRAN code used by Dumbser et al. to generate the numerical results presented for example in [15, 16, 17, 18, 19].
3. We extend the formulation to include a posteriori subcell limiting, introduce projection and reconstruction operations to equidistant subgrids and review the MUSCL-Hancock FVM scheme.

The chapter is set up as follows: First, we will begin with some remarks on the notation employed and state the problem at hand in its general form. Second, we will derive an element-local weak formulation and approximate it with respect to finite-dimensional function spaces. After introducing reference coordinates, corresponding mappings and orthogonal bases for the function spaces involved, we employ a predictor-corrector approach to arrive at a fully-discrete method that is of arbitrarily high order in both space and time.

2.2 Notation

Before we begin deriving the numerical schemes let us quickly introduce the following set of rules on how to depict common mathematical objects and operations:

- Vectors, i.e. first-order tensors, and vector-valued functions will be denoted by bold, lower case letters, e.g. $\boldsymbol{x}$, $\boldsymbol{u}(\boldsymbol{x}, t)$.
- Matrices, i.e. second-order tensors, or matrix-valued functions will be denoted by bold, upper case letters, e.g. $\boldsymbol{K}$, $\boldsymbol{F}(\boldsymbol{x}, t)$.
- Higher order tensors are always denoted as bold, lower-case letters with a "hat" on top, e.g. $\hat{\boldsymbol{u}}^{K,i}$, $\hat{\boldsymbol{q}}^{K,i}$. Note, however, that the opposite is not true[2].
- To avoid confusion in case we deal with tensors for which a superscript or a subscript is "part of its name" or where this indicates membership in a set, similar to the convention in many programming languages we denote "accesses" into the tensor by square brackets. An example illustrating the advantage of this notation based on common naming conventions in literature could be the following: Let $\{\tilde{\boldsymbol{u}}_h^{K,i}\}_{K\in\mathcal{K}_h, i\in\{0,1,\dots,I-1\}}$ be the set of vector-valued functions from an appropriate Hilbert space which are defined locally on a cell $K \in \mathcal{K}_h$ and in a time interval $[t_i, t_i + \Delta t_i]$, $i \in \{0, 1, \dots, I-1\}$. If we now want to access the v-th component of $\tilde{\boldsymbol{u}}_h^{K,i}$, we write

$$\left[\tilde{\boldsymbol{u}}_h^{K,i}\right]_v \tag{2.1}$$

[2] In general the dimensionality of an object will always be obvious from its indices. Since we will just allow scalar and vector-valued indices only this distinction is of critical importance.

instead of

$$\tilde{\boldsymbol{u}}^{K,i}_{h,v}. \tag{2.2}$$

In this way it is absolutely clear that K, i and h are "part of the name" and that v is an index used to access an element in the tensor. However in case an expression is absolutely unambiguous, for the sake of brevity we will often omit the square brackets. Most prominently we write

$$\frac{\partial}{\partial \boldsymbol{x}_d} \tag{2.3}$$

instead of

$$\frac{\partial}{\partial \left[\boldsymbol{x}\right]_d}. \tag{2.4}$$

- Throughout the thesis we use index notation following the Einstein summation convention whenever possible. This means that if an index within an expression is repeated exactly once, this implies summation over the whole value range of this index. The standard inner product of two vectors $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^N$ can then be written as

$$\left\langle \boldsymbol{x}, \boldsymbol{y} \right\rangle = \sum_{n=0}^{N-1} \left[\boldsymbol{x}\right]_n \left[\boldsymbol{y}\right]_n = \left[\boldsymbol{x}\right]_n \left[\boldsymbol{y}\right]_n (= \left[\boldsymbol{x}\right]_m \left[\boldsymbol{y}\right]_m) \tag{2.5}$$

and for unambiguous cases like above

$$\left\langle \boldsymbol{x}, \boldsymbol{y} \right\rangle = \sum_{n=0}^{N-1} \boldsymbol{x}_n \boldsymbol{y}_n := \boldsymbol{x}_n \boldsymbol{y}_n (= \boldsymbol{x}_m \boldsymbol{y}_m). \tag{2.6}$$

Such indices are called dummy indices in the sense that as illustrated in the example above it does not matter if the index is named n or m. Sometimes free indices, i.e. indices that are not dummy indices, appear twice in a term as a result of some algebraic manipulation. In these cases we will explicitly state that summation over the index is not intended unless it is absolutely obvious for example from the left-hand side of the equation that the index can only be a free index. We furthermore always give an explicit range for free indices. See [20] for more details on index notation, its advantages and limitations as well as a more formal definition.

In addition to increased brevity, index notation allows for less ambiguities compared to classical vector notation, simplifies derivation of identities from tensor calculus and – if done carefully – the resulting formulae can be mapped conveniently to loops in low-level programming languages such as C or FORTRAN.

- To keep all derivations dimension-agnostic we define accesses into tensors using vector indices as follows: Let $\hat{\boldsymbol{u}} \in \mathbb{R}^{I_0 \times I_1 \times \ldots \times I_{D-1}}$ be a tensor of order $D \in \mathbb{N}$ with $I_d \in \mathbb{N}_0$ for all $d \in \{0, 1, \ldots, D-1\} := \mathcal{D}$. Let furthermore $i_d \in \{0, 1, \ldots, I_d - 1\}$ for all $d \in \mathcal{D}$ and $\boldsymbol{i} \in \mathbb{N}_0^D$, $\left[\boldsymbol{i}\right]_d = i_d$ for $d \in \mathcal{D}$ a vector of indices. Then we define

$$\left[\hat{\boldsymbol{u}}\right]_{\boldsymbol{i}} = \left[\hat{\boldsymbol{u}}\right]_{[i_0, i_1, \ldots, i_{D-1}]} = \left[\hat{\boldsymbol{u}}\right]_{i_0, i_1, \ldots, i_{D-1}}. \tag{2.7}$$

If we only provide a vector of $D-1$ indices, i.e.

$$[\hat{\boldsymbol{u}}]_{[i_0,i_1,\ldots,i_{D-2}]}, \tag{2.8}$$

we obtain a vector of length I_{D-1}. If we only provide $D-2$ indices we obtain a matrix with I_{D-2} rows and I_{D-1} columns. In general if we provide $d \in \{0,1,\ldots,D\}$ indices we obtain a tensor of order $D-d$.

- In the style of numerical computing environments such as MATLAB or Octave [21] we define the following shorthand notation for sequences of consecutive integral numbers:

$$j{:}k := \begin{cases} \{j, j+1, \ldots, k\} & \text{if } j \leq k \\ \{\} & \text{otherwise.} \end{cases} \tag{2.9}$$

- We can now define access into a vector $\boldsymbol{x} \in \mathbb{R}^N$ of length N via sequences as

$$[\boldsymbol{x}]_{j:k} := \left[[\boldsymbol{x}]_j, [\boldsymbol{x}]_{j+1}, \ldots, [\boldsymbol{x}]_k \right] \tag{2.10}$$

for $j \leq k$ and $j,k \in 0{:}N-1$, which for unambiguous cases as above is equal to the definition

$$\boldsymbol{x}_{j:k} := [x_j, x_{j+1}, \ldots, x_k]. \tag{2.11}$$

Together with implicit set and vector concatenation we can then write

$$[\boldsymbol{x}]_{[0:k-1,k+1:N-1]} = \boldsymbol{x}_{[0:k-1,k+1:N-1]} \tag{2.12}$$

for $k \in 0:N-1$ to denote the vector of length $N-1$ that contains all values of the original vector $\boldsymbol{x}$ but the k-th component. Furthermore

$$\left[[\boldsymbol{x}]_{0:k-1}, x', [\boldsymbol{x}]_{k+1:N-1} \right] = \left[\boldsymbol{x}_{0:k-1}, x', \boldsymbol{x}_{k+1:N-1} \right] \tag{2.13}$$

denotes the vector of length N whose components are equal to the ones of $\boldsymbol{x}$ apart from the k-th one, which we have replaced by the scalar $x' \in \mathbb{R}$.

2.3 Hyperbolic Balance Laws

A D-dimensional balance law in a system with V quantities is described mathematically by a partial differential equation (PDE) of the form

$$\frac{\partial}{\partial t} \left[\boldsymbol{u}(\boldsymbol{x},t) \right]_v + \frac{\partial}{\partial \boldsymbol{x}_d} \left[\boldsymbol{F}\left(\boldsymbol{u}(\boldsymbol{x},t)\right) \right]_{vd} = \left[\boldsymbol{s}\left(\boldsymbol{u}(\boldsymbol{x},t)\right) \right]_v \text{ on } \boldsymbol{\Omega} \times [0,T] \tag{2.14}$$

together with initial conditions

$$\left[\boldsymbol{u}(\boldsymbol{x},0) \right]_v = \left[\boldsymbol{u}_0(\boldsymbol{x}) \right]_v \quad \forall \boldsymbol{x} \in \boldsymbol{\Omega}, \tag{2.15}$$

and boundary conditions

$$\left[\boldsymbol{u}(\boldsymbol{x},t) \right]_v = \left[\boldsymbol{u}_B(\boldsymbol{x},t) \right]_v \quad \forall \boldsymbol{x} \in \partial\boldsymbol{\Omega}, t \in [0,T], \tag{2.16}$$

for all $v \in \mathcal{V}$, where we define the index set $\mathcal{V} := \{1, 2, \ldots, V\}$. $[0, T]$ is the time interval of interest, $\boldsymbol{\Omega} \subset \mathbb{R}^D$ denotes the spatial domain and $\partial\boldsymbol{\Omega}$ its boundary. The function $\boldsymbol{F} : \mathbb{R}^V \to \mathbb{R}^{V \times D}, \boldsymbol{u} \mapsto \boldsymbol{F}(\boldsymbol{u}) = \left[\boldsymbol{f}_1(\boldsymbol{u}), \boldsymbol{f}_2(\boldsymbol{u}), \ldots, \boldsymbol{f}_D(\boldsymbol{u})\right]$ is called the flux function. For the problem to be hyperbolic we require that all Jacobian matrices $\boldsymbol{A}_d(\boldsymbol{u})$, $d \in \{0, 1, \ldots, D-1\} := \mathcal{D}$, defined as

$$[\boldsymbol{A}_d]_{ij} = \frac{\partial \left[\boldsymbol{f}_d\right]_i}{\partial \boldsymbol{u}_j}, \tag{2.17}$$

have V real eigenvalues in each admissible state $\boldsymbol{u} \in \mathbb{R}^V$.

2.4 Space and Time Discretization

Let $\mathcal{K}_h$ be a quadrilateral partition of $\boldsymbol{\Omega}$, i.e.

$$K \cap J = \emptyset \quad \forall\, K, J \in \mathcal{K}_h,\, K \neq J, \tag{2.18}$$

$$\bigcup_{K \in \mathcal{K}_h} K = \boldsymbol{\Omega}. \tag{2.19}$$

For the index set $\mathcal{I} := \{0, 1, \ldots, I-1\}$ let $\{t_i\}_{i \in \mathcal{I}}$ be an I-fold partition of the time interval $[0, T]$ such that

$$0 = t_0 < t_1 < \ldots < t_I = T. \tag{2.20}$$

For $i \in \mathcal{I}$ we furthermore define

$$\Delta t_i = t_{i+1} - t_i, \tag{2.21}$$

so that the subinterval $[t_i, t_{i+1}]$ can be written as $[t_i, t_i + \Delta t_i]$.

Without loss of generality we can solve the original PDE (2.14) on $\boldsymbol{\Omega} \times [0, T]$ simply by solving the PDE locally for each element $K \in \mathcal{K}_h$ in the time interval $[t_0, t_0 + \Delta t_0]$ and then proceeding to the next time interval until we have reached the final time T. This gives rise to an element-local formulation on a subinterval in time which we will focus on in the following.

2.5 Element-local Weak Formulation

Let $L^2(\boldsymbol{\Omega})^V$ be the space of vector-valued, square-integrable functions on $\boldsymbol{\Omega}$, i.e.

$$L^2(\boldsymbol{\Omega})^V = \left\{ \boldsymbol{w} : \boldsymbol{\Omega} \to \mathbb{R}^V \mid \int_{\boldsymbol{\Omega}} \left[\boldsymbol{w}(\boldsymbol{x})\right]_v \left[\boldsymbol{w}(\boldsymbol{x})\right]_v \, d\boldsymbol{x} := \int_{\boldsymbol{\Omega}} \|\boldsymbol{w}\|^2 \, d\boldsymbol{x} < \infty \right\}. \tag{2.22}$$

Let $\boldsymbol{w} \in L^2(\boldsymbol{\Omega})^V$ be a spatial test function. Multiplication of the original PDE (2.14) and integration over a space-time cell $K \times [t_i, t_i + \Delta t_i]$ yields an element-local weak formulation of the problem,

$$\int_{t_i}^{t_i + \Delta t_i} \int_K \frac{\partial}{\partial t} \left[\boldsymbol{u}\right]_v \left[\boldsymbol{w}\right]_v \, d\boldsymbol{x}dt + \int_{t_i}^{t_i + \Delta t_i} \int_K \frac{\partial}{\partial \boldsymbol{x}_d} \left[\boldsymbol{F}(\boldsymbol{u})\right]_{vd} \left[\boldsymbol{w}\right]_v \, d\boldsymbol{x}dt =$$
$$\int_{t_i}^{t_i + \Delta t_i} \int_K \left[\boldsymbol{s}(\boldsymbol{u})\right]_v \left[\boldsymbol{w}\right]_v \, d\boldsymbol{x}dt, \tag{2.23}$$

which we require to hold for all $v \in \mathcal{V}$, $\boldsymbol{w} \in L^2(\boldsymbol{\Omega})^V$, $K \in \mathcal{K}_h$ and $i \in \mathcal{I}$.

Integration by parts of the spatial integral in the second term yields

$$\begin{aligned}&\int_K \frac{\partial}{\partial \boldsymbol{x}_d} \left[\boldsymbol{F}(\boldsymbol{u})\right]_{vd} [\boldsymbol{w}]_v \, d\boldsymbol{x} = \\ &\int_K \frac{\partial}{\partial \boldsymbol{x}_d} \left(\left[\boldsymbol{F}(\boldsymbol{u})\right]_{vd} [\boldsymbol{w}]_v \right) d\boldsymbol{x} - \int_K \left[\boldsymbol{F}(\boldsymbol{u})\right]_{vd} \frac{\partial}{\partial \boldsymbol{x}_d} [\boldsymbol{w}]_v \, d\boldsymbol{x}.\end{aligned} \tag{2.24}$$

Application of the divergence theorem to the first term on the right-hand side of (2.24) yields

$$\int_K \frac{\partial}{\partial \boldsymbol{x}_d} \left(\left[\boldsymbol{F}(\boldsymbol{u})\right]_{vd} [\boldsymbol{w}]_v \right) d\boldsymbol{x} = \int_{\partial K} \left[\boldsymbol{F}(\boldsymbol{u})\right]_{vd} [\boldsymbol{w}]_v \, [\boldsymbol{n}]_d \, ds(\boldsymbol{x}), \tag{2.25}$$

where $\boldsymbol{n} \in \mathbb{R}^D$ is the unit-length, outward-pointing normal vector at a point $\boldsymbol{x}$ on the surface of K which we denote by ∂K.

Inserting eqs. (2.24) and (2.25) into eq. (2.23) yields the following more favorable element-local weak formulation of the original equation (2.14):

$$\begin{aligned}&\int_{t_i}^{t_i+\Delta t_i} \int_K \frac{\partial}{\partial t} [\boldsymbol{u}]_v [\boldsymbol{w}]_v \, d\boldsymbol{x}dt - \int_{t_i}^{t_i+\Delta t_i} \int_K \left[\boldsymbol{F}(\boldsymbol{u})\right]_{vd} \frac{\partial}{\partial \boldsymbol{x}_d} [\boldsymbol{w}]_v \, d\boldsymbol{x}dt + \\ &\int_{t_i}^{t_i+\Delta t_i} \int_{\partial K} \left[\boldsymbol{F}(\boldsymbol{u})\right]_{vd} [\boldsymbol{w}]_v \, [\boldsymbol{n}]_d \, ds(\boldsymbol{x})dt = \int_{t_i}^{t_i+\Delta t_i} \int_K \left[\boldsymbol{s}(\boldsymbol{u})\right]_v [\boldsymbol{w}]_v \, d\boldsymbol{x}dt.\end{aligned} \tag{2.26}$$

Again we require the weak formulation to hold for all $v \in \mathcal{V}$, $\boldsymbol{w} \in L^2(\boldsymbol{\Omega})^V$, $K \in \mathcal{K}_h$ and $i \in \mathcal{I}$.

2.6 Restriction to Finite-dimensional Function Spaces

To discretize eq. (2.26) we need to impose the restriction that both test and ansatz functions come from a finite-dimensional function space. First, let $\mathbb{Q}_N(K)^V$ and $\mathbb{Q}_N(K \times [t_i, t_i + \Delta t_i])^V$ be the space of vector-valued, multivariate polynomials of degree less or equal than N in each variable on K and $K \times [t_i, t_i + \Delta t_i]$, respectively. We can then define the following finite-dimensional function spaces:

- For spatial functions we define

$$\mathbb{W}_h = \left\{ \boldsymbol{w}_h \in L^2(\boldsymbol{\Omega})^V \mid \boldsymbol{w}_h|_K := \boldsymbol{w}_h^K \in \mathbb{Q}_N(K)^V \; \forall K \in \mathcal{K}_h \right\}. \tag{2.27}$$

- For space-time functions on the time subinterval $[t_i, t_i + \Delta t_i]$, $i \in \mathcal{I}$ we define

$$\begin{aligned}\tilde{\mathbb{W}}_h^i = \Big\{ &\tilde{\boldsymbol{w}}_h^i \in L^2\left(\boldsymbol{\Omega} \times [t_i, t_i + \Delta t_i]\right) \mid \\ &\tilde{\boldsymbol{w}}_h^i\Big|_K := \tilde{\boldsymbol{w}}_h^{K,i} \in \mathbb{Q}_N\left(K \times [t_i, t_i + \Delta t_i]\right) \; \forall K \in \mathcal{K}_h \Big\}.\end{aligned} \tag{2.28}$$

Replacing $\boldsymbol{w}$ by $\boldsymbol{w}_h \in \mathbb{W}_h$ and $\boldsymbol{u}$ by $\tilde{\boldsymbol{u}}_h^i \in \tilde{\mathbb{W}}_h^i$ in eq. (2.26), i.e. restricting ourselves to test and ansatz functions from finite-dimensional function spaces, yields an approximation of the weak formulation,

$$\int_{t_i}^{t_i+\Delta t_i} \int_K \frac{\partial}{\partial t} \left[\tilde{\boldsymbol{u}}_h^{K,i}\right]_v \left[\boldsymbol{w}_h^K\right]_v d\boldsymbol{x}dt - \int_{t_i}^{t_i+\Delta t_i} \int_{\partial K} \left[\boldsymbol{F}(\tilde{\boldsymbol{u}}_h^{K,i})\right]_{vd} \frac{\partial}{\partial \boldsymbol{x}_d} \left[\boldsymbol{w}_h^K\right]_v d\boldsymbol{x}dt +$$

$$\int_{t_i}^{t_i+\Delta t_i}\int_{\partial K}\left[\mathcal{G}(\tilde{\boldsymbol{u}}_h^{K,i},\tilde{\boldsymbol{u}}_h^{K',i},\boldsymbol{n})\right]_v\left[\boldsymbol{w}_h^K\right]_v\,ds(\boldsymbol{x})dt = \int_{t_i}^{t_i+\Delta t_i}\int_{K}\left[\boldsymbol{s}(\tilde{\boldsymbol{u}}_h^{K,i})\right]_v\left[\boldsymbol{w}_h^K\right]_v\,d\boldsymbol{x}dt, \tag{2.29}$$

which now has to hold for all $\boldsymbol{w}_h \in \mathbb{W}_h$, $K \in \mathcal{K}_h$ and $i \in \mathcal{I}$. Since for a cell $K \in \mathcal{K}_h$ and one of its Voronoi neighbors $K' \in V(K)$ in general it holds that

$$\tilde{\boldsymbol{u}}_h^{K,i}(\boldsymbol{x}^*) \neq \tilde{\boldsymbol{u}}_h^{K',i}(\boldsymbol{x}^*) \tag{2.30}$$

for $\boldsymbol{x}^* \in K \cap K'$, i.e. $\tilde{\boldsymbol{u}}_h^i$ is double-valued at the interface between K and K', in order to compute the surface integral we need to introduce the numerical flux function $\mathcal{G}(\tilde{\boldsymbol{u}}_h^{K,i},\tilde{\boldsymbol{u}}_h^{K',i},\boldsymbol{n})$. The numerical flux at a position $\boldsymbol{x}^* \in K \cap K'$ on the interface is obtained by (approximately) solving a Riemann problem in normal direction.

2.7 Excursus: The Riemann Problem

Let $\boldsymbol{x}^*$ be a point on the interface ∂K between a cell $K \in \mathcal{K}_h$ and one of its Voronoi neighbor $K' \in V(K)$ and let $\boldsymbol{n}$ be the outward pointing unit normal vector at this point. Then to obtain the numerical flux we need to solve the initial boundary value problem ("Riemann problem")

$$\frac{\partial}{\partial t}\left[\boldsymbol{g}\right]_v + \sum_{d=1}^{D}\frac{\partial}{\partial \boldsymbol{x}_d}\left[\boldsymbol{F}(\boldsymbol{g})\right]_{vd}\left[\boldsymbol{n}\right]_d = 0 \tag{2.31}$$

along the line $\boldsymbol{x} = \boldsymbol{x}^* + \alpha\boldsymbol{n}$ for $\alpha \in \mathbb{R}$ with discontinuous initial conditions

$$\boldsymbol{g}(\boldsymbol{x}^* + \alpha\boldsymbol{n}, 0) = \begin{cases} \tilde{\boldsymbol{u}}_h^{K,i}\Big|_{\boldsymbol{x}^*} & \text{if } \alpha < 0 \\ \tilde{\boldsymbol{u}}_h^{K',i}\Big|_{\boldsymbol{x}^*} & \text{if } \alpha > 0. \end{cases} \tag{2.32}$$

We then evaluate the similarity solution $\tilde{\boldsymbol{g}}(\alpha/t)$ of the problem and define

$$\left[\mathcal{G}\left(\tilde{\boldsymbol{u}}_h^{K,i},\tilde{\boldsymbol{u}}_h^{K',i},\boldsymbol{n}\right)\right]_v := \left[\tilde{\boldsymbol{g}}\big|_0\right]_v. \tag{2.33}$$

See [10] for an extensive overview on state of the art approximate Riemann solvers.

Continuing with eq. (2.29), integration by parts in time of the first term and observing that $\boldsymbol{w}_h$ is constant in time yields the following one-step update scheme for the cell-local time-discrete solution $\tilde{\boldsymbol{u}}_h^{K,i}$:

$$\begin{aligned}\int_K\left[\tilde{\boldsymbol{u}}_h^{K,i}\Big|_{t_i+\Delta t_i}\right]_v\left[\boldsymbol{w}_h^K\right]_v\,d\boldsymbol{x} = \int_K\left[\tilde{\boldsymbol{u}}_h^{K,i}\Big|_{t_i}\right]_v\left[\boldsymbol{w}_h^K\right]_v\,d\boldsymbol{x} + \\ \int_{t_i}^{t_i+\Delta t_i}\int_K\left[\boldsymbol{F}(\tilde{\boldsymbol{u}}_h^{K,i})\right]_{vd}\frac{\partial}{\partial \boldsymbol{x}_d}\left[\boldsymbol{w}_h^K\right]_v\,d\boldsymbol{x}dt - \\ \int_{t_i}^{t_i+\Delta t_i}\int_{\partial K}\left[\mathcal{G}(\tilde{\boldsymbol{u}}_h^{K,i},\tilde{\boldsymbol{u}}_h^{K',i},\boldsymbol{n})\right]_v\left[\boldsymbol{w}_h^K\right]_v\,ds(\boldsymbol{x})dt + \\ \int_{t_i}^{t_i+\Delta t_i}\int_K\left[\boldsymbol{s}(\tilde{\boldsymbol{u}}_h^{K,i})\right]_v\left[\boldsymbol{w}_h^K\right]_v\,d\boldsymbol{x}dt.\end{aligned} \tag{2.34}$$

Again we require eq. (2.34) to hold for all $v \in \mathcal{V}$, $\boldsymbol{w}_h \in \mathbb{W}_h$, $K \in \mathcal{K}_h$ and $i \in \mathcal{I}$. Note, however, that the scheme is incomplete, since we only know $\tilde{\boldsymbol{u}}_h^i|_t$ at a single point in time $t \in \{t_i, t_i + \Delta t_i\}$, not within the open interval, i.e. for $t \in (t_i, t_i + \Delta t_i)$. As commonly done in a DG framework we therefore proceed by replacing $\tilde{\boldsymbol{u}}_h$ on the interval $(t_i, t_i + \Delta t_i)$ by an approximation $\tilde{\boldsymbol{q}}_h^i \in \tilde{\mathbb{W}}_h^i$ which we call space-time predictor.

2.8 Space-time Predictor

To derive a procedure to compute the space-time predictor $\tilde{\boldsymbol{q}}_h^i \in \tilde{\mathbb{W}}_h^i$ we again start from the original PDE (2.14), but this time we do not use a spatial test function $\boldsymbol{w}_h \in \mathbb{W}_h$, but a space-time test function $\tilde{\boldsymbol{w}}_h^i \in \tilde{\mathbb{W}}_h^i$. If we furthermore replace the solution $\boldsymbol{u}$ by the space-time predictor $\tilde{\boldsymbol{q}}_h^i \in \tilde{\mathbb{W}}_h^i$ and integrate over the space-time element $K \times [t_i, t_i + \Delta t_i]$ we obtain the following relation:

$$\begin{aligned}
&\int_{t_i}^{t_i+\Delta t_i} \int_K \frac{\partial}{\partial t} \left[\tilde{\boldsymbol{q}}_h^{K,i}\right]_v \left[\tilde{\boldsymbol{w}}_h^{K,i}\right]_v \, d\boldsymbol{x}dt + \\
&\int_{t_i}^{t_i+\Delta t_i} \int_K \frac{\partial}{\partial \boldsymbol{x}_d} \left[\boldsymbol{F}(\tilde{\boldsymbol{q}}_h^{K,i})\right]_{vd} \left[\tilde{\boldsymbol{w}}_h^{K,i}\right]_v \, d\boldsymbol{x}dt = \\
&\int_{t_i}^{t_i+\Delta t_i} \int_K \left[\boldsymbol{s}\left(\tilde{\boldsymbol{q}}_h^{K,i}\right)\right]_v \left[\tilde{\boldsymbol{w}}_h^{K,i}\right]_v \, d\boldsymbol{x}dt.
\end{aligned} \tag{2.35}$$

We require eq. (2.35) to hold for all $v \in \mathcal{V}$, $\tilde{\boldsymbol{w}}_h^i \in \tilde{\mathbb{W}}_h^i$, $K \in \mathcal{K}_h$ and $i \in \mathcal{I}$. Note that we do not apply the divergence theorem this time to keep the space-time predictor independent of neighboring cells. This move is key to ensuring that the space-time predictor as the computationally most expensive part of the scheme fosters the principles of communication avoidance and data locality.

Integration by parts in time applied to the first term and some rearrangement yields

$$\begin{aligned}
&\int_K \left[\tilde{\boldsymbol{q}}_h^{K,i}\Big|_{t_i+\Delta t_i}\right]_v \left[\tilde{\boldsymbol{w}}_h^{K,i}\Big|_{t_i+\Delta t_i}\right]_v d\boldsymbol{x} - \int_{t_i}^{t_i+\Delta t_i} \int_K \left[\tilde{\boldsymbol{q}}_h^{K,i}\right]_v \frac{\partial}{\partial t} \left[\tilde{\boldsymbol{w}}_h^{K,i}\right]_v \, d\boldsymbol{x}dt = \\
&\int_{t_i}^{t_i+\Delta t_i} \int_K \left[\boldsymbol{s}\left(\tilde{\boldsymbol{q}}_h^{K,i}\right)\right]_v \left[\tilde{\boldsymbol{w}}_h^{K,i}\right]_v \, d\boldsymbol{x}dt - \\
&\int_{t_i}^{t_i+\Delta t_i} \int_K \frac{\partial}{\partial \boldsymbol{x}_d} \left[\boldsymbol{F}(\tilde{\boldsymbol{q}}_h^{K,i})\right]_{vd} \left[\tilde{\boldsymbol{w}}_h^{K,i}\right]_v \, d\boldsymbol{x}dt,
\end{aligned} \tag{2.36}$$

which we require to hold for all $v \in \mathcal{V}$, $\tilde{\boldsymbol{w}}_h^i \in \tilde{\mathbb{W}}_h^i$, $K \in \mathcal{K}_h$ and $i \in \mathcal{I}$. In conjunction with the initial condition

$$\tilde{\boldsymbol{q}}_h^{K,i}\Big|_{t_i} = \tilde{\boldsymbol{u}}_h^{K,i}\Big|_{t_i} \tag{2.37}$$

and an initial guess

$$\tilde{\boldsymbol{q}}_h^{K,i}\Big|_{t} = \tilde{\boldsymbol{u}}_h^{K,i}\Big|_{t_i} \quad \forall t \in (t_i, t_i + \Delta t_i] \tag{2.38}$$

this relation can be used as a fixed-point iteration to find the cell-local space-time predictor $\tilde{\boldsymbol{q}}_h^{K,i}$.

In the following two sections we will introduce mappings from spatial elements K and space-time elements $K \times [t_i, t_i + \Delta t_i]$ to spatial and space-time reference cells as well as orthogonal bases for the spaces $\mathbb{W}_h$ and $\tilde{\mathbb{W}}_h^i$. We will then insert these results into eq. (2.36) and derive a fully-discrete iterative method to compute the cell-local space-time predictor $\tilde{\boldsymbol{q}}_h^{K,i}$.

2.9 Reference Elements and Mappings

Let $\hat{K} := [0,1]^D$ be the spatial reference element and $\boldsymbol{\xi} \in \hat{K}$ be a point therein. Let $[0,1]$ be the reference time interval and $\tau \in [0,1]$ be a point in reference time. We can then introduce the following mappings:

Spatial mappings: Let $K \in \mathcal{K}_h$ be a cell in global coordinates with extent $\Delta \boldsymbol{x}^K$ and "lower-left corner" $\boldsymbol{P}_K$, more precisely that is

$$\left[\Delta \boldsymbol{x}^K\right]_d = \max_{\boldsymbol{x} \in K} [\boldsymbol{x}]_d - \min_{\boldsymbol{x} \in K} [\boldsymbol{x}]_d \tag{2.39}$$

and

$$[\boldsymbol{P}_K]_d = \min_{\boldsymbol{x} \in K} [\boldsymbol{x}]_d \tag{2.40}$$

for $d \in \mathcal{D}$. We can then define a mapping

$$\boldsymbol{\mathcal{X}}_K : \hat{K} \to K, \boldsymbol{\xi} \mapsto \boldsymbol{\mathcal{X}}_K(\boldsymbol{\xi}) = \boldsymbol{x} \tag{2.41}$$

via the relation

$$[\boldsymbol{x}]_d = \left[\boldsymbol{\mathcal{X}}_K(\boldsymbol{\xi})\right]_d = [\boldsymbol{P}_K]_d + [\Delta \boldsymbol{x}]_d \left[\boldsymbol{\xi}\right]_d \tag{2.42}$$

for all $d \in \mathcal{D}$ (i.e. obviously no summation on d), $\boldsymbol{x} \in K$, $\boldsymbol{\xi} \in \hat{K}$ and $K \in \mathcal{K}_h$.

Temporal mappings: Let $[t_i, t_i + \Delta t_i], i \in \mathcal{I}$ be an interval in global time. Then the function

$$\mathcal{T}_i : [0,1] \to [t_i, t_i + \Delta t_i], \tau \mapsto \mathcal{T}_i(\tau) = t_i + \Delta t_i \tau = t \tag{2.43}$$

maps a point $\tau \in [0,1]$ in reference time to a point $t \in [t_i, t_i + \Delta t_i]$ in global time for all $i \in \mathcal{I}$.

The inverse mappings, the Jacobian matrices and the Jacobi determinants of the mappings are given in the following:

Spatial mappings: The inverse spatial mappings

$$\boldsymbol{\mathcal{X}}_K^{-1} : K \to \hat{K}, \boldsymbol{x} \mapsto \boldsymbol{\mathcal{X}}_K^{-1}(\boldsymbol{x}) = \boldsymbol{\xi} \tag{2.44}$$

are defined via the relation

$$\left[\boldsymbol{\xi}\right]_d = \left[\boldsymbol{\mathcal{X}}_K^{-1}(\boldsymbol{x})\right]_d = \frac{1}{\left[\Delta \boldsymbol{x}^K\right]_d} \left([\boldsymbol{x}]_d - [\boldsymbol{P}_K]_d\right) \tag{2.45}$$

for all $d \in \mathcal{D}$ (again no summation), $\boldsymbol{\xi} \in \hat{K}$, $\boldsymbol{x} \in K$ and $K \in \mathcal{K}_h$. The Jacobian of $\boldsymbol{\mathcal{X}}_K$ is found to be

$$\left[\frac{\partial \boldsymbol{\mathcal{X}}_K}{\partial \boldsymbol{\xi}}\right]_{dd'} = \frac{\partial \left[\boldsymbol{\mathcal{X}}_K\right]_d}{\partial \boldsymbol{\xi}_{d'}} = \left[\Delta \boldsymbol{x}^K\right]_d \delta_{dd'}, \tag{2.46}$$

where $d, d' \in \mathcal{D}$ (i.e. no summation on d) and for all $K \in \mathcal{K}_h$. As usual $\delta_{dd'}$ denotes the Kronecker delta defined as

$$\delta_{dd'} = \begin{cases} 1 & \text{if } d = d' \\ 0 & \text{if } d \neq d'. \end{cases} \tag{2.47}$$

The Jacobi determinant of $\boldsymbol{\mathcal{X}}_K$ for $K \in \mathcal{K}_h$ then simply is

$$J_{\boldsymbol{\mathcal{X}}_K} := \det\left(\frac{\partial \boldsymbol{\mathcal{X}}_K}{\partial \boldsymbol{\xi}}\right) = \prod_{d=1}^{D} \left[\Delta \boldsymbol{x}^K\right]_d, \tag{2.48}$$

i.e. the determinant is equal to the volume of the quadrilateral K and constant with respect to all $\boldsymbol{\xi} \in \hat{K}$.

Temporal mappings: The inverse temporal mappings are given as

$$\mathcal{T}_i^{-1} : [t_i, t_i + \Delta t_i] \to [0,1], t \mapsto \mathcal{T}_i^{-1}(t) = \frac{t - t_i}{\Delta t_i} = \tau \tag{2.49}$$

for all $\tau \in [0,1]$, $t \in [t_i, t_i + \Delta t_i]$ and $i \in \mathcal{I}$. In the trivial case of a one-dimensional mapping the Jacobian of $\mathcal{T}_i$ is a scalar which in turn is its own determinant. One finds

$$J_{\mathcal{T}_i} := \frac{d\mathcal{T}_i}{\partial \tau} = \Delta t_i, \tag{2.50}$$

which again is constant with respect to all $\tau \in [0,1]$.

2.10 Orthogonal Bases for the Finite-dimensional Function Spaces

In section 2.6 we introduced finite-dimensional, cell-wise polynomial function spaces $\mathbb{W}_h$ and $\tilde{\mathbb{W}}_h^i$ for spatial and space-time ansatz and test functions, respectively. On our way towards a fully discrete version of the relations (2.36) and (2.34) to obtain the space-time predictor and the solution at the next time step, respectively, we will now derive a set of functions that form bases for the two function spaces of interest. Following the approach presented by Dumbers et al. in [11], throughout the thesis we will use the set of Lagrange functions with nodes located at the roots of the Legendre polynomials and tensor products thereof. In the later chapters of this work it will become obvious why this particular choice is highly favorable. For the moment the two major reasons shall be stated as an outlook:

1. Numerical integration using the Gauss-Legendre method is simple and computationally cheap, since the function values at the Gauss-Legendre nodes are directly available as they are equal to the degrees of freedom representing the local polynomial.
2. The resulting bases are orthogonal, which in turn ensures that the operator matrices exhibit a spares block structure allowing computations to be carried out efficiently in a dimension-by-dimension manner.

2.10.1 Lagrange Interpolation

Let $f \in \mathbb{Q}_N([0,1])$ be a polynomial of degree less or equal than N and for the index set $\mathcal{N} := \{0, 1, \ldots, N\}$ let $\{\hat{\xi}_n\}_{n\in\mathcal{N}}$ be a set of distinct nodes in $[0,1]$. Then the Lagrange interpolation of f,

$$\hat{f}(\xi) = \sum_{n=0}^{N} L_n(\xi) f(\xi_n) \tag{2.51}$$

with Lagrange functions

$$L_n(\xi) = \prod_{m=0, m\neq n}^{N} \frac{\xi - \hat{\xi}_m}{\hat{\xi}_n - \hat{\xi}_m} \tag{2.52}$$

is exact, i.e.

$$f(\xi) = \hat{f}(\xi) \quad \forall \xi \in [0,1]. \tag{2.53}$$

Since therefore every polynomial $f \in \mathbb{Q}_N([0,1])$ can be represented as a linear combination of the Lagrange polynomials L_n, $n \in \mathcal{N}$, the set of functions $\{L_n\}_{n\in\mathcal{N}}$ is a basis of $Q_N([0,1])$.

The following observation is an important property of the Lagrange polynomials:

$$L_n(\hat{\xi}_{n'}) = \delta_{nn'}, \tag{2.54}$$

i.e. at each node $\hat{\xi}_n$ only L_n has value 1 and all other polynomials evaluate to 0.

2.10.2 Legendre Polynomials and Gauss-Legendre Integration

Let $P_0 : [-1,1] \to \mathbb{R}, \xi \mapsto 1$ and $P_1 : [-1,1] \to \mathbb{R}, \xi \mapsto \xi$ define the zeroth and the first Legendre polynomial, respectively. Then the $N+1$-th Legendre polynomial can be defined via the following recurrence relation:

$$P_{N+1}(\xi) = \frac{1}{N+1}\left((2N+1)P_N(\xi) - nP_{N-1}(\xi)\right). \tag{2.55}$$

Let $\{\tilde{\xi}_n\}_{n\in\mathcal{N}}$ be the roots of the $N+1$-th Legendre polynomial P_{N+1}. Then $\{\hat{\xi}_n\}_{n\in\mathcal{N}}$ with

$$\hat{\xi}_n = \frac{1}{2}(\tilde{\xi}_n + 1) \tag{2.56}$$

are the roots of the $N+1$-th Legendre polynomial linearly mapped to the interval $[0,1]$. In conjunction with a set of suitable weights $\{\hat{\omega}_n\}_{n\in\mathcal{N}}$ Gauss-Legendre integration can be used to integrate polynomials of degree up to $2N+1$ over the integral $[0,1]$ exactly, i.e.

$$\int_0^1 f(\xi)\, d\xi = \sum_{n=0}^{N} \hat{\omega}_n f(\hat{\xi}_n) \quad \forall f \in \mathbb{Q}_{2N+1}\left([0,1]\right). \tag{2.57}$$

A Python notebook on how to compute the nodes $\{\hat{\xi}_n\}_{n\in\mathcal{N}}$ and weights $\{\hat{\omega}_n\}_{n\in\mathcal{N}}$ can be found in Appendix A.

2.10.3 Scalar-valued Basis Functions on the One-dimensional Reference Element

Let $\{\hat{\psi}_n\}_{n\in\mathcal{N}}$ be the set of $N+1$ Lagrange polynomials with nodes at the roots of the $N+1$-th Legendre polynomial linearly mapped to the interval $[0,1]$, i.e.

$$\hat{\psi}_n(\xi) = \sum_{n'=0}^{N} \frac{\xi - \hat{\xi}_{n'}}{\hat{\xi}_n - \hat{\xi}_{n'}} \tag{2.58}$$

for $n \in \mathcal{N}$. Since $\{\hat{\psi}_n\}_{n\in\mathcal{N}}$ are Lagrange polynomials and the roots $\{\hat{\xi}_n\}_{n\in\mathcal{N}}$ are distinct, the set is a basis of $\mathbb{Q}_N([0,1])$. Since furthermore

$$\left\langle \hat{\psi}_n, \hat{\psi}_m \right\rangle_{L^2([0,1])} = \int_0^1 \hat{\psi}_n(\xi)\hat{\psi}_m(\xi)\, d\xi = \sum_{n'=0}^{N} \hat{w}'_n \hat{\psi}_n(\hat{\xi}_{n'})\hat{\psi}_m(\hat{\xi}_{n'}) = \hat{w}_n \delta_{mn} \tag{2.59}$$

for all $m, n \in \mathcal{N}$ (i.e. no summation over n), the set is even an orthogonal basis of $\mathbb{Q}_N([0,1])$ with respect to the L^2-scalar product as defined above. In this derivation we used the fact that $\hat{\psi}_n\hat{\psi}_m$ has degree $2N$ and that Gauss-Legendre integration with $N+1$ nodes is exact for polynomials up to degree $2N+1$.

2.10.4 Scalar-valued Basis Functions on the Spatial Reference Element

For the vector-valued index set $\boldsymbol{\mathcal{N}} := \{0, 1, \ldots, N\}^D$ let us define the set of scalar-valued spatial basis functions $\{\hat{\phi}_{\boldsymbol{n}}\}_{\boldsymbol{n}\in\boldsymbol{\mathcal{N}}}$ on $\hat{K} := [0,1]^D$ as

$$\hat{\phi}_{\boldsymbol{n}}(\boldsymbol{\xi}) = \prod_{d=1}^{D} \hat{\psi}_{\boldsymbol{n}_d}(\boldsymbol{\xi}_d) = \hat{\psi}_{\boldsymbol{n}_d}(\boldsymbol{\xi}_d), \tag{2.60}$$

i.e. $\{\hat{\phi}_{\boldsymbol{n}}\}_{\boldsymbol{n}\in\boldsymbol{\mathcal{N}}}$ is the tensor product of $\{\hat{\psi}_n\}_{n\in\mathcal{N}}$ over $\boldsymbol{n} \in \boldsymbol{\mathcal{N}}$. In consequence it is a basis of $\mathbb{Q}([0,1]^D) = \mathbb{Q}(\hat{K})$. If we define

$$\left[\hat{\boldsymbol{\xi}}_{\boldsymbol{n}}\right]_d := \hat{\xi}_{[\boldsymbol{n}]_d} \tag{2.61}$$

and

$$\hat{\omega}_{\boldsymbol{n}} := \prod_{d=1}^{D} \hat{\omega}_{[\boldsymbol{n}]_d}, \tag{2.62}$$

for all $d \in \mathcal{D}$ and $\boldsymbol{n} \in \boldsymbol{\mathcal{N}}$, we furthermore observe that the basis is orthogonal with respect to the L^2-scalar product, since

$$\begin{aligned} \left\langle \hat{\phi}_{\boldsymbol{n}}, \hat{\phi}_{\boldsymbol{m}} \right\rangle_{L^2(\hat{K})} = \int_{\hat{K}} \hat{\phi}_{\boldsymbol{n}}(\boldsymbol{\xi})\hat{\phi}_{\boldsymbol{m}}(\boldsymbol{\xi})\, d\boldsymbol{\xi} = \\ \sum_{\boldsymbol{n}'\in\boldsymbol{\mathcal{N}}} \left(\hat{\omega}_{\boldsymbol{n}'} \hat{\phi}_{\boldsymbol{n}}(\hat{\boldsymbol{\xi}}_{\boldsymbol{n}'})\hat{\phi}_{\boldsymbol{m}}(\hat{\boldsymbol{\xi}}_{\boldsymbol{n}'}) \right) = \hat{\omega}_{\boldsymbol{n}} \delta_{\boldsymbol{n}\boldsymbol{m}} \end{aligned} \tag{2.63}$$

for all $\boldsymbol{n}, \boldsymbol{m} \in \boldsymbol{\mathcal{N}}$. The natural extensions of the Kronecker delta for vector-valued indices is defined as follows:

$$\delta_{\boldsymbol{n}\boldsymbol{m}} = \prod_{d=1}^{D} \delta_{[\boldsymbol{n}]_d[\boldsymbol{m}]_d} = \delta_{[\boldsymbol{n}]_d[\boldsymbol{m}]_d} (= \delta_{\boldsymbol{n}_d\boldsymbol{m}_d}). \tag{2.64}$$

2.10.5 Scalar-valued Basis Functions on the Space-time Reference Element

Analogous to the procedure illustrated above for the spatial reference element $\hat{K}$ we can define a basis $\{\hat{\theta}_{\boldsymbol{n},l}\}_{\boldsymbol{n}\in\boldsymbol{\mathcal{N}},l\in\mathcal{N}}$ of $\mathbb{Q}_N(\hat{K}\times[0,1])$ on the reference space-time element $\hat{K}\times[0,1]$ as

$$\hat{\theta}_{\boldsymbol{n},l}(\boldsymbol{\xi},\tau) = \hat{\phi}_{\boldsymbol{n}}(\boldsymbol{\xi})\hat{\psi}_l(\tau), \tag{2.65}$$

which again is orthogonal, since

$$\left\langle \hat{\theta}_{\boldsymbol{n},l}, \hat{\theta}_{\boldsymbol{m},k} \right\rangle_{L^2\left(\hat{K}\times[0,1]\right)} = \int_0^1 \int_{\hat{K}} \hat{\theta}_{\boldsymbol{n},l}\hat{\theta}_{\boldsymbol{m},k}\, d\boldsymbol{\xi} d\tau = \hat{\omega}_{\boldsymbol{n}}\hat{\omega}_l\delta_{\boldsymbol{nm}}\delta_{lk} \tag{2.66}$$

for all $\boldsymbol{n},\boldsymbol{m}\in\boldsymbol{\mathcal{N}}$ and $l,k\in\mathcal{N}$.

2.10.6 Vector-valued Basis Functions on the Spatial Reference Element

If we define $\{\hat{\boldsymbol{\phi}}_{\boldsymbol{n}v}\}_{\boldsymbol{n}\in\boldsymbol{\mathcal{N}},v\in\mathcal{V}}$ as

$$\hat{\boldsymbol{\phi}}_{\boldsymbol{n},v} = \hat{\phi}_{\boldsymbol{n}}\boldsymbol{e}_v, \tag{2.67}$$

where $\boldsymbol{e}_v$ is the v-th unit vector, i.e.

$$[\boldsymbol{e}_v]_{v'} = \delta_{vv'} \tag{2.68}$$

for $v,v'\in\mathcal{V}$, we obtain a basis of $\mathbb{Q}_N(\hat{K})^V$. Since furthermore

$$\begin{aligned}\left\langle \hat{\boldsymbol{\phi}}_{\boldsymbol{n},v}, \hat{\boldsymbol{\phi}}_{\boldsymbol{n}',v'} \right\rangle_{L^2\left(\hat{K}\right)^V} &= \int_{\hat{K}} \left[\hat{\boldsymbol{\phi}}_{\boldsymbol{n},v}\right]_j \left[\hat{\boldsymbol{\phi}}_{\boldsymbol{n}',v'}\right]_j d\boldsymbol{\xi} = \\ \left([\boldsymbol{e}_v]_j\,[\boldsymbol{e}_{v'}]_j\right) \int_0^1 \int_{\hat{K}} \hat{\phi}_{\boldsymbol{n}}\hat{\phi}_{\boldsymbol{n}'}\, d\boldsymbol{\xi} &= \hat{\omega}_{\boldsymbol{n}}\delta_{\boldsymbol{nn}'}\delta_{vv'}\end{aligned} \tag{2.69}$$

for all $\boldsymbol{n},\boldsymbol{n}'\in\boldsymbol{\mathcal{N}}$ and $v,v'\in\mathcal{V}$ the set is an orthogonal basis for $\mathbb{Q}_N(\hat{K})^V$.

2.10.7 Vector-valued Basis Functions on the Space-time Reference Element

The set $\{\hat{\boldsymbol{\theta}}_{\boldsymbol{n},l,v}\}_{\boldsymbol{n}\in\boldsymbol{\mathcal{N}},l\in\mathcal{N},v\in\mathcal{V}}$ defined as

$$\hat{\boldsymbol{\theta}}_{\boldsymbol{n},l,v}(\boldsymbol{\xi},\tau) = \hat{\theta}_{\boldsymbol{n},l}(\boldsymbol{\xi},\tau)\boldsymbol{e}_v = \hat{\phi}_{\boldsymbol{n}}(\boldsymbol{\xi})\hat{\psi}_l(\tau)\boldsymbol{e}_v \tag{2.70}$$

is a basis of $\mathbb{Q}_N(\hat{K}\times[0,1])^V$. Since furthermore

$$\left\langle \hat{\boldsymbol{\theta}}_{\boldsymbol{n},l,v}, \hat{\boldsymbol{\theta}}_{\boldsymbol{n}',l',v'} \right\rangle_{L^2\left(\hat{K}\times[0,1]\right)^V} = \int_0^1 \int_{\hat{K}} \left[\hat{\boldsymbol{\theta}}_{\boldsymbol{n},l,v}\right]_j \left[\hat{\boldsymbol{\theta}}_{\boldsymbol{n}',l',v'}\right]_j d\boldsymbol{\xi} d\tau = \hat{\omega}_{\boldsymbol{n}}\hat{\omega}_l\delta_{\boldsymbol{nn}'}\delta_{ll'}\delta_{vv'}, \tag{2.71}$$

for all $\boldsymbol{n},\boldsymbol{n}'\in\boldsymbol{\mathcal{N}}$, $l,l'\in\mathcal{N}$ and $v,v'\in\mathcal{V}$, the set is an orthogonal basis with respect to the respective L^2-scalar product.

2.11 Basis Functions in Global Coordinates

We can use the mappings derived in chapter 2.9 to map the basis functions to global coordinates. For the vector-valued basis functions on a spatial element K we obtain

$$\boldsymbol{\phi}^K_{\boldsymbol{n},v}(\boldsymbol{x}) = \begin{cases} \hat{\boldsymbol{\phi}}_{\boldsymbol{n},v}\left(\boldsymbol{\mathcal{X}}_K^{-1}(\boldsymbol{x})\right) & \text{if } \boldsymbol{x} \in K \\ 0 & \text{otherwise,} \end{cases} \tag{2.72}$$

and for the vector-valued basis functions on a space-time element $K \times [t_i, t_i + \Delta t_i]$ we have

$$\boldsymbol{\theta}^{K,i}_{\boldsymbol{n},l,v}(\boldsymbol{x},t) = \begin{cases} \hat{\boldsymbol{\theta}}_{\boldsymbol{n},l,v}\left(\boldsymbol{\mathcal{X}}_K^{-1}(\boldsymbol{x}), \mathcal{T}_i^{-1}(t)\right) & \text{if } \boldsymbol{x} \in K \text{ and } t \in [t_i, t_i + \Delta t_i] \\ 0 & \text{otherwise} \end{cases} \tag{2.73}$$

for $\boldsymbol{n} \in \boldsymbol{\mathcal{N}}$, $l \in \{0, 1, \ldots, N\}$ as well as $v \in \mathcal{V}$ and for all $K \in \mathcal{K}_h$ and $i \in \mathcal{I}$.

2.12 Iterative Method for the Space-time Predictor

We recall relation (2.38) for the space-time predictor. Plugging in the initial condition (2.37) yields

$$\begin{aligned} &\int_K \left[\tilde{\boldsymbol{q}}_h^{K,i}\Big|_{t_i+\Delta t_i}\right]_j \left[\tilde{\boldsymbol{w}}_h^{K,i}\Big|_{t_i+\Delta t_i}\right]_j d\boldsymbol{x} - \int_{t_i}^{t_i+\Delta t_i}\int_K \left[\tilde{\boldsymbol{q}}_h^{K,i}\right]_j \frac{\partial}{\partial t}\left[\tilde{\boldsymbol{w}}_h^{K,i}\right]_j d\boldsymbol{x}dt = \\ &\int_K \left[\tilde{\boldsymbol{u}}_h^{K,i}\Big|_{t_i}\right]_j \left[\tilde{\boldsymbol{w}}_h^{K,i}\Big|_{t_i}\right]_j d\boldsymbol{x} - \int_{t_i}^{t_i+\Delta t_i}\int_K \frac{\partial}{\partial \boldsymbol{x}_k}\left[\boldsymbol{F}(\tilde{\boldsymbol{q}}_h^{K,i})\right]_{jk}\left[\tilde{\boldsymbol{w}}_h^{K,i}\right]_j d\boldsymbol{x}dt + \\ &\int_{t_i}^{t_i+\Delta t_i}\int_K \left[\boldsymbol{s}\left(\tilde{\boldsymbol{q}}_h^{K,i}\right)\right]_j \left[\tilde{\boldsymbol{w}}_h^{K,i}\right]_j d\boldsymbol{x}dt, \end{aligned} \tag{2.74}$$

which we require to hold for all $j \in \mathcal{V}$, $\tilde{\boldsymbol{w}}_h \in \tilde{\mathbb{W}}_h$, $K \in \mathcal{K}_h$ and $i \in \mathcal{I}$.

Making use of the bases we derived in the previous section the cell-local space-time predictor $\tilde{\boldsymbol{q}}_h^{K,i}$ can be represented by a tensor of coefficients $\hat{\boldsymbol{q}}^{K,i}$ ("degrees of freedom") as follows:

$$\tilde{\boldsymbol{q}}_h^{K,i} = \left[\hat{\boldsymbol{q}}^{K,i}\right]_{\boldsymbol{n}lv} \boldsymbol{\theta}^{K,i}_{\boldsymbol{n},l,v}. \tag{2.75}$$

The initial condition $\tilde{\boldsymbol{u}}_h^{K,i}\Big|_{t_i}$ can be represented as

$$\tilde{\boldsymbol{u}}_h^{K,i}\Big|_{t_i} = \left[\hat{\boldsymbol{u}}^{K,i}\right]_{\boldsymbol{n}v} \boldsymbol{\phi}^K_{\boldsymbol{n},v}, \tag{2.76}$$

where

$$\left[\hat{\boldsymbol{u}}^{K,i}\right]_{\boldsymbol{n},v} = \left[\tilde{\boldsymbol{u}}_h^{K,i}\Big|_{\left(\boldsymbol{\mathcal{X}}_K(\xi_{\boldsymbol{n}}),t_i\right)}\right]_v. \tag{2.77}$$

The flux and source functions (assumed to be continuous) evaluated at the space-time predictor can furthermore be represented by coefficients $\hat{\boldsymbol{F}}^{K,i}$ and $\hat{\boldsymbol{s}}^{K,i}$ as

$$\left[\boldsymbol{F}(\tilde{\boldsymbol{q}}_h^{K,i})\right]_{vd} = \left[\boldsymbol{F}\left(\left[\hat{\boldsymbol{q}}^{K,i}\right]_{\boldsymbol{n}l}\right)\right]_{vd} \boldsymbol{\theta}^{K,i}_{\boldsymbol{n},l,v} := \left[\hat{\boldsymbol{F}}^{K,i}\right]_{\boldsymbol{n}lvd} \boldsymbol{\theta}^{K,i}_{\boldsymbol{n},l,v} \tag{2.78}$$

and

$$\left[s\left(\tilde{q}_h^{K,i}\right)\right]_v = \left[s\left(\left[\hat{q}^{K,i}\right]_{nl}\right)\right]_v \theta^{K,i}_{n,l,v} := \left[\hat{s}^{K,i}\right]_{nlv} \theta^{K,i}_{n,l,v}, \tag{2.79}$$

respectively.

Inserting eqs. (2.75), (2.76), (2.78) and (2.79) into eq. (2.74) and introduction of the iteration index $r \in \{0,1,\ldots,R\} := \mathcal{R}$ leads to the following iterative scheme for the degrees of freedom of the cell-local space-time predictor:

$$\underbrace{\int_K \left[\left[\hat{q}^{K,i,r+1}\right]_{nlv} \theta^{K,i}_{n,l,v}\Big|_{t_i+\Delta t_i}\right]_j \left[\theta^{K,i}_{\alpha,\beta,\gamma}\Big|_{t_i+\Delta t_i}\right]_j dx}_{\text{S-I}} -$$
$$\underbrace{\int_{t_i}^{t_i+\Delta t_i}\int_K \left[\left[\hat{q}^{K,i,r+1}\right]_{nlv} \theta^{K,i}_{n,l,v}\right]_j \frac{\partial}{\partial t}\left[\theta^{K,i}_{\alpha,\beta,\gamma}\right]_j dxdt}_{\text{S-II}} =$$
$$\underbrace{\int_K \left[\left[\hat{u}^{K,i}\right]_{nv} \phi^K_{n,v}\right]_j \left[\theta^{K,i}_{\alpha,\beta,\gamma}\Big|_{t_i}\right]_j dx}_{\text{S-III}} -$$
$$\underbrace{\int_{t_i}^{t_i+\Delta t_i}\int_K \sum_{d\in\mathcal{D}} \frac{\partial}{\partial x_d}\left[\left[\hat{F}^{K,i}\right]_{nlvd} \theta^{K,i}_{n,l,v}\right]_{jd}\left[\theta^{K,i}_{\alpha,\beta,\gamma}\right]_j dxdt}_{\text{S-IV}} +$$
$$\underbrace{\int_{t_i}^{t_i+\Delta t_i}\int_K \left[\left[\hat{s}^{K,i}\right]_{nlv} \theta^{K,i}_{n,l,v}\right]_j \left[\theta^{K,i}_{\alpha,\beta,\gamma}\right]_j dxdt}_{\text{S-V}}. \tag{2.80}$$

We require this relation to hold for all $\alpha \in \mathcal{N}$, $\beta \in \mathcal{N}$ and $\gamma \in \mathcal{V}$. $\hat{F}^{K,i}$ and $\hat{s}^{K,i}$ are computed based on $\hat{q}^{K,i,r}$ so that the left-hand side solely depends on $\hat{q}^{K,i,r+1}$ and the right-hand side solely depends on $\hat{q}^{K,i,r}$.

As initial condition, i.e. for $r = 0$, we use

$$\left[\hat{q}^{K,i,0}\right]_{nlv} = \left[\hat{u}^{K,i}\right]_{nv} \tag{2.81}$$

for all time degrees of freedom $l \in \mathcal{N}$.

We will now proceed in a term-by-term fashion to rewrite all integrals with respect to reference coordinates so that we can finally derive a complete scheme to compute $\hat{q}^{K,i,r+1}$ that holds for all $K \in \mathcal{K}_h$ and $i \in \mathcal{I}$.

2.12.1 Term S-I

The first term of eq. (2.80) can be rewritten with respect to reference coordinates as

$$\int_K \left[\left[\hat{q}^{K,i,r+1}\right]_{nlw} \theta^{K,i}_{n,l,v}\Big|_{t_i+\Delta t_i}\right]_j \left[\theta^{K,i}_{\alpha,\beta,\gamma}\Big|_{t_i+\Delta t_i}\right]_j dx =$$
$$\int_K \left[\hat{q}^{K,i,r+1}\right]_{nlv} \phi^K_n\ \psi^i_l\Big|_{t_i+\Delta t_i} [e_v]_j\, \phi^K_\alpha\ \psi^i_\beta\Big|_{t_i+\Delta t_i} \left[e_\gamma\right]_j dx =$$

$$
\begin{aligned}
& J\boldsymbol{\mathcal{X}}_K \int_{\hat{K}} \left[\hat{\boldsymbol{q}}^{K,i,r+1}\right]_{\boldsymbol{n}lv} \hat{\phi}_{\boldsymbol{n}}\ \hat{\psi}_l\Big|_1 \left[\boldsymbol{e}_v\right]_j \hat{\phi}_{\boldsymbol{\alpha}}\ \hat{\psi}_\beta\Big|_1 \left[\boldsymbol{e}_\gamma\right]_j d\boldsymbol{\xi} = \\
& J\boldsymbol{\mathcal{X}}_K \sum_{\boldsymbol{\alpha}'\in\boldsymbol{\mathcal{N}}} \left(\hat{\omega}_{\boldsymbol{\alpha}'} \left[\hat{\boldsymbol{q}}^{K,i,r+1}\right]_{\boldsymbol{n}lv} \hat{\phi}_{\boldsymbol{n}}(\hat{\boldsymbol{\xi}}_{\boldsymbol{\alpha}'})\ \hat{\psi}_l\Big|_1 \left[\boldsymbol{e}_v\right]_j \hat{\phi}_{\boldsymbol{\alpha}}(\hat{\boldsymbol{\xi}}_{\boldsymbol{\alpha}'})\ \hat{\psi}_\beta\Big|_1 \left[\boldsymbol{e}_\gamma\right]_j \right) = \\
& J\boldsymbol{\mathcal{X}}_K \sum_{\boldsymbol{\alpha}'\in\boldsymbol{\mathcal{N}}} \left(\hat{\omega}_{\boldsymbol{\alpha}'} \left[\hat{\boldsymbol{q}}^{K,i,r+1}\right]_{\boldsymbol{n}lv} \delta_{\boldsymbol{n}\boldsymbol{\alpha}'}\ \hat{\psi}_l\Big|_1 \delta_{vj}\delta_{\boldsymbol{\alpha}\boldsymbol{\alpha}'}\ \hat{\psi}_\beta\Big|_1 \delta_{j\gamma} \right) = \\
& J\boldsymbol{\mathcal{X}}_K \hat{\omega}_{\boldsymbol{\alpha}} \left[\hat{\psi}_\beta\Big|_1\ \hat{\psi}_l\Big|_1\right] \left[\hat{\boldsymbol{q}}^{K,i,r+1}\right]_{\boldsymbol{\alpha}l\gamma} = \\
& J\boldsymbol{\mathcal{X}}_K \hat{\omega}_{\boldsymbol{\alpha}} \left[\boldsymbol{R}\right]_{\beta l} \left[\hat{\boldsymbol{q}}^{K,i,r+1}\right]_{\boldsymbol{\alpha}l\gamma}, \qquad (2.82)
\end{aligned}
$$

where we remember from eq. (2.48) that

$$
J\boldsymbol{\mathcal{X}}_K = \prod_{d=1}^{D} \left[\Delta \boldsymbol{x}\right]_d \qquad (2.83)
$$

and we define the matrix $\boldsymbol{R}$ representing the Right Reference Element Mass Operator as

$$
\left[\boldsymbol{R}\right]_{ij} := \left[\hat{\psi}_i\Big|_1\ \hat{\psi}_j\Big|_1\right]_{ij} \qquad (2.84)
$$

for $i, j \in \mathcal{N}$. A Python script to compute $\boldsymbol{R}$ can be found in Appendix A.

2.12.2 Term S-II

The second term of eq. (2.80) can be rewritten with respect to reference coordinates as follows:

$$
\begin{aligned}
& \int_{t_i}^{t_i+\Delta t_i} \int_K \left[\left[\hat{\boldsymbol{q}}^{K,i,r+1}\right]_{\boldsymbol{n}lv} \boldsymbol{\theta}^{K,i}_{\boldsymbol{n},l,v} \right]_j \frac{\partial}{\partial t} \left[\boldsymbol{\theta}^{K,i}_{\boldsymbol{\alpha},\beta,\gamma}\right]_j d\boldsymbol{x}dt = \\
& \int_{t_i}^{t_i+\Delta t_i} \int_K \left[\hat{\boldsymbol{q}}^{K,i,r+1}\right]_{\boldsymbol{n}lv} \phi^K_{\boldsymbol{n}} \psi^i_l \left[\boldsymbol{e}_v\right]_j \phi^K_{\boldsymbol{\alpha}} \left(\frac{\partial}{\partial t} \psi^i_\beta \right) \left[\boldsymbol{e}_\gamma\right]_j d\boldsymbol{x}dt = \\
& J_{\mathcal{T}_i} J\boldsymbol{\mathcal{X}}_K \int_0^1 \int_{\hat{K}} \left[\hat{\boldsymbol{q}}^{K,i,r+1}\right]_{\boldsymbol{n}lv} \hat{\phi}_{\boldsymbol{n}} \hat{\psi}_l \left[\boldsymbol{e}_v\right]_j \hat{\phi}_{\boldsymbol{\alpha}} \left(\frac{1}{\Delta t_i} \frac{\partial}{\partial \tau} \hat{\psi}_\beta \right) \left[\boldsymbol{e}_\gamma\right]_j d\boldsymbol{\xi} d\tau = \\
& J_{\mathcal{T}_i} J\boldsymbol{\mathcal{X}}_K \sum_{\boldsymbol{\alpha}'\in\boldsymbol{\mathcal{N}}} \sum_{\beta'\in\mathcal{N}} \left(\hat{\omega}_{\boldsymbol{\alpha}'} \hat{\omega}_{\beta'} \left[\hat{\boldsymbol{q}}^{K,i,r+1}\right]_{\boldsymbol{n}lv} \hat{\phi}_{\boldsymbol{n}}(\hat{\boldsymbol{\xi}}_{\boldsymbol{\alpha}'}) \hat{\psi}_l(\hat{\tau}_{\beta'}) \left[\boldsymbol{e}_v\right]_j \dots \right. \\
& \qquad \left. \dots \hat{\phi}_{\boldsymbol{\alpha}}(\hat{\boldsymbol{\xi}}_{\boldsymbol{\alpha}'}) \left(\frac{1}{\Delta t_i} \frac{\partial}{\partial \tau} \hat{\psi}_\beta(\hat{\tau}_{\beta'}) \right) \left[\boldsymbol{e}_\gamma\right]_j \right) = \\
& J_{\mathcal{T}_i} J\boldsymbol{\mathcal{X}}_K \sum_{\boldsymbol{\alpha}'\in\boldsymbol{\mathcal{N}}} \sum_{\beta'\in\mathcal{N}} \left(\hat{\omega}_{\boldsymbol{\alpha}'} \hat{\omega}_{\beta'} \left[\hat{\boldsymbol{q}}^{K,i,r+1}\right]_{\boldsymbol{n}lv} \delta_{\boldsymbol{n}\boldsymbol{\alpha}'} \delta_{l\beta'} \delta_{vj} \delta_{\boldsymbol{\alpha}\boldsymbol{\alpha}'} \left(\frac{1}{\Delta t_i} \frac{\partial}{\partial \tau} \hat{\psi}_\beta(\hat{\tau}_{\beta'}) \right) \delta_{\gamma j} \right) = \\
& J_{\mathcal{T}_i} J\boldsymbol{\mathcal{X}}_K \hat{\omega}_{\boldsymbol{\alpha}} \frac{1}{\Delta t_i} \sum_{\beta'\in\mathcal{N}} \left(\left[\hat{\omega}_{\beta'} \frac{\partial}{\partial \tau} \hat{\psi}_\beta(\hat{\tau}_{\beta'}) \right] \left[\hat{\boldsymbol{q}}^{K,i,r+1}\right]_{\boldsymbol{\alpha}\beta'\gamma} \right) = \\
& J_{\mathcal{T}_i} J\boldsymbol{\mathcal{X}}_K \hat{\omega}_{\boldsymbol{\alpha}} \frac{1}{\Delta t_i} \left[\boldsymbol{K}\right]_{\beta\beta'} \left[\hat{\boldsymbol{q}}^{K,i,r+1}\right]_{\boldsymbol{\alpha}\beta'\gamma} \qquad (2.85)
\end{aligned}
$$

We recall from eq. (2.50) that

$$J_{\mathcal{T}_i} = \Delta t_i, \tag{2.86}$$

so that $J_{\mathcal{T}_i}$ and $1/\Delta t_i$ in eq. (2.85) cancel. In the derivation we made use of the fact that due to the chain rule

$$\frac{\partial}{\partial t}\psi^i_\beta = \frac{\partial}{\partial t}\left(\hat{\psi}_\beta \circ \mathcal{T}_i^{-1}\right) = \left(\frac{\partial}{\partial \tau}\hat{\psi}_\beta\right)\left(\frac{\partial}{\partial t}\mathcal{T}_i^{-1}\right) = \frac{1}{\Delta t_i}\frac{\partial}{\partial \tau}\hat{\psi}_\beta. \tag{2.87}$$

We furthermore introduce the matrix $\boldsymbol{K}$ representing the Reference Element Stiffness Operator given as

$$[\boldsymbol{K}]_{ij} = \hat{\omega}_j \frac{\partial}{\partial \tau}\hat{\psi}_i(\hat{\tau}_j) \tag{2.88}$$

for $i, j \in \mathcal{N}$. A Python script to compute $\boldsymbol{K}$ can be found in Appendix A.

2.12.3 Term S-III

The third term of eq. (2.80) can be rewritten with respect to reference coordinates as

$$\begin{aligned}
&\int_K \left[\left[\hat{\boldsymbol{u}}^{K,i}\right]_{nv} \boldsymbol{\phi}^K_{nv}\right]_j \left[\boldsymbol{\theta}^{K,i}_{\alpha,\beta,\gamma}\Big|_{t_i}\right]_j dx = \\
&\int_K \left[\hat{\boldsymbol{u}}^{K,i}\right]_{nv} \phi^K_n \left[\boldsymbol{e}_v\right]_j \phi^K_\alpha \left.\psi^i_\beta\right|_{t_i} \left[\boldsymbol{e}_\gamma\right]_j dx = \\
&J_{\boldsymbol{x}_K} \int_{\hat{K}} \left[\hat{\boldsymbol{u}}^{K,i}\right]_{nv} \hat{\phi}_n \left[\boldsymbol{e}_v\right]_j \hat{\phi}_\alpha \left.\hat{\psi}_\beta\right|_0 \left[\boldsymbol{e}_\gamma\right]_j d\xi = \\
&J_{\boldsymbol{x}_K} \sum_{\alpha' \in \mathcal{N}} \left(\hat{\omega}_{\alpha'} \left[\hat{\boldsymbol{u}}^{K,i}\right]_{nv} \hat{\phi}_n(\xi_{\alpha'}) \left[\boldsymbol{e}_v\right]_j \hat{\phi}_\alpha(\xi_{\alpha'}) \left.\hat{\psi}_\beta\right|_0 \left[\boldsymbol{e}_\gamma\right]_j\right) = \\
&J_{\boldsymbol{x}_K} \sum_{\alpha' \in \mathcal{N}} \left(\hat{\omega}_{\alpha'} \left[\hat{\boldsymbol{u}}^{K,i}\right]_{nv} \delta_{n\alpha'}\delta_{vj}\delta_{\alpha\alpha'} \left.\hat{\psi}_\beta\right|_0 \delta_{\gamma j}\right) = \\
&J_{\boldsymbol{x}_K} \hat{\omega}_\alpha \left[\left.\hat{\psi}_\beta\right|_0\right] \left[\hat{\boldsymbol{u}}^{K,i}\right]_{\alpha\gamma} = \\
&J_{\boldsymbol{x}_K} \hat{\omega}_\alpha \left[\boldsymbol{l}\right]_\beta \left[\hat{\boldsymbol{u}}^{K,i}\right]_{\alpha\gamma},
\end{aligned} \tag{2.89}$$

where we define the vector $\boldsymbol{l}$ representing the Left Reference Element Flux Operator as

$$[\boldsymbol{l}]_i = \left.\hat{\psi}_i\right|_0 \tag{2.90}$$

for $i \in \mathcal{N}$. A Python script to compute $\boldsymbol{l}$ can be found in Appendix A.

2.12.4 Term S-IV

Let us define $\mathcal{D}_k := \{d \in \mathcal{D} \mid d \neq k\}$. Then third term of eq. (2.80) can be rewritten with respect to reference coordinates as

$$\int_{t_i}^{t_i+\Delta t_i} \int_K \sum_{k \in \mathcal{D}} \frac{\partial}{\partial x_k} \left[\left[\hat{\boldsymbol{F}}^{K,i}\right]_{nlvk} \boldsymbol{\theta}^{K,i}_{n,l,v}\right]_{jk} \left[\boldsymbol{\theta}^{K,i}_{\alpha,\beta,\gamma}\right]_j dx dt =$$

$$\int_{t_i}^{t_i+\Delta t_i}\int_K \sum_{k\in\mathcal{D}} \left[\hat{\boldsymbol{F}}^{K,i}\right]_{\boldsymbol{n}lvk} \phi^K_{\boldsymbol{\alpha}}\psi^i_{\beta}\left[\boldsymbol{e}_\gamma\right]_j \left(\prod_{d\in\mathcal{D}_k}\psi^K_{\boldsymbol{n}_d}(\boldsymbol{x}_d)\right)\psi^i_l(t)\left[\boldsymbol{e}_v\right]_j \left(\frac{\partial}{\partial \boldsymbol{x}_k}\psi^K_{\boldsymbol{n}_k}(\boldsymbol{x}_k)\right) d\boldsymbol{x}dt =$$

$$J_{\mathcal{T}_i}J_{\boldsymbol{\mathcal{X}}_K}\sum_{k\in\mathcal{D}}\int_0^1\int_{\hat{K}}\left[\hat{\boldsymbol{F}}^{K,i}\right]_{\boldsymbol{n}lvk}\hat{\phi}_{\boldsymbol{\alpha}}\hat{\psi}_\beta\left[\boldsymbol{e}_\gamma\right]_j\left(\prod_{d\in\mathcal{D}_k}\hat{\psi}_{\boldsymbol{n}_d}(\boldsymbol{\xi}_d)\right)\hat{\psi}_l(t)\left[\boldsymbol{e}_v\right]_j\dots$$

$$\dots\left(\frac{1}{\left[\Delta\boldsymbol{x}^K\right]_k}\frac{\partial}{\partial\boldsymbol{\xi}_k}\hat{\psi}_{\boldsymbol{n}_k}(\boldsymbol{\xi}_k)\right)d\boldsymbol{\xi}d\tau =$$

$$J_{\mathcal{T}_i}J_{\boldsymbol{\mathcal{X}}_K}\sum_{k\in\mathcal{D}}\sum_{\boldsymbol{\alpha}'\in\mathcal{N}}\sum_{\beta'\in\mathcal{N}}\left(\hat{\omega}_{\boldsymbol{\alpha}'}\hat{\omega}_{\beta'}\left[\hat{\boldsymbol{F}}^{K,i}\right]_{\boldsymbol{n}lvk}\hat{\phi}_{\boldsymbol{\alpha}}(\hat{\boldsymbol{\xi}}_{\boldsymbol{\alpha}'})\hat{\psi}_\beta(\hat{\tau}_{\beta'})\left[\boldsymbol{e}_\gamma\right]_j\left(\prod_{d\in\mathcal{D}_k}\hat{\psi}_{\boldsymbol{n}_d}\left(\hat{\boldsymbol{\xi}}_{\boldsymbol{\alpha}'_d}\right)\right)\dots\right.$$

$$\left.\dots\hat{\psi}_l(\hat{\tau}_{\beta'})\left[\boldsymbol{e}_v\right]_j\left(\frac{1}{\left[\Delta\boldsymbol{x}^K\right]_k}\frac{\partial}{\partial\boldsymbol{\xi}_k}\hat{\psi}_{\boldsymbol{n}_k}\left(\hat{\boldsymbol{\xi}}_{\boldsymbol{\alpha}'_k}\right)\right)\right) =$$

$$J_{\mathcal{T}_i}J_{\boldsymbol{\mathcal{X}}_K}\sum_{k\in\mathcal{D}}\sum_{\boldsymbol{\alpha}'\in\mathcal{N}}\sum_{\beta'\in\mathcal{N}}\left(\hat{\omega}_{\boldsymbol{\alpha}'}\hat{\omega}_{\beta'}\left[\hat{\boldsymbol{F}}^{K,i}\right]_{\boldsymbol{n}lvk}\delta_{\boldsymbol{\alpha}\boldsymbol{\alpha}'}\delta_{\beta\beta'}\delta_{\gamma j}\left(\prod_{d\in\mathcal{D}_k}\delta_{\boldsymbol{n}_d\boldsymbol{\alpha}'_d}\right)\dots\right.$$

$$\left.\dots\delta_{l\beta'}\delta_{vj}\left(\frac{1}{\left[\Delta\boldsymbol{x}^K\right]_k}\frac{\partial}{\partial\boldsymbol{\xi}_k}\hat{\psi}_{\boldsymbol{n}_k}\left(\hat{\boldsymbol{\xi}}_{\boldsymbol{\alpha}'_k}\right)\right)\right) =$$

$$J_{\mathcal{T}_i}J_{\boldsymbol{\mathcal{X}}_K}\hat{\omega}_\beta\sum_{k\in\mathcal{D}}\left(\frac{1}{\left[\Delta\boldsymbol{x}^K\right]_k}\sum_{\alpha'_k\in\mathcal{N}}\hat{\omega}_{\boldsymbol{\alpha}_{0:k-1,k+1:D-1}}\left(\hat{\omega}_{\alpha'_k}\left(\frac{\partial}{\partial\boldsymbol{\xi}_k}\hat{\psi}_{\boldsymbol{\alpha}_k}\left(\hat{\xi}_{\alpha'_k}\right)\right)\dots\right.\right.$$

$$\left.\left.\dots\left[\hat{\boldsymbol{F}}^{K,i}\right]_{[\boldsymbol{\alpha}_{0:k-1},\alpha'_k,\boldsymbol{\alpha}_{k+1:D-1}]\beta\gamma k}\right)\right) =$$

$$J_{\mathcal{T}_i}J_{\boldsymbol{\mathcal{X}}_K}\hat{\omega}_\beta\sum_{k\in\mathcal{D}}\left(\frac{1}{\left[\Delta\boldsymbol{x}^K\right]_k}\sum_{\alpha'_k\in\mathcal{N}}\hat{\omega}_{\boldsymbol{\alpha}_{0:k-1,k+1:D-1}}\left([\boldsymbol{K}]_{\boldsymbol{\alpha}_k\alpha'_k}\dots\left[\hat{\boldsymbol{F}}^{K,i}\right]_{[\boldsymbol{\alpha}_{0:k-1},\alpha'_k,\boldsymbol{\alpha}_{k+1:D-1}]\beta\gamma k}\right)\right), \tag{2.91}$$

where we used that

$$\frac{\partial}{\partial\boldsymbol{x}_k}\boldsymbol{\theta}^{K,i}_{\boldsymbol{n},l,v}(\boldsymbol{x},t) = \left(\frac{\partial}{\partial\boldsymbol{x}_k}\phi^K_{\boldsymbol{n}}(\boldsymbol{x})\right)\psi^i_l(t)\boldsymbol{e}_v = \left(\frac{\partial}{\partial\boldsymbol{x}_k}\prod_{d\in\mathcal{D}}\psi^K_{\boldsymbol{n}_d}(\boldsymbol{x}_d)\right)\psi^i_l(t)\boldsymbol{e}_v =$$

$$\left(\prod_{d\in\mathcal{D}_k}\psi^K_{\boldsymbol{n}_d}(\boldsymbol{x}_d)\right)\left(\frac{\partial}{\partial\boldsymbol{x}_k}\psi^K_{\boldsymbol{n}_k}(\boldsymbol{x}_k)\right)\psi^i_l(t)\boldsymbol{e}_v =$$

$$\left(\prod_{d\in\mathcal{D}_k}\psi^K_{\boldsymbol{n}_d}(\boldsymbol{x}_d)\right)\left(\frac{\partial}{\partial\boldsymbol{x}_k}\hat{\psi}_{\boldsymbol{n}_k}\left(\left[\boldsymbol{\mathcal{X}}^{-1}_K(\boldsymbol{x})\right]_k\right)\right)\psi^i_l(t)\boldsymbol{e}_v =$$

$$\left(\prod_{d\in\mathcal{D}_k}\psi^K_{\boldsymbol{n}_d}(\boldsymbol{x}_d)\right)\left(\left(\frac{\partial}{\partial\boldsymbol{\xi}_k}\hat{\psi}_{\boldsymbol{n}_k}\left(\left[\boldsymbol{\mathcal{X}}^{-1}_K(\boldsymbol{x})\right]_k\right)\right)\left(\frac{\partial}{\partial\boldsymbol{x}_k}\left[\boldsymbol{\mathcal{X}}^{-1}_K(\boldsymbol{x})\right]_k\right)\right)\psi^i_l(t)\boldsymbol{e}_v =$$

$$\left(\prod_{d\in\mathcal{D}_k}\psi^K_{\boldsymbol{n}_d}(\boldsymbol{x}_d)\right)\left(\frac{1}{\left[\Delta\boldsymbol{x}^K\right]_k}\frac{\partial}{\partial\boldsymbol{\xi}_k}\hat{\boldsymbol{\phi}}_{\boldsymbol{n}_k}\left(\left[\boldsymbol{\mathcal{X}}_K^{-1}(\boldsymbol{x})\right]_k\right)\right)\psi^i_l(t)\boldsymbol{e}_v. \tag{2.92}$$

2.12.5 Term S-V

The fifth term of eq. (2.80) can be rewritten with respect to reference coordinates as follows:

$$\begin{aligned}
&\int_{t_i}^{t_i+\Delta t_i}\int_K\left[\left[\hat{\boldsymbol{s}}^{K,i}\right]_{\boldsymbol{n}lv}\boldsymbol{\theta}^{K,i}_{\boldsymbol{n},l,v}\right]_j\left[\boldsymbol{\theta}^{K,i}_{\boldsymbol{\alpha},\beta,\gamma}\right]_j d\boldsymbol{x}dt = \\
&J_{\mathcal{T}_i}J_{\boldsymbol{\mathcal{X}}_K}\int_0^1\int_{\hat{K}}\left[\hat{\boldsymbol{s}}^{K,i}\right]_{\boldsymbol{n}lv}\hat{\boldsymbol{\phi}}_{\boldsymbol{n}}\hat{\psi}_l\left[\boldsymbol{e}_v\right]_j\hat{\boldsymbol{\phi}}_{\boldsymbol{\alpha}}\hat{\psi}_l\left[\boldsymbol{e}_\gamma\right]_j d\boldsymbol{\xi}d\tau = \\
&J_{\mathcal{T}_i}J_{\boldsymbol{\mathcal{X}}_K}\sum_{\boldsymbol{\alpha}'\in\boldsymbol{\mathcal{N}}}\sum_{\beta'\in\mathcal{N}}\left(\hat{\omega}_{\boldsymbol{\alpha}'}\hat{\omega}_{\beta'}\left[\hat{\boldsymbol{s}}^{K,i}\right]_{\boldsymbol{n}lv}\hat{\boldsymbol{\phi}}_{\boldsymbol{n}}(\hat{\boldsymbol{\xi}}_{\boldsymbol{\alpha}'})\hat{\psi}_l(\hat{\tau}_{\beta'})\left[\boldsymbol{e}_v\right]_j\hat{\boldsymbol{\phi}}_{\boldsymbol{\alpha}}(\boldsymbol{\xi}_{\boldsymbol{\alpha}'})\hat{\psi}_\beta(\hat{\tau}_{\beta'})\left[\boldsymbol{e}_\gamma\right]_j\right) = \\
&J_{\mathcal{T}_i}J_{\boldsymbol{\mathcal{X}}_K}\sum_{\boldsymbol{\alpha}'\in\boldsymbol{\mathcal{N}}}\sum_{\beta'\in\mathcal{N}}\left(\hat{\omega}_{\boldsymbol{\alpha}'}\hat{\omega}_{\beta'}\left[\hat{\boldsymbol{s}}^{K,i}\right]_{\boldsymbol{n}lv}\delta_{\boldsymbol{n}\boldsymbol{\alpha}'}\delta_{l\beta'}\delta_{vj}\delta_{\boldsymbol{\alpha}\boldsymbol{\alpha}'}\delta_{\beta\beta'}\delta_{\gamma j}\right) = \\
&J_{\mathcal{T}_i}J_{\boldsymbol{\mathcal{X}}_K}\hat{\omega}_{\boldsymbol{\alpha}}\hat{\omega}_\beta\left[\hat{\boldsymbol{s}}^{K,i}\right]_{\boldsymbol{\alpha}\beta\gamma}.
\end{aligned} \tag{2.93}$$

2.12.6 The Complete Fixed-point Iteration ("Picard Loop")

Now collecting the results from eqs. (2.82), (2.85), (2.89), (2.91) and (2.93), plugging them back into eq. (2.80) and division of the resulting equation by $J_{\boldsymbol{\mathcal{X}}_K}\hat{\omega}_{\boldsymbol{\alpha}}$ yields

$$\begin{aligned}
&\left[\boldsymbol{R}-\boldsymbol{K}\right]_{\beta\beta'}\left[\hat{\boldsymbol{q}}^{K,i,r+1}\right]_{\boldsymbol{\alpha}\beta'\gamma} = \left[\boldsymbol{l}\right]_\beta\left[\hat{\boldsymbol{u}}^{K,i}\right]_{\boldsymbol{\alpha}\gamma} - \\
&J_{\mathcal{T}_i}\hat{\omega}_\beta\sum_{k\in\mathcal{D}}\left(\frac{1}{\left[\Delta\boldsymbol{x}^K\right]_k}\frac{1}{\hat{\omega}_{\boldsymbol{\alpha}_k}}\sum_{\alpha'_k\in\mathcal{N}}\left(\left[\boldsymbol{K}\right]_{\boldsymbol{\alpha}_k\alpha'_k}\left[\hat{\boldsymbol{F}}^{K,i}\right]_{[\boldsymbol{\alpha}_{0:k-1},\alpha'_k,\boldsymbol{\alpha}_{k+1:D-1}]\beta\gamma k}\right)\right) + \\
&J_{\mathcal{T}_i}\hat{\omega}_\beta\left[\hat{\boldsymbol{s}}^{K,i}\right]_{\boldsymbol{\alpha}\beta\gamma},
\end{aligned} \tag{2.94}$$

which has to hold for all $\boldsymbol{\alpha}\in\boldsymbol{\mathcal{N}}$, $\beta\in\mathcal{N}$ and $\gamma\in\mathcal{V}$. To speed up the computation of the new iterate $\hat{\boldsymbol{q}}^{K,i,r+1}$ we can invert the matrix on the left-hand side prior to the simulation to obtain the iteration matrix $\tilde{\boldsymbol{K}} := (\boldsymbol{R}-\boldsymbol{K})^{-1}$. A Python script to compute $\tilde{\boldsymbol{K}}$ can be found in Appendix A.

A possible termination criterion could be $\Delta < \varepsilon$, where $\varepsilon > 0$ is a suitable constant related to the desired accuracy of the iteration, e.g. $\varepsilon = 10^{-7}$ and the element update in squared norm Δ^2 is defined as follows:

$$\Delta^2 = \sum_{\boldsymbol{n}\in\boldsymbol{\mathcal{N}}}\sum_{l\in\mathcal{N}}\sum_{v\in\mathcal{V}}\left(\left[\hat{\boldsymbol{q}}^{K,i,r+1}\right]_{\boldsymbol{n}lv} - \left[\hat{\boldsymbol{q}}^{K,i,r}\right]_{\boldsymbol{n}lv}\right)^2. \tag{2.95}$$

For linear homogeneous scalar hyperbolic balance laws and neglecting floating point errors it can be proven that the iteration converges after at most N steps (see [22] for details).

2.13 Update Scheme for the Time-discrete Solution

Now that we have developed a method to compute the space-time predictor, we can go back to the original one-step, cell-local update scheme given in eq. (2.34). Inserting the local space-time predictor $\tilde{\boldsymbol{q}}_h^{K,i}$ yields

$$\begin{aligned}
&\int_K \left[\tilde{\boldsymbol{u}}_h^{K,i}\Big|_{t_i+\Delta t_i} \right]_v \left[\boldsymbol{w}_h^K \right]_v d\boldsymbol{x} = \int_K \left[\tilde{\boldsymbol{u}}_h^{K,i}\Big|_{t_i} \right]_v \left[\boldsymbol{w}_h^K \right]_v d\boldsymbol{x} + \\
&\int_{t_i}^{t_i+\Delta t_i} \int_K \left[\boldsymbol{F}(\tilde{\boldsymbol{q}}_h^{K,i}) \right]_{vd} \frac{\partial}{\partial \boldsymbol{x}_d} \left[\boldsymbol{w}_h^K \right]_v d\boldsymbol{x}dt + \\
&\int_{t_i}^{t_i+\Delta t_i} \int_K \left[\boldsymbol{s}\left(\hat{\boldsymbol{q}}^{K,i} \right) \right]_v \left[\boldsymbol{w}_h^K \right]_v d\boldsymbol{x}dt - \\
&\int_{t_i}^{t_i+\Delta t_i} \int_{\partial K} \left[\boldsymbol{\mathcal{G}}\left(\tilde{\boldsymbol{q}}_h^{K,i}, \tilde{\boldsymbol{q}}_h^{K',i}, \boldsymbol{n} \right) \right]_v \left[\boldsymbol{w}_h^K \right]_v ds(\boldsymbol{x})dt,
\end{aligned} \tag{2.96}$$

which has to hold for all $v \in \mathcal{V}$, $K \in \mathcal{K}_h$, $\boldsymbol{w}_h \in \mathbb{W}_h$ and $i \in \mathcal{I}$.

Making use of the bases we derived earlier the cell-local solution $\tilde{\boldsymbol{u}}_h^{K,i}$ at times t_i and $t_i + \Delta t_i$ can be represented by tensors of coefficients $\hat{\boldsymbol{u}}^{K,i}$ and $\hat{\boldsymbol{u}}^{K,i+1}$ as

$$\tilde{\boldsymbol{u}}_h^{K,i}\Big|_{t_i} = \left[\hat{\boldsymbol{u}}^{K,i} \right]_{n v} \boldsymbol{\phi}_{n,v}^K \tag{2.97}$$

and

$$\tilde{\boldsymbol{u}}_h^{K,i}\Big|_{t_i+\Delta t_i} = \left[\hat{\boldsymbol{u}}^{K,i+1} \right]_{n v} \boldsymbol{\phi}_{n,v}^K, \tag{2.98}$$

respectively. Inserting eqs. (2.97) and (2.98) and the ansatz for the space-time predictor (2.75) into eq. (2.96) yields

$$\begin{aligned}
&\underbrace{\int_K \left[\left[\hat{\boldsymbol{u}}^{K,i+1} \right]_{n v} \boldsymbol{\phi}_{n,v}^K \right]_j \left[\boldsymbol{\phi}_{\alpha,\gamma}^K \right]_j d\boldsymbol{x}}_{\text{U-I}} = \underbrace{\int_K \left[\left[\hat{\boldsymbol{u}}^{K,i} \right]_{n v} \boldsymbol{\phi}_{n,v}^K \right]_j \left[\boldsymbol{\phi}_{\alpha,\gamma}^K \right]_j}_{\text{U-II}} + \\
&\underbrace{\int_{t_i}^{t_i+\Delta t_i} \int_K \left[\boldsymbol{F}\left(\left[\hat{\boldsymbol{q}}^{K,i} \right]_{n l v} \boldsymbol{\theta}_{n,l,v}^{K,i} \right) \right]_{jk} \frac{\partial}{\partial \boldsymbol{x}_k} \left[\boldsymbol{\phi}_{\alpha,\gamma}^K \right]_j d\boldsymbol{x}dt}_{\text{U-III}} + \\
&\underbrace{\int_{t_i}^{t_i+\Delta t_i} \int_K \left[\boldsymbol{s}\left(\left[\hat{\boldsymbol{q}}^{K,i} \right]_{n l v} \boldsymbol{\theta}_{n,l,v}^{K,i} \right) \right]_j \left[\boldsymbol{\phi}_{\alpha,\gamma}^K \right]_j d\boldsymbol{x}dt}_{\text{U-IV}} - \\
&\underbrace{\int_{t_i}^{t_i+\Delta t_i} \int_{\partial K} \left[\boldsymbol{\mathcal{G}}\left(\hat{\boldsymbol{q}}^{K,i}, \hat{\boldsymbol{q}}^{K',i}, \boldsymbol{n} \right) \right]_j \left[\boldsymbol{\phi}_{\alpha,\gamma}^K \right]_j ds(\boldsymbol{x})dt}_{\text{U-V}},
\end{aligned} \tag{2.99}$$

which we require to hold for all $\boldsymbol{\alpha} \in \boldsymbol{\mathcal{N}}$, $\gamma \in \mathcal{V}$, $K \in \mathcal{K}_h$ and $i \in \mathcal{I}$. In the following we will again proceed by simplifying each term in reference coordinates separately and then in the end assemble the complete fully-discrete update scheme.

2.13.1 Term U-I

The first term of eq. (2.99) can be rewritten with respect to reference coordinates as follows:

$$\begin{aligned}
&\int_K \left[\left[\hat{\boldsymbol{u}}^{K,i+1} \right]_{nv} \boldsymbol{\phi}^K_{n,v} \right]_j \left[\boldsymbol{\phi}^K_{\boldsymbol{\alpha},\gamma} \right]_j \, d\boldsymbol{x} = \\
&\int_K \left[\left[\hat{\boldsymbol{u}}^{K,i+1} \right]_{nv} \phi^K_n \boldsymbol{e}_v \right]_j \left[\phi^K_{\boldsymbol{\alpha}} \boldsymbol{e}_\gamma \right]_j \, d\boldsymbol{x} = \\
&J_{\boldsymbol{\mathcal{X}}_K} \int_{\hat{K}} \left[\left[\hat{\boldsymbol{u}}^{K,i+1} \right]_{nv} \hat{\phi}_n \boldsymbol{e}_v \right]_j \left[\hat{\phi}_{\boldsymbol{\alpha}} \boldsymbol{e}_\gamma \right]_j \, d\boldsymbol{\xi} = \\
&J_{\boldsymbol{\mathcal{X}}_K} \sum_{\boldsymbol{\alpha}' \in \boldsymbol{\mathcal{N}}} \left(\hat{\omega}_{\boldsymbol{\alpha}'} \left[\hat{\boldsymbol{u}}^{K,i+1} \right]_{nv} \hat{\phi}_n(\hat{\boldsymbol{\xi}}_{\boldsymbol{\alpha}'}) \left[\boldsymbol{e}_v \right]_j \hat{\phi}_{\boldsymbol{\alpha}}(\hat{\boldsymbol{\xi}}_{\boldsymbol{\alpha}'}) \left[\boldsymbol{e}_\gamma \right]_j \right) = \\
&J_{\boldsymbol{\mathcal{X}}_K} \sum_{\boldsymbol{\alpha}' \in \boldsymbol{\mathcal{N}}} \left(\hat{\omega}_{\boldsymbol{\alpha}'} \left[\hat{\boldsymbol{u}}^{K,i+1} \right]_{nv} \delta_{n\boldsymbol{\alpha}'} \delta_{vj} \delta_{\boldsymbol{\alpha}\boldsymbol{\alpha}'} \delta_{\gamma j} \right) = \\
&J_{\boldsymbol{\mathcal{X}}_K} \hat{\omega}_{\boldsymbol{\alpha}} \left[\hat{\boldsymbol{u}}^{K,i+1} \right]_{\boldsymbol{\alpha}\gamma} .
\end{aligned} \tag{2.100}$$

2.13.2 Term U-II

Analogously to the first term of eq. (2.99), the second term can be rewritten as follows:

$$\begin{aligned}
&\int_K \left[\left[\hat{\boldsymbol{u}}^{K,i} \right]_{n,v} \boldsymbol{\phi}^K_{nv} \right]_j \left[\boldsymbol{\phi}^K_{\boldsymbol{\alpha},\gamma} \right]_j \, d\boldsymbol{x} = \\
&J_{\boldsymbol{\mathcal{X}}_K} \hat{\omega}_{\boldsymbol{\alpha}} \left[\hat{\boldsymbol{u}}^{K,i} \right]_{\boldsymbol{\alpha}\gamma} .
\end{aligned} \tag{2.101}$$

2.13.3 Term U-III

The third term of eq. (2.99) can be rewritten with respect to reference coordinates as

$$\begin{aligned}
&\int_{t_i}^{t_i+\Delta t_i} \int_K \left[\boldsymbol{F} \left(\left[\hat{\boldsymbol{q}}^{K,i} \right]_{nlv} \boldsymbol{\theta}^{K,i}_{n,l,v} \right) \right]_{jk} \frac{\partial}{\partial \boldsymbol{x}_k} \left[\boldsymbol{\phi}^K_{\boldsymbol{\alpha},\gamma} \right]_j \, d\boldsymbol{x}dt = \\
&\int_{t_i}^{t_i+\Delta t_i} \int_K \left[\boldsymbol{F} \left(\left[\hat{\boldsymbol{q}}^{K,i} \right]_{nlv} \phi^K_n \psi^i_l \boldsymbol{e}_v \right) \right]_{jk} \frac{\partial}{\partial \boldsymbol{x}_k} \left(\prod_{d=0}^{D-1} \psi^K_{\boldsymbol{\alpha}_d}(\boldsymbol{x}_d) \right) \left[\boldsymbol{e}_\gamma \right]_j \, d\boldsymbol{x}dt = \\
&\sum_{k \in \mathcal{D}} \int_{t_i}^{t_i+\Delta t_i} \int_K \left[\boldsymbol{F} \left(\left[\hat{\boldsymbol{q}}^{K,i} \right]_{nvl} \phi^K_n \psi^i_l \boldsymbol{e}_v \right) \right]_{jk} \left(\prod_{d \in \mathcal{D}_k} \psi^K_{[\boldsymbol{\alpha}]_d}([\boldsymbol{x}]_d) \right) \frac{1}{[\Delta \boldsymbol{x}^K]_k} \cdots \\
&\qquad \cdots \frac{\partial}{\partial \boldsymbol{\xi}_k} \hat{\psi}_{\boldsymbol{\alpha}_k} \left(\left[\boldsymbol{\mathcal{X}}_K(\boldsymbol{x}) \right]_k \right) \left[\boldsymbol{e}_\gamma \right]_j \, d\boldsymbol{x}dt = \\
&J_{\mathcal{T}_i} J_{\boldsymbol{\mathcal{X}}_K} \sum_{k \in \mathcal{D}} \int_0^1 \int_{\hat{K}} \left[\boldsymbol{F} \left(\left[\hat{\boldsymbol{q}}^{K,i} \right]_{nlv} \hat{\phi}_n \hat{\psi}_l \boldsymbol{e}_v \right) \right]_{kj} \left(\prod_{d \in \mathcal{D}_k} \hat{\psi}^K_{\boldsymbol{\alpha}_d}(\boldsymbol{\xi}_d) \right) \frac{1}{[\Delta \boldsymbol{x}^K]_k} \cdots \\
&\qquad \cdots \frac{\partial}{\partial \boldsymbol{\xi}_k} \hat{\psi}_{\boldsymbol{\alpha}_k} (\boldsymbol{\xi}_k) \left[\boldsymbol{e}_\gamma \right]_j \, d\boldsymbol{\xi}d\tau =
\end{aligned}$$

$$
\begin{aligned}
J_{\mathcal{T}_i} J_{\boldsymbol{\mathcal{X}}_K} \sum_{k\in\mathcal{D}} \sum_{\boldsymbol{\alpha}'\in\boldsymbol{\mathcal{N}}} \sum_{\beta'\in\mathcal{N}} \Bigg(\hat{\omega}_{\boldsymbol{\alpha}'} \hat{\omega}_{\beta'} \left[\boldsymbol{F}\left(\left[\hat{\boldsymbol{q}}^{K,i}\right]_{\boldsymbol{n}lv} \hat{\phi}_{\boldsymbol{n}}(\boldsymbol{\xi}_{\boldsymbol{\alpha}'}) \hat{\psi}(\hat{\tau}_{\beta'}) \boldsymbol{e}_v \right) \right]_{jk} \dots \\
\dots \left(\prod_{d\in\mathcal{D}_k} \hat{\psi}_{\boldsymbol{\alpha}_d}(\boldsymbol{\xi}_{\boldsymbol{\alpha}'_d}) \right) \frac{1}{\left[\Delta \boldsymbol{x}^K\right]_k} \frac{\partial}{\partial \boldsymbol{\xi}_k} \hat{\psi}_{\boldsymbol{\alpha}_k}\left(\boldsymbol{\xi}_{\boldsymbol{\alpha}'_k}\right) \left[\boldsymbol{e}_\gamma\right]_j \Bigg) = \\
J_{\mathcal{T}_i} J_{\boldsymbol{\mathcal{X}}_K} \hat{\omega}_{\boldsymbol{\alpha}} \sum_{k\in\mathcal{D}} \Bigg(\sum_{\alpha'_k\in\mathcal{N}} \sum_{\beta'\in\mathcal{N}} \Bigg(\frac{\hat{\omega}_{\beta'}}{\hat{\omega}_{\alpha'_k}} \frac{1}{\left[\Delta \boldsymbol{x}^K\right]_k} \frac{\partial}{\partial \boldsymbol{\xi}_k} \hat{\psi}_{\alpha'_k}\left(\hat{\boldsymbol{\xi}}_{\alpha'_k}\right) \dots \\
\dots \left[\boldsymbol{F}\left(\left[\hat{\boldsymbol{q}}^{K,i}\right]_{\left[\boldsymbol{\alpha}_{0:k-1},\alpha'_k,\boldsymbol{\alpha}_{k+1:D-1}\right]\beta' v} \boldsymbol{e}_v \right) \right]_{\gamma k} \Bigg) \Bigg) = \\
J_{\mathcal{T}_i} J_{\boldsymbol{\mathcal{X}}_K} \hat{\omega}_{\boldsymbol{\alpha}} \sum_{k\in\mathcal{D}} \Bigg(\sum_{\alpha'_k\in\mathcal{N}} \sum_{\beta'\in\mathcal{N}} \Bigg(\frac{1}{\hat{\omega}_{\alpha'_k}} \frac{1}{\left[\Delta \boldsymbol{x}^K\right]_k} \left[\boldsymbol{K}\right]_{\alpha'_k k} \dots \\
\dots \left[\boldsymbol{F}\left(\left[\hat{\boldsymbol{q}}^{K,i}\right]_{\left[\boldsymbol{\alpha}_{0:k-1},\alpha'_k,\boldsymbol{\alpha}_{k+1:D-1}\right]\beta' v} \boldsymbol{e}_v \right) \right]_{\gamma k} \Bigg) \Bigg),
\end{aligned}
\tag{2.102}
$$

where we made use of the fact that due to the chain rule it holds that

$$
\begin{aligned}
\frac{\partial}{\partial \boldsymbol{x}_k} \left(\prod_{d\in\mathcal{D}} \psi^K_{\boldsymbol{\alpha}_d}(\boldsymbol{x}_d) \right) = \left(\prod_{d\in\mathcal{D}_k} \psi^K_{\boldsymbol{\alpha}_d}(\boldsymbol{x}_d) \right) \frac{\partial}{\partial \boldsymbol{x}_k} \psi^K_{\boldsymbol{\alpha}_k}(\boldsymbol{x}_k) = \\
\left(\prod_{d\in\mathcal{D}_k} \psi^K_{\boldsymbol{\alpha}_d}(\boldsymbol{x}_d) \right) \frac{\partial}{\partial \boldsymbol{\xi}_j} \hat{\psi}_{\boldsymbol{\alpha}_k}\left(\left[\boldsymbol{\mathcal{X}}_K(\boldsymbol{x})\right]_k \right) \frac{\partial}{\partial \boldsymbol{x}_k} \left[\boldsymbol{\mathcal{X}}_K(\boldsymbol{x})\right]_j = \\
\left(\prod_{d\in\mathcal{D}_k} \psi^K_{\boldsymbol{\alpha}_d}(\boldsymbol{x}_d) \right) \frac{\partial}{\partial \boldsymbol{\xi}_j} \hat{\psi}_{\boldsymbol{\alpha}_k}\left(\left[\boldsymbol{\mathcal{X}}_K(\boldsymbol{x})\right]_k \right) \frac{1}{\left[\Delta \boldsymbol{x}^K\right]_k} \delta_k j = \\
\left(\prod_{d\in\mathcal{D}_k} \psi^K_{\boldsymbol{\alpha}_d}(\boldsymbol{x}_d) \right) \frac{1}{\left[\Delta \boldsymbol{x}^K\right]_k} \frac{\partial}{\partial \boldsymbol{\xi}_k} \hat{\psi}_{\boldsymbol{\alpha}_k}\left(\left[\boldsymbol{\mathcal{X}}_K(\boldsymbol{x})\right]_k \right).
\end{aligned}
\tag{2.103}
$$

2.13.4 Term U-IV

The fourth term of eq. (2.99) can be rewritten with respect to reference coordinates as follows:

$$
\begin{aligned}
\int_{t_i}^{t_i+\Delta t_i} \int_K \left[\boldsymbol{s}\left(\left[\hat{\boldsymbol{q}}^{K,i}\right]_{\boldsymbol{n}lv} \boldsymbol{\theta}^{K,i}_{\boldsymbol{n},l,v} \right) \right]_j \left[\boldsymbol{\phi}^K_{\boldsymbol{\alpha},\gamma}\right]_j \, d\boldsymbol{x}dt = \\
\int_{t_i}^{t_i+\Delta t_i} \int_K \left[\boldsymbol{s}\left(\left[\hat{\boldsymbol{q}}^{K,i}\right]_{\boldsymbol{n}lv} \phi^K_{\boldsymbol{n}} \psi^i_l \boldsymbol{e}_v \right) \right]_j \phi^K_{\boldsymbol{\alpha}} \left[\boldsymbol{e}_\gamma\right]_j \, d\boldsymbol{x}dt =
\end{aligned}
$$

$$
\begin{aligned}
&J_{\mathcal{T}_i} J_{\boldsymbol{\mathcal{X}}_K} \int_0^1 \int_{\hat{K}} \left[\boldsymbol{s} \left(\left[\hat{\boldsymbol{q}}^{K,i} \right]_{\boldsymbol{n}lv} \hat{\phi}_{\boldsymbol{n}} \hat{\psi}_l \boldsymbol{e}_v \right) \right]_j \hat{\phi}_{\boldsymbol{\alpha}} \left[\boldsymbol{e}_\gamma \right]_j d\boldsymbol{\xi} d\tau = \\
&J_{\mathcal{T}_i} J_{\boldsymbol{\mathcal{X}}_K} \sum_{\boldsymbol{\alpha}' \in \boldsymbol{\mathcal{N}}} \sum_{\beta' \in \mathcal{N}} \left(\hat{\omega}_{\boldsymbol{\alpha}'} \hat{\omega}_{\beta'} \left[\boldsymbol{s} \left(\left[\hat{\boldsymbol{q}}^{K,i} \right]_{\boldsymbol{n}lv} \hat{\phi}_{\boldsymbol{n}}(\hat{\boldsymbol{\xi}}_{\boldsymbol{\alpha}'}) \hat{\psi}_l(\hat{\tau}_{\beta'}) \boldsymbol{e}_v \right) \right]_j \hat{\phi}_{\boldsymbol{\alpha}}(\hat{\boldsymbol{\xi}}_{\boldsymbol{\alpha}'}) \left[\boldsymbol{e}_\gamma \right]_j \right) = \\
&J_{\mathcal{T}_i} J_{\boldsymbol{\mathcal{X}}_K} \sum_{\boldsymbol{\alpha}' \in \boldsymbol{\mathcal{N}}} \sum_{\beta' \in \mathcal{N}} \left(\hat{\omega}_{\boldsymbol{\alpha}'} \hat{\omega}_{\beta'} \left[\boldsymbol{s} \left(\left[\hat{\boldsymbol{q}}^{K,i} \right]_{\boldsymbol{n}lv} \delta_{\boldsymbol{n}\boldsymbol{\alpha}'} \delta_{l\beta'} \boldsymbol{e}_v \right) \right]_j \delta_{\boldsymbol{\alpha}\boldsymbol{\alpha}'} \delta_{\gamma j} \right) = \\
&J_{\mathcal{T}_i} J_{\boldsymbol{\mathcal{X}}_K} \hat{\omega}_{\boldsymbol{\alpha}} \sum_{\beta' \in \mathcal{N}} \left(\hat{\omega}_{\beta'} \left[\boldsymbol{s} \left(\left[\hat{\boldsymbol{q}}^{K,i} \right]_{\boldsymbol{\alpha}\beta' v} \boldsymbol{e}_v \right) \right]_\gamma \right).
\end{aligned} \tag{2.104}
$$

2.13.5 Term U-V

Let $d \in \mathcal{D}$ and $e \in \{0,1\} := \mathcal{E}$. Then if we define the $D-1$-dimensional quadrilateral $\partial \hat{K}_{d,e}$ as

$$
\partial \hat{K}_{d,e} = \left\{ \boldsymbol{\xi} \in \hat{K} \mid \left[\boldsymbol{\xi} \right]_d = e \right\}, \tag{2.105}
$$

the set $\{\partial \hat{K}_{d,e}\}_{d \in \mathcal{D}, e \in \mathcal{E}}$ is a partition of the surface $\partial \hat{K}$ of the spatial reference element. Utilizing the mappings $\boldsymbol{\mathcal{X}}_K$ that map points $\boldsymbol{\xi} \in \hat{K}$ to $\boldsymbol{x} \in K$ for all $K \in \mathcal{K}_h$ we can define

$$
\partial K_{d,e} = \boldsymbol{\mathcal{X}}_K \left(\partial \hat{K}_{d,e} \right), \tag{2.106}
$$

where now the set $\{\partial K_{d,e}\}_{d \in \mathcal{D}, e \in \mathcal{E}}$ is a quadrilateral partition of the surface ∂K for a cell $K \in \mathcal{K}_h$. Let us furthermore define $\boldsymbol{\mathcal{N}}^- := \{0, 1, \ldots, N-1\}^{D-1}$. Then the surface integral in the fifth term of eq. (2.99) can be rewritten as follows:

$$
\begin{aligned}
&\int_{t_i}^{t_i + \Delta t_i} \int_{\partial K} \left[\boldsymbol{\mathcal{G}} \left(\hat{\boldsymbol{q}}^{K,i}, \hat{\boldsymbol{q}}^{K',i}, \boldsymbol{n} \right) \right]_j \left[\boldsymbol{\phi}^K_{\boldsymbol{\alpha},\gamma} \right]_j ds(\boldsymbol{x}) dt = \\
&\int_{t_i}^{t_i + \Delta t_i} \sum_{d \in \mathcal{D}} \sum_{e \in \mathcal{E}} \left(\int_{\partial K_{d,e}} \left[\boldsymbol{\mathcal{G}} \left(\hat{\boldsymbol{q}}^{K,i}, \hat{\boldsymbol{q}}^{K',i}, (-1)^e \boldsymbol{e}_d \right) \right]_j \phi^K_{\boldsymbol{\alpha}} \left[\boldsymbol{e}_\gamma \right]_j ds(\boldsymbol{x}) \right) dt = \\
&J_{\mathcal{T}_i} J_{\boldsymbol{\mathcal{X}}_K} \int_0^1 \sum_{d \in \mathcal{D}} \sum_{e \in \mathcal{E}} \left(\frac{1}{\left[\Delta \boldsymbol{x}^K \right]_d} \int_{\partial \hat{K}_{d,e}} \left[\boldsymbol{\mathcal{G}} \left(\hat{\boldsymbol{q}}^{K,i}, \hat{\boldsymbol{q}}^{K',i}, (-1)^e \boldsymbol{e}_d \right) \right]_j \hat{\phi}_{\boldsymbol{\alpha}} \left[\boldsymbol{e}_\gamma \right]_j ds(\boldsymbol{\xi}) \right) d\tau = \\
&J_{\mathcal{T}_i} J_{\boldsymbol{\mathcal{X}}_K} \sum_{\beta' \in \mathcal{D}} \hat{\omega}_{\beta'} \sum_{d \in \mathcal{D}} \sum_{e \in \mathcal{E}} \sum_{\boldsymbol{\alpha}' \in \boldsymbol{\mathcal{N}}^-} \left(\hat{\omega}_{\boldsymbol{\alpha}'} \frac{1}{\left[\Delta \boldsymbol{x}^K \right]_d} \left[\boldsymbol{\mathcal{G}} \left(\hat{\boldsymbol{q}}^{K,i}, \hat{\boldsymbol{q}}^{K',i}, (-1)^e \boldsymbol{e}_d \right) \right]_j \ldots \right. \\
&\qquad \left. \ldots \hat{\phi}_{\boldsymbol{\alpha}^d}(\hat{\boldsymbol{\xi}}_{\boldsymbol{\alpha}'}) \left(\hat{\psi}_{[\boldsymbol{\alpha}]_d} \Big|_e \right) \left[\boldsymbol{e}_d \gamma \right]_j \right) = \\
&J_{\mathcal{T}_i} J_{\boldsymbol{\mathcal{X}}_K} \sum_{\beta' \in \mathcal{D}} \hat{\omega}_{\beta'} \sum_{d \in \mathcal{D}} \sum_{e \in \mathcal{E}} \sum_{\boldsymbol{\alpha}' \in \boldsymbol{\mathcal{N}}^-} \left(\hat{\omega}_{\boldsymbol{\alpha}'} \frac{1}{\left[\Delta \boldsymbol{x}^K \right]_d} \left[\boldsymbol{\mathcal{G}} \left(\hat{\boldsymbol{q}}^{K,i}, \hat{\boldsymbol{q}}^{K',i}, (-1)^e \boldsymbol{e}_d \right) \right]_j \ldots \right. \\
&\qquad \left. \ldots \delta_{\boldsymbol{\alpha}^d \boldsymbol{\alpha}'} \left(\hat{\psi}_{[\boldsymbol{\alpha}]_d} \Big|_e \right) \delta_{\gamma j} \right) = \\
&J_{\mathcal{T}_i} J_{\boldsymbol{\mathcal{X}}_K} \hat{\omega}_{\boldsymbol{\alpha}} \sum_{\beta' \in \mathcal{D}} \sum_{d \in \mathcal{D}} \sum_{e \in \mathcal{E}} \sum_{\alpha'_d \in \mathcal{N}} \left(\frac{\hat{\omega}_{\beta'}}{\hat{\omega}_{\alpha'_d}} \frac{1}{\left[\Delta \boldsymbol{x}^K \right]_d} \left[\boldsymbol{\mathcal{G}} \left(\hat{\boldsymbol{q}}^{K,i}, \hat{\boldsymbol{q}}^{K',i}, (-1)^e \boldsymbol{e}_d \right) \right]_\gamma \left(\hat{\psi}_{\alpha'_d} \Big|_e \right) \right) =
\end{aligned}
$$

$$J_{\mathcal{T}_i} J_{\boldsymbol{\mathcal{X}}_K} \hat{\omega}_{\boldsymbol{\alpha}} \sum_{\beta' \in \mathcal{D}} \sum_{d \in \mathcal{D}} \sum_{e \in \mathcal{E}} \sum_{\alpha'_d \in \mathcal{N}} \left(\frac{\hat{\omega}_{\beta'}}{\hat{\omega}_{\alpha'_d}} \frac{1}{\left[\Delta \boldsymbol{x}^K\right]_d} \left[\boldsymbol{\mathcal{G}} \left(\hat{\boldsymbol{q}}^{K,i}, \hat{\boldsymbol{q}}^{K',i}, (-1)^e \boldsymbol{e}_d \right) \right]_\gamma \dots \right.$$
$$\left. \dots \left(\delta_{e0} \boldsymbol{l}_{\alpha'_d} \right) \left(\delta_{e1} \boldsymbol{r}_{\alpha'_d} \right) \right). \tag{2.107}$$

In each term we have to solve a Riemann problem in direction of the unit vector $\boldsymbol{e}_d$ defined as

$$\left[\boldsymbol{e}_d\right]_{d'} = \delta_{dd'} \tag{2.108}$$

for $d' \in \mathcal{D}$. $\boldsymbol{l}$ and $\boldsymbol{r}$ denote the Left and Right Reference Element Flux Operator, respectively, and the latter is defined as

$$\left[\boldsymbol{r}\right]_i = \hat{\psi}_i \Big|_1 \tag{2.109}$$

for $i \in \mathcal{N}$. A Python script to compute $\boldsymbol{r}$ can be found in Appendix A.

2.13.6 The Complete One-step Update Scheme

Inserting eqs. (2.100) to (2.102), (2.104) and (2.107) into eq. (2.99) and dividing the resulting equation by $J_{\boldsymbol{\mathcal{X}}_K} \hat{\omega}_{\boldsymbol{\alpha}}$ yields

$$\left[\hat{\boldsymbol{u}}^{K,i+1}\right]_{\boldsymbol{\alpha}\gamma} = \left[\hat{\boldsymbol{u}}^{K,i}\right]_{\boldsymbol{\alpha}\gamma} +$$
$$J_{\mathcal{T}_i} \sum_{k \in \mathcal{D}} \left(\sum_{\alpha'_k \in \mathcal{N}} \sum_{\beta' \in \mathcal{N}} \left(\frac{1}{\hat{\omega}_{\alpha'_k}} \frac{1}{\left[\Delta \boldsymbol{x}^K\right]_k} \left[\boldsymbol{K}\right]_{\alpha'_k k} \left[\boldsymbol{F} \left(\left[\hat{\boldsymbol{q}}^{K,i}\right]_{\left[\boldsymbol{\alpha}_{0:k-1}, \alpha'_k, \boldsymbol{\alpha}_{k+1:D-1}\right], \beta', v} \boldsymbol{e}_v \right) \right]_{\gamma k} \right) \right) +$$
$$J_{\mathcal{T}_i} \sum_{\beta' \in \mathcal{N}} \left(\hat{\omega}_{\beta'} \left[\boldsymbol{s} \left(\left[\hat{\boldsymbol{q}}^{K,i}\right]_{\boldsymbol{\alpha}\beta' v} \boldsymbol{e}_v \right) \right]_\gamma \right) -$$
$$J_{\mathcal{T}_i} \sum_{\beta' \in \mathcal{D}} \sum_{d \in \mathcal{D}} \sum_{e \in \mathcal{E}} \sum_{\alpha'_d \in \mathcal{N}} \left(\frac{\hat{\omega}_{\beta'}}{\hat{\omega}_{\alpha'_d}} \frac{1}{\left[\Delta \boldsymbol{x}^K\right]_d} \left[\boldsymbol{\mathcal{G}} \left(\hat{\boldsymbol{q}}^{K,i}, \hat{\boldsymbol{q}}^{K',i}, (-1)^e \boldsymbol{e}_d \right) \right]_\gamma \left(\delta_{e0} \boldsymbol{l}_{\alpha'_d} \right) \left(\delta_{e1} \boldsymbol{r}_{\alpha'_d} \right) \right), \tag{2.110}$$

which we require to hold for $\boldsymbol{\alpha} \in \boldsymbol{\mathcal{N}}$, $\gamma \in \mathcal{V}$, $K \in \mathcal{K}_h$ and $i \in \mathcal{I}$.

2.13.7 Time Step Restriction

For the scheme to be stable we require that the following "global" time-stepping inequality holds for all cells $K \in \mathcal{K}_h$ and all Gauss-Legendre points $\hat{\boldsymbol{x}}_{\boldsymbol{\alpha}} := \boldsymbol{\mathcal{X}}_K(\hat{\boldsymbol{\xi}}_{\boldsymbol{\alpha}})$, $\boldsymbol{\alpha} \in \boldsymbol{\mathcal{N}}$:

$$\Delta t_i \leq \frac{1}{D} \frac{1}{(2N+1)} \min_{d \in \mathcal{D}} \left(\frac{\left[\Delta \boldsymbol{x}^K\right]_d}{\Lambda^{i,d}_{\boldsymbol{\alpha}}} \right), \tag{2.111}$$

where

$$\Lambda^{i,d}_{\boldsymbol{\alpha}} = \max_{v \in \mathcal{V}} \operatorname{abs} \left[\boldsymbol{\lambda}^{i,d}_{\boldsymbol{\alpha}}\right]_v \tag{2.112}$$

and $\lambda_{\alpha}^{d,i}$ is a vector containing the V real eigenvalues of the Jacobian

$$\frac{\partial}{\partial x_k}\left[\boldsymbol{F}\left(\boldsymbol{u}(\hat{\boldsymbol{x}}_{\alpha}, t_i)\right)\right]_{jd} \tag{2.113}$$

for the respective dimension $d \in \mathcal{D}$ and the index of the current time step $i \in \mathcal{I}$. For details on the derivation of this formula see [11, 18, 23]. For more information on so-called "local" time-stepping and other advanced time-stepping techniques see [14, 24].

2.14 A Posteriori Subcell Limiting

The unlimited ADER-DG scheme derived in the previous section allows us to solve non-linear hyperbolic balance laws with arbitrary order in both time and space for continuous data. For discontinuous initial data and in scenarios where even for smooth initial data due to the non-linear nature of the system shocks arise, however, our high-order discontinuous Galerkin method is unsuitable. It is a linear scheme in the sense of Godunov, i.e. the coefficients of the scheme are independent of the current state of the system (see [10, 25] for a derivation of the so-called Godunov theorem) and therefore shocks in the system do not only give rise to artificial and persistent oscillations, but also decrease pointwise accuracy in the vicinity to first order and even cause loss of pointwise convergence at the point of discontinuity (see [14] for details). The quality of the numerical solution is unacceptable at best, if not the simulation crashes altogether due to a strictly positive physical quantity such as pressure or density becoming negative and thereby invalidating the well-posedness of the problem.

A way to solving this issue is limiting. For an exhaustive overview on classical limiting approaches see again [10]. According to Dumbser et al. [11], there are two key challenges in designing limiter procedures:

1. A so-called troubled cell indicator needs to implement criteria on how to identify troubled cells, i.e. cells that need limiting.
2. The troubled DG solution needs to be replaced by a robust non-linear reconstructions scheme in a way such that the minimum amount of artificial numerical viscosity is injected in these cells ideally preserving the subcell resolution property of the high-order DG scheme.

In the following we will present a novel approach called a posterior subcell limiting. It was first introduced in 2014 (see [11]) and has very favorable properties with respect to the aforementioned challenges. The general idea is to first project the local ADER-DG solution onto a finer equidistant grid to check if the solution satisfies certain admissibility criteria. If this is not the case, we discard this candidate solution and instead rely on a more robust lower-order FVM scheme starting from the fine grid solution of the previous time step (which is either the projected ADER-DG solution or, if the cell had already been troubled in the previous time step, the available FVM fine grid solution). For troubled cells we finally use a reconstruction operator to replace the rejected candidate solution by the transformed fine grid solution. The remainder of this chapter is structured as follows: We will first introduce the necessary projection and reconstruction operators. Subsequently we will discuss a possible set of admissibility criteria and conclude

with a summary on the MUSCL-Hancock finite volume method as an example for a robust scheme that can be used in conjunction with a posteriori subcell limiting.

2.14.1 Projection and Reconstruction

In order to check for admissibility and in case of a troubled cell to employ a more robust FVM scheme, we need to project the ADER-DG degrees of freedom $\hat{\boldsymbol{u}}^{K,i}$ to subcell averages $\hat{\boldsymbol{p}}^{K,i}$ on an equidistant fine grid. We choose to split $K \in \mathcal{K}_h$ into $N_S = 2N+1$ subcells along each spatial dimension creating a total of N_S^D subcells denoted as $K_{\boldsymbol{\alpha}}$, $\boldsymbol{\alpha} \in \{0,1,\ldots,2N\}^D := \boldsymbol{\mathcal{N}_S}$. For explicit Godunov-type finite volume schemes on the subgrid we must satisfy the stability condition

$$\Delta t \leq \frac{1}{d}\frac{1}{N_S}\min_{d\in\mathcal{D}}\left(\frac{[\Delta \boldsymbol{x}]_d}{\Lambda^d}\right). \tag{2.114}$$

Comparing eq. (2.114) to the time step restriction for the ADER-DG scheme given in eq. (2.111) illustrates that the choice $N_S = 2N+1$ is optimal in the sense that it makes sure that, on the one hand, time steps on the ADER-DG grid are also stable on the equidistant subgrid and, on the other hand, that we add the minimum amount of numerical dissipation necessary. See [11] for additional remarks on the optimality of this particular choice of N_S.

Let as before $K_{\boldsymbol{\alpha}}$ be a cell in the equidistant subgrid of cardinality $N_S^D = (2N+1)^D$ on a cell $K \in \mathcal{K}_h$. Then we can define a projection $\boldsymbol{\mathcal{P}}$ of the degrees of freedom from the ADER-DG grid cell K to the subcell $K_{\boldsymbol{\alpha}}$ for all $\boldsymbol{\alpha} \in \boldsymbol{\mathcal{N}_S}$ by prescribing that the integral averages over $K_{\boldsymbol{\alpha}}$ are to be preserved. If we introduce

$$\tilde{\boldsymbol{\xi}}_{\boldsymbol{\alpha},\boldsymbol{\alpha}'} := \left(\frac{1}{N_S}\boldsymbol{\alpha} + \frac{1}{N_S}\hat{\boldsymbol{\xi}}_{\boldsymbol{\alpha}'}\right) \tag{2.115}$$

as a shorthand notation for the $\boldsymbol{\alpha}'$-th Gauss-Legendre point inside the $\boldsymbol{\alpha}$-th subgrid cell of K, mathematically this requirement can be expressed as

$$\begin{aligned}
&\left[\hat{\boldsymbol{p}}^{K,i}\right]_{\boldsymbol{\alpha},\gamma} \overset{!}{=} \frac{1}{|K_{\boldsymbol{\alpha}}|}\int_{K_{\boldsymbol{\alpha}}}\left[\tilde{\boldsymbol{u}}_h^{K,i}\right]_\gamma d\boldsymbol{x} = \frac{1}{|K_{\boldsymbol{\alpha}}|}\int_{K_{\boldsymbol{\alpha}}}\left[\hat{\boldsymbol{u}}^{K,i}\right]_{n,v}\phi_{\boldsymbol{n}}^K(\boldsymbol{x})\,[\boldsymbol{e}_v]_\gamma\, d\boldsymbol{x} = \\
&\sum_{\boldsymbol{n}\in\boldsymbol{\mathcal{N}}}\sum_{\boldsymbol{\alpha}'\in\boldsymbol{\mathcal{N}}}\left(\left[\hat{\boldsymbol{u}}^{K,i}\right]_{\boldsymbol{n}\gamma}\hat{\omega}_{\boldsymbol{\alpha}'}\hat{\phi}_{\boldsymbol{n}}\left(\tilde{\boldsymbol{\xi}}_{\boldsymbol{\alpha},\boldsymbol{\alpha}'}\right)\right) = \\
&\sum_{\boldsymbol{n}\in\boldsymbol{\mathcal{N}}}\sum_{\boldsymbol{\alpha}'\in\boldsymbol{\mathcal{N}}}\left(\prod_{d\in\mathcal{D}}\left(\hat{\omega}_{\boldsymbol{\alpha}'_d}\hat{\psi}_{\boldsymbol{n}_d}\left(\tilde{\boldsymbol{\xi}}_{\boldsymbol{\alpha}_d,\boldsymbol{\alpha}'_d}\right)\right)\left[\hat{\boldsymbol{u}}^{K,i}\right]_{\boldsymbol{n}\gamma}\right) = \\
&\sum_{\boldsymbol{n}\in\boldsymbol{\mathcal{N}}}\left(\sum_{\alpha'_0\in\mathcal{N}}\left(\hat{\omega}_{\alpha'_0}\hat{\psi}_{\boldsymbol{n}_0}\left(\tilde{\xi}_{\boldsymbol{\alpha}_0,\boldsymbol{\alpha}'_0}\right)\ldots\sum_{\alpha'_{D-1}\in\mathcal{N}}\left(\hat{\omega}_{\alpha'_{D-1}}\hat{\psi}_{\boldsymbol{n}_{D-1}}\left(\tilde{\xi}_{\boldsymbol{\alpha}_{D-1},\boldsymbol{\alpha}'_{D-1}}\right)\right)\right)\left[\hat{\boldsymbol{u}}^{K,i}\right]_{\boldsymbol{n}\gamma}\right) = \\
&\sum_{\boldsymbol{n}\in\boldsymbol{\mathcal{N}}}\left(\prod_{d\in\mathcal{D}}\left([\boldsymbol{P}]_{\boldsymbol{\alpha}_d\boldsymbol{n}_d}\right)\left[\hat{\boldsymbol{u}}^{K,i}\right]_{\boldsymbol{n}\gamma}\right),
\end{aligned} \tag{2.116}$$

where $|K_{\boldsymbol{\alpha}}|$ denotes the volume of $K_{\boldsymbol{\alpha}}$ for $\boldsymbol{\alpha} \in \boldsymbol{\mathcal{N}_S}$ and $\gamma \in \mathcal{D}$. The computation can be carried out efficiently in a dimension-by-dimension manner using the projection matrix

$$[\boldsymbol{P}]_{ij} = \sum_{k \in \mathcal{N}} \left(\hat{\omega}_k \hat{\psi}_j \left(\frac{1}{N_S} i + \frac{1}{N_S} \hat{\xi}_k \right) \right) \tag{2.117}$$

for $i \in \mathcal{N}_S$ and $j \in \mathcal{N}$. A Python script to compute the projection matrix $\boldsymbol{P}$ can be found in Appendix A.

To enable replacement of the invalid candidate solution $\hat{\boldsymbol{u}}^{K,i}$ we need to define a reconstruction operator $\boldsymbol{\mathcal{R}}$ which transforms the degrees of freedom $\hat{\boldsymbol{p}}^{K,i}$ of the solution computed by the FVM scheme back to the ADER-DG grid. Since we have more degrees of freedom on the fine grid than on the ADER-DG grid, we can now only require that the constraints based on preservation of local averages, i.e.

$$\frac{1}{|K_{\boldsymbol{\alpha}}|} \int_{K_{\boldsymbol{\alpha}}} \left[\tilde{\boldsymbol{u}}_h^{K,i} \right]_\gamma \, d\boldsymbol{x} \overset{!}{=} \frac{1}{|K_{\boldsymbol{\alpha}}|} \int_{K_{\boldsymbol{\alpha}}} \left[\tilde{\boldsymbol{p}}^{K,i} \right]_\gamma \, d\boldsymbol{x} = \left[\hat{\boldsymbol{p}}^{K,i} \right]_{\boldsymbol{\alpha}\gamma} \tag{2.118}$$

for $\boldsymbol{\alpha} \in \boldsymbol{\mathcal{N}_S}$ and $\gamma \in \mathcal{V}$, are fulfilled in a least squares sense. We do however insist that the overall integral average

$$\int_K \left[\tilde{\boldsymbol{p}}^{K,i} \right]_\gamma \, d\boldsymbol{x} \overset{!}{=} \int_K \left[\tilde{\boldsymbol{u}}_h^{K,i} \right]_\gamma \, d\boldsymbol{x}. \tag{2.119}$$

is preserved to make sure that the scheme remains conservative.

Analogous to eq. 2.116 the left-hand side of eq. (2.118) can be simplified as follows:

$$\begin{aligned} &\frac{1}{K_{\boldsymbol{\alpha}}} \int_{K_{\boldsymbol{\alpha}}} \left[\hat{\boldsymbol{u}}^{K,i} \right]_{\boldsymbol{n}\gamma} \phi_{\boldsymbol{n}}^K \, d\boldsymbol{x} = \\ &\int_{\hat{K}} \left[\hat{\boldsymbol{u}}^{K,i} \right]_{\boldsymbol{n}\gamma} \hat{\phi}_{\boldsymbol{n}} \left(\frac{1}{N_S} \boldsymbol{\alpha} + \frac{1}{N_S} \boldsymbol{\xi} \right) \, d\boldsymbol{\xi} = \\ &\sum_{\boldsymbol{n} \in \boldsymbol{\mathcal{N}}} \left(\prod_{d \in \mathcal{D}} \left([\boldsymbol{P}]_{\boldsymbol{\alpha}_d \boldsymbol{n}_d} \right) \left[\hat{\boldsymbol{u}}^{K,i} \right]_{\boldsymbol{n}\gamma} \right). \end{aligned} \tag{2.120}$$

Eq. (2.119) simplifies to

$$\sum_{\boldsymbol{\alpha} \in \boldsymbol{\mathcal{N}_S}} \frac{1}{|K_{\boldsymbol{\alpha}}|} \left[\hat{\boldsymbol{p}}^{K,i} \right]_{\boldsymbol{\alpha}v} = \frac{1}{N_S^D} \sum_{\boldsymbol{\alpha} \in \boldsymbol{\mathcal{N}_S}} \left[\hat{\boldsymbol{p}}^{K,i} \right]_{\boldsymbol{\alpha}v} \overset{!}{=} \sum_{\boldsymbol{n} \in \boldsymbol{\mathcal{N}}} \hat{\omega}_{\boldsymbol{n}} \left[\hat{\boldsymbol{u}}^{K,i} \right]_{\boldsymbol{n}v} \tag{2.121}$$

for $v \in V$. In order to be able to carry out the computation in a dimension-by-dimension manner as for the projection we need to solve the following constrained least squares optimization problem to find the reconstruction matrix $\boldsymbol{R} \in \mathbb{R}^{(N+1) \times N_S}$ for $\boldsymbol{P} \in \mathbb{R}^{N_S \times (N+1)}$ being the projection matrix and for vectors $\boldsymbol{p} \in \mathbb{R}^{N_S}$ and $\boldsymbol{u} \in \mathbb{R}^{N+1}$ representing the degrees of freedom along a coordinate axis for a single quantity:

$$\begin{aligned} \underset{\boldsymbol{u} \in \mathbb{R}^{N+1}}{\text{minimize}} \quad & \left\| [\boldsymbol{P}]_{ij} [\boldsymbol{u}]_j - [\boldsymbol{p}]_j \right\|^2 \\ \text{subject to} \quad & \hat{\omega}_i [\boldsymbol{u}]_i = \frac{1}{N_S} \sum_{i \in \mathcal{N}_S} [\boldsymbol{p}]_i . \end{aligned} \tag{2.122}$$

The corresponding Lagrange dual problem (see e.g. [26] for a general derivation) written in matrix vector notation reads

$$\begin{bmatrix} 2\boldsymbol{P}^T\boldsymbol{P} & \hat{\boldsymbol{\omega}}^T \\ \hat{\boldsymbol{\omega}} & 0 \end{bmatrix} \begin{bmatrix} \boldsymbol{u} \\ z \end{bmatrix} = \begin{bmatrix} 2\boldsymbol{P}^T \\ \frac{1}{N_S}\mathbf{1}^T \end{bmatrix} \Big[\boldsymbol{p}\Big], \tag{2.123}$$

where $z \in \mathbb{R}$ is a scalar Lagrange multiplier and $\mathbf{1}$ is a vector with all components equal to one. We can then compute

$$\Big[\tilde{\boldsymbol{R}}\Big] = \begin{bmatrix} 2\boldsymbol{P}^T\boldsymbol{P} & \hat{\boldsymbol{\omega}}^T \\ \hat{\boldsymbol{\omega}} & 0 \end{bmatrix}^{-1} \begin{bmatrix} 2\boldsymbol{P}^T \\ \frac{1}{N_S}\mathbf{1}^T \end{bmatrix}. \tag{2.124}$$

Since we are only interested in the first $N+1$ rows of $\tilde{\boldsymbol{R}}$, the reconstruction matrix $\boldsymbol{R}$ is obtained after discarding the last row, i.e.

$$\Big[\boldsymbol{R}\Big]_{0:N,0:N_S-1} = \Big[\tilde{\boldsymbol{R}}\Big]_{0:N,0:N_S-1}. \tag{2.125}$$

A Python notebook to compute $\boldsymbol{R}$ can be found in Appendix A. For more details on the transformations see [27].

2.14.2 Identification of Troubled Cells

Once a candidate solution in this section denoted as $\tilde{\boldsymbol{u}}_h^{K,i+1^*}$ represented by coefficients $\hat{\boldsymbol{u}}^{K,i+1^*}$ in a cell $K \in \mathcal{K}_h$ provided by the unlimited ADER-DG scheme is available, we can apply the set of rules defined by the troubled cell indicator to check if limiting is required. Following the terminology introduced in [11] we discriminate between two kinds of detection criteria:

1. Physical admissibility detection (PAD) can be used to incorporate domain knowledge into the simulation. Most commonly criteria of this kind are used to enforce that quantities with physical meaning stay within their respective domain of definition so that the well-posedness of the problem remains ensured. A PAD criterion is represented by a set of J inequalities of the following form:

$$\pi_j(\tilde{\boldsymbol{u}}_h^{K,i+1^*}) > 0. \tag{2.126}$$

   A candidate solution in a cell $K \in \mathcal{K}_h$ is considered admissible in a physical sense if all J inequalities are fulfilled. Considering the Euler equations as an illustrative example, if we want to ensure positivity of the pressure p and the density ρ, we will have two inequalities with $\pi_1(\tilde{\boldsymbol{u}}_h^{K,i+1^*}) = p$ and $\pi_2(\tilde{\boldsymbol{u}}_h^{K,i+1^*}) = \rho$.

2. Numerical admissibility detection (NAD) in the form of a relaxed version of the discrete minimum principle (DMP) ensures that the total variation of the solution remains bounded. In theory we would like to require that

$$\min_{\boldsymbol{y}\in V(K)} \Big[\tilde{\boldsymbol{u}}_h^{K',i}(\boldsymbol{y}, t_i)\Big]_v - \boldsymbol{\delta}_v \leq \Big[\tilde{\boldsymbol{u}}_h^{K,i+1^*}(\boldsymbol{x}, t_{i+1})\Big]_v \leq \max_{\boldsymbol{y}\in V(K)} \Big[\tilde{\boldsymbol{u}}_h^{K',i}(\boldsymbol{y}, t_i)\Big]_v + \boldsymbol{\delta}_v \tag{2.127}$$

holds for all $v \in \mathcal{V}$, $\boldsymbol{x} \in K$, $K \in \mathcal{K}_h$, i.e. the values of all quantities inside K in the candidate solution should be bounded by the solution on the Voronoi neighbors in the previous time step. The vector $\boldsymbol{\delta}$ is defined as

$$\boldsymbol{\delta}_v = \varepsilon \left(\max_{\boldsymbol{y} \in V(K)} \left[\tilde{\boldsymbol{u}}_h^{K',i}(\boldsymbol{y}, t_i) \right]_v - \min_{\boldsymbol{y} \in V(K)} \left[\tilde{\boldsymbol{u}}_h^{K',i}(\boldsymbol{y}, t_i) \right]_v \right), \tag{2.128}$$

where we choose for example $\varepsilon = 10^3$ to avoid problems with floating-point errors. Since it would be prohibitively expensive to compute the extrema for all polynomials, we settle for

$$\min_{\substack{K' \in V(K), \\ \boldsymbol{\alpha}' \in \boldsymbol{\mathcal{N}}_{\boldsymbol{\mathcal{S}}}}} \left[\hat{\boldsymbol{p}}^{K',i} \right]_{\boldsymbol{\alpha}',v} - \boldsymbol{\delta}_v \leq \left[\hat{\boldsymbol{p}}^{K,i+1^*} \right]_{\boldsymbol{\alpha},v} \leq \max_{\substack{K' \in V(K), \\ \boldsymbol{\alpha}' \in \boldsymbol{\mathcal{N}}_{\boldsymbol{\mathcal{S}}}}} \left[\hat{\boldsymbol{p}}^{K',i} \right]_{\boldsymbol{\alpha}',v} - \boldsymbol{\delta}_v \tag{2.129}$$

instead, i.e. we enforce the relaxed DMP criterion only for the subcell averages on the refined equidistant grid. We apply the projection operator to obtain $\hat{\boldsymbol{p}}^{K,i+1^*} = \boldsymbol{\mathcal{P}}(\hat{\boldsymbol{u}}^{K,i+1^*})$.

2.14.3 The MUSCL-Hancock Scheme

For the sake of completeness we will now give a quick summary on the nonlinear (in the sense of Godunov), second order accurate MUSCL[3]-Hancock FVM scheme. Of course this is only one possible choice for a robust, lower-order scheme that can be used as a "fallback" in the a posterior subcell limiting approach. Following the recipe given in [10] and modified to incorporate source terms and an arbitrary number of space dimensions, the scheme consists of the following steps:

1. **Compute slopes:**

$$\left[\hat{\boldsymbol{\delta}}^{K,i} \right]_{d\boldsymbol{\alpha}\gamma} = \operatorname{minmod} \left(\left[\hat{\boldsymbol{p}}^{K,i} \right]_{\boldsymbol{\alpha}+\boldsymbol{e}_d,\gamma} - \left[\hat{\boldsymbol{p}}^{K,i} \right]_{\boldsymbol{\alpha},\gamma}, \left[\hat{\boldsymbol{p}}^{K,i} \right]_{\boldsymbol{\alpha},\gamma} - \left[\hat{\boldsymbol{p}}^{K,i} \right]_{\boldsymbol{\alpha}-\boldsymbol{e}_d,\gamma} \right) \tag{2.130}$$

for all subgrid cells $K_{\boldsymbol{\alpha}} \subset K$, $\boldsymbol{\alpha} \in \{-1, 0, \ldots, N_S\}^D := \boldsymbol{\mathcal{N}}_{\boldsymbol{\mathcal{S}}}^*$, $\gamma \in \mathcal{V}$, troubled grid cells $K \in \mathcal{K}_h^*$, unit vectors $\boldsymbol{e}_d$ and dimension $d \in \mathcal{D}$. A component of the index $\boldsymbol{\alpha} \in \boldsymbol{\mathcal{N}}_{\boldsymbol{\mathcal{S}}}^*$ being -1 or N_S implies access into $\hat{\boldsymbol{p}}^{K',i}$, i.e. access into the fine grid solution on the respective Voroni neighbor $K' \in V(K)$ of K. We furthermore use the common definition of the minmod function, namely

$$\operatorname{minmod}(a,b) = \begin{cases} 0 & \text{if } ab \leq 0 \\ a & \text{if } ab > 0 \text{ and } |a| \leq |b| \\ b & \text{if } ab > 0 \text{ and } |b| < |a|. \end{cases} \tag{2.131}$$

2. **Evaluate source:**

$$\left[\hat{\boldsymbol{s}}^{K,i} \right]_{\boldsymbol{\alpha}\gamma} = \left[\boldsymbol{s} \left(\left[\hat{\boldsymbol{q}}^{K,i} \right]_{\alpha} \right) \right]_{\gamma} \tag{2.132}$$

for all subgrid cells $K_{\boldsymbol{\alpha}} \subset K$, $\boldsymbol{\alpha} \in \boldsymbol{\mathcal{N}}_{\boldsymbol{\mathcal{S}}}^*$, troubled cells $K \in \mathcal{K}_h^*$ and $\gamma \in \mathcal{V}$.

[3] Monotonic Upstream-Centered Scheme for Conservation Laws

3. **Extrapolate:**

$$\left[\boldsymbol{w}^{K,i}\right]_{de\boldsymbol{\alpha}\gamma} = \left[\hat{\boldsymbol{u}}^{K,i}\right]_{\boldsymbol{\alpha}\gamma} + \frac{e}{2}\left[\hat{\delta}_d^{K,i}\right]_{\boldsymbol{\alpha}\gamma} \tag{2.133}$$

for all subgrid cells $K_{\boldsymbol{\alpha}} \subset K$, $\boldsymbol{\alpha} \in \boldsymbol{\mathcal{N}}_{\mathcal{S}}^*$, troubled cells $K \subset \mathcal{K}_h^*$, $\gamma \in \mathcal{V}$, $d \in \mathcal{D}$ and $e \in \{-1, +1\} := \sigma$.

4. **Evolve:**

$$\begin{aligned}&\left[\boldsymbol{w}^{K,i+\frac{1}{2}}\right]_{de\boldsymbol{\alpha}\gamma} = \left[\boldsymbol{w}^{K,i}\right]_{d,e,\boldsymbol{\alpha},\gamma} + \\ &\frac{\Delta t_i}{2}\sum_{d'\in\mathcal{D}}\sum_{e'\in\sigma}\left(e'\left[\boldsymbol{F}\left(\left[\boldsymbol{w}^{K,i}\right]_{d'e'\boldsymbol{\alpha}}\right)\right]_{\gamma d'} / \left[\Delta \boldsymbol{x}^{K_{\boldsymbol{\alpha}}}\right]_{d'}\right) + \frac{\Delta t_i}{2}\left[\hat{\boldsymbol{s}}^{K,i}\right]_{\boldsymbol{\alpha}\gamma} := \\ &\left[\boldsymbol{w}^{K,i}\right]_{de\boldsymbol{\alpha}\gamma} + \frac{\Delta t_i}{2}\left[\hat{\boldsymbol{c}}\right]_{\boldsymbol{\alpha}\gamma}\end{aligned} \tag{2.134}$$

for all subgrid cells $K_{\boldsymbol{\alpha}} \subset K$, $\boldsymbol{\alpha} \in \boldsymbol{\mathcal{N}}_{\mathcal{S}}^*$, troubled cells $K \subset \mathcal{K}_h^*$, $\gamma \in \mathcal{V}$, $d \in \mathcal{D}$ and $e \in \sigma$.

5. **Solve Riemann problems:**

$$\left[\boldsymbol{f}^{K,i}\right]_{d\boldsymbol{\alpha}\gamma} = \left[\boldsymbol{\mathcal{G}}\left(\left[\boldsymbol{w}^{K,i+\frac{1}{2}}\right]_{d,+1,\boldsymbol{\alpha}-\boldsymbol{e}_d}, \left[\boldsymbol{w}^{K,i+\frac{1}{2}}\right]_{d,-1,\boldsymbol{\alpha}+\boldsymbol{e}_d}, \boldsymbol{e}_d\right)\right]_{\gamma} \tag{2.135}$$

for all subgrid cells $K_{\boldsymbol{\alpha}} \subset K$, $\boldsymbol{\alpha} \in \boldsymbol{\mathcal{N}}_{\mathcal{S}}^*$, troubled cells $K \subset \mathcal{K}_h^*$, $\gamma \in \mathcal{V}$ and $d \in \mathcal{D}$.

6. **Evolve source:**

$$\left[\hat{\boldsymbol{s}}^{K,i+\frac{1}{2}}\right]_{\boldsymbol{\alpha}\gamma} = \left[\boldsymbol{s}\left(\left[\hat{\boldsymbol{s}}^{K,i}\right]_{\boldsymbol{\alpha}} + \frac{1}{2}\left[\hat{\boldsymbol{c}}\right]_{\boldsymbol{\alpha}}\right)\right]_{\gamma} \tag{2.136}$$

for all subgrid cells $K_{\boldsymbol{\alpha}} \subset K$, $\boldsymbol{\alpha} \in \boldsymbol{\mathcal{N}}_{\mathcal{S}}^*$, troubled cells $K \subset \mathcal{K}_h^*$ and $\gamma \in \mathcal{V}$.

7. **Update solution:**

$$\begin{aligned}\left[\boldsymbol{p}^{L^K,i+1}\right]_{\boldsymbol{\alpha}\gamma} = \left[\boldsymbol{p}^{L^K,i}\right]_{\boldsymbol{\alpha}\gamma} - \\ \Delta t_i \sum_{d\in\mathcal{D}}\left(\left(\left[\boldsymbol{f}^{K,i}\right]_{d,\boldsymbol{\alpha}+\boldsymbol{e}_d,\gamma} - \left[\boldsymbol{f}^{K,i}\right]_{d,\boldsymbol{\alpha},\gamma}\right) / \left[\Delta \boldsymbol{x}^{K_{\boldsymbol{\alpha}}}\right]_d\right) + \\ \Delta t_i \left[\hat{\boldsymbol{s}}^{K,i+\frac{1}{2}}\right]_{\boldsymbol{\alpha}\gamma}\end{aligned} \tag{2.137}$$

for all subgrid cells $K_{\boldsymbol{\alpha}} \subset K$, $\boldsymbol{\alpha} \in \boldsymbol{\mathcal{N}}_{\mathcal{S}}^*$, troubled cells $K \subset \mathcal{K}_h^*$ and $\gamma \in \mathcal{V}$.

Chapter 3

Profiling and Energy-aware Computing in Modern x86 Systems

3.1 On the Importance of Performance Profiling in Software Engineering for High Performance Computing

In a High Performance Computing (HPC) context the standard balance of design goals in Software Engineering is shifted strongly in favor of maximum application performance. In view of the ever-growing complexity of computer architectures, performance profiling has become an inevitable tool that a) provides a baseline reference on the current state of a project, b) helps to prioritize, guide and track progress of optimization efforts and c) allows comparison to other state of the art solutions as well as theoretical optima. Modern x86 processors are equipped with one or more Performance Monitoring Units (PMU) which in conjunction with a suitable operating system provide means to obtain profiling information directly within an application. This so-called Hardware Performance Monitoring (HPM) interface enables programming of performance event counters that in turn allow computation of metrics that are correlated with good application performance and (more recently) energy efficiency (see [5, 28, 29]). Treibig et al. illustrate best practices on how HPM can be used for performance engineering on modern multi-core processors, offer a taxonomy of common causes for suboptimal application performance and offer guidance on how profiling can be used to identify and tackle them [30]. The rest of this section is organized as follows: We first give a brief summary on the x86 Instruction Set Architecture (ISA) and its importance in current HPC systems. We then focus in more detail on features and limitations of HPM in modern x86 processors and on means how to access them in the context of a normal user-space application. The section is concluded with remarks on on-chip energy monitoring provided by some of the most recent generations of x86 processors and a literature review on findings concerning the accuracy of this feature.

3.2 The x86 Instruction Set Architecture and the Current Prevalence of x86 in High Performance Computing

The term x86 refers to a set of Instruction Set Architectures (ISAs) based on the philosophy of the Intel 8086 16-bit microprocessor introduced in 1978 [31]. Over

Architecture	Count	Percentage
x86	468	93.6%
Power	23	4.6%
SPARC	7	1.4%
Sunway	2	0.4%

(a) CPU Architecture

Accelerator	Count	Percentage
None	406	81.2%
GPU	66	13.2%
Xeon Phi	23	4.6%
Other	5	1.0%

(b) Accelerator Cards

Table 3.1: Distribution of CPU architectures and accelerator cards of the supercomputers listed in the June 2016 Top500 list [3]. Systems that have both GPUs and Xeon Phi accelerator cards are listed as "Other."

the years many additions to the original ISA such as support for 32-bit (1985) and 64-bit (2003) memory addressing or the introduction of special purpose instructions e.g. for vectorized floating-point operations (MMX, SSE, AVX) or encryption (AES-NI) have been made, but nevertheless all of these changes in theory do not break backward compatibility [32]. This means that even though modern x86 processors are 64-bit capable, consist of multiple cores with complex multi-level cache hierarchies and sophisticated out of order execution pipelines, they can still correctly execute the instructions that make up a program originally written for an 8086 [31]. Due to the large number of instructions specified in a modern x86 ISA and due to the comparably complex and sometimes special purpose nature of some of them, x86 is considered to be an instance of the complex instruction set computing (CISC) paradigm (as opposed to RISC, reduced instruction set computing). Throughout this thesis we use the term "modern x86 processor" to denote a 64-bit capable multi-core processor that implements an x86 ISA that includes vector instructions for floating-point operations. Even though historically more than ten companies have produced x86 processors, nowadays this means that we refer to a processor manufactured by either Intel and AMD[1].

The famous TOP500 list ranks the world's fastest supercomputers in terms of floating-point performance measured using the LINPACK benchmark [8]. The list is published biannually in June and December by Strohmaier et al. [3]. The June 2016 edition illustrates the strong prevalence of x86 in HPC (see Table 3.1a). 468 of the 500 systems listed at the moment are based on x86 processors; 455 of them use CPUs manufactured by Intel.

A major trend directly reflected in the TOP500 list is the growing popularity of dedicated accelerator cards (see Table 3.1b). Such devices are usually connected to the main CPU via the PCI Express bus, require additional power supply and sometimes come with their own form of interconnection technology to allow for direct communication that avoids involvement of host memory or CPU. Applications that exhibit a great degree of mostly homogeneous and independent parallelism such as many types of simulations in scientific computing that in terms of runtime are dominated by vectorizable floating-point operations can greatly benefit from the presence of accelerator cards. Figure 3.1 illustrates the growing number of supercomputing systems that employ accelerators over time. Starting approximately

[1]The market share of Intel in this "duopoly" is estimated to be between 80% and 98% in the three market segments server, desktop and laptop [33].

in 2010 and 2012, respectively, two types of accelerators can be identified as the main drivers of this trend:

1. Graphics processing units used in the context of general purpose computing (GPGPU), most prominently NVIDIA devices implementing the company's Compute Unified Device Architecture (CUDA).

2. Accelerators based on the many integrated core (MIC) paradigm, that is devices from Intel's Xeon Phi series. These devices in essence consist of a great number (>30) of small, interconnected x86 processors all located on a single chip and support wider vector instructions than normal contemporary CPUs do.

As of June 2016, 94 of the 500 machines on the list employ accelerators. Even though more recently the rate of growth in the use of accelerators seems to be slowing down, it is predicted that the demand for accelerator devices specialized in parallel floating-point computations will nevertheless remain strong [34]. This is furthermore underscored by the fact that even though only three out of the ten fastest systems today employ accelerators, nine out of the ten most energy-efficient systems on the TOP500 list (consequently leading the so-called Green500 list [35]) do.

The review of the Top500 list above clearly illustrates that at the moment x86 dominates for HPC systems on a broad scale. Considering however trends such as the use of GPUs as special purpose accelerators and the fact that the leading system "Sunway TaihuLight" is based on a custom architecture developed within China[2] might give rise to the presumption that the market of HPC systems will become more diverse within the next couple of years. The four largest supercomputers expected to go in service in the United States by 2018 are either based on NVIDIA GPGPUs (Summit [36] and Sierra [37]) or the third generation of Intel's Xeon Phi product line then to be used directly as a CPU (Theta and Aurora [38]).

3.3 Hardware Performance Monitoring in Modern x86 Processors

In 1993 Intel introduced the Pentium 60 and Pentium 66 as the earliest members of the first generation Pentium microprocessor family [39]. From documentation available to the general public it was well known that Pentium processors collected a lot of statistics on the interaction between running code and hardware. The means how to access the data, however, were left undocumented or required signing a nondisclosure agreement with the manufacturer. Since obtaining the metrics hinted towards in the documentation would be of great value in optimizing application performance, enthusiasts such as Terje Mathisen started to employ reverse engineering techniques and eventually managed to recover the complete set of available performance counter events [40].

Today almost all processors independent of their architecture provide some form of hardware performance monitoring (HPM) for low overhead collection of metrics describing the interaction between code and hardware [41]. The performance

[2] partly in response to export restrictions

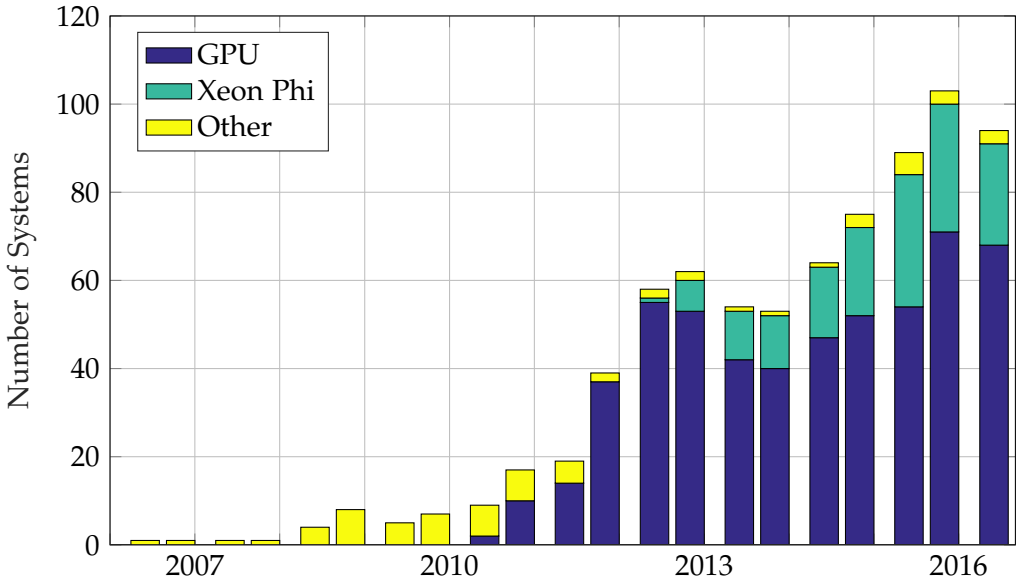


Figure 3.1: Number of supercomputing systems on the TOP500 list [3] that employ accelerator cards between June 2006 (the time when the first such system appeared on the list) and June 2016. Systems that have both GPUs and Xeon Phi accelerators are listed as "Other." The list is updated twice a year in June and November. The bars are centered on the day of release.

monitoring units (PMUs) on modern multi-core processors consist of a small number[3] of special-purpose registers (often called module specific registers (MSRs)). There is usually one PMU dedicated to each core (more precisely hardware thread if simultaneous multithreading (SMT) is available) and additional PMUs for resources shared between cores such as the memory controller or the interconnection bus. PMU registers can be programmed freely to count the number of occurrences of hardware events such as the retirement of an instruction, a cache miss or a successful branch prediction. If used correctly they provide insight into not only where bottlenecks are in the program, but also help explain the reason why they exist [42]. The special instructions necessary to do so can only be executed with kernel privileges (ring 0) and are not part of the ISA so that for example in the x86 case there is no guaranteed backward compatibility. In Linux there are several kernel modules that allow direct or indirect access to performance counters, most prominently there are OProfile [43], perf [44] and msr. The de-facto standard library for HPM on Linux is PAPI [45], which in turn is built on top of the perf interface. Other libraries such as LIKWID [46] instead make use of msr, a module that maps the MSRs to a device file interface and due to its simplicity promises lower overheads. Documentation on the use of MSRs is available in [32, chapter 19] and [47, chapter 3.2.5] for x86 processors by Intel and AMD, respectively. In principle it would of course be possible for example in a scientific simulation code to directly use the perf or msr kernel interface to program and read performance counters, but due to in general poor documentation and the volatile nature of the syntactics and semantics of counter events across the vari-

[3] most commonly four

ous microprocessor architectures, generations and manufacturers this approach is usually unfeasible in practice.

In general there are two approaches towards employing HPM, namely sampling and probing [41]. The former involves periodically interrupting program execution using system interrupts to read performance counter values and to sample the call stack which allows to correlate the obtained data with a section in the program. Overhead and inaccuracy introduced by this procedure is determined by system load and the rate at which interrupts take place. Sampling does not require modification of code or binary, but due to the unavailability of debug information and compiler optimizations such as inlining, the mapping of performance data to specific lines in code might be inaccurate. Probing, on the other hand, requires that before and after sections of interest instrumentation code is added. In essence this code programs, resets and starts the performance counters prior to the execution of the section of interest and once it has finished it reads the final counter values and stores them in a data structure ready for further analysis. The overhead of probing might be significant, especially if the part of the program to be profiled is executed very often (see [48] for a practical example). On the other hand this approach is considered to be more accurate than sampling, since it does not interrupt code execution (and thereby avoids altering the state of the processor artificially) and the procedure is only minimally invasive with respect to the generated instruction stream, i.e. the only way in which the instrumentation code can have an impact on the section of interest might be compiler optimizations across the boundaries of the section which can not be carried out any more.

We will now limit our scope to the probing approach applied in the context of modern x86 microprocessors and give an overview on overhead and accuracy of performance counters reported in literature. A great amount of research has been done over the years concerning the overhead of various ways to access performance counters in user-space applications (see for example [41, 45, 49]). Due to its great importance for the profiling infrastructure presented later in this thesis we cite Table 3.2 from [41] measuring the different overheads of reading and writing (i.e. resetting, starting and stopping) a performance counter from within the kernel, a privileged user-space application accessing the msr device file through the LIKWID library (“direct”) or an unprivileged user-space application that uses the access daemon provide by LIKWID on an Intel Haswell CPU. The latter scenario is common in the context of data centers where user applications can not run with superuser privileges but still need to use performance counters. The LIKWID daemon, for example, is available on both the SuperMUC and the CoolMUC-2 systems at Leibniz Supercomputing Center (LRZ). The overhead of libraries based on the perf interface such as PAPI is found to be slightly larger.

Literature on the accuracy of the several hundred (e.g. >400 for Haswell) counters available in modern x86 processors, on the other hand, is limited. While metrics such as cycle count are by design accurate down to a single cycle, accuracy varies dramatically for others and error margins might be dramatic in certain scenarios. An extensive review on the accuracy of various performance counter metrics focused on the Haswell-EP nodes in SuperMUC Phase 2 is given in the PhD thesis of Carla Guillén Carías from LRZ [50]. These results are in agreement with research presented by Thomas Röhl from the Erlangen Regional Computing Center (RRZE) carried out on Intel Haswell CPUs that indicates that common metrics

Mode	Read	Write	Read (CoolMUC-2)
Daemon	13144	11388	8018
Direct	1292	656	
Kernel	888	312	

Table 3.2: Counter access time (median) in cycles for LIKWID on an Intel Xeon E3-1240v3 CPU (Haswell microarchitecture). Cited from [41]; own measurement performed on a node of LRZ's CoolMUC-2 system equipped with two Intel Xeon E5-2697v3 CPUs (Haswell-EP). See chapter 5 for details.

such as branch prediction ratio, instruction retirement, memory bandwidth and load/store ratio have an average error of less or equal than 0.2% [51]. Performance counters concerned with the multi-level caching mechanism and its complex non-uniform memory access characteristics are prone to larger error margins; an example would be L1-L2 cache transfer bandwidth with an average error $>5\%$ in some scenarios. The undocumented counter for Advanced Vector Extensions (AVX) floating-point instructions in general is surprisingly accurate for a range of numerical benchmarks ($< 0.05\%$), especially when compared to results from earlier microprocessor generations. Significant overcounting can happen in presence of certain AVX instructions such as the ones used to copy a subset of the values in a vector register to another; undercounting happens for example when fused multiply-add instructions are used.

In summary we can conclude that most common performance metrics obtained from the PMUs are highly precise and if used correctly are invaluable in HPM-assisted performance engineering. Most of the time the counter values are accurate and stable enough to allow not just for qualitative but also for quantitative comparison. Since however availability, semantics and accuracy of the metrics depend to a great degree on manufacturer, generation and model of the microprocessor as well as on characteristics of the scenario under consideration (e.g. single-core vs. multi-core), a lot of factors need to be taken into consideration before valid conclusions can be drawn. The overhead introduced by layers of abstraction between counter register and user application can introduce significant overhead compared to a simple register access in certain scenarios. However due to the constant nature of the overhead and the fact that it usually does not dominate the measurement, a well-calibrated profiling instrumentation can be tuned to take overheads into account so that the adjust metrics come remarkably close to the ones that could have been obtained from direct register access.

3.4 Energy Monitoring in Modern x86 Processors

The servers in common data centers and to a slightly lesser extent also in supercomputing centers almost never operate at their peak capacity all at the same time. This is the reason why today power supply to such facilities is usually underprovisioned, i.e. it is not designed with respect to the absolute peak requirement, but rather to some worst-case average power consumption [29]. This gives rise to a more recent introduction to x86 processors: On-chip power estimation and capping capabilities.

Starting in 2011 with the Sandy Bridge microprocessor architecture Intel introduced the so-called Running Average Power Level (RAPL) interface as a way to prescribe upper bounds on the current power consumption of a CPU [29]. The following year AMD presented a similar technology called Application Power Management (APM) [52]. For the resulting embedded control task to be feasible a sophisticated on-chip model to estimate the current power consumption based on architectural events was added [53]. Both technologies expose these estimates in form of model-specific registers that are updated at a rate of about 1kHz (Intel) or 0.1kHz (AMD) [54]. In the case of Intel CPUs these MSRs contain the total amount of consumed energy as a multiple of an architecture dependent base unit (e.g. $61\mu J$ or $15.3\mu J$ for Sandy Bridge and Haswell) since the last reset of said counter. The estimates cover so-called power domains so that typically separate values for the sum over all cores, for all cores plus shared on-chip resources and for DRAM memory can be obtained. AMD CPUs, on the other hand, provide power estimates again as a multiple of an architecture defined granularity (e.g. 3.8mW for the Bulldozer microarchitecture). From now on we will use the terms "RAPL counters" to denote on-chip power estimation capabilities by both vendors.

As mentioned in the introduction energy efficiency is considered to be one of the greatest challenges in designing software for exascale machines. Reliable, low overhead on-chip estimation of CPU and DRAM energy consumption is an invaluable tool in optimizing applications towards this variable and seems to be a favorable alternative in comparison with actual measurements based on complex hardware instrumentation. Before we illustrate how RAPL counters can be used within the profiling infrastructure for ExaHyPE, we first conduct a literature review on accuracy validation, major pitfalls, subsystem modeling (from package to core) and other important observations from practice. Due to a lack of literature on AMD hardware and the fact that most of our test systems as well as those used in literature are based on these architectures, we will mostly focus on Intel Sandy Bridge and Haswell CPUs. We express the hope that accuracy bounds that hold for these architectures will act as upper bounds for future microprocessor generations.

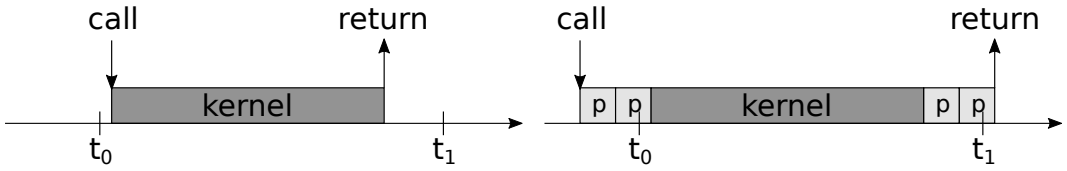
Hackenberg et al. [54] have used external equipment including high-frequency hall effect sensors to measure power consumption at the wall outlet and the mainboard power supply of two Intel Sandy Bridge systems and one AMD Bulldozer system. For a variety of benchmarks occupying all available physical cores for eight seconds they find a small constant offset between externally measured values and those reported by the RAPL interface. The offset is negative in general, i.e. RAPL slightly underestimates the amount of consumed energy; if the DRAM power domain is included the gap closes further and correlation increases. For computationally intensive tasks the offset is slightly lower than for memory-intensive tasks. The results furthermore shows that Sandy Bride reports an almost exact, only slightly overestimated value for an idling system; AMP on Bulldozer does not seem to compensate correctly for cores in low power states. The group highlights that measurements close to the counter update rate are challenging due to the variability in the time deltas between two consecutive updates and the lack of a precise time stamp that could compensate for it. In a more recent publication Hackenberger et al. repeat their measurements for Haswell-EP and find "a significant improvement compared to RAPL on previous processor generations." The correlation for the sum of package and DRAM RAPL is "almost perfect" and if in addition a quadratic fit is employed the deviation does not exceed 3W in

any measurement [55]. They point out that due to a new RAPL mode for the DRAM domain that is based on actual measurements the reported values are now extremely precise throughout all load scenarios.

Desrochers et al. [56, 57] instrumented a Haswell desktop system in a similar way as in [54]: Power supply is intercepted at the wall outlet, at the mainboard connector and due to their special interest in RAPL estimates for DRAM also at the respective memory banks. They again find a small constant offset in the reported RAPL CPU package values for both idle and fully loaded scenarios. If DRAM dims are idle and enter a lower power state RAPL on Haswell (unlike Haswell-EP in [54]) strongly underestimates its power consumption.

Hähnel at el. [58] use a "manually instrumented board" to compare power consumption values reported by the RAPL interface of a Sandy Bridge processor to measurements done with external equipment. They again find "that the curves' characteristics are identical", but measure a constant offset they attribute to the fact that they neglect DRAM power consumption altogether. They are particularly concerned with measuring short code paths ($<$5ms). Since the RAPL counters are only updated at a rate of 1kHz with jitter of about $\pm$2% this task is particularly challenging. Figure 3.2a illustrates the problem: We can not align the start of our section of interest labeled "kernel" with a counter update, in fact we do not even know precisely when the latter will happen. Similarly the point in time when the function returns will in general not be aligned with an update so that in this naive approach we would wrongly attribute energy consumption from before and after the function call to the measurement. Figure 3.2b, on the other hand, illustrates a possible solution. Before starting the actual "kernel" we first repeatedly poll the RAPL MSR and count the number of tries that are need until a change of the counter value is observed. We then execute the kernel and repeat the same procedure once it has finished. Via a priori calibration we can precisely determine the amount of energy that is required for one single poll simply by executing the procedure sequentially for a significant number of times and then computing the mean as the value for a single register poll. Since the number of polls we had to wait for the counter updates to happen is known we can correctly compensate for this overhead and adjust the overall energy measurement such that it only reflects the kernel function. In a case study based on single-core video decoding it was shown that if a frame is split into twelve slices and the RAPL energy consumption based on the approach explained above is measured for processing each slice individually, the sum over all slices on average differs by 1.13% compared to a measurement spanning the processing of the complete frame at once [58]. Since in ExaHyPE we will mostly be dealing with short kernel calls close to the minimum temporal resolution of RAPL energy counters we implemented this refined measurement procedure in the profiling infrastructure to be discussed in the next chapter.

With current generation x86 processors RAPL counters can not be used to obtain energy consumption estimates for individual cores. A way to get around this limitation is largely based on research tasked with predicting CPU energy consumption from a time before RAPL counters were introduced: Based on selected performance counters and other information available at runtime such as core temperature and clock speed, even relatively simple models were able to make remarkably accurate predictions. If these models are calibrating against RAPL es-



(a) Naive approach

(b) Approach based on repeated counter probing

Figure 3.2: Approaches towards measuring power consumption in short code paths using RAPL counters. For the illustration we assume $t_1 - t_0$ to be of the order of milliseconds and counter updates happen at t_0 and t_1 and at unspecified times in between if $t_1 - t_0 \gtrsim$ 2ms. The dark gray box indicates execution of the kernel, a light gray box labeled "p" indicates one instance of register probing. The width of the box is not true to scale and in practice we observe on average about 150 probes before the counter value changes (compare Tables 3.2 and 5.4). It has been shown that the approach illustrated on the right works even if the time it takes to execute the kernel is <1 ms. Illustration adopted from [58].

timates for the complete CPU they can be used to predict the energy consumption of a single core. A sever limitation of all these models is, however, that they only work if dynamic frequency scaling is disabled and if the frequency for each core is the same. Results on how the power consumption changes without fixed core frequencies in a node of CoolMUC-2 is presented in chapter 5.2.

Goel et al. [59] proposed a model based on piecewise linear regression that takes into account CPU temperature and a hardware-specific choice of the four performance counters with highest correlation to predict power for each core individually. Validation against measurements of total CPU power consumption for various x86 microprocessors introduced between 2007 and 2009 yields a median error of 1.1% - 5.2% for a diverse set of benchmark scenarios.

Also prior to the general availability of RAPL counters Takouna et al. [60] proposed a linear model based on frequency (required to be the same for all cores), squared frequency and number of active cores:

$$P(f, n) = \theta_3 f^2 + \theta_2 f + \theta_1 n + \theta_0. \tag{3.1}$$

Validation against a Nehalem-EP server with build-in CPU power measurement capabilities yields that the prediction error is less than 7% for 95% of the tested combinations of frequency f and active cores n.

Yasin et al. [61] used a particularly simple model based on normalized frequency of the individual cores of the form

$$P(\boldsymbol{c}) = \theta_0 + \theta_1 \sum_{n=1}^{N} \boldsymbol{f}_n, \tag{3.2}$$

where N is the total number of cores in the system and $\boldsymbol{f}_n = \boldsymbol{F}_n / F_{\max}$ is the normalized frequency of the n-th core. For a system consisting of two Sandy Bridge processors and the model calibrated against RAPL power measurements they report a mean absolute error of 4.01% and 5.23% if core power saving states (C-states, P-states) are disabled or enabled, respectively.

Even though literature in the context of RAPL counters is admittedly limited and oftentimes methodically unclear or questionable, we can still conclude that the approach of using simple linear models to predict core-level power consumption from per package aggregates is promising if core frequencies are fixed. In practice, however, careful calibration and extensive sanity checking is required on a case-by-case basis. Since RAPL counters on Intel platforms report total energy consumption rather than average power intake and due to the jitter in the update rate described above it is justified to assume that estimation of per-core energy consumption will be more accurate than of per-core power consumption. If core frequencies are not fixed as it is the case in most realistic scenarios more complicated environment-specific models need to be used (see for example [62, 63] for models developed and used at LRZ).

Summing up the key findings of the literature review we can conclude that RAPL counters as the standard way for energy monitoring on modern x86 processors are an important tool when optimizing applications for energy consumption. Under no circumstances do the provided estimates violate the ordering constraints of instruction streams with respect to an ordering based on real measurements with external instrumentation. They can therefore be used as a cost function to guide manual or automatic optimization. The accuracy of the counters seems to be good enough for most practical use cases; sampling frequencies at the same order of magnitude than that at which updates occur are challenging, but possible using the probing approach presented above. Nevertheless improvements with respect to temporal granularity in future architectures would be highly beneficial. The estimation of power consumption for individual cores on a multi-core processor using simple linear models seems to be feasible and usually results in low relative errors, however this is only possible if core frequencies are kept at a fixed level.

Chapter 4

Performance and Energy Profiling in ExaHyPE

In this section we will present in great detail the design and implementation of a generic performance and energy profiling infrastructure built into ExaHyPE as the main contribution covered by this thesis. The chapter is organized as follows: We will begin our discussion with an introduction to the ExaHyPE project at first focusing on high-level aspects such as where it stands in the context of current trends in Scientific Computing, what the key objectives and benefits are, which techniques and approaches it employs and what the agenda looks like with respect to past and future work. We will then give a brief summary on the architecture and usage of the ExaHyPE engine itself from a user's perspective since this defines the environment for all efforts discussed thereafter. Once this is done we can finally focus on the generic profiling infrastructure that was contributed to the project. We will again begin with some general considerations on the motivation behind the effort and what in consequence the main design goals are. After that we will illustrate the general architecture of the proposed solution, present exemplarily four implementations based on different profiling libraries and discuss ideas for future work.

4.1 The ExaHyPE Project

4.1.1 Context

Before we take a closer look at ExaHyPE, let us first consider three important aspects of contemporary scientific computing that will directly lead to the fundamental motivation that drives the project:

Simulation Pipeline: According to Bungartz et al. [64], generic simulation tasks in Scientific Computing can in general be subdivided into a set of steps as described by the so-called simulation pipeline depicted in Figure 4.1. The pipeline model is inherently iterative in nature and describes tasks that to a varying degree reappear in all modeling and simulation projects. For the most part all of these steps are already covered by some form of library, tool or approach with a proven track record in existing HPC applications. More often than not researchers nevertheless tend to "reinvent the wheel" and spend a significant amount of time on solving tasks that others have

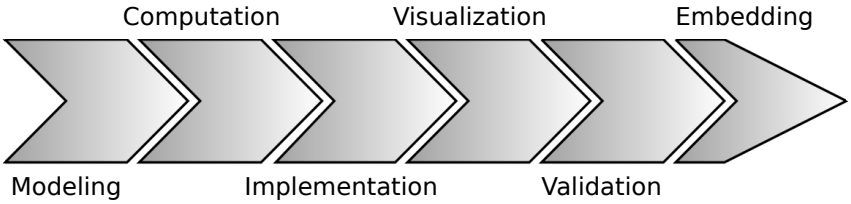


Figure 4.1: The simulation pipeline according to Bungartz et al. The process depicted is in general of iterative nature. Illustration adopted from [64].

already worked on extensively in the past that therefore can be considered solved with respect to state of the art requirements. Most of the time side efforts such as the development of a custom linear algebra system or a data structure for adaptive mesh refinement do not even meet the performance benchmark set by preexisting solutions. In an environment dominated by growing complexity and constrained availability of resources we would like to make the argument that the tasks as described by the simulation pipeline and subtasks thereof should be considered building blocks that need to be assembled correctly to fulfill the overall goal of such projects which is to enable new insights in the application domain. In the spirit of agile development it often makes sense to build a prototype based on existing solutions first and only after it has been determined that the overall solution does not meet the requirements to enable such insights, careful analysis and profiling needs to be applied to understand which parts are critical and need optimization. Then – only if the necessary expertise is available, of course – an effort to develop a new solution should be started and once it has been carefully validated shared with the scientific community as a building block to accelerate progress in other areas. The argument obviously has a lot of common sense built into it as this is normally the case for strong arguments, but too often in practice it is forgotten that nobody can be an expert on anything!

Exascale Computing: Simulation on supercomputers has obviously become a significant driver of scientific progress. First systems capable of performing more than 10^{18} floating-point operations per second, i.e. 1 ExaFLOP/s, are expected to be operational by 2020 [65]. A subset of the key challenges identified by Rosner et al. for the U.S. Department of Energy [1, 66] are illustrated in the following:

Energy consumption: Based on the simple extrapolation of the most energy-efficient system[1] currently listed on the Green500 list [35] an exascale machine would require about 150 MW. To put that in perspective the largest system in terms of peak power consumption[2] currently on the Top500 list [3] consumes less than 18 MW. Considering an average price of electricity for industrial customers in the U.S. of about 0.07 USD per kWh [67] this would result in annual expenses of more than 90M USD, a conservative estimate for Germany at 0.13 EUR/kWh would be about 170M EUR. The proposed energy target between 20 and 40 MW for en

[1]The Shoubu supercomputer at the RIKEN research institute in Japan provides 6673.84 MegaFLOP/s per Watt.

[2]The Tianhe-2 system at the National Super Computer Center in Guangzhou, China, consumed 17.808 MW while running the LINPACK benchmark [8].

exascale system [1] gives rise to doubts expressed by many in the HPC community [6, 68] whether the assumption of such a system being introduced by 2020 is realistic. Irrespective of the exact time frame, exascale computing will induce a major shift from the perception that energy efficiency is a hardware-only challenge towards the understanding that it needs to be considered equally from both a hardware and a software perspective.

Multi-level parallelism and hybrid architectures: Since the clock frequency of chips remained more or less stagnant over the last couple of years and is even expected to decrease in an effort to reduce power consumption, increased parallelism will be the main driver of performance gains to a much stronger degree than it already is today [66]. Parallelism will therefore increase on all levels starting with instructions (pipelining, superscalar execution, VLIW) over to cores and nodes (multi-core, MIC architectures, co-processors, accelerators) up to cross-site computation (multi datacenter computations, PRACE). Designing solutions that can make use of this trend towards "multi-level parallelism" on all levels is a great challenge both with respect to hardware and software.

Fault tolerance and resilience: If the number of cores in a chip and the number of nodes in a system increases dramatically as it will necessarily be the case in an exascale supercomputer it is an obvious observation from probability theory that the failure of components will be of significantly greater concern than in today's petascale systems. It is therefore unavoidable to consider the increased probability of hardware defects in HPC software solutions that operate at this scale. Similar to common frameworks for distributed data processing and storage available already today such as Google Flume [69] or Apache Hadoop [70], resilience and fault tolerance must be a major design goal from the very beginning on.

Memory and network bandwidth bottlenecks: Data collected by Patterson and Hennessy indicates that between 1980 and 2010 processor performance increased by approximately 30% on a year by year basis. Memory bandwidth, however, only increased by about 7% annually during the same time [71]. The observation that improvements in memory technology with respect to latency and bandwidth can not keep track with advances in microprocessor performance is commonly referred to as the memory wall [72] or the memory gap [73]. A similar observation can be made for interconnection networks that provide a link between the individual compute nodes in HPC systems [74]. As a software side answer towards these challenges induced by divergent rates of performance growth and the resulting limitations of modern hardware, novel strategies and schemes that foster even more principles such as data locality and algorithmic density must be developed to achieve scalable performance on future supercomputer architectures.

The considerations above have illustrated that there are major challenges towards exascale computing. In order to overcome them it is necessary to

Figure 4.2: Logo of the ExaHyPE project [79].

consider both the hardware and the software aspects of supercomputer development on equal terms. We will illustrate in chapters 4.1.2 and 4.1.3 how the ExaHyPE project directly tackles the challenges in the latter area and underscore the importance of the contributions in the context of exascale HPC.

Hyperbolic Balance Laws: The general form and a state of the art method to solve hyperbolic balance laws (HBLs) has been given in chapter 2. HBLs are of great interest in Scientific Computing since they can be used to model a variety of phenomena in physics and engineering. There a numerous areas of application including, but not limited to mechanical engineering, geophysics, electrical engineering, biology, chemistry, astrophysics, material science, civil engineering and even in economics, urban planing and the social sciences. Particularly interesting cases where numerical simulation of HBLs lead to new scientific insights reported in literature include earthquakes [75], pollutant transport [76], traffic modeling [77] and the merger of neutron stars [78]. According to Rosner et al. forefront computing problems can be divided into three categories, namely incrementally advanced computing, voracious computing and transformational computing [1]. Simulations based on HBL models can fall into all three categories. For problem scenarios that fall into the fist two categories the advance to exascale computing will enable a more fine-grained understanding of the characteristics of the simulated phenomenon and allows for more accurate risk assessment at larger scales. For problems of the third category that simply can not be simulated on today's petascale machines, however, exascale systems as a result of the incorporation of novel approaches in both hardware and software will lead to new fundamental scientific findings that can really transform disciplines.

4.1.2 Vision, Objectives and Benefit

The ExaHyPE project addresses the software side challenges posed by supercomputer development towards exascale [65]. Similar to the notion of a standardized engine commonly used in the context of computer games, the project seeks to build an engine for solving arbitrary problems formulated in terms of hyperbolic balance laws [80]. The abbreviation ExaHyPE stands for "exascale hyperbolic PDE engine"; any similarities to words in the English language are purely coincidental[3].

[3] Quotation needed...

Based on their strong expertise in areas such as numerical treatment of HBLs, large-scale simulations on some of the world's fastest HPC systems and in-depth application domain knowledge the four-year ExaHyPE project is jointly carried out by a consortium of seven institutions in Germany (Technische Universität München, Ludwig Maximilian Universität München, Frankfurt Institute for Advanced Studies, Bavarian Research Alliance GmbH), Italy (Università degli Studi di Trento), the United Kingdom (Durham University) and Russia (RSC Technologies). The consortium in addition is associated with the Leibniz Supercomputing Centre (LRZ) of the Bavarian Academy of Sciences and Humanities and cooperates with the Joint Supercomputer Center (JSCC) of the Russian Academy of Sciences. The project was started in October 2015 and has received funding of 2.8M EUR from the European Union's Horizon 2020 research and innovation program under grant agreement No. 671698.

The objectives of the project [79] directly reflect the challenges that are associated with the transition towards exascale computing as presented in chapter 4.1.1:

Energy efficiency on tomorrow's supercomputer hardware: The novel algorithmic and mathematical approaches developed as part of the project are inherently designed for energy efficiency and implemented in a way such that they can easily be optimized with respect to the requirements of future energy-efficient hardware.

Scalable algorithms through dynamic adaptivity: To achieve the greatest possible accuracy in a simulation with minimal computational effort, algorithms for dynamic adaptivity will be integrated into the ExaHyPE engine by experts with decades of accumulated experience in the field and a proven track record in designing scalable solutions for adaptive refinement.

Compute-bound simulations in spite of slow memory: The novel mathematical formulations and schemes used in ExaHyPE were designed specifically with important principles such as high algorithmic density and good data locality in mind. They are particularly memory efficient and reduce the required amount of communication during parallel processing to a minimum. As a result simulations based on ExaHyPE will be able to efficiently utilize future HPC systems in spite of ever-growing network and memory performance gaps.

Extreme parallelism on unreliable hardware: Resilience and fault tolerance are key design goals of ExaHyPE. The engine will be able to dynamically schedule computations on millions of processor cores and guarantees successful completion of the simulation even if individual nodes fail in the process.

The ExaHyPE project will empower application domain experts in a multitude of fields from both academia and industry to achieve scientific progress by enabling incrementally advanced, voracious and most importantly transformational computations on future exascale systems. The engine designed by computer scientists with vast experience in the development of large-scale HPC applications combines sublime performance on a wide variety of current and future hardware architectures with ease of use so that even researchers with only little programming experience can fully exploit available resources without ever leaving their core area of expertise.

A distinctive property of the ExaHyPE project is the fact that the consortium does not only include researchers with a background in HPC and numerics, but also experts from two key areas of application, namely astrophysics and geophysics. Researchers from all contributing groups work together as equal partners in order to never lose focus on the requirements of future users of the engine. In the area of geophysics ExaHyPE will enable simulations of earthquakes and aftershocks to assess risk scenarios at unprecedented accuracy. This work will fundamentally improve our understanding of what happens during large-scale earthquakes and helps to increase "the level of preparedness [...] in expectation of the next 'Big One'" [75]. With respect to applications in astrophysics ExaHyPE will enable researchers to tackle some of the long-standing mysteries in this field and simulations made possible by the engine are expected to result in fundamental scientific findings that will help us to better understand the origin of the galaxy we all live in. In 2019 ExaHyPE is to be released as an open-source package (BSD 3-Clause license) providing a user-friendly way to create state of the art, exascale ready simulation codes. This will help accelerate the pace at which application domain experts from both academia and industry can use simulations with ultimate benefits for society as a whole.

4.1.3 Approach and General Architecture

ExaHyPE is a spacetree-based engine for solving hyperbolic balance laws designed to exploit tomorrow's exascale computing infrastructure [81, 82]. It consists of three major building blocks to be discussed in the following:

High-order space-time Discontinuous Galerkin (ADER-DG) method with a–posteriori finite volume based subcell limiting: For solving hyperbolic balance laws an ADER-DG scheme limited on a subcell level based on the MUSCL-Hancock FVM scheme as described in chapter 2 of this thesis is embedded into the very core of the engine. This state of the art method of arbitrary order in space and time has very favorable properties with respect to application in HPC codes: The introduction of a cell-local space-time predictor as described in chapter 2.8 makes the scheme inherently communication-avoiding. The fixed-point iteration that is used to obtain the latter is very favorable in terms of algorithmic density, i.e. the large ratio between the number of floating-point operations and the total memory transfer volume supports the transition from a memory-bound to a compute-bound region.

Adaptive Cartesian grids and dynamic load balancing: To minimize computational cost and to achieve the greatest possible accuracy at the same time, ExaHyPE employs adaptive mesh refinement (AMR) on quadrilateral Cartesian grids. Since ExaHyPE uses the Peano framework [83] as its underlying data structure and as the main driver for load-balanced parallelized grid iteration, technically speaking the engine itself is a "Peano application." Peano is an open-source C++ solver framework that operates on spacetrees that employ the space-filling Peano curves for optimal cache-awareness. It can be used to solve multiple problems within one grid and has a proven track record as the foundation of numerous research projects (e.g. [84, 85, 86]) that spans over a decade.

Hardware-specific optimization of dominant compute kernels: An implementation of the ADER-DG scheme derived in chapter 2 essentially boils down

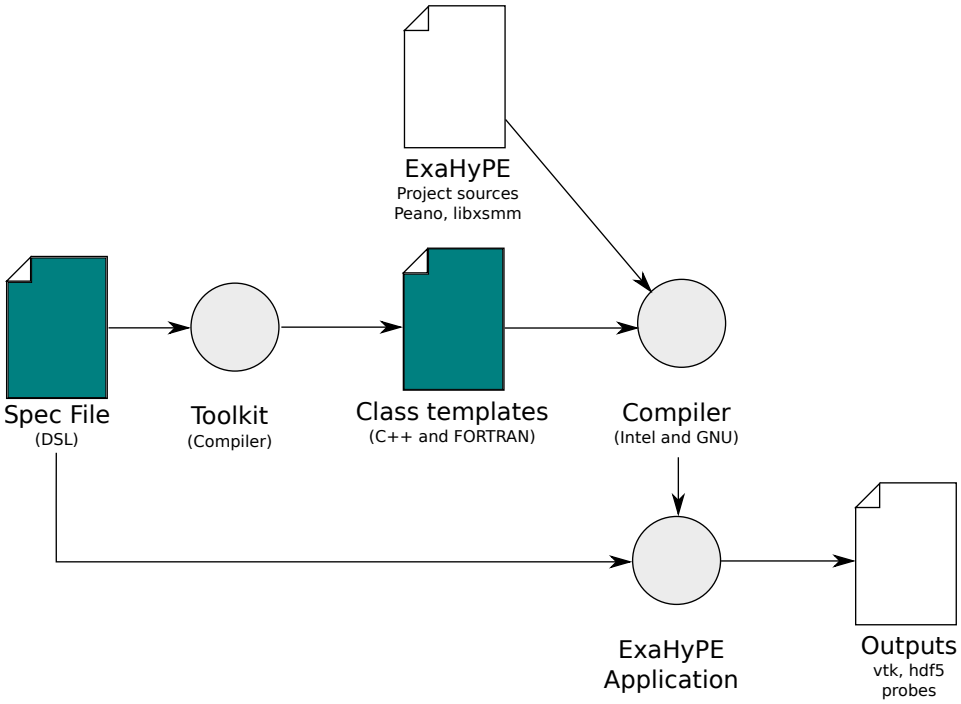


Figure 4.3: The ExaHyPE workflow as described in [89].

to a number of matrix multiplications between precomputed operator matrices and matrix-valued tensor slices. The dimensionality of the appearing matrices is $(N+1) \times (N+1)$, where N is the polynomial degree of the ansatz functions ("the order of the scheme"), which even though arbitrarily high in theory, for practical applications reported in literature usually one has $N \in \{3, 4, \ldots, 10\}$. Most libraries that provide high-performance linear algebra operations (e.g. all standard BLAS implementations) are optimized for large operands and therefore only perform unsatisfactorily for "small"[4] matrices.

The ExaHyPE engine comes with a set of generic kernels implemented in standard C++ suitable for rapid prototyping, but for the dominating small matrix operations to achieve the highest possible performance on current and future (Intel) x86 devices, i.e. on both CPUs and accelerators, ExaHyPE can make use of the LIBXSMM library [87] to generate optimized versions of the compute kernels for the specific hardware at hand. The use of optimized kernels is essential to achieve state of the art single-core/single-thread performance. Based on the polynomial degree fixed a compile time and the types of vector instructions available on the target machine LIBXSMM directly emits assembler code which is then linked into the ExaHyPE binary. The library is developed by Intel directly and has been used successfully in major HPC projects such as SeisSol (Gordon Bell Finalist in 2014, [75]) or CP2K [88].

[4] Let $\boldsymbol{A} \in \mathbb{R}^{M \times K}$, $\boldsymbol{B} \in \mathbb{R}^{K \times N}$, $\boldsymbol{C} \in \mathbb{R}^{M \times N}$ and $\alpha, \beta \in \mathbb{R}$. Then for our purposes we would for example consider the problem $\boldsymbol{C}'_{mn} = \alpha \boldsymbol{A}_{mk} \boldsymbol{B}_{kn} + \beta \boldsymbol{C}_{mn}$ small if $\sqrt[3]{MNK} \leq 80$. See github.com/hfp/libxsmm for the motivation behind this particular choice.

From a user's perspective the workflow to create simulations based on the ExaHyPE engine is schematically illustrated in Figure 4.3. It essentially consists of the following three steps:

1. The user needs to describe the problem at hand in a simple domain specific language (DSL). Information that needs to be provided includes the size of the domain, the total time the simulation should run for, the number of variables and spatially-varying (material) parameters the problem consists of, whether shared-memory and/or distributed-memory parallelization should be enabled, if and to which degree mesh refinement should take place, what the output of the simulation should look like and what language (C++ or FORTRAN) is preferred for the implementation of the problem-specific user functions. An example for a complete ExaHyPE configuration file can be found in Appendix B.
2. After the configuration file has been created it is passed to the "toolkit" application that is provided together with the actual engine. The toolkit parses the configuration file and generates glue code that is specific to both the problem and the platform under consideration. In the process stubs for all problem-specific functions such as those to prescribe eigenvalues, fluxes, sources and initial condition are created inside a folder separated from the actual ExaHyPE code.
3. The final step towards creating a complete simulation application is the implementation of problem-specific functions by the user. Only at this stage the actual "physics" of the HBL is specified. As mentioned earlier the complete problem is described in terms of fluxes, eigenvalues, sources and initial conditions. If boundary conditions other than standard reflection or periodicity are required this can furthermore be implemented using a special API. Native interfacing with C++ and FORTRAN is supported.

Once these three steps have been completed all that is left to do is to execute the provided build script using GNU Make and to run the simulation.

A general paradigm of the ExaHyPE project is to create one specification file per experiment/machine/setup and to sacrifice some degree of flexibility in exchange for performance. This requires the user to rerun the toolkit and to recompile the application after almost all changes to the configuration, but ensures that runtime overheads are minimized and aggressive compiler optimizations can be leveraged to enable maximum performance of the resulting simulation binary. As the guidebook [80] states "ExaHyPE prefers to run the compiler more often rather than working with one or few generic executables."

4.1.4 Current State and Next Steps

The ExaHyPE project was started in October 2015 and will span a total of four years. One year after launch, i.e. in October 2016, a prototype of the ExaHyPE engine has to be delivered that illustrates

- convergence with respect to analytic benchmarks,
- high performance of the optimized dominant compute kernels,
- low overhead of the simulation management code, as well as

- load-balancing and scaling on a petascale machine.

An incomplete list of work packages for the upcoming years includes the implementation of advanced techniques with respect to fault tolerance and resilience, the integration of tooling to measure and optimize for energy consumption (based on the contributions discussed in this thesis) and provision of ways to leverage sophisticated solutions for parallel IO and visualization.

All of the results that will be presented in chapter 5 of this thesis are preliminary and given merely to discuss the implemented techniques and to illustrate their use in practice. For results on the performance of a version of the ExaHyPE engine accessible to the general public the reader must be referred to future publications.

4.2 A Generic Profiling Infrastructure for ExaHyPE

In this chapter we will discuss the main contribution of this thesis, a generic profiling infrastructure designed for extensibility and ease of use built directly into the core of the ExaHyPE engine. The section is structured as follows: First, we will begin with a quick discussion on why such tooling is necessary and what benefits it comes with. We will then focus on the key design goals as well as the resulting architecture and interface derived thereof. Four implementations of the latter based on libraries and interfaces covering various aspects of performance profiling are presented. We conclude with a quick review of ideas that might be interesting to implement in the future.

4.2.1 Motivation

As mentioned before, in essence the goal of the ExaHyPE project is to build a highly sophisticated, nevertheless easy to use engine for solving large HBL PDE problems on a variety of platforms with stellar performance. To obtain a baseline on the latter, to track progress in this area and to be able to compare it to other state of the art solutions it is essential to have sophisticated tooling for profiling available.

Besides the more traditional metrics for performance in HPC such as floating point operations per second (FLOP/s) and time to solution, in recent years it has become obvious that overall energy consumption and maximum power requirement are key metrics that large scale simulation codes need to be optimized for. The two major computing centers in Bavaria, LRZ and RRZE, are pioneers in this area whose expertise is highly valued internationally by partners from both industry and academia. The ExaHyPE team should therefore incorporate as much of their knowledge as well as experience and as many of their technologies as possible into a profiling infrastructure for ExaHyPE, but still needs to keep the tooling simple enough such that even less experienced user can gain valuable insights.

4.2.2 Design

The key design goals that were identified for a generic profiling infrastructure are listed and ways to incorporate them into a consistent design are discussed in the following:

Ease of use: The instrumentation code necessary to enable profiling for regions of interest in the application should be as simple as possible. Similar to established solutions for performance probing such as the Google Census framework [90] or LIKWID's marker API [41] we therefore chose a design that employs a tag-based system to keep track of the various regions. Independent of the actual implementation yet to be discussed, instrumentation will then look similar to the following snippet of C++ code:

```
1 MyProfilerImplementation p;
2 p.registerTag("region1");
3 p.start("region1");
4 // work, work, work, ...
5 p.stop("region1");
6 p.writeToOstream(&std::cout);
```

Extensibility and support on a wide variety of HPC systems: In view of the plans to make ExaHyPE available as open-source software by 2019 it is essential to consider that the engine is expected to be deployed in numerous HPC systems that differ significantly not just in hardware, but also in terms of the employed software stack as well as available tools and libraries. This leads to the requirement that a profiling infrastructure for ExaHyPE needs to be designed with extensibility in mind. Researchers that require profiling to optimize their simulations in a new environment will then simply need to implement the provided profiling interface to be able to use available tools and libraries directly within ExaHyPE. Ideally this will also be a way to integrate community contributions into the project.

Low measurement overhead: The proposed generic profiling architecture for ExaHyPE forms a layer on top of existing tools and libraries. This layer is responsible to manage the assignment of individual measurements to the defined regions of interest. In order to ensure low measurement overhead it needs to be kept as thin as possible both in terms of memory and runtime requirements. Since they allow constant-time updates of stored measurement values, hash maps of carefully chosen size seem to be a suitable data structure for the task at hand.

Low maintenance effort for the ExaHyPE team: Code used for profiling is inherently specific to the hardware and software stack it was written for. In the case of hardware performance counters and model specific registers found in modern x86 processors, for example, support, naming and accuracy of events usually vary between microprocessor generations or even individual models; the consistent lag in or lack of publicly accessible documentation makes maintenance of such code a great challenge [41]. Since development of this kind of code is neither a key objective of the ExaHyPE project nor a particular area of expertise of the team, a generic profiling infrastructure must therefore be designed such that most if not all maintenance effort can be offloaded to the developers of specialized libraries.

Validation and sanity checking: A profiling infrastructure for ExaHyPE should be able to make use of all available sources of information. With respect to energy profiling, for example, this means that depending on availability in a particular HPC system we want to access data from software, hardware and

hybrid measurement units and whenever more than one interface is available to obtain them (e.g. perf and msr) we want to be able to switch between all of them. Only in this way can we assess the accuracy of our measurements and gain maximum confidence by employing techniques such as cross-validation and other types of sanity checking. If for a given system several ways to obtain certain metrics are available but only one or even none has yet been implemented for use with ExaHyPE this again stresses the importance of extensibility as a major design goal discussed above.

Flexibility with respect to internal and external analysis: Performance metrics obtained via the generic profiling interface need to be accessible for both internal use within the engine (e.g. for optimizations at runtime) as well as external post-processing and analysis. Internally this can be ensured simply by making the object that manages profiling visible and accessible within the relevant scopes. For external analysis and post-processing the generic profiling infrastructure provides standardize export capabilities to common formats such as CSV or JSON.

4.2.3 Architecture and Interface

Figure 4.4 illustrates the general architecture of the profiling infrastructure and hints towards examplary implementations based on different libraries for performance analysis to be discussed in the following chapter. As mentioned earlier in the discussion of design goals and ways to meet them, the virtual class `Profiler` prescribes that each concrete implementation of it needs to support a tag-based system to manage measurements with respect to predefined regions of interest in the actual simulation. In essence it describes a profiler as an object that exposes just two methods for performance measurements: `void start(const std::string& tag)` and `void stop(const std::string& tag)`. As already described above, a region of interest such as a kernel call will then simply be surrounded by calls to these two methods. For the sake of decreased overhead during the actual measurement, before a tag can be used for profiling, depending on the actual implementation, it either should or has to be registered via `void registerTag(const std::string& tag)`. Registration of a tag in the various implementations that are presented exemplarily in the following allocates the necessary field in one or more hash maps (e.g. `std::unordered_map<std::string, valuetype>` in C++) and initializes the fields to zero for aggregated values. In general the interface does not prescribe if a profiler has to report aggregated or individual measurement values. Registering tags in advance reduces the one-time overhead when the first measurement for this tag is started.

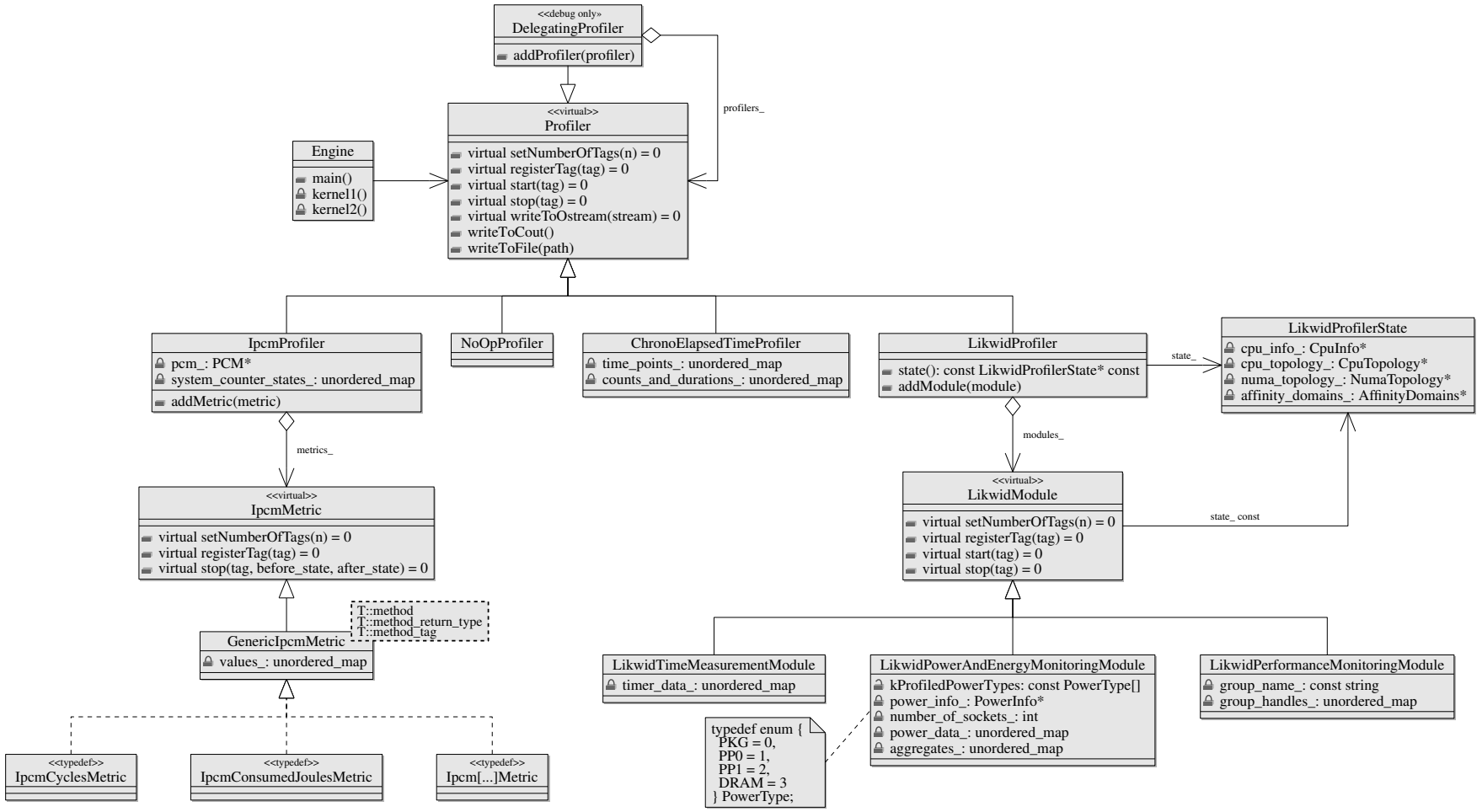


Figure 4.4: Class diagram illustrating the proposed generic profiling architecture and exemplary implementations based on different libraries for performance analysis.

Integration of the profiling infrastructure into the overall project has been carried out as follows: A new optional block in ExaHyPE's domain specific configuration language is used to enable profiling and to specify the desired profiler type and metrics (see Appendix B for details). Profiling is disabled by default. Furthermore, a member of type `std::unique_ptr<Profiler>` was added to the engine's `Solver` class. During initialization of the solver and based on the parsed configuration file available at runtime a `ProfilerFactory` creates the desired type of profiler. The solver object takes ownership of the profiler, i.e. the existence of the profiler is tied to the life time of the solver. It is responsible to register the required tags, e.g. one for each kernel in the solver. If profiling is enabled, the toolkit is used to add the necessary instrumentation that calls the profiler to start and stop the measurements. It simply becomes part of the glue code that needs to be generated either way to connect the ExaHyPE kernels and the problem-specific functions provided by the user.

After the simulation has finished the profiler can export the collected data via a call to `void writeToOstream(std::ostream* stream)` to an arbitrary output stream. JSON (JavaScript object notation) has been chosen as the default output format due to its simplicity, flexibility and widespread support by almost all common tools that might be used for analysis and visualization of the data. Export to the even simpler CSV (comma-separated values) format is available as well. If required in the future it would be trivial to implement export to other file formats such as XML (extensible markup language).

4.2.4 Exemplary Implementations

As part of this thesis four implementations of the profiling interface discussed in the previous section were developed and are to be presented in the following.

STL Chrono

A relatively trivial implementation of the interface called `ChronoElapsedTimeProfiler` makes use of the chrono library that is part of the C++ Standard Template Library (STL). It can be used to count the number of times a particular region in the code has been executed and to measure the overall wall clock time spent therein. Since an implementation of the STL is shipped with all operating systems and C++ compilers, the profiler does not have any external dependencies. The accuracy of chrono depends on the specific combination of compiler, operating system and hardware; it is usually at the order of nanoseconds. The overhead induced by profiler and library is analyzed at startup; results are presented in chapter 5.

Intel Performance Counter Monitor

The Intel Performance Counter Monitor (PCM) library [91] provides access to RAPL and performance event counters[5] from within a user-space application in an easy to use way. The library only supports Intel CPUs and is available as open-source software under the same BSD3-Clause license as ExaHyPE. The class `IpcmProfiler` that is shipped as part of ExaHyPE is a thin layer on top of PCM that is responsible for initialization, deinitialization and management of

[5]PCM uses the msr kernel module on Linux and a custom driver on Windows.

Value Type	PCM Function	Explanation
`double`	`getCycles`	Number of core clock cycles
`uint64`	`getIntructionsRetired`	Number of retired instructions [6]
`double`	`getIPC`	Average number of retired instructions per cycle
`uint64`	`getBytesReadFromMC`	Number of bytes read from DRAM memory controllers
`uint64`	`getBytesWrittenToMC`	Number of bytes written to DRAM memory controllers
`uint64`	`getL2CacheHits`	Number of L2 cache hits
`uint64`	`getL2CacheMisses`	Number of L2 cache misses
`uint64`	`getL3CacheHits`	Number of L3 cache hits
`uint64`	`getL23acheMisses`	Number of L3 cache misses
`double`	`getCyclesLostDue-L2CacheMisses`	Estimate of how many cycles were lost due to L2 cache misses
`double`	`getCyclesLostDue-L3CacheMisses`	Estimate of how many cycles were lost due to L3 cache misses
`double`	`getConsumedJoules`	Energy consumed by the CPU
`double`	`getDRAMConsumedJoules`	Energy consumed by DRAM

Table 4.1: Metrics provided by version 2.11 of the Intel PCM library [91] accessible via `IpcmProfiler` in ExaHyPE.

the mapping between individual measurements and regions of interest. An instance of `IpcmProfiler` owns a set of `IpcmMetric` objects as specified by the user in the configuration file. Each implementation of `IpcmMetric` can be mapped to exactly one of the metrics that the library provides. In PCM a metric is represented by a function of the form `double/int get...(SystemCounterState& before, SystemCoutnerState& after)`, where `...` is a placeholder for the name of the actual metric. Due to this particularly homogeneous structure all provided `IpcmMetric` implementations are simply instantiations of a single C++ template called `GenericIpcmMetric`. See Table 4.1 for the subset of metrics provided by version 2.11 of the Intel PCM library that are accessible via `IpcmProfiler` in ExaHyPE.

A binary that uses the PCM library to collect performance metrics is required to be executed with superuser privileges or needs the accompanying `pcm-sensor` daemon to be accessible. In consequence PCM-based profiling does not work on LRZ's SuperMUC and CoolMUC-2 by default at the moment. A minor drawback of using PCM in conjunction with ExaHyPE is the fact that the library uses exceptions, but exceptions are disallowed in ExaHyPE. The author has created a patched version of PCM 2.11[7] so that the library does not throw exceptions but aborts the

[6] Modern x86 processors employ speculative execution, i.e. they execute more instructions than absolutely necessary in an effort to accelerate the overall computation. An illustrative example might be a situation where the CPU encounters a conditional jump instruction. If the condition depends on a value which must be fetched from memory and in consequence would stall execution, the CPU may decide to execute both possible branches in the meantime and once the condition can be evaluated only the instructions on the correct branch are "retired."

[7] The patched version is available at git.io/vis7G.

application instead[8]. At startup `IpcmProfiler` performs as set of measurements to estimate the overhead introduced by profiling.

LIKWID

Without doubt the most interesting implementation of the `Profiler` interface is based on LIKWID. While the authors originally described LIKWID, short for "Like I Knew What I'm Doing", as "a set of performance-related command line tools targeting X86 processors" [41], today it can be considered a full-fledged library for application profiling. Even though the C-API available since version 4.0 feels hard to use from a subjective point of view[9] and examples are still scarce, it exposes all of LIKWID's features related to areas such as performance counters, topology analysis and execution control (e.g. "thread pinning"). The framework is under active development and maintained by researchers at the Erlangen Regional Computing Center (RRZE). It is designed to be as lightweight as possible and optimized for low overheads.

The class `LikwidProfiler` acts as a thin adapter between LIKWID and the ExaHyPE engine. The profiler consists of one or more modules that all implement the `LikwidModule` interface. A module directly mirrors all of the functionality of the respective Likwid C-API module. At the moment the API consists of the following five modules:

Time measurement module: This module exposes low-overhead functionality to measure how long it takes to execute a certain region of interest both with respect to wall clock time or the number of CPU cycles.

Power and Energy monitoring module: Based on the msr kernel module this part of the API provides convenient access to the processor's RAPL counters to obtain estimates for power and energy consumption of the CPU.

Performance monitoring module: This module allows programming and readout of the various PMUs that are part of modern x86 CPUs. Similar to the event sets defined in PAPI, the developers of LIKWID provide a set of predefined groups which are designed to be independent of underlying hardware-dependent events with respect to details such as exact naming, semantics and availability. Concerning RAPL counters the API is somewhat redundant in the sense that both the performance monitoring module as well as the power and energy monitoring module offer the same (but in case of the latter more convenient to use) functionality.

Thread affinity module: Servers in an HPC context are usually more complex than normal desktop computers with respect to NUMA characteristics as well as number and kind of available compute resources (multi-core CPUs, accelerators, etc.). The threat affinity module exposes functionality to simplify management of this degree of complexity, to allow for convenient logical addressing of the available resources and to restrict execution of an appli-

[8]This really does not change anything in our case, since the library only throws exceptions if performance analysis does not work (e.g. due to lack of superuser privileges) and in these scenarios there really is not much we can do to recover from the exception either way.

[9]Within the scope of this thesis changes by the author have been accepted into version 4.1.2 of LIKWID to improve the usability of the C-API in a C++ application.

cation to certain parts of the hierarchy (e.g. pinning to a specific hardware thread).

Thermal monitoring module: This module provides access to model specific registers holding static (e.g. thermal design point, maximum core temperature) and dynamic (e.g. current temperature in a core) values relevant to thermal management of the CPU. Temperature sensors had been available in microprocessors long before any other form of hardware monitoring was introduced. Since temperature can be seen as an indirect measure for running average power consumption with very wide sampling width, historically it has played an important role in the modeling of CPU power consumption.

The `LikwidProfiler` manages initialization and finalization of the various Likwid modules, hides some of the rough edges of the relatively new C-API in a reasonably robust piece of C++ code and takes responsibility for the export of collected data. The class `LikwidPerformanceMonitoringModule` makes use of the predefined event sets to collect performance metrics in a platform-agnostic way. Amongst others, at the moment the following 20 groups are available within ExaHyPE: `BRANCH`, `CACHES`, `CLOCK`, `DATA`, `ENERGY`, `ICACHE`, `L2`, `L2CACHE`, `L3`, `L3CACHE`, `MEM`, `TLB_DATA`, `TLB_INSTR`, `FLOPS_AVX`, `CYCLE_ACTIVITY`, `PORT_USAGE`, `UOPS`, `UOPS_EXEC`, `UOPS_ISSUE` and `UOPS_RETIRE`. For the sake of brevity we omit a detailed discussion of the provided metrics and refer to the respective online documentation[10] instead. The class `LikwidPowerAndEnergyMonitoringModule` provides profiling with respect to power and energy consumption based on RAPL counters. It implements the approach presented by Hähnel et al. [58] described in chapter 3.4 to enable accurate submillisecond analysis. The module `LikwidTimeMeasurementModule` – as the name implies – allows time measurement with respect to elapsed wall clock time and CPU cycles. The library needs either root privileges or its own daemon running with root privileges to be able to collect performance metrics. The corresponding daemon `likwid-accessD` is available on SuperMUC and CoolMUC-2. This type of indirect access to performance data induces overhead as a result of the employed socket file communication mechanism and additional security checks by the daemon necessary to filter out dangerous or malicious requests. The profiler implementation will give estimates of the latter at startup so that data can be recalibrated accordingly. Likwid supports modern x86 CPUs by Intel and AMD as well as Intel Xeon Phi accelerators.

IBM Active Energy Manager

For the sake of completeness with respect to all available sources for power measurements/estimates on LRZ's SuperMUC and CoolMUC-2 systems, a profiler implementation based on IBM's Active Energy Manager (AEM) technology is provided. AEM can be used to monitor and manage power and cooling requirements of IBM servers such as the nodes of SuperMUC and CoolMUC-2 [92]. The accompanying Linux kernel module ibm-aem exposes settings and per-node measurements via a sysfs interface. Most relevant within the scope of this thesis is the fact that a real hardware measurement of the power consumption is available in form of a counter for accumulated total energy consumption[11]. The counter has a resolution in the order of Milliwatts and is updated about two times per second.

[10] https://github.com/RRZE-HPC/likwid/tree/master/groups

[11] On SuperMUC/CoolMUC-2: /sys/devices/platform/aem.0/energy1_input

Similar to the corresponding module for LIKWID, `IbmAemProfiler` implements the approach described by Hähnel et al. to allow for accurate measurements at a temporal resolution at the order of the counter update frequency or below. Even though AEM is inherently vendor specific, the provided implementation can easily be adopted to work with the somewhat standardized hwmon interface simply by changing the sysfs path[12]. Leaving aside a small amount of jitter in the exact update intervals the energy measurement provided by ibm-aem and the power consumption values as reported by the hwmon interface seem to be equivalent on SuperMUC and CoolMUC-2. The hwmon counters are update only once a second, i.e. their temporal resolution is inferior compared to the vendor-specific solution. In general the update rates of both modules are too low for use in the kernel profiling experiments discussed in chapter 5, but they have been used to validate the accuracy of the estimates provided by the on-chip RAPL counters. The interface nevertheless seems to be promising for future profiling efforts of longer code paths.

4.2.5 Future Work

An interesting area for future work will be the extension of the generic profiling infrastructure to parallel ExaHyPE runs both with respect to shared-memory and distributed-memory parallelization. In this context, however, most work will need to be done within Peano rather than ExaHyPE. The presented profilers can easily be extended to support shared-memory parallelism using locks or preferably thread-local storage and distributed-memory parallelism by introducing a separate communication context, but at the moment not enough information with respect to the state of the simulation and the lifetime of individual threads is available within the ExaHyPE context. Careful evaluation regarding preexisting and overlapping functionality as well as possible synergies must proceed all efforts in this direction. The end goal should be a profiling infrastructure which enables insights across all levels of the distributed application so that a complete picture can be used to guide optimization efforts targeted at performance and energy efficiency of the whole simulation.

At the moment the available profiling infrastructure is used exclusively for a posteriori analysis. A promising idea for future work would be to use the available metrics directly within the engine, e.g. for tasks such as dynamic cpu frequency scaling or load-balancing based on energy consumption. The current integration of the profiling infrastructure where a profiler object is "owned" by a solver only allows profiling of the individual kernels called by this very solver. In the future profiling has to be integrated even deeper into the engine so that a profiler has "global scope" and arbitrary regions of the code (e.g. a complete update step, communication managed by Peano, etc.) can be analyzed. Profilers based on the LIKWID framework should be ported from version 4.0 to version 4.1 for better portability and an increased number of available metrics groups.

In an effort to strengthen confidence in the available metrics and to increase the number of HPC systems for which the engine can offer profiling out of the box, it will be worthwhile to implement support for additional performance measurement libraries. Intel PCM and LIKWID both use the same msr kernel module

[12] On SuperMUC/CoolMUC-2: /sys/class/hwmon/hwmon0/device/power1_average

to obtain data; IBM AEM is inherently vendor specific. It would for example be interesting to include libraries based on perf/PAPI or other sources for direct measurements e.g. via the hwmon kernel module in conjunction with libsensors. Nowadays almost all supercomputing centers offer ways to obtain real hardware measurements for energy-related metrics. Support for such interfaces would be of great value in the assessment of on-chip metrics. The proposed interface could furthermore be used to add instrumentation for libraries that go beyond classical CPU performance profiling such as Valgrind [93] or MeterPU [94].

Chapter 5

Preliminary Profiling Results and Case Studies

5.1 Setup

All of the experiments to be discussed in the following were carried out on the CoolMUC-2 system[1] at the Leibniz Supercomputing Centre (LRZ). The system was brought into service in fall 2015 and has a total peak performance of 447 TFlop/s (356/500 as of June 2016). It consists of 384 nodes each equipped with two Intel Xeon E5-2697v3 CPUs and 2x64GB of main memory. See Figure 5.1 for a graphical representation of the topology. The nodes are equivalent to those in SuperMUC Phase 2 and run the same SUSE Linux Enterprise Server 11 SP4 (kernel 3.0.101-77) operating system. The interested reader is encouraged to see [95] for details on the employed techniques that make CoolMUC-2 one of the most energy-efficient HPC systems.

The ExaHyPE applications used in the experiments were compiled either with the GNU C++ compiler (`g++`) in version 4.9.3 or the Intel C++ compiler (`icpc`) in version 16.0.3. We will refer to them simply as "gnu" and "intel" in the following. Optimization flags "`-O3 -fno-rtti -fno-exceptions -march=native`" and "`-fast -fno-rtti -no-ipo -ip -xHost`" were used in conjunction with `g++` and `icpc`, respectively.

Throughout the complete section we will run simulations based on a toy model we denote as Euler200 in three space dimensions. The traditional system of Euler equations in conservative form consists of five variables,

$$\boldsymbol{u} = \left[\rho, \rho u, \rho v, \rho w, E\right]^T, \tag{5.1}$$

where ρ denotes the density of the fluid, u, v, w are the velocity components along the coordinate axes and E is the total energy per unit volume. If e denotes the specific internal energy of the fluid then it can be defined as

$$E = \rho \left(\frac{1}{2}(u^2 + v^2 + w^2) + e \right). \tag{5.2}$$

[1] The author would like to express his sincere gratitude for receiving a dedicated reservation on this otherwise heavily used system.

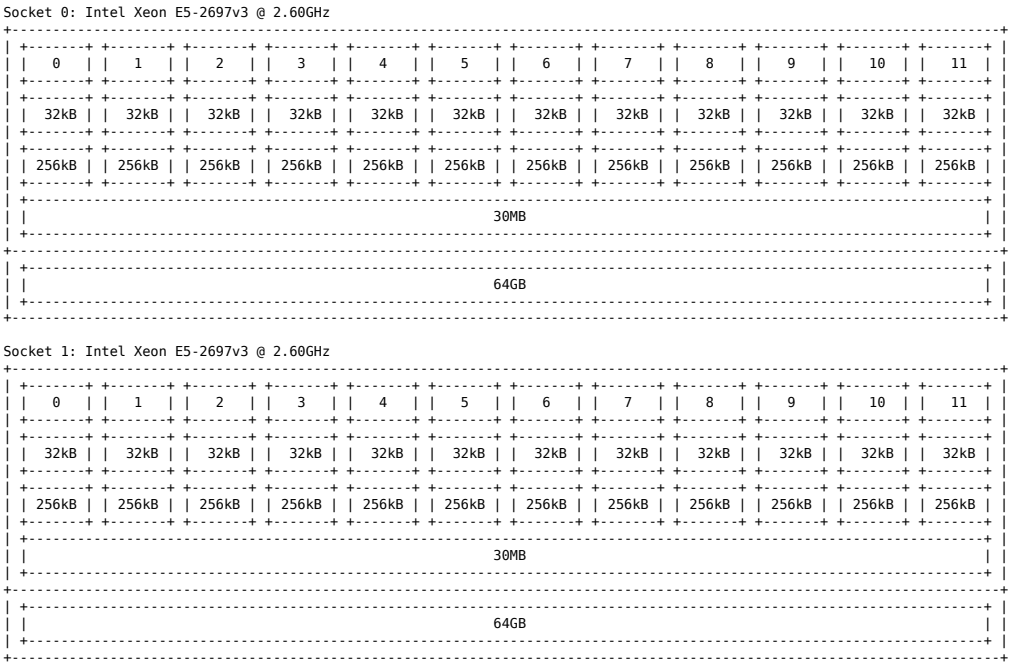


Figure 5.1: Graphical representation of the node topology in CoolMUC-2.

The corresponding flux $\boldsymbol{F}(\boldsymbol{u}) = [\boldsymbol{f}_1(\boldsymbol{u}), \boldsymbol{f}_2(\boldsymbol{u}), \boldsymbol{f}_3(\boldsymbol{u})]$ is given as

$$\boldsymbol{F}(\boldsymbol{u}) = \begin{bmatrix} \rho u & \rho v & \rho w \\ \rho u^2 + p & \rho uv & \rho uw \\ \rho uv & \rho v^2 + p & \rho vw \\ \rho uw & \rho vw & \rho w^2 + p \\ u(E+p) & v(E+p) & w(E+p) \end{bmatrix}, \tag{5.3}$$

where the pressure p must be defined via a suitable equation of state, e.g. for ideal gases one has

$$p = (\gamma - 1)\left(E - \frac{1}{2}\rho(u^2 + v^2 + w^2)\right). \tag{5.4}$$

The ratio of specific heats $\gamma = c_p/c_v$ is assumed to be a constant material parameter, e.g. $\gamma_{\text{air}} \approx 1.4$. See [10] for detailed derivations and thermodynamic considerations.

The Euler200 model is simply defined as a 40-fold concatenation of the traditional "Euler5" model, i.e. $\boldsymbol{u}_{200} = [\boldsymbol{u}; \ldots; \boldsymbol{u}]$ and $\boldsymbol{F}_{200} = [\boldsymbol{F}; \ldots; \boldsymbol{F}]$. It can be thought of as a model for the simulation of 40 independent fluids in parallel, e.g. within the scope of a parameter study. Even though this might be an interesting application, the foundation of the rational behind this particular choice of a model for our profiling experiments does not at all lie in any direct physical analogy or even practical applicability. It is really rather the case that in general a strong trend towards larger and therefore computationally more challenging problems can be seen in many fields of application. In astrophysics, for example, one of the initial areas of application of ExaHyPE, Rezzolla et al. have proposed a model of at least 58 equations for the simulation of binary neutron stars [96]. Within the scope of (relativistic) hydrodynamics in general systems quickly grow with respect to the

number of fluids that are involved and the kinds of interaction mechanism they are subject to [97]. Today in many cases researchers are forced to apply complicated approximations that are needed to ensure that a model does not become prohibitively expensive in view of limited computing resources. Exascale computing will push the boundaries of what simulations are deemed feasible to generate new insights and enable a more detailed understanding in numerous areas of application. The current ExaHyPE prototype was created to show what is possible on today's hardware. It therefore makes sense to demonstrate the profiling capabilities of the engine in a scenario that is – even though of limited immediate practical use – computationally challenging and in line with future trends. In some sense one might even go as far as to argue that the traditional Euler equation system has become a toy problem with respect to contemporary HPC.

The generic ADER-DG kernels of ExaHyPE that are used in the experiments in the following are intended for rapid prototyping purposes only. As mentioned in chapter 4.1.3, ExaHyPE uses LIBXSMM to generate optimized assembly code for the specific x86 architecture at hand. At the moment, however, there is additional work that needs to be done to make them easily available within ExaHyPE and to bring the initial version of the optimized kernels to feature equality with the generic kernels (and the ADER-DG schemes as presented in chapter 2). This work package has not been scheduled for completion within the time frame of this thesis. In the experiments and case studies presented subsequently we therefore work with the generic C++ kernels. They are very well suited to illustrate the value of the implemented techniques in practice and, even though it is not the primary goal to optimize the generic kernels, we will ensure in the process that we do not skip the "low-hanging fruits."

Throughout the complete chapter we define our domain as $\mathbf{\Omega} = [0.0, 15.0]^3$ and run our simulations on 9^3 equilateral cubes. We use "diffusing Gauss" initial conditions of the form $\boldsymbol{u}_0(\boldsymbol{x}) = [1, 0, 0, 0, \frac{1}{\gamma-1} + 2\exp(-\frac{|\boldsymbol{x}-\boldsymbol{x}_0|}{\sigma^3})]^T$, where $\boldsymbol{x}_0$ denotes the center of the domain. All experiments are executed single-threaded, but reserve a complete node. The simulation is pinned to processor core 0, all other cores remain idle. The reader may again be reminded that all results are preliminary and do in no way reflect the expected performance of the final ExaHyPE engine. The obtained data is used only to evaluate the implemented techniques and to make a strong case that underlines the importance of profiling and metrics-driven performance engineering.

5.2 Measurement Overheads

Before we can present preliminary profiling results from an ExaHyPE simulation we first need to quantify the measurement overhead induced by the instrumentation code, i.e. both temporal overhead that profiling adds to the runtime of a region of interest as well as the characteristic measurement offset that occurs even when an empty region is profiled. This will provide the necessary understanding of the limitations of the various approaches especially with respect to their temporal resolution. As stated earlier all data has been collected on the CoolMUC-2 system.

Let us begin the discussion exemplarily with results for the `IbmAemProfiler` presented in Table 5.1: We observe that a single energy counter read in the given

Time for single counter read:	1.740 ms
Energy for single counter read:	0.2477 J
Average number of reads until update is observed:	161.8
System power consumption during counter polling:	142.4 W
Average update interval:	0.563 s

Table 5.1: Measured overhead characteristics (averaged values) of `IbmAemProfiler` and derived quantities.

implementation takes more than 1.7 ms. This duration that we shall denote as t_a is three orders of magnitude longer than for example the time required to read out the value of a RAPL counter as presented subsequently. Even though the utilized file stream API of C++ used in the implementation to access and parse the counter value through the sysfs interface induces some minor overhead, the execution time is mostly dominated by the underlying kernel calls any file-like data access requires on Linux operating systems. Measuring the total energy consumption of a large number of counter readings and then dividing the obtained value by the iteration count we obtain an estimate for the energy consumption E_a of a single counter access. In our case we observe a value of about a quarter of a Joule. Employing the polling approach presented in chapter 3.4, on average the program reads the energy counter value 162 times until a change is observed. Assuming a uniform distribution of the number of register reads n then we can estimate the average time t_u that passes between two counter updates as $t_u = 2nt_a$. In our case we observe $t_u \sim 0.56$ s, a value which seems to be in line with the update rates documented by the vendor. For sanity checking we can compute the power consumption of the system during counter polling as about $P_a = E_a/t_a \approx 142$ W, again a reasonable value for the complete dual-CPU node with just a single busy core (operating system daemons slightly increase all given averages). In conclusion it becomes clear that the relatively large access overheads become somewhat negligible compared to the even lower update rates. IBM AEM is a very promising candidate for profiling long code paths such as a complete time step in the simulation, especially since it is hardware-backed and provides real energy consumption measurements, not just good estimates. For the purposes of profiling single kernel calls as it is the focus of this thesis, however, the low temporal resolution disqualifies the technique from further consideration.

Time measured for executing an empty function:	77.0 ns
Time measured for accessing the current time stamp:	115.7 ns
Time measured for accessing and storing the current time stamp in a small unordered map:	257.5 ns
Overhead attributed to the unordered map:	$\sim$ 142 ns

Table 5.2: Measured overhead characteristics (averaged values) of `ChronoElapsedTimeProfiler` and derived quantities.

Time measured for executing an empty function:	12.3 ns 40.6 cycles
Average CPU frequency	3.30 GHz

Table 5.3: Measured overhead characteristics (averaged values) of `LikwidTimeMeasurementModule` and derived quantities. Note that the CPU frequency could not be fixed for the tests.

The Chrono module of the Standard Template Library (STL) provides a platform independent set of functions and data structures to measure approximate elapsed wall clock time within C++ applications. On modern x86 platforms (Pentium and above) time is measured using a so-called time stamp counter (TSC), a register which counts the number of cycles since the last reset with respect to some fixed reference frequency independent of the current CPU/core frequency. On an application level time is usually reported in the unit of nanoseconds. In Table 5.2 overheads of the respective ExaHyPE profiler implementation are listed: Measurement over an empty region of code yields on average a value of about 77 ns, accessing a single time stamp (and issuing a subsequent memory store operation) is timed as about 116 ns. Since all of the presented ExaHyPE profilers use `unordered_map` as a hash map implementation to link individual measurements to a tag associated with a region of interest (e.g. a kernel), the time necessary to access and issue a store of the current time stamp into a small ($\approx$ 10 entries) map of this kind has been analyzed. Subtracting the timing and time stamp access overheads leaves us with about 142 ns which can be attributed to the update of the data structure itself.

Similar to the profiler based on Chrono also the `LikwidTimeMeasurementModule` uses the TSC counter to measure the number of reference cycles that have passed since the last reset to compute the elapsed wall clock time. It can furthermore read an additional fixed-purpose performance counter to similarly obtain the number of unhalted CPU cycles so that from the ratio between these two counts and the known reference frequency the actual average CPU frequency can be obtained. In general the overheads of LIKWID's time measurement module seem to be lower when compared to their Chrono counterparts. As illustrated in Table 5.3 only about 41 cycles or 12.3 ns pass between two subsequent accesses to the time stamp counter. This result seems plausible given that in essence the function provided by LIKWID consists of little more than just a single "RDTSC" special-purpose instruction. Again for the sake of checking the obtained valued for sanity we can compute the average frequency of the CPU core as 3.3 GHz, a reasonable value for the Intel Haswell-EP CPU under consideration when just one single core is in use.

As discussed earlier `LikwidPowerAndEnergyMonitoringModule` uses RAPL counters to provide an estimate for the energy consumption of a CPU and the associated main memory. For a single counter read via the LIKWID access daemon we measure an average overhead of about 3.3 μs or 8164 cycles (median reported in Table 3.2 was 8018 cycles). The average frequency of the active core can then be computed as 2.46 GHz, again a reasonable but this time lower value. Please be aware that manual frequency scaling was not available on CoolMUC-2 at the time the experiments where executed. Two results that are of great importance for the register polling approach presented in chapter 3.4 to work well are esti-

Time measured for single counter read:	3.32 μs 8018 cycles
Energy (RAPL PKG) for single counter read:	154.6 μJ
Energy (RAPL DRAM) for single counter read:	29.5 μJ
Average number of reads until update is observed:	158.0
Average update interval:	1.05 ms
Average core frequency:	2.46 GHz

Table 5.4: Measured overhead characteristics (averaged values) of `LikwidPowerAndEnergyMonitoringModule` and derived quantities. Note that the CPU frequency could not be fixed for the tests.

mated package and DRAM power consumption during a single read of these two counters. We obtain values of about 154.6 μJ and 29.5 μJ, respectively, resulting in reasonable estimates for the momentary power consumption (46.5 W for the CPU with one active core and 8.8 W for the DRAM). On average the counters need to be read 158 times until the values change resulting in an estimated update rate of 0.95 kHz. The measured average is slightly lower than the documented update interval of 1 ms for RAPL counters would indicate, but well in line with results by Hähnel et al. who observe periodic outliers due to interrupts [58].

The last measurement module under consideration with respect to overheads is the `LikwidPerformanceMonitoringModule`. Setting up, starting and stopping a group of five counters via the LIKWID access daemon takes on average 43.6 μs, for a counter group of 19 the duration increases to 264.8 μs. The overhead therefore strongly depends on the number of counters that are to be used and analysis must be carried out for each group individually. The accuracy of the resulting metrics varies according to whether the metric itself is a rate and depends on the update rates of the underlying counters and even their order in the group.

With respect to simple linear models that allow for prediction of the energy consumption of a single core from per-CPU aggregates as presented in chapter 3.4, the following observation can be made (see Figure 5.2): In realistic scenarios where the frequency of all cores can not be set to the same fixed frequency (and thereby disabling all of the essential energy saving mechanisms in modern CPUs and operating systems) there is no linear relation between the number of active cores and the total CPU power consumption. The Haswell-EP CPUs in CoolMUC-2, for example, have a nominal frequency of 2.6 GHz, but generous power and cooling budgets allow for frequencies up to 3.6 GHz. Depending on the number of active cores this limit decreases as indicated in the illustration. Since CPU frequency is one of the main drivers for energy consumption, in consequence the relation between the number of active cores and the overall power consumption is not lin-

Time required to set up, start and stop a group of 5 counters:	43.6 μs
Time required to set up, start and stop a group of 19 counters:	264.8 μs

Table 5.5: Measured overhead characteristics (averaged values) of the `LikwidPerformanceMonitoringModule`.

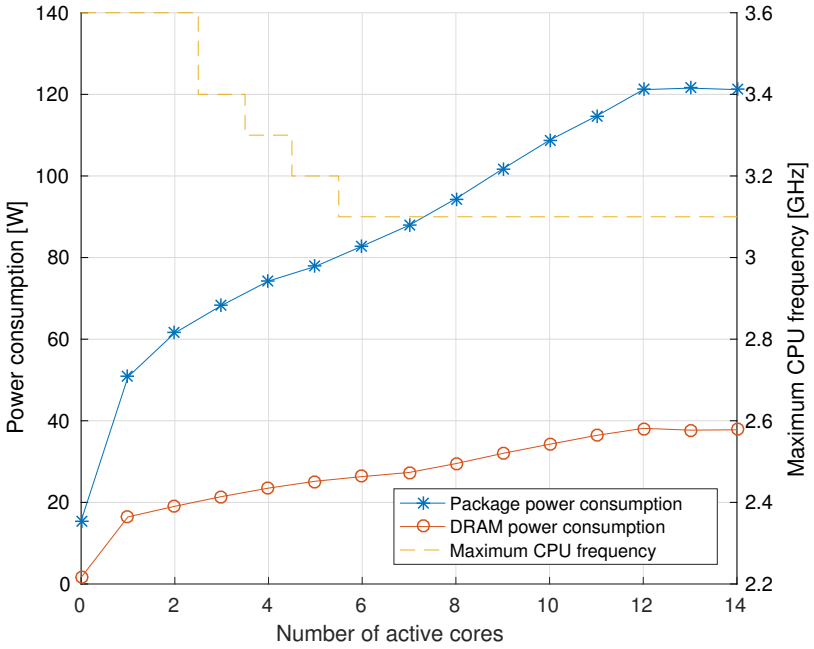


Figure 5.2: Relation between number of active cores and package/DRAM power consumption (RAPL domains PKG and DRAM) for an Intel Xeon E5-2697v3 14-core CPU. The measurements were carried out over the duration of 30s using the tool `likwid-powermeter` on a node of CoolMUC-2. A long-running ExaHyPE simulation was pinned to each active core, all remaining cores were pinned to `sleep`. The frequency of the CPU was not fixed and the default `ondemand` scaling governor was active. The respective frequency response of the Intel Haswell-EP processor model under consideration is indicated as a dashed line on the secondary axis (see [98] for details).

ear, but is rather described by a curve that is monotonically increasing and goes towards saturation (at around 120 W in our case). The underlying power and cooling budgets are inherently specific to the hardware at hand and vary for each individual configuration. It is in theory possible to calibrate sophisticated non-linear models as described in [62] for each individual scenario. Within the scope of this thesis, however, we will refrain from doing so and instead only consider per-CPU energy estimates. This means that for the single-threaded simulation experiments presented in the following we will not give absolute values on the amount of energy that was saved as a result of kernel optimizations since we do not adequately model the overhead of idling cores and shared "uncore" resources. Such absolute numbers, however, would be of little practical use since in real simulation runs all cores are kept busy via shared-memory parallelization and then the only metric of concern is the energy consumption of the complete core.

5.3 The Complete Simulation

Let us begin our discussion on preliminary profiling results of an ExaHyPE simulation as discussed in chapter 5.1 with considerations on the overall runtime. Figure 5.3 illustrates the runtime of the complete simulation (including setup and

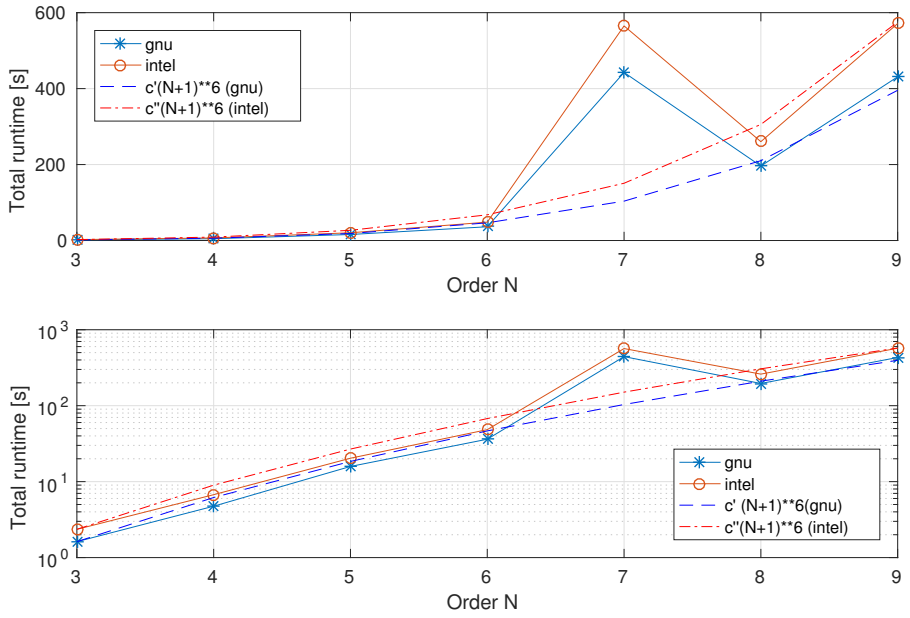


Figure 5.3: Total wall clock runtime of a simulation using the Euler200 model on linear and logarithmic ordinate axes, respectively. Domain $\mathbf{\Omega} = [0.0, 15.0]^3$ and "diffusing Gauss" initial conditions were chosen as described in section 5.1. The simulation has been carried out for $t \in [0.0, 2.0]$ on 9^3 cells resulting in a total of $9^3(N+1)^3$ grid points using the generic, unoptimized kernels.

Order	3	4	5	6	7	8	9
GNU	65.11%	75.23%	81.90%	85.49%	96.89%	89.05%	90.00%
Intel	58.78%	75.71%	83.62%	87.06%	96.11%	87.51%	90.52%

Table 5.6: Ratio between the time spent in the ADER-DG kernels of the simulation loop and the total wall clock time of the standard Euler200 simulation as described in chapter 5.1 for $t \in [0.0, 2.0]$.

teardown) with respect to the polynomial degree N in the range from three to nine. The data has been obtained via the standalone `likwid-perfctr` tool which takes care of timing and pinned execution of the simulation. As indicated by the dashed lines the overall runtime develops as $O((N+1)^6)$. This very well reflects the expected behavior since we obtain an order for each of the three spatial dimensions, one for time, one for the matrix multiplications and one for the overall number of simulation steps which grows linearly due to the time step restriction (see eq. (2.111)). For reasonably smooth input data the number of iterations of the Picard loop in contrast can be assumed to be constant and in our scenario to be almost always one. The overall execution time of the binary created with the GNU compiler is consistently lower compared to that created by the Intel counterpart by a margin between 20 and 30%. The significant jump for order $N = 7$ will be investigated in the case study presented in chapter 5.5.

Table 5.6 lists the ratio between the time spent in the ADER-DG compute kernels of the simulation loop and the overall wall clock time of the standard Euler200 simulation as defined in chapter 5.1. The measurements were carried out separately

Order	3	4	5	6	7	8	9
GNU	2.0	5.1	12.3	25.7	292.6	79.0	131.9
Intel	2.9	7.6	16.7	35.8	371.9	104.7	178.0

Table 5.7: Duration in ms of a complete ADER-DG element update in the Euler200 model.

so that overheads did not artificially increase the total simulation time. The ratios are a measure for the overhead induced by the management of simulation data, in the case of the ExaHyPE engine one could say it measures the "Peano overhead." In general the percentages are similar for both GNU and Intel compiler and the share of time spend in the compute kernels increases with increasing polynomial degree, i.e. the relative impact of the data management overhead decreases. We again observe a jump for polynomial order seven which is to be investigated using the implemented profiling techniques in chapter 5.5.

An important measure discussed in literature to compare the efficiency of an ADER-DG implementation is the time it takes to update a single element. As expected the runtime develops as $O((N+1)^6)$ with respect to the polynomial order of the ansatz functions. For the given set of kernels a value between two and three milliseconds for a third-order update of a three-dimensional element with 200 quantities is measured. Even though these values seem to be very well in line with remarks on typical performance by Michael Dumbser or even slightly better, once again the fact that the profiled kernels are unoptimized, generic and should only be used for rapid prototyping purposes or to generate reference solutions shall be stated. Comparing GNU and Intel compiler we once again see a 20 to 30% performance lead of the former.

5.4 A Kernel-by-kernel Breakdown

Apart from the total element update time the presented implementations of the profiling interface for time measurement can also provide insight on a finer level by allowing for a kernel-by-kernel breakdown. Figure 5.4 illustrates how the ratio between the runtime of individual kernels (generic, unoptimized) and the total element update time develops for polynomial degrees from three to nine. The data has been obtained using the `LikwidTimeMeasurementModule` and is shown exemplarily for the GNU compiler. Since the same illustration for results obtained in conjunction with the Intel compiler is visually indistinguishable we omit it. The kernel denoted as "spaceTimePredictor" implements the iterative method illustrated in equation (2.94) and carries out the subsequent evaluation of source and flux functions. The kernels "stableTimeStepSize", "surfaceIntegral" and "volumeIntegral" implement equations (2.111), (2.107) as well as the combination of equations (2.102) and (2.104), respectively. "Other" denotes the combined share of the kernels "riemannSolver", "solutionUpdate" and "boundaryConditions." The former two implement equations (2.33) and (2.110), respectively, the latter calls a user-defined function that implements the desired boundary conditions. For the purposes of our simulations we use simple copy boundary conditions, i.e. the DOFs of the cells at the surface of the domain are copied to the respective ghost cells that surround it. The figure clearly illustrates the dominance of the space-

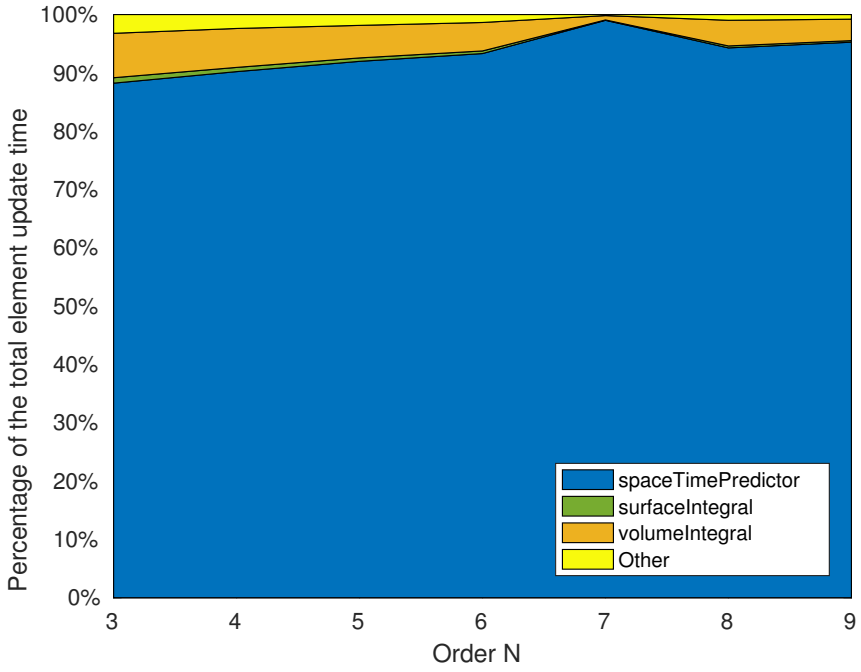


Figure 5.4: Share of the individual ADER-DG kernels (generic, unoptimized) with respect to the total element update time (stacked). The measurements were carried out using the `LikwidTimeMeasurementModule` within the standard Euler200 simulation. Data is shown exemplarily for the GNU compiler; results are visually indistinguishable if shown for the Intel counterpart.

time predictor kernel compared to all other kernels that comprise a complete ADER-DG update. The spurious bump at order seven is also visible and can be attributed to the space-time predictor kernel.

Probably the metric of greatest interest for future work that based on the provided implementations of the generic profiling interface can be obtained directly within ExaHyPE is energy. Figure 5.5 illustrates the relation between runtime and package energy consumption as reported by the on-chip RAPL counters for two selected kernels. For the long-running space-time predictor kernel there is as expected an almost visually indistinguishable linear correspondence between runtime and energy consumption. When considering the shorter "stableTimeStepSize" kernel for polynomial orders less than seven, on the other hand, then no such perfect linear relations ship is visible. This clearly illustrates the limitations of RAPL with respect to temporal resolution: The energy counters are updated about once every millisecond. For order three the space-time predictor compiled with the GNU compiler needs about 1.8 ms per execution, for order nine it needs about 126 ms. The kernel used to compute a stable time step needs about 0.09 ms at order three, 0.83 ms at order seven and 1.5 ms at order nine. Our results clearly show that using the register polling approach presented in chapter 3.4 precise measurements at the order of milliseconds or below are possible, however particularly vigilant validation is of the essence. In conclusion the assumed almost perfect linear relationship between runtime and energy consumption for more or less constant core frequencies and cycles per instruction (CPI) ratios as stated in

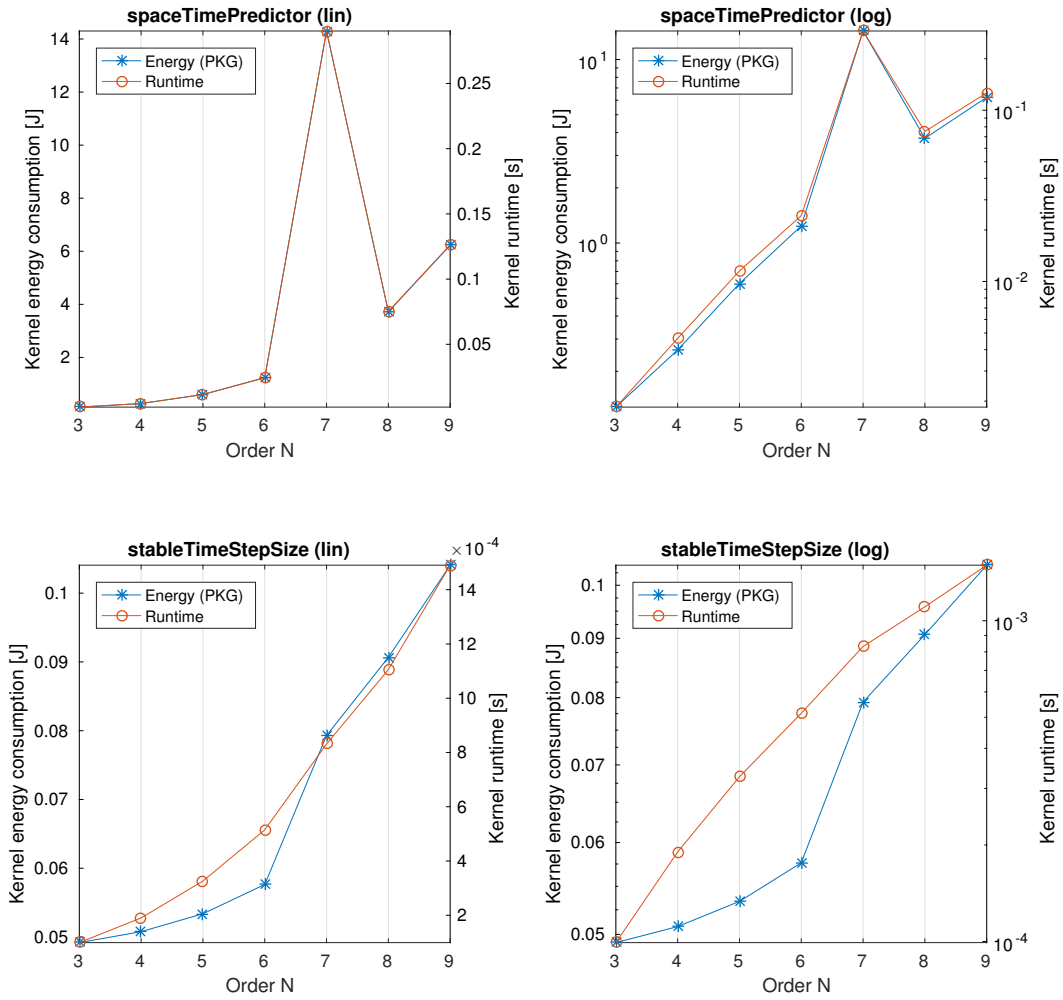


Figure 5.5: Comparison of energy consumption (RAPL domain PKG) and runtime for the kernels "spaceTimePredictor" and "stableTimeStepSize" on both linear and logarithmic ordinate axes. Results above are exemplarily shown for the GNU compiler; results for the Intel compiler show similar characteristics but at slightly longer runtimes and in consequence also higher energy consumptions.

literature (see for example [59, 61, 62, 63]) is clearly visible in the data for long code paths (runtime $\geq$1 ms). For the most parts, however, the ADER-DG kernels in ExaHyPE are at the verge of what can be measured using RAPL counters in current-generation x86 CPUs. This is especially true if models with few variables and optimize kernels are used in a simulation. Even though the implemented polling approach has proven to be invaluable in enabling measurements close to the counter update frequency or below, at this point the only thing that can be done with respect to very short code paths is to express the hope that the hardware performance monitoring units in future generations of micro architectures will provide a higher temporal resolution.

5.5 Case Study: "The seventh-order itch."

In this first case study we would like to illustrate how the embedded profiling facilities in ExaHyPE can not only be used for performance optimization, but also to safe time when analyzing (performance) bugs. Embedded profiling can be used to identify sections in the code that are most likely the cause for suboptimal performance without changing a single line of code. In view of the unexpected behavior at order seven first observed in Figure 5.3, let us therefore once again point our attention to Table 5.7. The data shown in the table depicting the total element update times for varying polynomial order, i.e. the wall clock time spent in the ADER-DG kernels to update one single element, also exhibits the same extended runtime for order seven, irrespective of the employed compiler. We can therefore conclude that the cause of the performance bug at order seven is not to be found inside Peano or any other part of ExaHyPE related to data management, but directly within the ADER-DG kernels. Considering once again Figure 5.4 we can now conclude that a), only the space-time predictor kernel has a sufficient share of the element update time to cause a jump in runtime that is visible at the scale of the complete simulation and b), that indeed for order seven the runtime of said kernel is disproportionately high compared to all other kernels. Figure 5.5 illustrates the latter even more explicitly. Up to this point simply by looking at a per-kernel runtime and energy breakdown we managed to identify the problematic region in the code; all without changing a single line of code or spending time to configure and execute an external profiler.

To identify the cause of the issue let us add more insight in the form of performance counter measurements. Figure 5.6 illustrates selected metrics based on performance counter values for a single execution of the space-time predictor kernel (averaged values). We can again draw two important conclusions, namely a), there is no intrinsic problem with respect to instruction throughput for example due to a compiler bug, since the average number of instructions that are executed per cycle (1/CPI) remains almost constant also for order seven, and b), the additional runtime is simply caused by additional work that has to be done by the CPU: It needs to execute more instructions in total, more particularly time-consuming vectorized floating-point operations (AVX) and needs to both read and write more data from and to main memory. The additional metrics give us even more confidence that the original observation of prolonged runtime for order-seven simulations is not a result of measurement fluctuation, but has a fundamental cause inside the application code.

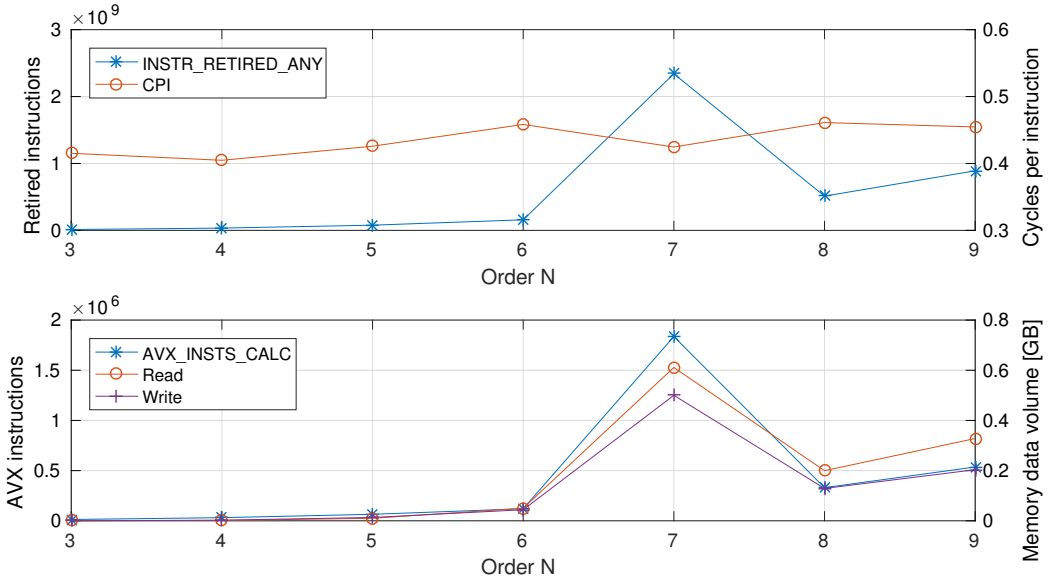


Figure 5.6: Selected performance counter metrics (averaged) for a single execution of the space-time predictor kernel. Results are exemplarily shown for the GNU compiler; for the Intel compiler similar characteristics can be observed.

Since we now know that for order seven we execute significantly more instructions we could take a look at the size of simulation binaries for different orders only to observe that there are no major differences. Sidenote: This is no surprise given the fact that even though the simulation order is a compile-time constant in ExaHyPE (leitmotif: "Compile often instead of making performance compromises"), it is not directly available inside the kernel compilation units and careful assessment of the linked assembly revealed that link-time optimization available in both the GNU and the Intel compiler was not able to optimize the kernel accordingly.

The only alternative that is left to explain the increased count of executed instructions is that parts of the space-time predictor kernel get executed multiple times. Remembering the fixed-point iteration ("Picard loop") described in eq. (2.94) we may suspect that for order seven the number of iterations is on average significantly higher than for all other orders. Up to this point not a single line of code needed to be changed. Figure 5.7 illustrates the relative frequency of the number of iterations in the Picard loop and confirms the suspicion. For orders other than seven and the smooth input data in our scenario almost always a single iteration is enough to achieve the prescribed upper bound for the iteration update of less than 10^{-14} in the squared norm (see eq. (2.95)). This is obviously not the case for order seven. To obtain the loop counts for the first time in this case study manual changes to the respective section in the code were necessary.

The number of iterations that are necessary in the fixed-point iteration in the space-time predictor obviously depends on the input data. As of writing the underlying root cause for the described behavior is under investigation but not yet completely clear. No matter what the issue turns out to be in the end[2], the point this case study is supposed to underscore is a different one: Throughout the section we were able

[2]The author's best guess is that is a simple bug in the routines that set the initial conditions and/or treat the boundaries, i.e. the bug is in the author's application code rather than in the

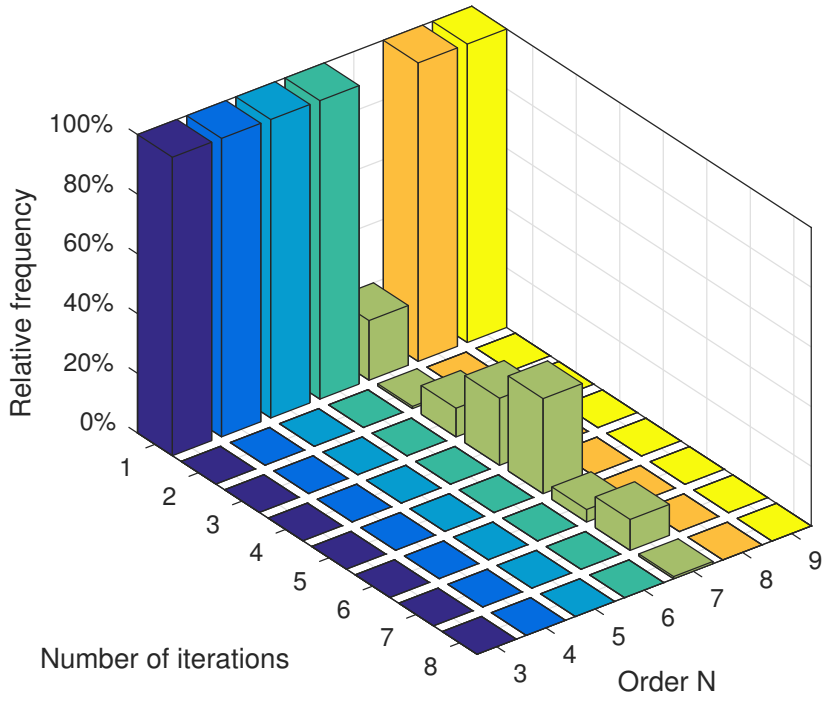


Figure 5.7: Relative frequencies of the number of iterations in the "Picard loop" fixed-point iteration within the space-time predictor. An Euler200 simulation for $t \in [0.0, 2.0]$ has been carried out as described in chapter 5.1. Results above are exemplarily shown for the GNU compiler; results for the Intel compiler show similar characteristics.

to illustrate how the generic profiling infrastructure embedded into ExaHyPE can be used to iteratively narrow down the amount of code that needs to debugged manually saving a lot of time in the overall process since not a single line of code needed to be changed. Even though the underlying cause of the issue under investigation is not yet known, the insight that the space-time predictor kernel or the input to it are good places to start manual efforts has already proven extremely valuable.

5.6 Case Study: "Finding for the Juiciest Fruits."

In this case study we show how the performance metrics provided by the implemented ExaHyPE profilers can be used to guide efforts to optimizing the performance of the ADER-DG kernels. As stated multiple times throughout this thesis performance is not a primary concern for the generic ADER-DG kernels that are used for illustration throughout this chapter; correctness, readability and ease of use are by far more important. For the sake of this case study, however, we will nevertheless act as if performance of the generic kernels was an issue and we will illustrate the necessary steps on how information obtained through profiling can be of great value in the optimization process. The general leitmotif of this section is "Measure first, then optimize." The proposed measures for performance optimization are chosen such that they can be implemented with little to no effort and

generic ExaHyPE kernels. Since nevertheless the kernels were also written by the author it is almost certain who is to take the blame...

so that by no means they impose additional restrictions on the flexibility of the code. Even though this section is definitely focused on the process rather than the results, nevertheless the obtained performance benefits are remarkable. The changes presented throughout the case study have therefore been merged into ExaHyPE. After all one does not simply skip the "low hanging fruits."

To start our optimization efforts we first need to understand which parts of the code actually dominate with respect to runtime and energy consumption. Only in these parts of the code can our optimizations generate real impact on the overall simulation. Referring back to Table 5.6 it becomes obvious that the ADER-DG kernels are indeed a good place to start since the great majority of the overall runtime (and therefore also energy) is spend in the kernels. Continuing with Figure 5.4 we furthermore see that we should without doubt begin our work with the space-time predictor kernel since about 90% of the element update time can be attributed to it.

We therefore continue by collecting various performance counter metrics for the space-time predictor kernel and as reference also for the other kernels using the `LikwidPerformanceMonitoringModule`. Figure 5.8 depicts four such metrics this time for the Intel compiler[3]. In the figure the metrics are plotted for the unoptimized space-time predictor kernel as well as exemplarily for the surface integral kernel as a typical reference for other unoptimized kernels. For the sake of compactness we furthermore forestall already the results of the optimized space-time predictor kernel. Considering the instruction cache miss rate of the space-time predictor kernel we see that it is relatively high compared to the surface integral kernel as an exemplary reference as well as when compared to other kernels omitted in the figure. Instruction cache misses typically happen when there is a great number of branch instructions in a code and in particular when the branching pattern is complicated or even unpredictable so that branch prediction does not work reliably. The second metric, namely the average number of instructions per branch, is lower than for most other kernels. The metric is simply defined as the ratio between the number of branching instructions and the total number of instructions that was executed. Both of the described phenomena, a high instruction cache miss ratio and a low number of instructions per branch, can lead to an extensive number of stalls of the superscalar execution pipeline in modern x86 processors in turn resulting in low overall throughput and suboptimal performance. To tackle this problem we will therefore in the following try to identify ways to eliminate unnecessary branches or more precisely try to give the compiler a better chance to optimize away branches that can be evaluated at compile time or are in other ways unnecessary.

Algorithm 1 gives a pseudocode description of the Picard fixed-point iteration as the computationally most expensive part of the space-time predictor kernel. The only kind of branching that is explicitly visible in the C++ code is hidden in the loop test condition of the "for" statements. Since all of the loop ranges are in principle available at compile-time, loop transformations such as loop fusion, loop fission or loop splitting can be used by the compiler to maximize expected performance with respect to its built-in performance model. Another layer of branching is visible at assembly level: When compilers try to vectorize a floating-point operation over arrays of unknown size a common pattern can be observed:

[3]Simply for the sake of presenting results for the Intel compiler at least once.

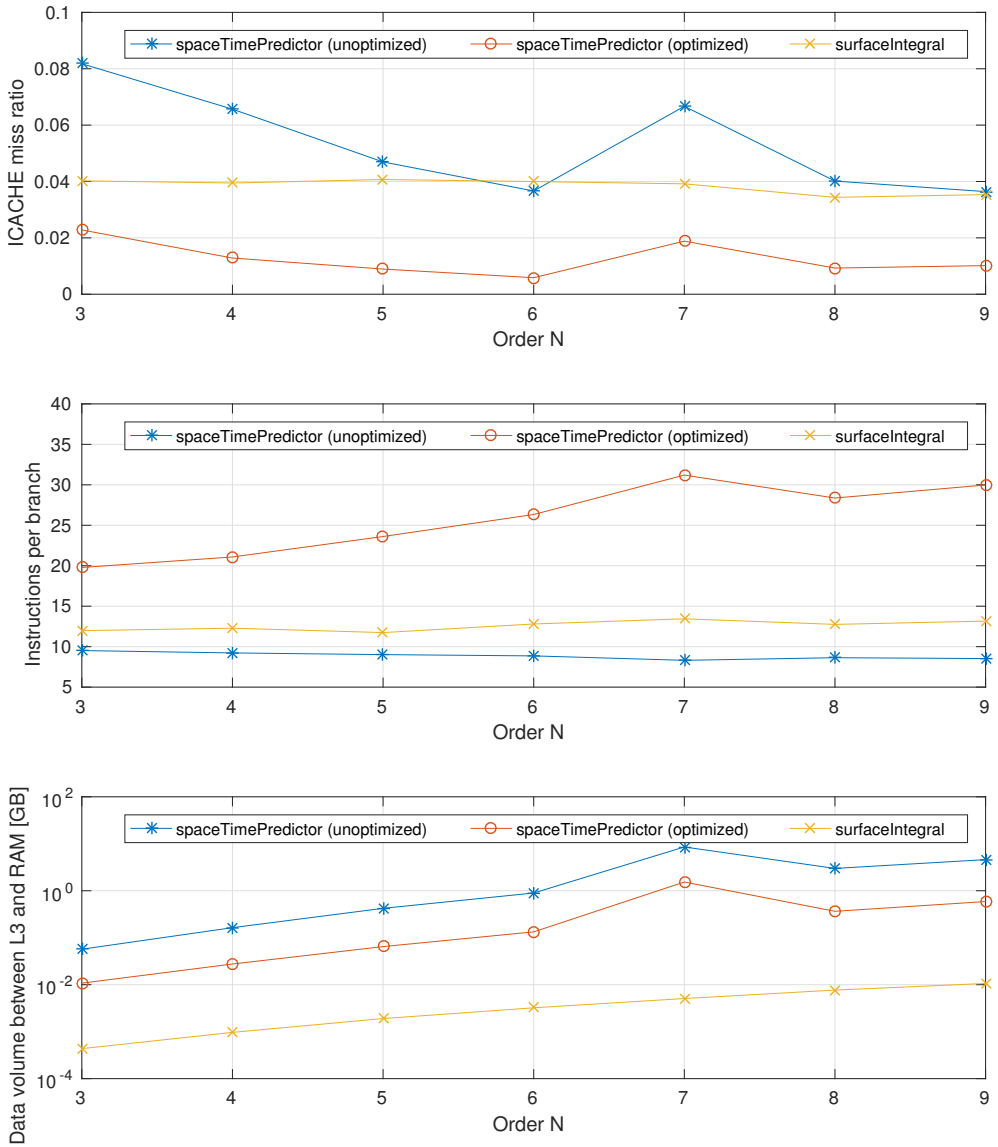


Figure 5.8: Selected performance counter metrics (averaged) for a single execution of the surface integral kernel as well as for both the unoptimized and the optimized space-time predictor kernel. Results above are exemplarily shown for the Intel compiler; results for the GNU compiler show similar characteristics.

Algorithm 1: Original implementation of the Picard iteration (eq. (2.94)). Ordering of indices illustrates linearized memory layout in row-major form.

Input: Spatial DOFs of the discrete solution: $\hat{\boldsymbol{u}}^{K,i}$

Parameter: time step size: Δt_i, dimension of the cell: $\Delta \boldsymbol{x}^K$,
order in space and time: N, number of quantities: V

Output: Space-time DOFs of the space-time predictor: $\hat{\boldsymbol{q}}^{K,i}$

Initialization: Set initial guess (eq. (2.81)): $\left[\hat{\boldsymbol{q}}^{K,i,\text{new}}\right]_{\boldsymbol{n}lv} := \left[\hat{\boldsymbol{u}}^{K,i}\right]_{\boldsymbol{n}v}$.

Evaluate term S-III (eq. (2.89)): $[\hat{\boldsymbol{r}}_0]_{l\boldsymbol{n}v} := \hat{\omega}_{\boldsymbol{n}} [\boldsymbol{l}]_l \left[\hat{\boldsymbol{u}}^{K,i}\right]_{\boldsymbol{n}v}$.

Set $r := 0$, $R := 2(N+1)$, $\Delta^2 := \infty, \varepsilon^2 = 10^{-14}$.

while $r < R$ **and** $\Delta^2 < \varepsilon^2$ **do**

Copy result of previous iteration: $\left[\hat{\boldsymbol{q}}^{K,i,\text{old}}\right]_{\boldsymbol{n}lv} := \left[\hat{\boldsymbol{q}}^{K,i,\text{new}}\right]_{\boldsymbol{n}lv}$.

Copy contribution of S-III to right-hand side: $[\hat{\boldsymbol{r}}]_{l\boldsymbol{n}v} := [\hat{\boldsymbol{r}}_0]_{l\boldsymbol{n}v}$.

for $l \in \mathcal{N}$ **do** *// time DOFs*

for $\boldsymbol{n} \in \boldsymbol{\mathcal{N}}$ **do** *// spatial DOFs*

Evalute fluxes (eq. (2.91)): $\left[\hat{\boldsymbol{F}}\right]_{l\boldsymbol{n}dv} := \left[\boldsymbol{F}\left(\left[\hat{\boldsymbol{q}}^{K,i,\text{old}}\right]_{\boldsymbol{n}l}\right)\right]_{dv}$.

Evaluate sources (eq. (2.93)): $[\hat{\boldsymbol{s}}]_{l\boldsymbol{n}v} := \left[\boldsymbol{s}\left(\left[\hat{\boldsymbol{q}}^{K,i,\text{old}}\right]_{\boldsymbol{n}l}\right)\right]_{v}$.

end

for $\boldsymbol{n} \in \boldsymbol{\mathcal{N}}$ **do** *// spatial DOFs*

Evaluate S-IV and subtract from right-hand side (eq. (2.91)):

$$[\hat{\boldsymbol{r}}]_{l\boldsymbol{n}v} :\overline{=} J_{\mathcal{T}_i} \hat{\omega}_l \sum_{d \in \mathcal{D}} \left(\frac{1}{\left[\Delta \boldsymbol{x}^K\right]_d} \hat{\omega}_{\boldsymbol{n}_{0:d-1,d+1:D-1}} \cdots \right.$$
$$\left. \cdots \sum_{n'_d \in \mathcal{N}} \left([\boldsymbol{K}]_{\boldsymbol{n}_d n'_d} \left[\hat{\boldsymbol{F}}\right]_{l[\boldsymbol{n}_{0:d-1}, n'_d, \boldsymbol{n}_{d+1:D+1}]dv} \right) \right).$$

Evaluate S-V and add to right-hand side (eq. (2.80)):
$[\hat{\boldsymbol{r}}]_{l\boldsymbol{n}v} :\overset{+}{=} J_{\mathcal{T}_i} \hat{\omega}_{\boldsymbol{n}} \hat{\omega}_l [\hat{\boldsymbol{s}}]_{l\boldsymbol{n}v}$.

end

for $\boldsymbol{n} \in \boldsymbol{\mathcal{N}}, l \in \mathcal{N}$ **do** *// space-time DOFs*

Multiply with inverse iteration matrix: $\left[\hat{\boldsymbol{q}}^{K,i,\text{new}}\right]_{\boldsymbol{n}lv} := \frac{1}{\hat{\omega}_{\boldsymbol{\alpha}}} \left[\tilde{\boldsymbol{K}}\right]_{ll'} [\boldsymbol{r}]_{l'\boldsymbol{n}v}$.

end

Update squared element-wise residual (eq. (2.95)):
$\Delta^2 := \sum_{\boldsymbol{n} \in \boldsymbol{\mathcal{N}}} \sum_{l \in \mathcal{N}} \sum_{v \in \mathcal{V}} \left(\left[\hat{\boldsymbol{q}}^{K,i,\text{new}}\right]_{\boldsymbol{n}lv} - \left[\hat{\boldsymbol{q}}^{K,i,\text{old}}\right]_{\boldsymbol{n}lv} \right)^2$.

end

end

a), the compiler needs to add checks to make sure that the current position in the operand arrays do not overlap (in most scientific codes this would not make much sense with regard to the implemented scheme, but since the C/C++ standard allows for pointer aliasing the compiler in general needs to assume it is there), and b), separate code is created to treat several values at once using a vector instruction (e.g. four "doubles" for AVX-256) and to treat a single value. Now when the operation is executed on an array there is a branching instruction to check if there are still enough values available to execute the vectorized code or whether the values of a possible remainder need to be treated separately. Even though branch prediction limits the impact of the latter it is unnecessary overhead that can be avoided if the loop count is known to the compiler.

As stated already in chapter 5.5 the number of variables as well as the order of the polynomial ansatz functions is fixed once the ExaHyPE toolkit has been executed. The two numbers are enough to fix the range of all loops in the Picard iteration at compile time. A problem we observed, however, is that since the constants are not visible in the compilation unit of the kernel and since link-time optimization is not yet sophisticated enough, no optimization can happen. In an effort to force the compiler to be aware of the fixed loop ranges we therefore promote the function parameters V and N denoting the number of variables and the polynomial order in space and time, respectively, to kernel parameters as illustrated in Algorithm 2. In this way we enforce that kernel code can only be created in compilation units in which the two constants are directly available. We furthermore unwind all of the loops of the degrees of freedom to ensure that only so-called perfect loop nests are left, i.e. there are no statements in any but the innermost loop of the loop nest. In this way the compiler can optimally use its heuristics to "understand" the structure of the operation to then decide which loop transformations should be applied to maximize expected performance. As hinted towards above we also add the keyword "__restrict__"[4] where appropriate to input and output arrays. This indicates to the compiler that the programmer ensures that there is no pointer aliasing, i.e. no legal access to the same value in memory can be made through two or more different pointers. The compiler may try to proof that there is no aliasing for a specific combination of pointers, but in general this procedure is based on heuristics and pattern matching and once a boundary between compilation units is met these techniques becomes unreliable. It is conventional wisdom that adding the "__restrict__" keyword wherever applicable can close a possible performance gap between otherwise equivalent C/C++ and FORTRAN code.

Now considering the third metric depicted in Figure 5.8, we see that the amount of data that the space-time predictor kernel transfers between main memory and the L3 cache as the lowest level of on-chip caching is almost two orders of magnitude larger than for all other kernels (for brevity only the surface-integral kernel is shown in the plot). In view of the severe consequences on application performance of the so-called memory gap, i.e. the growing discrepancy between CPU and DRAM performance, all unnecessary data movement operations should be avoided, if only to reduce the pressure on the various on-chip caches that in general are to be regarded as scarce resources. Considering Algorithm 1 there are two instances where data is accessed unnecessarily: In the very beginning of the itera-

[4] In C, the keyword "restrict" is part of the language standard. In C++, however, "__restrict__" is a compiler-specific feature available in this form with the GNU, Intel and Clang compilers.

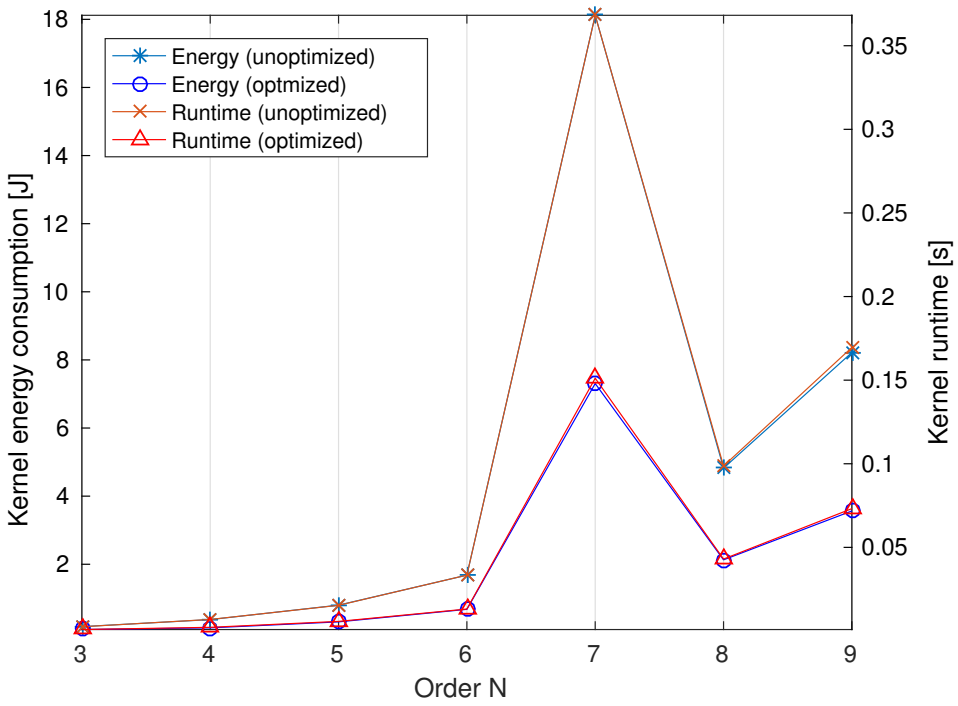


Figure 5.9: Comparison of energy consumption (RAPL domain PKG) and runtime between the optimized and the unoptimized version of the space-time predictor kernel. In the optimized version both energy consumption and runtime could be reduced by a factor ranging from 2.0 to 2.8 and 2.3 to 3.2, respectively. Results above are exemplarily shown for the Intel compiler; results for the GNU compiler show similar characteristics but at slightly shorter runtimes and in consequence also lower energy consumptions.

tion the degrees of freedom from the previous execution are copied into an array that holds the "old" solution. The program then subsequently writes the "new" solution into the array that acted as the source for the copy operation. This common pattern can be solved more efficiently by simply swapping pointers to the two arrays as illustrated in Algorithm 2 to avoid unnecessary copying. A second observation that reveals itself after careful consideration of the equations embedded into the algorithm is the fact that the Gauss-Legendre integration prefactor $\hat{\omega}_\alpha$ can be pulled out of every single term that contributes to the final computation of the new degrees of freedom. In this way we can avoid the strided data accesses to compute $\hat{\omega}_\alpha$. It is a reasonable assumption that the relatively small array of Gauss-Legendre weights will remain in on-chip caches either way, but if we optimize away the factor we can at least reduce the pressure on the respective caches.

The complete optimized Picard fixed-point iteration is listed in Algorithm 2. Figure 5.8 illustrates the improvement with respect to the depicted metrics enabled by the discussed optimizations: The instruction cache miss ratio dropped by a factor of about four, the average branch length more than doubled and the data transfer volume between L3 cache and DRAM (bidirectional) also decreased significantly, probably due to clever loop fusion done by the compiler. The improved metrics give rise to the assumption that runtime and energy consumption of the space-

Algorithm 2: Modified implementation of the Picard iteration (eq. (2.94)). Ordering of indices illustrates linearized memory layout in row-major form.

Template parameter: order in space and time: N, number of quantities: V

Input: Spatial DOFs of the discrete solution: $\hat{\boldsymbol{u}}^{K,i}$ (__restrict__)

Parameter: time step size: Δt_i, dimension of the cell: $\Delta \boldsymbol{x}^K$

Output: Space-time DOFs of the space-time predictor: $\hat{\boldsymbol{q}}^{K,i}$

Initialization: Set initial guess (eq. (2.81)): $\left[\hat{\boldsymbol{q}}^{K,i,\text{new}}\right]_{\boldsymbol{n}lv} := \left[\hat{\boldsymbol{u}}^{K,i}\right]_{\boldsymbol{n}v}$.

Evaluate term S-III (eq. (2.89)): $[\hat{\boldsymbol{r}}_0]_{l\boldsymbol{n}v} := [\boldsymbol{l}]_l \left[\hat{\boldsymbol{u}}^{K,i}\right]_{\boldsymbol{n}v}$.

Set $r := 0$, $R := 2(N+1)$, $\Delta^2 := \infty, \varepsilon^2 = 10^{-14}$.

while $r < R$ **and** $\Delta^2 < \varepsilon^2$ **do**

Swap $\left[\hat{\boldsymbol{q}}^{K,i,\text{old}}\right]_{\boldsymbol{n}lv}$ and $\left[\hat{\boldsymbol{q}}^{K,i,\text{new}}\right]_{\boldsymbol{n}lv}$.

Copy contribution of S-III to right-hand side: $[\hat{\boldsymbol{r}}]_{l\boldsymbol{n}v} := [\hat{\boldsymbol{r}}_0]_{l\boldsymbol{n}v}$.

for $l \in \mathcal{N}, \boldsymbol{n} \in \boldsymbol{\mathcal{N}}$ **do** // *space-time DOFs*

Evalute fluxes (eq. (2.91)): $\left[\hat{\boldsymbol{F}}\right]_{l\boldsymbol{n}dv} := \left[\boldsymbol{F}\left(\left[\hat{\boldsymbol{q}}^{K,i,\text{old}}\right]_{\boldsymbol{n}l}\right)\right]_{dv}$.

Evaluate sources (eq. (2.93)): $[\hat{\boldsymbol{s}}]_{l\boldsymbol{n}v} := \left[\boldsymbol{s}\left(\left[\hat{\boldsymbol{q}}^{K,i,\text{old}}\right]_{\boldsymbol{n}l}\right)\right]_{v}$.

end

for $l \in \mathcal{N}, \boldsymbol{n} \in \boldsymbol{\mathcal{N}}$ **do** // *space-time DOFs*

Evaluate S-IV and add to right-hand side (eq. (2.91)):

$$[\hat{\boldsymbol{r}}]_{l\boldsymbol{n}v} :\overline{=} J_{\mathcal{T}_i}\hat{\omega}_l \sum_{d\in\mathcal{D}} \left(\frac{1}{\left[\Delta \boldsymbol{x}^K\right]_d} \frac{1}{\hat{\omega}_{\boldsymbol{n}_d}} \sum_{n'_d \in \mathcal{N}} \left([\boldsymbol{K}]_{\boldsymbol{n}_d n'_d} \left[\hat{\boldsymbol{F}}\right]_{l[\boldsymbol{n}_{0:d-1}, n'_d, \boldsymbol{n}_{d+1:D+1}]dv} \right) \right).$$

end

for $l \in \mathcal{N}, \boldsymbol{n} \in \boldsymbol{\mathcal{N}}$ **do** // *space-time DOFs*

Evaluate S-V and add to right-hand side (eq. (2.80)):

$[\hat{\boldsymbol{r}}]_{l\boldsymbol{n}v} :\overset{+}{=} J_{\mathcal{T}_i}\hat{\omega}_l\, [\hat{\boldsymbol{s}}]_{l\boldsymbol{n}v}$.

end

for $l \in \mathcal{N}, \boldsymbol{n} \in \boldsymbol{\mathcal{N}}$ **do** // *space-time DOFs*

Multiply with inverse iteration matrix: $\left[\hat{\boldsymbol{q}}^{K,i,\text{new}}\right]_{\boldsymbol{n}lv} := \left[\tilde{\boldsymbol{K}}\right]_{ll'} [\boldsymbol{r}]_{l'\boldsymbol{n}v}$.

end

Update squared element-wise residual (eq. (2.95)):

$\Delta^2 := \sum_{\boldsymbol{n}\in\boldsymbol{\mathcal{N}}} \sum_{l\in\mathcal{N}} \sum_{v\in\mathcal{V}} \left(\left[\hat{\boldsymbol{q}}^{K,i,\text{new}}\right]_{\boldsymbol{n}lv} - \left[\hat{\boldsymbol{q}}^{K,i,\text{old}}\right]_{\boldsymbol{n}lv} \right)^2$.

end

time predictor could be reduced. Figure 5.9 indeed reveals remarkable improvements: Throughout the complete range of orders the data shows that both energy consumption and runtime could be reduced by a factor ranging from 2.0 to 2.8 and 2.3 to 3.2, respectively. Nevertheless we would like to emphasize once again that the case study is not about the actual performance improvements, but rather focuses on the way how the improvements have been achieved in a systematic way. We measured first, then optimized! For an extensive review on best practices for performance engineering assisted by hardware measurements and common patterns that appear in the process the reader is referred to [30]. Carla Guillén Carías from LRZ gives an extensive list of actionable recipes on how to turn measurement insights into opportunities for performance optimization in [50].

5.7 Concluding Remarks

The presented case studies clearly illustrate how the proposed profiling interface and the various implementations thereof can be used to guide and track progress in performance and energy-efficiency optimization efforts as well as in the analysis of a broad range of issues. The key advantage of the available tooling is that no manual changes to the code are needed other than activating the desired profiler in the configuration file. It is therefore a great way to safe time and allows developers and users to really apply the motto "measure first, then optimize" in practice. With respect to energy measurements a greater temporal resolution of the available hardware measurement units would be desirable. Future work in this direction should include an online energy model that would allow for dynamic frequency tuning from within ExaHyPE. A first step in this direction would be to run analytic benchmark scenarios where adaptive mesh refinement is employed to meet a fixed error bound. In this way a realistic real-world optimum with respect to polynomial degree and frequency can be found for the given scenario.

Chapter 6

Conclusion and Outlook

In this thesis a generic profiling infrastructure for the hyperbolic PDE solver engine ExaHyPE has been presented. In view of the great importance of hyperbolic balance laws in practice and the great computational challenges that is posed by large-scale simulations of models based on the latter, a complete state of the art ADER-DG method with robust a posteriori subcell limiting has been derived for an arbitrary number of spatial dimensions and physical quantities. Against the background of the challenges induced by exascale high performance computing especially with respect to overall energy consumptions the theory behind hardware performance monitoring in x86 has been discussed. Starting from major trends and challenges in contemporary and future Scientific Computing the vision of the ExaHyPE project has been illustrated and an extensive overview on current and future work within the scope of the project has been given. The notion of how ExaHyPE is supposed to be a strong software side answer towards the challenges of exascale computing has been justified extensively and the argument has been made that the use of frameworks and engines is necessary to manage the increasing complexity in HPC. Starting from standard ExaHyPE use-cases the motivation behind the introduction and the most important design goals of a generic profiling infrastructure has been illustrated. Several profiler implementations based on existing libraries and interfaces for performance analysis have been implemented within the scope of this thesis. They are discussed in great detail and benefits as well as limitations especially with respect to accuracy and temporal resolution are analyzed for each of them. Preliminary profiling results are given to illustrate the employed techniques. Based on the generic ADER-DG kernels of ExaHyPE two case studies are presented: In the first one profiling is used to generate valuable insights for the analysis of an issue in a simulation without modifying a single line of code. The second case study illustrates how profiling information is essential in identifying parts of the simulation that are worth optimizing and in guiding systematic metrics-driven performance engineering. The leitmotif of this approach is "measure first, then optimize."

The proposed profiling infrastructure is simple and generic enough to foster future extension also in directions other than classical performance profiling. The ultimate goal must be to have the necessary tooling to fully understand the phenomena related to runtime performance and energy consumption throughout the complete exascale system. In the short term efforts towards extending the set of supported profiling solutions should be carried out to provide these important

capabilities on a wider range of platforms. Experiments such as finding the optimum parameters with respect to polynomial order, processors frequency and adaptive mesh refinement seem to be an interesting area for future research and an essential step towards fully dynamic performance and power consumption optimization from within the engine. Once all of these components of ExaHyPE are in place the engine is without doubt going to make important contributions towards enabling a better understanding of phenomena in various areas of research that will ultimately be of benefit for all of society. It is well-documented that applications made possible by HPC have generated insight that could not have been obtained otherwise and this will hold true to an even greater extent in the exascale era. The availability of sublime profiling capabilities in ExaHyPE will be essential in reaching these ambitious goals.

Chapter 7

Acknowledgement

First of all sincere gratitude goes to my supervisor Dr. Vasco Varduhn. The contribution of his helpful suggestions, invariably friendly support and the stimulating discussions with him to this thesis can not be overestimated. In his role as the scientific coordinator of the ExaHyPE project I highly appreciate his openness towards my ideas and concerns and his subsequent thoughtful consideration of every single one of them. Mr. Varduhn has been a reliable and patient mentor from early on in my studies and has catalyzed the development of my passion for HPC and software engineering.

I would like to express my very great appreciation to both Prof. Michael Bader at TUM and Dr. Tobias Weinzierl, Lecturer at Durham University, for giving me the opportunity to write this thesis as part of the ExaHyPE project and for their courtesy to act as examiners of the latter. Mr. Weinzierl has furthermore been an outstanding host at Durham and I would like to thank him for his almost instant answers to my numerous questions and his appreciation of my ideas.

Advice and notes provided by Prof. Micheal Dumbser (University of Trento), Dominic Etienne Charrier (Durham University) and Angelika Schwarz (TUM, Umeå University) have proven invaluable in fostering my understanding of the theoretical context of the project. Thank you for all your time and effort. Last but by no means least assistance provided by Jean-Matthieu Gallard (TUM), Sven Köppel (FIAS), Dr. Kenneth Duru (LMU), Dr. Carmen Navarrete, Dr. Wolfram Hesse, Juan Pancorbo Armada (all LRZ) and Thomas Röhl (RRZE) is greatly appreciated.

My best wished go to the whole ExaHyPE team. All the best for the next steps in the project!

This thesis is dedicated to a good friend of mine. She knows why.

Appendix A

Computation of the Discrete ADER-DG Operators

In [1]:
```
import numpy as np
np.set_printoptions(precision=3);
import sympy as sp
sp.init_printing(use_latex=True);
x = sp.symbols("x");

## Settings
N = 4;          # Degree of the ADER-DG scheme in space and time
N_S = 2*N + 1;  # Number of subcell averages on the fine grid
##
```

Legendre Polynomials

$P_0(x) = 1, P_1(x) = x, P_{n+1}(x) = \frac{1}{n+1}[(2n+1)xP_n(x) - nP_{n-1}(x)]$

In [2]:
```
def NextLegendrePolynomial(n, P_, P__):
    P = ((2*n + 1) * x * P_ - n * P__) / (n+1);
    return P.simplify();

P = []; P.append(1); P.append(x);

for i in range(1, N+1):
    P.append(NextLegendrePolynomial(i, P[i], P[i-1]));

P = P[N+1];
P
```

Out[2]: $\frac{x}{8}\left(63x^4 - 70x^2 + 15\right)$

Gauss-Legendre nodes $\tilde{\xi}$

In [3]:
```
xi_tilde_sym = sorted(sp.solve(P, x));
xi_tilde_sym
```

Out[3]: $\left[-\sqrt{\frac{2\sqrt{70}}{63} + \frac{5}{9}}, \quad -\sqrt{-\frac{2\sqrt{70}}{63} + \frac{5}{9}}, \quad 0, \quad \sqrt{-\frac{2\sqrt{70}}{63} + \frac{5}{9}}, \quad \sqrt{\frac{2\sqrt{70}}{63} + \frac{5}{9}}\right]$

Gauss-Legendre nodes $\hat{\xi}$ on $[0, 1]$

In [4]:
```
xi_hat_sym = [((xi+1) / 2).simplify() for xi in xi_tilde_sym];
xi_hat = np.array([xi.evalf() for xi in xi_hat_sym]).astype(np.float64);
print xi_hat;
```

```
[ 0.047  0.231  0.5    0.769  0.953]
```

Gauss-Legendre weights $\hat{\omega}$ on $[0, 1]$

In [5]:
```
A = np.matrix([xi_hat**i for i in range(0, N+1)]);
b = np.array([sp.integrate(x**i, (x, (0, 1))).evalf()
              for i in range(0, N+1)]).astype(np.float64);

omega_hat = np.linalg.solve(A, b);
print omega_hat;
```

```
[ 0.118  0.239  0.284  0.239  0.118]
```

Lagrange interpolation polynomials L_i on $[0,1]$ with nodes $\hat{\boldsymbol{\xi}}$

$L_i(\xi) = \prod_{j \neq i} \frac{\xi - \hat{\boldsymbol{\xi}}_j}{\hat{\boldsymbol{\xi}}_i - \hat{\boldsymbol{\xi}}_j}, i = 0, \ldots, N$

In [6]:
```
def L(i, xi_hat):
    xi_skip = xi_hat[:i] + xi_hat[i+1:];
    numerator = sp.prod([x - xi for xi in xi_skip])
    denominator = sp.prod([xi_hat[i] - xi for xi in xi_skip])
    return numerator / denominator

psi = [sp.lambdify(x, L(i, xi_hat_sym)) for i in range(N+1)]
```

In [7]:
```
def dL(i, xi_hat):
    return sp.diff(L(i, xi_hat))

dpsi = [sp.lambdify(x, dL(i, xi_hat_sym)) for i in range(N+1)]
```

Left Reference Element Flux Operator $\boldsymbol{l}$

In [8]:
```
l = np.array([psi[i](0.0) for i in range(0, N+1)]);
print l
```

```
[ 1.551 -0.893  0.533 -0.268  0.076]
```

Right Reference Element Flux Operator $\boldsymbol{r}$

In [9]:
```
r = np.array([psi[i](1.0) for i in range(0, N+1)]);
print r
```

```
[ 0.076 -0.268  0.533 -0.893  1.551]
```

Right Reference Element Mass Operator $\boldsymbol{R}$

In [10]:
```
R = np.fromfunction(np.vectorize(
        lambda i, j: psi[i](1.0) * psi[j](1.0)), (N+1, N+1), dtype=int);
print R
```

```
[[ 0.006 -0.02   0.041 -0.068  0.118]
 [-0.02   0.072 -0.143  0.239 -0.416]
 [ 0.041 -0.143  0.284 -0.476  0.827]
 [-0.068  0.239 -0.476  0.798 -1.386]
 [ 0.118 -0.416  0.827 -1.386  2.407]]
```

Reference Element Stiffness Operator $\boldsymbol{K}$

In [11]:
```
K = np.fromfunction(np.vectorize(
        lambda i, j: omega_hat[j] * dpsi[i](xi_hat[j])), (N+1, N+1),
                    dtype=int);
print K
```

```
[[-1.201 -0.46   0.171 -0.117  0.131]
 [ 1.825 -0.363 -0.817  0.444 -0.464]
 [-0.958  1.15   0.    -1.15   0.958]
 [ 0.464 -0.444  0.817  0.363 -1.825]
 [-0.131  0.117 -0.171  0.46   1.201]]
```

Iteration Matrix $\tilde{K}$

In [12]:
```
Ktilde = np.linalg.inv(R - K);
print Ktilde
```

```
[[ 0.53  -0.124  0.086 -0.066  0.044]
 [ 1.039  0.559 -0.151  0.102 -0.066]
 [ 1.007  1.022  0.57  -0.151  0.086]
 [ 0.981  1.016  1.022  0.559 -0.124]
 [ 1.017  0.981  1.007  1.039  0.53 ]]
```

Projection Operator P

In [13]:
```
P = np.fromfunction(np.vectorize(
        lambda i, j:
        sum([omega_hat[k] *
             psi[j](1.0/N_S * i + 1.0/N_S * xi_hat[k])
             for k in range(0, N+1)])), (N_S, N+1), dtype=int);
print P
```

```
[[  9.472e-01   5.620e-02  -2.341e-03  -1.894e-03   8.191e-04]
 [  2.055e-01   9.434e-01  -2.174e-01   9.415e-02  -2.558e-02]
 [ -5.363e-02   8.698e-01   2.435e-01  -7.997e-02   2.030e-02]
 [ -6.623e-02   4.256e-01   7.658e-01  -1.644e-01   3.913e-02]
 [ -1.333e-03   1.087e-02   9.809e-01   1.087e-02  -1.333e-03]
 [  3.913e-02  -1.644e-01   7.658e-01   4.256e-01  -6.623e-02]
 [  2.030e-02  -7.997e-02   2.435e-01   8.698e-01  -5.363e-02]
 [ -2.558e-02   9.415e-02  -2.174e-01   9.434e-01   2.055e-01]
 [  8.191e-04  -1.894e-03  -2.341e-03   5.620e-02   9.472e-01]]
```

Reconstruction Operator R

In [14]:
```
m1 = np.append(2*P.T.dot(P), [omega_hat], axis=0);
m1 = np.append(m1, np.append(omega_hat, 0.0).reshape(N+2, 1), axis = 1)

m2 = np.append(2*P.T, [1.0/N_S * np.ones(N_S)], axis=0)

Rtilde = np.linalg.solve(m1, m2);
R = np.delete(Rtilde, N+1, 0)
print R
```

```
[[ 1.014  0.113 -0.13  -0.073  0.04   0.074  0.007 -0.075  0.03 ]
 [-0.064  0.514  0.468  0.195 -0.041 -0.112 -0.021  0.095 -0.034]
 [ 0.038 -0.137  0.066  0.32   0.426  0.32   0.066 -0.137  0.038]
 [-0.034  0.095 -0.021 -0.112 -0.041  0.195  0.468  0.514 -0.064]
 [ 0.03  -0.075  0.007  0.074  0.04  -0.073 -0.13   0.113  1.014]]
```

Appendix B

A Sample ExaHyPE Configuration File

```
/**
 * This file is part of the ExaHyPE project.
 * Copyright (c) 2016  http://exahype.eu
 * All rights reserved.
 *
 * The project has received funding from the European
 * Union's Horizon 2020 research and innovation
 * programme under grant agreement No 671698. For
 * copyrights and licensing, please consult the
 * webpage.
 *
 * Released under the BSD 3 Open Source License.
 * For the full license text, see LICENSE.txt
 **/

/**
 * Euler Flow. A simple project.
 **/

exahype-project Euler
  peano-kernel-path = ./Peano
  exahype-path      = ./ExaHyPE
  output-directory  = ./ApplicationExamples/EulerFlow
  architecture      = noarch

  computational-domain
    dimension = 2
    width     = 1.0, 1.0
    offset    = 0.0, 0.0
    end-time  = 0.12
  end computational-domain

  shared-memory
    identifier      = autotuning
    cores           = 4
    properties-file = sharedmemory.properties
  end shared-memory

  distributed-memory
    identifier  = static_load_balancing
```

```
41      configure   = {hotspot,fair,ranks_per_node:4}
42      buffer-size = 64
43      timeout     = 120
44    end distributed-memory
45
46    optimisation
47      fuse-algorithmic-steps        = on
48      fuse-algorithmic-steps-factor = 0.99
49    end optimisation
50
51    profiling
52      profiler = LikwidProfiler
53      metrics  = {LikwidPowerAndEnergyMonitoringModule}
54      profiling-output = results.json
55      likwid_inc = /usr/local/include
56      likwid_lib = /usr/local/lib
57    end profiling
58
59    solver ADER-DG MyEulerSolver
60      variables         = 5
61      parameters        = 0
62      order             = 3
63      maximum-mesh-size = 0.15
64      time-stepping     = global
65      kernel            = generic::fluxes::nonlinear
66      language          = C
67
68      plot vtk::Cartesian::cells::ascii
69        variables = 5
70        time      = 0.0
71        repeat    = 0.05
72        output    = ./conserved
73        select    = {}
74      end plot
75    end solver
76  end exahype-project
```

Bibliography

[1] Robert Rosner, et al. The opportunities and challenges of exascale computing. *US Dept. of Energy Office of Science, Summary Report of the Advanced Scientific Computing Advisory Committee (ASCAC) Subcommittee*, 2010.

[2] Jeffrey S. Vetter, editor. *Contemporary High Performance Computing: From Petascale toward Exascale, Volume Two*. Chapman and Hall/CRC, April 2015.

[3] Erich Strohmaier, et al. TOP500 Supercomputer Sites (June 2016). `https://www.top500.org/lists/2016/06/`, June 2016.

[4] K. De Bosschere. *Applications, Tools and Techniques on the Road to Exascale Computing*. IOS Press, May 2012. Google-Books-ID: 9QmL0yNnmIgC.

[5] Bob Steigerwald, et al. *Energy Aware Computing: Powerful Approaches for Green System Design*. Intel Press, Hillsboro, Or., March 2012.

[6] C. Edwards. The exascale challenge. *Engineering Technology*, 5(18):53–55, December 2010.

[7] Erich Strohmaier, et al. TOP500 Supercomputer Sites — Performance Development. `https://www.top500.org/statistics/perfdevel/`, June 2016.

[8] Jack J. Dongarra, et al. *LINPACK Users' Guide*. Siam, 1979.

[9] Lewis Fry Richardson. *Weather Prediction by Numerical Process*. Cambridge University Press, 2007.

[10] Eleuterio F. Toro. *Riemann Solvers and Numerical Methods for Fluid Dynamics*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[11] Michael Dumbser, et al. A Posteriori Subcell Limiting of the Discontinuous Galerkin Finite Element Method for Hyperbolic Conservation Laws. *Journal of Computational Physics*, 278:47–75, December 2014.

[12] Bernardo Cockburn and Chi-Wang Shu. TVB Runge-Kutta local projection discontinuous Galerkin finite element method for conservation laws. II. General framework. *Mathematics of computation*, 52(186):411–435, 1989.

[13] Bernardo Cockburn, et al., editors. *Discontinuous Galerkin Methods: Theory, Computation and Applications*. Springer, Berlin; New York, softcover reprint of the original 1st ed. 2000 edition edition, January 2000.

[14] Dominic Etienne Charrier. *ADER-DG on Adaptive Spacetrees*. (unpublished), 2016, July 6.

[15] P. Sagaut, et al. Computational AeroAcoustics: From acoustic sources modeling to farfield radiated noise predictionADER discontinuous Galerkin schemes for aeroacoustics. *Comptes Rendus Mécanique*, 333(9):683–687, September 2005.

[16] Arne Taube, et al. Arbitrary High-Order Discontinuous Galerkin Schemes for the Magnetohydrodynamic Equations. *Journal of Scientific Computing*, 30(3):441–464, June 2006.

[17] Michael Dumbser, et al. Finite volume schemes of very high order of accuracy for stiff hyperbolic balance laws. *Journal of Computational Physics*, 227(8):3971–4001, April 2008.

[18] Olindo Zanotti, et al. Solving the relativistic magnetohydrodynamics equations with ADER discontinuous Galerkin methods, a posteriori subcell limiting and adaptive mesh refinement. *Monthly Notices of the Royal Astronomical Society*, 452(3):3010–3029, 2015.

[19] Michael Dumbser, et al. High order ADER schemes for a unified first order hyperbolic formulation of continuum mechanics: Viscous heat-conducting fluids and elastic solids. *Journal of Computational Physics*, 314:824–862, June 2016.

[20] W. Michael Lai, et al. *Introduction to Continuum Mechanics*. Butterworth-Heinemann, 2009.

[21] John W. Eaton, et al. *GNU Octave Version 4.0.0 Manual: A High-Level Interactive Language for Numerical Computations*. 2015.

[22] Michael Dumbser, et al. A unified framework for the construction of one-step finite volume and discontinuous Galerkin schemes on unstructured meshes. *Journal of Computational Physics*, 227(18):8209–8253, September 2008.

[23] Michael Dumbser. *Arbitrary High Order Schemes for the Solution of Hyperbolic Conservation Laws in Complex Domains*. Shaker, 2005.

[24] Dominic Etienne Charrier. *ADER-DG on Adaptive Spacetrees – Nine Months Review*. (unpublished), September 2016.

[25] Sergei Konstantinovich Godunov. A difference method for numerical calculation of discontinuous solutions of the equations of hydrodynamics. *Matematicheskii Sbornik*, 89(3):271–306, 1959.

[26] C. Lawson and R. Hanson. *Solving Least Squares Problems*. Classics in Applied Mathematics. Society for Industrial and Applied Mathematics, January 1995.

[27] Michael Dumbser and Martin Käser. Arbitrary high order non-oscillatory finite volume schemes on unstructured meshes for linear hyperbolic systems. *Journal of Computational Physics*, 221(2):693–723, February 2007.

[28] Shajulin Benedict. Energy-aware performance analysis methodologies for HPC architectures—An exploratory study. *Journal of Network and Computer Applications*, 35(6):1709–1719, November 2012.

[29] H. David, et al. RAPL: Memory power estimation and capping. In *2010 ACM/IEEE International Symposium on Low-Power Electronics and Design (ISLPED)*, pages 189–194. August 2010.

[30] Jan Treibig, et al. Best practices for HPM-assisted performance engineering on modern multicore processors. *arXiv:1206.3738 [cs]*, 7640:451–460, 2013.

[31] Lyla B. Das. *The x86 Microprocessors: 8086 to Pentium, Multicores, Atom and the 8051 Microcontroller, 2nd Edition*. Pearson India, second edition, May 2014.

[32] Intel Corporation. *Combined Volume Set of Intel®64 and IA-32 Architectures Software Developer's Manual*. Number 253665-059US. June 2016.

[33] Roger Kay. Intel And AMD: The Juggernaut Vs. The Squid. `http://www.forbes.com/sites/rogerkay/2014/11/25/intel-and-amd-the-juggernaut-vs-the-squid/`, November 2014.

[34] M. Véstias and H. Neto. Trends of CPU, GPU and FPGA for high-performance computing. In *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–6. September 2014.

[35] Wu-chun Feng and Tom Scogland. GREEN500 Supercomputer Sites. `https://www.top500.org/green500/lists/2016/06/`, June 2016.

[36] Summit. Scale new heights. Discover new solutions. `http://www.olcf.ornl.gov/summit/`.

[37] Sierra Advanced Technology System. `http://computation.llnl.gov/computers/sierra-advanced-technology-system`.

[38] Aurora. `http://aurora.alcf.anl.gov/`.

[39] Intel Timeline: A History of Innovation. `http://www.intel.com/content/www/us/en/history/historic-timeline.html`.

[40] Terje Mathisen. Pentium Secrets. *Byte magazine*, October 1999.

[41] T. Röehl, et al. Overhead Analysis of Performance Counter Measurements. In *2014 43rd International Conference on Parallel Processing Workshops*, pages 176–185. September 2014.

[42] Kevin S. London, et al. End-user Tools for Application Performance Analysis Using Hardware Counters. In *ISCA PDCS*, pages 460–465. 2001.

[43] John Levon, et al. OProfile, a system-wide profiler for Linux systems. *Homepage: http://oprofile. sourceforge. net*, 2008.

[44] Arnaldo Carvalho de Melo. The new linux'perf'tools. In *Slides from Linux Kongress*, volume 18. 2010.

[45] Philip J Mucci, et al. PAPI: A portable interface to hardware performance counters. In *Proceedings of the Department of Defense HPCMP Users Group Conference*, pages 7–10. 1999.

[46] Jan Treibig, et al. LIKWID: Lightweight Performance Tools. *arXiv:1104.4874 [cs]*, pages 207–216, September 2010.

[47] Advanced Micro Devices Inc. AMD64 architecture programmer's manual volume 2: System programming. 2016.

[48] Bernd Mohr, et al. TAU: A portable parallel program analysis environment for pC++. In *Parallel Processing: CONPAR 94—VAPP VI*, pages 29–40. Springer, 1994.

[49] V. M. Weaver, et al. Measuring Energy and Power with PAPI. In *2012 41st International Conference on Parallel Processing Workshops*, pages 262–268. September 2012.

[50] Carla Beatriz Guillén Carías. *Knowledge-Based Performance Monitoring for Large Scale HPC Architectures*. Ph.D. thesis, München, Technische Universität München, Diss., 2015, 2015.

[51] Thomas Röhl. Performance monitoring on Intel Haswell platforms, October 2015.

[52] Advanced Micro Devices, Inc. AMD Opteron 6200 Series Processors Linux Tuning Guide, 2012.

[53] E. Rotem, et al. Power-Management Architecture of the Intel Microarchitecture Code-Named Sandy Bridge. *IEEE Micro*, 32(2):20–27, March 2012.

[54] D. Hackenberg, et al. Power measurement techniques on standard compute nodes: A quantitative comparison. In *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 194–204. April 2013.

[55] Daniel Hackenberg, et al. An energy efficiency feature survey of the intel haswell processor. In *Parallel and Distributed Processing Symposium Workshop (IPDPSW), 2015 IEEE International*, pages 896–904. IEEE, 2015.

[56] Spencer Desrochers, et al. Initial Validation of DRAM and GPU RAPL Power Measurements. Tech report, UMaine VMW Group, August 2015.

[57] Spencer Desrochers, et al. A Validation of DRAM RAPL Power Measurements. 2016.

[58] Marcus Hähnel, et al. Measuring Energy Consumption for Short Code Paths Using RAPL. *SIGMETRICS Perform. Eval. Rev.*, 40(3):13–17, January 2012.

[59] B. Goel, et al. Portable, scalable, per-core power estimation for intelligent resource management. In *Green Computing Conference, 2010 International*, pages 135–146. August 2010.

[60] Ibrahim Takouna, et al. Accurate mutlicore processor power models for power-aware resource management. In *Dependable, Autonomic and Secure Computing (DASC), 2011 IEEE Ninth International Conference on*, pages 419–426. IEEE, 2011.

[61] M. Yasin, et al. Ultra compact, quadratic power proxies for multi-core processors. In *2013 IEEE 20th International Conference on Electronics, Circuits, and Systems (ICECS)*, pages 954–957. December 2013.

[62] Robert Schöne, et al. Tools and Methods for Measuring and Tuning the Energy Efficiency of HPC Systems. *Scientific Programming*, 22(4):273–283, 2014.

[63] Renato Miceli, et al. AutoTune: A Plugin-Driven Approach to the Automatic Tuning of Parallel Applications. In Pekka Manninen and Per Öster, editors, *Applied Parallel and Scientific Computing*, number 7782 in Lecture Notes in Computer Science, pages 328–342. Springer Berlin Heidelberg, June 2012.

[64] Hans-Joachim Bungartz, et al. Introduction. In *Modeling and Simulation*, Springer Undergraduate Texts in Mathematics and Technology, pages 1–15. Springer Berlin Heidelberg, 2014.

[65] The ExaHyPE consortium. ExaHyPE - An Exascale Hyperbolic PDE Engine (project flyer), 2016.

[66] U.S. Department of Energy. Exascale Challenges — U.S. DOE Office of Science (SC). `http://science.energy.gov/ascr/research/scidac/exascale-challenges/`, May 2016.

[67] U.S. Energy Information Administration. EIA - Electricity Data. `https://www.eia.gov/electricity/monthly/epm_table_grapher.cfm?t=epmt_5_6_a`, August 2016.

[68] Robert F. Service. Who Will Step Up to Exascale? *Science*, 339(6117):264–266, January 2013.

[69] Craig Chambers, et al. FlumeJava: Easy, Efficient Data-parallel Pipelines. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 363–375. ACM, New York, NY, USA, 2010.

[70] Apache Software Foundation. Hadoop. `https://hadoop.apache.org`.

[71] David A. Patterson. *Computer Architecture: A Quantitative Approach*. Elsevier, October 2011.

[72] Wm. A. Wulf and Sally A. McKee. Hitting the Memory Wall: Implications of the Obvious. *SIGARCH Comput. Archit. News*, 23(1):20–24, March 1995.

[73] Maurice V. Wilkes. The Memory Gap and the Future of High Performance Memories. *SIGARCH Comput. Archit. News*, 29(1):2–7, March 2001.

[74] Maxime Martinasso and Jean-François Méhaut. A Contention-Aware Performance Model for HPC-Based Networks: A Case Study of the InfiniBand Network. In Emmanuel Jeannot, Raymond Namyst, and Jean Roman, editors, *Euro-Par 2011 Parallel Processing*, number 6852 in Lecture Notes in Computer Science, pages 91–102. Springer Berlin Heidelberg, August 2011.

[75] Alexander Heinecke, et al. Petascale high order dynamic rupture earthquake simulations on heterogeneous supercomputers. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 3–14. IEEE Press, 2014.

[76] Davide Vanzo, et al. Pollutant transport by shallow water equations on unstructured meshes: Hyperbolization of the model and numerical solution via a novel flux splitting scheme. *Journal of Computational Physics*, 321:1–20, 2016.

[77] J. Greenberg. Extensions and Amplifications of a Traffic Model of Aw and Rascle. *SIAM Journal on Applied Mathematics*, 62(3):729–745, January 2002.

[78] Kentaro Takami, et al. Constraining the Equation of State of Neutron Stars from Binary Mergers. *Physical Review Letters*, 113(9):091104, August 2014.

[79] The ExaHyPE consortium. Website of the ExaHyPE project. `http://exahype.eu/`, 2016.

[80] The ExaHyPE team. *ExaHyPE Guidebook*. August 2016.

[81] Dominic Etienne Charrier. ADER-DG on spacetrees in the ExaHyPE project, 12-15 April 2016.

[82] Dominic Etienne Charrier. In-situ uncertainty identification on many solver adaptive spacetrees in ExaHyPE, 23-26 May 2016.

[83] Tobias Weinzierl and others. Peano—a Framework for PDE Solvers on Spacetree Grids. 2016. Http://www.peano-framework.org.

[84] Markus Brenk, et al. Numerical simulation of particle transport in a drift ratchet. *SIAM Journal on Scientific Computing*, 30(6):2777–2798, 2008.

[85] T. Weinzierl, et al. PaTriG –Particle Transport Simulation in Grids. In *High Performance Computing in Science and Engineering 2014*, pages 128–129. 2014.

[86] Moritz Simon and Michael Ulbrich. Optimal Control of Partially Miscible Two-Phase Flow with Applications to Subsurface CO2 Sequestration. In Michael Bader, Hans-Joachim Bungartz, and Tobias Weinzierl, editors, *Advanced Computing*, number 93 in Lecture Notes in Computational Science and Engineering, pages 81–98. Springer Berlin Heidelberg, 2013.

[87] Alexander Heinecke, et al. LIBXSMM: A High Performance Library for Small Matrix Multiplications. 2015.

[88] Jürg Hutter, et al. cp2k: Atomistic simulations of condensed matter systems. *Wiley Interdisciplinary Reviews: Computational Molecular Science*, 4(1):15–25, January 2014.

[89] Tobias Weinzierl and Michael Dumbser. ExaHyPE - An Exascale Hyperbolic PDE Engine: A (very) brief introduction, July 2016.

[90] Google Inc. and contributors. Census - A stats collection framework. `https://github.com/google/census-java`, 2016.

[91] Roman Dementiev, et al. *Intel Performance Counter Monitor*. 2016.

[92] A. Bieswanger, et al. Power and thermal monitoring for the IBM System z10. *IBM Journal of Research and Development*, 53(1):14:1–14:9, January 2009.

[93] Nicholas Nethercote and Julian Seward. Valgrind: A Framework for Heavy-weight Dynamic Binary Instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 89–100. ACM, New York, NY, USA, 2007.

[94] L. Li and C. Kessler. MeterPU: A Generic Measurement Abstraction API Enabling Energy-Tuned Skeleton Backend Selection. In *2015 IEEE Trustcom/BigDataSE/ISPA*, volume 3, pages 154–159. August 2015.

[95] Arndt Bode. *Jahresbericht 2015 Des Leibniz-Rechenzentrums*. Garching bei München, 2015.

[96] Daniela Alic, et al. Constraint damping of the conformal and covariant formulation of the Z4 system in simulations of binary neutron stars. *Physical Review D*, 88(6), September 2013.

[97] Luciano Rezzolla and Olindo Zanotti. *Relativistic Hydrodynamics*. Oxford University Press, 2013.

[98] Ian Cutress. AnandTech: Intel Haswell-EP Xeon 14 Core Review: E5-2695 V3 and E5-2697 V3. `http://www.anandtech.com/show/8730/intel-haswellep-xeon-14-core-review-e52695-v3-and-e52697-v3`, November 2014.