Predicting Power and Timing of Large-Scale Distributed Applications on Highly Heterogeneous Platforms

Von der Fakultät für Informatik, Wirtschafts- und Rechtswissenschaften der Carl von Ossietzky Universität Oldenburg zur Erlangung des Grades und Titels eines

Doktors der Ingenieurwissenschaften (Dr.-Ing.)

angenommene Dissertation

von Herrn Jörg Walter
geboren am 15. März 1977 in Kiel

Gutachter:
Prof. Dr.-Ing. Wolfgang Nebel

Weiterer Gutachter:
Prof. Dr.-Ing. Axel Hahn

Tag der Disputation:
10. Dezember 2020

Zusammenfassung

Im Bereich des Hoch- und Höchstleistungsrechnens oder auch High Performance Computings (HPC) werden hochgradig parallele Anwendungen auf verteilten Systemen mit hunderten von Rechenknoten ausgeführt. Durch die Dimensionen von Plattformen und Programmen sind Vorhersagen über Laufzeit und Energiebedarf dieser Programme schwierig. Diese Schwierigkeiten werden noch verstärkt, wenn Entwickler ihre Anwendungen optimieren wollen, ohne dazu Messungen durch instrumentierte Probeläufe auf der Zielplattform zu machen. Zudem behandeln die meisten Vorhersagemethoden nur den Zeitaspekt, da bis vor kurzem der Energiebedarf für die meisten HPC-Entwickler keine Bedeutung hatte. Zwei unabhängige Entwicklungen haben diese Situation verändert: Zum einen werden auch bei eingebetteten Systemen immer öfter Cluster-Architekturen eingesetzt, und zum anderen wird im HPC-Bereich inzwischen mit einer Überprovisionierung von Rechenknoten gearbeitet, wodurch ein direkter Einfluss von Energiebedarf auf die verfügbare Rechenleistung entsteht.

Ich stelle in dieser Arbeit einen neuen Ansatz vor, der mit einer Simulationstechnik arbeitet, wie er im Bereich der eingebetteten Systeme verbreitet ist. Will man eine solche Technik auf die Größenordnung von HPC-Systemen anwenden, kann jedoch die eigentliche Anwendungsfunktionalität nicht mehr ausgeführt werden. Stattdessen benutzt die vorgestellte Methodik eine abstrakte Simulation auf Basis von Task-Graphen, wie sie im HPC-Bereich beliebt sind, und die Gemeinsamkeiten mit synchronen Datenflussgraphen aus dem Bereich der eingebetteten Systeme haben.

Vorhersagen zu Rechenoperationen werden über eine messbasierte Charakterisierungsprozedur realisiert. Anders als existierende Trace-basierte Ansätze benötigt diese Prozedur jedoch kein Ausführen der gesamten Applikation auf der gesamten Zielplattform. Es reicht ein einzelner instrumentierter Rechenknoten aus, um die notwendigen Eingabeparameter für die abstrakte Simulation zu generieren. In dieser Arbeit stelle ich auch eine beispielhafte Plattform vor, die zeigt, welche Messdatenerfassung und welche Hardwareeigenschaften die vorgestellte Vorhersagemethodik benötigt.

Während Rechenoperationen abstrakt gehandhabt werden, behandelt die Simulation Kommunikationsvorgänge mit deutlich höherem Detailgrad, da dies mehr Einfluss auf die Genauigkeit der Vorhersagen hat. Das Ergebnis ist eine Vorhersagemethode für Zeit und Energie, die schnell genug für interaktives Feedback bei der Anwendungsentwicklung ist, und die dabei ein hinreichendes Maß an Genauigkeit hat. Dadurch eignet sie sich außerdem für automatisierte Exploration von Entwurfsmöglichkeiten hinsichtlich Algorithmen, Parallelisierungsgrad, oder Task-Granularität.

Ich habe die Arbeit mit Hilfe von direkten physikalischen Messungen evaluiert, anstatt mich nur auf detailliertere Referenzmodelle oder indirekte Messungen wie z.B. Energiemodelle auf Basis von CPU-Ereigniszählern zu verlassen. Dadurch sind unerwartete Effekte sichtbar geworden, die erheblichen Einfluss auf das Systemverhalten haben können. Insgesamt betrachtet konnten die Zeitvorhersagen eine gute Genauigkeit erreichen, die mit etablierten Verfahren vergleichbar ist. Die Evaluation der Energievorhersagen zeigte weniger eindeutige Ergebnisse: Vorhersagen haben ein klares Potential für hinreichende Genauigkeit, aber sie wurden von externen Einflüssen (insbesondere Herstellungsqualität und Umgebungsbedingungen) beeinträchtigt, die im Rahmen dieser Arbeit nicht vollständig kompensiert werden konnten.

Abstract

Applications in the high-performance computing (HPC) domain are often designed to run on cluster-like distributed platforms with hundreds of nodes. Due to the size of both – applications and platforms – predictions of application run time and energy usage is challenging. Difficulties increase when developers want to guide application design without profiling on a massively parallel platform. Furthermore, most HPC prediction methodologies only address timing, because energy predictions used to have little relevance for HPC application design. Two different developments changed this: Cluster architectures are becoming popular in the embedded domain, and hardware overprovisioning in recent HPC systems creates a direct influence of energy usage on application performance.

I propose a new approach to this challenge based on a simulation technique well known in the embedded computing domain. In order to apply such a methodology at HPC scale, I cannot execute actual applications during simulation. I use abstract simulation based on the Task Graph model of computation, which is popular in HPC and which has properties similar to the synchronous dataflow model that is popular in the embedded domain.

For computation predictions, the methodology uses a measurement-based characterisation procedure. Unlike trace-based HPC prediction methodologies, this procedure does not require a full application execution over the entire target platform. Instead, a single instrumented cluster node is sufficient to create input parameters for the abstract simulation. I present an exemplary embedded cluster that demonstrates the measurement instrumentation and hardware properties required by the proposed methodology.

While the simulation handles execution in an abstract way, it keeps a high level of detail for communication, since this has much bigger impact on overall accuracy. The end result is a time and energy prediction methodology that is fast enough for interactive feedback in the application design workflow while still maintaining a useful level of accuracy. It also allows designers to employ automatic design space exploration across a wide range of high-level design choices like choice of algorithms, degree of parallelism, or task granularity.

I have performed the evaluation of the proposed methodology against direct physical measurements, and not against more detailed reference models or indirect measurements like CPU performance counter based energy models. This has uncovered unexpected practical issues that can have significant impact on system behaviour. Overall, timing predictions can reach a high degree of accuracy, fully on par with established HPC prediction methodologies. The evaluation of energy predictions is less conclusive: results clearly show a potential for useful accuracy, but they have a higher dependency on external factors like hardware build quality and environmental conditions than could be controlled for in this thesis.

Publications

Some ideas and figures have appeared previously in the following publications:

[1] J. Walter, R. Görgen, and W. Nebel, "Predicting performance and energy efficiency for large-scale parallel applications on highly heterogeneous platforms," in *19th GI/ITG/GMM Workshop Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen, MBMV 2016, Freiburg im Breisgau, Germany, March 1-2, 2016.*, 2016, pp. 116–127.

[2] J. Walter and W. Nebel, "Energy-Aware Mapping and Scheduling of Large-Scale Macro Data-Flow Applications," in *1st International Workshop on Investigating Dataflow in Embedded Computing Architecture*, 2015. 9.6.2

[3] P. Knocke, R. Görgen, J. Walter, D. Helms, and W. Nebel, "Using early power and timing estimations of massively heterogeneous computation platforms to create optimized HPC applications," in *Proceedings of the 12th IEEE International Conference on Embedded and Ubiquitous Computing*, 2014.

[4] J. Walter, M. Fakih, and K. Grüttner, "Hardware–Based Real–Time Simulation on the Raspberry Pi," in *2nd Workshop on High-performance and Real-time Embedded Systems (HiRES 2014)*, Jan. 2014. 10.1.4.1

Contents

I. Introduction 15

1. Context and Motivation 17

2. Scope of Contribution 19
2.1. Research Questions 19

3. Outline 23

II. Fundamentals 25

4. Timing and Energy Prediction for HPC Applications 27
4.1. Hardware Aspects 27
4.1.1. Speed Variation and Synchronisation 27
4.1.2. Current and Future Hardware Architectures 28
4.1.3. HPC Cluster Scale 28
4.1.4. Energy Considerations 29
4.1.5. Heat 30
4.1.5.1. Thermal Management 30
4.1.5.2. Observable Effects 31
4.1.5.3. Modelling 31
4.1.6. CPU-specific Performance Optimisation 31
4.2. Software Aspects 33
4.2.1. HPC Problem Classes 33
4.2.2. Parallelism and its Limitations 34
4.2.3. Application Patterns 35
4.2.3.1. Structural Patterns 35
4.2.3.2. Parallel Algorithm Strategy Patterns 36
4.2.4. Scheduling 36
4.2.4.1. Dynamic Scheduling 37
4.2.4.2. Static Scheduling 37
4.2.4.3. Rescheduling 37
4.2.5. Resource Modelling 38
4.2.6. Auxiliary Aspects 39
4.2.6.1. Data Representation 39

4.2.6.2. Data Distribution . . . 39
4.2.6.3. Fault Tolerance . . . 40
4.3. Parallelisation Tools . . . 40
4.3.1. Local Parallelism . . . 40
4.3.2. Communication Middleware . . . 41
4.3.3. Deployment . . . 42
4.4. Common Benchmarks . . . 42
4.5. Common Prediction Approaches . . . 44
4.5.1. Analytical Models . . . 44
4.5.2. Simulation . . . 45
4.5.3. Symbolic Simulation . . . 45
4.5.4. Trace-Based Simulation . . . 46

5. Time and Energy Measurement 47
5.1. Data Collection . . . 47
5.2. Time Measurement . . . 48
5.2.1. Eligible Effects . . . 48
5.2.2. Measurement Techniques . . . 48
5.2.2.1. Internal Measurement . . . 48
5.2.2.2. Comparability . . . 48
5.2.2.3. External Measurement . . . 49
5.3. Energy Measurement . . . 49
5.3.1. Eligible Effects . . . 49
5.3.2. Measurement . . . 50
5.3.2.1. Microarchitectural Power Estimation . . . 51
5.3.2.2. Power Management Circuits . . . 51
5.3.2.3. Dedicated Measurement Circuits . . . 52
5.3.2.4. External Measurement . . . 52
5.3.2.5. Parallel Measurement . . . 53

6. Modelling and Simulating with SystemC 55
6.1. Components of SystemC Models . . . 55
6.2. Discrete-Event Simulation . . . 56
6.3. System Level Modelling . . . 57
6.3.1. Computation . . . 58
6.3.2. Communication . . . 59
6.4. Time and Energy Traces . . . 59

7. Thesis Contributions 61
7.1. Contributions . . . 61
7.2. Assumptions . . . 63

8. Related Work **67**
8.1. Time and Energy Prediction . . . 67
8.1.1. Execution-Driven Simulation . . . 67
8.1.2. Trace-Based Simulation . . . 68
8.1.3. Abstract Simulation . . . 69
8.1.4. Analytical Modelling . . . 69
8.2. Measurement Platforms . . . 70

III. Models and Methodology **71**

9. Power and Timing Prediction Methodology **73**
9.1. Overall Design Flow . . . 73
9.2. Abstract Application Model . . . 76
9.2.1. Task Graph . . . 76
9.2.2. Application Semantics . . . 79
9.2.3. Modelling Process . . . 80
9.2.4. Discussion of Design Decisions . . . 81
9.3. Execution Runtime Model . . . 82
9.3.1. Initialisation . . . 83
9.3.2. Main Loop . . . 84
9.3.3. Communication Scheduler . . . 84
9.3.4. Hardware Abstraction Layer . . . 85
9.3.5. Overall Design Decision . . . 86
9.4. Abstract Platform Model . . . 86
9.4.1. Platform Model . . . 86
9.4.1.1. Platform Hierarchy Graph . . . 86
9.4.1.2. Platform Communication Graph . . . 87
9.4.1.3. Platform Graph . . . 88
9.4.2. Platform Resource Model . . . 89
9.4.3. Discussion of Design Decisions . . . 89
9.5. Computation Resource Model . . . 92
9.5.1. Abstract Resource Model . . . 93
9.5.2. Model Building . . . 94
9.5.3. Discussion of Design Decisions . . . 94
9.6. Mapping . . . 95
9.6.1. Representation . . . 95
9.6.2. Automatic Mapping . . . 95
9.7. Simulation Model . . . 96
9.7.1. SystemC Object Hierarchy . . . 96
9.7.2. Power Models . . . 98
9.7.3. Computation Modelling . . . 98
9.7.4. Communication Modelling . . . 100
9.7.4.1. Basic Operation . . . 100

9.7.4.2. Routing 102
9.7.5. Resulting Prediction 102
9.8. Physical Execution 103
9.8.1. Deployment 103
9.8.2. Initialisation 104
9.8.3. Execution 104
9.8.3.1. Node-Local Time 105
9.8.3.2. Time Synchronisation 105
9.8.4. Communication 106
9.8.4.1. TCP Backend 106
9.8.4.2. Eth Backend 106

10. Measurement Platform **111**
10.1. Power Distribution and Measurement 111
10.1.1. Power Distribution 111
10.1.2. Analogue Front-End 112
10.1.3. Signal Acquisition 115
10.1.3.1. Multiplexer 115
10.1.3.2. Analog-to-Digital Converter 115
10.1.3.3. Data Transfer 117
10.1.4. Signal Processing and Transmission 117
10.1.4.1. Data Reduction 119
10.1.4.2. Transmission 119
10.1.4.3. System Control 119
10.2. Embedded Cluster Platform 119
10.2.1. Cluster Management 122
10.3. Measurement Process 122
10.3.1. Energy 122
10.3.2. Time 122
10.3.3. Time Correlation 123
10.3.4. Kernel Characterisation 124
10.3.4.1. Energy Markers 125
10.3.4.2. External Time Measurement 126
10.3.4.3. Secondary Time Model 126
10.4. Platform Characterisation 126
10.4.1. Sustainable Clock Frequency 127
10.4.2. Platform Power 127
10.4.3. Communication Timing 128
10.4.3.1. Cutoff Metric 128
10.4.3.2. Model Parameters 129
10.4.3.3. Benchmark Suite 129
10.4.4. Communication Power 130

IV. Evaluation **131**

11. Evaluation Goals **133**
11.1. Overall Methodology . . . 133
11.1.1. End User Requirements . . . 133
11.1.2. Evaluation Criteria . . . 133
11.1.2.1. Time . . . 133
11.1.2.2. Energy . . . 134
11.1.2.3. Performance . . . 134
11.2. Measurement Accuracy . . . 135
11.2.1. Time . . . 135
11.2.2. Energy . . . 136
11.3. Evaluation Platform . . . 136
11.3.1. Platform Characterisation . . . 137
11.3.2. Power Variation and Heat . . . 137
11.4. Individual Design Decisions . . . 137

12. Evaluation of Measurement Accuracy **139**
12.1. Setup . . . 139
12.1.1. DC Accuracy . . . 139
12.1.2. Channel Independence . . . 141
12.1.3. Frequency Response . . . 141
12.1.4. Time Measurement . . . 141
12.2. Results . . . 142
12.2.1. DC Accuracy . . . 142
12.2.2. Channel Independence . . . 145
12.2.3. Frequency Response . . . 145
12.2.4. Time Measurement . . . 147
12.3. Discussion . . . 150
12.3.1. Energy . . . 150
12.3.2. Timing . . . 151
12.3.3. Summary . . . 152

13. Characterisation of the Evaluation Platform **153**
13.1. Setup . . . 153
13.1.1. Platform Model . . . 153
13.1.2. Platform Characterisation . . . 153
13.2. Results . . . 154
13.2.1. Clock Speed . . . 154
13.2.2. Platform Power . . . 155
13.2.3. Communication Timing . . . 155
13.2.4. Communication Power . . . 159
13.3. Discussion . . . 159
13.3.1. Sustainable Clock Speed . . . 159

13.3.2. Platform Power 162
13.3.3. Communication Timing 163
13.3.4. Communication Power 163
13.3.5. Summary 164

14. Evaluation of the Overall Methodology **165**
14.1. Setup 165
14.1.1. Overview 165
14.1.2. Measurement Details 165
14.1.3. Cholesky Matrix Subdivision 166
14.1.4. Computation Resource Model 167
14.1.5. Application Benchmark 167
14.2. Results and Discussion 168
14.2.1. Computation Resource Model 168
14.2.2. Application Benchmark 171
14.2.2.1. Time Predictions 173
14.2.2.2. Simulation Speed 173
14.2.2.3. Energy Predictions 175
14.3. Summary 177

15. Evaluation of Individual Design Decisions **181**
15.1. Fixed Clock Frequencies 181
15.2. External Time Measurement 182
15.2.1. Reproduction of Inconsistent Behaviour 182
15.2.2. Results 182
15.2.3. Discussion 185
15.3. Custom Network Protocol 185
15.4. Custom Runtime System 187
15.4.1. Discussion 187

V. Conclusion **189**

VI. Appendix **195**

References **197**

List of Figures **203**

List of Tables **205**

Listings **207**

Nomenclature **209**

Part I.

Introduction

1. Context and Motivation

In the high-performance computing (HPC) world, platforms have become much more complex over time. Ever since the first Cray supercomputer, a core principle of HPC systems was parallelisation. There is cluster-level parallelism where multiple distinct machines collaborate through explicit communication, and local parallelism where multiple processors within one machine collaborate through shared memory.

The logical continuation of this principle led to clusters of HPC servers, each equipped with multiple processors, each of which as multiple CPU cores. In these clusters, every computation resource (i. e. CPU core) is the same – this is the age of homogeneous supercomputing, which has become rare in the HPC field.

A real change came about when graphics processing units (GPUs) evolved from fixed geometry manipulation circuits to highly parallel single-instruction-multiple-data (SIMD) processors. CPUs had SIMD capabilities as well, but GPUs exceeded the parallelism of CPUs by an order of magnitude or two. People started to use them for scientific computing tasks, and GPUs optimised for this use case are called general purpose GPUs (GPGPUs).

Thus was the concept of heterogeneous supercomputing born. The Top 500 list of the fastest computers on earth is dominated by such machines that consist of clusters of machines, each with multiple multi-core CPUs and multiple GPGPUs.

In order to increase the performance for certain workloads, companies invented various even more unconventional computation resources. Manycore processors like the Intel Xeon Phi or the Adapteva Epiphany tried to bridge the gap between highly parallel but inflexible GPGPUs and versatile but less parallel CPUs. These had impressive capabilities in theory, but were difficult to program, which led to their ultimate demise.

Field-programmable gate arrays (FPGAs) are another unconventional computation resource – programmable logic circuits. Hardware programming has always been challenging to normal software developers. Vendors tried to offer high level synthesis (HLS), i. e. the translation of imperative program code into a hardware description; for many years, this generated inefficient hardware designs. Over time the situation has improved: HLS is now capable of delivering competitive designs, but programmers still need high familiarity with the strengths and weaknesses of programmable logic.

The most important development towards accessibility of FPGAs to normal programmers was the combination of traditional CPU cores and FPGA fabric on a single chip. This made it easy to accelerate those parts of a program that would profit the most, while everything else could be written in a traditional programming language. Essentially, people now use FPGAs as add-on accelerators like GPGPUs. As an example for their success, big cloud providers now offer all three types of computation resources (CPUs, GPGPUs, FPGAs) for rent.

What keeps FPGAs from becoming a mainstream accessory is complexity: HPC programmers have adapted to CPU+GPGPU programming, but they already need two fundamentally different mental models for them. Even assuming that HLS works perfectly, a third type of computation element massively increases the amount of possible hardware setups and parallelisation strategies. HPC needs new ways to manage complexity. How do you optimise programs for such systems? How do you design efficient hardware platforms?

Another emerging challenge is energy usage of HPC systems. The 'Green 500' list is a companion to the Top 500 list of supercomputers that lists the most energy-efficient machines on earth. Yet users of shared HPC systems have no energy constraints whatsoever. They apply for computation time, get assigned a time slot, and can do whatever they want during that time. All this despite the fact that power supply leads to many follow-up costs: there is the cost of energy itself, there is the cost of cooling systems, and finally there is the cost of power distribution, which is massively overprovisioned in order to cope with the worst case, even though that is so rare that it may never occur during the lifetime of the cluster.

On the other hand, there are no established methodologies for programmers to consider energy usage of their programs. They want to optimise their programs *before* running them on the HPC system, but how do you find out if your program will stay within a desired energy limit?

Meanwhile, a research field on the opposite end of the 'hardware size' scale answers all these questions on a daily basis: Embedded system design, more specifically electronic system-level design, deals with exactly these questions: Given an abstract description of some functionality and constraints on timing and energy usage of that functionality, how do you build a system that performs this functionality within those constraints? How do you optimise such a system to minimise some arbitrary cost function? How do you decide which parts to implement in hardware or in software? What computation resources do you need? How do they communicate?

2. Scope of Contribution

In this thesis, I try to answer the emerging questions from the HPC field with lessons learned while answering the same questions in the embedded field. I propose a methodology that allows users to model software applications and hardware systems, to map applications to systems, and to get a prediction for time and energy of the resulting execution. The massive difference in system and application size requires trade-offs that would be unforgivable in embedded systems, but are inevitable for HPC applications.

One scenario I had in mind while developing the methodology is the process of design space exploration (DSE). In embedded system design, it is an established way to explore many different system configurations in order to find the set of Pareto-efficient solutions.

Thus my secondary goal is that the proposed prediction methodology is fast enough so that users can perform semi-automatic DSE over many design choices: Choice of fundamental algorithms, granularity of parallelisation, or usage of and mapping to computation resources. Commercial HPC users might want use predictions to design a hardware platform that is cost-optimised for a specific, known set of applications.

I consider this thesis a first step: It does not yet handle the full heterogeneity described above, but the methodology I present is intentionally generic and extensible so that future work can expand on it.

2.1. Research Questions

The resulting research questions I want to answer are these:

Research Question 1. *How can programmers of HPC applications manage the complex design space of different algorithms, possible target platforms, and mappings of their applications onto a platform?*

This an overarching question which the following questions derive from. It addresses the increased complexity of the modern HPC landscape. A systematic methodology is a common strategy for complexity management. For embedded systems, model-based design methodologies are well established. High performance computing does not yet have as much acceptance for more formal ways of designing applications.

Research Question 2. *How can programmers model applications so that their fundamental structure becomes more accessible to automated tools and formal methodologies?*

When trying to formalise a design methodology, especially when introducing automated processing steps, the source application should make it easy to extract the required

information. Just like assembler code is difficult to translate into a high-level programming language, arbitrary C++ code does not readily show opportunities for parallelisation or optimisation.

Research Question 3. *How can programmers model the platform they are targeting in an abstract way?*

For the same reason, such a design methodology also needs information about available computation or communication resources and key properties like computation speed or communication bandwidth.

On the other hand, many hardware details are not accessible to users of the hardware. Therefore it would be pointless to model every microarchitectural detail of a system. The model must support a granularity that matches observable metrics.

As a bonus, having an abstract model also enables users to model hypothetical platforms and explore the impact of a different target system.

Research Question 4. *How should an application behave when executed on the modelled hardware?*

Having an application model does not mean that its execution behaviour is fully specified. Programmers probably don't want to create a full semantic specification for themselves; it is sufficient that the methodology includes a specific runtime behaviour that is reasonably efficient.

This execution model is also required for the following group of questions, because predictions can only work if execution behaviour is sufficiently specified.

Research Question 5. *How can programmers of HPC applications optimise them without running them on the intended target platform?*

This is the second fundamental question that guides more detailed research questions. It is based on the observation that HPC platforms are often not accessible to programmers while they create the bulk of their applications. In the embedded field, hardware/software co-design poses a similar challenge: How do you test software written for hardware that does not yet exist? A common answer is 'simulation', and this thesis expands on this insight.

Research Question 6. *How can a simulation predict timing and energy of an application running on a target platform without executing functional behaviour?*

This is the key challenge when trying to simulate HPC applications on a typical development system: how to get it fast. The faster predictions can be generated, the better will the methodology be accepted by programmers for daily use. Indirectly this also addresses the question of making simulation fast enough for design space exploration.

Executing real HPC code on a developer workstation would already be much too slow outside of a simulator. If no functional behaviour can be executed, then the simulation needs a different way to determine time and energy; this question asks for some other resource usage model.

Research Question 7. *How can programmers create an abstract resource model when they don't have access to the target platform?*

This assumes that programmers cannot simply go to the target machine, hook up a multimeter, execute their program, and measure energy. The resource model should be designed in a way that developers can create missing parts of the resource model using tools that fit on their desks and in their budgets.

Research Question 8. *How do you measure energy and timing for the abstract resource model properly?*

Going further into details, it is surprisingly complex to acquire good measurement data. This is not about the general question of how to measure correctly, which should be part of basic technical education. This is about the question which aspects are important for the proposed methodology. Ideally, a recommended best practice shows a systematic way of performing relevant measurements.

3. Outline

This thesis is structured in five parts.

Part I - Introduction gives a general overview of the topic of this thesis. It concludes with this outline.

Part II - Fundamentals contains five subsections covering relevant topics from the technological environment this thesis is based on.

Section 4 covers topics that relate to HPC hard- and software, execution behaviour, and existing time prediction approaches; it also covers energy, but there is little work that specifically relates to HPC applications – this is a very recent field of research.

Instead, this is covered more generically in Section 5, which addresses the practical side of measurement of time and energy of hardware that is running applications.

After that, Section 6 deals with the simulation side of things, specifically SystemC as a discrete-event simulation framework.

After all related technologies have been presented, Section 7 states the specific contributions that this thesis makes; The part concludes with an overview of related work in Section 8.

Part III - Models and Methodology introduces the overall methodology. Section 9 presents the proposed overall design flow and all of its components, many directly addressing the research questions.

Section 10 then presents the evaluation platform I have built for this thesis. While part of the platform only exists for evaluation of the proposed methodology, the actual measurement infrastructure is a more generic proposal for creation of the resource model. Furthermore, the section also proposes a systematic approach to create a specific model and to check platforms for some effects that would impair usage with the proposed methodology.

Part IV - Evaluation then presents the evaluation process and its results. In Section 11, I present four evaluation goals, give an overview of each and derive detailed evaluation criteria. I then describe the evaluation process for each of the goals in Section 12 through Section 15, where Section 14 evaluates the main contributions.

Part V - Conclusion sums up the results of this thesis and presents promising extensions for future work.

Part II.

Fundamentals

4. Timing and Energy Prediction for HPC Applications

In the field of high performance computing (HPC), applications and platforms are much larger than in virtually any other computing domain. This poses unique challenges for attempts to predict execution times of programs.

Most importantly, the platforms being predicted are orders of magnitude bigger than the platforms used for development. Moreover, they often use specialised hardware and specialised application modelling approaches. The latter two properties also apply to embedded systems engineering, which is why the methodology proposed in this thesis actually has its roots in the embedded domain.

4.1. Hardware Aspects

Apart from the size, HPC hardware has another significant difference to embedded systems: Embedded systems usually run tightly synchronised, up to the point that different compute resources run off a single master clock signal. In contrast, HPC systems are built as an asynchronous cluster architecture.

A typical HPC cluster – as assumed for this thesis – consists of multiple compute nodes. A compute node is a computer that has one or more performance-optimised multi-core CPUs and possibly specialised compute accelerators. It is equipped with main memory (RAM) that is directly accessible to all compute resources in that node. That memory might have different timing characteristics depending on which computing resource wants to access which address (nonuniform memory architecture, NUMA).

Nodes are connected through a high-bandwidth networking technology, often Ethernet or InfiniBand, and the topology of this network need not be a simple star topology, e. g. it might be hierarchically structured.

Due to the distributed nature, and due to the size of applications and data sets, inter-node communication is a critical resource. Foreign traffic can have significant impact on application performance, so the cluster network is mostly isolated from the outside world, often only offering a single gateway node that manages application and data deployment.

4.1.1. Speed Variation and Synchronisation

With regard to this thesis, the most important implication of this architecture is the fact that different parts of the system run completely asynchronously. Clock speeds vary

due to manufacturing variations. Their speed even changes over time due to thermal effects. And nodes usually do not know their exact speed in relation to other nodes.

The employed networking technology does not offer a real-time mode of operation[1]; it works asynchronously, usually employing a store-and-forward approach for routing network packages.

Thus, time based synchronisation only works in the millisecond range, e. g. through the Network Time Protocol (NTP) [39]. Event based synchronisation needs to happen on higher levels of the communication protocol stack.

The net result is that events happening on different nodes cannot easily be synchronised, and event sequences from different nodes are difficult to relate to each other. It also means that applications that need frequent global synchronisation pay a significant performance overhead, which has implications for the suitability of different models of computation.

4.1.2. Current and Future Hardware Architectures

Current HPC systems have a slight degree of heterogeneity. Nodes tend to be identical. They usually consist of one or more high performance x86 CPUs and one or more general purpose GPUs (GPGPUs) as accelerators.

Some problems benefit from a higher count of slower CPU cores. For these applications, ARM based servers exploit the power saving nature of current ARM CPUs to densely pack many ARM CPUs in a small form factor, improving computation density.

GPUs are popular accelerators specialised for numerical computation. They are highly parallel floating point arithmetic units that can process thousands of (homogeneous) calculations in parallel. On such tasks, they usually exceed CPUs in performance per Watt or performance per cost. But for some problems, Field Programmable Gate Arrays (FPGAs) achieve higher efficiency than GPUs (e. g. [40]), so they start to appear in HPC systems as well.

Finally, there are approaches that allow combination of these elements in a single server rack mount [14]. With these HPC systems, users can build their hardware to solve the problem at hand in the most efficient way (by whatever efficiency metric they choose).

4.1.3. HPC Cluster Scale

HPC systems exist in various sizes. Smaller ones are used in commercial applications. For example, many engineering domains use a finite elements approach to physics simulations. These divide a physical volume of the real world into millions of sub-volumes and predict the interaction of physical properties (gas pressure, liquid flows, forces, etc.) for each of them. Another application of highly parallel commercial computation is video processing (encoding/transcoding, rendering of computer generated imagery,

[1]There is Ethernet Time Sensitive Networking (TSN), but it is not commonly used in HPC.

image analysis). Finally, machine learning is an emerging high-performance problem that also has applications in the embedded domain.

These systems may be as small as a single cluster node (making it just a regular computer) or as big as a few 19″ server racks, resulting in tens or hundreds of nodes, typically with a power consumption below 1 MW [24].

The other notable application is research, where the biggest HPC systems are used [11]. These consist of thousands of nodes and typically consume up to 16 MW [24]; even 20 MW is acceptable for future systems [23].

4.1.4. Energy Considerations

In research HPC systems, actual end users usually don't need not take energy into consideration, because resource allocation is solely based on time. In contrast, the operators do care about energy. First of all, power limits the size of HPC systems as power distribution is far from trivial [23]. Secondly, power translates into operating costs.

In special-purpose HPC systems as found in commercial settings, the set of applications is limited and possibly even known ahead-of-time. Since end users and operators are much closer, an energy-aware development methodology like the one presented in this thesis might be beneficial: Energy usage of applications directly relates to the profitability of the commercial activity. Additionally, predicting energy usage prior to building the HPC system can lead to significant cost reductions by using an optimised hardware setup.

In fact, this approach is also actively researched in the scientific HPC community, just from a slightly different angle.

First of all, there is simple reduction of operating costs by trying to reduce power of unmodified applications. Dynamic voltage/frequency scaling (DVFS) is a way to reduce power which usually comes at the expense of performance. A smart scheduler (see Section 4.2.4) can avoid this. For example, the SuperMUC-NG HPC system already employs an energy-aware DVFS scheduler that uses performance predictions to save energy without performance impact [22].

A different approach is hardware overprovisioning: Traditional HPC cluster design is usually power overprovisioned, i. e. much of its power distribution capacity is never used in real-world applications. The ratio between average operating power and provisioned worst-case power can be as bad as 60 % in real world clusters, yet there are rare workloads (HPC LINPACK, see Section 4.4) that exploit the worst-case limit [49]. Running at 60% peak power also means that cooling systems and power converters run at less efficient operating points.

In contrast, a hardware overprovisioned cluster has more compute nodes than its power distribution system can power simultaneously. It can then dynamically use as many nodes as the power limit allows, which leads to efficient operation for cooling and power distribution. Since cooling and power distribution are expensive for Megawatt class clusters, this can be economically beneficial [50] while also improving overall performance [49].

Such a hardware platform effectively is a dynamically reconfigurable one. Cluster management software must decide which nodes to use for a given workload – which is a classic allocation optimisation problem. It could be addressed by a prediction-based design space exploration workflow like proposed in this thesis. In fact, existing management software does exactly that, just with massively simplified prediction models. For example, the RMAP resource manager [48] uses a set of instrumented executions in various configurations to build a linear regression model which predict all other configurations.

4.1.5. Heat

All electronic circuits convert the largest part of their supply power into heat. This power can be separated into static and dynamic parts, where dynamic power leakage scales linearly with clock frequency. If heat rises too high, CPUs suffer transient failures or even permanent damage.

While changes in energy consumption can happen in almost arbitrarily short time spans, heat distribution and dissipation is a comparably inert effect. This means that overall temperature depends on the average power over a certain time span; with real-world materials, it is inherently low-pass filtered.

4.1.5.1. Thermal Management

Modern performance-oriented CPUs support clock frequencies that would cause heat failure when used permanently. They assume that the workload is mixed and contains enough idle time so that overall heat production stays within the safe operating range.

The limitations of heat dissipation are so prevalent that modern CPUs don't even advertise their maximum clock speed as nominal clock speed. They can no longer guarantee continuous operation under their maximum clock frequency. Instead they specify a safer, slower speed, and call the ability to exceed this clock depending on circumstances by various marketing terms (e. g. Precision Boost for AMD and Turbo Boost for Intel). CPU vendors for mobile devices sometimes use a different approach and specify a maximum duration for the nominal clock speed. CPUs might also switch to lower than nominal clock speeds if heat rises too high.

Another approach to manage heat is to completely shut off parts of an integrated circuit. Parts of circuits that cannot be powered due to heat or energy constraints are also known as *dark silicon*. Some CPUs temporarily (and repeatedly) suspend the entire chip in order to prevent dangerous heat levels; this is also known as *thermal throttling*.

Finally, CPUs may contain a last-resort protection mechanism that turns the CPU off or resets it if – after employing all previously mentioned mechanisms – temperature still rises to a damaging level.

4.1.5.2. Observable Effects

The net effect is that practically achievable CPU performance depends on the amount of cooling. More efficient cooling means that the CPU spends less time throttling, or at a higher clock frequency.

Another effect of heat is that it affects energy consumption. For metal-oxide semiconductor field effect transistors (MOSFETs) as used inside virtually all digital circuits these days, heat increases their so-called subthreshold power leakage, leading to increased static energy consumption.

Furthermore, real-world electronic components have a nonzero temperature coefficient: their internal resistance changes with temperature. The temperature coefficient of silicon-based components is a complex topic, but most other components always have a positive temperature coefficient, i. e. their resistance increases with heat. For example, if the overall resistance of voltage converter components increases, conversion efficiency drops, and this directly affects overall power consumption. Even the copper PCB traces, which inevitably heat up through heat dissipation from the CPU, will add to the losses.

These effects are big enough that there is a significant overall effect on modern electronic circuits that is easy to observe through measurements.

4.1.5.3. Modelling

Unfortunately, accurate heat modelling requires deep insight into the physical structure of the CPU (see [46] for an example), which is not available for off-the-shelf hardware.

As a substitute, heat effects can be accounted for through controlled circumstances. Heat dissipation from a hot into a cool material is proportional to their temperature difference. Thermal coefficients on the other hand are constant[2]. At some point, a circuit with positive thermal coefficient will reach thermal equilibrium, i. e. a balance between heat generation and dissipation. Under the assumption that dynamic power behaviour doesn't significantly change the thermal equilibrium, modelling can then pretend that there are not thermal effects.

This has the potential to interact in complex ways with the active thermal management mechanism presented above. Clock frequency scaling should behave mostly predictable once thermal equilibrium is reached. However, the more extreme protection mechanisms (throttling, reset) have the potential to disrupt predictability due to their blunt nature. These should be avoided if predictability (or performance, or stability, for that matter) is desired.

4.1.6. CPU-specific Performance Optimisation

Most CPUs include complex techniques to speed up execution. While CPU clock frequency should have direct influence on application performance, bottlenecks of the memory subsystem may lead to CPU cycles spent on waiting for data, for example.

[2] At least within a few 10 K around room temperature. Universally speaking, they are not constant at all, and MOSFETs are special as well, but that doesn't matter here.

With desktop applications, users take these optimisations as they are and simply buy a faster computer if performance is insufficient. If you're working with one of the fastest computers that exist, that is not an easy option[3]. Thus, authors of HPC software usually invest much time in optimising algorithms for these effects, despite the fact that most of the optimisations are time consuming and highly CPU-specific.

Some of these effects are:

Caching Caches keep frequently used memory content in a fast temporary memory area. To profit from this, applications need cache friendly data arrangement. This is one of the most important small scale optimisations in HPC, and one of the most generic ones.

Prefetching If applications know in advance what data they need, they can instruct the CPU to fetch that data early enough so that memory latency is partially or completely hidden. This optimisation usually depends on specifics of CPU and/or cache architecture.

Out-of-order execution The CPU itself might be able to rearrange instructions so that operations that wait for data are set aside. It can then execute instructions that have no unfulfilled data dependencies instead. This is typically not accessible to manual optimisation.

Multiple issue Some CPUs can even execute multiple instructions at the same time, as long as they don't use the same internal resources (FPU, ALU, etc.). Again, this is almost impossible to optimise for.

Speculative execution When there is no better alternative, CPUs might execute instructions speculatively, i. e. without knowing the data they are dependent on. They simply guess, and if the guess is correct, speed is gained. Otherwise, CPU state is reverted without any loss[4]. To optimise for this technique, developers can arrange branches so that the most likely outcome is the one guessed by the CPU.

Simultaneous Multi-Threading (SMT) With SMT, the CPU provides multiple logical CPU cores that share the resources of a single physical core. By executing multiple independent threads tightly interleaved, out-of-order execution and multiple issue optimisations become more effective as the CPU can choose between completely independent instruction streams. Effectively, this is makes multiple issue hardware visible through the threading API.

Write reordering Not just instructions can be reordered. Writes to memory can also benefit from reordering and combining. This is less of an optimisation target and more of a performance impediment to watch out for; it slows down synchronisation between threads. Reordered writes may not become globally visible in the

[3]Just like the embedded domain, but for a totally different reason: upgrading hardware is easy, but cost per unit matters a lot when producing millions of embedded control units.

[4]For the purpose of this thesis, the Spectre and Meltdown class of security vulnerabilities are not relevant. Mitigating them does incur a measurable performance impact.

order they appear in the program, so extra effort must be spent for safe thread synchronisation.

As an overall measure of the availability and effectiveness of these optimisations, the IPC metric (instructions per cycle) describes typical throughput of a CPU.

The methodology proposed in this thesis assumes that task granularity is so large that these effects are not relevant to overall application modelling. However, Section 9.5 shows a simple way to account for the effect of sharing common resources between tasks (cache, SMT).

4.2. Software Aspects

Due to developments in the space of HPC hardware, awareness for energy aspects is increasing. Nevertheless, the usual goal during development of HPC applications is to minimise their overall execution time (called *makespan*), since the result is what the user of an application is interested in most. This is a hard enough problem by itself.

Since HPC platforms these days are highly parallel platforms, one of the most important high-level optimisation challenges is to overcome parallelisation limits inherent in algorithms in general or implementations in particular. And since most HPC problems belong to one of a few generic classes of algorithms, there are also generic strategies to address these challenges. The following subsections give an overview of existing concepts that are related to the methodology proposed in this thesis.

4.2.1. HPC Problem Classes

In order to discuss HPC application optimisation, it is useful to classify them. Many problems have a natural formulation in one of a number of computational patterns (as defined in [12]). Notable problem classes are:

State Space Exploration Many problems require searching a state space for a state that either fulfils some condition or that optimises a criterion (often called *cost function*). Various search strategies exist, either exact methods (backtracking branch and bound, dynamic programming) or probabilistic heuristics (simulated annealing, genetic algorithms, ant colony optimisation).

Linear Algebra Problems that require linear algebra to solve, often revolving around matrix multiplications. This is one of the largest classes of problems; many real-world problems can be reduced to linear algebra, e. g. artificial neural networks. Subclasses are dense and sparse linear algebra, where sparse problems take advantage of the fact that their matrices have many zero elements, eliminating a significant amount of computation.

Mesh Problems that model a system as a mesh of discrete points or volumes. Each mesh node's behaviour depends on its neighbourhood. This is another huge class of

problems, as finite element physics simulations fall into this category. Meshes can be structured (tightly coupled to the geometry they model) or unstructured. Structured meshes can exploit the regularity of their structure to optimise calculations, while unstructured meshes allow higher resolution in areas of interest without forcing the whole mesh to have that high resolution, effectively reducing data size.

N-Body Problems consisting of a system of N members where the behaviour of each member depends on all other members, not just its neighbours. Naïve solutions have a time complexity of $O(N^2)$, which requires approximation schemes for large N. For example, physics simulations that consider long range interactions (like gravity or electromagnetic fields) fall into this category.

This thesis is not restricted to a single problem class, but evaluation is heavily based on linear algebra (see Section 14.2.2).

4.2.2. Parallelism and its Limitations

Since the possible speed of a single CPU is limited by available technology, using many CPUs in parallel is the only way to speed up software further. Of course, applications must be written in a way that allows distribution across multiple CPUs. Due to various effects, application speed does not scale linearly with the number of CPUs.

Most obviously, theoretically achievable speedup depends on the relative amount of work that can be parallelised. Amdahl's Law [16] describes this in a generic way.

Communication is another important aspect that can significantly limit achievable speedup. Assume two processing steps A and B, where B processes data that A generates. If they run on different cluster nodes, communication increases the latency of the combined processing sequence. This can slow down maketime directly.

This does not exclusively apply to inter-node communication. Node-local communication can result in memory copies and cache pollution, adding measurable latency as well.

The distributed case, however, allows an easy improvement: Interleave multiple processing sequences so that some independent computation C is performed while B waits for data. This is sometimes called *communication hiding* and is an important optimisation for distributed computing.

If there is no other computation that the node of B can perform during communication, computing resources go unused and the actually achieved parallelisation is less than the theoretical maximum. *Utilisation* expresses this as the ratio of used over available computing resources.

Some problems are very prone to such effects. For example, mesh and N-body problems need bidirectional exchange between elements after every simulation time step. This tight coupling makes communication hiding difficult. More generally speaking, any kind of global synchronisation has the potential to severely limit utilisation.

As a result, communication patterns are an important optimisation target. For regularly structured problems on mostly homogeneous hardware, efficient communication

strategies are easy to implement using established HPC middleware like Message Passing Interface (MPI). On platforms with higher degrees of heterogeneity or with less regularly structured problems, maximising utilisation is difficult[5].

In fact, communication in such complex situations is a major aspect of the methodology proposed in this thesis.

4.2.3. Application Patterns

Scheduling parallel systems is a varied, well established research field [53]. To make matters more complex, application design influences which scheduling algorithms are even applicable. For example, dynamic task parallelism cannot be scheduled ahead-of-time.

Therefore it is useful to classify parallel applications through design patterns. From the classification in [12], some notable patterns for HPC problems are structural and parallel algorithm strategy patterns.

4.2.3.1. Structural Patterns

describe high level software architecture, mainly concerning data flow and only to a lesser extent control flow. Examples are:

Pipe/Filter Streaming data is processed by filters (computation) that are connected by pipes (communication). Filters are assumed to be stateless, which makes the whole setup easily parallelisable.

Map/Reduce Independent data sets are first transformed individually in a *map* operation, then summarised in a *reduce* operation. Due to the independence of the data sets, he *map* step is easy to parallelise, while the *reduce* step may or may not be parallelisable. The latter often uses a binary partitioning approach.

Iterative Refinement A data set represents some initial state. The program is repeatedly applied to the data, creating an increasingly accurate approximation of the desired solution. Iteration stops when some terminating condition is met.

In the case of mesh and N-body problems, iterations usually represent time steps, and the terminating condition is either a desired time span or the detection of a stable state.

Arbitrary Task Graph This is a generalisation of the other design patterns. The task graph expresses how different parts of the application (the *tasks*) interact with each other. There are many variants; some are pure data flow graphs (like used in this thesis, see Section 9.2.1), some are control flow graphs, or a combination of these, e. g. [10]. Likewise, there are various semantics for tasks.

[5] Probably an NP-hard problem, although the claim is too vague for a definite proof.

4.2.3.2. Parallel Algorithm Strategy Patterns

describe how the parallelisation of a software architecture is organised. As a result, these patterns have a high impact on how work can be distributed and scheduled across a cluster. Examples are:

Task Parallelism The algorithm is decomposed into a collection of *tasks*, and dependencies between tasks are known. Except for these dependencies, execution order is unspecified and does not affect the final result. If tasks are completely independent, the pattern is also known as *embarrassingly parallel*.

Task parallelism can be static or dynamic. In the dynamic case, running tasks spawn new tasks on demand, while otherwise the set of tasks is known in advance. Dynamic task parallelism is especially suitable for dynamic programming problems.

Pipeline This is a specialisation of task parallelism: a sequence of dependent tasks executes on an independent set of data elements. While a single data element needs to visit each task in sequence, each task can work on a different data element at the same time. Stream data processing (audio/video) often uses a pipeline architecture. If task execution times are data independent, scheduling a pipeline is much easier than scheduling generic task parallelism.

Data Parallelism With this pattern, the same operation is applied to multiple data sets at the same time. This is most useful for node-local parallelisation using CPU vector instructions (SIMD – single instruction multiple data), which process a fixed number of data streams in parallel. This is also a useful pattern for GPGPU acceleration.

In the distributed case, data parallelism mostly results in an embarrassingly parallel problem. There are exceptions: The reduction step of the Map/Reduce structural pattern also counts as data parallelism but isn't embarrassingly parallel.

Discrete Event This pattern is based on a collection of tasks that interact in unpredictable (or at least highly irregular) intervals. It is popular in the embedded domain as a simulation technique for intrinsically parallel hardware [7], and in modern industry automation [5]. It is uncommon in HPC because it needs frequent synchronisation between cluster nodes, which reduces performance significantly (see above).

In this thesis, the focus lies on task parallel programs expressed as static task graphs (see Section 9.2.1).

4.2.4. Scheduling

Static task graphs are a fairly generic model of computation that allow a wide variety of execution strategies. Furthermore, both dynamic (event driven) and static scheduling approaches apply.

In [53], the authors developed a detailed taxonomy of scheduling algorithms. In the area of modern high performance computing, a few classes are notable:

4.2.4.1. Dynamic Scheduling

Dynamic scheduling in a cluster is more or less equivalent to load balancing when applied to embarrassingly parallel problems on homogeneous platforms. Since there are no dependencies between tasks, overall execution speed is merely a matter of keeping execution units busy. Dynamic scheduling can be *centralised* (doesn't scale well; e. g. [41]), *hierarchical* (scales better, but has problems beyond 10000 cores; e. g.[42]), or fully *distributed*.

The most popular fully distributed dynamic parallel scheduling method is *work stealing* [54], which has evolved into an entire class of schedulers. It works by managing local queues of tasks to execute. If an execution unit is idle it will execute the next task from the local task queue. If the queue is empty the node will remove a task from a random other node's task queue and add it to its own queue. Work stealing provides upper bounds in space and time that are within a constant factor of optimum values [55]. It has been shown that it also scales well in practice [61].

The complementary class to work stealing is called *work sharing*. In this class, local or global schedulers push tasks out to execution units instead of execution units pulling tasks. Centralised and hierarchical schedulers are usually work sharing, although it is possible to implement work stealing on top of a hierarchical structure [56].

4.2.4.2. Static Scheduling

Besides fully dynamic scheduling there is static scheduling. There, all cluster nodes have a list of tasks to execute, and possibly also a time when to execute them. The latter part is common in embedded time triggered architectures. Due to the synchronisation difficulties in large cluster architectures, they usually use some kind of dependency management instead of fixed execution times, i. e. tasks start as soon as their dependencies are fulfilled.

While dynamic scheduling usually doesn't even know the exact workload, static schedulers can reason about the entire application and thus (in theory) can calculate an optimal schedule. For the general case this is an NP-hard problem, however, so a multitude of approximations and heuristics exist.

Furthermore, schedule quality depends on the resource model used for predicting execution and communication time. On the other hand, schedules can easily take into account other metrics like energy (as done in this thesis), as this does not change the fundamental nature of the optimisation problem, only its size.

4.2.4.3. Rescheduling

Finally, there is the approach of periodic rescheduling as a way to make static schedulers able to react to changes in applications or platforms. It uses a static scheduler that is

invoked in response to some event (application changes, platform changes, or simply at regular intervals).

This allows a static scheduler to reevaluate its choices to compensate for unpredictable developments during execution. Most importantly, it allows compensating for incorrect timing predictions due to insufficient resource models.

The main problem with this approach is of course the processing time the scheduler needs. If the scheduler needs more resources than an updated schedule can save, its execution is pointless. If the latency of the scheduler results in long idle times for execution units, its execution is actively detrimental. Also, publishing a new schedule needs some kind of synchronisation, which further hurts performance.

This thesis assumes a fully static scheduler and proposes an improved scheduler with $O(n)$ time complexity as a secondary contribution (see Section 9.6). Rescheduling could be introduced in the future.

4.2.5. Resource Modelling

All task schedulers have in common that they need some kind of model that assigns a cost to each decision. For dynamic schedulers, this is often as simple as a bimodal model where idle processing units have high costs and busy ones have low costs assigned; there is no distinction made between different tasks. In other words, they use utilisation as their main metric. Unused processing capacity is avoided under the assumption that this will optimise application makespan.

Since dynamic schedulers react on observed task performance, this assumption works well, as shown in [55].

This is not generally true for heterogeneous platforms. For example, an idling general purpose processing unit may better be left idling if a suitable highly efficient unit will become idle within a short time span. Furthermore, optimising for other metrics than makespan is difficult. This is where static schedulers have an advantage.

As mentioned before, static schedulers can exploit more detailed resource models, but creating them is difficult. On smaller platforms, various approaches work: register transfer level (RTL) simulations, analytical models provided by CPU manufacturers, or plain measurements. However, HPC platforms usually don't have enough idle time for measurements (assuming they even have measurement infrastructure), don't come with manufacturer-supplied models, and use CPUs that are too complex for RTL simulation.

Many static scheduling algorithms use a simple weighted approach in which tasks and dependencies have abstract weights assigned to them that represent relative resource usage. Instead of providing measurements, weights can be assigned through extrapolation of previous observations (i. e. guessing), expert estimation (i. e. guessing), or just by guessing.

Abstract weights don't solve the problem of how to systematically determine HPC application resource usage. Furthermore, modelling overall execution behaviour also depends on correctly modelling the run time system used to execute the application. The

communication and execution behaviour of code-centric[6] middleware like the popular MPI is difficult to model in a generic way. These modelling difficulties may be some of the reasons why dynamic scheduling is so dominant in the HPC domain.

Nevertheless, there are some options to create a suitable resource model. There are HPC platforms with integrated power measurement infrastructure that even provide out-of-band data transmission [14], thus allowing noninvasive task measurement. It may then be feasible to allocate a very small subset of the cluster to model building without significantly affecting production use.

Another solution would be if the cluster was modular so that users can build a dedicated measurement platform based on a small number of production modules with added measurement infrastructure.

The methodology proposed in this thesis works with either kind of representative measurement platform (see Section 10).

4.2.6. Auxiliary Aspects

During production use, there are some additional restrictions and practical difficulties. While these are out of scope for this work, the chosen task graph application model does allow for them, increasing its usefulness and relevance.

4.2.6.1. Data Representation

With heterogeneous systems, data representations may vary. For example, floating-point numbers may have different bit pattern representations on different CPU architectures, or integer numbers may have different byte order or sign representations.

For simplicity, this thesis assumes a common wire format for all data. Required data conversion is accounted for as part of the computing time of those tasks that need it.

4.2.6.2. Data Distribution

In large applications, it may be a challenge to get data to the place it is needed in time, especially if certain data is used in widely separated locations. Scheduling should try to hide delays caused by this. An additional degree of freedom is gained if the scheduler supports *task cloning*:

Assume there are four tasks A, B, C, D and task dependencies $A \to B \to \{C, D\}$, where C and D are executed on different cluster nodes and $B \to \{C, D\}$ requires significant unhideable transmission time. If transmission size of $A \to B$ is much smaller than $B \to \{C, D\}$, and execution time of B is small enough, it may actually be faster to clone task B (creating task B'), and to execute $B \to C$ on one node, and $B' \to D$ on another. This replaces the transmission time of $B \to \{C, D\}$ with $A \to \{B, B'\}$ plus the extra CPU time spent on B'.

[6] as opposed to model based approaches

4.2.6.3. Fault Tolerance

In the real world, systems may fail. Cluster nodes may vanish at any time, e. g. due to hardware defects. When this happens, a significant amount of work may be lost. A popular approach for this is snapshotting, i. e. saving of intermediate application states.

With task graphs, snapshots can actually be represented as a special kind of task that just saves its input to persistent storage (and passes it through unmodified). This way, snapshotting is a local operation that does not require global synchronisation. Snapshot tasks are inserted into task graphs according to external reliability requirements. To resume execution, the task graph is reduced by checking which snapshots exist and pruning all predecessor nodes that don't have any successors without snapshots. The pruned task graph is then scheduled as usual.

4.3. Parallelisation Tools

Parallel programs face three practical issues beyond the actual work they try to perform:

- They need to start processes on different cluster nodes. This requires transfer of the required executables and possibly initial data as well.
- Those processes need to communicate with each other.
- They probably also want to exploit local parallelism in the form of multiple CPU cores or dedicated accelerator hardware.

Some programs might not need special tool support for this. For example, deployment of an application across the cluster might happen through standard operating system (OS) tools like networked filesystems and SSH. Problems that are embarrassingly parallel in their entirety are efficient to run just by starting multiple processes with no communication between them. This addresses local parallelism as well as distributed computing.

4.3.1. Local Parallelism

A single HPC cluster node already offers multiple forms of parallelism. At the very least, there are multiple CPU cores with data-parallel vector instructions (SIMD – single instruction multiple data). GPUs can be used as math accelerators, and then there are more exotic accelerators like FPGAs, dedicated artificial intelligence accelerators (e. g. the Intel Myriad coprocessors), or manycore coprocessors like the Intel Xeon Phi.

The definition of what parts actually count as a single local node is twofold: The most important property is that of shared memory. All computational units that can access the primary RAM can exchange data very efficiently. Some accelerators might not have direct memory access at all, but need explicit data transfer from the CPU. Others might have non-uniform memory access characteristics, with fast local RAM and slower access to CPU memory.

For the purposes of this thesis, a single node consists of all processing units that cannot enter global communication on their own, plus the main CPU that performs communication on their behalf. An equivalent definition for the scope of this thesis is that a single node is running exactly one instance of a general purpose operating system.

To exploit node-local parallelism, applications might simply use operating system threads (POSIX / Win32 threads), handwritten SIMD instructions and APIs provided by accelerator vendors. But this is tedious, error-prone, and often not portable.

Various libraries exist that offer abstractions to hide low level optimisations. For example, the basic linear algebra subprograms (BLAS) specification has multiple optimised implementations. Among them are ATLAS with a portable automatic optimisation strategy, OpenBLAS with hand-optimisations for many CPUs, and cuBLAS/nvblas using NVIDIA GPU acceleration.

Such libraries work well if they contain the required algorithms. If not, libraries and language extensions exist for higher level parallelisation. The most prominent C/C++ language extension for parallelism is probably OpenMP [19]. It works through annotations (compiler pragmas) attached to serial code. These annotations tell the OpenMP runtime in what way work can be parallelised. They can express data parallelism (vectorisation), task parallelism, and many other parallel patterns.

OmpSs [20] is an extension to OpenMP that tries to integrate OpenMP with GPUs, FPGAs, and other accelerators. StarPU [21] is another library with similar goals. Both use task parallelism as their primary pattern.

The proposed methodology of this thesis models multiple CPU cores as separate execution resources, so it manages multi-threading internally (via OS threads). Accelerators are not addressed at all, but they offer opportunities for future work. Instruction level parallelisation and optimisation is out of scope; tasks may use vector instructions or libraries at will. In fact, the primary evaluation example consists of BLAS routines (see Section 14.2.2).

4.3.2. Communication Middleware

Coordinated work across multiple cluster nodes needs communication. Applications could simply use operating system networking APIs, which most probably means something derived from the BSD socket interface. For advanced communication patterns and deployment strategies, it is more efficient to use specialised libraries and tools.

For inter-node communication, there are several popular choices beyond low-level socket programming. Some also address deployment:

PVM (Parallel Virtual Machine) is one of the oldest projects for cluster computing. It is a fairly comprehensive solution that encompasses deployment, communication, and even data conversion in a heterogeneous cluster and across different programming languages [17]. Furthermore, PVM supports dynamic changes in the cluster setup including fault tolerance.

MPI (Message Passing Interface) is almost as old as PVM and still actively developed. Being a vendor driven standard, it focuses on high performance instead of maximum

interoperability. It provides a rich set of communication patterns (point-to-point, group communication, asynchronous opportunistic messaging, etc.) and topologies. MPI only has limited support for deployment or interoperability. It is the most widely used HPC middleware, thus high level job schedulers often have dedicated MPI support, and it is often used in conjunction with OpenMP.

RPC Interfaces (Java RMI, CORBA, Sun RPC, etc.) These interfaces transparently map local function calls to network communication. They are usually tied to object oriented programming languages and try to hide the distributed nature of the application behind a remote object abstraction. In practice, this abstraction has shown to be too widely separated from the network layer and its failure modes [18]. These interfaces provide data conversion in heterogeneous environments, but do not address deployment.

Ømq ZeroMQ is a fairly new middleware that has many similarities to MPI. Like MPI, it supports a wide variety of communication patterns. Unlike MPI, it emphasises asynchronous communication. Furthermore, its communication primitives are optimised for heterogeneous communication technologies, including an efficient node-local (shared memory) transport. Data format and language interoperability is integral part of ZeroMQ, deployment is not addressed.

All these middlewares have in common that distributed modelling is still fairly low-level: Two different programs on different cluster nodes send and receive data. Even though they are part of a single operation, a *send* primitive and its corresponding *receive* primitive are completely separated in the code. Some patterns express more integrated communication, but overall functionality is still modelled as multiple distinct processes.

The task graph executor used in the evaluation of this thesis doesn't use one of these, it only needs OS facilities (see Section 9.8).

4.3.3. Deployment

For deployment, HPC machines usually have a standalone high level job scheduler. It performs resource allocation and load balancing on the application level. One of the most popular examples is the Slurm Workload Manager. It can deploy and monitor multiple independent applications on different or even overlapping subsets of cluster nodes while improving communication by optimising application locality [57].

This thesis' evaluation did not use any specialised deployment tool. OS facilities were sufficient since access to the evaluation testbed was fully controlled (see Section 10).

4.4. Common Benchmarks

LINPACK For HPC systems, the stereotypical benchmark is LINPACK [25]. Due to its historically long and widespread use, it is the de-facto standard metric to compare HPC systems of the past four decades. The actual metric that is used in comparisons is

floating-point operations per second (FLOP/s). This is also the metric used in the Top500 ranking of supercomputers as it is an indicator of achievable computation speed. In contrast, theoretical maximum speed (derived from clock frequency and assuming ideal vectorisation/code arrangement) is almost never actually attainable due to the many bottlenecks in modern systems (see Section 4.1.6).

Like all benchmarks, LINPACK is not representative for all workloads. Its strength lies in the comparability across many generations of computers. A big weakness is that it originates in an era of shared-memory HPC systems, i. e. single nodes that don't need networking. This is why derivatives exist, most prominently Highly-Parallel LINPACK (HPL) for distributed-memory cluster architectures. Another degree of variation deals with problem size: 100×100 element matrices were a challenge in 1979, but are trivial today. With HPL, 10000×10000 is an equally valid benchmark. Multiple sizes may be measured, and the highest observed result is reported. Since results are always normalised to FLOP/s, this does not significantly impact comparability.

LINPACK and its variants essentially solve a linear system of n equations $Ax = b$ for x, where $A \in \mathbb{R}^{n \times n}$ and $x, b \in \mathbb{R}^n$. The original LINPACK enforced a specific sequence of vector operations for this. HPL allows any implementation, but the total number of floating-point operations is assumed to be $\frac{2}{3}n^3 + 2n^2$ regardless of the algorithm that is actually executed. This gives implementors enough freedom to optimise for unusual architectures. It also resolves the difficulties in counting modern floating-point operations like fused multiply-add (FMA), which almost – but not exactly – correspond to two traditional operations.

Cholesky Matrix Decomposition A conforming HPL implementation has additional restrictions that aim to improve comparability but result in additional implementation effort. Most algorithms use matrix factorisation followed by triangular solving, where factorisation represents the main amount of work. Benchmarking just that is still a strong indicator for LINPACK performance but is much faster to implement and analyse. Cholesky and LU (lower/upper) decomposition are both popular benchmarks for this reason. This is also one of the reasons why Cholesky is the running example in most parts of this thesis.

SPEC Benchmarks The Standard Performance Evaluation Corporation is a non-profit corporation that publishes various benchmarks to compare different aspects of computing systems. They have HPC benchmarks, but focus on shared-memory nodes, i. e. not cluster architectures. Unlike LINPACK, they regularly publish changed/updated benchmark suites. This way, benchmarks allow better evaluation of new computational hardware features, but long term comparability is much reduced.

PARSEC Benchmark Suite The PARSEC benchmark suite [26] (not to be confused with the PaRSEC parallel programming model, which also relates to the HPC world) is a diverse collection of real-world programs for benchmarking shared-memory computers. They represent computationally intense tasks from a wide variety of application fields,

not just traditional HPC problems; for example, one benchmark is based on the x264 video encoder. Overall, problems represent more modern workloads than comparable previous benchmarks [27]. Since these benchmarks are real-world programs, distributed benchmarks use MPI as middleware.

PARSECSs Benchmark Suite PARSECSs[16] is a subset of the PARSEC benchmarks ported to a task parallel execution model using OmpSs (see Section 4.3.1). After the transformation, some applications yield very irregular structure that was not adequately addressed in the original thread-based implementation. As a result, benchmark makespan improved by 13% on average while also reducing code complexity. Even though OmpSs tasks are not exactly the same as the tasks used in this thesis, this shows that the task paradigm can outperform established parallelisation methods.

Single Application Benchmarks There is a wide variety of benchmarks that consist of a single HPC application from some important subdomain. These usually consist of code that is highly optimised for the middleware in use (usually OpenMP+MPI) and thus is difficult to port to other parallelism paradigms. These are more suitable to perform comparisons between related cluster platforms or middleware implementations.

4.5. Common Prediction Approaches

With the amount of applications, middlewares, parallelisation strategies, and hardware platforms, and due to the size of data sets and hardware platforms, predicting application performance is difficult in HPC. As mentioned in Section 4.2.5, HPC systems often do not allow iterative performance optimisation based on test suites and/or profile generation. Nevertheless, programmers need metrics to optimise application performance, just like future operators of HPC systems want to optimise their machines for expected workloads. Since the main contribution of this thesis is a new prediction approach, Section 8 lists current practical examples of these approaches.

4.5.1. Analytical Models

Simple analytical models can already give a lot of insight. In fact, Amdahl's Law [16] itself is such a model. It only returns an abstract measure, but one that is suitable for certain application optimisation decisions. More complex models may involve linear regression or integer linear programming. They have in common that they are comparatively fast.

However, modern hardware architectures contain optimisations that have nonlinear behaviour (see Section 4.1.6), and asynchronous network communication adds highly irregular behaviour on top. As a result, analytical models are easier to apply to limited issues within a cluster instead of an HPC system as a whole.

For whole system modelling, analytical models might extrapolate from measurements by using statistical methods (e. g. curve fitting or histograms). These can work well

as long as all significant boundary conditions are known. Thus the main challenge for statistically derived models is to identify such boundary conditions that limit the applicability of the model. For example, a model that predicts performance for high numbers of CPU cores based on linearly fitted measurements on fewer cores might become invalid once RAM bandwidth is saturated.

4.5.2. Simulation

A simulation model is composed of chosen elements of the modelled system, and elements operate at a chosen abstraction level. The behaviour of each simulation element then leads to a change in system state over time. Simulation adds overhead over native execution. The abstractions represented by the simulation model determine accuracy and speed of the simulation.

For example, a CPU in an HPC cluster might be represented by a register transfer level simulation of the digital circuit, an instruction set simulator, or by application code compiled for the simulation host machine. The latter only models functional behaviour, the second version is slower but introduces some degree of timing behaviour, while the first one results in clock cycle accurate behaviour of all I/O signals at greatly reduced speed.

Abstraction decisions may depend on abstractions chosen in other parts of the simulation. For example, simulations might or might not include an instruction cache model. Including it improves timing accuracy but needs to know binary code layout on the target platform; it can't work with host-based executables.

A big advantage of simulation over analytical modelling is that it is much easier to vary abstraction levels between different parts of the system. For example, users could combine instruction set simulators for some area of interest while using host-based execution for all other simulated CPUs in order to provide a fast but functionally accurate environment.

Simulation is popular in the embedded domain, where simulated machines usually are much slower that simulation hosts. In such setups, it is easy to achieve parity between simulation time and simulated time. For HPC, the situation is reversed: simulated systems are much bigger than simulation hosts, even when using massively parallel simulation. Existing simulators (see Section 8) often focus on a specific issue and use very simple modelling for all other parts of the system.

4.5.3. Symbolic Simulation

A special subclass of simulation is *symbolic simulation*. In these, simulation models do not accumulate concrete values of time or other properties. Instead, they record symbolic variables of these processes that can represent multiple states or entire ranges of values.

For example, instead of modelling the time required for each floating-point operation in a CPU, a symbolic simulation might only record the number of operations. This could

then represent an analytical performance model: dividing the number of operations by a given target machine's LINPACK FLOP/s value results in a rough makespan prediction[7].

An analytical model derived this way has the advantage that the abstraction choices of the simulation model are preserved. The practical disadvantage is that symbolic variables represent state that might not be composable in a simple way. Linear execution of machine instructions might allow linear combination of symbolic variables. Parallel processes on the other hand result in a superposition of states; if these are not reduced in some way, the symbolic state space grows exponentially and simulation becomes impractical.

4.5.4. Trace-Based Simulation

A related class of simulations is trace-based simulation. It consists of two steps: First, the application is executed on a regular HPC system while recording events of interest (the trace). Event abstraction level ranges from as low-level as individual memory read/write instructions up to communication transactions using MPI semantics. This approach has a very practical advantage: it works with unmodified existing applications; tracing instrumentation can be added automatically through MPI facilities.

In the second step, the trace is replayed on a simulation model of a different machine. Computation speed and communication latency for the simulated machine is derived from on trace contents. Computation models might be as simple as a linear speedup factor, while communication models tend to be more complex (e. g. in [58]).

Tracing imposes an important limitation: a trace must exhibit the same amount of parallelism as the target machine. To simulate a 10000-core machine, a trace from a different 10000-core machine is needed.

Logical transactions might be recorded as their constituent low-level communication primitives. If that is the case the causal relationship between them is lost and simulations might produce incorrect timings. Generally speaking, the main challenge for trace-based simulations is to make sure that a trace is actually portable between tracing machine and target machine.

The second limitation concerns trace acquisition itself. It is usually invasive, i. e. it affects execution behaviour of the traced application. In literature, 5-10% performance loss is reported [58]. This creates the question if the trace itself is an accurate representation of application behaviour.

For trace recording, the application must already be in a runnable state. This may sound obvious, but it precludes feedback during the initial development process. As a workaround, a skeleton application can generate synthetic traces (e. g. in [60]). Such traces have less timing accuracy than recorded traces but may still give useful insight, especially for high-level application structuring.

Parts of the methodology proposed in this thesis have similarities to synthetic trace-based simulation.

---

[7] Assuming a lot of things that most probably are not actually true.

5. Time and Energy Measurement

Models that predict application makespan for a given target platform need some timing input to base their predictions on. To the best of my knowledge, there is no CPU clock cycle accurate reference simulator (or one approximating this accuracy) for HPC platforms, hence physical measurement is about the only choice to reliably assess the quality of performance models. For energy models, similar arguments apply.

Measurement is surprisingly difficult. The main issues are:

1. Collect measurement data without interfering with the system being measured.

2. Find out what physical effects there are that can and/or should be measured. That's an input to model building: the decision of what should be modelled is influenced by what can be measured.

3. Measure *correctly.*

5.1. Data Collection

Two main styles for data collection exist:

In-band means measurement data is collected by the monitored CPU and/or transmitted over the target network. This modifies time and energy properties of the target system. If this interference is predictable, it can be accounted for after the fact. For platforms with complex and/or undocumented microarchitectural details (see Section 4.1.6), this is not feasible.

Out-of-band means data is acquired by some external mechanism and transmitted/collected over a separate communication path. Implemented correctly, this does not interfere with target system behaviour.

Experience from trace-based prediction methodologies (see Section 4.5.4) suggests that performance overhead is 5-10% when capturing time traces. Power or energy measurements would double the amount of data.

With energy, the impact is actually much worse: Time tracing only needs to take a time stamp at predetermined events. Power tracing needs to take continuous measurements and average (for power) or integrate (for energy) them. The performance impact is likely significant. As a result, any measurement infrastructure needs to be out-of-band.

5.2. Time Measurement

5.2.1. Eligible Effects

Measuring the overall makespan of an application is fairly easy. There is just one effect, the passage of physical time. This is suitable for evaluation but helps little when building performance models.

When building those, there are lots of detailed effects that could be of interest. Related to the generic overview in Section 4.1.6, modern CPUs offer a catalogue of performance events that can be measured, sometimes down to the level of individual instructions. These are based on microarchitectural features and thus vary widely between CPUs.

Various approaches (including the methodology proposed in this thesis) opt for a middle ground: they are interested in net execution time for a fragment of application code. Obviously, the larger code fragments and their data sets are, the less their timing varies due to microarchitectural features. This means that a simple measurement captures all of these effects; they need not be modelled individually. The downside is that longer code fragments may introduce data-dependent control flow variations, but that's a problem of modelling, not measurement.

5.2.2. Measurement Techniques

For overall makespan, it's sufficient to take time stamps at start and end. Assuming HPC scale, the few milliseconds drift of an internet-synchronised real-time clock don't matter. On the other hand, at this time scale there could be time zone changes or leap seconds. Time stamps should therefore be in International Atomic Time (TAI), since it most closely represents physical passage of time without regional or astronomical adjustments.

5.2.2.1. Internal Measurement

For measuring intervals at the granularity of single functions, CPUs nowadays offer a high precision timer. The exact details vary between CPUs, but every relevant CPU does have a monotonically increasing high resolution clock. The monotonic property means that it is unaffected by manual changes of system time as might happen with daylight savings time, for example.

For even finer granularity down to microarchitectural effects, modern CPUs offer event counters that capture performance-related events, e. g. the number of clock cycles spent waiting for data that was not in cache memory. Since modelling these effects is extremely difficult, these performance event counters are not useful in the context of this thesis.

5.2.2.2. Comparability

Monotonicity of a time source does not preclude clock adjustment and synchronisation. Since cluster nodes run asynchronously, measurements from one machine probably

don't exactly match measurements on another machine. Clock synchronisation is a way to address this problem.

For example, the network time protocol (NTP) process will synchronise system time to a master clock. On the Linux operating system it uses a system call that slightly slows down or speeds up the system timer[1]. Observed timer values will always be monotonically increasing despite the fact that these values are synchronised to an external source.

NTP synchronisation offset between machines is in the millisecond range. For smaller measurements, such synchronisation might not be useful. There is also the precision time protocol (PTP, IEEE 1588) that is part of the IEEE Time Sensitive Networking (TSN) collection of standards. In ideal conditions it can achieve synchronisation offsets of less than a microsecond. Both make assumptions about communication latency that might not apply to heavily loaded networks.

5.2.2.3. External Measurement

As a result, it might be more beneficial to account for timing differences between cluster machines in some other way. An effective alternative is centralised measurement by an external mechanism. For example, a dedicated measurement system could be connected to a binary general purpose input/output (GPIO) signal on each node. The software on each node can then just toggle that GPIO signal, and the central measurement system records time stamps of all bit flip events. GPIO toggling can be very fast, sometimes a single machine instruction suffices. On the other hand, this mechanism does not allow to distinguish different event types, and events from multiple CPU cores may interfere with each other.

If there are no true GPIO signals accessible, other output signals might be available that allow software control with low latency. It might even be possible to determine timing purely by observing physical properties of the system during normal operation, like energy consumption. This is what this thesis uses as auxiliary time source.

The big advantage of centralised external time measurement is that all time stamps have the same time base and thus are comparable to each other. Measurement accuracy is independent of CPU model or speed. This advantage comes at the price of additional effort required to relate captured events to their originating events in the measured program.

5.3. Energy Measurement

5.3.1. Eligible Effects

Current hardware offers solid timing measurement for many effects. On the other hand, built-in energy measurement is often completely absent. The first question therefore is to determine what can be measured at all.

[1] Unless the difference was too high. Initial synchronisation may lead to a non-monotonous time step.

On embedded development boards there sometimes are external energy measurement circuits, but not usually on production scale HPC systems. One example for an HPC system with measurement hardware is the RECS|Box system [14]. It contains measurement circuits that monitor supply voltage and current of the individual nodes.

Generally speaking, measuring power supply of entire circuit boards should always be possible. However, there are several individual components that are desirable for model building:

- Entire processing units (models for CPUs or GPUs)
- Individual compute units that are part of a CPU (models for a single CPU core, different CPU microarchitectures in a single CPU, internal GPU blocks)
- RAM (potentially significant secondary energy for computation)
- Ethernet transceivers (models for communication energy)
- External chips in general (models for peripheral functions)
- The power supply itself (losses in voltage converters)

Below that granularity, models might want to use measurements of individual CPU instructions or functions of circuits. The smaller the granularity, the less likely these effects can be measured directly.

A second dimension of effects are different power states. In the most simple case, components have an idle power consumption that always occurs while they are powered, even when inactive. To make this orthogonal to the effects listed above, the latter can be expressed as power difference to idle power. CPUs with DVFS might have many more different operating modes, and usually there is a nonlinear relationship between operating modes and power[2].

5.3.2. Measurement

Energy measurement facilities are not a common part of computing systems. They do exist, but availability and suitability is insufficient for systematic modelling and evaluation purposes.

Power measurement works by measuring voltage and current: $P = U \cdot I$, where U is the measured voltage in Volts, and I is the measured current in Amperes, resulting in electrical power P in Watts. In practice, current measurement works by measuring voltage drop across a shunt resistor, as shown in Figure 5.1. After applying Ohm's Law ($U = R \cdot I$), this results in $P = U_{\mathsf{Device}} \cdot \frac{U_{\mathsf{Shunt}}}{R_{\mathsf{Shunt}}}$. To get electrical energy, multiply average power within a time interval by its duration: $E = P_{\mathsf{average}} \cdot t$ (to be precise, $E = \int P \cdot \mathrm{d}t$). Since this thesis assumes parallel time and power measurement, either average power or energy is sufficient. The other property can always be derived using timing data.

[2] Most importantly because power scales quadratically with voltage.

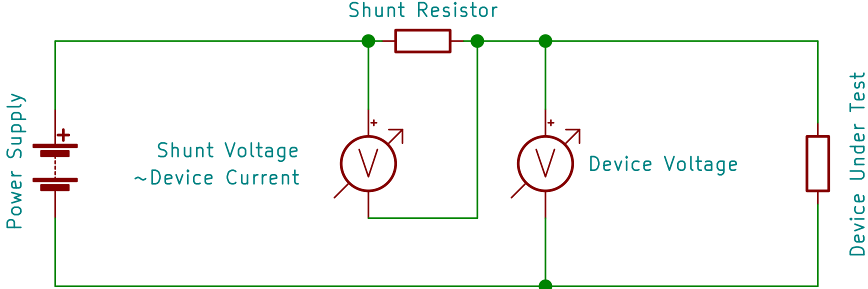


Figure 5.1.: Basic electrical power measurement circuit.

The most important observation from the physical aspects is that some sort of measurement resistance (the shunt resistor in Figure 5.1) is required in series with the device being measured. All measurement circuits, no matter if built from discrete components or using an integrated circuit (IC), work this way[3].

5.3.2.1. Microarchitectural Power Estimation

For CPUs that have them, it is possible to build power models through the use of performance event counters [28]. The Intel Power Gadgets software also supports the integrated GPU. Of course, this is only an estimation and can only cover the CPU package itself, not that of external peripherals. On the other hand, it requires no extra measurement hardware.

The Running Average Power Limit (RAPL) feature present in recent Intel CPUs allows direct measurement (and control) of processor and RAM power with decent accuracy, despite also being based on CPU activity counters [51]. Its impact on system performance is small. Within its accuracy limitations, this approach has been shown to work in an HPC context [48]. It might be an acceptable substitute if the physical measurement approach proposed in this thesis is not possible, e. g. for existing clusters.

5.3.2.2. Power Management Circuits

On modern systems, power supply is usually managed through one or more dedicated power management integrated circuits (PMIC). These contain multiple output power rails with different voltages for different system components.

Some have current sensors that are readable through a communication interface like Inter-IC (I^2C) or Serial Peripheral Interface (SPI). Availability, accuracy, and time resolution of these vary wildly.

Even if they are present for a given target platform measurements might not be useful for model building. For one, power rails are often shared between unrelated components.

[3]There is an alternate way of measuring current using the electromagnetic effect of moving charges, but this has no relevance at the scale being discussed here.

Furthermore, their main purpose is power management at the level of fault detection and protection; another use is battery charge control. Both use cases do not require high time resolutions. Fault detection/protection doesn't even need that much accuracy.

5.3.2.3. Dedicated Measurement Circuits

As a result, the most reliable measurement infrastructure is one that specifically exists for the purpose of monitoring system power at a given level of detail. Unfortunately, even those few systems that include such measurement circuits still fall short of the capabilities required by this thesis.

Two examples of systems that do have such an infrastructure are the modular RECS|Box HPC systems, and the ODROID XU3 single-board computers (SBCs). The RECS|Box systems have current and voltage measurement circuits on each node's printed circuit board (PCB). They also have an administrative Ethernet network that is separate from the main network connectivity so that measurements can be collected out-of-band.

The ODROID XU3 single-board computer uses the INA219 power measurement ICs from Texas Instruments for the power supply of four main components: the four ARM Cortex A15 CPU cores, the four ARM Cortex A7 CPU cores, the GPU, and the RAM. The INA219 offers an analogue bandwidth of 1 kHz and an ADC sample rate of up to 10 kHz. Data collection must be done by the main CPU, i.e. there is no out-of-band access.

Both systems have in common that their effective time resolution is only about 1 ms. A further disadvantage is that they are not representative for the majority of systems.

5.3.2.4. External Measurement

Therefore the only universal solution is a dedicated external measurement infrastructure. The downside is that only power supply lines external to the CPU can be measured. Given that the previously mentioned solutions provide no substantially finer granularity, this is not really a limitation. The real challenge is to access the desired power rails.

Voltage measurement is the easy part: The measurement device just needs electrical contact to ground potential and the desired power rail. There is a good chance that somewhere on the printed circuit boards (PCBs) of the target system is a place where a wire can be attached (possibly soldered) to the desired signal.

The true limitation is current measurement. As shown in Figure 5.1, a shunt resistance needs to be added to an existing power supply line. This is potentially destructive and might not be possible in practice. In current PCB designs, the main power rails exist as buried copper layers in a multi-layered PCB. Since modern CPUs use ball grid array packages, the electrical contacts are located below the CPU itself, and there is no exposed place at all that lies between power rail and CPU.

This means that external power measurement is unable to reach the same granularity as built-in measurement circuits. Measuring the external power supply of the entire system is a pragmatic and generic solution. The advantage is that this captures all components. The disadvantage is that this captures all components. In other words,

while the applicability is almost universal and it covers all aspects of a system, the effort of model building increases. Section 9.5 argues that this can be a useful trade-off.

A second disadvantage is that internal voltage regulators commonly use smoothing capacitors. These can add a delay between the moment system power changes and the changed power is observable at the external power supply. For HPC applications, this should not be significant, since systems can be expected to run near their maximum power, which would lessen or eliminate this delay. But this effect needs consideration when low power situations are involved.

When measuring system power through the external lines, signal quality is reduced. These mainly stem from voltage conversion. Common voltages for main PCBs range from 3.3 V through 12 V. Onboard voltage regulators or PMICs then create the required target voltages, often multiple different voltages between 1 V and 3.3 V.

These voltage regulators add undesirable effects to the power measurement signal. At the very least, stabilising capacitors effectively create a low pass filter. Switching mode regulators modulate one or more periodic signals on top of the desired signal. The operating frequencies range from 100 Hz to 100 kHz, wave forms and frequencies may change during operation. The signal is symmetric, however, so when integrating or averaging such a power signal over a sufficient time span, the net impact is negligible.

5.3.2.5. Parallel Measurement

Another question is how to measure a complete cluster instead of just a single node. For model building, a single node might be sufficient. For evaluation of prediction accuracy, the entire cluster must be measured. On the other hand, evaluation needs much less detail, so the external power supply measurement approach is quite sufficient. If more details are needed for model building, a single node can be treated more invasively.

6. Modelling and Simulating with SystemC

SystemC [7] is a C++ class library that was originally designed as a hardware description language (HDL) working at the register transfer level [29]. If authors use a supported subset of C++, hardware designs expressed in SystemC can be synthesised to hardware by commercial tools like Xilinx Vivado.

The task of designing a modern electronic system requires design methodologies that are more complex than just using a traditional HDL for hardware design. System level design [30] is a systematic approach rooted in a high-level functional specification of system behaviour.

The main advantages of SystemC over established HDLs like VHDL and Verilog are twofold: First of all, by expressing the hardware design as C++ code, SystemC models are also executable simulation models. This supports the continuous validation approach of system level design.

Secondly, C++ facilities allow easy abstraction and composition so that SystemC spans a wide range of abstraction levels. It is able to express system models from the purely functional specification level down to the register transfer level. Even better, different abstraction levels can coexist in the same model; refinement steps can leave some parts of the system untouched while fully refining others.

Add-on libraries add flexibility to this process: Transaction Level Modelling (TLM [7]) provides an abstract communication model that allows designers to specify physical implementations (data bus structures and protocols) very late in the design process while providing accurate communication timing and behaviour much earlier.

The Oldenburg System Synthesis Subset (OSSS, [31]) adds abstraction features that ease the refinement process from functional specification models to synthesisable hardware/software models. Other add-on libraries provided extended support for verification of models or modelling of mixed-signal (analogue/digital) systems.

The methodology of this thesis uses SystemC as simulation platform. Despite its original purpose, SystemC is not used as a design tool[1], merely as a tool to model existing systems in varying degrees of abstraction.

6.1. Components of SystemC Models

The fundamental modelling unit in SystemC is the MODULE. Every SystemC component is contained in a module, and the simulation model as a whole is a collection of modules. Technically, a module is a subclass of C++ class `sc_core::sc_module`.

[1] To be exact, there is no low-level hardware design. In the terminology of system level design, some aspects of the methodology might be interpreted as a platform-based system modelling approach.

Modules consist of several components:

Child Modules (or rather, instances of modules) inside a parent module represent hierarchical composition of a model. It is possible to build a module hierarchy algorithmically. This allows parametric static modelling but excludes dynamic models. The hierarchy cannot be modified during simulation.

Processes implement functional and extrafunctional behaviour of a module. These are C++ functions that exist in two basic flavours: `SC_THREAD` and `SC_METHOD`. The differences between them are their semantics during simulation. Threads are basically coroutines that are interrupted and resumed based on events, while methods are called once for each event of interest and run to completion before simulation proceeds.
Processes declare EVENTS they want to react on. They can do so statically or dynamically during simulation.

Channels represent communication facilities. These can be as simple as objects of class `sc_core::sc_signal<bool>`, which models a single-bit digital signal or as complex as a TLM-based memory bus model with multiple bus masters, priority-based arbitration, and pipelined bus protocol.
Channels are either classes derived from class `sc_core::sc_prim_channel` (PRIMITIVE channels) or they are modules themselves, in which case they are called HIERARCHICAL channels.

Ports and Sockets are interface objects for channels. Modules are not expected to provide channels for external communication. Instead, they contain PORTS (for simple channels) or SOCKETS (for TLM channels) that are bound to channels during model instantiation.
Canonical SystemC design expects a parent module to contain channels for its child modules, and to bind these together during instantiation. They should not contain channels for communication outside its local module hierarchy.
The exception to this guideline are hierarchical channels. These commonly contain internal channels which they partially expose to the outside world through a special forwarding port called an EXPORT.

6.2. Discrete-Event Simulation

SystemC has a well defined execution behaviour for such a structural model. Before simulation starts, ELABORATION happens. It consists of the instantiation of all SystemC modules by the main program. During this process, module constructors and the main program itself bind all ports to appropriate channels. The result is the final connected object hierarchy used as simulation model.

The actual SIMULATION then proceeds in distinct steps:

1. A one-time INITIALISATION phase gives all modules the chance to set up initial EVENTS.

2. Each event has an annotation determining the simulation time it will be triggered. The SystemC scheduler determines the earliest simulation time that has pending events and forwards simulation time accordingly.

3. For all pending events at the current simulation time, the scheduler collects all processes that wait for that event. This results in the set of RUNNABLE processes. This starts the EVALUATION phase.

4. The SystemC kernel executes the runnable processes one after the other in an indeterminate order. Methods are simply called and run to completion, while threads are resumed and run until they issue a `wait()` statement, at which point they are paused.
   Processes may trigger (NOTIFY) new events at defined moments in the future, or even during the current evaluation phase. Notifications are categorised in three classes:

   Immediate notifications take effect during the present evaluation phase; they can directly modify the set of runnable processes.

   Delta notifications represent an unspecified (or infinitesimally small) time span for immediately consecutive processes. In other words, delta notifications enforce a causal order on processes that would otherwise run concurrently, but without specifying an exact temporal relationship.

   Timed notifications express passage of physical simulation time.

5. Conceptually the processes in the evaluation phase run concurrently, so communication through channels does not take place immediately. Communication operations and signal changes are noted down by channels but not executed. After all runnable processes have been handled, the UPDATE phase is used by channels to put the noted down effects in place. This can trigger new events.

6. This process repeats with step 2. If delta notifications are pending, these are considered earlier than any timed notification and the resulting simulation loop iteration is called a DELTA CYCLE. Simulation ends if no events are pending anymore or the model explicitly requests simulation end.

6.3. System Level Modelling

Given the wide range of abstraction levels, SystemC models support a typical system level design flow starting at a purely functional specification without any hardware structure. In subsequent steps, computation and communication refinement would then introduce a hardware platform and mappings of system behaviour to various computational elements and their communication infrastructure.

Hardware models can then introduce extrafunctional properties. Chief among them is timing, while energy is another common property of interest. More rarely, simulations

model temperature and ageing effects. Time is a first class citizen in SystemC; other physical properties are supported through add-on libraries, e. g. Timed Value Streams [32].

While the typical design target is an embedded system, this flow can also work on larger systems like a networked cluster architecture. In this case, some existing facilities might be based on assumptions that either do not hold for large systems or are too detailed for feasible simulation. This affects both, computation and communication modelling, as described in the following sections.

6.3.1. Computation

In regular system design, computation modelling has three major levels of detail: untimed host-based execution, annotated host-based execution, and simulated execution.

Purely functional behaviour models run application code compiled natively inside SystemC processes; this is also called host-based execution. In this mode, no extrafunctional information (timing, energy, etc.) of the code is present in the simulation. On the other hand, simulation speed is maximised.

The next level of detail adds annotations to host-based code. SystemC offers the `wait()` call that suspends the running SystemC thread for a specified amount of simulation time. These can be inserted into application code in appropriate places so that simulation time is advanced in intervals. More `wait()` calls mean more timing accuracy but less performance due to additional scheduling overhead.

Actual time values can be determined in a variety of ways. For simple CPUs, a static timing model might be able to calculate timings from compiled machine code. Measurement is another common approach; it works well on different granularities like whole functions or basic blocks. Expert estimation may also work, and even some arbitrary constant delay sometimes conveys enough information for the desired insight. The overview in Section 4.2.5 applies here as well.

The most detailed way to model computation timing is simulated execution. Application code is ported to and compiled for the target platform and executed inside an instruction set simulator (ISS) for the target architecture. Models can then use the executed instruction stream to provide detailed timings. Accuracy ranges from counting of number of executed instructions up to clock cycle accurate simulation of all microarchitectural details. While theoretically offering best possible accuracy, performance is often several orders of magnitude slower than host-based execution.

Computation timing often also depends on more resources than just a CPU core. RAM bandwidth, cache size, and other details can influence computation timing between different CPU cores. The hierarchical modelling approach that is common in SystemC is well suited to represent these effects: For example, two CPU cores could be part of an intermediate SystemC module that also contains the shared cache. By routing both core's RAM accesses through that cache, the hierarchy models the interference quite naturally.

6.3.2. Communication

For on-chip communication, there are two major modelling approaches: modelling individual signals and transaction level modelling.

Signal-level models contain all individual digital signals of a transmission path, which can easily amount to over a hundred signals for a parallel bus. Simulating the exact signal waveforms is costly, but is clock cycle accurate.

That's needlessly much detail for many use cases. Transaction level modelling (TLM) raises the modelling abstraction so that communication is described in terms of communication transactions. A TLM transaction is a full read or write operation across a communication path, e. g. a bus or a serial line. Since communication protocols tend to be very regular, TLM can still reach clock cycle accuracy. TLM leaves out the actual signals involved, but keeps communication content and timing. Timing accuracy is variable; TLM offers multiple speed vs. accuracy trade-offs.

When modelling networked communication, application-level communication is usually much more complex and spans the entire range of the OSI reference model [33]. Multiple communication protocols work on different layers. A single read or write transaction involves effects that are hard to predict (e. g. multiple round trips for session establishment, retransmission upon packet loss, or computational overhead of encryption). Thus a simple TLM approach can't model communication timing as easily as with local communication.

To fill that gap, dedicated network simulation tools exist. A popular tool is OMNET++ [34] (also known as OMNEST, which is the name of the commercially licensed variant). It is a network simulator that supports co-simulation with SystemC. The INET framework for OMNET++ provides simulation models for a wide range of internet-related network protocols from all protocol layers. With such a simulator, even nondeterministic effects like packet loss can be simulated.

Simulating all network layers is slow of course. This effort may not be needed: For switched Ethernet, transmission is pretty reliable and the major source of nondeterministic behaviour is competing (unaccounted) network traffic. In a fully controlled network environment, there is no competing network traffic. Furthermore, protocol effects that are hard to model individually might average out over longer transfers. A simple time model based on average net bandwidth would then allow TLM to model networked communication.

6.4. Time and Energy Traces

In the end, a simulation run should produce some kind of useful output. For functional testing for example, input/output behaviour of the system is of interest. SystemC includes a tracing facility with which internal state can be recorded in value change dump (VCD) files . VCD is standardised as part of Verilog (IEEE 1364) and supported by various electronic design automation (EDA) tools. VCD files contain a set of variables (signals) and their values over time, including timestamps for each value change.

The source of VCD values can be SystemC signals, plain C++ variables, or explicit API calls that assign a new value to a given VCD entry. As a result, VCD is not restricted to tracing communication signals and their timings. Values being traced could also represent current power consumption of individual system components, for example.

For more detailed analogue systems, simple discrete snapshots of average power might be insufficient. These are often modelled in a time-continuous manner. The add-on library SystemC-AMS (analogue mixed signals) allows such models and offers its own tracing facility designed after core SystemC tracing.

Both tracing facilities have the drawback that the amount of information in big simulation models can become unwieldy. When looking at extrafunctional properties, a full trace is often not needed, only consolidated values. For example, when comparing a simulation to measurements of the main power supply (see Section 5.3.2), only the (momentary) overall power consumption is of interest. In a simulation model that calculates power in many different subcomponents, the amount of data could be reduced by only storing the sum of all power values instead of all the individual values. This consolidation means extra effort in the SystemC model.

To make matters worse, systems might use different clock frequencies and thus timing resolutions for their extrafunctional properties. This makes consolidation even more difficult. The Timed Value Streams [32] add-on library solves this problem by providing a tracing framework that supports on-the-fly consolidation and resampling. Furthermore, tracing code remains separated from simulation logic, improving maintainability of both.

7. Thesis Contributions

Applying the methods and technologies presented so far to the context outlined in Section 1, this thesis proposes a number of contributions to address the research questions presented in Section 2.1.

7.1. Contributions

Contribution 1. *An application meta-model for large parallel applications that is suitable for static modelling of a significant subset of high performance computing (HPC) workloads.*

The application meta-model is based on task dependency graphs. Unlike current approaches in HPC that use dynamic construction of task graphs during run time, application models are meant to be built statically (ahead of time). This is the crucial property that the methodology proposed in this thesis builds upon.

Two notable drawbacks of static construction are graph size and the inability to vary graph structure based on intermediate results. The application meta-model and associated methodology address the scalability issue by employing algorithms with (at worst) linear time complexity. Data dependencies in general cannot be handled, but there are ways to express some important kinds of data-dependent graph variation.

Contribution 2. *A hardware platform meta-model of highly heterogeneous cluster systems that allows variable abstraction and irregular communication infrastructure.*

The hardware meta-model allows users to model heterogeneous compute resources and their equally heterogeneous communication infrastructure. The level of detail in these models is intended to capture relevant system parameters (e. g. communication bandwidth). A given platform model should contain just enough structure and all required parameters to build a complete simulation model.

Contribution 3. *A well defined execution model for applications provided as Contribution 1 models executed on platforms captured in Contribution 2 models, and an implementation of this model for clusters running the Linux operating system.*

Predictability requires well defined execution semantics for applications. The methodology proposed in this thesis builds upon a custom execution runtime hosted on Linux[1] for applications modelled as described above. The most distinctive property is that it assumes a static task graph.

[1] Probably working other operating systems as well, but that is untested.

This execution runtime does not aim to be exceptionally fast. Nevertheless, application speed should be at least in the same order of magnitude as a straightforward best-effort implementation. Otherwise, optimisations discovered through the methodology of this thesis are of little use.

Contribution 4. *A parameterised resource model (timing and energy) for applications, and a measurement-based methodology to determine platform and application model parameters.*

Since the ultimate goal of this thesis are time and energy predictions, a platform model needs to contain parameters that facilitate such predictions. Physical measurements are one way to provide the required data. The important part of this contribution is that such measurements must be practical even in the context of HPC hardware.

Contribution 5. *A simulation-based methodology for time and energy prediction for large parallel applications on parallel cluster architectures.*

This is the central step that joins the preceding contributions to create actual predictions. It supports multiple usage scenarios:

- In an interactive optimisation flow developers would use predictions to make design decisions for an application while they are still developing it.
- In an interactive hardware tailoring scenario users would explore different hardware platforms in order to find the one most suitable for a given application or set of applications.
- In automated variants of the above users would perform design space exploration (DSE). They would generate application and/or platform variants and determine the set of pareto-optimal variants.

The key property to support all of these is speed: Prediction is fast enough so that these scenarios work with a useful delay.

Contribution 6. *An affordable evaluation platform for evaluation of Contributions 1 to 5.*

To evaluate the proposed methodology in its original scale, a high performance computing system with suitable energy measurement infrastructure would be needed. Existing approaches do not meet the capabilities needed for evaluation.

As this doesn't exist as an off-the-shelf obtainable product, this thesis proposes a scaled-down cluster system made of common single-board computers. The ratio of computation speed over communication speed matches HPC systems, so the evaluation of Contribution 5 likely stays valid when applying it to HPC-scale systems.

Since off-the-shelf components evolve quickly, the design principles behind this contribution can also be used to build updated variants of the embedded cluster system while systematically checking for common pitfalls of consumer grade components.

Contribution 7. *A measurement system for synchronous multi-channel power measurements.*

As a minor byproduct, this thesis proposes a custom measurement platform. It offers multiple synchronised channels of power supply current and voltage measurement to support the external measurement approach presented in Section 5.3.2. Signal bandwidth is high enough to identify systematic noise, which means it can observe all meaningful details of a power supply measurement approach.

The measurement platform is not strictly required, and other means of obtaining measurement data for Contribution 4 are equally valid. The proposed platform is much cheaper than traditional measurement equipment, however. It makes it financially feasible for employers to give all developers their own measurement system.

This platform is integrated with Contribution 6 for measurement of the entire cluster system.

7.2. Assumptions

The methodology proposed in Contribution 5 relies on several assumptions:

Assumption 1. *Applications can be expressed as a static task graph with up to* 10^7 *nodes.*

The chosen application meta-model is not capable of expressing all kinds of parallelism, but many algorithms can be unrolled to generate a static task graph. The graph size limit is big enough for practical applications, e. g. for finite element simulations with a million cells.

Due to linear scaling throughout the methodology, the actual limit is dependent on how slow a single prediction may get, and at what point external nonlinear effects become noticeable, e. g. from the operating system.

Assumption 2. *The majority of application tasks can be grouped into a few classes (*$N < 10$*).*

In other words, an application executes a small set of code fragments repeatedly, but in arbitrary combinations and with different data sets. Among others, this covers the class of HPC applications that make heavy use of linear algebra libraries, but also many mesh and N-body problems (see Section 4.2.1).

Assumption 3. *Parallelism is coarse grained.*

More specifically, execution times of atomic units of computation are assumed to be between 10^{-2} s and 10^4 s. Shorter tasks are permitted as long as they do not contribute significantly to the overall execution time. Application maketime is expected to be on the order of hours or days.

Longer tasks are permitted and should work as well, but evaluation does not cover larger time scales.

Assumption 4. *Communication transactions are on a similar time scale as the atomic computation operations.*

The majority of transmission operations are expected to have a size between 10^5 bytes and 10^9 bytes. This precludes applications that perform frequent small scale synchronisation.

The main reason for this requirement is of a practical nature: The exact internal operation of networking devices (e. g. Ethernet switches and network interface hardware) is almost impossible to predict. It shows nondeterministic behaviour, so this thesis relies on scale to minimise the impact of that nondeterminism.

In theory, a communication stack with improved determinism (e. g. Ethernet TSN) could improve or even lift this requirement, but practical availability of Ethernet TSN is still insufficient for building evaluation systems.

Assumption 5. *Applications have no requirement for significant amounts of variable-time input/output besides explicitly modelled communication.*

This mainly addresses mass storage. Applications that process data sets much larger than can be stored in distributed RAM need to perform input/output operations to a mass storage medium. Predicting the latency of these operations is an entire research field in itself.

Consequently, loading of initial data and storing of final results is assumed to happen in constant time. An alternative view is that loading and storing of data happen outside the predictions.

Assumption 6. *Hardware platforms provide sufficient control over automatic mechanism that vary execution time.*

The main effects addressed here are dynamic voltage and frequency scaling (DVFS) and thermal throttling as explained in 4.1.4 and 4.1.5. These may react to influences outside of the modelled system, like ambient temperature. As a result, meaningful prediction needs the execution runtime to turn off automatic clock frequency changes and lower the operating frequency to a sustainable value.

This assumption means that performance may be lost because running at a lower clock frequency may waste temporarily available thermal budget. On the other hand, the higher the utilisation is the less CPU temperature will vary. So this might not be a significant loss for fully optimised applications.

While not addressed in this thesis, it should be possible to extend the proposed methodology to support explicitly controlled DVFS in future work.

Assumption 7. *The atomic units of computation have already been fully optimised for microarchitectural effects.*

The methodology does not address this level of optimisation; due to Assumption 3, this can be separated into a separate step that is outside the scope of this methodology.

This also means that future changes in the code will invalidate predictions: They cannot easily be modified for a change in low-level application code. After such a change, the prediction process needs to be rerun partially or entirely.

Assumption 8. *The hardware platform is a real-world heterogeneous cluster system.*

This is mostly not meant as a limiting assumption, but as an explicit statement towards adverse effects the methodology does take into account. Among them are:

- more than two different kinds of computation resources
- at least tens to hundreds of compute resources
- manufacturing variations between nodes concerning power and timing characteristics
- asynchronous operation of individual cluster nodes

More restricted platforms will work, of course, but for those there may be more efficient methodologies for time and energy prediction.

One limiting factor this assumption includes is that the proposed methodology does not explicitly model heat, so the target platform needs to run in a temperature-controlled environment for reliable results.

8. Related Work

The methodology proposed in this thesis mainly relates to other approaches to predict extrafunctional application properties in high performance computing. Physical measurement is a secondary contribution. This section will discuss these fields separately.

8.1. Time and Energy Prediction

Table 8.1 shows an overview of the prediction techniques discussed in this subsection.

8.1.1. Execution-Driven Simulation

Execution-driven system simulation is an obvious technique for time and energy predictions. Many flexible simulation platforms exist that manage power as well as timing, e. g. Gem5 [64] or Simics [63].

These can be many orders of magnitude slower than real time, so they are not suitable for HPC applications. Optimisations include instrumentation in combination with static recompilation or dynamic binary translation, e. g. [8].

SystemC [7] as used in this thesis is a popular simulation framework in embedded system design, although usually at a less abstract level. The COMPLEX framework [38] achieves high performance by using SystemC with host-based execution.

Parallel discrete-event simulation provides further speedup, and combining this with dynamic binary translation [8] yields impressive results.

Still, none of these can overcome the inherent performance limits of execution-driven (i. e. functional) simulation. As long as the simulated platform is much bigger than the simulation host, execution-driven simulation has performance limits that cannot be overcome.

Another approach to speed up execution driven simulation is sampling and checkpointing. Simulators like SimPoint [6] or SMARTS [9] use this to only execute certain parts of a system simulation. While useful for localised optimisation even for HPC applications, this does not help with predictions of overall application properties.

There is one place where execution-driven simulation might be of use in the context of this thesis: A suitably accurate simulator could be used for the characterisation phase instead of doing measurements on real hardware specimens.

Table 8.1.: Typical characteristics of prediction techniques (represented by prominent examples) and comparison to the proposed methodology

Section/Class (Examples)	Speed (est.)	Granularity	Energy?	Data Deps.?	Heterogeneous Platforms?
8.1.1 Execution					
Gem5 [64]	$10^{-3}\times$	Instruction	Yes	Yes	Yes
8.1.2 Trace					
PSINS [58]	$10^{3}\times$	MPI	-	Yes	-
TaskSim [59]	$1\times$ – $10^{-5}\times$	Task –Instruction	-	Yes	Yes
RMAP [48]	$O(1)$	Application	Yes	-	-
8.1.3 Abstract					
SimMatrix [61]	$1\times$	Task	-	-	-
8.1.4 Analytical					
Petri Nets [62]	$10^{2}\times$	Task	-	-	Yes
This Thesis	$> 10^{3}\times$	Task	Yes	(Histogram)	Yes

8.1.2. Trace-Based Simulation

If exact functional behaviour isn't needed, abstraction is a way to simulate extrafunctional behaviour of HPC-scale applications. More specifically, trace-driven simulators skip executing actual code during the prediction process. Dimemas [66], PSINS [58], and TaskSim [59] are examples of this class of simulators.

Dimemas and PSINS work at the granularity of MPI API calls. In some circumstances, these can reach speeds comparable to the methodology proposed in this thesis.

TaskSim is significantly more detailed, down to individual memory operations. This allows it to provide accuracy much closer to execution-driven simulators. Of course, simulation speed gets closer to execution-driven simulation as well: even in its fastest mode, TaskSim only reaches parity between simulation time and simulated time.

Trace-driven simulators have drawbacks over this thesis' methodology, as explained in Section 4.5.4: they require execution traces from a comparable system, which in turn means that the application must already be completely runnable.

SST/Macro [60] is an example of the synthetic trace approach. It allows performance prediction during application design similar to the methodology proposed in this thesis. The main difference is that SST/Macro uses a simulated MPI API as programming model. With MPI, the skeleton application encodes computation, communication, and parallelisation as C++ code. This means that target platform variations require changing the skeleton application. Similar complexity affects time modelling; in fact SST/Macro

focuses on communication and uses a very simple computation time model.

A recent publication [67] shows an approach using Dimemas or SST/Macro driven by skeleton applications, which are also an important option for the methodology proposed in this thesis. Similar to this thesis, the paper uses characterisation data from a test run on a small subset of the target machine to predict application behaviour on the full machine. Skeleton applications eliminate the need for full execution, but hurt Dimemas performance; except for small configurations, simulation time is much slower than simulated time. As the approach is based on Dimemas or SST/Macro, their other limitations still apply.

In HPC simulators, energy is usually ignored. A notable exception is the RMAP resource manager [48], which predicts power and timing of a given HPC application in order to find the optimal subset of the available hardware for execution. In relation to the proposed methodology, it has two main drawbacks: its linear regression model probably only works for homogeneous platforms, and it needs multiple executions of the full application on the target platform for model building. On the other hand it is very fast, since a linear regression model has $O(1)$ time complexity for predictions.

8.1.3. Abstract Simulation

Abstract simulation is not based on execution of real application code, neither directly nor indirectly through traces; instead it uses abstract approximations for significant parts of the application.

In the HPC world, it is rarely seen (except in the form of trace-based simulation). SimMatrix [61] is an example that explores future exascale systems. It is specialised for a specific scenario, though: it assumes a fixed cluster architecture and workload. Again, it doesn't handle energy.

The main point of this thesis is to propose a much more general approach of abstract simulation without incurring the inflexibility of existing trace-based methods.

8.1.4. Analytical Modelling

In theory, analytical modelling approaches can be faster than simulation-based approaches. One example for HPC-style hardware is [65], but it addresses single-core performance only.

The authors of [62] use Petri nets on task graphs; they demonstrated it on small task graphs only (less than 10k tasks) and use an even more abstract platform model than this thesis.

In practice, analytical modelling of multi-core architectures and big cluster systems is difficult or not feasible at all. There are various approaches using Integer Liner Programming or Timed Automata, but they all suffer from poor scalability due to the inherent nonlinear nature of asynchronous and heterogeneous cluster systems.

8.2. Measurement Platforms

There are commercial vendors for highly parallel power measurements. For example, dSPACE offers modular measurement systems with variable channel count and accuracy. Such systems support a wide range of voltages and currents, but easily cost tens of thousands of Euros even for low channel counts. As a result, various research projects built low-cost solutions tailored to their requirements.

The MAGEEC (Machine Guided Energy Efficient Compilation) project has designed an analogue measurement interface board [35] measuring three channels at up to 2 MHz if paired with a suitable ADC. There is no published evaluation of accuracy or effective analogue bandwidth, however.

The ADEPT (Addressing Energy in Parallel Technologies) project similarly produced a custom measurement solution [36]. It is based around a central FPGA-based processing board that supports up to about 80 channels of dedicated 18-bit ADC boards each running with up to one million samples per second. There is no published evaluation of the platform itself either. The ADEPT solution has additional hardware tools for measurement of PCI Express and SDRAM boards.

In [51], Khan et al. present a power model that uses the Intel RAPL feature present in modern Intel CPUs to derive system power. While this is based on CPU activity counters and doesn't really measure power, the accuracy is close to physical measurements. The time resolution is 1 ms (although jitter is reported to be high), and mean error is less than 5 %. This is within one order of magnitude of the proposed measurement infrastructure and widely (but not universally) available.

Apart from the lack of accuracy evaluation, the ADEPT solution would have fulfilled the requirements of this thesis. However, it wasn't available at the time this thesis' evaluation platform was built. Furthermore, the measurement platform designed for this thesis combines the cost effectiveness of the MAGEEC approach with the most important performance aspects of the ADEPT approach.

Part III.

Models and Methodology

9. Power and Timing Prediction Methodology

Figure 9.1 shows the setting in which the proposed method is assumed to work. Users work on a (potentially) parallel application that they want to deploy on a parallel cluster system. The deployment process involves mapping parts of the application to individual cluster servers for execution. Measurement of the running application can then provide feedback for optimisation of the application's structure or its mapping to the platform.

The methodology I propose adds an alternative path, where the mapped application is not executed; a simulation predicts application timing and energy usage instead. Users can then perform optimisation the same way as if they had measurements.

Predictions are only useful if they are faster than physical execution. They must be orders of magnitude faster if they should support useful iterative development workflows of high performance computing (HPC) applications on typical development workstations. This is only possible through abstraction – a simulation that leaves out many details of physical execution. As a result, predictions will not match physical measurements, and the accuracy describes how reliable they are.

9.1. Overall Design Flow

Figure 9.2 shows the final development flow as proposed in this thesis including the models that are needed. When starting out with a new application design, the overall methodology operates in a series of steps:

Task Graph Creation To use the proposed prediction methodology, users first need an abstract model of their application in form of a task graph (see Section 9.2.1). There are various ways to create these, but I anticipate three main approaches:

- An application might already use task graph representation. Users would only need to provide a transformation into the models specified above.
- Users start from scratch and use our flow to guide design decisions in a top-down development process. They model their application using skeleton applications.
- For a bottom-up approach, users can decompose a sequential reference implementation of an application. They extract computation kernels, which contribute to the characterisation database, and control structure, which they transform into a task graph.

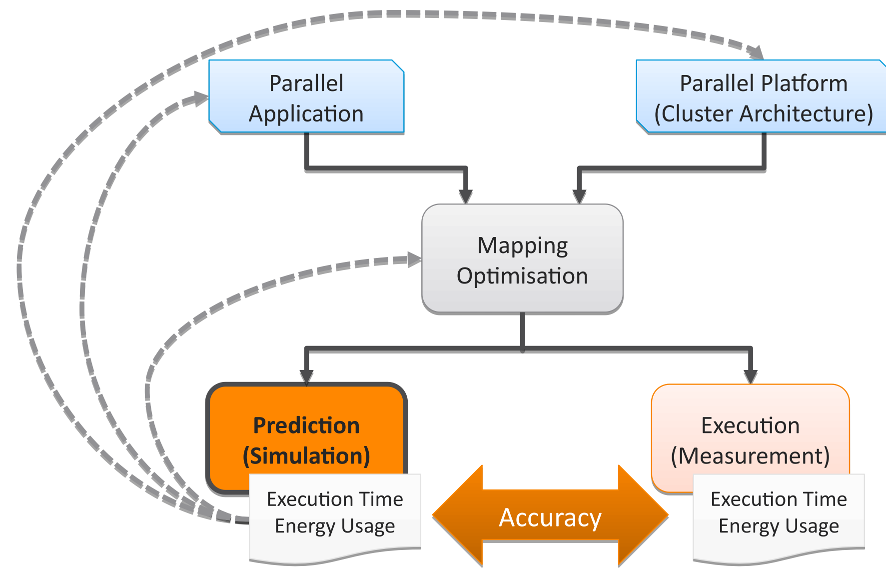


Figure 9.1.: Assumed development approach for the prediction methodology.

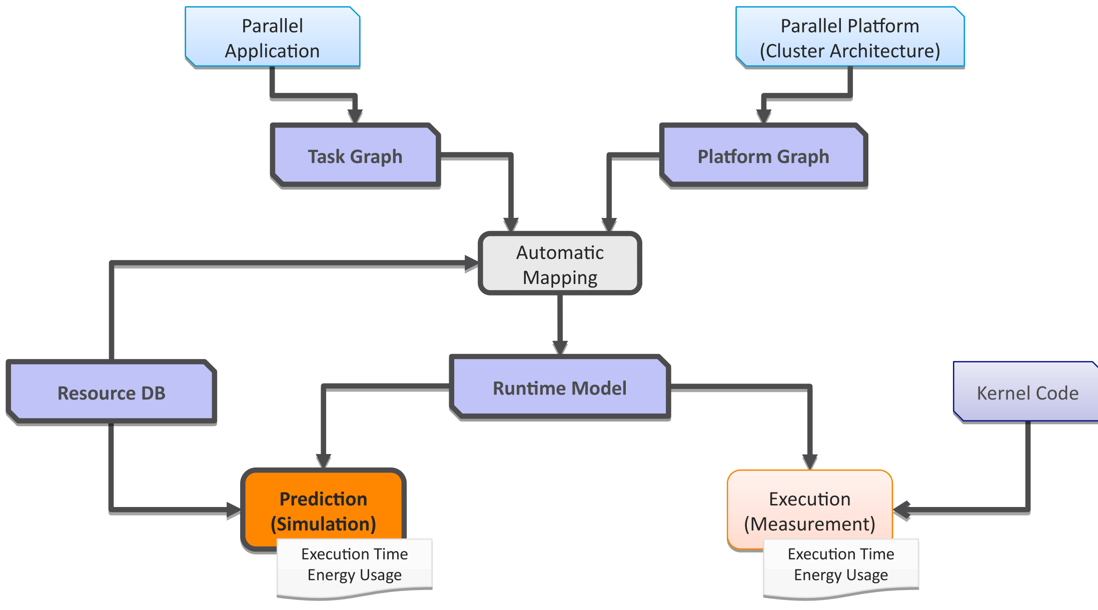


Figure 9.2.: Final development flow including all models. Bold borders indicate elements introduced by this thesis.

The task graph does not fully define application behaviour. An execution runtime model (see Section 9.3) defines how tasks are executed across a given target platform, both for simulation and real execution. This model is part of the methodology; the user does not need to create it.

Platform Graph Creation A platform model provides information about the target compute system (see Section 9.4). It captures type, number, and connectivity of processing elements, and their hierarchical composition. Users can easily create these by hand, or use existing ones.

Platform Resource Model Creation Users need to perform a platform characterisation process or have the results of one already available. (See Section 9.4.2)

Computation Resource Model Creation A characterisation database supplies performance and energy data for functional kernel code (see Section 9.5). In the case of top-down modelling from scratch, users could delay implementing kernel code and provide estimates for early feedback at less accuracy. Once kernels are implemented, prediction accuracy increases.

Mapping The next step is mapping (see Section 9.6), which uses task graph, resource model, and platform model to map each task of the task graph to a processing element on the target platform and generates a task list for each PE.

Simulation Simulation then uses this mapping in conjunction with the resource model to predict maketime and power consumption of the entire system in detail. In addition to those, it may calculate other metrics provided by simulation models (see Section 9.7).

Iteration With these predictions, users can successively refine and optimise the application: they can restructure the task graph, e. g. by changing the amount of parallelism, they can change the application's separation into kernels, and they can even start over and rewrite the application using different algorithms. Simulation is intended to be fast enough for frequent iterations of this process in an interactive workflow.

Deployment Finally, users run the optimised application using the physical task graph execution runtime, which executes concrete implementations of computation kernels according to the specifications of the mapped task graph.

The following subsections explain each model and the simulation process in detail.

9.2. Abstract Application Model

The main principle of the proposed methodology is that simulation does not execute functional application code. Therefore, the proposed methodology uses an abstract application model, labelled TASK GRAPH in Figure 9.2.

9.2.1. Task Graph

The application model is an attributed directed acyclical multigraph, the task graph. In literature, there are many variations of task graphs with subtle differences and capabilities (see [10] for an overview). For some applications (e. g. finite element simulation using iterative refinement), this task graph can be thought of being executed periodically, while predictions cover a single iteration.

The task graph I propose is the primary application model; it expresses an application's high level parallel data flow structure. Experience has shown that trying to extract it from existing program code by automated means is difficult or impossible; users of the methodology should design and maintain it explicitly.

An interesting comparison for these task graphs is that they are similar to the synchronous dataflow (SDF) model of computation as used in embedded system design. More specifically, task graphs are roughly equivalent to homogeneous SDF graphs. This means an SDF model can be transformed into a functionally equivalent task graph by determining a static schedule and transforming SDF activations into task graph tasks[1]. The resulting task graph should be executed repeatedly. The reverse transformation is a trivial 1:1 mapping of graph elements.

Task Graph In the task graph, nodes represent computation and are called TASKS, while edges represent communication and are called DATA DEPENDENCIES. More formally defined, a task graph G is a tuple

$$G = (T, D, p, s)$$

where T is the set of tasks and D is the set of data dependencies., and p, s are functions that specify the PREDECESSOR and SUCCESSOR of each data dependency, respectively.

This definition includes graphs with multiple disjoint task subsets. This represents running multiple independent applications in parallel.

Figure 9.3 shows an example for Cholesky matrix decomposition.

Kernels Each task is an invocation of a code fragment called a KERNEL. Kernels have attributes that define required INPUT data and produced OUTPUT data. Kernels can be parametrised by VARIABLES, e. g. to specify the data size they operate on.

Formally, an application is based on a set K of kernels. Each kernel $k \in K$ is annotated with a set I_k of inputs, a set O_k of outputs, and a set V_k of variables. Inputs and outputs

[1] Additional work needs to be done to convert communication, but that's similar in complexity.

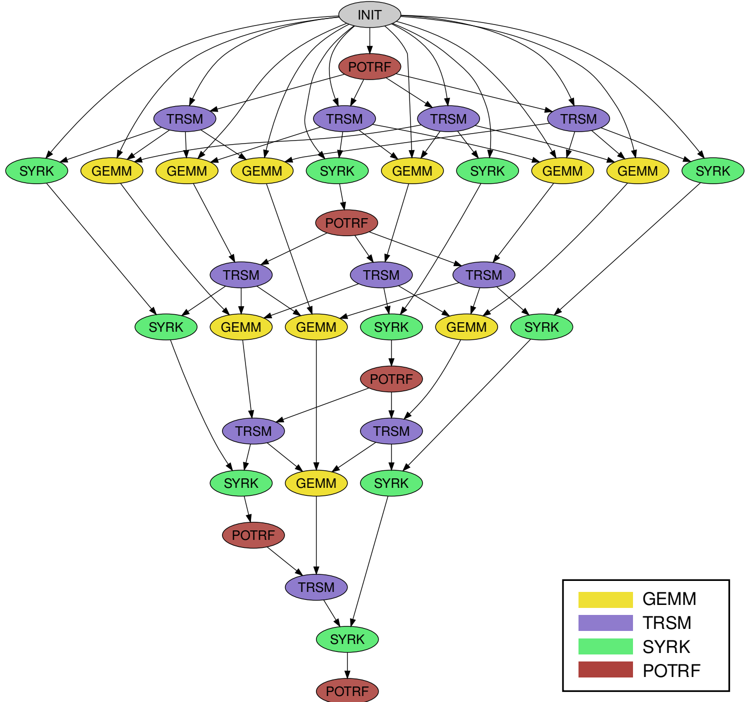


Figure 9.3.: Example of a task graph performing Cholesky matrix decomposition with 5×5 matrix subdivision. Colour denotes different kernels.

Listing 9.1: Example of a kernel definition in an XML serialisation.

```
<kernel id="GEMM">
    <variable id="tile_size" />
    <input id="GEMM.A" size="tile_size * tile_size * 8"/>
    <input id="GEMM.B" size="tile_size * tile_size * 8"/>
    <input id="GEMM.C" size="tile_size * tile_size * 8"/>
    <output id="GEMM.Cout" size="tile_size * tile_size * 8"/>
</kernel>
```

Listing 9.2: Example of a task definition in an XML serialisation.

```
<task id="GEMM-0" kernel="GEMM">
    <assign var="tile_size" val="1024"/>
</task>
```

have one attribute, the data size expressed as a mathematical expression that can reference kernel variables. Listing 9.1 shows an example for the `GEMM` kernel which has one variable and three inputs; it calculates $C_{\text{out}} = AB + C$, where A, B, and C are two-dimensional square matrices of double-precision floating-point values. The `GEMM` subroutine from the BLAS library modifies C in place, but as a kernel this is not relevant; conceptually, inputs and outputs are distinct.

Tasks Every task $t \in T$ is annotated with the kernel $k(t) \in K$ it executes, and a function $\nu_t : V_{k(t)} \to \mathbb{R}$ that assigns a value to each kernel variable. Listing 9.2 shows an example for a task executing the `GEMM` kernel on a set of 1024×1024 element matrices.

Data Dependencies Each dependency has annotations that specify which output of the PREDECESSOR task is transmitted to which input of the SUCCESSOR task. If multiple data sets are transmitted between two tasks, they are modelled as parallel edges with differing annotations.

This means that a dependency $d \in D$ with predecessor $p(d) \in T$ and successor $s(d) \in T$ is further annotated with output $o_d \in O_{k(p(d))}$ and input $i_d \in I_{k(s(d))}$. Listing 9.3 shows an example for two `GEMM` tasks executing in sequence.

Task inputs must always have exactly one dependency associated with them; the tuple $(s(d), i_d)$ uniquely identifies every dependency. On the other hand, the tuple $(p(d), o_d)$ uniquely identifies each data set (some of which may be transmitted to multiple target tasks or not at all).

Validity Constraints Not all task graphs that can be expressed this way are actually valid. Some additional constraints apply:

Listing 9.3: Example of a dependency in an XML serialisation

```
<dependency predecessor="GEMM-0" successor="GEMM-1"
            src="GEMM.Cout"      dest="GEMM.A"/>
```

1. The graph must not have cycles.
2. All annotations as specified above must be present.
3. For each dependency $d \in D$, the size of output o_d of its predecessor must match the size of input i_d of its successor[2].
4. For every task $t \in T$ and every input $i_{k(t)} \in I_{k(t)}$ there must be exactly one dependency $d \in D$ associated with it, i. e. $i_d = i_{k(t)}$.

There are additional properties that might prevent execution of a task graph on real-world systems, like memory required to store incoming inputs until they are consumed. These do not invalidate the task graph itself, but might invalidate a particular assignment of tasks to nodes in a target platform.

This thesis does not address these platform-specific validity constraints. Platform resources are always assumed to be sufficient for any given assignment.

9.2.2. Application Semantics

Conceptually, tasks run atomically and without side effects. They only start execution after all incoming data dependencies have been transmitted from their predecessors. Incoming communication happens strictly before the start of the task, while outgoing communication happens strictly after the end of the task.

On a real system, various effects can delay or interrupt the execution of a task, e. g. hardware resource conflicts, runtime middleware, or operating system scheduler. Tasks also have the obvious side effects of causing a computation resource to consume energy and time. The application model does not cover these properties; rather, they emerge from the properties of a particular mapping to a target platform.

In real-world applications, some tasks must have functional side effects. Input data must enter the application in some way, and results must be output somehow. Thus some tasks must perform I/O activities that cannot be expressed as data dependencies.

In cases where the task graph describes a single iteration of a periodic process, users can use special tasks to transfer state from one iteration to the next. Since these tasks only influence the next iteration, they do not have side effects for the currently running task graph.

Generally speaking, side effects are allowed as long as they do not break the isolation between tasks (both, functionally and regarding time and energy). Furthermore, I/O

[2] Strictly speaking, their data types must match, but type checking is out of scope for this thesis.

tasks must either be predictable by the proposed characterisation approach or they may only consume a negligible share of the overall maketime and energy[3].

9.2.3. Modelling Process

There are multiple ways to create such a task graph. Possible approaches are fully manual specification, static code analysis (see for example [13]), or partial evaluation techniques.

Skeleton applications are a form of partial evaluation. They offer the advantage of configurability: with little effort, users can create skeleton applications that generate multiple task graphs that differ in degree of parallelisation, problem size, and other properties. This allows fully automated design space exploration, for example. Listing 9.4 shows a code excerpt that works this way. The main loop is exactly the code a serial implementation would run, except it would call Basic Linear Algebra Subprograms (BLAS) functions instead of creating tasks.

Skeleton applications can also model dynamic programming techniques to a certain degree. For example, they may be able to build optimised task graphs for sparse matrix operations if enough information about the input data set is present.

Listing 9.4: Skeleton program excerpt that generates Cholesky matrix decomposition task graphs in different parallelisation granularities.

```
// function that adds task and returns outgoing data set id
extern DataId addTask(string kernel, vector<DataId> inputs);

// task graph granularity parameter
int num_tiles = 5;

// dependency tracking matrix
DataId matrix[num_tiles][num_tiles];

// create data source tasks
... // (omitted for brevity)

// run algorithm main loop
for (int k = 0; k < num_tiles; k++) {

    matrix[k][k] = addTask("POTRF",
        { matrix[k][k] });

    for (int i = k+1; i < num_tiles; i++) {
        matrix[k][i] = addTask("TRSM",
```

[3]Otherwise simulation would need to model those I/O activities, which gets complex pretty fast (e. g. for spinning hard disks).

```
            { matrix[k][k], matrix[k][i] });
    }

    for (int i = k+1; i < num_tiles; i++) {

        for (int j = k+1; j < i; j++) {
            matrix[j][i] = addTask("GEMM",
                { matrix[k][i], matrix[k][j], matrix[j][i] });
        }

        matrix[i][i] = addTask("SYRK",
            { matrix[k][i], matrix[i][i] });
    }
}

// create data sink tasks
... // (omitted for brevity)
```

9.2.4. Discussion of Design Decisions

Task Graph Task graphs in general apply to modern HPC problems; compared to traditional parallel program code they have been shown to offer performance advantages, code size improvements, and better parallelisation properties [16].

This is consistent with the observation that one of the biggest HPC challenges these days is to distribute load effectively. Task graphs address exactly this challenge. Assumption 7 also emphasises this focus.

Static Task Graph Many task-based runtime systems use dynamic tasks. Since the goal of this thesis is to generate predictions without actually running application code, task graphs cannot be dynamic. As a useful substitute, skeleton programs can generate static task graphs algorithmically. Being a form of partial evaluation, skeleton programs can include data properties known at prediction time.

Kernel Variables Skeleton programs can use any information they can derive without running the full application. This even allows them to generate task graphs for some dynamic programming algorithms or sparse matrix operations.

Kernel variables are simply a generalisation of that ability, so that simulation models may base their behaviour on them.

Fully Generated Static Task Graph Static task graphs that are generated algorithmically might not need to be generated all at once. Different parts of a methodology might possibly work with an evolving, partially generated graph as well (as done for example

in [13]). The advantage would be that it could handle task graphs that are too big to be represented in their fully generated form. The main disadvantage would be that no part of the methodology can base its decisions on a global view of the application.

The decision to fully specify the task graph instead of generating it incrementally is more or less arbitrary. It makes some parts of the proposed methodology easier to implement, and achievable sizes are sufficient for real world problems. The methodology could also be adapted to partial generation.

Furthermore, the proposed methodology scales linearly with the number of tasks – regardless of how they were generated. As a result, a task graph that exceeds the limits specified in Assumption 1 is slow regardless of task graph representation.

Communication Granularity (Assumption 4) Since the goal is to simulate large applications on distributed platforms, the expected bottleneck is communication. Frequent synchronisation and small communication transactions can severely reduce overall performance. Consequently, users will want to avoid them in any case.

Task Granularity (Assumption 3) Task granularity simply matches communication timing. More detailed modelling would not add significant optimisation opportunities, i. e. opportunities to perform communication hiding.

Another important aspect of the chosen task/communication granularity is that it makes the methodology mostly orthogonal to low level optimisation. Cache effects, branch prediction, and other microarchitectural effects do not have significant effect across independent tasks. Competition between simultaneously executing tasks are modelled more generically.

Kernels (Assumption 2) The kernel concept is central to the feasibility of the proposed methodology. Limiting the total number of kernels is required as characterisation effort scales polynomially, but usually not linearly. Nevertheless, the stated limits cover a significant set of applications.

9.3. Execution Runtime Model

The dependencies of the task graph constrain the order in which tasks can be executed. Since the task graph's purpose is to express potential parallelism, it does not usually enforce one unique order. It also does not specify actual parallelism, i. e. how tasks are distributed across cluster servers and how they are executed there.

The EXECUTION RUNTIME model (see Figure 9.4) fulfils this role. It is specified as an executable program that takes information from several models (as shown in Figure 9.2) and then deterministically executes the application.

The execution runtime uses abstracted APIs for initialisation, communication, and execution of kernels. For actual execution, a hardware abstraction layer (HAL) provides interfaces to the physical hardware, while a simulation could provide a virtual HAL that

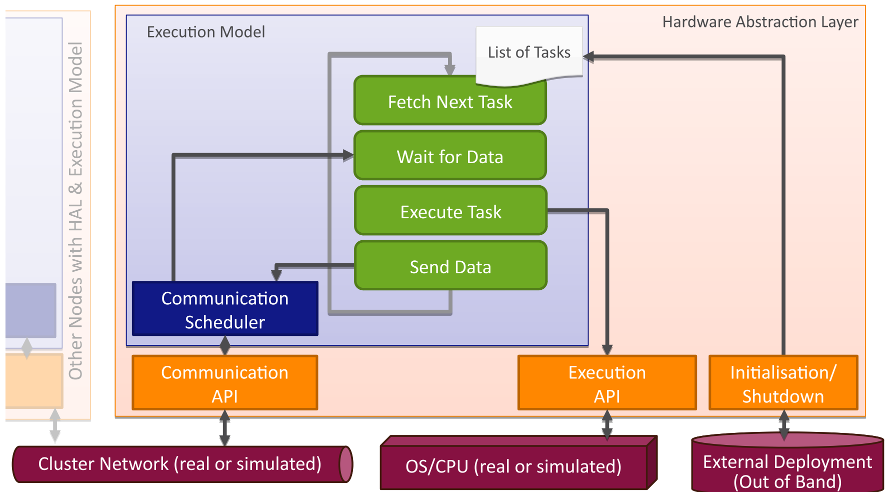


Figure 9.4.: Execution Runtime Model

interfaces with the simulation model. That way, physical execution semantics always match simulation.

Section 9.8 gives details on the physical HAL used in this thesis, while Section 9.7 shows the simulation model.

9.3.1. Initialisation

During initialisation, one instance of the execution runtime is started up on each cluster server. This deployment is strictly separated from productive operation of the cluster; the HAL takes care of that by temporal separation (see steps 4 and 5 below).

Each runtime instance receives task and platform graphs with an assignment of tasks to PEs. It also gets the information which node in the platform graph is the current node.

The task assignment can be created in any way. Section 9.6 explains an automated process I propose.

The runtime then performs the first half of initialisation:

1. Start communication threads. They are restricted to execute on a single CPU core dedicated for this purpose.
2. Start one worker thread per local PE. Each one is restricted to run on exactly one CPU core fully dedicated to that PE.
3. Set up task list for each worker thread.
4. Worker threads wait for a separate start signal.

After that, the HAL proceeds with its own initialisation until all nodes are ready for operation, then it sends a signal for the second half of initialisation. This synchronisation ensures that the cluster network is free of deployment-related traffic.

5. Signal start to each worker thread.

6. Each worker thread calls a HAL method `mark` that allows the HAL to signal PE start to an external control system.

7. Each worker thread enters the main loop.

9.3.2. Main Loop

During normal operation, the execution runtime (or rather, each worker thread) operates in four steps:

1. Fetch the first task from the task list that has not yet been executed.

2. Wait until all input data sets have arrived.

3. Allocate memory buffers for output data sets.

4. Instruct the HAL to execute the task.

5. Release input data sets.

6. Submit all output data sets to the communication scheduler

This repeats until all tasks have been executed. At the end, each worker thread waits until the others signal completion, then they issue another `mark` call to signal completion to an external control system.

9.3.3. Communication Scheduler

In order to make communication more predictable, transmission of each data set is also deterministically ordered. While execution uses explicit mapping and ordering, the communication scheduler determines a fixed communication order during intialisation. It also determines the mapping between communication transactions and transmission paths.

The precise algorithm of the communication scheduler may have a significant impact on application performance and might even decide if a given application can be executed at all[4]. It does not impact prediction accuracy, however, since simulation and physical execution both use the same scheduler.

In this thesis, the communication scheduler simply orders communication by their dependency identifier (see Data Dependencies on page 76) and uses a straightforward static routing table.

[4]Due to memory requirements for communication buffers, for example.

Table 9.1.: Main API calls of the execution runtime HAL

Name	Parameters	Effect
(initialisation)	cluster server id, task graph, platform mapping	starts runtime, sets up task list
`mark`	none	notify external control system that PE execution started/ended
`startThread`	thread function, CPU core index	runs the supplied function in a hardware thread on the given CPU core
`receive`	none	returns received data set handle
`send`	destination node, output handle	sends data set to destination
`run`	kernel, input handles, output handles variable assignments	executes kernel
`allocate`	size	allocates a memory buffer and returns its data set handle
`release`	data set handle	frees a memory buffer

9.3.4. Hardware Abstraction Layer

The HAL consists of three APIs: Initialisation, Communication, and Execution. Table 9.1 summarises the core API calls. The actual implementation is more complex due to practical issues.

Initialisation provides all required information to each instance of the execution runtime. This is not a single API call but spread across object constructors and additional information methods.

Communication provides high level interfaces to send and receive a single data set. Data sets are represented by abstract handles which may or may not reference actual data in memory.

Execution provides an interface to run a single kernel with input data provided as abstract handles, returning output data as abstract handles. Auxiliary functions are for thread and memory management.

9.3.5. Overall Design Decision

The prediction approach needs clearly defined execution semantics; it must be reasonably ensured that simulation and real-world execution actually execute the same program.

An alternative to specifying these semantics from scratch would have been to map them to the facilities offered by a popular middleware, i. e. MPI. MPI does not have a specification for its temporal behaviour and multiple vendors may implement MPI in different ways. This makes modelling its behaviour very difficult.

On the other hand, if applications can reach high utilisation and overhead of the execution runtime is negligible, there is no significant advantage of using MPI.

9.4. Abstract Platform Model

In order to execute a task graph on a target platform, tasks must somehow be assigned to execution units of the platform. Among other purposes, the platform model specifies what execution units are available. Additionally, it models communication paths, and it contains parameters for simulation models.

9.4.1. Platform Model

The platform model actually consists of two graphs that share a set of nodes but have disjoint sets of edges.

The PLATFORM COMMUNICATION GRAPH is a directed graph that expresses the communication structure of a platform. The PLATFORM HIERARCHY GRAPH expresses physical composition of the target platform as a rooted tree. It serves two main purposes: It identifies cluster servers for deployment, where a single execution runtime manages all resources of the local node; and it allows users to control abstraction levels for energy models by grouping multiple components that shall not have individual energy models.

9.4.1.1. Platform Hierarchy Graph

The platform hierarchy graph (PHG) is an annotated rooted tree with implicitly directional edges that express composition: Any node contains its directly connected nodes further away from the root. This can be simple physical accumulation (e. g. a system contains the computers it is made up from) or components inside a system-on-chip. Formally speaking, it is a tuple

$$PHG = (N, H, r)$$

where N is the set of nodes, H is the set of edges (the hierarchy), and $r \in N$ is the root element. Each node $n \in N$ is annotated with an ARCHITECTURE $a(n) \in A$, where A is the set of known architectures. Architectures identify resource characteristics of a given node, see Section 9.4.2 for details.

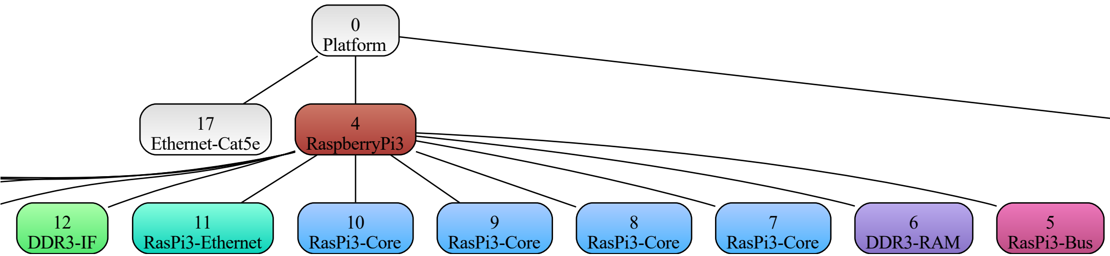


Figure 9.5.: Excerpt of an example platform hierarchy graph.

Each node has a type, so that the generic set of nodes is in fact a union of several subclasses of nodes: $N = N_P \cup N_C \cup N_B \cup N_M \cup N_H$

Processing Elements (N_P) (PEs) are CPU cores that can execute tasks.

Communication Elements (N_C, N_B) (CEs) are nodes that take part in explicit transmission of data, like network interface ports, Ethernet cable, or network switches. These are the nodes that both platform graphs share. CEs themselves are subdivided into BRIDGES and CHANNELS (see below).

Main Memory (N_M) nodes represent shared RAM that PEs can use without the need for explicit transmission of data. The parent node of a RAM node must contain all PEs that communicate by sharing memory. RAM nodes are also CEs, and they are the end points of explicit data transmissions. Memory nodes have an additional annotation that specifies their size. Both, physical execution and simulation use this for memory management purposes.

Hierarchy Nodes (N_H) are all other nodes. The leaf nodes consist of PEs and CEs, while internal nodes serve to group components of the system for energy modelling. Through the architecture annotation, users can associate energy models with each node. Such an energy model could then model energy consumption of components not explicitly specified in the PHG (e. g. voltage regulators), or it could provide a unified model for all descendants. Finally, hierarchy nodes also allow modelling of implicit resource conflicts, e. g. slowdowns due to competing memory accesses.

9.4.1.2. Platform Communication Graph

The platform communication graph (PCG) is a directed multigraph that expresses the communication structure of a platform. Each edge represents a BRIDGE between two communication CHANNELS, which are the nodes. Or in a formal way, the PCG is a tuple

$$PCG = (N_C, N_B, i, o)$$

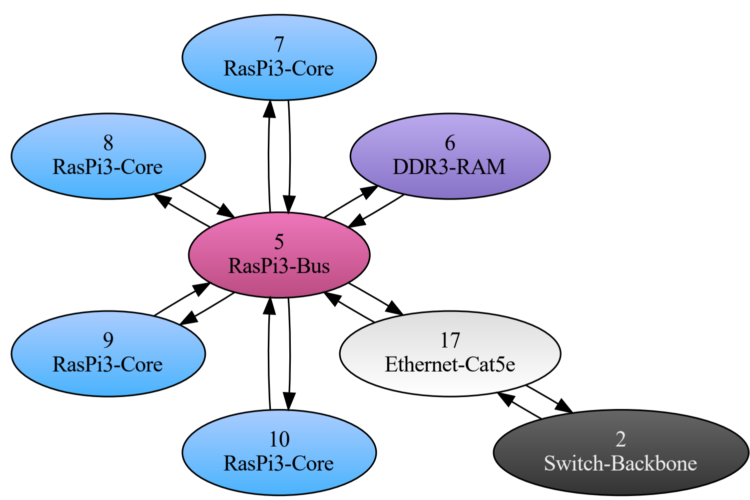


Figure 9.6.: Example of a platform communication graph.

where $N_C \subset N$ is the set of nodes (channels), $N_B \subset N$ is the set of edges (bridges), and functions $i : N_B \to N_C$, $o : N_B \to N_C$ specify the input and output channel for each bridge. Figure 9.6 shows the PCG counterpart of Figure 9.5.

This definition shows that channels and bridges of the PCG are nodes in the PHG and thus carry architecture annotations. For communication modelling, architectures also identify timing properties of a communication path. See Section 9.4.2 for details.

Bridges usually represent interface circuits like an Ethernet driver IC with its associated electronic parts and the connector. Channels either represent transmission media like cables and buses, or they are communication end points. Since bridges always have exactly two associated channels and are directional, bidirectional interfaces are modelled as two bridges with opposing direction. In contrast, channels have no directionality and can have any number of associated bridges. This way, they can model conflicts of shared transmission media.

Channels and bridges may have implicit requirements on their connections. Enforcing these requirements is up to the user creating the platform model. In other words, both ends of a bridge have a type, and channels have typed connection points, but type checking is out of scope for this thesis.

The algorithms in this thesis assume the communication graph is strongly connected, i. e. it is possible to send data from every node to any other node in the platform.

9.4.1.3. Platform Graph

The combination of PHG and PCG is the platform graph. Listing 9.5 shows an XML serialisation of the unified graphs of Figures 9.5 and 9.6. The platform contains an

Ethernet switch and a system-on-chip (SoC) with four PEs, RAM interface, Ethernet interface, and an internal SoC bus to connect these. For simplicity, the separation between SoC and RAM is not modelled, but the outgoing Ethernet cable is not part of the node, it is a sibling channel.

This example shows that the platform graph is meant to express just the amount of structure needed. When using energy models derived from supply power measurements (see page 50), a cluster server can be as little as the PEs, a memory node, a network interface, and the minimum amount of communication infrastructure.

At the abstraction level shown here, the Ethernet interface actually represents the entire networking stack of the execution runtime, the operating system, and the hardware.

Listing 9.5 also shows a convenience shortcut: Child elements of channel nodes specify which bridges interface with a given channel, and in which direction. Since bidirectional links are commonplace, the XML serialisation allows bidirectional connections via the `<inout>` element. Conceptually, this specifies two instances of the referenced bridge with opposing directional links on either. In fact, the example model does not contain pure unidirectional or asymmetric links at all. The latter does occur in real world hardware. See Listing 9.6 for an example.

9.4.2. Platform Resource Model

The main platform graph only contains explicitly modelled information. Simulation models might need data derived from measurements on the physical target platform. The main targets for this are power measurements and communication speed. Since these are application-independent, they only need to be measured once.

Resource model parameters are associated with platform graph nodes through the architecture annotation. As a result, any parameters must represent the same level of abstraction. It is usually not sufficient to use nominal best case metrics. Instead, these parameters should be measured using a representative benchmark that takes all effects into account that are hidden behind the chosen abstraction.

Listing 9.6 shows some examples of entries in the platform resource model, including an asymmetric Ethernet interface that sends packets faster than it can receive them. In theory, any parameter that a simulation model needs could be stored here.

When comparing different applications and/or mappings on a single platform with the sole intent of determining the best alternative, even idle power can be left out, since it does not influence the relative ranking.

The actual characterisation process to determine platform parameters depends on details of the target hardware. Section 10.4 shows a process that works for the evaluation cluster proposed in this thesis, and which should be a good starting point for other target platforms.

9.4.3. Discussion of Design Decisions

Dual Graphs The most unusual feature of the platform graph is probably the split between two different views onto a common data set. Initially, the hierarchical structure

Listing 9.5: Example of a small platform graph in an XML serialisation

```
<node id="Switch" architecture="Switch-1Gb">
    <channel id="Switch.Backbone" architecture="Switch-Backbone">
        <inout peer="Switch.Port0"/>
    </channel>
    <bridge id="Switch.Port0" architecture="1GbSwitchPort"/>
</node>
<node id="RPi3_0" architecture="RaspberryPi3">
    <channel id="RPi3_0.RasPi3-Bus_0" architecture="RasPi3-Bus">
        <inout peer="RPi3_0.DDR3-IF"/>
        <inout peer="RPi3_0.Ethernet"/>
        <inout peer="RPi3_0.BusIF0"/>
        <inout peer="RPi3_0.BusIF1"/>
        <inout peer="RPi3_0.BusIF2"/>
        <inout peer="RPi3_0.BusIF3"/>
    </channel>
    <main-memory id="RPi3_0.DDR3-RAM" architecture="DDR3-RAM"
                 size="1073741824">
        <inout peer="RPi3_0.DDR3-IF"/>
    </main-memory>
    <pe id="RPi3_0.RasPi3-Core_0" architecture="RasPi3-Core">
        <inout peer="RPi3_0.BusIF0"/>
    </pe>
    <pe id="RPi3_0.RasPi3-Core_1" architecture="RasPi3-Core">
        <inout peer="RPi3_0.BusIF1"/>
    </pe>
    <pe id="RPi3_0.RasPi3-Core_2" architecture="RasPi3-Core">
        <inout peer="RPi3_0.BusIF2"/>
    </pe>
    <pe id="RPi3_0.RasPi3-Core_3" architecture="RasPi3-Core">
        <inout peer="RPi3_0.BusIF3"/>
    </pe>
    <bridge id="RPi3_0.Ethernet" architecture="RasPi3-Ethernet"/>
    <bridge id="RPi3_0.DDR3-IF" architecture="DDR3-IF"/>
    <bridge id="RPi3_0.BusIF0" architecture="CPU-BusIF"/>
    <bridge id="RPi3_0.BusIF1" architecture="CPU-BusIF"/>
    <bridge id="RPi3_0.BusIF2" architecture="CPU-BusIF"/>
    <bridge id="RPi3_0.BusIF3" architecture="CPU-BusIF"/>
</node>
<channel id="Ethernet-Cable_0" architecture="Ethernet-Cat5e">
    <inout peer="Switch.Port0"/>
    <inout peer="RPi3_0.Ethernet"/>
</channel>
```

Listing 9.6: Excerpt of an XML serialisation of a platform resource model. Parameters are given in W, nJ, or ns, where applicable.

```
<node-architecture id="RaspberryPi3"
                   name="Raspberry Pi 3"
                   idle-power="1.3" />

<bridge-architecture id="RasPi3-Ethernet"
                     name="RasPi3 1Gb Ethernet Interface"
                     init-latency="830000"
                     packet-size="1492"
                     packet-latency="45000"/>

<bridge-architecture id="Exynos-Ethernet-1Gb-in"
                     name="Exynos 1Gb Ethernet Interface"
                     init-latency="2040000"
                     packet-size="1492"
                     packet-latency="150000"
                     packet-energy="7000"/>

<bridge-architecture id="Exynos-Ethernet-1Gb-out"
                     name="Exynos 1Gb Ethernet Interface"
                     init-latency="916000"
                     packet-size="1492"
                     packet-latency="100000"
                     packet-energy="12000"/>
```

only served to express the recommended way of building models in SystemC, while the platform communication graph was the primary platform model.

The execution runtime model created the true need to express a 'contains' relationship. Nodes in the PCG are not all equal – some share RAM, and shared memory is the granularity at which common operating systems work. Data transmission through shared RAM occurs as part of the computation, so there is no way to separate computation from communication on a local node.

The execution runtime model thus runs as a single process on each shared-memory node with one hardware thread per PE. It then manages local communication simply by passing memory references between threads. The platform model needs to specify what nodes are part of such a local node, and the platform hierarchy graph contains exactly this information.

PCG Edges as PHG Nodes The platform hierarchy contains every element that has a corresponding simulation model with model parameters. That way, simulation has a uniform way to access these. Bridges are important for modelling of communication time, so they need to be nodes in the PHG.

On the other hand, bridges are single input, single output data links, so in the communication graph they are best represented by edges. The alternative would have been to make the PCG a bipartite graph of channels and bridges as nodes, which would have a more complicated structure with no obvious benefit.

Strong Connectivity Constraint This is an easy constraint to make, as it probably matches all of the target platforms users would like to use. A platform that does not have a strongly connected communication graph is quite exotic, and mapping and routing gets much easier if this can be excluded.

This is not a strict requirement if the mapping is aware of connectivity deficits, but that case is not explored in this thesis. Users would have to supply their own algorithms for that.

Choice of Platform Resource Parameters The platform resource model allows for any amount of model parameters. However, at the granularity addressed by this thesis, simple idle power, communication timing, and packet energy have shown to be sufficient.

Since the overall methodology is based on physical measurements, there would be the additional challenge of how to obtain more detailed platform parameters if those were desired.

9.5. Computation Resource Model

As argued before, simulation of the abstract task graph cannot use functional code to determine time and energy of task execution. It uses a computation resource model instead.

9.5.1. Abstract Resource Model

The core resource model is based on a set E of available execution configurations. Each entry $e \in E$ is a tuple $e = (k, v, a)$, where $k \in K$ is a kernel (see Section 9.2.1) that executes on a PE of architecture $a \in A$ (see Section 9.4) using kernel variable assignment $v : V_k \to \mathbb{R}$.

E is not intended to contain all combinations of k, a, and v. Combinations that are missing denote invalid mappings. These could be intentional to control execution of tasks on special-purpose nodes (storage or other I/O for example); they could also be of a technical nature, e. g. missing implementations for a given CPU type or variable assignments that would exceed system resources on certain architectures.

The actual resource model is a function $m : E \to \mathbb{R}^2$ that returns an execution time measured in seconds and an execution energy measured in Joule for each execution configuration.

Secondary Resource Model A secondary resource model describes the way that other tasks running at the same time slow down execution of a kernel. The secondary model is a simple slowdown factor that is multiplied with the baseline execution time.

Energy is not adjusted. Dynamic energy consumption mainly depends on switching activity, and a slower task will still perform the same basic calculations inside the CPU. Of course, slowdown due to resource conflicts does incur additional activity like context switching, waiting for cache loads, or waiting for free bus cycles. Due to the targeted task granularity, I assume that this has negligible power overhead compared to the main work.

Formally speaking, function $m_{\text{load}} : K \times K^{c-1} \times A \to \mathbb{R}^2$, where c is the number of CPU cores of A, returns a slowdown factor s so that $s \cdot m$ specifies the execution time of a kernel $k \in K$ that is executed on a CPU with architecture $a \in A$ concurrently with a set of kernels $(k_1, \ldots, k_n) \in K^{c-1}$.

Due to practical constraints, m_{load} may not be fully defined. In theory, an exhaustive characterisation of all combinations of concurrent kernels could be performed, but this can easily lead to combinatorial explosion for CPUs with many cores.

There are two feasible strategies to limit characterisation effort. Many combinations might not occur at all, others might be so rare that they can't significantly affect predictions. A simulation without the secondary time model can already rule out many possible combinations, and it can also give an impression of what combinations are most significant.

For typical HPC workloads, it is common to have a small set of relevant interactions; for example, matrix multiplication often dominates execution time of linear algebra workloads, so interaction with matrix multiplication is most important and all other combinations might not be relevant. In order to judge if a given interaction can have a significant effect at all, users can assume their hardware behaves sanely: Activity on two CPU cores should never slow them down by a factor of 2.0 or more, since otherwise

both pieces of code could better have been executed serially than in parallel[5].

Secondly, in larger sets of kernels there might be very similar, yet subtly different kernels (e. g. variants of matrix multiplication). In order to address their interference, users can define $m_{\text{load}} : C \times C^{c-1} \times A \to \mathbb{R}^2$, where C is a set of kernel CLASSES, and $c(k) \in C$ is an additional annotation that assigns a class to each kernel. Apart from changing kernels to kernel classes, the application of m_{load} stays the same.

9.5.2. Model Building

The intention behind the resource model as specified is that users can derive model values from automated measurements on a single node.

They first decide on the set E of execution configurations that are relevant for their application. Then they create a set of representative input data sets for each kernel and variable assignment. Finally, they run each execution configuration once for each input data set on a single PE of a single cluster server while measuring time and energy. The average results constitute function m.

Then they perform additional measurement runs based on the identified relevant interactions. In these, the other PEs run a load in parallel that competes for the same implicitly shared resources, e. g. RAM bandwidth, cache, or low-level execution units. They measure how long the main kernel runs, and from the ratio to m they create m_{load}.

Users may choose other ways to determine model parameters. For example, they could apply techniques like in [65] to extrapolate values from existing timings with decent accuracy. A shared database of well known kernels and their properties is another obvious source.

9.5.3. Discussion of Design Decisions

Core Model The resource model is at the core of the feasibility versus accuracy trade-off. As evaluation has shown, using an average value of multiple measurements works well as approximation. At the targeted granularity, data dependent variation of execution time cancelled out in most cases.

An advantage of such a simple model is the ability to address unconventional processing elements in future work, e. g. accelerators like GPUs or FPGAs. Since the model is so abstract, code running on them can be measured just like execution on a CPU.

Furthermore, since the model is expressed as a simple function, unconventional processing elements could use an entirely different model as long as it yields time and energy of a task.

Secondary Model Modelling competition for cache lines or RAM bandwidth suffers from the abstractions just like everything else. The slowdown factors have shown good results in practice, but creating a generic $K \times K \times \ldots$ matrix of all possible combinations

[5]Of course it's still possible that this situation happens. However, if this happens regularly, users have far worse problems than prediction accuracy of the presented methodology.

of running kernels is infeasible. As evaluation has shown, the subset selection process works as a compromise.

Kernel Classes The concept of kernel classes reduces the number of measurements required and at the same time can improve model accuracy. Classifying kernels is a manual task, however, and requires the user to understand kernel behaviour with regard to resource conflicts. There have been attempts to use machine learning for kernel classification [15], but that is out of scope for this thesis.

Measurement-based Approach It is difficult to get reliable simulators or analytical models of HPC CPUs. Physical measurement at the level of detail shown in this thesis is a safe fallback, however. It uses only a single CPU of each architecture, and it works with simple supply power measurement. This should enable users to model an entire HPC system without access to the full platform and without need for special measurement infrastructure. If both are present, they can be used to speed up the characterisation process, of course

9.6. Mapping

One final part of the input models is still missing, which is the assignment of tasks to processing elements for execution, and with it the order of tasks on each PE.

9.6.1. Representation

The mapping is expressed as an additional annotation on the tasks in a task graph. Function $s : T \to N_P \times \mathbb{N}$ assigns a PE and a relative execution order to each task. Reusing the definitions above, T is the set of tasks in the task graph, N_P is the set of PEs in the platform, and $n \in \mathbb{N}$ defines the order of tasks on the PE, with lower n being executed before higher values. Listing 9.7 shows an example, where the task from Listing 9.2 is mapped to the platform from Listing 9.5.

Listing 9.7: Task graph task with mapping annotation in an XML serialisation.

```
<task id="GEMM-0" kernel="GEMM">
    <assign var="tile_size" val="1024"/>
    <map pe="RPi3_0.RasPi3-Core_0" priority="1" />
</task>
```

9.6.2. Automatic Mapping

The purpose of the methodology proposed in this thesis is to predict timing and energy for a given mapping. Strictly speaking the process of creating such a mapping is out

of scope. Once a resource model exists, however, automatic creation of mappings that approximate optimal solutions becomes feasible.

Such an automated mapping algorithm would need to do predictions itself, somewhat overlapping with the goals of this thesis. However, it also needs to have linear runtime in the number of tasks due to targeted task graph sizes. This means it cannot explore significant portions of the space of valid mappings; it must use approximations and heuristics instead. Predictions done by the mapper would need to be very simple to yield usable mapping speed.

In [2] I presented an automated mapping/scheduling algorithm that has linear time complexity in the number of tasks and thus is suitable to create mappings at the scale targeted by this thesis. By using an extremely simplified communication model, it performs suitably for the goals of this thesis.

The algorithm uses a constructive hybrid heuristic: A list-based mapper processes a topologically sorted list of tasks, using the earliest-finishing-time-first strategy. In a second step, a simulated annealing heuristic explores random permutations of topological orders. While the list-based mapper/scheduler only considers timing, the heuristic optimises a user-selectable cost function that can include timing and energy. Furthermore, users can guide the mapper through manually specified mapping constraints.

At this point, all models are in place to describe the exact execution of an application on a platform.

9.7. Simulation Model

The final part of the prediction methodology is the simulator that creates the actual time and energy predictions. The simulation model is a dynamic SystemC model that builds the final simulation model on the fly during elaboration, then performs initialisation of the execution runtime model and passes control to it. During application execution on the simulated system, SystemC tracing facilities record time and power traces.

9.7.1. SystemC Object Hierarchy

Figure 9.7 shows the SystemC module classes that simulation models use. For each type of platform graph node (`PE`, `Memory`, `Bridge`, `Channel`, `Node`; see Section 9.4), there is one class. They are generic and parameterised through the platform graph's architecture data (see Section 9.4.2), although future extensions of the methodology presented here might use custom subclasses.

`PE` and `Memory` implement computation models while `Bridge` and `Channel` instances model communication. `Node` instances serve as containers for the other classes.

Since PEs count as channels in the platform communication graph, class `PE` is a subclass of class `Channel`. For practical reasons, class `Memory` is a subclass of `PE`: `Memory` implements most of the abstract `HAL` interface, while `PE`s implement the `run` call and forward everything else to their associated `Memory` instance. `Memory` poses as `HAL` instance to an `ExecutionEngine` instance, while `PE` does the same for execution

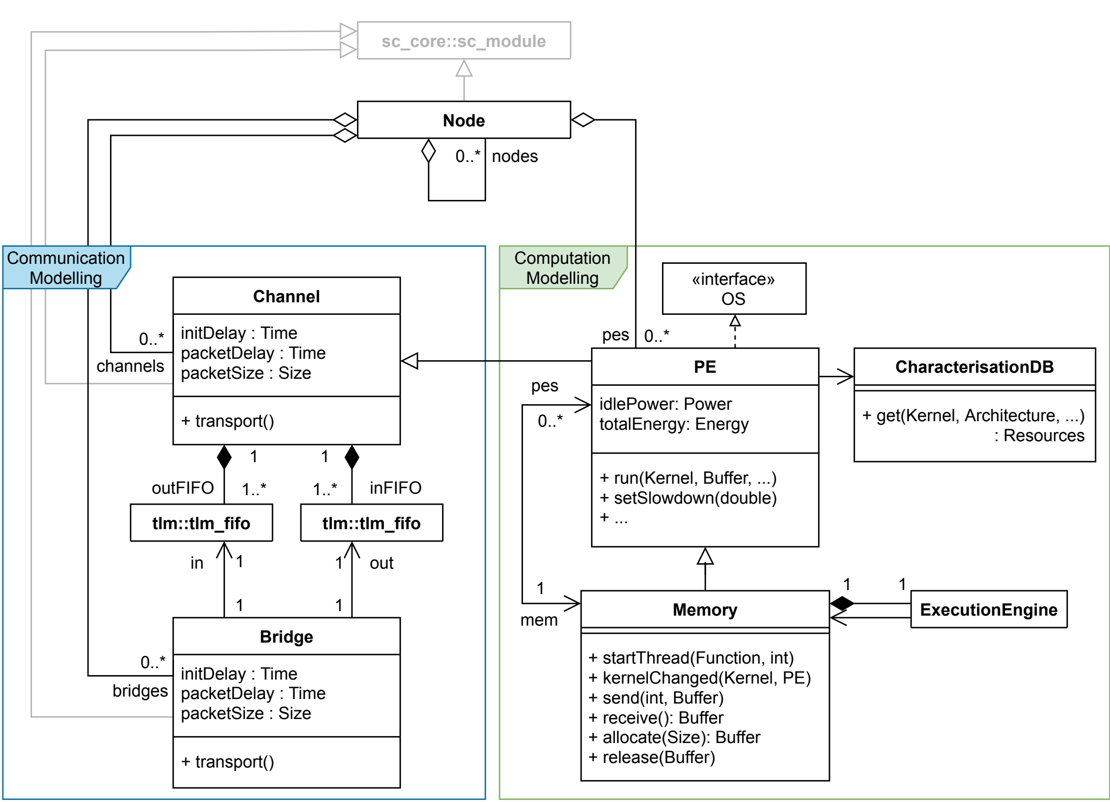


Figure 9.7.: Class diagram of SystemC simulation models (simplified).

threads. Memory management uses the same memory allocator as the physical execution runtime, except returned memory handles are not backed by actual memory.

During elaboration, the main program instantiates a `Node` representing the whole system. Each `Node` instantiates one module for each immediate child node in the platform hierarchy graph, leading to recursive model creation. An additional constructor argument is a SystemC trace file handle for power signals.

During model instantiation, `Bridge`s and `Channel`s are recorded in a global associative map. After instantiation, this map is passed recursively across the hierarchy so that each `Bridge` element can bind to its corresponding `Channel` according to the communication graph. This map also serves to build routing tables using a minimal spanning tree algorithm. Furthermore, each `Node` that contains a `Memory` instance will recursively collect all descendant `PE` nodes and link them to their local `Memory`.

Finally, the main program passes the task graph down the hierarchy, which starts the initialisation phase of the execution runtime model (Section 9.3).

9.7.2. Power Models

Due to the flexibility of SystemC, each module could have a power model attached to it. The model building approach used in this thesis only measures supply power, so there is not enough data for detailed power models. There is only idle power consumption at the granularity of cluster servers (Section 9.4.2), and dynamic power expresses by the computation resource model (Section 9.5).

For this thesis I even leave out idle power, since it is a constant offset independent of workload. A very simple runtime model extension could temporarily shut down unused cluster servers or employ other power saving procedures. For such an extended runtime model, the simulation could easily support multiple power states of non-computation resources. In any case, model parameters would be supplied through architecture data (Section 9.4.2).

Since the prediction methodology is actually supposed to predict energy, class `PE` implements this directly using the computation resource model. When a task starts execution, it adds the kernel's energy value to an internal energy accumulator variable. These accumulators are collected and summed up at the end of the simulation.

In theory it is also possible to calculate (and regularly update) average power from time and energy values in the resource model and create a true power over time trace this way. That would still be inaccurate, however, because kernels can have varying power while they run, which the resource model doesn't capture. This might be solvable without too much extra effort, but it is out of scope for this thesis.

9.7.3. Computation Modelling

As shown earlier, memory nodes uniquely represent cluster servers – each server has exactly one memory node in the platform graph. For that reason, the global parts of the execution runtime model are embedded into the `Memory` class.

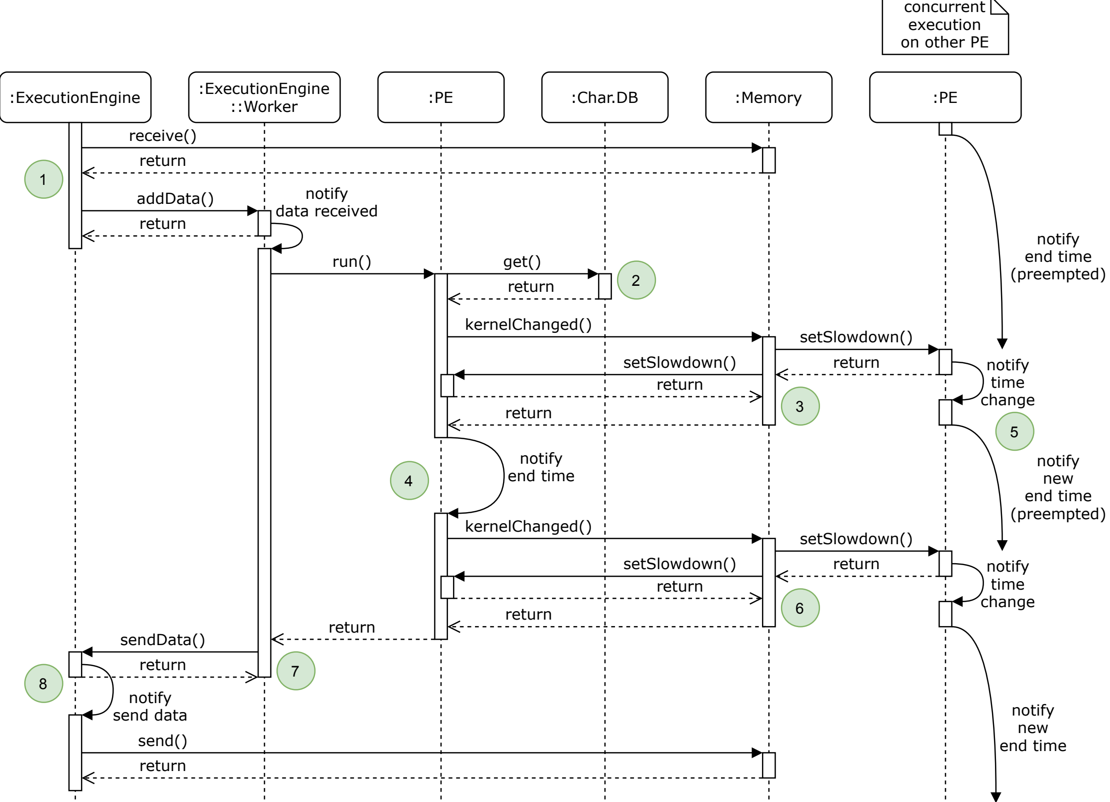


Figure 9.8.: Sequence Diagram of a single task execution with communication and concurrent execution on a second PE. Numbers refer to the explanation in the text.

Worker threads are associated with a specific PE. For them, class `PE` implements the HAL API as a proxy. It only provides the `run` call itself and forwards all other calls to the common `Memory` object.

Initialisation Initialisation begins during elaboration, after instantiation of all modules and binding communication channels has finished.

The `startThread` call creates a dynamic `SC_THREAD`. If a specific CPU core is requested, the corresponding `PE` instance provides the HAL APIs.

The `mark` API call does nothing.

Communication The `Memory` class will forward the communication calls to the communication infrastructure described below.

Execution Figure 9.8 shows the process of executing a single task, starting with the arrival of required data up until output data has been submitted for sending. It shows

one `PE` (left) that runs a task uninterrupted, while another one (right) runs a task that is subject to multiple slowdown factor changes.

The core of computation modelling happens in the `run` call provided by class `PE`. Execution of a task consists of several steps (highlighted in Figure 9.8 as circled numbers):

1. Wait for all data packets required by the next task to arrive via `receive()` and `addData()`.
2. `get()` task resource usage from the characterisation database and record task execution time and energy.
3. Signal start of execution to the common `Memory` instance, so that it can check if the slowdown factor of the resource model needs to be applied (see Section 9.5).
4. Wait until either the execution time has elapsed or the `Memory` instance signals that a new slowdown factor is in effect.
5. If the execution time has not elapsed, calculate the remaining time including the new slowdown factor and repeat at the previous step.
6. Signal end of execution to the `Memory` instance to have it update all slowdown factors.
7. Record simulated end time of task.
8. Submit output data to execution runtime for sending.

The common `Memory` instance receives notification from all `PE`s about their workload. It will consult the secondary resource model shown in Section 9.5 each time a `PE` changes its workload and update the slowdown factors of all `PE`s if necessary.

9.7.4. Communication Modelling

Communication local to a cluster server is managed by the execution runtime model itself through shared memory. Non-local communication starts with a call from the communication scheduler to the `send` call (Section 9.3). It is implemented by class `Memory`, which counts as a `Channel` from the perspective of the platform communication graph. Figure 9.9 shows the process as a sequence diagram.

9.7.4.1. Basic Operation

A complete transmission originates and terminates at a `Memory` instance[6]. Between them there are alternatingly `Channel`s performing routing and `Bridge`s modelling communication timing. A `Bridge` has four parameters: a *startup latency* that is applied once per frame, a *packet latency* that is applied once per low-level data packet, the payload

[6]Besides pragmatic implementation considerations, this probably reflects reality, since modern network interfaces use direct memory access (DMA) transfers for packet data.

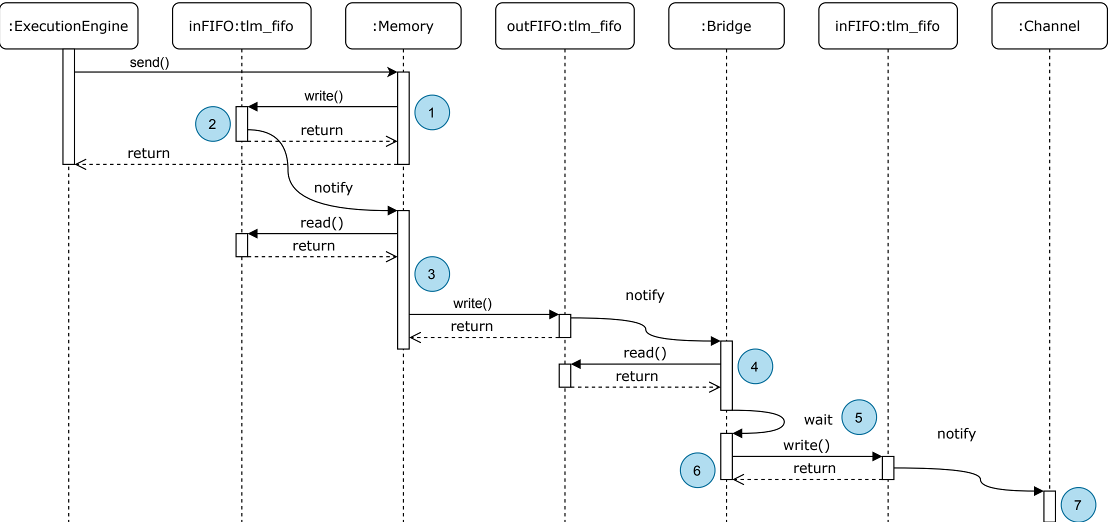


Figure 9.9.: Sequence diagram of a single data set transmission (initial part). Numbers refer to the explanation in the text.

size of its low-level data packets, and an optional *packet energy* that is added to an internal energy accumulator variable for each packet crossing the bridge.

In the simulator, a data set is always transmitted en bloc. The underlying assumption is that data size is large enough that latency jitter of individual packets evens out statistically. Since there is no actual data, an abstract frame is transmitted. It consists of four values:

- A PE identifier that specifies the frame's destination,
- a data set identifier that specifies the frame's content,
- the size of its (virtual) payload, and
- a latency value that is updated at each step to contain the worst-case *packet latency* of the entire communication path.

Transmission proceeds in several steps (highlighted in Figure 9.9 as circled numbers):

1. The originator assembles the abstract data frame.
2. The originator queues the packet in an input FIFO (of size 1) that the inherited `Channel` class evaluates.
3. The `Channel` reads the abstract frame from the input FIFO, determines the destination, and writes the frame to an output FIFO (of size 1).
4. The `Bridge` connected to that output FIFO reads the abstract frame and checks if its own packet latency is higher than the worst-case packet latency the frame has encountered so far. If so, it will update the worst-case packet latency and wait for

a time span equal to the difference between the old and the new worst-case packet latency.

5. The `Bridge` waits for a time span equal to its startup latency.
6. The `Bridge` writes the frame to its output port, which is connected to the input FIFO of another `Channel`.
7. If the `Channel` is not the destination, this process repeats at step 3.
8. The destination `Channel` (actually a `Memory` instance) reads the abstract frame and stores it so that the `receive` call of the execution runtime model communication API can return it.

The sequential nature of this sequence has the desired side effect that no other transmission can use the involved communication elements at the same time.

For example, an Ethernet switch can be modelled as a set of `Bridge`s (its ports) and a `Channel` representing the internal backbone bus. Assuming the backbone bus is fast enough for full duplex operation on all ports, it is perfectly fitting that the `Channel` has no time model of its own – it has enough capacity to appear as a resource usable in parallel. The ports are not usable in parallel, and the procedure outlined above ensures that.

9.7.4.2. Routing

In many communication systems, a bus (which would be a `Channel`) is a passive component, while bus masters (which would be a kind of `Bridge`) contain the addressing logic that selects the communication path. In the abstracted view I have chosen for the simulator, these roles are reversed for better modularity and encapsulation: A `Bridge` always has exactly two connected `Channel`s, so it always knows where to forward a data packet to (the opposite end from where it entered).

On the other hand, `Channel`s can have any number of `Bridge`s connected to them, so they need to decide where to forward a data packet. They have a static routing table to determine which destinations can be reached via which `Bridge`.

In the implementation evaluated in this thesis, the routing tables are created during elaboration by building a simple minimal spanning tree. They happen to be equivalent to the default routing tables used by the evaluation system. For more complex communication setups, routing tables might have to be specified manually so that simulation matches the cluster servers.

9.7.5. Resulting Prediction

Timing data is recorded by `PE`s during execution and written as additional annotations in the task graph. This is the same format that the physical execution runtime uses.

The accumulated energy prediction from the `run` function is printed out to the console at the end of the simulation run.

9.8. Physical Execution

When finally running the optimised application, a physical implementation of the execution runtime model (Section 9.3) calls implementations of kernels according to the defined execution and communication semantics. The physical execution runtime should try hard to avoid performance or energy variations not covered by the prediction methodology.

For this thesis, I created a Linux-based execution runtime that provides the hardware abstraction layer APIs as shown in Figure 9.4. Additionally, a platform-specific deployment phase occurs before starting application initialisation.

9.8.1. Deployment

The execution runtime is a single Linux executable that could be statically linked, so that system dependencies are minimal. Kernel code can be provided as dynamically loadable shared objects (DSOs), or they can be statically compiled into the runtime. The runtime probably works on different operating systems as well, but this has not been explored during this thesis.

For execution, the runtime needs only a few files:

- the executable itself, with kernel code compiled into it or as DSOs,
- XML serialisations of supported kernel data, platform graph, and architecture data,
- a list of network addresses and corresponding PE identifiers in order to identify the local node and to set up communication with other PEs.

Distribution of these files can use any means, from cluster management software through hand scripted SSH connections down to network booted nodes (which is what the evaluation platform presented in Section 10 uses). At the end of the platform-specific deployment phase, the execution runtime executable is started on each cluster server.

Execution Controller A cluster running successfully deployed execution runtimes can execute multiple task graphs in sequence. A controller executable running on a dedicated deployment node manages actual application execution. It proceeds in a sequence of steps:

1. Check that nodes are still responsive, or wait for all nodes to signal successful startup.
2. Transmit the task graph to execute to all nodes.
3. Wait for all nodes to signal successful initialisation.
4. Broadcast a UDP packet as synchronised start signal.
5. Wait for all nodes to signal completion.

6. Save measurement data to disk.

The synchronised start helps to improve predictability and timing measurements. During step five, the nodes also transmit start and end time of task executions as measured by their local clocks, relative to the start signal.

Timeouts and runtime consistency checks provide error detection. Upon encountering an error condition, a platform-specific error handler might then collect logging data from each node and shut down the execution runtime executables across the platform.

9.8.2. Initialisation

During startup, the execution runtime first waits for the execution controller to transmit the task graph to execute. Then the sequence of initialisation steps begins:

1. Read all provided data files and build in-memory models from these.
2. Initiate communication links to every other cluster server.
3. Instantiate the execution runtime model (first half of initialisation)
4. Signal readiness to the execution controller and wait for synchronisation packet.
5. Set relative execution time to zero for local time measurement.
6. Signal start to execution runtime (second half of initialisation)

The `startThread` API call simply starts a new POSIX thread and uses the POSIX API to set CPU core affinity.

The `mark` method that each worker threads runs is a ten second code sequence that consumes a variable amount of energy; this is intended to be easy to identify on energy measurement traces in order to synchronise external energy measurements with the local execution time. It also toggles a general-purpose input/output pin, meant as trigger signal for an external time measurement circuit.

9.8.3. Execution

Execution proceeds under control of the common execution runtime model as shown in Section 9.3.

The `run` API call from Table 9.1 will first try to locate kernel code (a native function using C calling conventions) by checking multiple places and stop as soon as it is found:

1. Check table of statically linked kernels.
2. Try to load DSO for that specific kernel.
3. Try to load a fallback DSO.
4. Use a fallback kernel.

It then calls the kernel function with the kernel name, a list of input buffers, a list of output buffers, and a list of kernel variable assignments.

For practical reasons, the execution model supports one extension to the application model as laid out in Section 9.2.1. Kernels can specify bidirectional ('inout') data sets. These share a single memory buffer for algorithms that do in-place modification. This is an important memory saving mechanism because the frequently used generic matrix multiplication kernel supports it.

The implementations for the `allocate` and `release` calls use a dedicated fixed-size memory arena using a simple first-fit allocation algorithm. Since the expected workload is tens of active allocations with only a few different sizes, this has shown to be quite sufficient. Using a custom allocator has the additional benefit that simulation can use the same algorithm (without actual physical memory), so allocation behaviour will be identical.

9.8.3.1. Node-Local Time

For time measurements, there is a function `now` that returns the time elapsed since start of the application.

Beyond the generic functionality of the runtime model, the physical runtime stores start and end time of each task. It also stores start and end of each non-local data transmission. This information is used by the proposed characterisation process (Section 10.3.4); the runtime stores it in additional annotations in the task graph.

Timings obtained this way are not always reliable: during evaluation it became apparent that values can be incorrect under high computation loads. Depending on the hardware employed, an external measurement circuit may be needed (see Section 10.3.2). To support this as well, the physical runtime toggles a general-purpose input/output pin at the start and end of each task.

9.8.3.2. Time Synchronisation

Clock speed usually varies between nodes. There are manufacturing variations that lead to slightly different base speeds, and heat changes oscillator frequency as well. Nevertheless, cluster servers do not synchronise their clocks.

Network-based clock synchronisation (e. g. through NTP) would lead to additional network traffic interfering with the regular load, while the network load could significantly reduce its accuracy. It could also interfere with the computation workload of the node. Finally, temperature variations can happen faster than NTP can compensate for, so the result is not worth the effort.

The Precision Time Protocol (IEEE 1588-2008) might be worth the effort, but like Ethernet TSN, the required hardware support is rare and wasn't available for this thesis.

As a result, node-local time can only be considered a rough estimate; for reliable figures an external measurement mechanism must be used (see Section 10.3.2 for example).

9.8.4. Communication

Communication between tasks uses an abstract interface so that the physical execution runtime can be adapted to different networking interconnects. The runtime assumes the network is isolated and carries no other traffic.

During startup, each cluster server initiates a persistent connection to every other node. When the communication scheduler submits a data set for transmission, actual transmission only happens when communication scheduler on the receiving node is ready for that transmission. The sender first transmits a header of 16 bytes that identifies the data set ready to be sent; it does not proceed to send the data. The receiver signals readiness to receive the data set by sending an acknowledge byte. Then the sender proceeds to send the actual data.

While the underlying network protocols can cope with a receiver that is stalling, this situation leads to interference with existing traffic: Additional traffic interferes with ongoing transmissions, buffer memory used by an inactive transmission reduces buffer memory for active transmissions, and in the end these additional packets may end up being dropped entirely. The explicit handshake avoids this situation by adding just a single packet (plus protocol overhead) in each direction until the transmission path is available for exclusive transmission.

As part of this thesis, I evaluated two communication backends: one based on regular TCP/IP, and one based on a custom protocol implemented over raw Ethernet packets.

9.8.4.1. TCP Backend

The TCP/IP backend uses plain TCP connections. For each pair of nodes, the node with the higher numeric PE identifier initiates the connection, while the other node waits for a connection attempt. Transmission maps to the regular `read` and `write` system calls. Routing happens through the system's static IP routing tables.

9.8.4.2. Eth Backend

During evaluation, the TCP backend showed significant variations in transmission bandwidth (see Section 15.3). These were too large to build a useful timing model. For this reason, I implemented a custom network protocol dubbed 'Eth' that was optimised for deterministic transmission at high bandwidth over an unloaded Ethernet network path with minimal packet loss. I consider it a provisional arrangement, as I designed it in a pragmatic approach after an unsuccessful search for an existing network protocol with suitable characteristics.

The actual solution I envision is Ethernet TSN real-time scheduling (IEEE 802.1Qbv). With TSN, network packets can be assigned exclusive access time on the physical medium, which could facilitate deterministic high performance networks. Unfortunately, this technology is not yet widely available in general purpose networking equipment, so it could not be explored during this thesis.

Table 9.2.: Eth packet format

Offset	Size	Content
0	2	user data size
2	1	destination port
3	1	packet type
4	4	packet metadata
8	0-1492	user data

Table 9.3.: Packet types in the Eth protocol

Name	Numeric	Function
`CONNECT`	0	initiate connection
`CONN_ACK`	1	confirm connection establishment
`CLOSE`	2	close connection
`ACK`	3	acknowledge complete transmission
`NAK`	4	request packet retransmission
`DATA`	5	transmit user data

The core design idea of Eth is to transmit packets in fixed intervals for determinism and to rely on the high reliability of a local Ethernet network in order to eliminate much of the recovery mechanisms of TCP, thus maximising bandwidth. Retransmission of data is still available, but optimised for rare cases of packet loss.

The Eth protocol uses regular layer 2 IEEE 802.3x Ethernet II frames, but with a custom EtherType (0x88b5, IEEE 802.1 Local Experimental 1). Within the 1500-byte Ethernet payload, an eight byte header precedes user data.

Table 9.2 shows the basic structure of Eth packets. It starts with a size field for additional error checking, as the size field from the layer 2 header sometimes did not match the payload size. The port number serves a similar purpose as in IP networking, except that fewer ports are available. The packet type differentiates between control packets and data packets, and the packet metadata field holds additional parameters for the packet type.

Table 9.3 shows the available packet types. The protocol has a state machine with three main states (`CLOSED`, `CONNECTING`, `OPEN`). Opening and closing connections works similar to TCP. The main difference is data transmission, which significantly differs from TCP on multiple levels and has a number of implicit sub-states of `OPEN`.

First of all, Eth transmits in transactions, where each transaction corresponds to a

call to the `send` method of the Eth backend. The receiver must issue a matching call to `receive`; the communication scheduler has enough information to do this.

Successive transactions are logically separated. The advantage is that special handling of start and end of a data set reduces timing variation over TCP, which cannot handle some data differently from other data. Furthermore, transactions help in identifying lost packets and to eliminate acknowledge packets.

Retransmissions of packets of a recently finished transaction may accidentally confuse a new transaction. To prevent that, the lowest bit of `DATA` packet metadata identifies a transaction. It is toggled at the start of one, so that packets belonging to two adjacent transactions cannot be confused. The remaining part of the metadata field is a packet counter that starts at 0 for the first packet in a transaction and which is incremented for every successive packet.

During a transaction, user data packets are sent in regular intervals according to a statically configurable table of transmission timings. Users can use the platform characterisation process (Section 9.4.2) to determine optimal values.

Unlike TCP, data packets are not acknowledged. I assume that the local network has almost no packet loss and that packet timing does not exceed the capabilities of any networking component, so packet loss should be a rare event. The metadata field will inform the receiver if it has missed a packet, in which case it sends a `NAK` packet to the sender to explicitly request retransmission of a single packet.

TCP has a send data buffer and can only retransmit data in its buffer. It can discard data only after receiving an acknowledge packet. Since Eth directly uses the user supplied data buffer, it does not need to discard data and can serve any `NAK` at any time.

Eth has only one acknowledge at the end of a transaction: The receiver signals to the sender that it has received all data of that transaction, so that the sender can return from the `send` method call and discard all buffered data.

A retransmission does not honour the regular packet interval. It is sent immediately and does not replace the scheduled regular packet. Consequently, occasional retransmissions can be hidden between regular traffic and don't affect overall timing, unless it is one of the last packets of a transaction that needs retransmission.

Despite being optimised for very low packet loss, the protocol can cope with channels of any quality. One source for excessive packet loss is a packet interval that exceeds the capabilities of the hardware. This can create an excessive amount of retransmissions and thus ruin transaction timing completely. But still there are only two outcomes: Either the transaction succeeds eventually, or the connection is aborted due to a timeout. The latter result would lead to program abortion.

Power Hiding The Eth protocol runs in its own thread on the CPU core reserved for the runtime. It uses pretty aggressive busy wait loops in order to get low jitter for the packet timer. As such, it has a significant impact on the energy consumption. Since Eth is only meant as a stopgap solution, I chose to reduce its energy profile by spawning a second low-priority task that just consumes energy when no other threads run on the core.

This increases the baseline power of the system, but reduces load-dependent effects on system power. The dynamic power behaviour will be closer to the behaviour of a system which has hardware support for real-time communication, since such a system would spend much less CPU time for communication.

10. Measurement Platform

One important aspect of this thesis is the physical evaluation platform. For HPC-scale applications, evaluating energy predictions is particularly hard, since HPC systems don't usually have the measurement equipment that is required.

Another approach common in embedded system design is to compare an approach to a more accurate but slower simulation, but that again is hardly feasible: Simulating a full HPC system in more detail would be extremely slow, and how would one determine accuracy of that more accurate simulator?

To solve these difficulties, I designed and built a hardware platform that serves two major purposes: to evaluate the methodology, but also to provide a reference setup to fill the (intentional) gaps left by the platform-independent methodology.

The platform consists of a networked cluster of single-board computers (SBCs) that are powered over a central power distribution board with integrated energy measurement. It addresses an implicit assumption of the methodology described in Section 9: It demonstrates that the hardware requirements for measurement are low enough that any developer can use the methodology; inexpensive hardware is sufficient for characterisation of server grade hardware.

10.1. Power Distribution and Measurement

The power distribution and measurement board supports up to 15 channels of synchronous voltage and current measurement with a combined supply power of more than 600 W and a combined sampling rate of 2 million samples/s. The total cost of one board was about 200 Euro at the time of this thesis.

It consists of three separate power distribution paths, analogue front-ends for measurement of voltage and current, and a microcontroller-based data acquisition system. It is a two layer design meant for manufacturing using 70 µm copper layers.

While the design was tailored for the evaluation platform, using it for other targets like an HPC-grade mainboard using ATX power connectors is perfectly feasible. The distribution layout is flexible enough that the additional 3.3 V and 12 V power rails can be routed through two additional measurement channels; a suitably manufactured cable could then provide the ATX connector.

10.1.1. Power Distribution

An ATX power supply provides all power to the measurement board. Its standby 5 V power output powers a step-up DC-DC converter for the data acquisition microcontroller.

Figure 10.1 shows the a schematic of the main power distribution path. It connects the power supply's 5 V rail with ten connectors rated for up to 4 A. I have chosen 10 mm width and 70 μm thickness of the traces. Each channel can supply up to 4 A, but the ten primary channels in total may only draw 20 A. At that load, power loss due to PCB trace resistance is about 1 W; the PCB will heat up to about 10 K above ambient temperature; the connectors have similar performance specifications. A 20 A fuse protects the path from damage through overload.

The board has two secondary power distribution paths with three channels each, also shown in Figure 10.1. I have chosen dimensions so that they can supply 7 A at 12 V on each channel; performance characteristics of all parts are similar to the 5 V rail. A 21 A fuse protects each path.

The actual voltage provided through the secondary channels can be configured, however. A wire jumper with large soldering pads connects each path to the 12 V rail of the power supply. By removing the jumper and connecting it to an alternate source (e. g. the 3.3 V power rail), other voltages can be supplied. Figure 10.2 shows the physical arrangement of this.

In total the board can distribute a maximum power of 604 W. In this thesis, I only use the 5 V part, which allows up to 100 W.

10.1.2. Analogue Front-End

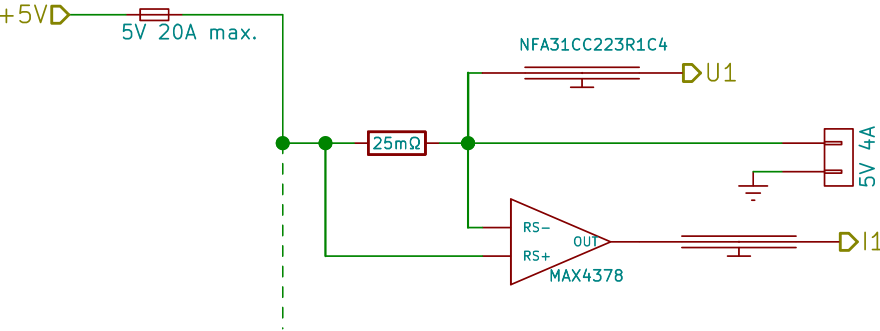
Current measurement works via a shunt resistor in the power supply path (see Figure 10.1). It should be dimensioned such that the maximum expected load results in a voltage drop of 100 mV. The surface mount soldering pads for the shunt resistors are intentionally large and easily accessible from the top side even on a fully populated measurement board (see Figure 10.2). For this thesis, I used 25 mΩ for boards that were specified up to 4A, and 40 mΩ for those specified up to 2.5 A; all shunt resistors had a nominal 0.1% accuracy and a temperature coefficient of less than 75 ppm/K.

A Maxim MAX4378 instrumentation amplifier with a fixed gain of 20 V/V amplifies the current signal into the range 0-2 V. Supply voltage is measured after the shunt resistor. This part of the setup is very similar to the MAGEEC measurement hardware (see Section 8).

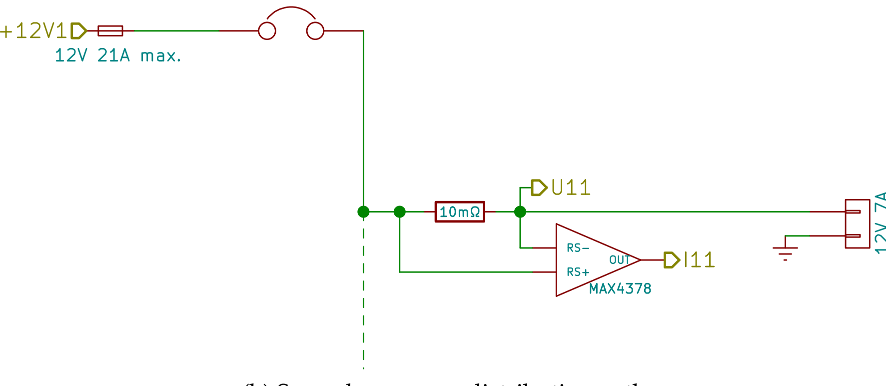
When using all 15 channels, the sampling rate per channel would be 66.7 kHz. As used in this thesis, 10 channels lead to 100 kHz per channel, and single channel operation would use the full 1 MHz. By the Nyquist-Shannon sampling theorem, the signal should not contain frequency components beyond 33.3 kHz, 50 kHz, or 500 kHz, respectively. Otherwise, the time-discrete digitised signal may show artificial distortions (aliasing).

To reduce the impact of aliasing, both signals are passed through a capacitive filter that is specified for 10 dB attenuation at 1 MHz and roughly 3 dB at 500 kHz. This is a safeguard to at least have the highest frequency mode protected. Other components and the PCB layout itself may have frequency-limiting effects as well, so the actual effective bandwidth can only be determined through measurement.

Another potential source of errors would be if signals measured on one channel somehow influenced the signal measured on another channel, sometimes called cross

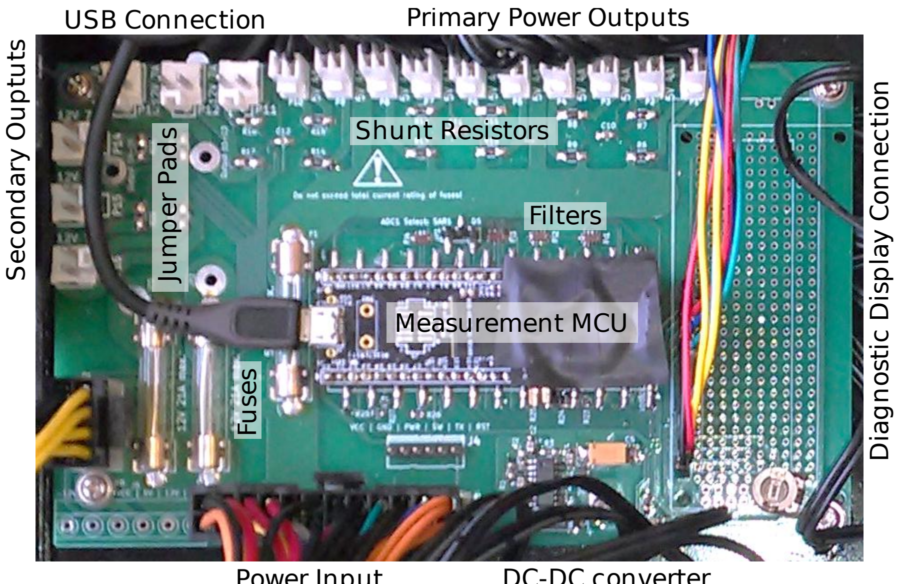


(a) Primary power distribution path including analogue instrumentation front-end.

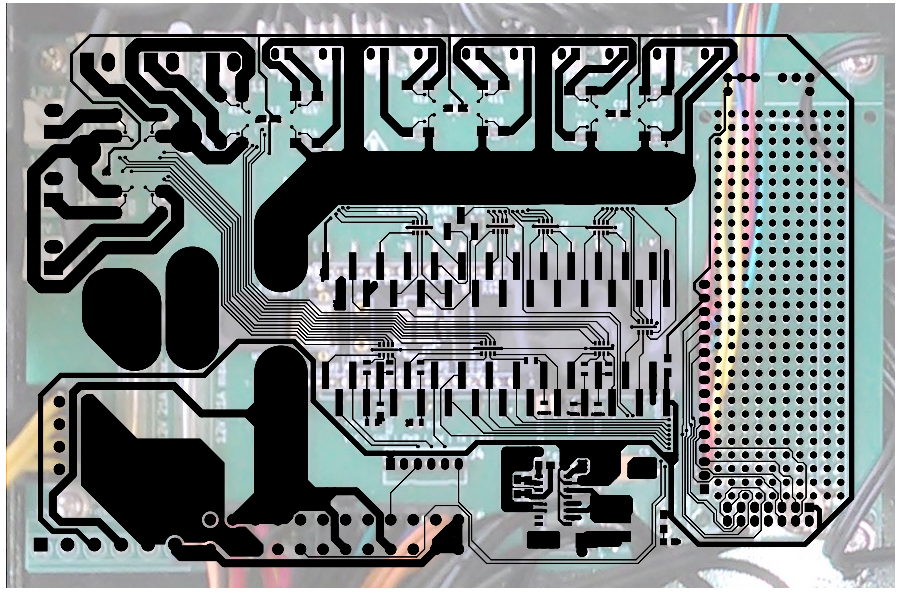


(b) Secondary power distribution path.

Figure 10.1.: Power distribution path of the power distribution and measurement board.



(a) Layout of the major functional units.



(b) Overlay of PCB layout, front copper layer. Notable are the wide traces that can carry up to 20 A. The back copper layer is mostly one big ground plane.

Figure 10.2.: Physical design of the power distribution and measurement board.

talk. Cross talk from the main power rail to the measurement signal lines is realistic danger. To minimise potential influence, signal lines cross power lines in right angles only, with as little parallel traces as possible. As Figure 10.2 shows, the secondary outputs do have a short segment where they run in parallel with power rails, but they are not used in this thesis.

10.1.3. Signal Acquisition

After analogue conditioning, the signal is led to the central microcontroller unit (MCU), a Cypress PSoC 5LP (specifically, the CY8C5888LTI-LP097). It has two successive approximation analog-to-digital converters (ADCs) that work at up to 1 MHz at 12 bit resolution.

A unique property of the Cypress PSoC microcontrollers is that they have a flexible input/output (I/O) signal routing layer combined with user programmable hardware logic in the form of Complex Programmable Logic Device (CPLD) cells. This makes it possible to build a robust signal acquisition circuit with integrity checks. Figure 10.3 shows the CPLD schematics as provided by Cypress PSoC Creator version 4.

10.1.3.1. Multiplexer

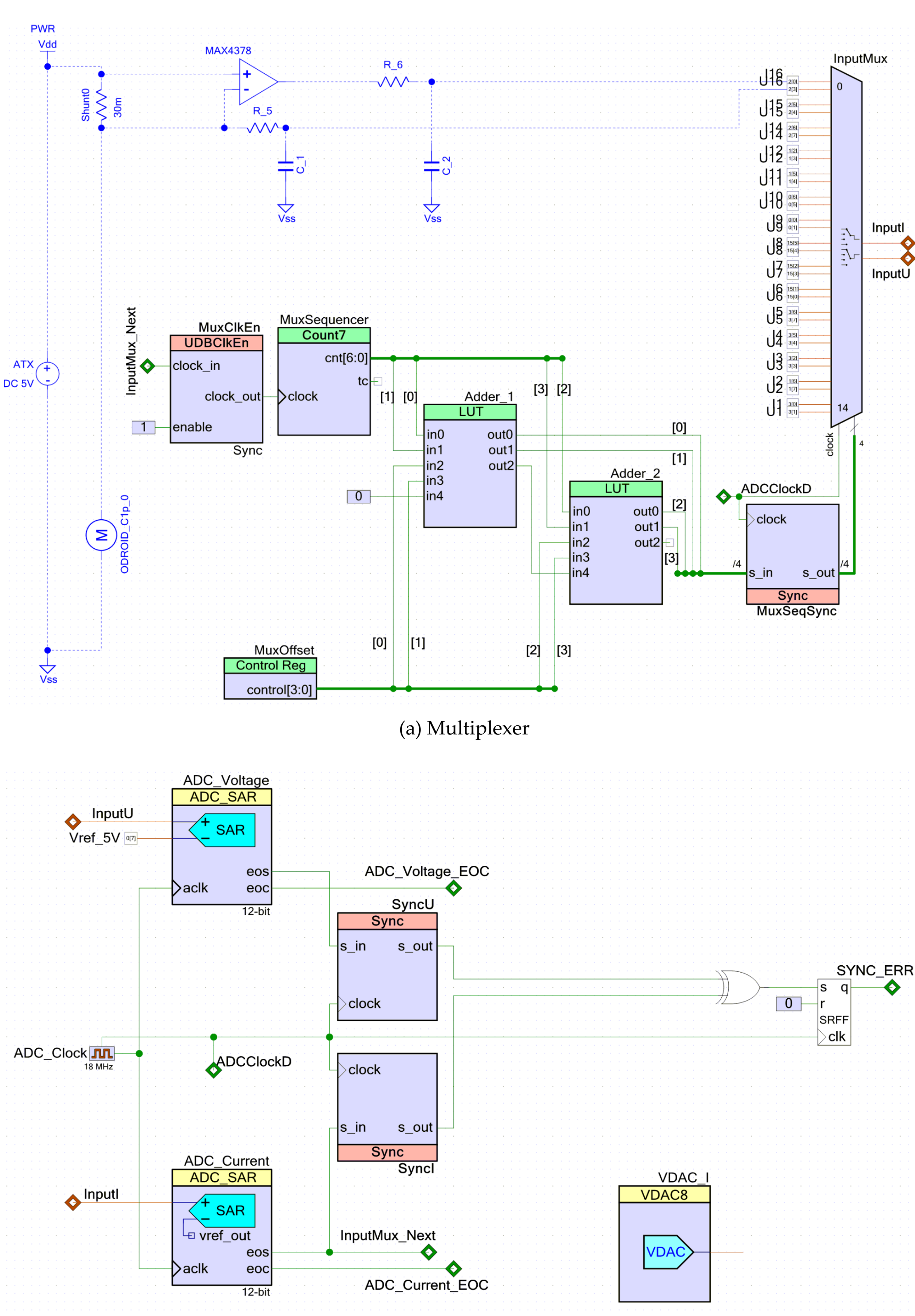
The analogue signal enters through one of 30 pins configured as analogue inputs. They are connected to two 15-channel multiplexers, shown in Figure 10.3a. A custom memory mapped register sets the start channel and multiplexing period; this way the firmware can control the sampling sequence.

The multiplexer must leave enough time for a new input voltage to settle at the ADC. The `EOS` output shown in Figure 10.3b signals that the input voltage has been buffered by the ADC and a new voltage may be applied. This is much earlier than the finished conversion, so via signal `InputMux_Next`, the multiplexer switches to a new channel long before the new value is sampled. I could not determine an exact value, but documentation suggests that signal sampling takes 4 cycles, so the settling period is up to 14 clock cycles.

10.1.3.2. Analog-to-Digital Converter

The output of each multiplexer connects to one ADC, shown in Figure 10.3b. The current ADC works in single-ended mode where its effective range is 0-2.048 V. Each ADC count is thus equivalent to 0.5 mV of the amplified signal, or 0.025 mV of the voltage drop across the shunt resistor.

The voltage ADC works in differential mode, where it measures the difference to a reference voltage. The measurement board contains a precision reference voltage source (Maxim MAX6126AASA50+) that provides a reference voltage of 5.0 V, thus the effective range of the voltage ADC is 4-6 V (more exactly: 5.0 V +/- 1.024 V); one ADC count represents 0.5 mV.



(a) Multiplexer

(b) ADC

Figure 10.3.: CPLD schematics of the power measurement board.

The ADCs cannot measure (absolute) voltages higher than the supply voltage of the SoC. ATX power supplies on the other hand are usually a bit inaccurate, so the nominal 5 V rail could in fact provide more than 5 V. In order to measure such voltages, the step-up DC-DC converter that powers the MCU outputs about 5.45 V, just below the rated maximum voltage of 5.5 V.

The ADC signal inputs have a reasonably high input impedance. I could not determine it exactly, but the data sheet suggests that it is on the order of 100 kΩ. This means the signal lines carry only a small current, probably a few µA. With these values, there is little danger of signal to signal cross talk. At the same time it is low enough for a new signal to settle quickly after switching the multiplexer to a new channel.

10.1.3.3. Data Transfer

The ADCs run continuously and take 18 clock cycles for a single conversion, so they are driven by an 18 MHz clock. Both ADCs are started on the same clock cycle and thus are expected to run fully synchronously. On completion of a single conversion, an output signal triggers a preconfigured direct memory access (DMA) transfer into a buffer that the firmware will read, shown in Figure 10.4.

To ensure that logic synthesis and mapping did not introduce any delays, and to catch transient errors, additional logic checks that both ADCs always finish conversions on the same clock cycle; this is the `SYNC_ERR` signal in Figure 10.3b). In the event of a mismatch, an error bit in a firmware-readable status register is set.

To ensure that no samples are lost, DMA transfers use a double buffering scheme: After the required amount of samples have been written into the first transfer buffer, an interrupt is triggered (`Buffer_Full` in Figure 10.4). The firmware can then access it safely. In the meantime, DMA will fill the second transfer buffer. The firmware signals end of processing by submitting the finished buffer back to the DMA engine.

The DMA completion signal will also increment a DMA buffer counter; another counter tracks the number of individual ADC samples. At buffer boundaries, the sample counter equals the buffer counter times the buffer size in samples.

Should the DMA engine fail to transmit a sample (e. g. because the firmware did not return an empty transfer buffer in time), these two counters would no longer be consistent. That way, the firmware can ensure that no sample has been lost even when it runs at the limit of the MCU's computational power and the timing is difficult to predict[1].

10.1.4. Signal Processing and Transmission

The MCU's main CPU is an ARM Cortex-M4 core running at 72 MHz. The firmware running on the CPU has the job of reading the input buffers and transmitting them to the attached computer system.

[1] Most importantly, USB transmission introduces unpredictable latency.

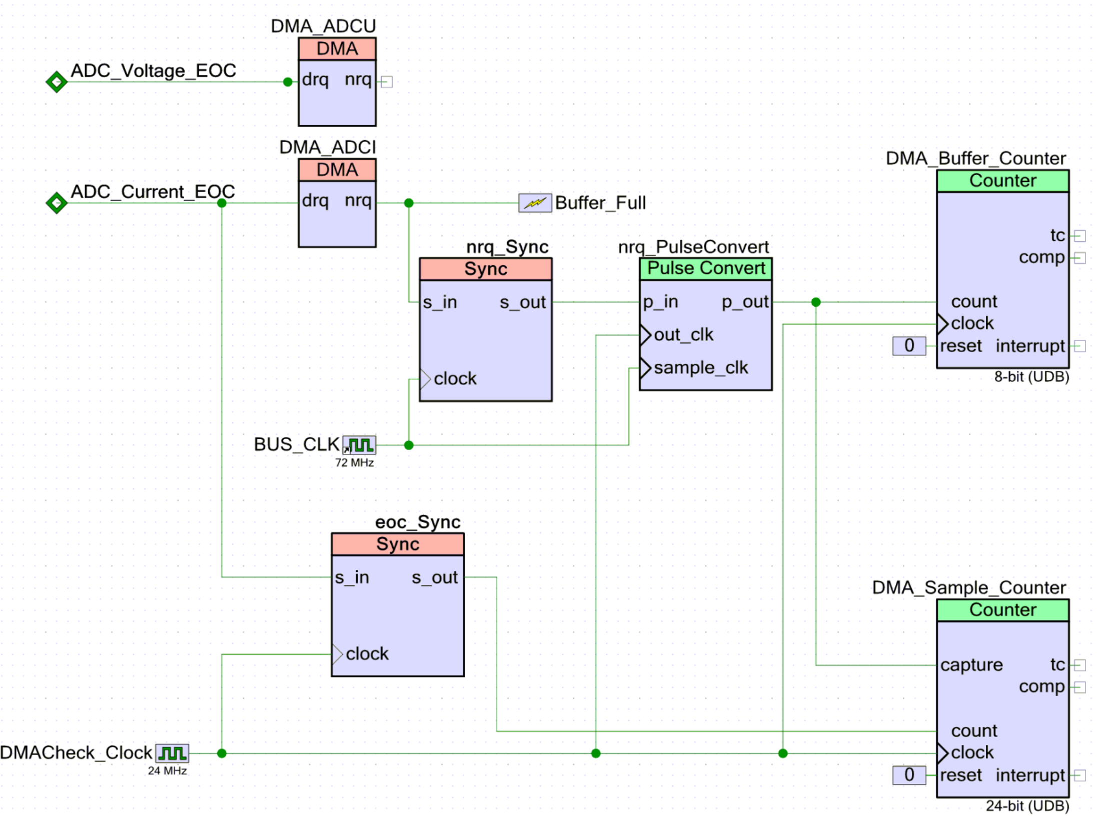


Figure 10.4.: CPLD schematics: DMA error checking.

10.1.4.1. Data Reduction

Unfortunately, the fastest way to get measurement data off of the MCU is a USB (full speed) connection as a CDC (communications device class, serial port) device. An obvious alternative would have been a serial peripheral interface (SPI) link, but only the SPI master module would have supported the required clock speed, while the single-board computers either don't support SPI slave operation at all, or are severely limited [4].

As the nominal gross data rate of USB full speed is 12 Mbit/s, but the maximum ADC data rate is 2×12 Mbit/s, the firmware must first reduce the amount of data to be transmitted. I have implemented two dynamically selectable strategies: Averaging over a selectable number of samples, and reducing the resolution to 8 bit.

10.1.4.2. Transmission

The firmware will read the input buffer, perform averaging and resolution reduction (in that order) as configured, write the output as 64 byte packets into a ring buffer, and submit the packets to the USB stack. A packet header of 4 byte identifies the amount of averaging, resolution, number of channels, and error flags from the hardware stage. Finally, there is an error flag if the ring buffer overflows.

Since USB transfers are always initiated by the host, the attainable net data rate varies with host interface circuit and software stack behaviour. Furthermore, more USB packets mean more processing time spent in the USB stack, so high data rates reduce the amount of available processing time. The error flags ensure that any data loss can be detected reliably. Evaluation has shown that a safe operating point for all tested host platforms is 10× averaging at 8 bit resolution.

Even though this reduces the signal's temporal resolution, averaging preserves the overall energy measurement, which is the point of this measurement platform. With a 100 kHz sampling rate (1 MHz distributed over 10 channels), this means that analogue bandwidth can be be up to 50 kHz before aliasing becomes a problem.

10.1.4.3. System Control

In the reverse direction, a host can send command packets that configure which channels to sample, sampling resolution, and number of samples to average. There are also commands to turn power distribution on or off, and to restart the MCU.

10.2. Embedded Cluster Platform

Figure 10.5 show the full evaluation platform. The computation platform is a down-scaled version of a high-performance computing system. It has roughly 1/10th of the computation and communication capabilities of (small) HPC clusters: CPU performance, RAM size, and network speed are each on the order of 10% of a small HPC system. Most

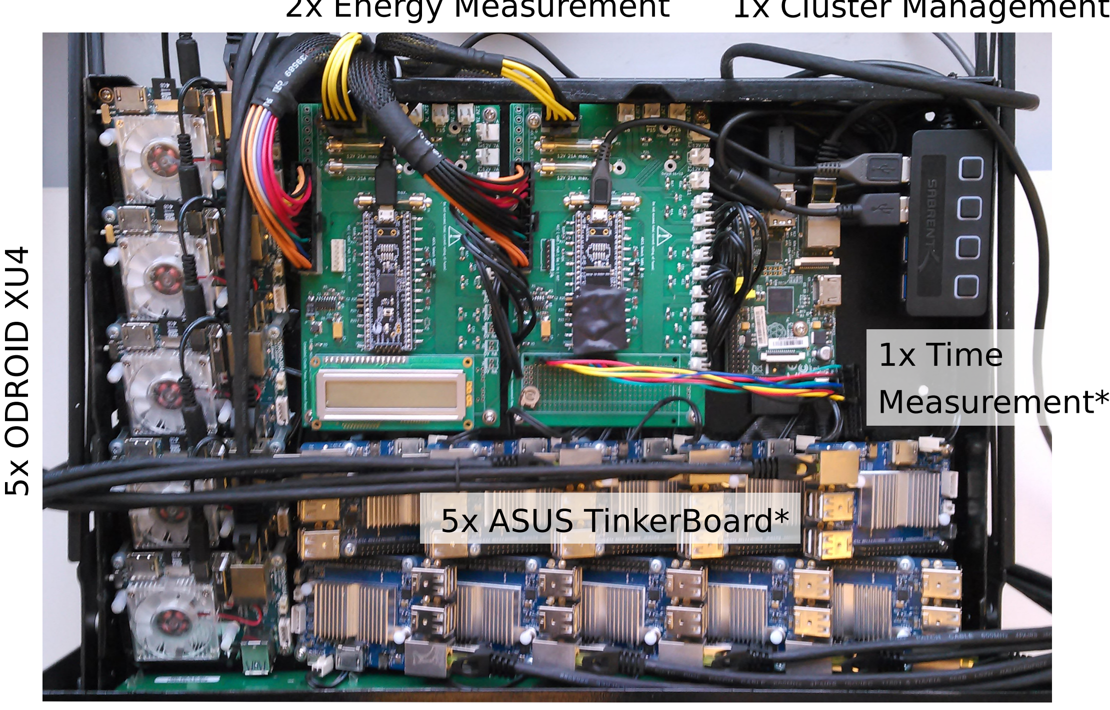


Figure 10.5.: Picture of the embedded cluster platform with integrated measurement infrastructure (initial version). Components marked with an asterisk were added/changed later due to significant evaluation challenges.

importantly this means that the ratio of computation speed over communication speed roughly matches that of HPC systems.

Besides the scaled overall performance, another design goal of the platform was to exhibit heterogeneity beyond state-of-the-art HPC systems. Therefore it consists of three different types of SBCs. It contains:

- five ASUS TinkerBoard with a quad-core ARM Cortex A17 CPU and 2 GB RAM
- five ODROID C1+ with a quad-core ARM Cortex A5 CPU and 1 GB RAM
- five ODROID XU4 with an octa-core ARM CPU (four Cortex A7 and four Cortex A15) and 2 GB RAM
- one Raspberry Pi as cluster management node ('head node')

At the time of construction, none of the available single-board computers reached the theoretically possible throughput on their gigabit Ethernet interface. I selected the ODROID boards due to their high reported real-world network speeds. I added the ASUS TinkerBoards later, selecting this particular model for its fully open drivers and no (known) hidden firmware, which was a problem with the C1+ boards (see Section 15.2 for details).

Since the ODROID boards do not have full mainline kernel support[2], the platform is limited to those kernel versions provided by the vendor. This means that ODROID boards use Linux kernel version 3.10.107, while ASUS boards use Linux kernel version 4.4.132 (they do not support 3.10 kernels).

Excluding the management node this cluster offers 80 CPU cores and 25 GB RAM. Each of the 15 compute nodes also offers GPU processing capacities, but they have not been used during this thesis.

The networking infrastructure consists mainly of a Cisco SG100-16 16 port 1 Gbit/s Ethernet network switch. As the switch has an internal bandwidth of 16 Gbit/s, it supports simultaneous full duplex transmission on all ports. With the exception of the Raspberry Pi, all SBCs offer 1 Gbit/s Ethernet ports.

All boards need a 5V power supply. The XU4 boards have a nominal maximum supply current of 4 A. The other boards have 2.5 A nominal maximum; in practice all boards stay significantly below these limits. Since the measurement boards support up to 20 A on the 5 V rail distributed over up to 10 channels, the platform contains two measurement boards, one for five XU4 boards, the other for all other boards. This also means that every channel gets a minimum sampling rate of 100 kHz.

Each measurement board is powered by an LC-Power LC6560GP3 560 W ATX power supply, which is specified for up to 27 A on the 5 V rail.

[2] At the time of this writing, there was partial mainline kernel support, but it was not in a sufficiently complete state for this thesis.

10.2.1. Cluster Management

The cluster nodes are configured to boot from an SD card. They will load a vendor-supplied version of the U-Boot boot loader; I changed the default configuration so that each board will load a system-specific Linux kernel and a common RAM filesystem from the head node via TFTP.

In the common part of the boot process, an initialisation script loads an archive containing the current experiment's executables and data files, extracts it, and passes control to it.

During experiment startup, a startup script configures some Linux kernel parameters to make the system more predictable. Most importantly, it will set a fixed clock frequency and try to prevent any intervention of the processor[3]. After that, it starts a minimal SSH server for debug purposes and then passes control to the task graph execution runtime. The end result is that no processes run on the system except for the SSH server and the task graph runtime.

10.3. Measurement Process

Measurements on the platform consist of three parts: energy measurement, local time measurement, and time correlation with energy traces.

10.3.1. Energy

The execution controller program described in Section 9.8 also performs energy measurement. It spawns one thread for each measurement board. The measurement thread will connect to the corresponding USB interface and configure 10 channels of 8 bit data with 10× averaging.

When the controller sends out the start signal, the measurement threads start recording data to disk. After the last node signalled completion, recording is stopped. The measurement thread continuously checks that no error flag is set and that the data stream parameters match the expected values.

The end result is one file per measurement channel containing raw samples with no header, totalling 30 files (15 boards, voltage and current).

10.3.2. Time

Challenges Time measurement turned out to be much more difficult than anticipated, which is why this measurement facility is more involved than it could have been. I discovered the main problem only after finishing the energy measurement boards, at which point they did not have enough resources (CPU, I/O pins, transmission bandwidth) left to perform time measurement as well.

[3]Not always successfully, see Section 15.2.

During evaluation, significant discrepancies occurred between node-local timers and timings inferred through energy traces. Time stamps could be much earlier or much later than the energy trace suggests; they could vary by several seconds in either direction over the course of a few minutes. Section 15.2 analyses these effects in more detail.

Energy traces of multiple channels are recorded synchronously, but timing differences were independent across channels. This means that it is highly unlikely that clock drift of the measurement boards was the source of this effect.

Instead, this strongly suggests that some timers vary their clock frequency. A second observation suggests that his is a platform-specific effect: Reliability correlates with board type – timers for the TinkerBoard were consistently plausible, XU4 timers were somewhat inconsistent, and C1+ boards had the largest variations.

Solution To eliminate the internal timers as a source of errors, the platform uses external time measurement. An external microcontroller board records time stamp signals sent via general-purpose input/output (GPIO) pins. A single Arduino board is connected to one GPIO pin on each of the 15 cluster nodes, using a voltage level shifter where required. Data is transmitted to the cluster management node via a USB serial connection.

The Arduino cyclically samples each pin and transmits a time stamp whenever pin levels change. An internal buffer stores time stamps for transmission; it is big enough that it does not overflow during the pin patterns that occur during evaluation.

Overall accuracy of time predictions depends on the accuracy of the Arduino's system clock. For use cases with high precision requirements, the Arduino can be clocked from an external high precision clock.

CPU Time There is a significant drawback of external time measurement, and that is the fact that the measurement approach shown above can only reliably measure a single CPU core. This covers the use cases that are time critical: Characterisation uses a single core and is very sensitive to clock variation, and overall execution time is also easy to measure this way.

It might be desirable to visualise activity across all CPU cores. For this only CPU timers work, and thus their measurements (start and end time of kernels, start and end time of network communication) are still recorded.

This usage is not directly part of the proposed methodology and can be thought of as a debugging tool, so the resulting potential inaccuracy will not matter as much. Users should just bear in mind that these timings might not be as reliable, depending on board type and load situation.

10.3.3. Time Correlation

To measure overall execution time more exactly, the energy measurement traces can be related to the pin change timings. As explained earlier, the execution runtime provides the `mark` API call that worker threads call during startup. It leaves a distinct pattern in

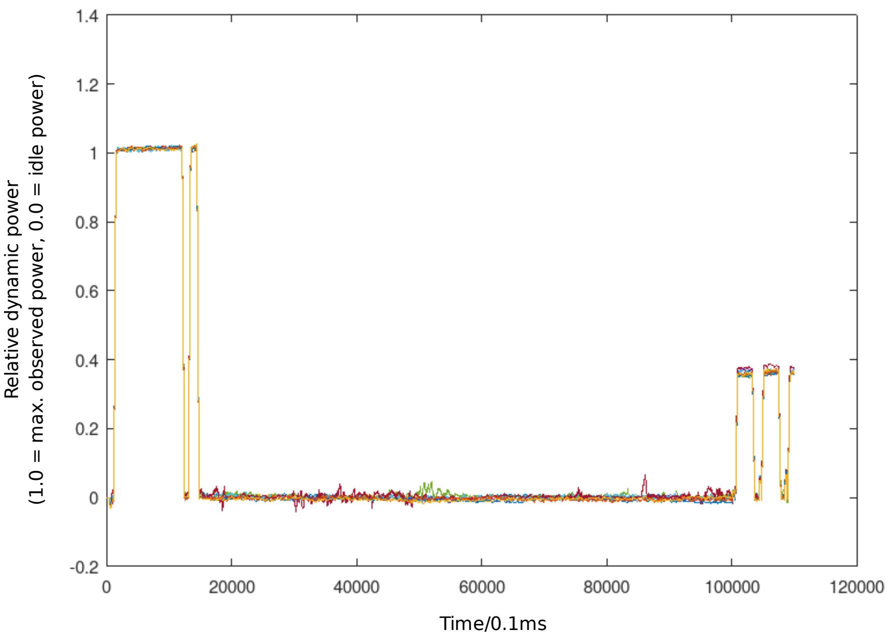


Figure 10.6.: Power trace of the initial `mark` sequence on five channels (denoted by colour). At 100 ms (1,000 power samples), the marker energy begins. After 10 s (100,000 power samples), the first task executes.

the energy trace so that the start time can be determined with about 1 ms accuracy by searching for the start of this pattern (shown in Figure 10.6).

At the same time, the runtime will toggle GPIO pins for the timing measurement board. This helps during kernel characterisation, when only a single core is creating energy and pin events.

The runtime will also create a distinct energy pattern at the end of execution, again paired with GPIO pin events. This makes it possible to determine the overall running time of any task graph. This also allows to calculate a linear correction factor to match GPIO timings to energy samples.

It also eliminates timing inaccuracies due to clock speed variations: The Arduino samples all 15 GPIO signals in a round robin fashion, so the timing between channels is reliably synchronised. The same is true for each energy trace. Thus, clock speed differences can be corrected for by correlating GPIO time stamps and energy trace markers.

10.3.4. Kernel Characterisation

To build the computation resource model according to Section 9.5, it is trivial to build a task graph that creates some input data, feeds it to the kernel to be measured, and discards the output. Characterisation then consists of running this specially crafted task graph, measuring energy and time, and extracting those parts of the measurement

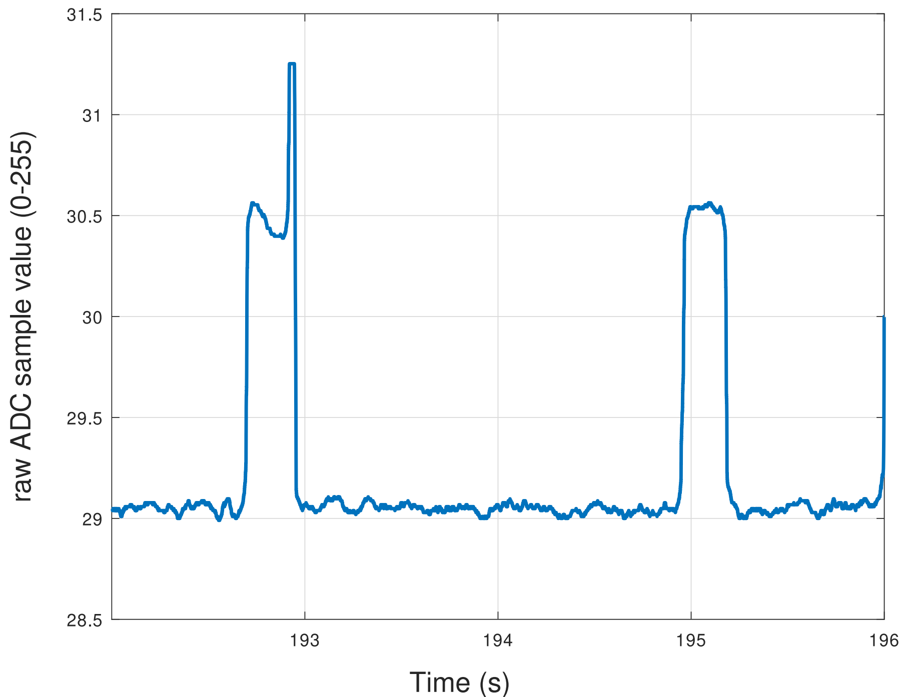


Figure 10.7.: Supply current trace of the MARK kernel. The left spike is the marker energy, followed by 2 s pause. The right spike is from the following kernel.

results that belong to the kernel being characterised.

10.3.4.1. Energy Markers

Unfortunately, even producing dummy input does take some time, so it is not entirely trivial to determine the start and end time of the task to measure. As has been discussed before, the local time source must be considered unreliable. Instead, the approach used to measure overall node execution time also works for kernel characterisation: create detectable energy patterns to mark start and end of a task, and use external time measurement to locate these patterns with high reliability.

A special kernel MARK creates a pattern that can be detected in an energy trace. The pattern consists of a short pause of 300 ms, then an assembler instruction sequence that consumes much power, then another pause of 2 s (see Figure 10.7). By placing the MARK kernel before and after the kernel being characterised, its start and end times can be derived from the energy trace. Assuming the edge rise time of the mark sequence is sufficiently short, timings should be accurate to about a millisecond.

Besides time measurement, the MARK kernel has a second use: The 2 s pause means that it's easy to determine the current idle energy. This is useful because the idle energy can change slightly over time, for example due to CPU temperature.

Using the MARK kernel, multiple characterisations can be run in sequence. This reduces

the amount of time spent in the deployment phase. If a fully instrumented platform is available for characterisation, the task graph can actually contain many characterisations in parallel across all cluster nodes. This can speed up the characterisation process significantly. Still, this also works if only a single CPU is available for measurements.

10.3.4.2. External Time Measurement

Using energy markers to determine kernel duration isn't sufficiently reliable to be automated. Using various signal processing algorithms to detect relevant `MARK` events, there are always spurious false positives or false negatives. Thus the Arduino time base is used to locate kernels in the trace (see above). The `MARK` signature surrounding the kernel to be measured is long enough that time measurement errors can be compensated for.

I chose the start pause to be much longer than any sensible[4] approximation error between time measurement and energy measurement clock. Also, a typical HPC kernel is unlikely to have such a long pause, as it will perform heavy calculation, and its atomic nature means it should not wait on synchronisation primitives or the like. Unless characterising an atypical kernel, there should be no 2 s pauses except for those created by the `MARK` kernel. This makes a visual inspection of energy traces easier and can be used as an additional consistency check during automated measurements. The trailing pause is intentionally small so that it cannot be mistaken for a start pause. It is just long enough that it can compensate for a reasonable approximation error.

When measuring dynamic energy, i. e. the energy consumption beyond idle power, it does not matter if the energy trace contains segments of pure idle power (i. e. pauses). Thus, kernel characterisation can sample a slightly larger window of the energy trace than the measured kernel duration suggests. That way, characterisation can determine the full energy consumed by the kernel even in the presence of small timing errors.

10.3.4.3. Secondary Time Model

To determine the slowdown factors for concurrent loads, another series of measurements can be made on task graphs constructed to exhibit the required amount of concurrent load. The exact details vary depending on what kernels are involved or if kernels are grouped into classes.

For this use case, GPIO pins are only toggled by the first compute core, so that external measurement can reliably determine behaviour on that core regardless of concurrent load on the other cores.

10.4. Platform Characterisation

The measurements that are needed to characterise the platform according to Section 9.4.2 can be expressed as (pretty trivial) task graphs as well. That way, they run in the exact

[4] i. e. if it's bigger, there are worse problems than characterisation.

same environment as the final application, especially regarding external influences – there is only the execution runtime itself, which will be present during application runs as well.

For the evaluation platform, three main parameters are relevant: (fixed) clock speed, communication timing, and communication energy.

10.4.1. Sustainable Clock Frequency

As explained in Section 4.1.5, if a CPU is getting too hot, it may reduce its clock frequency or take other measures to protect itself from damage. This can impact everything on the local system and will most certainly make the system very difficult to predict. Apart from improving cooling, this can only be prevented by setting the clock frequency so that no thermal throttling can happen.

Some thermal management mechanisms might be acceptable, if the throttling happens predictably and can be modelled with the primary/secondary time models proposed in this thesis. In this case, 'thermal budget' becomes another shared resource that is not explicitly modelled, much like cache size, RAM bandwidth, and others.

However, there are mechanisms that are completely unsuitable for this approach, as has been encountered during evaluation. So during platform characterisation, users should test if they have accidentally configured their platform for such a case.

The test is a simple task graph that executes computationally demanding tasks in parallel over a significant time span. In order to detect local timing variations, it should contain short pauses every once in a while, possibly using the marker used in kernel characterisation (see above).

Users can then check that these pauses in the energy trace match the time stamps recorded by the execution runtime and that they are distributed uniformly. If times deviate significantly, some undesirable effect has happened. Users can then try to lower the CPU clock frequency and run the test again.

10.4.2. Platform Power

During this thesis, I mostly ignore static (idle) power consumption of platform components, since it should be a constant offset. Thermal effects might create variations, but modelling temperature is out of scope for this thesis. During kernel characterisation, the 2 second pause before each kernel makes it possible to determine the current idle power including any thermal effects, so that small variations can be compensated for.

However, if users want to produce absolute predictions, they might need a more detailed model for static power behaviour of each node. Variance in analogue or digital components might lead to different idle power for different systems of the same architecture. If the efficiency factors of voltage converters and similar components vary, then computational activity might yield different offsets from idle power as well.

For the embedded cluster, it is possible to normalise power traces across compute boards. In all measurements, the start and end of each power trace contains idle time

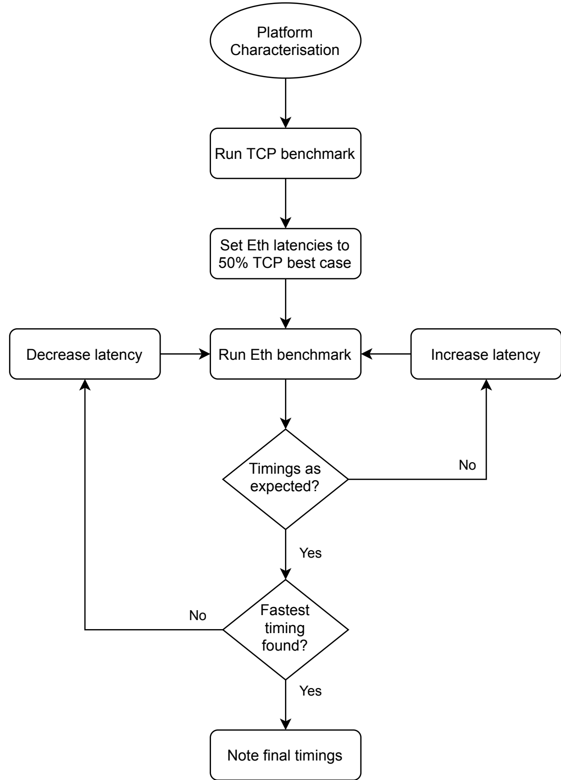


Figure 10.8.: Suggested communication timing characterisation procedure.

and well defined load sequences. Users can choose one system as reference and perform a linear regression over these matching data points to create a table of correction factors.

If users want to check for heat effects that could impair accuracy of this approach, the benchmark from Section 10.4.1 is suitable to check how big load-dependent heat variation is. It has clearly identifiable periods of different activity. For a single architecture these will even be at similar points in time, so that it should show board-specific and heat-related variation, if present.

10.4.3. Communication Timing

In order to determine optimal timing values for the Eth networking layer (see Section 9.8.4), I suggest a trial-and-error approach as shown in Figure 10.8. It uses repeated execution of a transmission benchmark to successively test packet latencies until the best value is found.

10.4.3.1. Cutoff Metric

To find out if a given latency configuration is acceptable, a good criterion is timing variance: The closer packet latencies approach their lowest possible value, the less time

will be available for hidden retransmissions, and benchmark results will show increasing timing variation. A noticeable variation increase indicates that a given packet latency has crossed its lower threshold.

I thus propose a simple metric to identify good timings: the difference between measured timings and ideal transmission time, calculated as $t_{\text{extra}} = t_{\text{measured}} - t_{\text{packet}} \cdot n_{\text{packets}}$, where $t_{\text{packet}}$is the configured packet delay for the given path, and n_{packets} is the number of packets needed for a given transmission size.

This difference consists of total path latency, measurement error, and excess retransmissions due to too small packet delays. Assuming measurement error and path latency stay roughly the same, excess retransmissions easily dominate this number. End-to-end path delay is below 2 ms for typical Ethernet equipment. As evaluation showed, relative error of time measurement should be 10^{-4} or less, i. e. 1 ms per 10 seconds of transmission time. This means that if t_{extra} is lager than a few milliseconds, the error must be from retransmissions.

Since spurious retransmissions errors occur even for good packet timings, a second metric helps to fine tune the result. For a given path, almost perfect timings will have a median t_{extra} in the low milliseconds. The mean can be larger, however: occasional outliers will produce a mean that is noticeably larger but still within a few percent difference, but regular outliers will have a significantly bigger mean.

The exact difference between good mean and median values differs by board type; I have observed differences up to 400% for one board type, no matter how slow packet timings were configured. Instead of a fixed boundary, evaluation has shown that this difference can increase by several orders of magnitude for a small (e. g. 10%) decrease of packet delay; this is a good indicator that the higher packet delay is close to being optimal.

10.4.3.2. Model Parameters

As stated before, one pair of bridges per node implement the time model; it consists of a startup delay and a packet delay. Once good real-world timings have been identified, the configured packet delays can directly be used as packet delays in the platform model.

t_{extra} can be used to determine sensible values for startup delay: Create a system of linear equations that has two variables for each available hardware architecture, one representing incoming packet latency, the other one representing outgoing packet latency. For all data points from the benchmark suite, identify sender and receiver architecture and add a suitable equation ($t_{\text{extra}} = t_{\text{out}} + t_{\text{in}}$). Then perform a linear regression to find the best set of latencies for the system of equations.

10.4.3.3. Benchmark Suite

The transmission benchmark suite uses a simple dummy data source kernel and a dummy data sink kernel. A number of source and sink tasks are linked to each other and mapped to different cluster nodes. If that mapping covers all relevant transmission paths, the only missing piece is reliable transmission timing.

For the platform used in this thesis, I have devised a suite of benchmarks that tests all important aspects. It has several test classes:

`iobench-1` is a simple non-overlapping transmission between two nodes.

`iobench-1bidir` is a bidirectional transfer between two nodes that checks if parallel transmission and reception impacts possible packet timing.

`iobench-1c3` is like `iobench-1`, except all PEs on one node try to send a packet to another node. This checks if local packet queuing has any unexpected impact.

`iobench-max` is like `iobench-1bidir`, except all nodes transmit data simultaneously. This tests if the switch is indeed capable of handling a fully loaded network.

`iobench-crosscore` is like `iobench-1`, but between different node architectures. This identifies asymmetric transmission paths (see Listing 9.6 for example).

`iobench-crosscore-bidir` is the same, but bidirectional. This checks if asymmetric timings deteriorate during full duplex operation.

This set of tests is then executed for various data sizes between 1 and 150,000,000 bytes[5], and repeated ten times. This helps identifying additional startup delays.

While this benchmark suite is tailored to the proposed evaluation platform, the general principle behind it should be a good starting point for tests on other platforms as well.

10.4.4. Communication Power

Communication activities can consume significant amounts of energy. As described in Section 9.8.4, the Eth network layer tries to reduce its dynamic energy profile using an idle priority thread that wastes energy while the communication stack isn't active. But even if this was perfectly working or the anticipated hardware support of real-time Ethernet was present, network interface hardware would still consume non-negligible amounts of energy.

The simple two state power model (idle vs. active power) of the platform resource model (Section 9.4.2) should work. Therefore, results from above benchmarks can be reused: `iobench-1` and `iobench-1bidir` contain long phases of transmission without any computation activity. Idle and active states are easy to locate in the energy trace, from this the communication power offset can easily be calculated. As a sanity check, traces will also show if the assumption of a simple two state power model is correct.

`iobench-1` has transmission on one channel and reception on another, so that these measurements can be used to derive model parameters. `iobench-1bidir` can show if full-duplex transmission needs a slightly expanded model, or if the two directions are independent of each other.

[5] The maximum that fits into memory on all nodes.

Part IV.

Evaluation

11. Evaluation Goals

In order to structure the evaluation process, I divide the overall evaluation effort into four separate steps, three for the main methodology and one for additional aspects.

The main goal is to establish the viability of the presented methodology (Contributions 1 through 5). To do so, the accuracy of the measurement infrastructure needs to be established first (Contribution 7), then the evaluation platform must be evaluated and characterised (Contribution 6). As far as possible, this part will resemble the steps that users of the proposed methodology would also perform.

After evaluating the main methodology, I then present justifications for some design decisions that do not directly fit into the design flow and that would otherwise appear to be arbitrary.

This section gives an overview of these four goals and derives evaluation criteria for each of them. The actual evaluation process and discussion of results will then be covered in the following sections, one section per goal.

11.1. Overall Methodology

11.1.1. End User Requirements

As stated in Section 2, the anticipated use of the presented methodology is to give HPC programmers a way to optimise their applications by comparing different variants without running them on the designated target platform. As a result, the main goal is to minimise approximation error of simulation results: For a given simulation result, the true result lies within a range of possible values. If the error ranges of two different results do not overlap, they accurately determine the better variant.

Furthermore, having a correct estimate of expected maketime and energy usage helps in planning execution, so even for singular predictions a high accuracy is desirable.

11.1.2. Evaluation Criteria

11.1.2.1. Time

As shown in Section 8, there is no directly comparable methodology in terms of flexibility and speed. Still, it is desirable to have an accuracy that is on par with established methods. Table 11.1 shows accuracy values reported by some of them. None of these are actually comparable, as they measure different kinds of predictions at widely varying orders of magnitude (see Table 8.1 for details). Nevertheless, these values suggest that HPC predictions with an error of about 5-15 % are useful results.

Table 11.1.: Reported time prediction error in related methodologies. Multiple values are given if more than one simulation technique is evaluated.

Name	Reported Error(s)
PSINS [58]	15 %
TaskSim [59]	8 % - 17 %
SimMatrix [61]	2.5 %
Petri Nets [62]	4.5 %

As a result, I consider a relative error of 5 % to be a good result for overall maketime predictions, while an error of more than 15 % can be deemed of limited practical use.

11.1.2.2. Energy

For energy, there are few related HPC-scale energy prediction methodologies to compare with. The main energy-related concern is power provisioning (see Section 4.1.4), and the relative differences are significant: on some machines, more than 40% of the provisioned power stay unused over long time spans [48]. The proposed methodology could be applied to a hardware overprovisioned platform in order to select the fastest subset for a program that fits within the power limit.

This means that even with 20 % prediction error, a hardware-overprovisioned platform would still have a performance advantage over a traditional power-overprovisioned platform, if both had the same power limit.

When comparing to an existing predictions approach, the RMAP resource manager [48] has a reported prediction error of up to 15 % for the worst case, and less than 10 % for 96 % of the predictions. Unlike this thesis, these figures are not based on physical measurement and only include dynamic CPU and RAM power, so they are only suitable as a rough guideline.

As a result, I consider an error of 5 % or better an ideal result, and anything below 10 % a good result. Anything below 20 % would still be a result with some usefulness.

11.1.2.3. Performance

Since predictions should be used in interactive workflows and in automated exploration of larger design spaces, individual simulation runs should ideally complete within milliseconds to seconds on real-world developer equipment. Simulation times of more than a minute would limit usefulness for the intended purpose.

The definition of 'real-world developer equipment' is vague, of course – a reasonably recent mid-range to high-end consumer CPU would be a valid choice. The exact times are not too important, the order of magnitude is. Also, I expect the methodology to scale

linearly with the number of tasks so that hardware upgrades can effectively increase usable application sizes.

11.2. Measurement Accuracy

Since evaluation depends on the measurement infrastructure presented in this thesis, its accuracy needs to be established first. Its margin of error increases the perceived error of the proposed methodology, so it should be as low as possible within the design goal of universal (i. e. low-cost) availability.

11.2.1. Time

Relative Error The inputs for the prediction methodology use the same reference clock as the measurements of the evaluation benchmarks. The way input timings are used, every linear relationship between system clock and real-world time results in the same relative error. This means that measurements involving relative error are correct by definition.

If accurate absolute readings are required and linearity is good, then wall clock time can easily be calibrated each time the system is used, simply by comparing a long enough time span to a highly accurate external time source, for example the GPS time signal.

As shown in Section 10.3.2, the Arduino clock is the reference time source. Assuming stable ambient conditions, the its clock crystal should not experience a significant non-linear drift. This results in a linear relationship between reference clock and real-world time. In practice, unstabilised quartz crystal oscillators have a frequency error of much less than 1 %, down to 0.01 % in the best case [43], so this should be suitable for the intended purpose without needing to determine the exact error range.

In the long term, ambient conditions (temperature, humidity, etc.) of a simple office or laboratory setting change, and these do influence crystal oscillators. The experiments for this thesis were short enough that these conditions did not vary significantly during a single benchmark run.

I therefore do not evaluate the accuracy of converting Arduino clock cycles into wall clock time. As a sanity check for the assumed linearity, I compare all clocks in the platform against each other and expect less than 1 % deviation from ideal linear behaviour.

Absolute Error As discussed in Section 10.3.4, kernel characterisation needs energy trace timing to be correlated with reference time in order to measure the energy of kernel execution. As has been stated, a task running the kernel to be characterised is preceded by a 2 s pause and followed by a 300 ms pause.

Since idle power will be subtracted from the resulting energy, the time window used for energy measurement can safely include pause times. Assuming an absolute error of up to 150 ms, the sampling window can thus be increased by 150 ms without risking inclusion of unrelated parts of the energy trace.

Thus, the reference clock will be converted into the time base of the measurement boards using a linear approximation. The absolute approximation error should not exceed 150 ms.

This evaluation needs to be done under realistic power settings, because there might be an additional timing error due to voltage regulator capacitance on the compute boards (see Section 5.3.2). Since the evaluation platform includes several low power CPU cores, this effect may be large enough to be significant.

11.2.2. Energy

The arguments about time linearity apply to energy measurements in a similar way: as far as relative error is concerned, only non-linearity of the measuring equipment matters. Evaluation therefore needs to assure that measurement results deviate from linear behaviour as little as possible.

The main components that affect accuracy and linearity are the Maxim MAX4378T instrumentation amplifier and the ADC itself. According to the data sheets, the former has a typical error of 0.5 % above a minimum sense voltage of about 25 mV, the latter has a maximum error of 0.1 % (+/- 2 mV).

For absolute readings, the voltage references also affect the result. The internal voltage reference is specified at 0.5 % error. For supply voltage measurements, there is an additional Maxim MAX6126 voltage reference with a specified error of 0.02 %.

Given these technical specifications and the fact that timing error is expected to be less than 1 %, I would consider this a good result for energy as well.

11.3. Evaluation Platform

I consider the evaluation platform to be a minor contribution, born out of the need for a compact evaluation system with a suitable integrated measurement infrastructure. Therefore, its most distinguishing feature is the measurement infrastructure, which has been covered in the previous subsection.

Nevertheless, there is a clear distinction between measurement itself and running the workload; the evaluation platform should at least be examined for any properties that would affect the primary evaluation goal of this thesis. This applies to more than the evaluation platform itself: Users of the methodology also benefit from a systematic check that their target hardware has no surprising and/or undesirable properties.

One design goal of the evaluation platform itself is that the ratio of computation over communication speed is similar to HPC systems, making it a downscaled model of HPC systems. While not central to the proposed methodology, it would also be advantageous if the platform had a similar ratio between idle and active power as a HPC cluster. This would improve comparability of energy prediction quality with HPC research, e. g. [48]. In [52], an idle system is reported to have about 50% of active power; this is for the same HPC architecture (IBM BlueGene/Q) as used in [48].

11.3.1. Platform Characterisation

Characterisation of the target platform (Section 10.4) is a standard step for the proposed methodology. It covers the most important aspects: timing and power of communication and application-independent computation (clock speed and idle power).

I designed simulation models to match common hardware behaviour. I expect that platform characterisation shows no anomalies that reduce or preclude applicability.

11.3.2. Power Variation and Heat

The platform consists of multiple independent compute boards. Manufacturing variation can have a significant effect on the power properties of each board. Furthermore, heat effects might affect boards, but the simulation models do not model heat.

Ideally, all boards of the same type should show the same general power behaviour. If the difference is too large, absolute predictions become meaningless without a full platform characterisation.

Variations should be small enough that characterisation of a single specimen is sufficient to predict the behaviour of a full platform. If this is not viable, at least comparisons of dynamic power should be reliable.

An additional test should check for heat effects. Since HPC loads will lead to heavily loaded systems, behaviour of cold systems is not relevant, but there should be no significant variance over longer periods of mixed usage. I expect that under a uniform compute load, power should not vary significantly over time.

11.4. Individual Design Decisions

There are some design decisions that are not evaluated individually in the above sections. The previous steps will already have established the overall methodology's accuracy and performance, so strictly speaking, the design decisions as a whole have been assessed already.

Since these tests (or variations of them) were done (or at least attempted) during the development of the presented methodology, they still offer valuable insight. So I present them at the end of the evaluation part in order to show how some decisions affect the methodology:

1. Fixed clock frequencies instead of dynamic frequency scaling (Section 10.2.1) might affect computation performance.

2. Using an external time measurement circuit (Section 10.3.2) introduces additional complexity in the measurement setup.

3. The more predictable Eth network protocol instead of TCP (Section 9.8.4) affects communication performance.

4. The physical execution runtime (Section 9.8) might introduce additional overhead that traditional runtime systems avoid.

12. Evaluation of Measurement Accuracy

Since all other evaluation procedures depend on the measurement platform presented in Section 10, it had to be evaluated first.

As stated in Section 11.2, the main goal is to ensure that energy measurements exhibit linear behaviour. As has been argued there, time measurements should not need a detailed evaluation. To be on the safe side, the expected time linearity should at least be confirmed in a simple test.

12.1. Setup

To determine basic energy measurement accuracy, DC signal linearity needs to be established first. Then the channels need to be checked if they are independent of each other. Finally, the frequency response should be appropriate for the sampling rate the boards will be operated at.

For time measurement, a simple test compares all clock sources in the system to each other to see if there are any inconsistencies that would invalidate the linearity assumption stated in Section 11.2.

For all tests in this thesis, the measurement boards have been operating for at least one hour prior to starting measurements, so that they should have reached thermal equilibrium.

12.1.1. DC Accuracy

To establish DC accuracy, I use a Rohde & Schwarz HMC8042 programmable power supply as power source, an ITECH IT8500 programmable electronic load as consumer, and a Rohde & Schwarz HMC8012 digital multimeter as reference measurement device. All three devices can be controlled from a computer using a serial connection. I used a shunt resistance of 40 mΩ (at 1 % accuracy and a temperature coefficient of 35 ppm/K) for all tests.

Current Measurement In the first test run, I configured the power source to supply exactly 5 V and no current limit (beyond the physical limit of 5 A of the power supply itself) and connected it to ground and the 5 V fuse clamp. I connected the electronic load to the output pins of one channel. Finally, I set up the multimeter to measure the voltage across the shunt resistor.

During a one hour run-in period, the electronic load was set to draw 1 A. After that, measurement began.

A script I wrote set the load to 0 A and then increased it by 1 mA for each measurement. For each step, it waits 3 s for the new load value to settle, averages five consecutive measurements from the multimeter, and averages 80 µs of measurements (one data packet) from the analog-to-digital converter (ADC) on the measurement board. It then writes all values to a tabular text file. The result is a look-up table that associates electrical current flowing through the shunt with shunt voltages and ADC sample values for the channels that have been measured.

With a look-up table, piecewise linear interpolation can be used to get highly accurate absolute readings. On the downside, this approach depends on the accuracy of multiple devices (multimeter and electronic load). Also, it requires long measurements to calibrate each board to be used.

To determine that shunt voltages and ADC sample values have a linear relationship, the accuracy of the electronic load does not matter at all, and only the linearity of the multimeter matters. As discussed in Section 11.2, linearity is the most important goal.

If absolute values are desired, good linearity means a single calibration measurement[1] on each channel can provide the conversion factor from ADC samples to physical units.

Thus I compared the lookup table to a linear formula derived by linear regression of the results. The maximum difference determines the worst case non-linearity.

In all calculations I ignored shunt resistor measurements below 25mV and did not use them for linear regression, because the MAX4378T instrumentation amplifier has a much bigger error below that value.

The evaluation so far only tests the accuracy of shunt resistor voltage measurement. The final gap that needs to be closed is the relation between actual current and shunt resistor voltage. The shunt resistor value is meant to be chosen by users according to their needs, yet this evaluation can only evaluate the resistors used during this thesis. Furthermore, assuming a linear relationship between current and voltage (which resistors should certainly exhibit), this is only relevant for users that require absolute predictions, not relative comparisons.

As a result, I checked the absolute values of the measurements described above for any inconsistencies with these assumptions.

Voltage Measurement Voltage measurements use an ADC identical to the one used by current measurement, and the analogue signal path is almost the same but with one less active component (the instrumentation amplifier). Voltage measurement linearity should therefore be similar to or better than current measurement linearity. Therefore, I performed a shorter examination to confirm my assumption.

Using the same setup as before, but connecting the multimeter to the output pins instead, I again let the system run in with a load of 1 A. Then I adjusted the input voltage so that after the shunt resistor, a voltage of about 4.5 V registered on the multimeter and noted voltage and ADC sample value. I repeated this measurement across the range of 4.5 V through 5.4 V in steps of roughly 100 mV.

[1] or two, to eliminate a possible offset error.

12.1.2. Channel Independence

If the design succeeded in preventing cross talk between channels or between main power and measurement signal, this can be tested easily: using an execution run that exhibits sudden and significant current changes on one channel, there should be no trace of current changes on another channel.

Evaluation already includes various characterisation procedures with suitable expected energy patterns across all channels, so I selected a random current measurement that showed the desired current changes to confirm that there is no detectable cross talk.

Note that voltage signals will not be independent of each other: If a channel causes a voltage drop in the power source (e. g. due to a sudden increase in current), this voltage drop will of course be visible on all channels. Therefore, voltage signals cannot be sensibly tested for cross talk. However, they are routed parallel to current signals, so it is safe to assume that they behave the same regarding cross talk.

12.1.3. Frequency Response

As argued in Section 10.1.2, it is important to establish the frequency response of the measurement signal path. To determine the influence of different parts of the analogue signal path, I performed three separate tests:

1. Signal path from 5 V supply line to MCU pin
2. Signal path from MCU pin to ADC
3. Entire signal path (1 and 2 combined)

I repeated the first two tests with five randomly selected channels; I did the third measurement on all 5 V channels on both boards.

For the first test, I used a Siglent SDG2122X function generator to generate a white noise signal on a 5 V output pin. I used a Tektronix MSO2012B digital storage oscilloscope to measure and calculate the frequency spectrum of both, the source and the signal as it arrives at the MCU.

For the second and third test, I used the function generator to generate a fixed-amplitude sine wave. I looked at a live display of the signal and increased the frequency to the point where the signal amplitude drops to 71 % of the DC value (since -3 dB is equivalent to a relative amplitude of 0.708). I then recorded the current frequency of the sine wave.

12.1.4. Time Measurement

As discussed before, I want to confirm that all clock sources behave as can be expected from quartz crystals. Assuming that these sources do not all have the same systematic error, this is possible by comparing the behaviour of all available time sources (CPU timers on 15 compute nodes, sample clock from two energy measurement boards, and GPIO timings from one time measurement board).

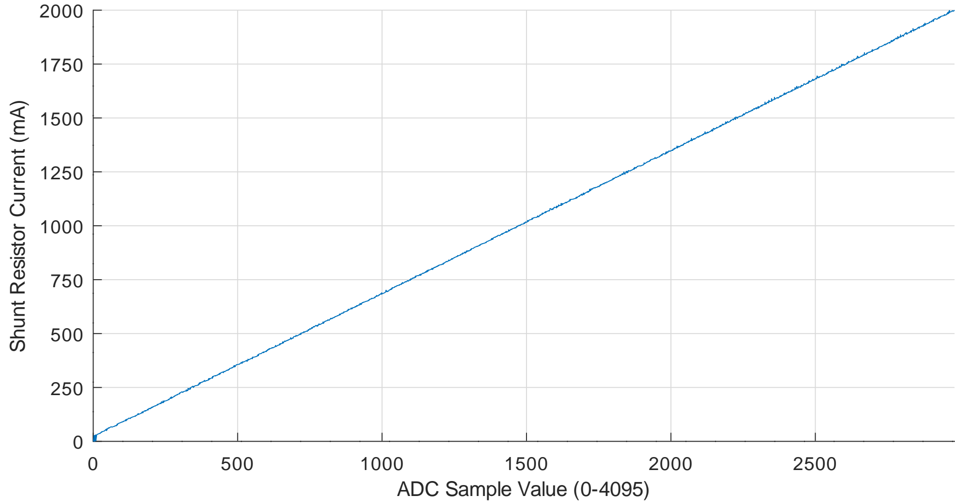


Figure 12.1.: DC current measurement: shunt resistor current over ADC sample values, channel 10.

For this evaluation step, I treated energy trace timings as accurate reference values. I created a benchmark where the cluster servers would do nothing but wait for multiple spans of 3 minutes, interrupted by a `MARK` task (see Section 10.3.4). In total, a single benchmark ran for one hour and had about 50 tasks that can be correlated between traces. I used the procedure outlined in Section 10.3.4 to detect kernel timings in the energy trace.

I then matched GPIO timings to their energy trace events and performed a linear regression, noting the approximation error of the result. I did the same with CPU timer values. For comparison, I also calculated a simple linear correction factor using just the trace end markers.

Since the compute nodes might have different energy behaviour due to construction details, I performed these tests for all CPU types; I tested the A7 and A15 cores on the ODROID XU4 boards separately, because as noted in Section 11.2.1, different power settings might lead to different influences by voltage regulator capacitance.

12.2. Results

12.2.1. DC Accuracy

Figure 12.1 shows a plot of shunt resistor current over the ADC readings obtained from a measurement board equipped with 40 mΩ (±0.1 %) shunt resistors. To the naked eye, sample values seem to be perfectly linear, even for low currents. However, shunt voltages were actually lower than expected, shown in column two of Table 12.1. At 1 A, the shunt voltage should be exactly the resistor value, with Ohms replaced by Volts.

Table 12.1.: Shunt resistor voltages for a nominal current of 1 A

Channel	Shunt Voltage at 1 A (40 mΩ)	Shunt Voltage at 1 A (100 mΩ)
1	36.957 mV	1.009 V
2	37.017 mV	0.992 V
3	37.308 mV	1.005 V
4	37.360 mV	0.996 V
5	37.430 mV	0.999 V
6	37.109 mV	1.007 V
7	37.753 mV	1.008 V
8	37.809 mV	1.001 V
9	37.407 mV	0.991 V
10	37.617 mV	0.990 V

In order to eliminate the resistors as source of the errors, I desoldered them and measured them in a standalone setting. Using a constant current source of 1 A, I measured the voltage drop across them. All resistors were well within their specified accuracy.

Due to the unclear situation and the fact that the compute boards turned out to have much less typical current than expected, I decided to replace the 40 mΩ resistors with 100 mΩ (±0.1 %) resistors. Column three of Table 12.1 shows the measurements of that configuration; they match the expected values within an error margin of 1 %.

Table 12.2 shows a closer analysis of shunt resistor voltage using three different fitting methods: a linear regression using all measurements $> 25\mu V$, a linear regression using just two measurements (one in the middle, one at the upper end of the data set), and a static formula derived from ideal values according to the data sheets of all involved components: $U = \frac{ADC}{4096} \cdot 2 \cdot U_{\text{ref}} \cdot \frac{1}{g}$, where U_{ref} is the reference voltage of 1.024 V and g is the instrumentation amplifier gain factor of 20; effectively, $U = \text{ADC} \cdot 25\mu\text{V}$.

Using the full regression, no sample value deviated more than 1 % from a linear approximation. The relative error increased slightly when using just two measurements for linear regression, but the error is still around 1 %. The static formula has up to 4 % error, channels with a high voltage offset are worst.

Figure 12.2 shows a detailed plot of the relative error of one channel (the one with the largest offset voltage). It shows that below 25 mV (an ADC sample value of roughly 1000), relative accuracy suffers significantly. In fact, an ADC sample value of 1500 seems to be a better lower boundary for high accuracy. This corresponds to slightly less than 40 mV shunt resistor voltage.

Table 12.2.: Maximum error of different linear fitting methods for ADC to shunt voltage conversion: linear regressions using 2000 and 2 data points per channel, and the theoretical value of 25 µV per ADC count.

Chan.	2000-Point Linear Fit				2-Point Linear Fit		25 µV
	Factor	Offset	Rel. Err.	Abs. Err.	Rel. Err.	Abs. Err.	Error
1	25.11 µV	48.30 µV	0.79 %	491 µV	1.13 %	554 µV	1.41 %
2	25.07 µV	75.41 µV	0.82 %	737 µV	0.98 %	630 µV	1.24 %
3	25.01 µV	282.92 µV	0.88 %	599 µV	1.20 %	449 µV	2.02 %
4	25.01 µV	200.27 µV	0.97 %	541 µV	0.98 %	551 µV	1.41 %
5	25.12 µV	421.39 µV	0.82 %	567 µV	1.18 %	875 µV	2.81 %
6	24.99 µV	462.77 µV	0.80 %	969 µV	1.21 %	846 µV	2.58 %
7	24.92 µV	490.21 µV	0.79 %	945 µV	0.77 %	921 µV	2.31 %
8	25.15 µV	548.11 µV	0.73 %	705 µV	1.18 %	488 µV	3.37 %
9	25.10 µV	291.51 µV	0.81 %	463 µV	1.01 %	584 µV	2.19 %
10	25.05 µV	722.12 µV	0.86 %	929 µV	0.92 %	942 µV	3.77 %

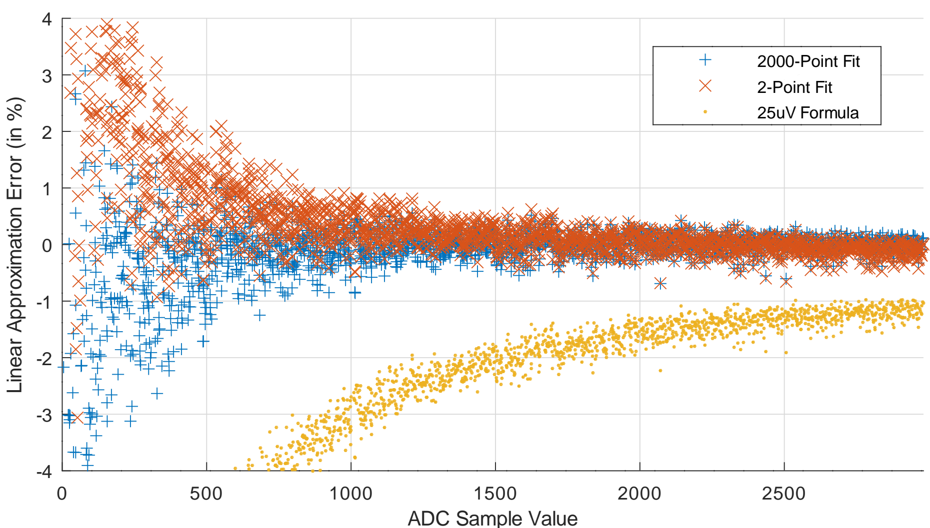


Figure 12.2.: DC current measurement: relative error over ADC sample values, channel 10.

Measured Voltage	ADC sample value	Ideal Voltage	Relative Error
4.496 V	1052	4.502 V	-0.13 %
4.611 V	1280	4.616 V	-0.11 %
4.715 V	1486	4.719 V	-0.08 %
4.789 V	1632	4.792 V	-0.06 %
4.893 V	1837	4.895 V	-0.03 %
5.000 V	2048	5.000 V	0
5.115 V	2275	5.114 V	0.03 %
5.208 V	2460	5.206 V	0.04 %
5.287 V	2613	5.283 V	0.09 %
5.386 V	2811	5.382 V	0.08 %

Table 12.3.: DC voltage measurement results for a single channel.

Voltage Table 12.3 shows the results of the reduced measurement series for voltage readings. Ideally, voltages would conform to $U_{\text{ideal}} = \frac{ADC}{4096} \cdot 2 \cdot U_{\text{ref}} + U_{\text{ext}} - U_{\text{ref}}$, where U_{ref} is the internal reference voltage of 1.024 V and U_{ext} is the external reference voltage of 5.00 V. Effectively, $U_{\text{ideal}} = ADC \cdot 0.5\text{mV} + 3.976\text{V}$.

12.2.2. Channel Independence

Figure 12.3 shows two channels recorded during their typical initialisation sequence in the worst possible configuration: Channel 5 is an ASUS TinkerBoard, which has the largest current swing of the three board types. Channel 6 is an ODROID C1+, which has a very low current consumption. These two channels use the same MAXIM MAX4378T instrumentation amplifier, and their PCB traces are physically close.

The graph shows that the big current step of channel 5 has no noticeable effect on channel 6. During evaluation, I looked at hundreds of such traces and did not see any bigger effect.

When looking closely at the pause around 0.6 s, it seems a tiny effect might be present within the noise floor. I filtered the signal using a 10 ms sliding mean; the effect was barely visible and I determined its size to be 0.3 sample counts, which is an error of about 0.5 %.

12.2.3. Frequency Response

Figure 12.4 shows the frequency response of the analogue path between 5 V power supply and measurement MCU pin of one channel. The external signal path (orange

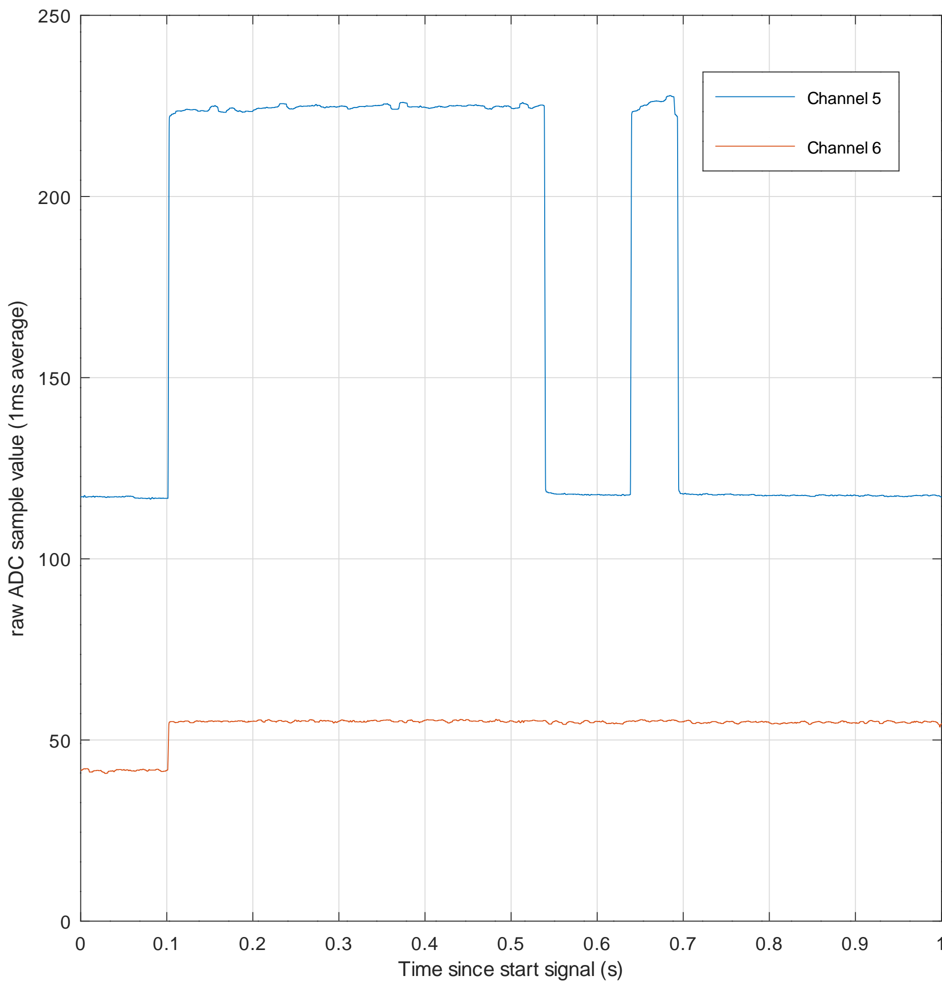


Figure 12.3.: Electrical current of two physically adjacent channels with significant power consumption steps to make cross talk visible, if present.

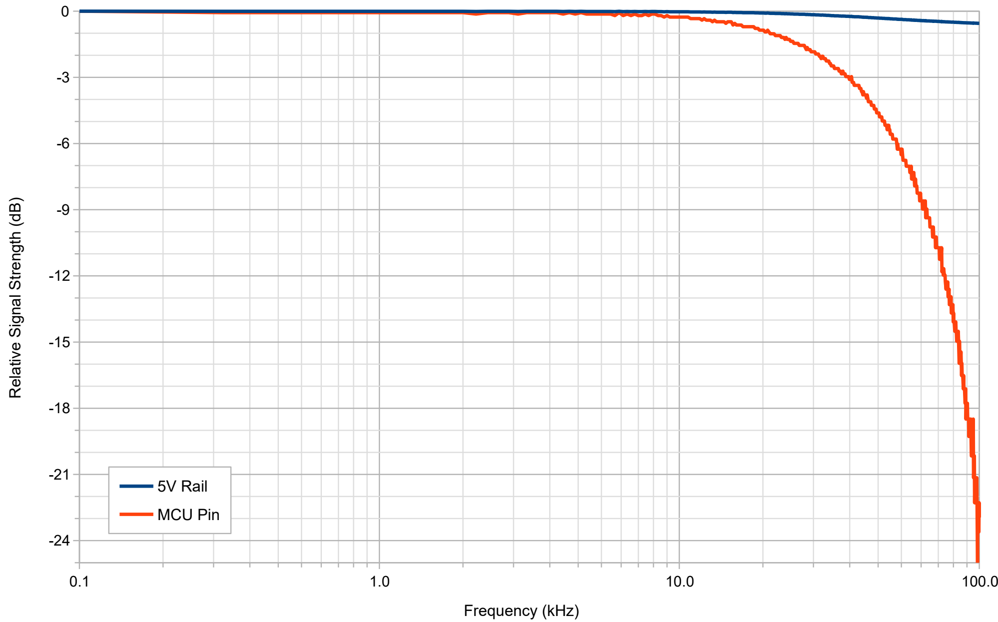


Figure 12.4.: Frequency response of the external analogue signal path.

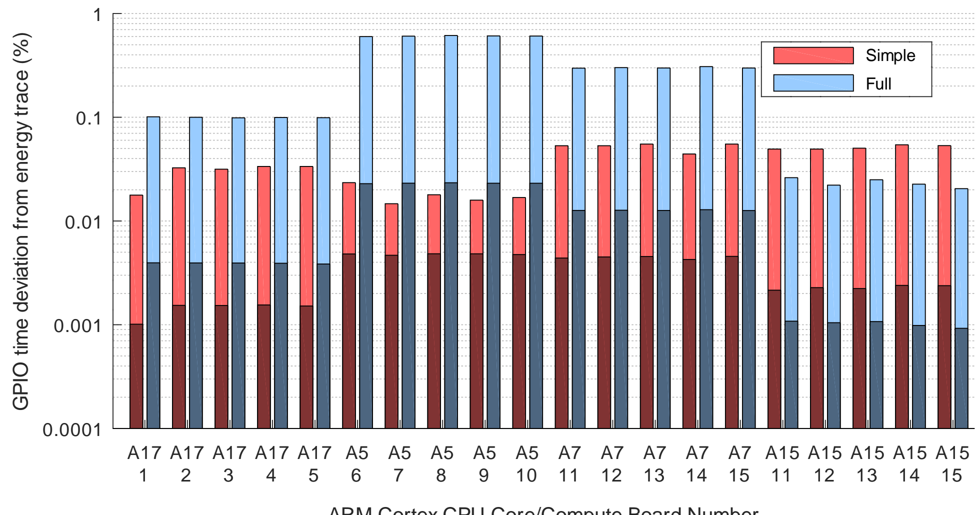
line) has a frequency limit of 38 kHz for 3 dB attenuation, while the 5 V rail itself (blue line) has an almost linear response within the frequency range of interest.

The signal path between MCU pin and ADC showed a frequency limit of about 20 kHz. For overall frequency response, I measured about 15 kHz as frequency limit. Thus the measurements are consistent with each other. The other channels showed similar behaviour.

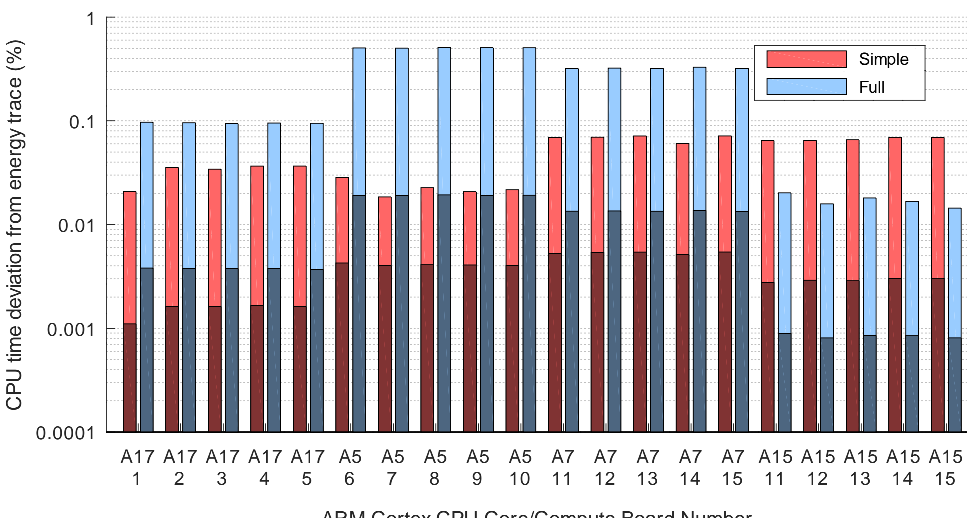
12.2.4. Time Measurement

Figure 12.5 shows the results of the clock linearity tests. CPU and GPIO timings have a worst-case error of less than 0.1 % or less when scaling them via a simple linear approximation derived from trace end markers only. For most boards, a full regression creates worse relative error.

The absolute error between different time sources is shown in Figure 12.6. Worst case errors below 40 ms error in many cases; low power CPU cores are noticeably worse again, but still below 80 ms. A full linear regression reduces the mean absolute error as expected, but its effect on worst-case timings is less pronounced.

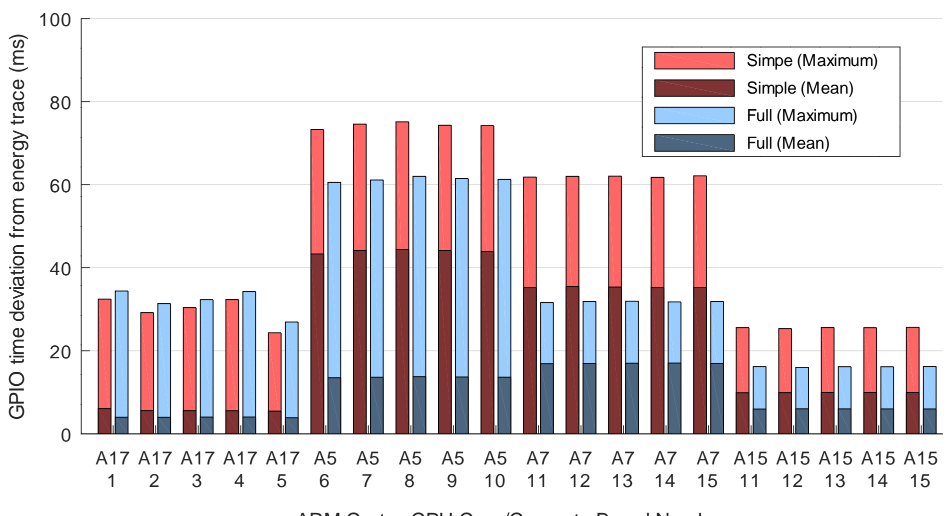


(a) GPIO and energy trace timings

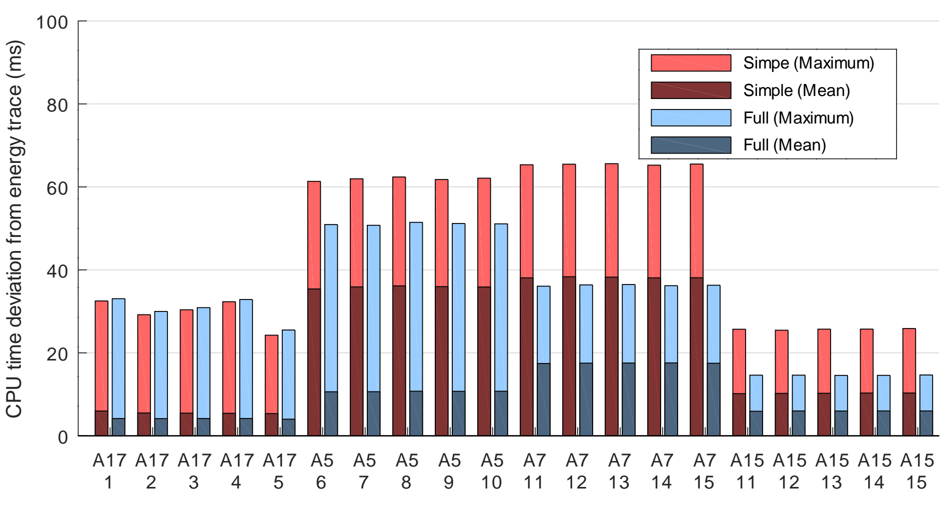


(b) CPU timer and energy trace timings.

Figure 12.5.: Maximum and mean relative deviation between different timings using two different approximations: 'Simple' uses a linear correction factor calculated from trace end markers, while 'Full' is a linear regression over all task execution events.



(a) GPIO and energy trace timings



(b) CPU timer and energy trace timings.

Figure 12.6.: Absolute deviation between different timings using two different approximations: ‘simple’ uses a linear correction factor calculated from trace end markers, while ‘full’ is a linear regression over all task execution events.

12.3. Discussion

12.3.1. Energy

Shunt Resistance The most glaring deviation from expectations is the effective value of shunt resistances. The results in Table 12.1 show that actual resistor values seem to be up to 7.6 % lower than the nominal value.

I could eliminate the resistors as source of this discrepancy. The instrumentation amplifier is another possible source, because it is connected in parallel to the shunt resistor. The input resistance might be the origin of the difference, but its data sheet does not specify it. Since the second set of shunt resistor values showed much less error, this is not a plausible explanation.

This leaves the electronic load as a possible culprit. If it had a faulty calibration, that could explain the error, including the fact that the overall system still had a good linearity. Unfortunately, at this point in the investigations I did not have access to this exact unit anymore – the measurement process was automated and lasted several days, so I did not inspect the results immediately.

I used a different device for the measurements of the 100 mΩ shunt resistors, and I made sure that the actual current matched the nominal current by measuring it independently. Given that this test was successful, it is likely that the first electronic load had a defect.

There is no way to test this hypothesis. Nevertheless, even the faulty measurements behaved linearly, just the scale seemed to be off. The new set of resistors show good values overall, so these findings can simply be accepted as they are with no consequence on further evaluation steps.

DC Current The measurements show a linear approximation error that is below the expected accuracy of 1 %, at least for shunt voltages over 25 mV. The error shrinks to 0.5 % for shunt voltages over 40 mV.

Channels have a slight offset error which result in a worst-case error of about 3 % when using the naïve scaling factor of 25 µV with no offset voltage. If users would like to have more accurate predictions, they can perform two calibration measurements on each channel. On the other hand, an overall error below 5 % is acceptable in many situations, so it may not be worth the effort.

This is no problem for the intended usage: simulation models primarily use the difference between idle power and active power, which eliminates a constant offset; the remaining idle power offset is constant over the entire prediction time, so it can be corrected after the fact, should the need arise.

DC Voltage Results show that accuracy is very good; relative error is below 0.15 %. This was to be expected, since the involved components are identical, apart from one less active component, and one less shunt resistor.

The ADC uses a differential amplifier as input stage. It samples voltages centred around the reference voltage, and a 5.000 V signal indeed leads to the exact centre value.

With increasing difference from the centre, the relative error increases. The measurement results suggest a voltage range of $5.000 \pm 1.038\,\mathrm{V}$ instead of $\pm 1.024\,\mathrm{V}$ as specified in the data sheet. Since the error is so low, eliminating it is not worth the effort. I used the ideal formula for the remaining evaluation.

Channel Independence Measurement channels are sufficiently independent from each other. Even under worst possible conditions (sudden high current change on one channel, small signal on neighbouring channel) the effect, if there is any at all, is smaller than signal noise. Given that I could only observe an induced error of 0.3 sample counts, I did not examine whether the observation was actually caused by the source signal or if it was just a random fluctuation.

Frequency Response The frequency limit is much lower than the explicit filter element would have enforced. Layout of PCB signal traces and internal microcontroller signal path contribute to this result in similar orders of magnitude.

The final frequency limit of 15 kHz almost perfectly fits the intended usage: Low-pass filtering does not change overall energy when integrating over a sufficiently long time span. For the same reason, the exact shape of the frequency response curve does not matter (yet it is almost ideal below 10 kHz). Only aliasing must be suppressed effectively. For a sampling rate of 100 kHz, 15 kHz is much lower than the Nyquist-Shannon frequency, thus aliasing is sufficiently suppressed.

Furthermore, the recording mode used for this thesis averages 10 samples at a time. This reduces the effective sampling rate to 10 kHz; since the frequency limit is above that, the effective temporal resolution in the averaged signal is not reduced by the analogue signal path.

12.3.2. Timing

Linearity Deducing timings from energy traces works well: When comparing linear approximations of different time sources with energy traces, the simple linear approximation deviates by less than 0.1 % in the worst case. A full linear regression almost reaches the documented error of uncompensated crystal quartz oscillators (0.01 %) for some boards.

For most boards, the average error is better than the worst case error by an order of magnitude. One ASUS TinkerBoard reaches an average error of 0.001 %. The big difference is probably due to the comparably imprecise detection of energy signatures in the energy trace.

I regard it as highly unlikely that these results are just due to a coincidental correlation – it would involve 18 devices of five unrelated sources (different manufacturers, different device classes, acquisition several years apart). Thus, the ability to create a linear approximation with such a low margin of error confirms the accuracy assumption stated in Section 11.2.1.

Absolute Error Likewise, absolute accuracy is excellent: Errors were less than 80ms throughout. As discussed in Section 11.2.1, during kernel characterisation there is a 2 s pause before and a 300 ms pause after each kernel execution. In fact, I chose the trailing pause based on the absolute error measured here: For an execution that is measured to run from t_{start} until t_{end}, kernel characterisation can safely sample the energy signal from $t_{\text{start}} - t_{\text{err}}$ through $t_{\text{end}} + t_{\text{err}}$ without including unrelated energy data[2], if both pauses are at least $2t_{\text{err}}$.

I decided to use $t_{\text{err}} = 150\text{ms}$ during kernel characterisation so that I can be sure that even a slowly rising signal is captured entirely, while keeping a safe distance from following unrelated energy data.

12.3.3. Summary

Overall, the evaluation of the measurement platform shows that it is well suited for the intended purpose. Accuracy is good throughout. Relative comparisons show less than 1 % error even without calibration, and absolute readings are below 3 % error (uncalibrated).

The most notable exception is measurement of small currents. Users of the measurement platform should make sure that they choose shunt resistor values so that the expected load leads to a voltage drop of more than 25 mV; for maximum accuracy, values should exceed 40 mV.

As a conclusion, the measurement infrastructure fully meets the goals of Contribution 7, but users are advised to note the design guideline stated in the previous paragraph.

[2]Assuming, as stated earlier, that I subtract idle energy.

13. Characterisation of the Evaluation Platform

The evaluation platform itself is a minor contribution which only aims to deliver a ratio of computation speed over communication speed in the same order of magnitude as an HPC system.

The output of the characterisation process is required by the prediction methodology, so it is an important preparation for the evaluation of the core methodology.

13.1. Setup

13.1.1. Platform Model

I created a program that can generate platform graphs of the evaluation platform containing any subset of available cluster servers.

The communication time model is realised by a single pair of bridges (of opposing direction) per cluster server. These represent the Ethernet interface of the server. The other communication graph nodes do not get time models, since the platform characterisation process doesn't yield more detailed data.

13.1.2. Platform Characterisation

Sustainable Clock Speed I expressed the clock speed test presented in Section 10.4 as a trivial task graph: On one core, an alternating sequence of `MARK` and a dummy calculation kernel runs. On another core, there runs a special kernel which just tries to produce as much heat as possible, which happens to be a tight floating-point math loop on the CPUs I tested. That second core has a flat energy profile, so it does not interfere with detection of the `MARK` kernels on the first core.

Using this test, I tried various frequencies using 200 MHz steps until I found the fastest working configuration.

Platform Power As discussed in Section 10.4.2, I used the final clock speed benchmark to check for adverse effects. I also use it to determine a table of correction factors between boards of the same architecture.

For each `MARK` kernel I calculate the average power of the 2 s pause and the maximum power of the following dummy calculation kernel. From these value pairs, I discarded the last one, since the heat-up kernel on the second core might have ended at that point;

in order to ignore data points from a completely cold system, I also discarded the first three value pairs.

I chose the last board of each group as reference. I then calculated the mean of the pause power values and the mean of the maximum power values and used them in a simple linear regression. The result was a table of scaling factors and absolute offsets between the reference boards and all other boards. To determine idle power, I calculated the average power of the initial pause before the first kernel. I checked all results for any extraordinary heat effects that should be addressed.

Idle power and correction factors might vary with ambient conditions. For later measurements I calculated new correction factors and idle power from each set of power traces. This is not possible for predictions, of course, so I used the idle power determined in this step in the platform resource model.

In order to get an impression of the proportions between idle and active power, I looked at the power trace start markers, since they are designed to consume as much energy as possible and should approximate peak power.

Communication Timing I then performed the communication characterisation as described in Section 10.4. Using the TCP networking layer, I generated an initial latency estimate. I then iterated a few times with different Eth packet latencies, until I arrived at the fastest timings that had reasonably few outliers.

I used transmission sizes of 100 kByte, 1 MByte, 10 MByte, 50 MByte, 100 MByte, and 150 MByte for each of the benchmarks presented in Section 10.4. This covers the range of data sizes that will be used in later evaluation steps.

Communication Power As described in Section 10.4.4, I use the above benchmark to measure the average value of each architecture's idle and transmission power (per direction) and used the difference as power value in the platform resource model.

After this step, the platform resource model was complete.

13.2. Results

13.2.1. Clock Speed

As a result from the benchmark procedure, I initially set the CPU frequencies of the XU4 Cortex A7 cores to 1400 MHz, the A15 cores to 1600 MHz, and the TinkerBoards to 1 GHz. I was unable to find a reliable operating point for the C1+ boards after trying all frequencies between 200 MHz and 1.4 GHz; see Section 15.2 for further evaluation of the anomalies. I decided on a mid-range 1 GHz for these boards.

Later during the evaluation process, I discovered that the XU4 boards, specifically the A15 cores, showed signs of thermal throttling after prolonged usage. While executing a series of identical kernels, a random kernel in that sequence would suddenly have twice the execution time compared to the other kernels in that sequence. This only happened

after more than one hour of operation, and I could not identify a correlation with any other event.

The relative effect of this anomaly lessened with lower clock speed, and at 600 MHz it disappeared completely. It appears as if the XU4 firmware suddenly decided to unconditionally switch to 600 MHz for a few seconds.

The anomaly had a significant effect on the predictability of execution times, so I had to repeat the evaluation with a lower clock speed than (apparently) possible. Unless otherwise noted, the remaining evaluation shows results for the 600 MHz configuration.

13.2.2. Platform Power

Figure 13.1 gives a rough impression of the power behaviour of all 15 boards. Table 13.1 shows the calculated correction factors and idle power values for the procedure as described. Idle power (i. e. static power for the purposes of this thesis) varies by up to 5 % between boards of the same type. Scale (i. e. dynamic power) varies a bit more, with up to 20 % worst-case difference for XU4 boards. Two boards (one C1+ and one XU4) have a noticeable power offset, they consume about 0.6 W more than their siblings.

Figure 13.1 shows power over a longer time span without applying the linear correction factors[1]. Most channels show virtually no power increase over time, except for ASUS TinkerBoards. Those show increasing power that resembles a logistic function or a similar asymptotically constant function.

Note that pauses are not at idle power – these measurements also originate from the load benchmark as before, which means that a second core is still running dummy calculations meant to increase power usage. Pauses are merely wait times on the first CPU core.

Figure 13.2 shows the ATB boards in more detail. The power difference between active periods and pauses seems to be constant, but some channels have high variance – some outliers differ by more than 10 %.

13.2.3. Communication Timing

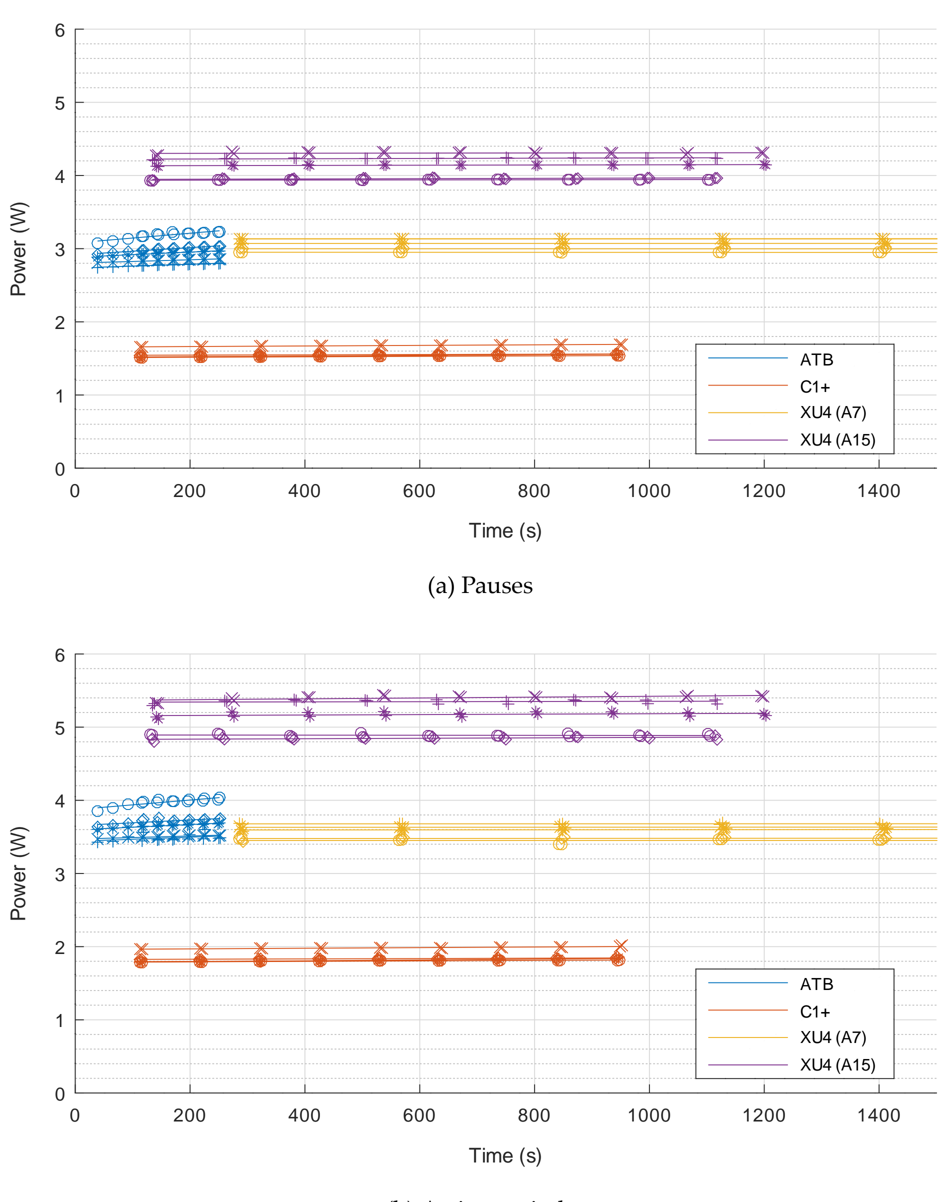
After performing the iterative communication characterisation process, I arrived at the timings shown in Table 13.2; it shows the delay between two sent packets, enforced on the sender side.

For a given packet delay, ideal transmission timing is the packet delay times the number of packets required for the transmission. Figure 13.3 shows the differences between ideal and measured transmission time across all benchmark for two sets of packet delays. One set is the selected set of packet delays, and one set is a tighter set of delays that shows how pronounced these measures react when timings cross their lower threshold. Table 13.2 shows the timings I used for the remaining evaluation.

[1] These two figures use the 1.6 GHz configuration for the XU4 boards in order to analyse a more challenging scenario.

Table 13.1.: Correction factors and idle power for the 15 boards.

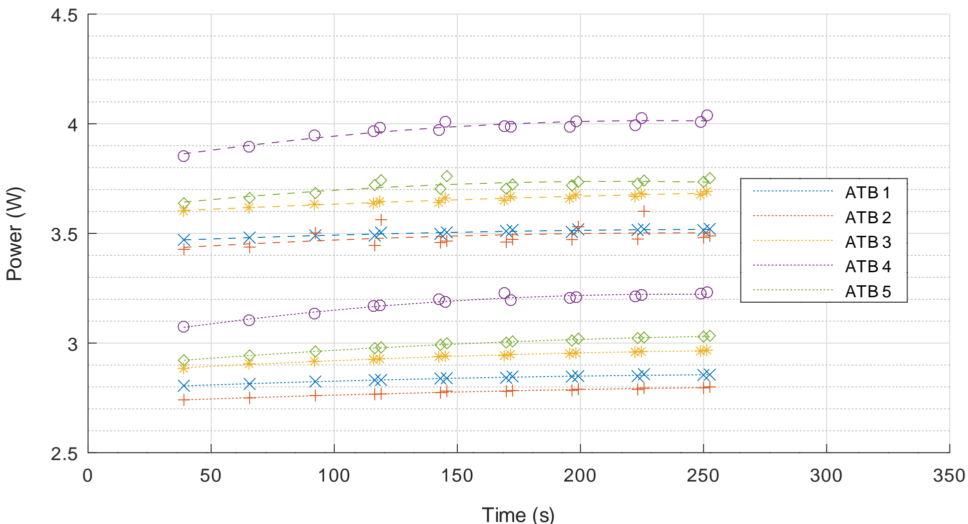
Board	Short	Scale	Offset (W)	Idle (W)
ASUS TinkerBoard	ATB 1	1.062	0.030	2.190
	ATB 2	1.033	0.158	2.195
	ATB 3	0.983	0.053	2.182
	ATB 4	0.984	-0.074	2.154
	ATB 5	1.000	0.000	2.177
ODROID C1+	C1+ 1	0.982	-0.627	1.126
	C1+ 2	1.011	0.094	1.138
	C1+ 3	0.999	-0.019	1.123
	C1+ 4	0.976	0.092	1.123
	C1+ 5	1.000	0.000	1.124
ODROID XU4	XU4 1	1.120	-0.585	2.279
	XU4 2	0.914	0.139	2.347
	XU4 3	1.037	-0.066	2.364
	XU4 4	1.030	0.066	2.355
	XU4 5	1.000	0.000	2.356



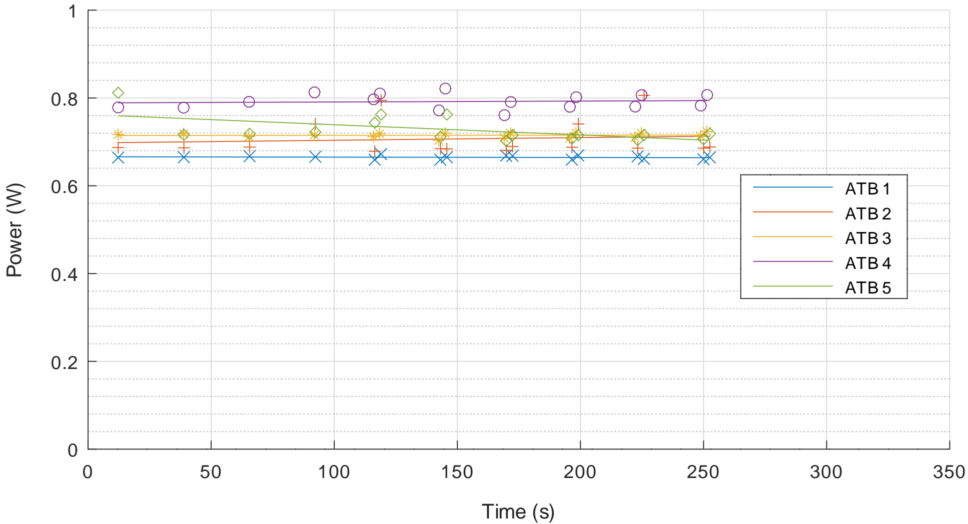
(a) Pauses

(b) Active periods

Figure 13.1.: Power over time during pauses and active periods. Lines are linear approximations.



(a) Pause (dotted) and active (dashed) current. This time, lines are quadratic approximations in order to show the gradual change in slope.



(b) Difference between active and pause. Lines are linear approximations.

Figure 13.2.: Power over time for the five ASUS TinkerBoards.

Table 13.2.: Final communication packet timing and net bandwidth (BW) for the Eth networking layer.

Destination →	ATB		C1+		XU4	
Source ↓	Delay	BW	Delay	BW	Delay	BW
ATB	50 µs	30 MB/s	110 µs	14 MB/s	220 µs	7 MB/s
C1+	80 µs	19 MB/s	110 µs	14 MB/s	220 µs	7 MB/s
XU4	190 µs	9 MB/s	190 µs	9 MB/s	220 µs	7 MB/s

I was unable to reduce the remaining variance for the ATB boards, even with massively increased packet times. Also, the repeated evaluation of the XU4 boards showed that timings are sensitive to clock frequency – at 1.6 GHz, stable XU4 packet delays were exactly half as long.

13.2.4. Communication Power

I examined the above benchmarks to derive communication power as described in Section 10.4.4. Figure 13.4 shows the entire energy trace from one `iobench-1` run. It is easy to see the active communication phase, which adds slightly less than 0.1 W to idle power.

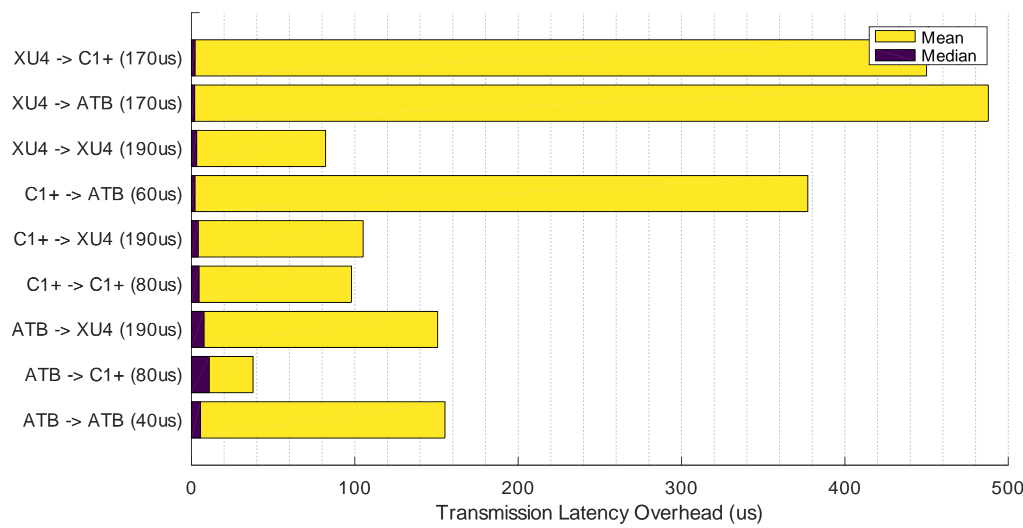
Figure 13.5 shows an analysis of all communication time benchmarks, converted to energy per packet and plotted in relation to packet delay. It shows that the energy per packet is in the same order of magnitude for all boards and constellations. In order to emphasise the general trend, Figure 13.5 also contains linear approximations where possible. Due to the limited data set, these cannot be considered very reliable; what they do show is that bidirectional transfer (solid lines) not exactly the sum of send and receive traffic (dashed and dotted lines), but it is close.

13.3. Discussion

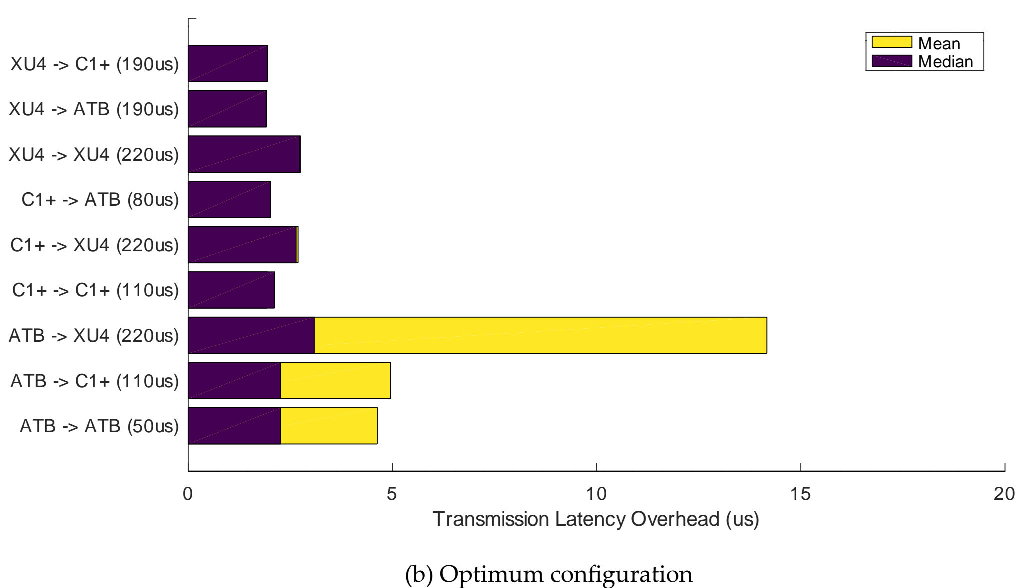
13.3.1. Sustainable Clock Speed

No board could run at its nominal clock speed without intrusive thermal protection measures kicking in – obviously, the clock ratings are only available in certain circumstances, similar to 'Turbo' frequencies of desktop CPUs. In Section 15.1 I report on my effort to gain more insight into the effects of fixed frequency selection.

The clock speeds ultimately selected might still trigger thermal throttling, but in a way that does not disturb timing of the nodes in an unpredictable way. The exception are the C1+ boards, and I analyse their anomalies in more detail in Section 15.2. I consider them



(a) Configuration with timings that are slightly too tight.



(b) Optimum configuration

Figure 13.3.: Eth networking layer timing difference between measured and ideal values (i. e. calculated from the stated packet delays).

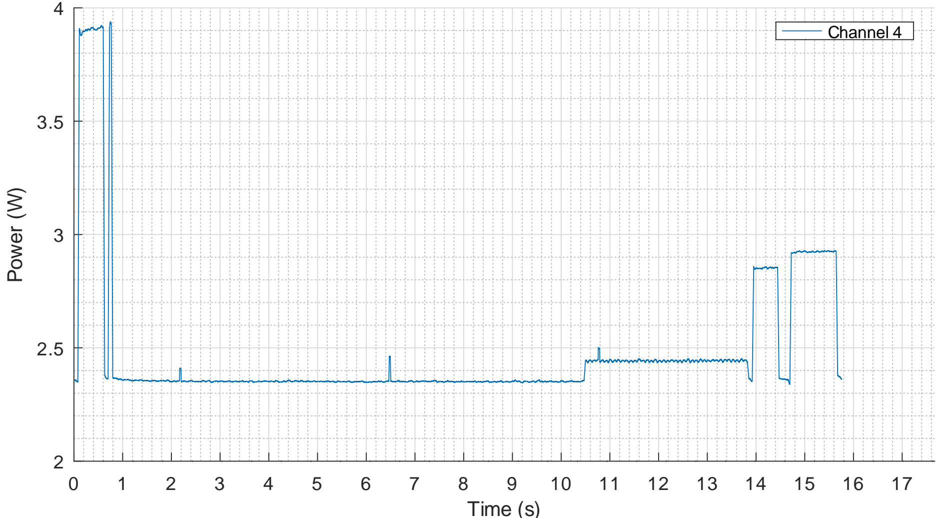


Figure 13.4.: Energy trace of a single communication benchmark, showing the receiver of a unidirectional ATB-to-ATB transfer. Communication happens between 10.5 s and 13.8 s.

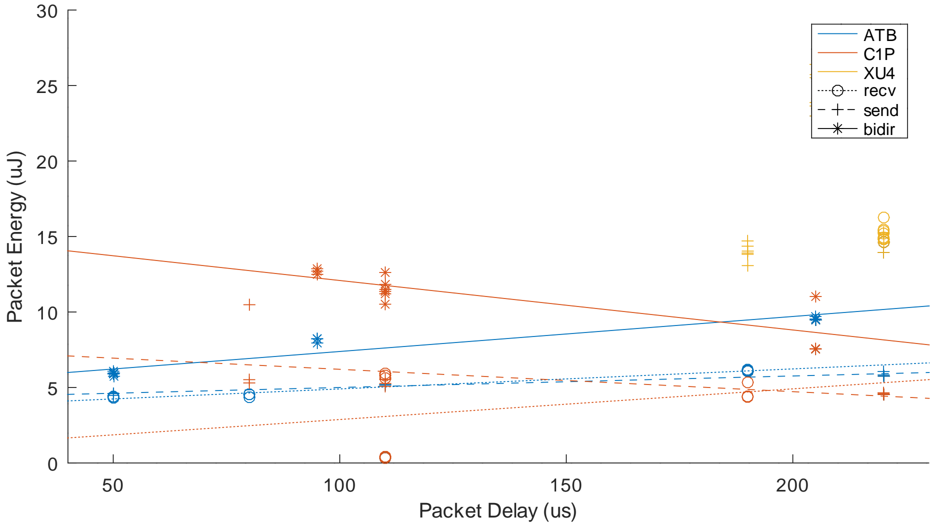


Figure 13.5.: Communication energy per packet, plotted over packet delay. Colour denotes board type, while shape distinguishes between receive-only, send-only, and bidirectional transfers. Lines are linear approximations.

to be too unreliable for the proposed methodology, but still use them during further evaluation simply to see what effect hardware has that shows adverse behaviour.

For the XU4 boards, initial results are consistent with other findings in literature: The selected clock frequencies are very similar to the best-performing clock frequencies reported in [45]. Unfortunately, the surprising long term behaviour makes this configuration unusable for the purposes of this thesis.

In [47], the author reports similar findings from other embedded platforms, where some embedded platforms seemed to be well behaved but had very rare anomalies. This means that when selecting a target platform, users should check long term behaviour over many hours or even days.

13.3.2. Platform Power

One important insight of the power characterisation is that boards of the same type vary noticeably – about 10 % deviation is common. The exact reason is unknown, but plausible causes are variations in efficiency factor (of power converters for example) and differences in board components other than CPU and RAM.

There are two outliers with a constant power offset of about 0.6 W. Given that this is off-the-shelf consumer grade hardware, this shows that build quality can significantly affect power behaviour, even when performance is not distinguishable from other specimens.

This natural variation means that for a platform of this style, accurate energy predictions only make sense if either natural variation is low or the entire platform has been characterised. The intention that a single specimen of each CPU or board is sufficient for the methodology is only valid under such a constraint. For their own target platforms, users should at least check a few random samples until they have sufficient evidence about natural variation.

Furthermore, the difference in (apparent) efficiency factor means that if minimising energy is a user goal, the resource model used for scheduling should take individual board differences into account; the algorithm presented in Section 9.6 doesn't.

On the other hand, predictions using an idealised platform model without variations are useful on their own. They might not be able to predict exact energy values, but might still produce valid rankings of multiple alternative implementations, which is the main goal anyway.

On the positive side, the load test confirms one design goal of the evaluation platform: to provide a high degree of heterogeneity. The load test has shown quite clearly that board types vary widely in power and timing. This will exercise the proposed methodology's ability to predict highly heterogeneous hardware.

Finally, the initialisation sequence in Figure 13.4 shows that idle power is about 60% of peak power. While there are many factors that preclude a direct comparison, this shows that the power ratio is at least in the same order of magnitude as the HPC setup referenced in Section 11.3.

Heat-related Power The only platform that showed a heat-up effect at all was the ASUS TinkerBoard. This is a result of the evaluation setup: I intentionally disregarded the first three data samples in order to skip the heat-up phase, and for the other boards this strategy worked. However, ATB boards are much faster than other boards – they finished the entire benchmark before A7 cores even generated their first recorded data sample.

The results show that thermal equilibrium seems to be achieved within about two to three minutes of active computation. This is suitable for the benchmarks chosen for the main evaluation, which run for much longer time spans.

Finally, the visible heat-up effect of the ATB boards matches the subthreshold leakage effect presented in Section 4.1.5: there is a significant effect that affects static power, while there is no visible effect on dynamic power.

13.3.3. Communication Timing

The characterisation procedure for the Eth protocol works well, exhibiting a very strong indicator for too tight timings. This means that it is easy to optimise throughput while maintaining a high degree of prediction accuracy. ATB boards show a higher variance than the other boards, however. This might be due to system-specific differences in the respective Linux kernel version, configuration, and drivers. Nevertheless, their behaviour remains sensible enough for the purposes of this thesis.

One of the design goals of the embedded cluster system is to provide a ratio of computation speed over communication speed that is similar to HPC systems. The net bandwidth that the predictability-optimised Eth protocol shows is well within this range: at up to 30 Megabytes per second, the net bandwidth, it is about one quarter of the achievable bandwidth of Gigabit Ethernet.

A real HPC cluster would likely use faster technologies, thus would have at least 40 times faster communication speed for 10 GBit/s Ethernet, for example. Computational speed is difficult to state in general, but it sounds plausible that a factor of 40 or more matches real hardware.

The de-facto standard LINPACK benchmark supports this impression: According to [45], the Cortex A15 cores of the XU4 boards reach up to 4.96 GFLOPS (billion floating-point instructions per second). On the other end of the spectrum, one of the fastest HPC nodes in 2019, a dual-processor configuration of the AMD Epyc 7H12 processor, reaches 4296 GFLOPS according to a press release [44]. Such a system is roughly 800 times faster than a single XU4 board; a high end 200 GBit/s InfiniBand communication infrastructure would match this factor of 800.

13.3.4. Communication Power

Communication power does not show any surprising results. The effect is small but noticeable. Given that communication power is much smaller than computation power, a constant energy per packet sent and received is a sufficiently accurate approximation.

Based on Figure 13.5, the simulation model will use 5 μJ per packet for ATB and C1+ boards, and 10 μJ for XU4 boards.

13.3.5. Summary

The platform behaved in a way that is mostly compatible with the modelling approach used in this thesis, albeit with two important exceptions: the reduced clock speed of XU4 boards is acceptable for the purposes of this thesis, but it makes comparison to other publications using the XU4 infeasible. Secondly, the ODROID C1+ boards show unexplained behaviour which might affect predictability, thus they must be considered as unreliable for timing and energy behaviour. The error is large enough that the proposed kernel characterisation process might not work. Only the ASUS TinkerBoard did not exhibit any unexpected behaviour.

I actually introduced the ASUS TinkerBoards only after discovery of this problem, so that the platform still shows significant heterogeneity even when disregarding C1+ boards.

A main insight of this part of the evaluation is that stability and predictability of embedded platforms can vary widely. In [47], the author observes that this not an issue for server-grade x86 hardware, at least not at that time. Users should confirm that their hardware does not exhibit anomalies that would preclude usage of the proposed methodology.

As far as the other system properties are concerned, the platform meets its main design goals of being complex enough for the proposed methodology, and being a scaled-down version of an HPC system. This means that under the restrictions discussed above, the proposed platforms meets the goals of Contribution 6. However, the following section will evaluate an important additional constraint that is part of Assumption 8 as stated in Section 7.

14. Evaluation of the Overall Methodology

For evaluation of overall prediction accuracy, I performed the design steps as presented in Section 9. As stated in 11.1, the main goal is to determine the approximation error of simulation predictions.

14.1. Setup

14.1.1. Overview

As benchmark application, I chose Cholesky matrix subdivision, a classic linear algebra algorithm. I created a generator program for application models of different problem sizes (Section 9.2.1). Then I performed kernel characterisation (Section 9.5).

With these generic inputs, I then examine multiple scenarios. For each one, I created a mapping (Section 9.6), ran the mapped application on the evaluation platform (Section 9.8), and simulated it on the simulation model of the platform (Section 9.7).

Due to their questionable reliability, the C1+ boards are not used throughout this section.

14.1.2. Measurement Details

I uses the evaluation and measurement platform presented as part of this thesis. Due to their unreliability, I excluded C1+ nodes. The platform ran in a plain office environment that did not offer systematic ambient temperature control.

For all measurements, I only considered dynamic power. In order to do so, I measured each board's power after the application had ended and the execution runtime was completely idle. I used the end in order to account for heat – idle power during startup was usually about 10% less, but Section 13.2.2 did establish that temperature seemed to be stable after about two to three minutes of operation.

Figure 13.4 illustrates this: Initial idle power can be measured during the 10-second startup sequence. At around 14 s, the end marker sequence begins, with two easily visible intervals of high energy consumption. At the end of the first marker, all worker threads have exited. At the end of the second marker, the runtime has been shut down completely and idle power can be measured.

I also normalised measurement channels as shown in Section 13.2.2: From each set of power traces, I determined new linear correction factors in order to reduce differences due to ambient temperature changes.

14.1.3. Cholesky Matrix Subdivision

I chose distributed Cholesky matrix decomposition as primary benchmark application. It belongs to the class of linear algebra programs (see Section 4.2.1) and has a nontrivial task graph structure with highly parallel sections as well as synchronising/consolidating tasks. Properties of linear algebra programs are also found in other important application classes: For example, deep learning applications rely on matrix multiplication a lot, and neural net structure has a natural representation as data flow graph. Finite element simulations also emphasise arithmetic over control flow, and they exhibit complex data flow if irregular meshes are involved. Thus Cholesky represents aspects of a significant set of HPC programs.

Distributed Cholesky decomposition works by subdividing the source matrix into equal-sized square tiles; each task works on one to three tiles at a time. Figure 9.3 shows the Cholesky task graph for a subdivision of 5×5 tiles. Using this subdivision as parameter, I wrote a skeleton application that can create task graphs of different complexity and size. Each task graph solves the same underlying problem, but with different computation and communication granularities. This makes them ideal to test how well the methodology supports different task sizes.

Cholesky matrix decomposition uses four kernels:

`GEMM` is generic matrix multiplication: $C := AB + C$.

`TRSM` solves matrix equation $AX = B$, where A is a triangular matrix and X is the output matrix.

`SYRK` performs a rank k update of a symmetric matrix: $C := AA' + C$.

`POTRF` is (non-distributed) Cholesky matrix decomposition.

Each kernel has one kernel variable, the tile size. It specifies the number of rows (and columns) of one matrix tile.

For evaluation, I added two technical kernels that operate under Assumption 5 (see Section 7):

`MATSRC` pseudo-randomly generates a single tile of a Hermitian, positive-definite matrix as required by the Cholesky algorithm. It does so without generating the whole matrix, which would not fit into memory. It has one extra variable, the tile index that identifies the tile to generate.

`MATSINK` takes the resulting value of a matrix tile and performs some finalising action on it. Actions might include checking the value for correctness, storing the data for later checks, or just ignoring it. I chose the third option in order to minimise the influence on the running algorithm, as the other two behaviours would incur significant memory usage and/or data transmission.

Memory size for these is an important factor; I tried to maximise tile size in order to stress the platform. On the evaluation platform the benchmark works well with 1024×1024

element tiles for 10×10 matrix subdivision (800 MiB matrix size), while 2048×2048 element tiles (32 MiB each for 3.2 GiB total matrix size) did lead to occasional memory overflow on some nodes.

Just as a side note: The 3.2 GiB configuration shows a potential advantage of task graph applications: No compute node is able to keep the entire matrix in memory, but due to inherently distributed data the application still works (in theory – if the runtime had smarter memory management).

14.1.4. Computation Resource Model

With platform and application known, I then proceeded with kernel characterisation according to Section 9.5.2. To increase the data set, I used the full parallelism of the platform: I ran one task graph for each CPU core type. Each task graph contains 10 consecutive executions of the kernel to characterise, repeated for each of the five boards with that CPU type. This results in 50 executions for a single kernel configuration, each with different input data.

Each single characterisation consists of an appropriate number of `MATSRC` tasks, the kernel to characterise, then a `MATSINK` task. Between these tasks, there are `MARK` tasks. In order to vary input data, each `MATSRC` task was configured to generate a different matrix tile.

I repeated the whole process for all four main kernels and four tile sizes from 128×128 through 1024×1024.

For the secondary time model (see Section 9.5) I looked at the task graph itself and at a simulation run (without secondary time model) to see if it is possible to reduce the characterisation space. To get a sense of what such a subset selection would miss, I characterised every main kernel with a competing load of one or two of each of the four main kernels.

14.1.5. Application Benchmark

Using the above inputs, I generated a set of task graphs that solve the same overall problem (decomposition of a 10240×10240 matrix) using the tile sizes characterised during the previous step. Table 14.1 shows the application configurations I chose for evaluation. They fulfil Assumptions 1 through 4.

This selection represents the optimisation question: Which task granularity is best for the given platform?

Using the computation resource model I then ran the mapping algorithm presented in Section 9.6 in order to get executable (i. e. mapped) task graphs. I also configured simulation and physical execution runtime according to the platform resource model.

Finally, I ran simulation and physical execution of each task graph and compared overall maketime and energy consumption.

Since time measurement also captures individual task timings, I also plotted a detailed task execution trace and compared it to a similarly plotted simulation trace for additional insights.

Table 14.1.: Evaluated application configurations.

Name	Matrix Tiles	Tile Size	Matrix Size	Tasks
Ch-10	10×10	1024×1024	10240×10240	330
Ch-20	20×20	512×512	10240×10240	1,960
Ch-40	40×40	256×256	10240×10240	13,120
Ch-80	80×80	128×128	10240×10240	95,040

In order to determine the time the simulator needs to run one simulation, I note the execution time on an otherwise unloaded Intel Core i7-8750H CPU running at 2.2 GHz.

14.2. Results and Discussion

14.2.1. Computation Resource Model

Table 14.2 shows the values of the primary computation resource model as generated by the kernel characterisation process.

An additional result is that the characterisation confirmed that kernels have no significant data-dependent timing variation – differences were less than 1 %. Given the coarse granularity, this is actually not too surprising: With large data sets, the chance of different effects cancelling each other out is pretty high. A special case might be sparse matrices with matrix tiles that are entirely zero, but those can be more efficiently handled at the task graph level anyway.

Figure 14.1 shows the result of a preliminary simulation run without a secondary time model. The mapping algorithm did not allocate any tasks on the C1+ boards, and ATB boards only ran source and sink tasks, plus a few POTRF tasks.

With the increased parallelism in the other configurations, ATB boards did see significant usage, but the mapping algorithm never allocated any task on C1+ boards; since I consider them unreliable, this was my intention in the first place, so I did not need to force the mapper to ignore them.

It's somewhat surprising that even during the massively parallel initial parts of Cholesky, the low speed of the Cortex A5 cores plus communication overhead resulted in no task allocations at all[1]. The Cortex A7 cores of the XU4 are used sometimes – fast cross-core communication seems to outweigh their speed disadvantage.

By visual inspection of the simulation result, we can see several kernel properties of the Cholesky benchmark:

[1] As Section 9.6 shows, the mapper uses a significantly simplified prediction model, so its output should not be used to infer properties of applications and/or the platform.

Table 14.2.: Results from the characterisation procedure.

CPU	Size	GEMM		TRSM		SYRK		POTRF	
		t in s	E in J	t in s	E in J	t in s	E in J	t in s	E in J
C1+	128	0.226	0.030	1.009	0.036	0.098	0.014	0.005	0.010
XU4 A7		0.459	0.041	0.101	0.033	0.111	0.027	0.007	0.018
XU4 A15		0.136	0.056	0.041	0.038	0.050	0.041	0.003	0.021
ATB		0.073	0.048	0.026	0.022	0.026	0.022	0.001	0.006
C1+	256	2.379	0.463	0.882	0.170	0.995	0.149	0.044	0.015
XU4 A7		4.496	0.369	0.924	0.162	1.027	0.120	0.053	0.027
XU4 A15		1.157	0.414	0.325	0.144	0.424	0.190	0.024	0.028
ATB		0.600	0.364	0.210	0.141	0.233	0.159	0.013	0.013
C1+	512	51.80	22.80	22.03	10.00	22.26	9.624	0.540	0.150
XU4 A7		59.85	12.92	24.52	5.770	23.26	4.959	0.637	0.126
XU4 A15		17.69	6.332	3.566	1.700	5.806	1.879	0.179	0.088
ATB		15.09	9.146	5.573	3.849	4.563	2.911	0.122	0.089
C1+	1024	422.5	173.6	158.2	72.10	147.8	64.86	3.213	1.308
XU4 A7		510.3	110.9	215.3	50.44	240.4	53.74	5.635	1.052
XU4 A15		269.1	140.2	70.70	45.84	126.1	59.18	1.684	0.933
ATB		134.7	79.84	52.76	35.01	54.35	34.04	0.971	0.831

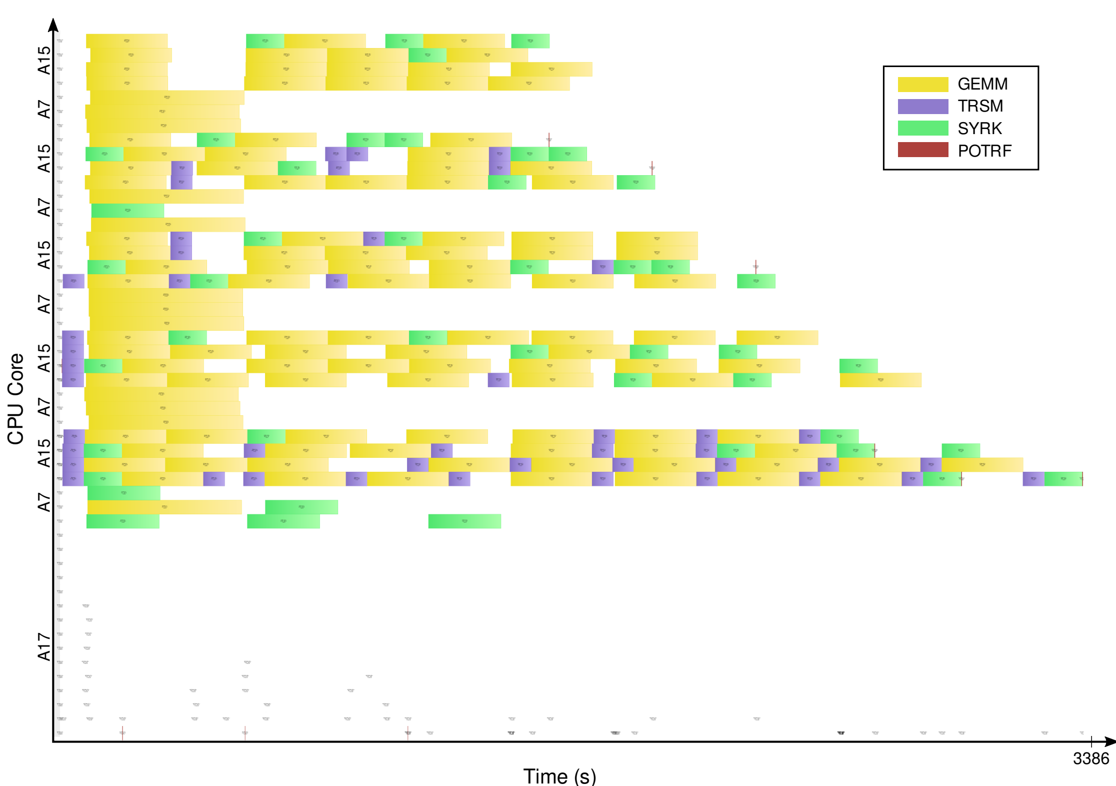


Figure 14.1.: CPU activity for `Ch-10` as determined by a simulation without secondary time model. Each row of boxes represents one compute core, while the coloured boxes themselves represent tasks, with colours showing the kernel being executed.

- GEMM is the most important kernel by far, almost everything runs concurrently to GEMM kernels.
- TRSM is the second-most frequent, and it seems to be on the critical path, so its behaviour matters, too. The chance of TRSM/TRSM concurrency seems to be high.
- SYRK occurs almost as often as TRSM. It is often followed by pauses in CPU activity, so its impact on global prediction accuracy is unclear.
- TRSM/SYRK concurrency is rare and doesn't seem to occur without GEMM also in the mix.
- POTRF is rare. Given that its execution time is less than $\frac{1}{100}$ of GEMM, its detailed behaviour is bound to be insignificant.

Table 14.3 shows a subset of the slowdown factors generated by the secondary characterisation runs. The results confirm the expectation that slowdown factors are less than 2.0 (see Section 9.5).

The table shows that the different board types vary a lot regarding reaction to different load situations. For example, the SYRK kernel scales badly on the C1+, while it is pretty constant on the XU4, except when run with two competing SYRK tasks. Another unusual example are the Cortex A15 cores of the XU4 boards: They have about 10% slowdown in most situations, but the TRSM kernel scales particularly badly on them. As a final example, ATB boards have higher slowdowns overall.

These three examples show that slowdown factors are important, since board behaviour can be quite counterintuitive. If users choose to characterise only a subset of load situations as suggested in Section 9.5, they should not make assumptions about platform behaviour. They should base subset selection solely on application structure like in the example shown above.

Characterisation Time The entire characterisation process presented in this subsection takes about two days of continuous automated characterisation runs. Post-processing of measurement results is fast and can be done in parallel to execution, so the limiting factor is execution speed of the platform itself.

This is a significant time factor that users must consider in their workflow, but it only needs to be done once for a given set of kernels and size parameters, so this is still viable for the intended purpose. This is also addressed by Assumption 7: if a kernel itself is modified or application structure changes produce new kernel interactions, the changed parts need to be reexamined.

14.2.2. Application Benchmark

Table 14.4 shows the resulting values for execution and simulation of the selected benchmark suite.

Table 14.3.: Slowdown factors of the secondary time model (competing POTRF loads left out due to insignificance).

Kernel	Competing Load	C1+	XU4 (A7)	XU4 (A15)	ATB
GEMM	1× GEMM	1.035	1.007	1.046	1.217
	2× GEMM	1.145	1.115	1.257	1.501
	1× SYRK	1.030	1.003	1.026	1.190
	2× SYRK	1.111	1.008	1.073	1.459
	1× TRSM	1.044	1.002	1.056	1.222
	2× TRSM	1.113	1.009	1.166	1.519
POTRF	1× GEMM	1.004	1.056	1.025	1.309
	2× GEMM	1.078	1.101	1.087	1.589
	1× SYRK	1.010	1.060	1.032	1.363
	2× SYRK	1.082	1.098	1.104	1.691
	1× TRSM	1.002	1.058	1.086	1.335
	2× TRSM	1.091	1.100	1.266	1.646
SYRK	1× GEMM	1.448	1.047	1.024	1.185
	2× GEMM	1.575	1.056	1.056	1.400
	1× SYRK	1.427	1.032	1.027	1.287
	2× SYRK	1.556	1.048	1.063	1.634
	1× TRSM	1.420	1.020	1.052	1.209
	2× TRSM	1.566	1.024	1.215	1.500
TRSM	1× GEMM	1.309	1.072	1.376	1.175
	2× GEMM	1.466	1.300	1.714	1.437
	1× SYRK	1.304	1.075	1.074	1.202
	2× SYRK	1.428	1.105	1.130	1.523
	1× TRSM	1.276	1.037	1.457	1.292
	2× TRSM	1.428	1.068	1.754	1.560

Table 14.4.: Results of the initial benchmark run. Power values are average power. E_{total} and P_{total} include idle power.

Name		t in s	E_{dyn} in kJ	E_{total} in kJ	P_{dyn} in W	P_{total} in W	t_{sim} in s
Ch-10	pred.	4127	21.19	114.73	5.13	27.80	0.034
	meas.	4146	18.81	111.22	4.54	26.83	
	rel.	-0.45 %	+12.64 %	+3.15 %	+13.16 %	+3.62 %	
Ch-20	pred.	1249	9.25	37.56	7.41	30.07	0.160
	meas.	1312	9.20	39.08	7.01	29.78	
	rel.	-4.81 %	+0.55 %	-3,88 %	+5.63 %	0.97 %	
Ch-40	pred.	663	4.17	19.20	6.29	28.96	0.880
	meas.	670	4.71	20.12	7.03	30.03	
	rel.	-1.04 %	-11.51 %	-4,60 %	-10.57 %	-3.59 %	
Ch-80	pred.	612	4.56	18.43	7.45	30.11	8.200
	meas.	608	3.97	17.91	6.52	29.45	
	rel.	+0.58 %	+14.85 %	+2.90 %	+14.12 %	+2.23 %	

14.2.2.1. Time Predictions

These results show that the proposed prediction methodology is able to predict timing of task graph executions with an error of less than 5 %. This is well within the targeted accuracy range and is also consistent with the accuracy of the measurement infrastructure.

In order to check the internal time model, I plotted an entire execution trace of simulation versus execution. Figure 14.2 shows the result. It shows that there is some variation, but errors appear to be randomly distributed and cancel each other out over time.

14.2.2.2. Simulation Speed

Simulation speed is also fully consistent with the goals of this thesis, in particular the usage scenarios stated for Contribution 5. Furthermore, it scales roughly linearly with the number of tasks, which means that hardware upgrades can yield much higher task counts – tens of millions of tasks would be achievable on fast simulation hosts, enough for real-world problems in the field of finite element simulations, for example (also see Assumption 1).

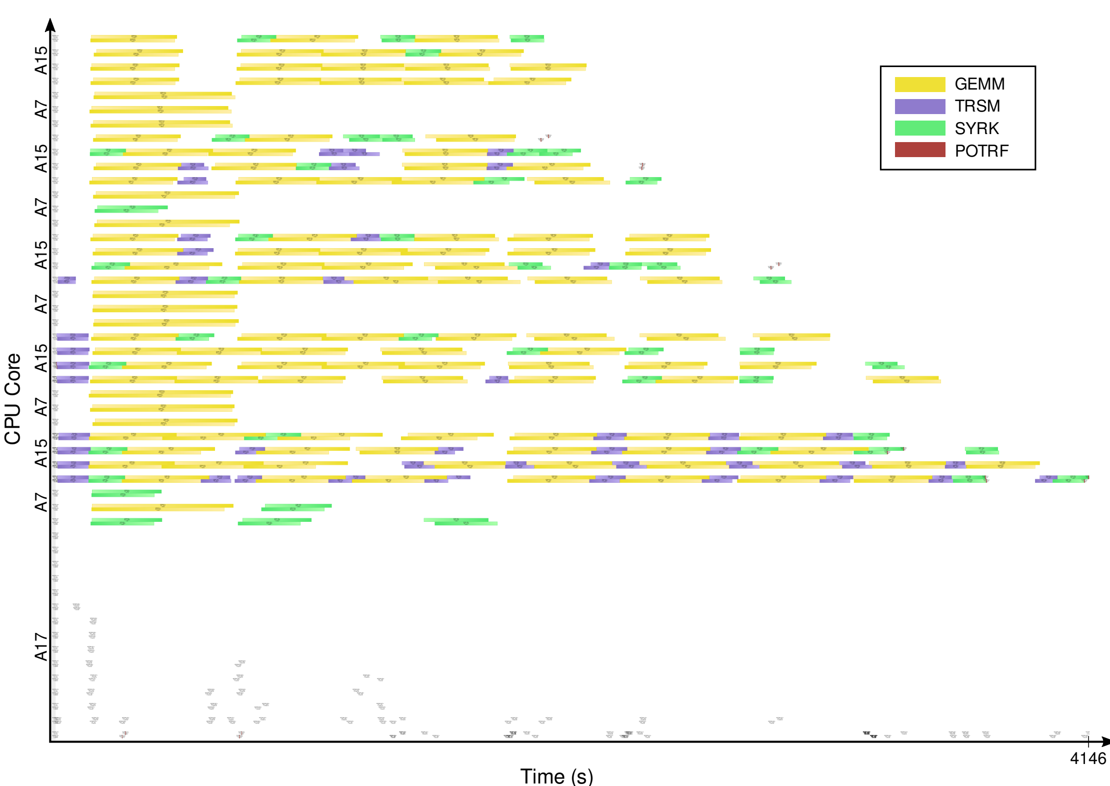


Figure 14.2.: CPU activity for `Ch-10`, similar to Figure 14.1 but showing measurement and final simulation side-by-side. Each pair of rows represents one compute core, with measurement result on the top row and simulation result on the bottom row.

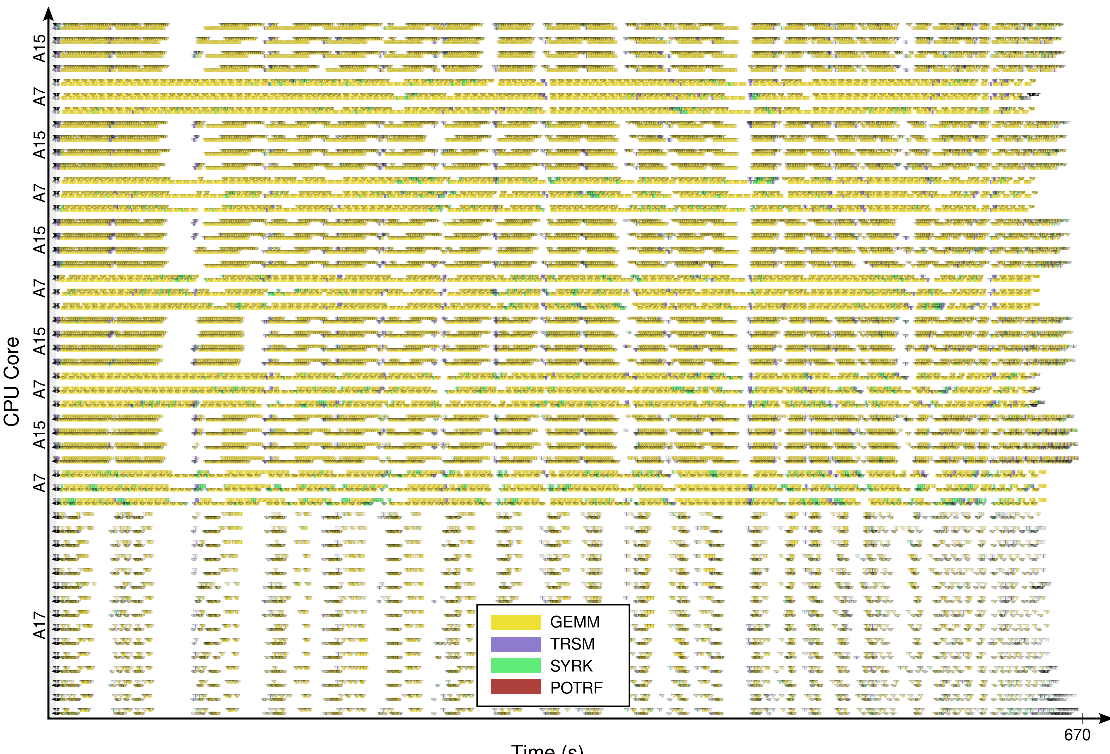


Figure 14.3.: CPU activity for `Ch-40`, similar to Figure 14.1 but showing measurement and final simulation side-by-side. Each pair of rows represents one compute core, with measurement result on the top row and simulation result on the bottom row.

14.2.2.3. Energy Predictions

Energy predictions are not as accurate as time predictions. They have an error of roughly ±15 %, which is more than anticipated. Even worse, the error is big enough that it changes the relative order of the individual benchmarks.

Values that include idle power yield less relative error, which is not too surprising, as idle power should not vary too much. In Section 13.2.2, combined idle power was measured to be 22.6 W for ATB and XU4 boards combined. This means that average dynamic power was only up to 30 % of idle power.

As shown in Section 13.3.2, peak dynamic power is only slightly less than idle power, so a real-world application does not use nearly the power that would be available. Figure 14.3 shows that even for the highest power configuration, there are long idle phases on most CPUs, and ATB boards are still not used a lot; this explains the low proportion of dynamic power.

For a better analysis of possible sources of the inaccuracy in dynamic energy prediction, Table 14.5 shows a comparison of predictions and measurements by individual board.

The table shows that with a few exceptions, per-board predictions are similar in

Table 14.5.: Comparison of energy measurements (M) and predictions (P) by individual board. All values are given in Joules.

Board	Ch-10		Ch-20		Ch-40		Ch-80	
	M	P	M	P	M	P	M	P
ATB-1	317	8	396	337	192	164	238	238
ATB -2	84	1	282	240	201	163	238	238
ATB-3	4	0	192	159	208	163	237	238
ATB-4	2	0	79	51	229	161	205	238
ATB-5	0	0	29	12	213	161	250	237
XU4-1	6009	6179	1889	1773	766	684	580	680
XU4-2	3979	4745	1650	1712	746	681	533	680
XU4-3	3349	4077	1564	1691	719	678	567	678
XU4-4	2652	3238	1545	1659	690	676	546	697
XU4-5	2412	2955	1575	1648	747	690	572	721
Sum	18810	21188	9203	9254	4713	4171	3967	4557

accuracy to global prediction accuracy. Yet there are some noteworthy observations.

First of all, ATB boards show that measurement is challenging at low energies. Examples are ATB-1 for `Ch-10` (309 J difference) or `Ch-40`, where ATB predictions are 30-60 J below measurements. In these situations, small errors in idle power detection will lead to big relative errors. On the other hand these are small absolute values, so they will have little effect on total energy. XU4 boards differ by similar values, but since the reference values are much bigger, relative error is smaller.

Another unexpected result are XU4 boards for `Ch-40`. They have almost identical loads (as can be seen in Figure 14.3), which results in pretty uniform predictions, but measurements show 10% worst-case difference (between XU4-1 and XU4-4).

Second Benchmark Run Natural variation could be a cause for this. In order to get an idea how big natural variation is for multiple executions, I ran the benchmark suite a second time. Table 14.6 shows the results of the second run. They show that timing behaviour varies very little, but energy measurements differ significantly. In fact, the difference between benchmark runs is almost as big as the difference between prediction and measurement. It is between 5 % and 10 %.

This variation can also be seen in Table 14.7, which shows energy per board. The observed irregularity in `Ch-40` is present again, but other data points show that the results really are insufficient to draw definite conclusions: In `Ch-20`, XU4-2 and XU4-3 even reversed their relative energy consumption.

Since I did not see such variation within a single characterisation run or similar situations like the one shown in Figure 13.1, the effect must be related to the temporal distance between both benchmark runs, which was ten days. This suggests external influence like ambient temperature as a possible explanation, since this is sufficient time

for slow influences to take effect, e. g. outside temperature.

Almost all boards in Table 14.7 show increased energy usage. This supports the hypothesis of a systematic external influence. This does not exclude additional random variation: the case of XU4-2 in `Ch-20` shows that there might be more than one influence.

A statistically significant test series to examine board behaviour would require that the external influence can be controlled. Temperature is the most probable explanation, so that would require ambient temperature controls. Since this was not available for this thesis, I could not investigate natural variation of the boards any further.

If ambient temperature is indeed the source of the variations, it might even explain the higher inaccuracy of `Ch-40`: The full characterisation takes about two days to complete, and the automated characterisation scripts happen to order individual characterisation runs by tile size. This means that it is possible that a slow external influence on ambient temperature affected size 256 kernels only.

Comparison with Stable Ambient Conditions To add another data point, Table 14.8 shows early evaluation results produced about one year earlier. It ran on an earlier version of the evaluation platform, which had five additional C1+ boards instead of the ATB boards, but otherwise used an identical setting. The huge difference in `Ch-40` was due to the reliability problems of the C1+ boards[2], but the other two benchmarks show good accuracy.

The main difference between this early evaluation result and the other ones presented in this thesis is the environment: A year ago, the platform ran in a regular office environment with lots of natural ventilation and stable weather conditions. The other results had two adverse factors: The office environment was not ventilated due to a global reduction of office activity[3]. At the same time, outside temperature fluctuated much more.

This again supports the assumption that ambient conditions have much more impact than expected. It might even be possible that this caused the unexpected behaviour of XU4 boards reported in Section 13.2.1.

14.3. Summary

This section has shown that the proposed modelling approach fulfils the expectations of Contributions 1 through 5 – at least for time predictions. Given the amount of not attributable influence, energy predictions retain some usefulness: prediction errors are similar to observed execution variations, and the worst observed error was about 20 %. Nevertheless, evaluation of this aspect had to remain inconclusive.

Ambient temperature seems to be much more significant than earlier experiments indicated. Further evaluation would either require heat modelling (which is out of scope for this thesis) or a temperature-controlled environment (which was not available

[2] In fact, this benchmark led to the discovery of these problems – `Ch-40` is the only benchmark with significant C1+ activity.

[3] the platform was operated remotely

Table 14.6.: Results of the second benchmark run, including relative difference to energy measurements of the first benchmark run.

Name		t in s	E_{dyn} in kJ	E_{total} in kJ	P_{dyn} in W	P_{total} in W
Ch-10	predicted	4,127	21.19	114.73	5.13	27.80
	measured	4,134	20.01	109.55	4.84	26.50
	relative	-0.17 %	+5.90 %	+4.73 %	+6.08 %	+4.90 %
	rel. to prev.	-0.29 %	+6.36 %	-1.51 %	+6.67 %	-1.22 %
Ch-20	predicted	1,249	9.25	37.56	7.41	30.07
	measured	1,275	9.72	37.94	7.63	29.76
	relative	-2.04 %	-4.84 %	-0.99 %	-2.85 %	+1.07 %
	rel. to prev.	-2.82 %	+5.66 %	-2.91 %	+8.73 %	-0.09 %
Ch-40	predicted	663	4.17	19.20	6.29	28.96
	measured	646	5.18	19.56	8.01	30.28
	relative	+2.54 %	-19.41 %	-1.86 %	-21.47 %	-4.37 %
	rel. to prev.	-3.58 %	+9.80 %	-2.79 %	+13.88 %	+0.82 %
Ch-80	predicted	612	4.56	18.43	7.45	30.11
	measured	608	4.39	17.97	7.23	29.56
	relative	+0.65 %	+3.69 %	+2.54 %	+3.03 %	+1.87 %
	rel. to prev.	0.00 %	+10.76 %	+0.35 %	+10.76 %	+0.35 %

Table 14.7.: Difference between both sets of energy measurements by individual board. Absolute values are given in Joules.

Board	Ch-10		Ch-20		Ch-40		Ch-80	
	abs.	rel.	abs.	rel.	abs.	rel.	abs.	rel.
ATB-1	-128	-40.4 %	42	10.6 %	15	7.8 %	17	7.1 %
ATB -2	-33	-39.3 %	13	4.6 %	3	1.5 %	14	5.9 %
ATB-3	0	0.0 %	5	2.6 %	5	2.4 %	15	6.3 %
ATB-4	1	50.0 %	4	5.1 %	16	7.0 %	65	31.7 %
ATB-5	0	0.0 %	0	0.0 %	18	8.5 %	15	6.0 %
XU4-1	190	3.2 %	117	6.2 %	117	15.3 %	89	15.3 %
XU4-2	176	4.4 %	-78	-4.7 %	44	5.9 %	56	10.5 %
XU4-3	414	12.4 %	168	10.7 %	79	11.0 %	47	8.3 %
XU4-4	311	11.7 %	152	9.8 %	84	12.2 %	52	9.5 %
XU4-5	267	11.1 %	98	6.2 %	82	11.0 %	57	10.0 %
Sum	1197	6.4 %	521	5.7 %	462	9.8 %	427	10.8 %

Table 14.8.: Results of an early benchmark on an earlier variant of the evaluation platform, leading to the discovery of irregularities related to C1+ boards. Energy figures exclude idle power.

Name	Predicted		Measured		rel. Difference	
	t in s	E in J	t in s	E in J	t	E
Ch-10	4,171	27,153	4,146	25,580	+0.6 %	+6.1 %
Ch-20	1,335	16,914	1,330	16,425	+0.4 %	+3.0 %
Ch-40	543	7,802	582	13,933	-6.7 %	-44.0 %

for this thesis). Earlier experiments with promising results ran under a more uniform environment, but that was more by chance than systematically. This result is consistent with observations about other platforms – power models using Intel RAPL as input encountered similar problems on some CPU architectures [51].

This could have consequences for absolute predictions that include static power, e. g. when planning operating cost or power provisioning. With the observed multitude of hardware effects across board types, different boards of the same type, and environment conditions, users cannot reasonably expect accurate absolute predictions. However, the primary target for this thesis are classic HPC systems. These usually run in a fully temperature-controlled environment, so the impact should be low enough. Assumption 8 must be interpreted this way.

The results also mean that the systems that users use for characterisation need to be temperature controlled the same way as the production system. In a big cluster setting, one approach would be that a small number of nodes could be reserved for characterisation, while the rest operates normally. The advantage over trace-based prediction methodologies is that a single node per architecture is sufficient for characterisation, while trace recording needs a much bigger cluster subset.

When looking beyond traditional HPC platforms, an final aspect of these results is that passively cooled platforms like emerging embedded compute platforms might indeed require heat modelling. A development cluster with integrated power measurement as proposed in this thesis could even be adapted for exploring different heat scenarios.

15. Evaluation of Individual Design Decisions

This final evaluation section discusses additional experiments I considered or performed for the evaluation goals described in 11.4. While the previous sections follow the proposed design flow just like users would apply it, the experiments in this section do not belong to the design flow itself.

15.1. Fixed Clock Frequencies

I intended to evaluate how fixed clock frequencies affect the performance of applications. My expectation is that this would not show a significant difference, since I already tried to find the highest sustainable clock speed using the procedure shown in Section 10.4.

For this test, I used the benchmark of Section 14.2.2, but configured the platform for maximum performance, i. e. default Linux CPU frequency scaling using the 'performance' governor algorithm.

Unfortunately, the test did not succeed: a few seconds into the benchmarks, some boards spontaneously rebooted. This happened multiple times with different boards, so I aborted the test. Since I did not encounter this behaviour otherwise, I assume this happened as a thermal protection measure due to the heat caused by higher system clock frequencies.

As a result, I can't quantify how much fixed clock frequencies affect execution performance. For a fully utilised system, the effect should be small, since during platform characterisation, the maximum clock frequency for exactly that load situation is selected.

Another observation supports the assumption that variable clock speed would not improve system performance much, if at all: In [45], the authors examine the Samsung Exynos 5422 system-on-chip on the ODROID XU3 board, which is for all practical matters identical to the XU4 boards used in this thesis. The computational speed measurement they report hit their maximum roughly around the clock frequency initially selected in this thesis.

In any case, the long term stability issues I encountered forced me to use a much slower clock frequency for the XU4 boards (see Section 13.2), so the selected hardware is not suitable for such a comparison anyways. It is probably safe to assume that system performance for the final platform configuration is not on par with an unrestrained system. Had the used hardware had better long term behaviour, there would be a chance that fixed frequencies do not harm overall throughput.

15.2. External Time Measurement

Initially, I had planned to use CPU timers for time measurement, without an external time measurement board. However, I encountered severe difficulties when trying to match CPU time values to energy trace positions. Especially the ODROID C1+ boards consistently failed to deliver stable results when running under load. In an attempt to find the source of the error, I introduced the measurement board described in Section 10.3.2.

The remainder of this subsection describes the insights I gained by introducing external time measurement.

15.2.1. Reproduction of Inconsistent Behaviour

In order to find out more about the inconsistent behaviour I saw, I performed a similar set of benchmarks as done for the measurement platform in Section 11.2.1, with one difference: instead of keeping the compute boards idle, I used a workload designed to maximise power consumption, similar to the one described in Section 10.4. The additional heat might introduce clock drift, but external time measurement will be unaffected by this.

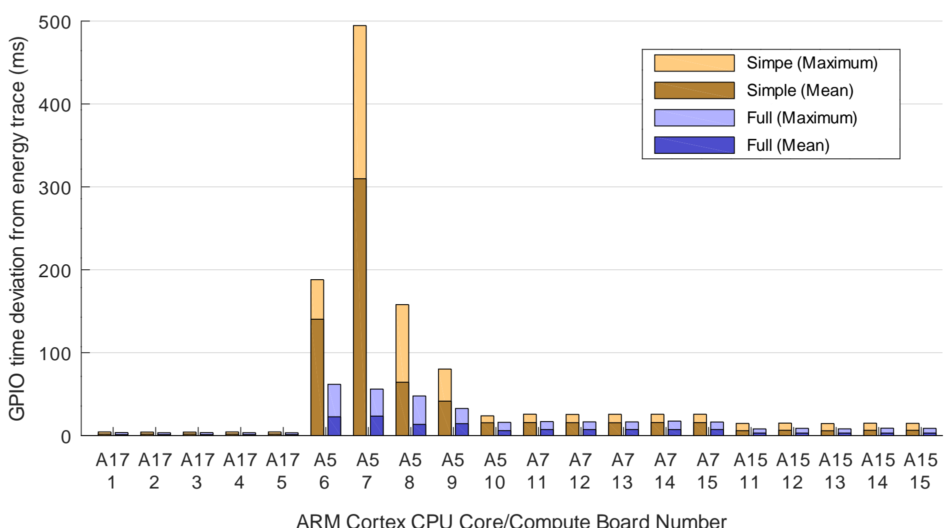
I then examined the resulting measurements in a variety of ways in order to gain more insight.

15.2.2. Results

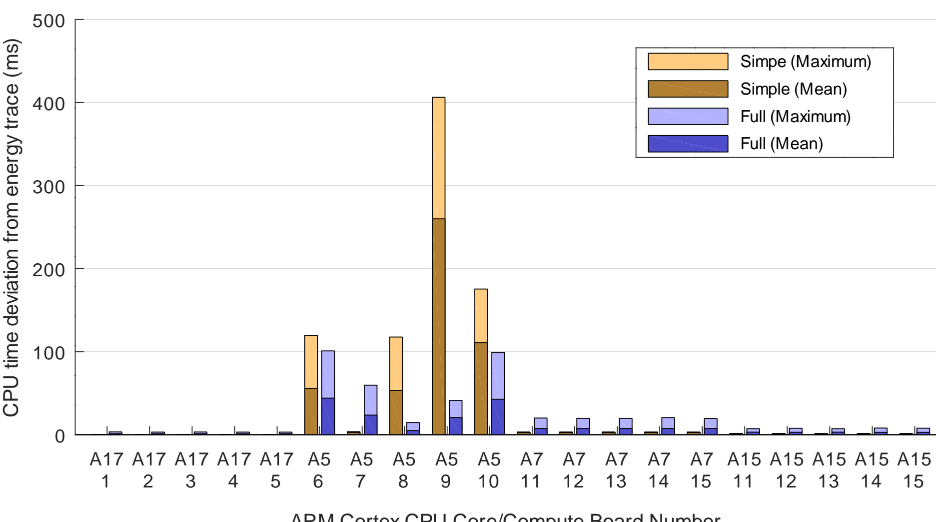
Figure 15.1 shows the absolute difference between energy trace timings and both, CPU and GPIO timings. For the ASUS TinkerBoard and the ODROID XU4, they don't show anything unusual, but the ODROID C1+ (compute boards 6 to 10) is much worse than in the idle case, with no clear pattern.

In fact, when looking at the energy traces of channels 6 to 10, the time error is so large that CPU/GPIO timings could not be matched with energy events anymore. Figure 15.2 shows an example from channel 7 where the difference between the expected time of the `WAIT` kernel and its occurrence in the energy trace was so large that it did not get matched for the analysis shown above – the algorithm that does the matching ignores events that are too far apart, because otherwise CPU/GPIO events might get matched with wrong energy events. In the example of Figure 15.2, the leading edge of the following `MARK` kernel is almost close enough to be confused as the leading edge of `WAIT`.

The net result is that the true error for the ODROID C1+ boards is even larger than shown in Figure 15.1. In order to eliminate energy trace event detection as source of inaccuracy, I finally compared GPIO times directly to CPU times. Figure 15.3 shows the result, comparing idle and busy systems.



(a) GPIO and energy trace timings



(b) CPU timer and energy trace timings.

Figure 15.1.: Absolute deviation between different timings using two different approximations: ‘simple’ uses a linear correction factor calculated from trace end markers, while ‘full’ is a linear regression over all task execution events.

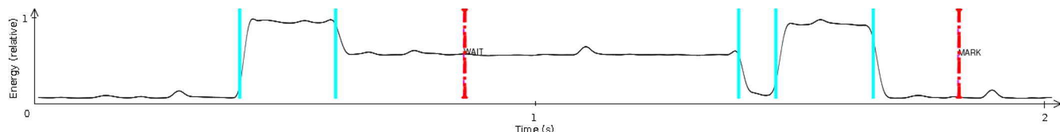


Figure 15.2.: Excerpt of an energy trace showing two seconds of channel 7. Cyan bars mark algorithmically detected edges in the energy trace, red bars mark GPIO and CPU timings (which coincide in this example).

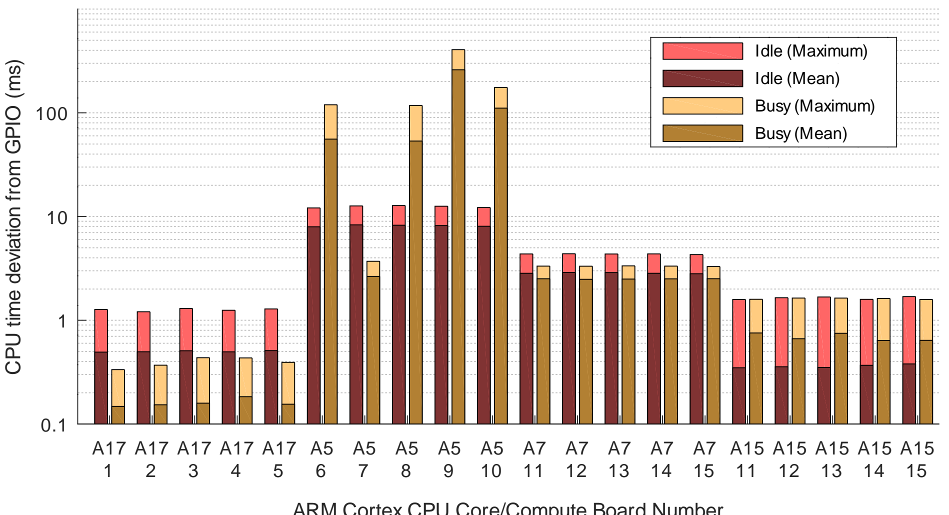


Figure 15.3.: Absolute deviation between GPIO and CPU timings using the simple correction factor, for an unloaded system as in Figure 12.6 and for a loaded system like in Figure 15.1.

15.2.3. Discussion

It is obvious that there is a problem with the ODROID C1+ boards. When testing them when mostly idling, there are no surprising results. Timing error is a bit higher than for the other boards (see Figure 12.6), but not by much, and it is uniform across all boards.

Under load however, board behaviour becomes unpredictable. Sometimes CPU timers are much worse than GPIO timing, but for some boards the situation is reversed. Furthermore, the simple linear correction factor seems to be unusably inaccurate – except for the one board where it is excellent.

This is a behaviour that doesn't match anything seen during this thesis, and the direct comparison between GPIO and CPU timings strongly suggests that the timer values themselves have weird behaviour. Given that there is a single GPIO time measurement board, and that it behaved consistently throughout this thesis, the source must be the local timers on the ODROID C1+ boards.

An internet search brought me to a discussion on the armbian (Debian on ARM) forum that suggests that the direct successor of the SoC used in the ODROID C1+ (Amlogic S805) doesn't honour the DVFS settings made by the Linux kernel [37]; real CPU speed would vary without the Linux kernel ever knowing.

A hidden firmware which switches clock frequencies under the hood can very well affect timer values – it would have to compensate for missed timer interrupts, which might lead to temporary fluctuations of locally perceived time. That would match the behaviour observed here: globally, CPU time seems to be within the expected accuracy, only local variations are unusual. There is no easy way to test this hypothesis, however, and it doesn't add insights for the goals of this thesis.

The effect of local time variation can be severe, however. When characterisation fails to identify the correct time span of a kernel in the power trace, the resource model will be wrong. If the sampling window for a kernel shifts into the leading 2 s pause, it will cover more idle time due to the pauses inserted around characterised kernels. The results shown in Table 14.8 are consistent with this: predictions for `Ch-40` were much too low.

Before introducing an overall synchronous time base, I was unable to analyse this behaviour at all. I regularly saw inconsistencies, but didn't have a way to identify a root cause, let alone to quantify the error.

External time measurement through GPIO signals made this problem clear, and I could finally distinguish the erratic behaviour of C1+ boards from heat-related issues of XU4 boards. The latter led to the clock characterisation process shown in Section 10.4.1.

Other parts of the methodology benefit as well, as there is less effort required to determine correct timings. The main disadvantage is that the target hardware must provide a low-latency GPIO signal.

15.3. Custom Network Protocol

I decided to develop the custom Eth network protocol (Section 9.8.4) instead of using TCP. I already evaluated the general performance of the Eth networking layer in Section 13.

Table 15.1.: Comparison of TCP networking and Eth network protocol bandwidth. Columns specify maximum, mean, and standard deviation for a given pair of communication partners.

Source	Destination	TCP MByte/s			Eth MByte/s		
		Max	Mean	SD	Max	Mean	SD
ATB	ATB	117.6	73.6	28.1	29.8	26.7	5.9
ATB	C1+	72.3	45.2	24.0	13.6	12.2	3.32
ATB	XU4	40.1	27.3	11.4	6.8	6.0	1.69
C1+	C1+	51.3	37.9	9.2	13.6	13.0	1.07
C1+	XU4	41.4	30.1	11.0	6.8	6.6	0.47
C1+	ATB	51.0	45.8	6.3	18.7	17.7	1.87
XU4	XU4	24.3	18.7	4.8	6.8	6.6	0.40
XU4	ATB	24.3	21.7	3.2	7.9	7.7	0.38
XU4	C1+	24.3	21.2	3.2	7.9	7.7	0.37

In this section I show the observations that led to that decision and evaluate the impact of the Eth networking layer.

I designed the physical execution runtime so that I could easily swap the networking backend. In order to compare the TCP and Eth protocols, I simply run the same benchmark as described in Section 13.1.2, once with the TCP backend and once with the Eth backend.

The difference is that I don't measure deviation from the expected or ideal transmission time. With TCP, unlike Eth, I do not directly control transmission timing. Instead, I look at the effective bandwidth of each individual benchmark run, measured in Megabyte per second. Table 15.1 shows the results of this set of benchmarks.

Two major facts are easy to see: TCP networking has a higher maximum bandwidth, up to eight times the value of Eth. This is not too surprising, as Eth uses uncommon and thus less optimised network APIs with no hardware acceleration support, and it is CPU intensive by itself.

When comparing mean bandwidth, this difference is only half as big, and this is also visible in the standard deviation. This variability made TCP network connections difficult to predict. I tried to slow down TCP connections by inserting artificial pauses, but no amount of slowdown would reduce this variability to an acceptable degree.

On the other hand, Eth networking has much more predictable behaviour. The general trend established in Section 13 is visible in this benchmark as well. C1+ and XU4 boards have excellent results and ATB boards slightly less so.

When comparing mean bandwidth of TCP and Eth, C1+ and XU4 performance differ-

ence is about a factor of two. ATB results differ by a factor of roughly three. This is the origin of the claim that Eth is not meant as a production-ready protocol (Section 9.8.4), although it is in a similar order of magnitude.

As has already been discussed in Section 13, the overall bandwidth of Eth meets the design goals of the evaluation platform, so its introduction is reasonable and offers a significant improvement in predictability. Since more expensive Ethernet interfaces already support Ethernet TSN, it is reasonable to assume that HPC networking equipment has or will have hardware support for a similar level of predictability.

15.4. Custom Runtime System

Several times during this thesis, I trade performance for predictability. For this reason, Contribution 3 states that the physical execution runtime as presented in see Section 9.8 does not have the goal to be as fast or faster than established runtimes. But the results of this thesis should not put users at such a large disadvantage that the performance loss would make predictions correct but useless. In other words, the physical execution runtime should at least show a credible potential to be competitive in real world settings. This is expressed by the claim that the performance of the physical execution runtime is at the same order of magnitude as established runtime systems.

The physical execution runtime of Section 9.8 has not been optimised for performance or energy. This means that if a task graph can execute in roughly the same time as an established (presumably optimised) runtime system executes the same application using a similar parallelisation strategy, it is reasonable to assume that the presented execution runtime could be optimised to be competitive. It also means that the presented methodology applies to at least some real-world settings.

15.4.1. Discussion

While there are MPI implementations of Cholesky matrix decomposition, they assume a fully homogeneous platform. The commonly used algorithm uses global synchronisation[1], so the slowest board would force all other boards to wait. Since one of the distinguishing points of the proposed methodology is the applicability to highly heterogeneous platforms, such a comparison would be of limited use.

Disregarding this aspect, a homogeneous subset consisting of the five ATB boards could be used for a comparison. But even then, it is more than likely that this would primarily benchmark the difference in algorithmic structure. In a non-distributed multi-core setting, the performance difference of task graph based applications has been reported to be between a 6 % slowdown and a 42 % speedup when compared to implementations using more traditional parallelisation strategies [16]. This is a nontrivial influence on application performance, which means that such a comparison would not really show the overhead of the runtime itself.

[1] In fact, the high likelihood of `TRSM`/`TRSM` seen in Figure 14.1 concurrency is due to a natural synchronisation behaviour of the algorithm itself.

On the other hand, evaluation of the overall methodology in Section 14 shows no sign of significant overhead. Simulation models do not include any time modelling of the execution runtime itself, yet time prediction accuracy is good. Measurements show no unexpected pauses between task executions during physical execution: the time span between end of a task and start of a following, immediately runnable task is usually less than 0.1 ms, and the visualisation in Figure 14.2 is consistent with this observation.

So while an actual evaluation of this aspect is difficult, it can safely be deduced from Section 14 that the performance claim of Contribution 3 is fulfilled.

Part V.

Conclusion

Conclusion

In this thesis I have presented a fully integrated, novel design flow for prediction-based optimisation of high-performance computing (HPC) application behaviour. It is based on familiar concepts from embedded system design, but changes critical aspects in order to cope with the huge difference in scale between embedded and HPC systems.

The methodology uses abstract models to represent applications, hardware, and resource usage of applications. It also explicitly models application semantics. As a result, an abstract simulator can use these models to predict execution time and energy of a given application on a given platform without executing application code.

I have evaluated the methodology using Cholesky matrix decomposition, a linear algebra algorithm that exhibits properties of several important classes of HPC applications.

Performance of the simulation that generates time and energy predictions is fast enough for interactive workflows and design space exploration. It scales linearly with the number of tasks in the application model.

The error of time predictions is below 5%, which is on par with the best prediction methods from the HPC world.

Unfortunately, energy predictions raised questions about the circumstances under which energy predictions are valid. The result of this thesis is that the target platform must run in a temperature-controlled environment, which limits applicability for emerging embedded compute platforms that might be passively cooled.

In a setting with varying ambient conditions, I have observed prediction errors of up to 20 % for dynamic energy consumption and up to 5 % when including idle power. This means that predictions could be useful in overprovisioning scenarios, although they do not outperform existing trace-based methods.

In this regard, the main advantage of the proposed methodology is that it does not need final applications nor traces from a full cluster, so that predictions are available much earlier and can be updated more often. This allows user to shift focus from optimising platform management to optimising applications for energy.

As a minor contribution, I have shown a scaled-down version of an HPC cluster system. The decision to use consumer grade off-the-shelf hardware did lead to complications, and the evaluation offers valuable insight for selection of reliable compute nodes. After accounting for several adverse effects, the platform worked within its design goals.

I have also presented an inexpensive, yet powerful way of obtaining synchronised time and energy measurements of up to ten consumers with high quality. Accuracy of the measurement platform is fully in line with the main contributions: energy measurements have less than 1% error over most of the measurement range. Time measurement error is less than 0.1 %. While it was optimised for the evaluation platform, the measurement board also supports usage with HPC-grade CPUs.

Future Work

Most importantly, the relation between ambient conditions and energy behaviour of off-the-shelf hardware needs further investigation. A temperature-controlled evaluation platform is required to evaluate how the proposed methodology would behave under ideal circumstances.

Apart from this, there are other interesting topics that emerge from the modelling and simulation approach:

Power over Time To fully support use cases involving power provisioning, the simulation model would have to be extended to produce power traces as mentioned in Section 9.7.2. This is actually not too difficult: For a simple approximation, PEs can calculate momentary power by distributing the energy of the resource model over the task run time (including the dynamically changing slowdown situation). This ignores that tasks can have varying power over time themselves. That power profile is captured during characterisation and integrated over time to get task energy, but the entire profile could also be stored in the resource model. PEs could then replay it.

Memory The observation that the evaluation platform cannot reliably execute Cholesky decomposition of a 20480×20480 matrix (see Section 14) is actually another interesting question: Will the application be able to complete under the memory constraints of the platform?

This is more important than with traditional HPC applications, because those distribute work and data in a very regular manner. In contrast, task graphs as shown in this thesis lead to dynamic data distribution. While this can reduce overall memory consumption (as it does for Cholesky), it creates the question of maximum required memory size.

There are formal approaches to this, but they might struggle with HPC scale applications. An extended execution model may even be able to prevent failures by delaying data reception (i. e. memory allocation) or prioritising transmissions (i. e. memory deallocation). Accurate simulation could predict this with a suitably specified execution model.

Profiling Metrics A simulation-based approach offers the advantage that it can easily be extended to provide additional metrics to guide optimisation. Memory usage could be one metric. Another useful metric for developers would be the number of pending communication requests across the platform, which can highlight bottlenecks in algorithm or network structure.

Accelerators One aspect of the initial problem statement has not been covered in depth. While I did show heterogeneity in principle, I did not explore GPGPUs or FPGAs. Due to the high level of abstraction, the methodology is prepared for unconventional compute elements. The execution runtime could simply contain the ability to execute OpenCL code or FPGA accelerator functions; GPUs and FPGAs would be modelled as further processing elements, and OpenCL code and bitstream would function as kernel code. In a similar vein, fixed function units like video encoders could be represented as PEs that only support a limited set of kernels. Using the task graph approach for characterisation, this might work in a straightforward way, but this needs evaluation.

Multiple PE Allocation A related question concerns multi-threaded kernels. A highly optimised kernel might use multiple cores more efficiently than running two unrelated tasks on them.

In general, a parallel execution unit like a CPU, GPU, or FPGA can be represented as multiple independent PEs or as one big PE with internal parallelism, or as a combination of these. This would increase the search space for automatic mapping algorithms, but might give developers additional optimisation opportunities.

Part VI.

Appendix

References

[5] International Electrotechnical Commission, "IEC 61499-1: Function blocks – Part 1: Architecture," Nov. 2012. 4.2.3.2

[6] E. Perelman, G. Hamerly, M. V. Biesbrouck, T. Sherwood, and B. Calder, "Using SimPoint for accurate and efficient simulation." in *SIGMETRICS*. ACM, 2003, pp. 318–319. 8.1.1

[7] IEEE Computer Society, "IEEE Standard for Standard SystemC Language Reference Manual," *IEEE Std 1666-2011*, 2012. 4.2.3.2, 6, 6, 8.1.1

[8] O. Almer, I. Böhm, T. J. K. E. von Koch, B. Franke, S. C. Kyle, V. Seeker, C. Thompson, and N. P. Topham, "A Parallel Dynamic Binary Translator for Efficient Multi-Core Simulation." *International Journal of Parallel Programming*, vol. 41, no. 2, pp. 212–235, 2013. 8.1.1

[9] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe, "Statistical sampling of microarchitecture simulation," *ACM Trans. Model. Comput. Simul.*, vol. 16, no. 3, pp. 197–224, Jul. 2006. 8.1.1

[10] V. Adve and R. Sakellariou, "Application representations for multiparadigm performance modeling of large-scale parallel scientific codes," *International Journal of High Performance Computing Applications*, vol. 14, no. 4, pp. 304–316, 2000. 4.2.3.1, 9.2.1

[11] P. M. Kogge and T. J. Dysart, "Using the top500 to trace and project technology and architecture trends," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2011, p. 28. 4.1.3

[12] K. Keutzer, B. L. Massingill, T. G. Mattson, and B. A. Sanders, "A design pattern language for engineering (parallel) software: merging the PLPP and OPL projects," in *Proceedings of the 2010 Workshop on Parallel Programming Patterns*. New York, NY, USA: ACM, 2010, pp. 9:1–9:8. 4.2.1, 4.2.3

[13] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Herault, and J. J. Dongarra, "PaRSEC: Exploiting Heterogeneity to Enhance Scalability," *Computing in Science & Engineering*, vol. 15, no. 6, pp. 36–45, 2013. 9.2.3, 9.2.4

[14] R. Griessl, M. Peykanu, J. Hagemeyer, M. Porrmann, S. Krupop, M. vor dem Berge *et al.*, "A Scalable Server Architecture for Next-Generation Heterogeneous Compute

Clusters," in *Proceedings of the 12th IEEE International Conference on Embedded and Ubiquitous Computing*, 2014. 4.1.2, 4.2.5, 5.3.1

[15] T. Dwyer, A. Fedorova, S. Blagodurov, M. Roth, F. Gaud, and J. Pei, "A Practical Method for Estimating Performance Degradation on Multicore Processors, and Its Application to HPC Workloads," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012, pp. 83:1–83:11. 9.5.3

[16] D. Chasapis, M. Casas, M. Moretó, R. Vidal, E. Ayguadé, J. Labarta, and M. Valero, "PARSECSs: Evaluating the Impact of Task Parallelism in the PARSEC Benchmark Suite." *TACO*, vol. 12, no. 4, p. 41, 2016. 4.2.2, 4.4, 4.5.1, 9.2.4, 15.4.1

[17] G. Geist, J. A. Kohl, and P. M. Papadopoulos, "PVM and MPI: A comparison of features," *Calculateurs Paralleles*, vol. 8, no. 2, pp. 137–150, 1996. 4.3.2

[18] J. Spolsky, *The Law of Leaky Abstractions*. Berkeley, CA: Apress, 2004, pp. 197–202. 4.3.2

[19] L. Dagum and R. Menon, "OpenMP: An industry-standard API for shared-memory programming," *Computing in Science & Engineering*, no. 1, pp. 46–55, 1998. 4.3.1

[20] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas, "Ompss: a proposal for programming heterogeneous multi-core architectures," *Parallel processing letters*, vol. 21, no. 02, pp. 173–193, 2011. 4.3.1

[21] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures," in *Euro-Par 2009*, Delft, Netherlands, Aug. 2009. 4.3.1

[22] A. Auweter, A. Bode, M. Brehm, L. Brochard, N. Hammer, H. Huber, R. Panda, F. Thomas, and T. Wilde, "A Case Study of Energy Aware Scheduling on SuperMUC," in *Supercomputing*, J. M. Kunkel, T. Ludwig, and H. W. Meuer, Eds. Cham: Springer International Publishing, 2014, pp. 394–409. 4.1.4

[23] K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, K. Hill, J. Hiller *et al.*, "Exascale computing study: Technology challenges in achieving exascale systems," *Defense Advanced Research Projects Agency Information Processing Techniques Office (DARPA IPTO), Tech. Rep*, vol. 15, 2008. 4.1.3, 4.1.4

[24] M. Pospieszny, "Electricity in HPC Centres," Partnership for Advanced Computing in Europe (PRACE) Project, Tech. Rep., 2014. 4.1.3

[25] J. J. Dongarra, P. Luszczek, and A. Petitet, "The LINPACK benchmark: past, present and future," *Concurrency and Computation: practice and experience*, vol. 15, no. 9, pp. 803–820, 2003. 4.4

[26] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: Characterization and architectural implications," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. ACM, 2008, pp. 72–81. 4.4

[27] C. Bienia, S. Kumar, and K. Li, "PARSEC vs. SPLASH-2: A quantitative comparison of two multithreaded benchmark suites on chip-multiprocessors," in *2008 IEEE International Symposium on Workload Characterization*. IEEE, 2008, pp. 47–56. 4.4

[28] K. Singh, M. Bhadauria, and S. A. McKee, "Real time power estimation and thread scheduling via performance counters." *SIGARCH Computer Architecture News*, vol. 37, no. 2, pp. 46–55, 2009. 5.3.2.1

[29] F. Kesel, *Modellierung von digitalen Systemen mit SystemC: von der RTL-zur Transaction-Level-Modellierung*. Walter de Gruyter, 2012. 6

[30] D. D. Gajski, S. Abdi, A. Gerstlauer, and G. Schirner, *Embedded system design: modeling, synthesis and verification*. Springer Science & Business Media, 2009. 6

[31] K. Grüttner, A. Herrholz, P. A. Hartmann, A. Schallenberg, and C. Brunzema, *OSSS – A Library for Synthesisable System Level Models in SystemC*, 2008. 6

[32] P. A. Hartmann, K. Grüttner, and W. Nebel, "Advanced SystemC Tracing and Analysis Framework for Extra-Functional Properties." in *ARC*, K. Sano, D. Soudris, M. Hübner, and P. C. Diniz, Eds., vol. 9040. Springer, 2015, pp. 141–152. 6.3, 6.4

[33] International Organization for Standardization, "ISO/IEC 7498-4:1989 – Information technology – Open Systems Interconnection – Basic Reference Model: Naming and addressing," International Standards Organisation, Tech. Rep., 1989. 6.3.2

[34] A. Varga and R. Hornig, "An overview of the OMNeT++ simulation environment," in *Proceedings of the 1st international conference on Simulation tools and techniques for communications, networks and systems & workshops*. ICST (Institute for Computer Sciences, Social-Informatics and …, 2008, p. 60. 6.3.2

[35] S. Kerrison, "Monitoring the energy consumption of a Raspberry Pi with a MAGEEC Wand," The MAGEEC Project, Tech. Rep., 08 2016. 8.2

[36] A. McCormick, N. Johnson, D. Dolman, C. Holyoake, S. Kaxiras, and V. Spiliopoulos, "D3.2 – Power Usage of Hardware Platforms," The ADEPT Project, Tech. Rep., Mar. 2015. 8.2

[37] Various pseudonymous authors, "Amlogic still cheating with clockspeeds," Apr. 2018, accessed: 2019-12-17. [Online]. Available: https://web.archive.org/web/20191217200444/https://forum.armbian.com/topic/7042-amlogic-still-cheating-with-clockspeeds 15.2.3

[38] K. Grüttner, P. A. Hartmann, K. Hylla, S. Rosinger, W. Nebel, F. Herrera, E. Villar, C. Brandolese, W. Fornaciari, G. Palermo *et al.*, "The COMPLEX reference framework for HW/SW co-design and power management supporting platform-based design-space exploration," *Microprocessors and Microsystems*, vol. 37, no. 8, pp. 966–980, 2013. 8.1.1

[39] D. L. Mills, "Internet time synchronization: the network time protocol," *IEEE Transactions on communications*, vol. 39, no. 10, pp. 1482–1493, 1991. 4.1.1

[40] M. Kierzynka, L. Kosmann, M. vor dem Berge, S. Krupop, J. Hagemeyer, R. Griessl, M. Peykanu, and A. Oleksiak, "Energy efficiency of sequence alignment tools - Software and hardware perspectives," *Future Generation Computer Systems*, pp. –, 5 2016. 4.1.2

[41] M. Harchol-Balter, "Job placement with unknown duration and no preemption." *SIGMETRICS Performance Evaluation Review*, vol. 28, no. 4, pp. 3–5, 2001. 4.2.4.1

[42] M. Willebeek-LeMair and A. P. Reeves, "Strategies for Dynamic Load Balancing on Highly Parallel Computers." *IEEE Trans. Parallel Distrib. Syst.*, vol. 4, no. 9, pp. 979–993, 1993. 4.2.4.1

[43] J. R. Vig, "Introduction to Quartz Frequency Standards," Army Research Laboratory, Electronics and Power Sources Directorate, Tech. Rep. SLCET-TR-92-1, 1992. 11.2.1

[44] Atos SE, "Atos' BullSequana supercomputer powered by AMD processor sets world-record performance," Sep. 2019. [Online]. Available: https://web.archive.org/web/20200711032057/https://atos.net/en/2019/press-release_2019_09_18 13.3.3

[45] A. Butko, A. Gamatié, G. Sassatelli, L. Torres, and M. Robert, "Design Exploration for next Generation High-Performance Manycore On-chip Systems: Application to big.LITTLE Architectures." in *ISVLSI*. IEEE Computer Society, 2015, pp. 551–556. 13.3.1, 13.3.3, 15.1

[46] S. Rosinger, M. Metzdorf, D. Helms, and W. Nebel, "Behavioral-Level Thermal- and Aging-Estimation Flow," in *Test Workshop (LATW), 2011 12th Latin American*, 3 2011, pp. 1–6. 4.1.5.3

[47] C. Emde, "Man nehme: ARM oder x86?" in *Embedded Software Engineering Kongress 2014*, 2014. 13.3.1, 13.3.5

[48] T. Patki, D. K. Lowenthal, A. Sasidharan, M. Maiterth, B. Rountree, M. Schulz, and B. R. de Supinski, "Practical Resource Management in Power-Constrained, High Performance Computing." in *HPDC*, T. Kielmann, D. Hildebrand, and M. Taufer, Eds. ACM, 2015, pp. 121–132. 4.1.4, 5.3.2.1, 8.1, 8.1.2, 11.1.2.2, 11.3

[49] T. P. D. Lowenthal, B. Rountree, M. Schulz, and B. de Supinski, "Exploring Hardware Overprovisioning in Power-Constrained, High Performance Computing." 4.1.4

[50] T. Patki, D. K. Lowenthal, B. L. Rountree, M. Schulz, and B. R. de Supinski, "Economic Viability of Hardware Overprovisioning in Power-Constrained High Performance Computing." in *E2SC@SC*. IEEE Computer Society, 2016, pp. 8–15. 4.1.4

[51] K. N. Khan, M. Hirki, T. Niemi, J. K. Nurminen, and Z. Ou, "Rapl in action: Experiences in using rapl for power measurements," *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS)*, vol. 3, no. 2, pp. 1–26, 2018. 5.3.2.1, 8.2, 14.3

[52] S. Wallace, V. Vishwanath, S. Coghlan, Z. Lan, and M. E. Papka, "Measuring power consumption on IBM Blue Gene/Q," in *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*. IEEE, 2013, pp. 853–859. 11.3

[53] T. L. Casavant and J. G. Kuhl, "A taxonomy of scheduling in general-purpose distributed computing systems," *Software Engineering, IEEE Transactions on*, vol. 14, no. 2, pp. 141–154, 1988. 4.2.3, 4.2.4

[54] R. D. Blumofe and D. S. Park, "Scheduling large-scale parallel computations on networks of workstations," in *Proceedings of 3rd IEEE International Symposium on High Performance Distributed Computing*. IEEE, 1994, pp. 96–105. 4.2.4.1

[55] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," *Journal of the ACM (JACM)*, vol. 46, no. 5, pp. 720–748, 1999. 4.2.4.1, 4.2.5

[56] G. P. Pezzi, M. C. Cera, E. N. Mathias, N. Maillard, and P. O. A. Navaux, "On-line Scheduling of MPI-2 Programs with Hierarchical Work Stealing." in *SBAC-PAD*. IEEE Computer Society, 2007, pp. 247–254. 4.2.4.1

[57] J. A. Pascual, J. Navaridas, and J. Miguel-Alonso, "Effects of topology-aware allocation policies on scheduling performance," in *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer, 2009, pp. 138–156. 4.3.3

[58] M. M. Tikir, M. Laurenzano, L. Carrington, and A. Snavely, "PSINS: An Open Source Event Tracer and Execution Simulator for MPI Applications." in *Euro-Par*, H. J. Sips, D. H. J. Epema, and H.-X. Lin, Eds., vol. 5704. Springer, 2009, pp. 135–148. 4.5.4, 8.1, 8.1.2, 11.1

[59] A. Rico, F. Cabarcas, C. Villavieja, M. Pavlovic, A. Vega, Y. Etsion, A. Ramírez, and M. Valero, "On the simulation of large-scale architectures using multiple application abstraction levels." *TACO*, vol. 8, no. 4, p. 36, 2012. 8.1, 8.1.2, 11.1

[60] C. L. Janssen, H. Adalsteinsson, and J. P. Kenny, "Using simulation to design extremescale applications and architectures: programming model exploration." *SIGMETRICS Performance Evaluation Review*, vol. 38, no. 4, pp. 4–8, 2011. 4.5.4, 8.1.2

[61] K. Wang and I. Raicu, "SimMatrix: SIMulator for MAny-Task computing execution fabRIc at eXascales," in *High Performance Computing Symposia (HPC)*, Apr. 2013. 4.2.4.1, 8.1, 8.1.3, 11.1

[62] N. Lopez-Benitez and J.-Y. Hyon, "Simulation of task graph systems in heterogeneous computing environments," in *Proceedings of the 8th Heterogeneous Computing Workshop, 1999*. IEEE, 1999, pp. 112–124. 8.1, 8.1.4, 11.1

[63] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner, "Simics: A Full System Simulation Platform." *IEEE Computer*, vol. 35, no. 2, pp. 50–58, 2002. 8.1.1

[64] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The Gem5 Simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011. 8.1.1, 8.1

[65] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith, "A mechanistic performance model for superscalar out-of-order processors." *ACM Trans. Comput. Syst.*, vol. 27, no. 2, 2009. 8.1.4, 9.5.2

[66] J. Labarta, S. Girona, and T. Cortes, "Analyzing Scheduling Policies Using Dimemas." *Parallel Comput.*, vol. 23, no. 1-2, pp. 23–34, 1997. 8.1.2

[67] C. Camarero, C. Martínez, and J. L. Bosque, "Simulation with skeletons of applications using dimemas." in *CF*, F. Palumbo, M. Becchi, M. Schulz, and K. Sato, Eds. ACM, 2019, pp. 274–278. 8.1.2

List of Figures

5.1. Basic electrical power measurement circuit. 51

9.1. Assumed development approach for the prediction methodology. 74
9.2. Final development flow including all models. Bold borders indicate elements introduced by this thesis. 74
9.3. Example of a task graph performing Cholesky matrix decomposition with 5×5 matrix subdivision. Colour denotes different kernels. 77
9.4. Execution Runtime Model 83
9.5. Excerpt of an example platform hierarchy graph. 87
9.6. Example of a platform communication graph. 88
9.7. Class diagram of SystemC simulation models (simplified). 97
9.8. Sequence Diagram of a single task execution with communication and concurrent execution on a second PE. Numbers refer to the explanation in the text. 99
9.9. Sequence diagram of a single data set transmission (initial part). Numbers refer to the explanation in the text. 101

10.1. Power distribution path of the power distribution and measurement board. 113
10.2. Physical design of the power distribution and measurement board. 114
10.3. CPLD schematics of the power measurement board. 116
10.4. CPLD schematics: DMA error checking. 118
10.5. Picture of the embedded cluster platform with integrated measurement infrastructure (initial version). Components marked with an asterisk were added/changed later due to significant evaluation challenges. 120
10.6. Power trace of the initial `mark` sequence on five channels (denoted by colour). At 100 ms (1,000 power samples), the marker energy begins. After 10 s (100,000 power samples), the first task executes. 124
10.7. Supply current trace of the `MARK` kernel. The left spike is the marker energy, followed by 2 s pause. The right spike is from the following kernel. 125
10.8. Suggested communication timing characterisation procedure. 128

12.1. DC current measurement: shunt resistor current over ADC sample values, channel 10. 142
12.2. DC current measurement: relative error over ADC sample values, channel 10. 144
12.3. Electrical current of two physically adjacent channels with significant power consumption steps to make cross talk visible, if present. 146

12.4. Frequency response of the external analogue signal path. 147
12.5. Maximum and mean relative deviation between different timings using two different approximations: 'Simple' uses a linear correction factor calculated from trace end markers, while 'Full' is a linear regression over all task execution events. 148
12.6. Absolute deviation between different timings using two different approximations: 'simple' uses a linear correction factor calculated from trace end markers, while 'full' is a linear regression over all task execution events. 149

13.1. Power over time during pauses and active periods. Lines are linear approximations. 157
13.2. Power over time for the five ASUS TinkerBoards. 158
13.3. Eth networking layer timing difference between measured and ideal values (i. e. calculated from the stated packet delays). 160
13.4. Energy trace of a single communication benchmark, showing the receiver of a unidirectional ATB-to-ATB transfer. Communication happens between 10.5 s and 13.8 s. 161
13.5. Communication energy per packet, plotted over packet delay. Colour denotes board type, while shape distinguishes between receive-only, send-only, and bidirectional transfers. Lines are linear approximations. 161

14.1. CPU activity for `Ch-10` as determined by a simulation without secondary time model. Each row of boxes represents one compute core, while the coloured boxes themselves represent tasks, with colours showing the kernel being executed. 170
14.2. CPU activity for `Ch-10`, similar to Figure 14.1 but showing measurement and final simulation side-by-side. Each pair of rows represents one compute core, with measurement result on the top row and simulation result on the bottom row. 174
14.3. CPU activity for `Ch-40`, similar to Figure 14.1 but showing measurement and final simulation side-by-side. Each pair of rows represents one compute core, with measurement result on the top row and simulation result on the bottom row. 175

15.1. Absolute deviation between different timings using two different approximations: 'simple' uses a linear correction factor calculated from trace end markers, while 'full' is a linear regression over all task execution events. 183
15.2. Excerpt of an energy trace showing two seconds of channel 7. Cyan bars mark algorithmically detected edges in the energy trace, red bars mark GPIO and CPU timings (which coincide in this example). 184
15.3. Absolute deviation between GPIO and CPU timings using the simple correction factor, for an unloaded system as in Figure 12.6 and for a loaded system like in Figure 15.1. 184

List of Tables

8.1. Typical characteristics of prediction techniques (represented by prominent examples) and comparison to the proposed methodology 68

9.1. Main API calls of the execution runtime HAL 85
9.2. Eth packet format 107
9.3. Packet types in the Eth protocol 107

11.1. Reported time prediction error in related methodologies. Multiple values are given if more than one simulation technique is evaluated. 134

12.1. Shunt resistor voltages for a nominal current of 1 A 143
12.2. Maximum error of different linear fitting methods for ADC to shunt voltage conversion: linear regressions using 2000 and 2 data points per channel, and the theoretical value of 25 µV per ADC count. 144
12.3. DC voltage measurement results for a single channel. 145

13.1. Correction factors and idle power for the 15 boards. 156
13.2. Final communication packet timing and net bandwidth (BW) for the Eth networking layer. 159

14.1. Evaluated application configurations. 168
14.2. Results from the characterisation procedure. 169
14.3. Slowdown factors of the secondary time model (competing `POTRF` loads left out due to insignificance). 172
14.4. Results of the initial benchmark run. Power values are average power. E_{total} and P_{total} include idle power. 173
14.5. Comparison of energy measurements (M) and predictions (P) by individual board. All values are given in Joules. 176
14.6. Results of the second benchmark run, including relative difference to energy measurements of the first benchmark run. 178
14.7. Difference between both sets of energy measurements by individual board. Absolute values are given in Joules. 179
14.8. Results of an early benchmark on an earlier variant of the evaluation platform, leading to the discovery of irregularities related to C1+ boards. Energy figures exclude idle power. 179

15.1. Comparison of TCP networking and Eth network protocol bandwidth. Columns specify maximum, mean, and standard deviation for a given pair of communication partners. 186

Listings

9.1. Example of a kernel definition in an XML serialisation. 78
9.2. Example of a task definition in an XML serialisation. 78
9.3. Example of a dependency in an XML serialisation 79
9.4. Skeleton program excerpt that generates Cholesky matrix decomposition task graphs in different parallelisation granularities. 80
9.5. Example of a small platform graph in an XML serialisation 90
9.6. Excerpt of an XML serialisation of a platform resource model. Parameters are given in W, nJ, or ns, where applicable. 91
9.7. Task graph task with mapping annotation in an XML serialisation. . . . 95

Nomenclature

ADC Analog-to-Digital Converter

ALU Arithmetic and Logical Unit

API Application Programming Interface

ARM CPU architecture originally devised by Acorn Ltd.

BLAS Basic Linear Algebra Subprograms

CE Communication Element

CPLD Complex Programmable Logic Device

CPU Central Processing Unit, i.e. Main Processor

DC Direct Current

DMA Direct Memory Access

DSE Design Space Exploration

DSO Dynamically Shared Object

DVFS Dynamic Voltage and Frequency Scaling

EDA Electronic Design Automation

FLOP/s Floating-Point Operations per Second

FPGA Field Programmable Gate Arrays

FPU Floating Point Unit

GPGPU General Purpose GPU

GPIO General-Purpose Input/Output

GPU Graphics Processing Unit

HAL Hardware Abstraction Layer

HDL Hardware Design Language

HLS High-Level Synthesis

HPC High Performance Computing

HPL Highly-Parallel LINPACK

IC Integrated Circuit

IPC Instructions Per Cycle

ISS Instruction Set Simulator

I²C Inter Integrated Circuit

MCU Microcontroller Unit

MOSFET Metal–Oxide Semiconductor Field Effect Transistor

MPI Message Passing Interface

NTP Network Time Protocol

OS Operating System

OSSS Oldenburg System Synthesis Subset

PCB Printed Circuit Board

PCG Platform Communication Graph

PE Processing Element

PHG Platform Hierarchy Graph

PMIC Power Management Integrated Circuit

PTP Precision Time Protocol

RAM Random Access Memory

RTL Register Transfer Level

SBC Single-Board Computer

SDF Synchronous Dataflow

SIMD Single Instruction Multiple Data

SMT Simultaneous Multi-Threading

SoC System on Chip

SPI Serial Peripheral Interface

SSH Secure Shell

TAI International Atomic Time

TLM Transaction Level Modelling

TSN Ethernet Time Sensitive Networking

TSN Time Sensitive Networking

VCD Value Change Dump

x86 CPU architecture originally devised by Intel

Acknowledgements

Parts of the research leading to these results has received funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement No. 609757 (FiPS - Developing Hardware and Design Methodologies for Heterogeneous Low Power Field Programmable Servers).