Predicting Performance and Energy Efficiency for Large-Scale Parallel Applications on Highly Heterogeneous Platforms

Jörg Walter, Ralph Goergen
OFFIS – Institute for Information Technology
Oldenburg, Germany
joerg.walter, ralph.goergen@offis.de

Wolfgang Nebel
University of Oldenburg
Oldenburg, Germany
wolfgang.nebel@uni-oldenburg.de

Abstract. Predicting the performance of parallel programs for large-scale parallel platforms is challenging due to the disparity between development system and target platform. This is even worse now that energy efficiency is a universal concern and platforms move towards highly heterogeneous reconfigurable systems containing GPUs, FPGAs, and other unconventional processing elements.
In this paper we present a simulative approach that predicts energy usage and performance of parallel software on large heterogeneous platforms. It simulates communication activity in detail while abstracting functional behaviour. This allows developers to quickly compare and optimise application designs, hardware configurations, and mapping alternatives even without a fully working target platform.

1. Introduction

Energy efficiency is a universal concern by now: the world's fastest supercomputers exhibit a de facto 20 MW power limit, in mobile computing it affects battery life time; even high-end workstations are constrained due to size and noise requirements. An equally universal concern is parallel application design. Well-established in high-performance computing (HPC) as in embedded system level design, by now general-purpose computing has embraced heterogeneous parallelism as well.

Emerging embedded systems, e. g. from the automotive domain, already span hundreds of compute units and run performance-demanding algorithms. Essentially, embedded system level design will face the challenge of optimising high-performance computing applications at a scale not suitable for current approaches.

In this paper we present a generalisation of an earlier experiment of using design space exploration for high-performance parallel application assessment [10]. The overall flow works with the core idea that simulation is abstract; no application code executes during simulation. It has the following advances over the state of the art in either the embedded or HPC domain:

1. It is able to simulate HPC scale applications running on large heterogeneous cluster platforms fast enough to conduct application assessment and optimisation on a developer workstation; at this, it also copes with unconventional processing elements like FPGAs or many-core processors.

R. Wimmer (Hrsg.): MBMV 2016
ISBN: 978-3-00-052380-9

2. Simulation handles performance and energy at a similar level of detail. Platform component models may generate additional optimisation hints, like a bus model that tracks excessive communication congestion.

3. It only needs a single specimen of each processing element architecture, it does not need the final target system. Therefore, our methodology is equally suitable to configure hardware platforms optimised for a given workload.

This paper has the following structure: The next section lists research related to our methodology. Section 3 explains how our methodology works and defines the hard- and software models required as input. In Section 4, we evaluate an application example based on a real-world scenario. Finally, Section 5 summarizes our findings and gives an outlook to future research.

2. Related Work

Execution-driven system simulation is a common technique for timing and energy predictions. Many flexible simulation platforms exist that manage power as well as timing, e. g. Gem5 [3] or the COMPLEX framework [8]. The former is several orders of magnitude slower than real-time; the latter achieves high performance by using SystemC with host-based execution, but it is still too slow for the scale we target.

Parallel discrete-event simulation (PDES) provides a speed-up, and combining this with dynamic binary translation [2] yields impressive results. Still, these do not overcome the inherent performance limits of execution-driven (functional) simulation. Our abstract approach is many orders of magnitude faster than that.

Analytical modelling approaches can be faster than simulation-based approaches. One example is [6], but it addresses single-core performance only. Purely analytical modelling of multi-core architectures or even entire cluster systems, with energy as well, is unsolved so far. Instead, our simulator can use such models as an alternative characterisation process.

In the high-performance computing (HPC) world, a common approach is to record execution traces and simulate these event traces instead. Examples are PSINS [12] and TaskSim [11]. The main disadvantages of trace-driven simulation are that a fully parallelised application must be executed on real hardware, that those traces have limited portability to other systems, and that they usually do not address energy usage. Our approach allows designers to optimise application and hardware without access to a full HPC system, and with almost no restrictions on target platforms; it doesn't even require a finished application.

[9] shows an approach similar to ours, using PDES with skeleton applications. We offer energy predictions and a more detailed, characterisation-based model of performance. SimMatrix [14] even handles exa-scale systems, but is restricted to a specific cluster architecture and work-load; it, too, ignores energy.

3. Application Assessment Flow

Fig. 1 shows the overall flow of our methodology: Designers have a source application model in form of a sequential program. They separate the application into several compute-intensive kernels and extract them from the application, including support code as needed. Finally, they build a task

ISBN: 978-3-00-052380-9

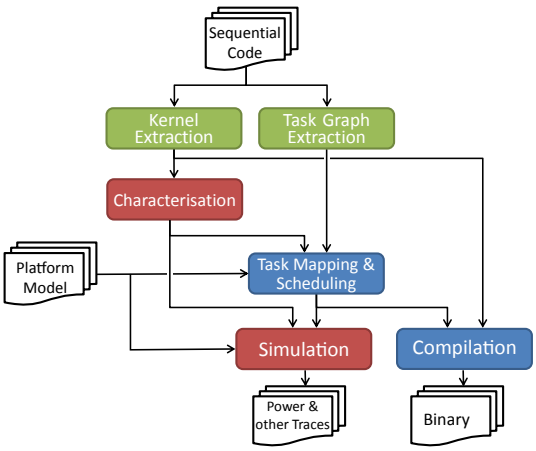

Figure 1: Overview of our assessment flow.

graph (see Section 3.1.1) out of the kernels, so that task graph plus kernel code is functionally equivalent to the source application. With the stand-alone kernels, they perform a characterisation process (see Section 3.1.3).

Alternatively, designers can perform a top-down modelling process from scratch: They start out with just a task graph representing the application in its current design stage. In this case, characterisation data is derived from e. g. pre-existing characterisation databases, proof-of-concept kernel implementations, or even just estimates of attainable or desired performance.

The next step is mapping and scheduling (see Section 3.2), which uses the task graph, characterisation data, and an abstract model of the target platform (see Section 3.1.2) to map each task of the task graph to a processing element (PE) on the target platform and generates a queue of tasks for each PE. Simulation then uses this mapping in conjunction with characterisation results to predict run-time and energy consumption of the entire system, as well as other metrics provided by simulation models (see Section 3.3).

With these predictions, designers 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. When satisfied, they can create an executable version of the parallelised application from the mapped task graph in conjunction with the extracted kernels.

3.1. Abstract Models

This subsection defines our abstraction formally; technically, these models are encoded in XML.

3.1.1. Application Model

Applications are represented as task dependency graphs, also called task precedence graphs or just task graphs, as commonly defined in the HPC research community [1]. Recent work has confirmed that it still applies to modern HPC problems and even offers performance improvements over traditional programming styles [5]. It is closely related to macro data-flow programming and the synchronous data-flow model of computation as used in the embedded domain. Fig. 2 shows an example.

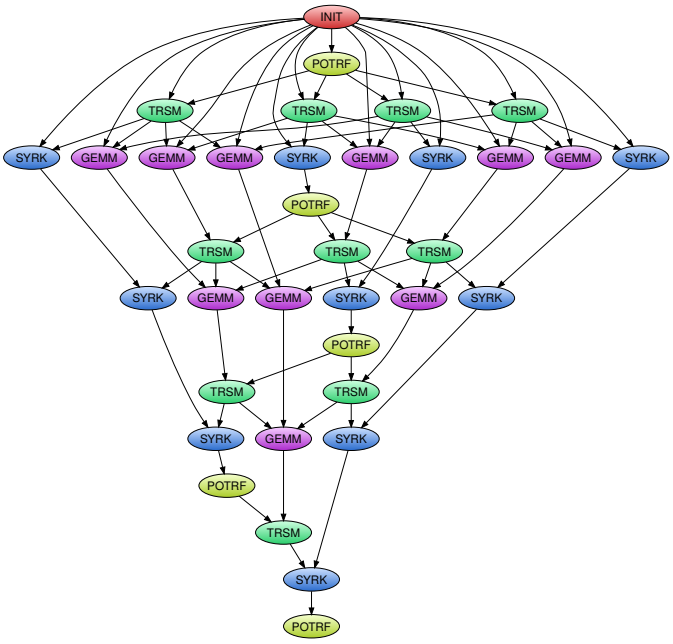

Figure 2: Example of a task graph performing Cholesky matrix decomposition with 5×5 matrix subdivision. Colour denotes different kernels.

We add annotations in order to express computation and communication complexity: a task graph $TG = (T, D, K, \kappa, \delta)$ is an annotated directed acyclic graph, where T is the set of tasks and $D = \{(t_i, t_j) \in T^2 | t_j \text{ depends on completion of } t_i\}$ are the precedence constraints between them.

Each task is an execution of a kernel, i. e. kernels represent code, and tasks are invocations of that code. Kernels can be parametrised with variables, for example to express variations in data size with an otherwise identical algorithm. Function $\kappa : T \to K \times V$ maps tasks to kernels, where K is the set of known kernels, and V is the set of variable assignments.

Kernels have any number of data inputs and data outputs. Function $\delta : D \to \mathcal{P}(O \times I)$ associates zero or more data dependencies with each precedence constraint, where I denotes the set of all possible inputs and O denotes the set of all possible outputs, both of which are derived from K.

Conceptually, tasks $t_i \in T$ execute as soon as all data objects associated with incoming dependencies have been received. They execute atomically and have no side-effects, only communication as expressed by δ. Communication always happens after the originating task has completed execution and before the destination task starts execution.

On a real system, tasks may be delayed or interrupted/resumed depending on schedule, run-time middle-ware, and operating system. They also have the obvious side-effects of causing a PE to consume energy and time. Our application model does not cover these properties; rather, they emerge during simulation. Section 3.3.1 shows how we derive them.

Since the goal is to simulate large applications on distributed platforms, we assume coarse-grained task parallelism. Another assumption is that the application is a composition of relatively few kernels, i. e. $|K| \ll |T|$, otherwise the characterisation process becomes infeasible.

We do not prescribe any particular method to create such task graphs. Common examples are fully manual specification, static code analysis (see for example [4]), or partial evaluation.

Stand-alone skeleton applications are a common approach to partial evaluation. They offer the advantage of configurability: with little effort, designers can create skeleton applications that

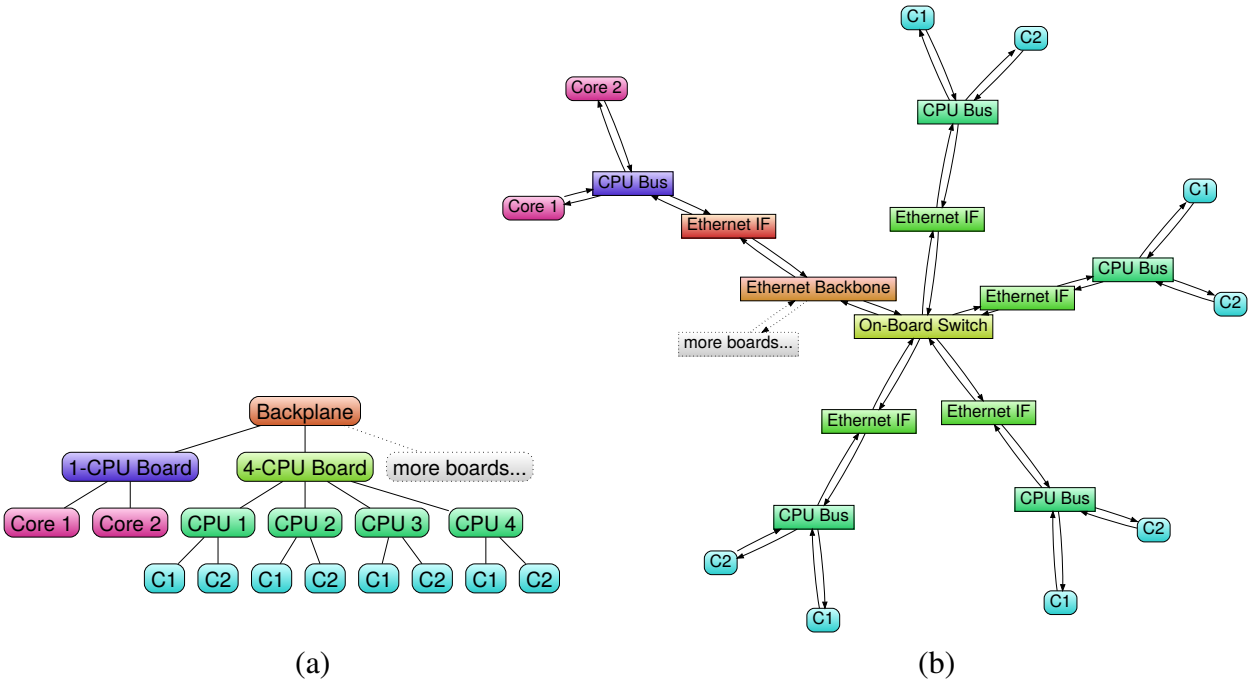

Figure 3: Platform hierarchy (a) and communication (b) graph of a heterogeneous cluster architecture. Colour indicates different hardware models, shape indicates common nodes.

generate multiple task graphs that differ in degree of parallelisation, problem size, and similar properties. This allows fully automated design space exploration (as shown in Section 4).

3.1.2. Platform Model

Simulation handles computation and communication in different ways, so we use an abstract platform description that is actually a pair $PG = (PHG, PCG)$ of two related graphs:

The platform hierarchy graph $PHG = (N, H, M_N, \eta)$ is an annotated rooted tree, where N is the set of hierarchically nested nodes, and relation $H = \{(n_i, n_j) \in N^2 | n_i \text{ contains } n_j\}$ expresses the node hierarchy that must result in a single root element. Our flow uses this structure to automatically build a simulation model as outlined in Section 3.3. Fig. 3 (a) shows an example.

Leaf nodes represent actual processing elements (PEs), for example a single CPU core, a fixed-function ASIC or an FPGA fabric. Internal nodes represent groupings of nodes, like a multi-core CPU or a multi-CPU main board. The set P of available processing elements is thus defined as $P = \{p_i \in N | \nexists n_j \in N : (p_i, n_j) \in H\}$ (all leaf nodes in the hierarchy).

M_N is the set of available hardware architecture models. Function $\eta : N \rightarrow M_N$ associates each node with such a model. In Fig. 3 (a), colours denote different models. The set M_P of PE models is thus $M_P = \eta(P)$.

The second graph, the platform communication graph $PCG = (C, L, M_C, \lambda)$ is an annotated directed graph; see Fig. 3 (b) for a simplified example. C is the set of communication elements (CEs), e. g. network interfaces, bridges, or even just the PCB wiring of a shared bus. Technically, the latter is a purely passive component. It is still an active CE during simulation, since something must represent the arbitration protocol between multiple bus connections.

Set $L = \{(c_i, c_j) \in C^2 | c_i \text{ can transmit data to } c_j\}$ represents links between CEs, for example a

connection of a system-on-chip external bus driver to the bus on the containing PCB. Similar to η, function $\lambda : C \to M_C$ maps CEs to simulation models.

3.1.3. Characterisation Database

We use a database as source for timing data. Our design intention is to perform $|K| \cdot |M_P|$ measurements on real hardware: Each measurement consists of multiple isolated runs of a single kernel with typical input data. Designers provide that input and repeat the process for different kernel variable assignments as required. For this, they only need a single specimen of each PE architecture.

Designers can populate the characterisation database from other sources as well, like pre-existing databases with similar characteristics. By applying techniques like in [6], timings can be extrapolated from existing timings with decent accuracy. Ultimately, the source of this data is irrelevant—even made-up values are useful, e. g. to find out if a (hypothetical) optimisation would have significant impact on overall application behaviour.

For each kernel $k \in K$, and each kernel variable assignment, the database contains information about the set of data objects that k requires as inputs, the set of data objects that k produces as outputs, both of which include their transmission size in bytes, and a set of average execution times for k running on each PE architecture $a \in M_P$.

The database doesn't need to contain execution times for all kernel/PE combinations. Some PE architectures may not be able to execute arbitrary kernels (e. g. a hardware video codec) or there might not be a suitable implementation of a kernel (e. g. for an FPGA, where the effort might be deemed too high). The mapping stage will not create mappings for uncharacterised combinations.

We assume constant communication size across all PEs. At this scale and abstraction level, this is a reasonable assumption. Since kernel code may differ between architectures (as long as it is functionally equivalent), kernels can handle data conversion themselves, if required.

3.2. Mapping and Scheduling

We use a combined mapper/scheduler that generates a queue of tasks for each PE in $O(|P| \cdot |T|)$ time [13]. It uses a constructive hybrid heuristic: A simple earliest-finishing-time list-based scheduler works on a linear task queue, while simulated annealing explores random permutations of tasks in that queue. It uses a simple two-state power model (active and idle) in order to minimise a user-selectable cost function, e. g. energy delay product.

3.3. Simulation

From these models, our simulator builds a simulation model that is able to provide timing and energy estimates for a full application run. It consists of two main parts: Models for processing elements (PEs) that model abstract computation, and models for communication elements (CEs) that model data transmission. Attached to each is a power state machine (PSM) that tracks energy consumption.

PE and CE models as well as PSMs are implemented as generic, parametrisable SystemC modules. A separate platform characterisation process supplies required parameters. Currently, this is used to configure bandwidth and latency of CE models, and to configure PSMs.

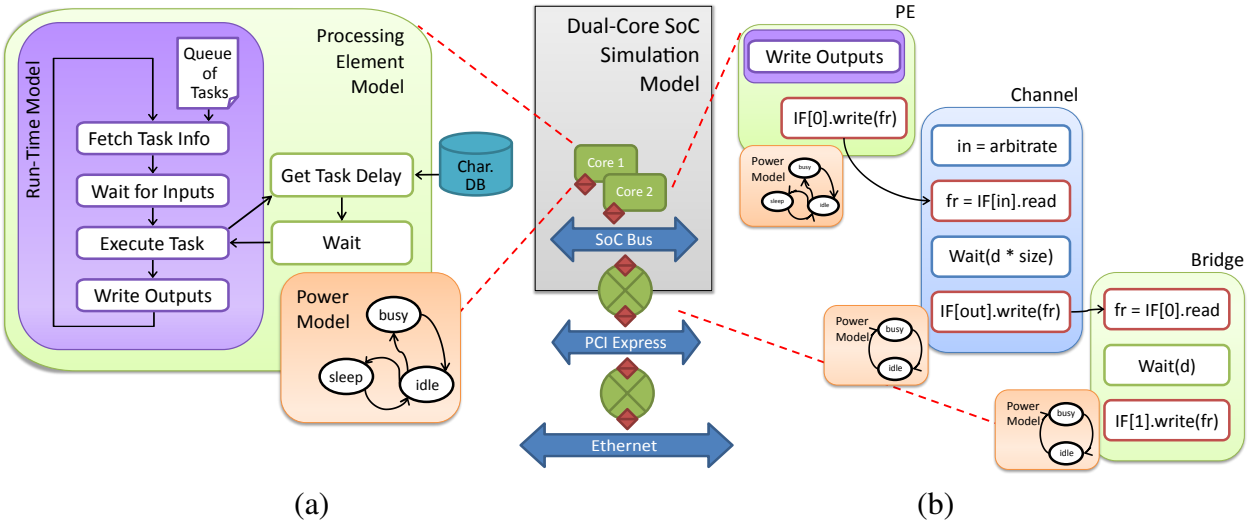

Figure 4: Simulation models of processing (a) and communication (b) elements.

3.3.1. Processing Element Models

Fig. 4 (a) shows the general structure of PE models. As main component, they contain a portable run-time (middle-ware) model. The run-time model interacts with its PE model through a minimal OS abstraction API. There is also a Linux implementation of this abstraction layer. This allows simulation and execution to run the exact same middle-ware code, making its behaviour highly predictable.

Our run-time middle-ware implementation takes a list of tasks to execute. It looks at the first task, removes it from the queue, and waits until all data dependencies are fulfilled. Then it asks the PE to execute the task. After the PE returns, the run-time writes abstract output frames representing generated data to the output channels. This process repeats until no tasks are left in the queue.

The PE model executes a task by fetching its kernel execution time from the characterisation database and waiting for that amount of simulation time.

The associated power state machine (PSM) models energy usage of the executing task based on discrete power states that represent actual hardware behaviour. It may be a trivial state machine with one state, it might employ a fixed transition policy (lowest power during communication delays, highest power during execution delays), or it could account for more complex run-time management of hardware behaviour like voltage/frequency scaling.

Non-PE Nodes In order to account for power usage of other resources, all nodes of PHG have an associated hardware model and PSM. They can also model cases like suspending or powering down entire cluster servers, which can yield important energy savings during phases of low utilisation.

3.3.2. Communication Element Models

We model communication in an abstracted way as well, but we do consider contention, since this tends to be the most critical aspect on large parallel platforms. Fig. 4 (b) shows various CE models representing PEs as end points, channels as transmission media, and bridges as translation and forwarding mechanism between channels.

All CEs work in a similar way: they read the abstract data frame, wait for the time required to transmit it (based on frame content and platform characteristics), and finally write it to the appropriate output port. PEs and bridges just originate, forward, and terminate data frames. The most significant difference of channels is that they model access arbitration, so they perform the arbitration algorithm as an additional step.

Data frames contain no application data, only information about size and destination. A second abstraction is that these communication models do not just model a physical transmission medium and its arbitration behaviour, but also the full protocol stack including forwarding and routing behaviour.

The exact level of detail is up to the designer; in our evaluation we accounted for all protocol overhead by using measured average net bandwidth as speed metric, but we modelled arbitration and congestion explicitly. Channels route frames using static routing tables, which is sufficient to model for CPU buses and local Ethernet networks.

Most CEs use a simple two-state PSM that models idle and active power.

3.3.3. Simulation Outputs

The simulation model generates a stream of data values that are accumulated appropriately to generate several outputs: total energy consumption and execution time, power-over-time traces for selected components, task start and finish times, communication delays, PE utilisation, and CE contention (i. e number of blocked ready-to-send transmissions over time). It uses the generic tracing facilities integrated in SystemC and SystemC AMS to output VCD or tabular files.

4. Use Case: Combined Hardware/Software Optimisation

In order to demonstrate the viability of our approach, we show how our method handles a well-known parallel algorithm, targeting a reconfigurable heterogeneous cluster system. We chose a scenario that matches our actual use-case: The application designer wants to find the optimal hardware configuration for an application, although the machine is not yet finished. We explored hardware designs that are still under development; it was based on characterised components, but without a full system available.

4.1. Benchmark Application

We chose double-precision Cholesky matrix decomposition as demonstration workload because it has a non-trivial parallelism structure with non-uniform task run times. It combines common properties of parallel applications: A massively parallel phase with high fan-out in the beginning stresses the communication subsystem, while a mostly serial critical path in the end requires accurate computation models (see Fig. 2).

We used a skeleton application to generate the task graph in different sizes. Each variant solves the same overall problem, just in different degrees of parallelisation. Table 1 lists the exact application configurations we tested.

Table 1: Evaluated application configurations.

Name	Matrix Tiles	Tile Size	Matrix Size	Tasks
Ch-5	5×5	8192×8192	40960×40960	37
Ch-10	10×10	4096×4096	40960×40960	223
Ch-20	20×20	2048×2048	40960×40960	1.5 k
Ch-40	40×40	1024×1024	40960×40960	11 k

4.2. Reference Platform

We characterised our application via measurements done on a reconfigurable cluster server system [7] that offers power measurements for each individual cluster node. It can contain up to 18 base boards that adhere to the ComExpress standard. Base boards were interconnected through a switched 1 GBit/s Ethernet network.

The characterisation platform was a hardware development server which was not in a final configuration: It had one base board carrying an Intel i3 3120-ME dual-core processor running at 2.4 GHz and three ARM base boards, each equipped with four Samsung Exynos 5250 dual-core processors (ARM Cortex A15) running at 1.7 GHz. Each processor had 2 GiB local memory. The processors on a single ARM base board communicated via locally-switched 1 GBit/s Ethernet. Fig. 3 shows a subset of this platform.

While the characterisation platform is incomplete, our methodology only requires one CPU of each architecture, so the development server was sufficient to conduct this evaluation.

4.3. Characterisation

For kernel characterisation, we isolated four kernels: POTRF, SYRK, TRSM, and GEMM. We ran each of them 10 times on a single ARM and a single x86 core using randomly generated matrices as inputs. We repeated this procedure for each matrix tile size listed in Table 1.

For platform characterisation, we measured PE power usage in the idle (active but not computing/communicating) state and during full computation load using an artificial benchmark.

We characterised communication by measuring time and energy while transferring large blocks of dummy data. We performed this measurement for each link type in the platform (see Fig. 3).

Platform characterisation data became part of our SystemC simulation models, while kernel characterisation data constituted the characterisation database.

4.4. Simulation Models

We configured our generic PE and communication models matching the characterised hardware and created a platform description matching the full system.

We did not have access to physical platforms matching the explored hardware variants, and we explored future system compositions based on components that are currently in development: ARM boards with less power consumption, x86 boards with more powerful CPU, and a 10 GBit/s Ethernet backbone. For comparison we also simulated the same platforms using the existing 1 GBit/s backbone.

Table 2: Target platforms used for simulation.

Name	x86 Boards	ARM Boards	x86 Cores	ARM Cores
ALL-X86	18	0	36	0
ALL-ARM	0	18	0	144
EQ-BOARDS	9	9	18	72
EQ-CORES	14	4	28	32

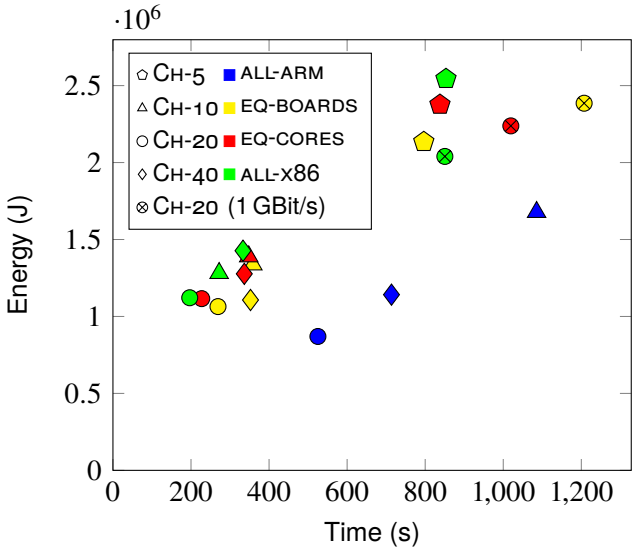

Figure 5: Simulation results.

Table 2 lists all hardware configurations that we used: two homogeneous platforms (ALL-X86 and ALL-ARM) use a single CPU architecture, while two heterogeneous platforms consist of an equal number of baseboards and a roughly equal number of CPU cores (EQ-BOARDS and EQ-CORES, respectively).

4.5. Execution

We simulated each application configuration on each platform, totalling 16 simulation runs. The entire simulation set takes less than a minute on a 2.13 GHz Core2 Duo CPU. The majority of time is actually spent in the XML parser. Simulation itself is on the order of tens of microseconds.

As an example for hardware design decisions beyond CPU type, we ran four additional simulations with a 1 GBit/s Ethernet backbone instead of 10 GBit/s.

4.6. Results

Fig. 5 shows a scatter plot of energy versus time after simulating a fully populated server system in various configurations. Colours denote the four target platform candidates, while shapes indicate the degree of parallelisation. Crossed circles represent the 1 GBit/s Ethernet platform.

Obviously, Cholesky-20 is the optimal parallelisation strategy for systems at this scale: there is a Pareto-optimal frontier that solely consists of hardware configurations running CH-20. The ALL-X86

configuration is fastest, while the ALL-ARM configuration uses least energy. Using half x86 and half ARM boards is about twice as fast as ALL-ARM, at 22 % increased energy consumption. ALL-X86 adds another 5% energy for another 30% time saved.

From the 1 GBit/s backbone results (crossed circles) we conclude that a faster backbone has much more impact on time and energy than any other change in machine configuration, so for this particular application it might be worthwhile to explore more communication variants.

5. Conclusion

In this paper, we have shown an abstract simulation methodology for parallel applications on large, highly heterogeneous platforms. It combines established models from high-performance computing with simulation techniques from embedded computing. Models are generic enough to express irregular processing elements like FPGAs or many-core processors.

Simulation speed was several orders of magnitude faster than running the application natively. This makes it suitable for large-scale design space exploration. We have shown how our approach allows exploring high-level algorithm variants at the same time as platform variants (CPU types, communication hardware, communication topology). This did not require target platforms to be available. Due to run-time middle-ware modelling, it could as well be used to evaluate different task graph execution strategies.

Due to the lack of a full-scale target platform, we were unable to perform a solid accuracy analysis. Preliminary comparisons between smaller simulated and real systems yielded promising results, but these tests did not reach our target scale. We are in the process of setting up a full-scale reference platform that non-intrusively monitors timing and energy in parallel for all nodes. We plan to use the PARSECSs benchmark suite [5] with that platform in order to perform much improved evaluation.

A notable limitation of our approach is that, due to the static nature of our task graphs, it can't represent dynamic control flow. The main mitigation strategy would be a skeleton application that generates many or even all possible control flows. This allows designers to make a trade-off between analysing algorithm/platform variants with few control flow variants, or to concentrate on evaluating application performance for a specific platform and algorithmic structure.

We plan to record histograms of run times to address data-dependent behaviour; simulation performance then allows extensive Monte Carlo simulations using random histogram sampling. In addition, statistical modelling lets us provide confidence estimates for predictions, e. g. an upper bound representing at least a given fraction of all cases.

Another area of ongoing research is how to model non-communication resource conflicts, e. g. inter-core cache pollution.

Acknowledgement 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).

References

[1] Adve, Vikram and Rizos Sakellariou: *Application representations for multiparadigm performance modeling of large-scale parallel scientific codes*. International Journal of High

Performance Computing Applications, 2000.

[2] Almer, Oscar, Igor Böhm, Tobias J. K. Edler von Koch, Björn Franke, Stephen C. Kyle, Volker Seeker, *et al.*: *A Parallel Dynamic Binary Translator for Efficient Multi-Core Simulation.* International Journal of Parallel Programming, 2013.

[3] Binkert, Nathan, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, *et al.*: *The Gem5 Simulator*. SIGARCH Comput. Archit. News, 2011.

[4] Bosilca, George, Aurelien Bouteiller, Anthony Danalis, Mathieu Faverge, Thomas Herault, and Jack J Dongarra: *PaRSEC: Exploiting Heterogeneity to Enhance Scalability*. Computing in Science & Engineering, 2013.

[5] Chasapis, Dimitrios, Marc Casas, Miquel Moretó, Raul Vidal, Eduard Ayguadé, Jesús Labarta, and Mateo Valero: *PARSECSs: Evaluating the Impact of Task Parallelism in the PARSEC Benchmark Suite.* TACO, 12(4):41, 2016.

[6] Eyerman, Stijn, Lieven Eeckhout, Tejas Karkhanis, and James E. Smith: *A mechanistic performance model for superscalar out-of-order processors.* ACM Trans. Comput. Syst., 27(2), 2009. http://dblp.uni-trier.de/db/journals/tocs/tocs27.html#EyermanEKS09.

[7] Griessl, René, 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 *12th IEEE Int. Conf. on Embedded and Ubiquitous Computing*, 2014.

[8] Grüttner, Kim, P. Hartmann, K. Hylla, S. Rosinger, W. Nebel, F. Herrera, *et al.*: *The COMPLEX reference framework for HW/SW co-design and power management supporting platform-based design-space exploration*. Microprocessors and Microsystems, 2013.

[9] Janssen, Curtis L., Helgi Adalsteinsson, and Joseph P. Kenny: *Using simulation to design extremescale applications and architectures: programming model exploration.* SIGMETRICS Performance Evaluation Review, 2011.

[10] Knocke, Patrick, R. Görgen, J. Walter, D. Helms, W. Nebel, *et al.*: *Using early power and timing estimations of massively heterogeneous computation platforms to create optimized HPC applications*. In *12th IEEE Int. Conf. on Embedded and Ubiquitous Computing*, 2014.

[11] Rico, Alejandro, Felipe Cabarcas, Carlos Villavieja, Milan Pavlovic, Augusto Vega, Yoav Etsion, *et al.*: *On the simulation of large-scale architectures using multiple application abstraction levels.* TACO, 2012.

[12] Tikir, Mustafa M., Michael Laurenzano, Laura Carrington, and A. Snavely: *PSINS: An Open Source Event Tracer and Execution Simulator for MPI Applications.* In *Euro-Par*, 2009.

[13] Walter, Jörg and Wolfgang Nebel: *Energy–Aware Mapping and Scheduling of Large–Scale Macro Data–Flow Applications*. In *1st International Workshop on Investigating Dataflow in Embedded Computing Architecture (IDEA 2015)*, January 2015.

[14] Wang, Ke and Ioan Raicu: *SimMatrix: SIMulator for MAny-Task computing execution fabRIc at eXascales*. In *High Performance Computing Symposia (HPC)*, 2013.