Using early power and timing estimations of massively heterogeneous computation platforms to create optimized HPC applications

Patrick Knocke*, Ralph Görgen†, Jörg Walter‡ and Domenik Helms§
OFFIS - Institute for Information Technology, Oldenburg, Germany
Email: *patrick.knocke@offis.de, †ralph.goergen@offis.de,
‡joerg.walter@offis.de, §domenik.helms@offis.de

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

***Abstract*—The ever rising energy and accordingly cooling demands are a major hurdle for the scalability of todays supercomputers. We are challenged with the need to increase computation performance to cope with the rising complexity of calculations on the one hand and the need to keep the energy/cooling demand stable or in the best case even to reduce it. Recently, one widely discussed way to do this is the integration of heterogeneous computation devices into the supercomputer systems as these tend to have a far better performance/energy ratio for large classes of applications. The obvious drawback of heterogeneous systems is the additional design complexity for the software development in order to efficiently use these devices in terms of performance as well as power.**
For this reason we propose a flow, which assists the software developer at design time, offering immediate power- and performance-estimation. Such approaches are already known in the embedded world, helping there to select between different design possibilities, and will be used to get the best possible performance from a massively heterogeneous computation platform, while still keeping the energy consumption in mind.

***Keywords*—Computer aided software engineering; Multicore processing; Performance analysis; System analysis and design;**

I. Introduction

With the rise of heterogeneous systems the effort needed to write optimal applications running on these systems has increased manyfold. Multiple technologies like OpenCL or MPI have been developed to ease the development process and enable the user to harvest the processing power of different types of computation devices. But the complexity of new massively heterogeneous systems, consisting of multiple CPUs, GPUs, embedded CPUs, and/or FPGAs, is so high that the developer of an application does not really know which computation device would be best fitted for a specific piece of the application in regard to the overall performance and power usage. To find an optimal solution to the problem at hand the developer would have to write different versions of the same application, test it on the system and fine tune it based on the resulting performance and used power values. As processing time on supercomputers is quite expensive this is not a desirable situation.

The research leading to these results has received funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement № 609757 (FiPS - Developing Hardware and Design Methodologies for Heterogeneous Low Power Field Programmable Servers).

In this work we therefore propose an early power and timing estimation approach for HPC applications running on massively heterogeneous systems. The application is split into smaller so called kernel codes, which are mapped to the different processing elements available, and is being simulated in regard to its timing and power behavior. The resulting simulation data enables the developer to fine tune the application without the need to actually run it on the target system.

After a review of the related work in the following section, we present the graph representation of the architecture and the algorithm in Section Section III. Section Section IV describes how to determine the energy demand and runtime of a algorithm to architecture allocation under test and how to heuristically optimize the allocation. Different design alternatives in terms of hardware selection and algorithm tiling for the Cholesky Matrix decomposition are assesed in Section Section V and Section Section VI finally concludes.

II. Related Work

The increased use of supercomputer systems has led to a vast research area with a multitude of different approaches to estimate and optimize high performance computing systems.

In [1] Gurumurthi et. al. present a complete system estimation framework built on top of the SimOS infrastructure to study the power and performance behaviour of applications. It does so by simulating the complete computing platform including the operating system, CPU, memory hierarchy and low-power disk subsystem with their so-called SoftWatt approach. But because of the inherent runtime problem of complete system simulations, this approach is not really feasible for the current challenge to estimate supercomputing systems with thousands of nodes and processing elements.

The research therefore shifted to estimate the power and energy usage of applications in a more abstract way like by Tiwari et. al. in [2], where three specific kernels, matrix multiplication (MM), stencil computation and LU factorization, are characterized on the target platform and are used to train CPU and memory models based on artificial neural networks. The resulting models further model the influence of compiler-based optimizations and different hardware settings. Many of these abstract approaches use a task graph representation of the application as presented by Adve et. al. in [3].

Witkowski et. al. [4] present another approach to estimate the power consumption of HPC applications where the estimation is done during execution of the application of the actual hardware. This results in pretty precise results but is not really feasible during the optimization phase. Shin and Kim [5] try to optimize the power and performance of an application during runtime on a multiprocessor system by utilizing task scheduling and dynamic voltage scaling of the processing elements; Verma et. al. [6] try to dynamically place HPC applications on virtual machines throughout a server cluster to enable migration of applications and power management of the cluster hardware.

The EU FP7 project PEPPHER [7] targets efficient usage of hybrid computing systems; it makes applications performance-portable between different types of computation devices through the use of a component-based programming framework. The project offers a high level of automation, but targets single-node platforms only.

Furthermore, the emerging heterogeneity of supercomputers is well known in the embedded world, where specialised processing elements are used to increase performance and/or reduce the energy consumption. There are already some simulation and profiling approaches like the results of the COMPLEX project presented in [8].

We therefore propose to combine the expertise gained in the embedded world to tackle the need for early power and timing estimations in the high performance computing world. Essentially combining the task graph representation used in the high performance computing world with new statistical based power and timing models, and a simulation approach used in the embedded world. This will increase the usability and performance of early estimations and will thereby enable the user to choose the best hardware platform and algorithm parameters in regard to power and/or timing constraints for a given problem.

III. Power and Timing Estimation of Massively Heterogeneous Systems

In the FiPS project, partners from the high-performance computing world and from the embedded world collaborate. These two communities use partially conflicting terminology. The following overview of the design flow (see Figure 1) also sets a common vocabulary of terms used throughout this paper.

1) Software: The USER has written a sequential golden model of a high-performance computing (HPC) program to run on the FiPS platform. In the context of FiPS, the APPLICATION is that part of the program that she processes using the FiPS resource allocation methodology explained in this document.

The remaining support code including all libraries, the operating system, etc. is considered the RUN-TIME. This even includes parts not usually considered run-time support software, e. g. an application-specific outer loop for batch processing of multiple data sets.

Users separate the application into a collection of TASKs. A task is an atomic unit of computation that does not communicate or have other side effects while it executes. Each task is an instance of a KERNEL, which is a standalone part of

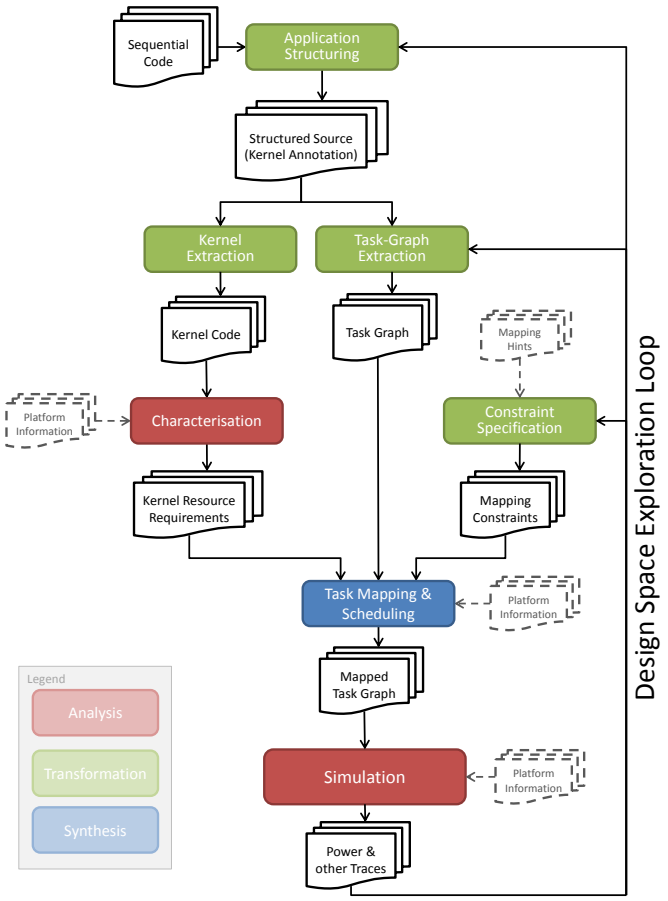

Fig. 1. Design Space Exploration Loop of the FiPS flow

the application source code; each kernel can have many task instances.

Tasks may have DEPENDENCIES on other tasks. These are precedence constraints: A task can only start executing after *all* preceding tasks have completed. Additionally, dependencies can express communication or other side effects, which happen strictly before or after a task's execution (at least conceptually).

Tasks as vertices and dependencies as edges form a directed acyclic graph of the application, the TASK GRAPH. Subsection III-B explains it in more detail.

Tasks by themselves do not imply resource allocation; they merely facilitate resource allocation and scheduling by expressing *potential* parallelism of the application. Section IV shows the resource allocation process, which results in a MAPPED TASK GRAPH, which is then simulated to give detailed feedback about predicted application behaviour.

Users repeat the process of structuring the application into a set of tasks, mapping, and simulating them. This design space exploration (DSE) loop is an integral part of our proposed methodology.

2) Hardware: The PLATFORM is a specific, complete computing system that is targeted while applying the FiPS methodology. It consists of individual PROCESSING ELEMENTS (PEs) that can execute exactly one task at a time. Each PE has an

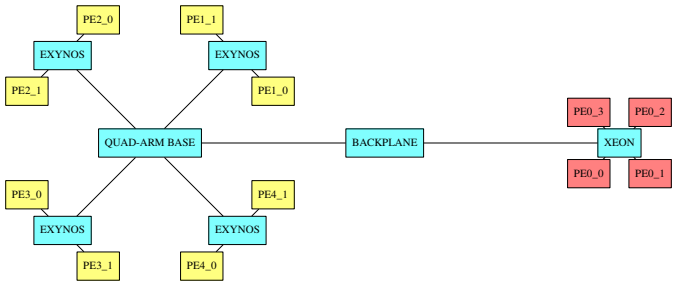

Fig. 2. Platform graph for a simple heterogeneous system. Edges specify available communication links, cyan nodes represent hierarchical composition, while leaf nodes are actual processing elements.

ARCHITECTURE. Typical heterogeneous platforms contain PEs of about two to four different architectures (e. g. HPC systems built from general purpose CPUs and GPUs).

A platform contains run-time support software as well (e. g. operating system, communication infrastructure, etc.). Since the FiPS methodology does not rely on specific run-time support features, platform-specific and application-specific run-time support software are not distinguished further, they are subsumed under the term RUN-TIME.

Multiple PEs can be grouped in compute NODES, which share some resources (e. g. RAM). Nodes can contain other nodes, leading to a hierarchical platform model.

PEs form the vertices of a PLATFORM GRAPH, with available communication links as edges, as can be seen in Figure 2.

A. Platform Representation

The target platform specification consists of two overlapping graphs. One graph is a tree representing the hierarchical composition of system components; it contains information about implicitly shared resources like arithmetic units, shared cache, or memory buses. The second graph specifies explicitly shared communication resources and is not restricted to a tree structure. Figure 2 shows an example platform that consists of two base boards: one carrying an Intel Xeon quad-core processor, and one carrying four ARM daughterboards equipped with one Exynos dual-core System-on-Chip each.

1) Hierarchical View: The hierarchical view consists of generic nodes nested inside each other. Each node can contain other nodes, actual processing elements (PEs), and communication elements (bridges/transceivers, CEs). Processing and communication elements carry information about what time and energy model implementation to use.

In the example in Figure 2, BACKPLANE is the root of this hierarchy, with two child nodes representing the two base boards. The XEON node then contains four processing elements representing four cores of the CPU. It implies shared resources, in this case common cache and memory, which can be taken into account by simulation models. This depends on the actual power and time models and is not specified further.

2) Communication View: PEs and CEs have ports that connect to channels. Channels represent physical transmission media like optical point-to-point-links or gigabit Ethernet. They contain information about communication behaviour (e. g. speed, latency, or arbitration/conflict resolution). Any port can link to any compatible channel.

In Figure 2, communication structure coincides with hierarchical system composition. The communication view does not require this. For example, if both had an additional compatible communication interface, one of the Exynos chips could connect to the Xeon directly without going through the backplane.

B. Task Graph Representation

For resource allocation, applications are represented as task dependency graphs as commonly defined in the HPC research community, e. g. in [3]. By using this representation, the FiPS methodology can base its scientific contributions on state-of-the-art techniques and does not need to reinvent established practices.

A task graph with n tasks is a directed, acyclic graph $G = (V, E)$ where $V = \{v_i | 1 \leq i \leq n\}$ is the set of tasks and $E = \{(v_i, v_j) | v_i, v_j \in V,\ v_j \text{ depends on completion of } v_i\}$ is the set of dependencies.

In most definitions, a task v_i can execute as soon as all predecessor tasks $\text{pred}(v_i) = \{v_j | (v_j, v_i) \in E\}$ have completed execution. The FiPS methodology also takes communication into account: as an extension to usual task graph semantics, a task v_i can actually only execute after all predecessors $\text{pred}(v_i)$ have completed *and* all data associated with incoming dependencies $\text{in}(v_i) = \{(v_j, v_i) | v_j \in \text{pred}(v_i)\}$ have been transmitted. Communication happens after the originating task has completed execution and before the destination task starts execution.

The annotated task graph as used in FiPS is more complex than a simple vertex/edge-weighted graph because of significant heterogeneity in the target platform. The mapping stage uses these annotations for analytical performance and power modelling; our abstract (non-functional) simulator uses characterisation results to achieve fast and accurate system simulation. Tasks and dependencies carry annotations about resource requirements and scaling behaviour so that the mapping and evaluation stages can estimate a task graph's temporal and energetic behaviour.

1) Task Attributes: All tasks are instances of a kernel (i. e. executions of a piece of kernel code). Tasks merely note which kernel they are an instance of, while kernels carry resource requirement data. Tasks also provide the exact kernel parameter configuration they execute. After the mapping stage has run, tasks also carry annotations about which PE will execute them, and in what order.

2) Kernel Attributes: The user specifies some kernel attributes manually, while the characterisation process measures the rest. Kernels carry the following user-specified characteristics:

a) Parameter Declarations: These specify variables that can be used in other characteristics in order to parametrise them. For this reason, other characteristics can be given as simple mathematical expressions instead of constant values. For example, a generic matrix multiplication kernel's resource

requirements vary significantly with matrix size. Users can specify scaling functions for individual characteristics so that the mapping and evaluation processes can extrapolate resource requirements for uncharacterised parametrisations.

b) Inputs/Outputs: Mapping and evaluation requires the amount of data that needs to be transmitted from predecessors/to successors in the task graph. Each logically separate in/output is specified separately to allow data dependency analysis.

c) Complexity Functions: If memory usage and/or runtime are continuous functions of kernel parameters, complexity functions allow interpolation of uncharacterised parameter configurations. In that case, characterization only needs to create as many data sets as required for curve-fitting.

In addition to these attributes, kernel characterisation generates multiple sets of execution time statistics, average power consumption and maximum memory usage, plus an estimate of code size/area consumption.

3) Dependency Attributes: Dependencies carry annotations about which output of the predecessor is connected to which input of the successor. In combination with kernel attributes and task parameters mapping and evaluation can then consider communication time and energy as well.

4) Task Graph Limitations: Since the task graph is an acyclic directed graph, this results in a fully unrolled representation of the application's control flow. This representation has two potential limitations:

1) Task graphs can get very large if fine-grained parallelism is involved (e. g. filtering a video stream pixel-by-pixel).
   This is no real problem: On one hand, users identify kernels manually. This way they control task granularity and can prevent needless task graph expansion. A typical example of fine-grained parallelism are data parallel algorithms implemented using single-instruction-multiple-data (SIMD) instructions. Since these run on a single processing element (PE), the complete algorithm is modelled as a single task, or the task is divided into a fixed number of subtasks that process parts of the input in parallel.
   If, on the other hand, users want to map fine-grained parallelism individually, excessive problem size is not a result of the FiPS methodology. They deliberately chose a large problem size because they expect significant gains despite a slower work flow.
2) Task graph structure may depend on processed data. This is actually the common case: For HPC applications, the most important data dependency is a dependency on input size (e. g. matrix size). This has significant impact on application behaviour. We introduced parametrised kernels for this reason; mapping results are only optimal for a fixed parameter set.
   For other data dependencies, including dynamic loop bounds (e. g. numeric convergence criteria), we follow the observations of [9] and assume that this is either of little practical relevance for HPC programs, or that an outer loop can handle them. Thus, these dynamic loop bounds are not part of the hot code path but part of the surrounding support code.
   Note that data dependencies inside a task are fully supported. These are modelled by the characterisation process.

IV. Simulation and Optimization

5) Extraction: Deriving a fully expanded static task precedence graph from an HPC program is equivalent to static program analysis, which has inherent limitations as to what can be analysed algorithmically and what can't.

Moreover, for typical HPC programs (including the FiPS benchmark applications), task graph size and structure is dependent on the size (and sometimes structure) of the data set to be processed.

As a result, for many real-world problems this can only be done semi-automatically. An established method to generate task graphs is to use a staging program, which is a technique for partial evaluation. Staging programs are reduced versions of programs that only contain high-level algorithmic structure. A staging program for the FiPS methodology contains exactly those computations that influence the task graph in its size and structure, and it outputs that task graph.

Staging can be applied to any program, but it involves manually writing (or rather, reducing) program code. For some programs, fully automatic extraction of the task graph is possible [10]. Consequently, users can choose how the task graph should be generated as long as the result captures all required properties of tasks and precedence constraints.

In particular, beyond a plain task precedence graph, task graph extraction must provide information about data dependencies and problem size(s). Data dependencies connect specific inputs and outputs of dependent tasks so that later synthesis and analysis steps can take communication into account. Problem size is required in order to calculate real resource requirements from parametrised kernel characteristics.

6) Mapping: Task mapping takes a task precedence graph and maps each task to a suitable processing element (PE) of the target platform. It also creates a task schedule for each PE. The mapper tries to minimize energy-delay (task run-time times PE energy consumption during that time), and beyond task execution it considers communication cost as well.

Figure 3 shows an example where the mapper favours energy-efficient ARM cores for highly parallel sections of the application, while x86 cores with high single-thread-performance are preferred for sections with little parallelism.

The mapper outputs a mapped task graph, which is the input task graph annotated with mapping and scheduling decisions. The latter are represented as a ranking of tasks on each processing element, not by actual execution times. Tools using the mapped task graph still need to track all cross-PE precedence constraints in order to determine task start times.

7) Constraint Specification: Mapping must be fast, since our flow includes a human user in the loop. Moreover, actual evaluation happens in a separate simulation stage, so we use a simple heuristic that favours mapping speed over optimality (currently, we use simulated annealing). Due to the big solution space, this results in less optimal solutions.

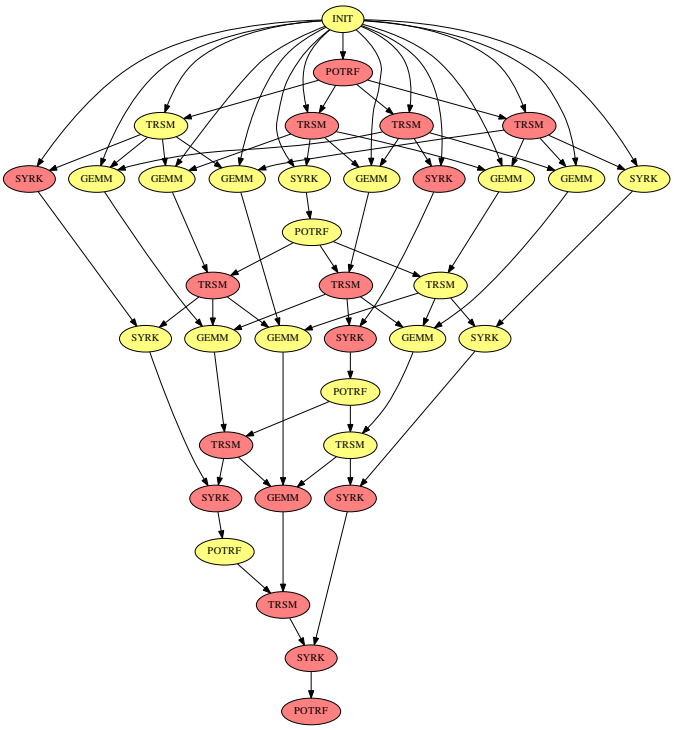

Fig. 3. Mapped task graph of 5x5 Cholesky matrix decomposition, optimized for low energy-delay. Light yellow tasks run on ARM cores, red tasks run on an x86 core.

Since human users are part of the design space exploration loop, they can and should influence or enforce mapping decisions: simulation-derived program statistics from previous runs of the design space exploration loop help them to identify hot-spots in computation and communication that the heuristic did not account for. They can then specify mapping constraints in order to make the mapper favour or discourage certain mapping alternatives in the next iteration; users can also constrain communication in order to improve locality of certain calculations or to spread them out.

A. Kernel Extraction and Characterisation

1) Kernel Extraction: In this step, the code of individual kernels is separated from the overall application. Then, the extracted kernel source can be used for later steps such as kernel characterization or synthesis and compilation. Right now, extraction of the code has to be performed manually by the user. In some cases, the extraction can be done automatically. Prerequisites for automatic extraction are that the code is given as C code and that the kernel is a function. If the code of a proposed kernel is distributed over more than one function, it has to be combined into a single top-level function. Then, extraction is supported by the SMOG tool [11]. This tool reads an input designs source and writes a self-contained portion of the code into an individual target directory. Even though the executable input design may consist of C++ code, a block of code that should be separated must be encapsulated in a single C-style function. The function specifies the boundary of the code implementing a kernels behaviour that is bound for detachment. Its signature defines its interface. The function may call additional sub- routines. Every sub-routine is considered as part of the separated code and included in the separated source code in order to yield a self-contained implementation. After identifying the designated C-function that marks the top level of the block, SMOG gathers all variable, function, and type definitions that are used from the top level function and writes them to separate source code files that represent the input for kernel characterization, synthesis, or compilation.

2) Characterisation: The exact execution time and energy demand of a kernel depends on a number of parameters, of which some are explicitly modelled, such as architecture selection or global configuration parameters. For others especially the dependency onto application data and cache/bus conflicts - explicit modelling is prohibitive when aiming at an above functional abstraction. We thus decided, to use a statistical model, describing the influence of such run-time effects. In order to set up such a statistical model for a kernel, the user has to specify all architectures and algorithm configurations (e.g. tile sizes) for which a delay and energy model is needed and provide typical stimuli (application data) for each configuration. In this initial version, for each stimuli, we measure run-time as well as average power over one stimuli execution with an initially uninitialized cache. We assume having no correlation between the run-times and power averages of two kernels, running on two different PEs. Even though this assumption is obviously not perfectly right, it enables us to treat run-times as statistically distributed and expectation value of the average power consumption as independent from run-time effects. For the total power consumption, which is simply the sum of all PEs power consumption, adding all expectation values directly leads to the expected total power. For the total execution time, this is more complicated. Due to parallelization, the combined run-time of two tasks is the maximum, not the sum of both times, and as maximum is not a linear operator, the expectation value $E(max(a,b))$ is not simply $max(E(a),E(b))$, but a complicated function, depending on the exact distribution of the random variables a and b. Thus we need to store the distribution of run-times per kernel in form of a histogram in order to being able to compute expected total application run-times later on. The following example illustrates our considerations:

Assume to have two kernels which both either run 10 or 20 ms, each with an independent 50% probability. Parallel execution of both finishes after 10ms only in 25% of all cases and in 20ms else. The expectation value of both tasks is 15ms, running in parallel, they are expected to terminate after 17.5ms. We needed to know run-time statistic in order to compute this. Going further with this example: The execution will terminate after 20ms with both PEs busy only in 25% of all cases. In 50% of all cases, one kernel will finish earlier. Assuming average power values P_{act} and P_{idle}, the average energy for parallel execution equals $10ms \cdot 2 \cdot P_{act}/4 + 10ms \cdot (3 \cdot P_{act} + P_{idle})/2 + 20ms \cdot 2 \cdot P_{act}/4 = 30ms \cdot P_{act} + 5ms \cdot P_{idle}$. This means that in the 17.5ms from above, one of the PEs is on average 5ms idle.

Statistic timing propagation is well known from the area of gate level static timing analysis (SSTA) and commercially available (e.g. in Synopsys PrimeTime), but still it is very limited (e.g. very complex for non.Gaussian distributions). In contrast to a conventional SSTA, we are interested in average,

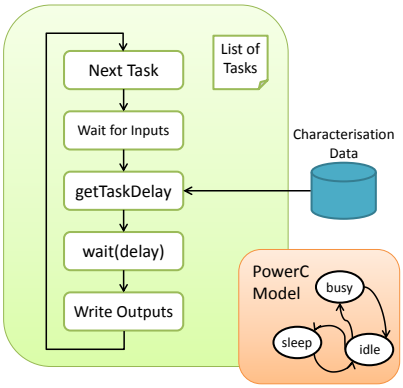

Fig. 4. Simulation model of processing element.

rather than worst case timing, so that we could simplify the problem to ease up its solution and remove the major drawbacks: A simple, yet powerful solution is to fold the input histograms case by case into a new output histogram. This results in an exact solution, but unfortunately leads to a state explosion in the histograms. Looking for average timing, we thus afforded to resample each resulting histogram after each operation to maintain a constant number of histogram classes. This has a drastic impact onto the distribution tails, but is a good prediction for the distribution centers, we are interested in.

B. Simulation

After the mapping phase, a more detailed evaluation of the generated mapping alternative is done by simulation. In contrast to the mapping algorithm, variation of timing and power values is considered here, as well as conflicts in shared ressources. Afterwards, the simulation results can be used to improve the application structuring or mapping.

1) Simulation Model: The simulation model consists of three parts, the platform model, the application model, and the characterisation data. The basic blocks of a server are provided as configurable SystemC modules and channels [12]. Processing elements are modelled in a very abstract way. They do not execute functional program code but only realise the power and timing models. Same for the communication links. They do not transmit actual application data. They do arbitration of concurrent accesses and then only consume time and power according to the amount of data to transmit. With these blocks, the platform model of a server is assembled.

The application model, in fact the mapped task graph, is annotated to the platform model in form of lists of tasks. To each processing element, a list of the tasks supposed to run on this PE is annotated together with the data dependabilities of each of them.

Finally, we need the characterisation data for each kernel gathered in the characterisation phase. For each task execution, the simulation model inquires the characterisation data base for information about timing, accesses to local memory and some more. They are queried dynamically at runtime to allow variation. With these values, the execution time of the individual task is calculated.

Figure 4 shows the general structure of the simulation model of a processing element. The annotated list of tasks is processed in an infinite loop. For each task, the model waits until all data dependencies are fulfilled. Then, it inquires the characterisation data base for the tasks execution time, calculates the according delay, and waits. After the delay has passed, it writes abstract output frames representing the generated data to the ouput channels. A power model is connected to the processing element that tracks in which power state the PE is at what point in time.

Communication channels are modelled in a similar way. For each requested data transmition, they read the abstract data frame, wait for the time required to transmit the data and write the data to the output port. The data frame contains no application data here but only information about the amount of data and its destination.

2) Task Delay Calculation: The execution time of an individual task running in an entire system depends on a number of influencing parameters. Since there are several other tasks running on the platform at the same time, there are penalties due to access to shared ressources like memories, communication links, or cache ressources.

In the characterization phase, the execution time of a kernel is measured when running alone, without interference with other tasks. In addition to the execution time, memory footprint and memory access paterns are analyzed. With this information combined with the current state of the platform, the delay of the individual task is calculated.

At each point in time, the platform is aware of its own state. That is: Which platform elements are currently active and which are not. As mentioned before, the PE model inquires the characterization data base for the execution time of a task when it is supposed to run. If no other task is running at the same time, this value is taken as is. If there are other tasks running, the value is increased by penalties depending on the current platform state and the platform architecture. For instance, if other tasks are running on other cores of the same multicore CPU, a penalty for shared caches is added. If other tasks are running on PEs sharing memory or communication links, other penalties are added. These penalty values are estimated based on the memory footprints and memory access paterns of the running tasks.

3) Simulation Results: The main simulation results are power over time traces. They allow the analysis of the overall energy consumtion and execution time of the system. Moreover, the user is able to see the utilization of each platform element. For now, the traces are generated by the generic tracing facilities integrated in SystemC and SystemC AMS [13]. Each platform element contains a signal representing its current power state and traces of these signal may be put out in form of VCD or tabular files.

V. Evaluation

A. Experimental Setup

To evaluate the presented approach for early power and timing estimation of massively heterogeneous computation platforms, we chose the well known Cholesky algorithm

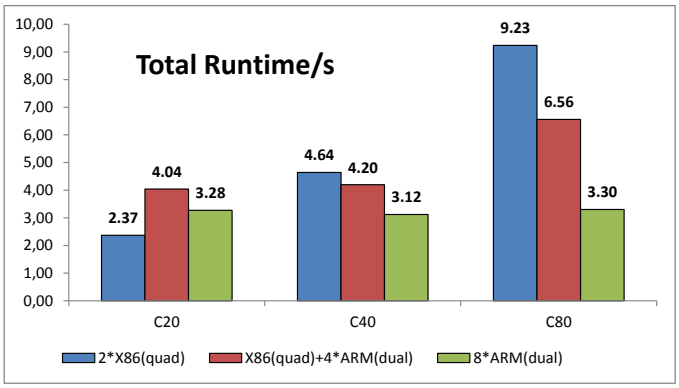

Fig. 5. Total run-time of the different Cholesky partitions on the different hardware platforms

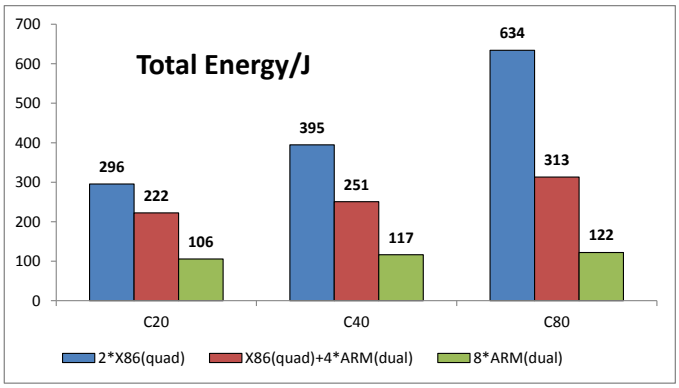

Fig. 6. Total energy consumption of the different Cholesky partitions on the different hardware platforms

already mentioned earlier to decompose a 10,240 by 10,240 matrix.

On multicore systems running a Cholesky decomposition the problem is split in so called tiles so that the decomposition can be run in parallel. To examine the influence of the number of tiles on the overall performance we selected three different partitions as specified in Table I.

TABLE I. USED ALGORITHM PARAMETERS FOR THE EVALUATED CASE STUDY

Algorithm	Number of tiles	Values per tile
C20	400 (20 by 20)	512 by 512
C40	1600 (40 by 40)	256 by 256
C80	5600 (80 by 80)	128 by 128

We further selected three exemplary compute platforms for which the power and timing estimation of the decomposition was done. The three platforms consist of a) two x86 compute boards, each consisting of 4 cores, b) two ARM compute boards, each consisting of 8 cores, and c) a mixture of one x86 and one ARM compute board, consisting of 4 x86 cores and 8 ARM cores. For this case study we simplified the power and calculation performance values of the different types of computation devices according to Table II.

TABLE II. USED HARDWARE PARAMETERS FOR THE EVALUATED CASE STUDY

Processor	Power(idle)	Power(active)	MIPS
x86	7 W	35 W	35000
ARM	0.1 W	2.5 W	7000

B. Experimental Results

Using the presented estimation approach the different variants of the Cholesky decomposition have been analysed in regard to their power and timing behaviour on the specified hardware platforms.

Figure 5 shows the total run-time of the different Cholesky variants needed to decompose the complete 10,240 by 10,240 matrix. The first three bars represent the values of the C20, the middle ones of the C40, and the bars on the right side of the C80 version respectively. The blue bars are the estimations for the homogeneous x86 platform, the red ones for the heterogeneous x86 and ARM platform, and the green ones for the homogeneous ARM platform. As can be seen the run-time varies greatly between the different hardware platforms but also between different partition sizes used. The best combination in regard to the run-time of the decomposition according to the timing estimation is reached by running the C20 algorithm on the homogeneous x86 platform with a run-time of approximately 2.37 s.

Corresponding to Figure 5 the used energy values are shown in Figure 6. Comparing the different energy values it is visible that the homogeneous ARM platform shows the best results in regard to the overall energy consumption and C20 performs slightly better than C40 and C60. All other platforms consume significantly more energy to run the decompositions.

Depending on the given constraints the presented estimation flow therefore enables the user to select the best combination. If run-time is key variant C20 on the homogeneous x86 platform should be used, but at the cost of high energy consumption. The homogeneous ARM platform shows significantly less energy consumption at increased run-time. Here, C40 shows the best run-time result. While run-time is inreased by about 30 % compared to the x86 platform, energy consumption is reduced by more than 60 %. Comparing C40 and C20 on the ARM platform, another 10 % of energy can be spared by accepting a run-time increase of only 5 %.

Another trace which can be generated by the proposed flow is visible in Figure 7. It displays the utilisation of each core of the platform during the run-time of the C40 application and shows when each core is active or idling. A similar trace is available for the communication links in the platform to show the occurrence of congestions during data transfer. These informations are valuable for application designers to optimize applications through adapting the mapping or the structure of the application.

VI. CONCLUSION AND FUTURE WORK

In this paper we presented a new early power and timing estimation approach for high performance computing applications. Combining task graph representations of the applications with statistical power and timing models and a simulation approach used in the embedded world enables design space

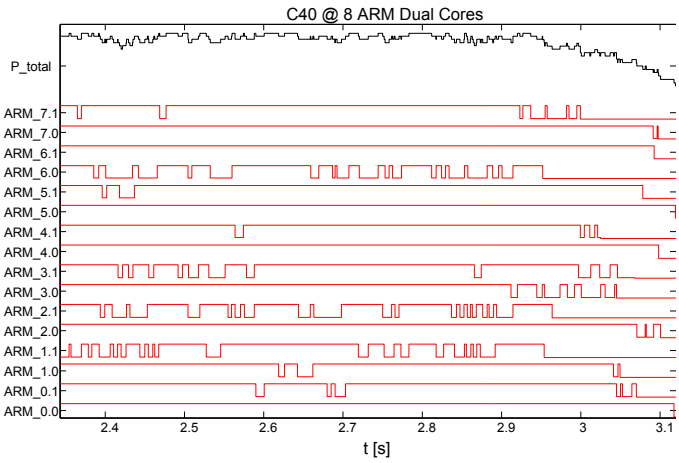

Fig. 7. Utilisation of all ARM cores during execution of the Cholesky 40 by 40 variant

exploration, which is needed to tackle the rising complexity of newly heterogeneous systems. We demonstrated that the presented flow is a usable way to compare different kinds of hardware platforms and algorithms to find the best combination of hardware and software for given design constraints.

During the evaluation we showed that for the given case study a homogenous system consisting of 16 ARM cores would be best fitted for the Cholesky decomposition. This is in contrast to the expected advantage of heterogeneous systems and needs to be investigated further.

The presented estimation flow and evaluation case study gives an early look at the current status of the FiPS project. We are aware, especially with the acquired evaluation results, that there is still a lot of work to be done for a reliable and conclusive estimation. Main focus of our next steps will be the improvement of the interconnect modeling and estimation, and to integrate better constraints into the task mapping & scheduling phase of the design flow.

We also plan to extend our statistical characterization towards a statistical determination of resource conflict probabilities. Additionally, we will try to further optimize the downsampling of the histograms.

We are also working on increasing the level of automation to minimize the necessary user input. For instance, we are currently evaluating together with a project partner to interface the presented flow with their automatic kernel identification approach to simplify and reduce the time spent at this step of the design flow.

REFERENCES

[1] S. Gurumurthi, A. Sivasubramaniam, M. Irwin, N. Vijaykrishnan, and M. Kandemir, "Using complete machine simulation for software power estimation: the SoftWatt approach," in *High-Performance Computer Architecture, 2002. Proceedings. Eighth International Symposium on*, Feb 2002, pp. 141–150. [Online]. Available: http://www.cs.virginia.edu/~gurumurthi/papers/hpca02.pdf

[2] A. Tiwari, M. Laurenzano, L. Carrington, and A. Snavely, "Modeling Power and Energy Usage of HPC Kernels," in *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2012 IEEE 26th International*, May 2012, pp. 990–998.

[3] V. Adve and R. Sakellariou, "Application representations for multi-paradigm performance modeling of large-scale parallel scientific codes," *International Journal of High Performance Computing Applications*, vol. 14, no. 4, pp. 304–316, 2000.

[4] M. Witkowski, A. Oleksiak, T. Piontek, and J. Weglarz, "Practical power consumption estimation for real life HPC applications." *Future Generation Comp. Syst.*, vol. 29, no. 1, pp. 208–217, 2013. [Online]. Available: http://dblp.uni-trier.de/db/journals/fgcs/fgcs29.html#WitkowskiOPW13

[5] D. Shin and J. Kim, "Power-aware scheduling of conditional task graphs in real-time multiprocessor systems," in *Proceedings of the 2003 International Symposium on Low Power Electronics and Design*. New York, NY, USA: ACM, 2003, pp. 408–413.

[6] A. Verma, P. Ahuja, and A. Neogi, "Power-aware Dynamic Placement of HPC Applications," in *Proceedings of the 22Nd Annual International Conference on Supercomputing*, ser. ICS '08. New York, NY, USA: ACM, 2008, pp. 175–184. [Online]. Available: http://doi.acm.org/10.1145/1375527.1375555

[7] S. Benkner, S. Pllana, J. L. Träf, P. Tsigas, U. Dolinsky, C. Augonnet, B. Bachmayer, C. Kessler, D. Moloney, V. Osipov *et al.*, "PEPPHER: Efficient and productive usage of hybrid computing systems," *IEEE Micro*, vol. 31, no. 5, pp. 28–41, 2011.

[8] B. Vanthournout, D. Quaglia, P. A. Hartmann, C. Brandolese, G. Palermo, W. Fornaciari, H. Posadas, F. Herrera, and C. Ykman-Couvreur, "Final report on system simulation and profiling," COMPLEX project deliverable, Tech. Rep. COMPLEX/SNPS/R/D3.1.2/1.1, February 2013. [Online]. Available: http://complex.offis.de/documents/doc_details/55

[9] V. S. Adve and R. Sakellariou, "Compiler Synthesis of Task Graphs for Parallel Program Performance Prediction," in *Proceedings of the 13th International Workshop on Languages and Compilers for Parallel Computing-Revised Papers*, ser. LCPC '00. London, UK, UK: Springer-Verlag, 2001, pp. 208–226.

[10] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, and J. Dongarra, "DAGuE: A generic distributed DAG engine for high performance computing," *Parallel Computing*, vol. 38, no. 1, pp. 37–51, 2012. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167819111001347;http://ash2.icl.utk.edu/sites/ash2.icl.utk.edu/files/publications/2010/icl-utk-417-2010.pdf

[11] E. Vaumorin, B. Vanthournout, S. Bocchio, D. Quaglia, F. Herrera, P. Peñil del Campo, E. Villar, K. Hylla, T. Fandrey, P. A. Hartmann, and K. Grüttner, "Final report and tools on virtual system generation," COMPLEX project deliverable, Tech. Rep. COMPLEX/MDS/R/D2.5.3/1.1, December 2012. [Online]. Available: http://complex.offis.de/documents/doc_details/48

[12] IEEE Computer Society, "IEEE Standard for Standard SystemC Language Reference Manual," *IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005)*, pp. 1–638, Jan. 2012.

[13] SystemC AMS Working Group, "SystemC AMS Extension 1.0," 2010, http://www.accellera.org.