This is the author's version of the work. The definitive work was published in *Workshop Proceedings of the 31st GI/ITG International Conference on Architecture of Computing Systems (ARCS), Braunschweig, Germany, April 09, 2018.*

A Target Platform Description Language for Parallel Code Generation

Christian Schmitt, Frank Hannig, Jürgen Teich
Hardware/Software Co-Design,
Department of Computer Science,
Friedrich-Alexander University Erlangen-Nürnberg

***Abstract*—Today, facilities used for scientific computing are highly parallel and becoming more and more heterogeneous. This trend can be easily seen in the TOP500 list, where an increasing number of systems is equipped with accelerators, such as graphics processor units (GPUs) or many-cores. To achieve the best performance on such machines, special tweaking of the code is necessary, which takes time and expert knowledge of the hardware and corresponding optimization techniques. Domain-specific languages (DSLs) are a remedy to this dilemma by separating the algorithm specification from its implementation, leaving room for optimizations to be applied automatically by the DSL compiler. Thus, the compiler needs to have a profound knowledge of the target platform, e.g., available accelerators and how to program them, details of the network topology to optimize communication patterns, as well as CPU specifications for cache optimizations and vectorization. In this paper, we introduce our approach to modeling hardware and software information to provide platform details that our code generator requires to optimize and emit code for the solution of partial differential equations (PDEs) using the geometric multigrid method.**

I. Introduction and Motivation

Contemporary high-performance computing platforms are highly heterogeneous. While more than 90 % of the supercomputing systems ranked in the TOP500 list (November 2017) run on an x86-based central processing unit (CPU) architecture, there are major exceptions such as the TaihuLight, Sequoia, and K Computer, which use Sunway processors, PowerPCs, and SPARCs, respectively. Each of these technologies has its trade-offs and requires different programming and optimization techniques, which usually vary between vendors and hardware generations.

Beyond this issue, there are more details to consider, such as the network architecture and hardware, versions of operating system or compiler, and specific communication libraries, such as vendor-specific implementations of MPI. Thus, writing programs and optimizing them for a given setting is tedious work that often requires a few months even for an expert who knows all the programming techniques and trade-offs of each platform. If done right, the program will perform comparably on similar hardware. However, in case of any new hardware or platform configuration, the program has to be modified to take advantage of the new features. In many cases, tuning has to be done over and over again—usually by specialized, interdisciplinary teams. Here, domain-specific languages (DSLs) have an advantage. They allow the separation of the description of the algorithm from that of its implementation. Thus, when new hardware must be supported, only one program has to be modified: the DSL compiler. That done, it takes just a recompilation of the unaltered DSL programs to make use of the new features.

In Project ExaStencils[1] [7], it is our goal to improve the level of automatism for code generation in the domain of scientific computing. In previous works, we introduced ExaSlang [18, 19], a DSL for the solution of partial differential equations (PDEs) using geometric multigrid methods, and our approach to emit C++ code from there.

Consequently, to apply the optimal set of transformations to an ExaSlang program, our code generator needs to have a profound knowledge of the target platform. Examples include the selection of a discretization scheme that will perform well on the target platform. For instance, selecting a higher numerical order when discretizing a PDE—depending on the problem—yield a better convergence, but will require more neighboring values to compure a new value. This may exceed certain hardware parameters such as available cache sizes, so that after careful pondering the best solution might be to go with a lower numerical order, but a finer mesh to achieve the best combination of convergence rates and performance on the target platform. Of course, platform knowledge is also crucial for applying low-level optimizations, e.g., the availability of single instruction, multiple data (SIMD) extensions is essential for vectorization, or network topology on distributed-memory parallel systems for selecting the best mesh partitioning and communication scheme.

In this paper, we will present ExaSlang's target platform description language, which we call ExaSlang Platform Description (EPD). We will describe its concepts, how it enables us to model the hardware supported by the ExaStencils code generator, and its run-time application programming interface (API) features to query details on the specified platform.

II. Theoretical Background and Basics

In this section, information on domains touched by this paper, namely domain-specific languages and our language ExaSlang, are given.

A. Domain-Specific Languages

Modeling complex real-world scenarios in a machine-readable form is more natural for domain experts when they

[1] http://www.exastencils.org

can use concepts, objects, and terms of their domain. These abstractions can be provided in a number of different ways. One of the most popular approaches is by implementing a domain-specific language (DSL) [11]. Such a language can be created by extending and/or restricting an existing programming language. Quite often, this *host* language is a general-purpose language. The derived language is called an *embedded*, or *internal*, DSL. Alternatively, it is also possible to create a new language, a so-called *external* DSL, from first principles [2].

The implementation of the DSL compiler or interpreter depends on the type of DSL. Usually, for an internal DSL, the host language compiler is modified whereas, for an external DSL, a new compiler or interpreter must be implemented. Consequently, an external DSL requires a higher implementation effort, but also provides greater freedom in the choice of syntax and semantics.

B. ExaSlang

ExaSlang is a DSL that aims at the solution of PDEs using the multigrid method [4, 22] as the fundamental approach. It tries to provide suitable abstractions for various groups of users, corresponding to the different steps needed to solve a PDE: specification, discretization, selection of appropriate multigrid components, and finally implementation of the solver. Thus, ExaSlang is not a monolithic language, but a hierarchy of four different language layers [18, 19].

At the highest layer, a continuous equation with a corresponding computational domain and boundary conditions can be specified. ExaSlang 2 is one refinement step more concrete. It enables users to specify (or, if generated automatically, modify) discretized versions of the equations at layer 1. ExaSlang 3 exposes the underlying multigrid algorithm by allowing its components to be specified. Finally, ExaSlang 4 is the most concrete layer of the ExaSlang hierarchy. It enables the full specification of a multigrid algorithm and exposes aspects of the parallelization and communication for modification by the user.

We implemented ExaSlang as an external DSL, as only this approach allowed us to tailor each level towards the designated user groups optimally. Opting for an external DSL obviously has the drawback of higher efforts not only on the language implementation side but also on the users' side, who have to learn a entirely new language.

In Figure 1, an excerpt of an ExaSlang 4 application to solve Poisson's Equation is depicted. We already see a number of ExaSlang features, for instance repition loops (line 2) and field loops (line 3). A crucial feature are *fields*, which represent discretized mathematical variables, or from the computer's point of view, arrays that store the numerical values. A language feature stemming from ExaSlang's designated base algorithm, the multigrid method, are *level specifications*. Multigrid methods represent the data on different grid sizes, and continously step through this level hierarchy. Level specifications allow the specialization of program parts for certain multigrid levels, e.g., variable declaration and accesses or function definitions and corresponding calls. In the example code, only references to the current level's grid are made using `@current`. There exist specialized keywords to access the grids of neighboring levels (`@finer` and `@coarser`), but it also possible to reference the bottom-most and topmost grid in the hierarchy, use lists of level specifications, and so on. Another feature are *slots* (lines 4 and 6) that add a ring-buffer like interface to fields, superseding the need to declare identical fields when alternating accesses are needed, as is the case for Jacobi iterations. In our example, the currently active slot is shifted using the keyword **`advance`**. Accesses to the current or the next slot is specified with the corresponding keywords `activeSlot` and `nextSlot`. In the rest of this paper, we will use this code snippet to illustrate the benefits of EPD for our code generation process. For a more detailed description, we refer to [18].

III. Related Work

A lot of previous work has been put into the research of the modeling of computing platforms. However, for many reasons, they do not fulfill our requirements: First of all, most languages provide a level of abstraction that is not suitable for our purposes: We require the description of a system from a mainly architectural point of view, as we need to check if certain technology is available to make use of it. For example, we need to know which set of vectorization instructions is available, but do not need to know details of each instruction. Secondly, many languages serve other purposes, e.g., for the description of systems for synthesizing virtual models. Thus, they have stricter constraints and often require technical details that are hard to find out about existing systems. As a consequence, most languages also do not provide a flexible and powerful interface to query the system specification in a manner comparable to databases.

VHDL [14] and Verilog [21] are popular hardware description languages that instantly come into minds when talking about languages for the description of hardware, as they are used for the specification of hardware implementations in field-programmable gate arrays (FPGAs) or as integrated circuits. They work on the register-transfer level (RTL) level and are processed by synthesis programs for simulation or implementation of a concrete piece of hardware. As such, their level of abstraction is immensely too detailed for our needs. Note our nomenclature: In this paper, we chose to use the term *platform description* to describe the components (hard- and software wise) of a system, not their internal structure.

In the context of the co-design of processors and corresponding tools, a vast body of work in the area of architecture description languages (ADLs) exists and serves as a basis for architecture design (i.e., hardware description) as well as the generation of retargetable compilers and simulators [12]. Our approach is at a much higher level of abstraction (e.g., single instructions of a processor are not modeled) and deals with the modeling of up to an entire cluster, including memory hierarchy and communication topology. For an exhaustive list of languages for the structural or behavioral description of computing platforms, we refer to *Architectural Languages Today* [8].

```
1 Function Smoother() : Unit {
2   repeat 5 times {
3     loop over Solution@current {
4       Solution[nextSlot]@current = Solution[activeSlot]@current + (0.8 / diag(
      Laplace@current) * (RHS@finest - Laplace@current * Solution[activeSlot]@current))
5     }
6     advance Solution@current
7   }
8 }
```

Fig. 1. Specification of a Jacobi smoother/solver for Poisson's Equation in ExaSlang 4

In the following, we want to present two approaches that come close to the work presented in this paper:

PDL [17] has been designed for the description of larger electronic systems, such as complete embedded systems, which typically consist of a number of different (ready-made) components. It uses eXtensible Markup Language (XML) for their underlying structure. As such, it enforces a hierarchical view of the system, which is adequate for most of today's computing systems. It divides the systems into processing units, memory regions, and interconnects. For the processing units, it features a master-worker connotation, aiming at offloading computational tasks. Available software is modeled as key-value pairs. A major concern of PDL is weak modularity, impairing reusability severely. Furthermore, it does not feature a run-time interface to query platform details.

This led to the creation of XPDL [6], an improved version of PDL that tries to resemble the described hardware already in its structure to overcome limitations stemming from PDL's rigid master-slave concept. Furthermore, this also enables improvements such as the possibility to annotate energy consumption to hardware components. It increases reusability of individual components due to modularization. However, no implementation is currently freely available. XPDL has stronger constraints concerning the omission of parts of the hardware, as it was designed for the design of systems for the purpose of simulation and synthesizing them. In ExaStencils, we require a less rigid system, as we need to describe already existing hardware, and quite often, some information is either not necessary for our purposes or specifications are hard or even impossible to find.

Of course, many other DSL compilers also face the issue of requiring platform information. In the following, we will present a small selection of approaches related to ExaStencils and ExaSlang:

From a Fortran-based description of stencils on regular grids, PATUS [1] generates vectorized C implementations that are shared-memory parallelized using pthreads. Its compiler uses a list of SIMD extensions that contains mappings from vectorizable arithmetic operations to technical details, such as function names and expected vector sizes. As PATUS optimizes on a kernel basis, it does need to have information about network details or accelerators.

SPIRAL [15] is a DSL to describe linear transforms and other mathematical functions, with a focus on signal processing kernels. Generated codes do not use distributed-memory parallelization. Therefore, the description of clusters and networks is not needed. However, SPIRAL does support code generation for GPUs using OpenCL.

SDSLc [16] is able to apply a large range of optimizations on stencil computations specified in a custom syntax and embedded in C, C++, and MATLAB. It can generate code for CPUs, GPUs, and emit accelerator descriptions for FPGAs. SDSLc does not apply distributed-memory parallelization. Hence, the description of such systems is not needed.

Physis [10] is a stencil DSL that targets CPU-GPU clusters using MPI. Rather than composing a good solution from the given hardware description before generating the code, as ExaStencils does, it employs an auto-tuning approach to find the best combination of parameters for the provided hardware.

IV. ExaSlang Platform Description

In the ExaSlang Platform Description(EPD), hardware components are represented by a number of specialized XML tags (see Section IV-C), with rules defining how these tags may be combined. Since EPD is a structural, not a behavioral description of platforms, and it is not sensible to describe hardware that our code generator does not support, it is sufficient to resort to a fixed set of components. Wherever appropriate, properties are modeled by enumerating them in key-value pairs to ease parsing and evaluation of the supplied information. One example for these tradeoffs are CPUs: For every platform supported by ExaStencils, a C++ compiler has to be available; thus, we do not need to know about the target's instruction set or its endianness. However, we need to know about available SIMD extensions to generate appropriate vectorized code. For ExaStencils, a simple enumeration is sufficient, i.e., we do not need to know about the construction of vector units or registers.

EPD has a hierarchical structure, resembling common hardware architectures. To ease the specification of systems by enabling the reuse of specified components, EPD supports the division into templates and their instantiations. The first are usually used to describe individual components of the overall architecture, optionally with certain parameters that need to be supplied during instantiation. We introduced this division to increase modularity, and thus, reusability of platform component descriptions. In the following subsections, we will present both parts of EPD and their interplay to describe target platforms for ExaStencils.

A. Templates

In every EPD description, a number of templates should be employed. They can be seen as templates, or classes, which are combined to form a concrete description of a system. In our XML notation, template are introduced by specifying the attribute `name`. To improve template reuse, they may carry placeholders where corresponding values need to be specified when instantiating in a concrete model. A typical use case is the specification of memory sizes or clock rates: For one CPU family, key performance figures such as cycles per instruction (CPI) are all the same. What differs between different concrete CPU models are the number of cores and their clock rates, and cache sizes. These values may be set when instantiating the template. Naturally, templates may reference other templates.

B. Concrete Models

Concrete models are the starting point when parsing and evaluating a platform description. When templates are seen as classes, concrete models resemble their instantiations. Thus, the definition of a concrete model should (but does not have to) reference a number of templates. An instantiation is invoked using the XML attributed `type`, followed by the corresponding value in the template's `name` attribute. By defining appropriate `param` tags, values for the placeholders are set. Additionally, for numeric values, units may be given. These will be automatically converted to the correct unit and magnitude, e.g., a certain amount of GibiBytes will be converted to kilobytes if the component's parameter has been specified to use that base unit. Naturally, this is feasible only for compatible units.

C. Description of Platform Components

A number of XML tags are allowed to describe the different aspects of a system, be it a hardware component or available software.

The `group` element is a special element as it does not directly correspond to a concrete hardware component. It is used every time the hardware component described by its child node is available multiple times. Naturally, in a cluster, this is the case for racks, which in turn are made from several nodes. For a group element, the quantity attribute needs to be specified. Optionally, energy consumption for the whole group may be specified, e.g., when the data is not available for a single element. As a bonus, this speeds up evaluation since energy consumption of sub-elements does not need to be computed.

For the description of platform components, specific elements are available:

- `system`: This is the root element for every concrete model described in EPD.
- `cluster` marks the system as being distributed-memory parallel.
- `interconnect` is a very important element that connects the various hardware elements using point-to-point specifications. It features the specification of bandwidth and latency for the creation and evaluation of performance models.
- `node` is used to specify that the description of a single node in a cluster follows. Hence, it can only be used in a platform description that contains a `cluster` element.
- `cpu` is used to declare a CPU, and needs to consist of at least one `core` tag.
- `core` is used to describe the cores of a CPU. Usually, all cores in a CPU are homogeneous. However, there are also heterogeneous CPUs such the ARM big.LITTLE architectures, hence we introduced this element.
- `memory` is used to describe available memories, e.g., random access memory (RAM).
- `cache` describes the size, associativity, and bandwidth of a single cache. To describe a cache hierarchy between CPU and RAM, multiple `cache` elements need to be connected by `interconnect` elements.
- `workingunit` is used to describe any additional hardware that is available and can be used for computations. Depending on its `programming_model`, currently CUDA, OpenCL, or OpenMP, a number of other parameters have to be specified. For CUDA, one example is the GPU's compute capability.

For most of these tags, energy consumption may be specified by using the `power` attribute.

However, to efficiently generate efficient code, not only the target platform's hardware is of interest: The software is equally important. Thus, we introduced the `software` tag to allow the specification of available third-party libraries and other software. Apparently, the most basic piece of software information is the system's operating system. Furthermore, available target compilers are of particular importance, as certain workarounds may be necessary not to trigger bugs or achieve the optimal performance. The situation is comparable for libraries. Sometimes, competing libraries that would achieve the same goal are available, but for some reasons, it might be desirable to select one instance over the other. One use-case are MPI implementations, e.g., on TSUBAME 3.0 it is possible to choose between OpenMPI and Intel MPI.

D. Examples

In Figure 2, an excerpt from the definition of an IBM Bluegene/Q cluster is presented. Already from this small excerpt, we can derive a number of facts important for our code generation process: The `group` tag in combination with the `quantity` attribute, embedded in a `cluster` tag, tells us that we are generating code for a distributed-memory parallel system, thus our parallelization will need to use MPI[2]. As a consequence, the application's field data will be partitioned across the MPI ranks and statements such as the **`loop over`** in line 3 of Figure 1 will be split into corresponding loops over the local partitions and communication statements.

Note that each node in the EPD description consists of one CPU, which has been defined by instantiating the corresponding

[2]While other parallelization approaches such as GPI [3] or HPX [5] might be also possible, our code generator only supports MPI at the moment. As we strive to increase automation in code generation, we see the selection of the technology as a compiler decision, not a user decision.

```
1  <system id="JUQUEEN">
2    <cluster>
3      <group prefix="rack" quantity="28">
4        <group quantity="1" power="70" power_unit="KiloWatt">
5          <node>
6            <socket>
7              <cpu type="IBM_PowerPC_A2" />
8              <memory type="DDR3_Mem" />
9            </socket>
10           ...
11   <software>
12     <hostOS type="linux" />
13     <compiler type="ibmxl" version_major="15" version_minor="0" />
14     <compiler type="gcc" version_major="4" version_minor="8" />
15   </software>
16   ...
```

Fig. 2. Excerpt from the example definition of the JUQUEEN cluster

template. This definition is depicted in Figure 3. Inside this template, the L1 and L2 caches refer to other templates. Concrete cache sizes are given as a parameter. Different units supplied within the `param` tag will be converted automatically where feasible: Here, KiloByte (line 6) and MebiByte (line 9) will be adjusted to the base unit defined in the CPU's template. Another example of parameters is the specification of the interface from the CPU's cache hierarchy to the memory. In line 13 of Figure 3, a parameter `l2_tail` has been defined and is used in line 14. By prefixing the name with the `$` operator, this value has been defined to refer to a parameter of the same name one level above (in this case the `cpu` tag).

From the group declaration inside the cpu block (line 6 of Figure 3), we know that the target machine also offers shared-memory parallelism. In combination with the compilers specified as available, namely the IBM XL compiler suite as well as the GNU GCC compilers (Figure 2, lines 13 and 13), we know that generation of OpenMP-parallel code is also possible. However, the decision which parallelization strategy to apply, i.e., pure MPI parallelization or hybrid MPI-OpenMP, is left to the compiler. In case of transformation Figure 1, the grid partitioning may be adapted to a single partition per node instead of one per CPU, and appropriate pragmas may be added to the loop over the grid elements.

However, information about available compilers will also be used to implement workarounds in the generated code. Examples can be found for nearly any of the supported compilers: For IBM's compiler suite, use of initializer lists will be disabled, whereas CLANG provides support for OpenMP only starting with version 3.6. The Microsoft Visual Studio Compiler uses a different keyword to specify alignment of variables, and GCC requires specification of `std::move` at certain occasions. Additionally, compiler flags for optimization levels, specification of target CPU architecture, use of OpenMP, etc vary between compiler vendors.

Finally, in line 12, the system's operating system was defined to be Linux. In combination with knowledge of available compilers, our code generator will generate Makefiles instead of, i.e., Visual Studio project files.

V. Run-time Query Interface

An accurate description of the target platform is worthless for our code generator when the desired information cannot be extracted. Thus, we designed and implemented a flexible and powerful query interface. For a number of reasons, it has been implemented in Scala [13]: First and foremost, our ExaStencils code generator is written in Scala [20]. Obviously, this makes interfacing much more comfortable than using a library that needs to be natively compiled. Currently, our code generator is used on all three major operating systems: Linux, Windows, and macOS. Thus, bringing improvements in EPD's query interface to the ExaStencils code generator would come with the overhead of compiling and linking it statically for all these platforms. Another reason is the object-functional nature of Scala: By a clever combination of features such as higher-order functions (basically, functions work on other functions), we can define syntax and semantics that are essentially a Scala-embedded DSL. This is supported by the use of the excellent scalaz[3] library, an extension to the Scala standard library bringing many features from Haskell [9].

A. Types

For the representation of components, we use a mixture of dynamic and static data structures: Specialized types are used for elemental components (see Section IV-C). However, all their attributes, and child nodes, are saved in an abstract and dynamic way. Compared to purely static data types, i.e., carrying all possible attributes as class members, we do not need to modify any code when introducing new attributes. This approach has the disadvantage of reduced type safety, however, we can introduce constraints to the model, e.g., require certain dependencies or cardinalities of child nodes, and apply a verification step to ensure formal model correctness.

Usually, the result of queries are integral Scala types, e.g., integers for cardinalities, or Boolean values for availability

[3]https://github.com/scalaz/scalaz

```
<cpu name="IBM_PowerPC_A2">
  <group quantity="18" id="a2_cores">
    <core frequency="1.6" frequency_unit="GHz" power="60" power_unit="Watt" />
  </group>
  <cache level="1" type="cache_with_linesize">
    <param name="size" value="16" unit="KiloByte" />
  </cache>
  <cache level="2" type="cache_with_linesize">
    <param name="size" value="32" unit="MebiByte" />
  </cache>
  <interconnect head="a2_cores" tail="l1" />
  <interconnect head="l1" tail="l2" />
  <param name="l2_tail" value="?" />
  <interconnect head="l2" tail="$l2_tail">
  <param name="FPU_flops" value="12.8" unit="GigaFlops" />
  ...
</cpu>
```

Fig. 3. Excerpt from the definition of an IBM BlueGene/Q CPU, as used in the JUQUEEN supercomputer

```
// list of all working units
val workers = predefinedQuery(epdTree,
    GetWorkingUnits)

// total energy consumption
val power = predefinedQuery(epdTree,
    PowerInWatts)
```

Fig. 4. Call to predefined queries to get the total energy consumption of a (sub-) system and references to all working units

```
val q = lowLevelQuery(epdTree,
  (y : EPDValue , x : List[String]) => y
    match
  {
    case wu : WorkingUnit => wu.safeGetValue
    ("id").head :: x
    case _ => x
  }
)
```

Fig. 5. Call to a user-defined query for all working units' id in a system

checks. However, it is also possible to receive a list of results, e.g., the list of vector extensions provided by a CPU. When querying for a value accompanied by a unit, such as energy consumption, the designated magnitude and unit may be specified for automatic conversion.

B. Queries

For a number of common tasks, such as the determination of energy consumption of (components of) a system, or checking the availability of certain elements, predefined queries may be used, as depicted in Figure 4. In this example, first all worker units, i.e., CPUs and accelerators, are extracted from the platform description. Secondly, total energy consumption for the platform is computed. The function `PowerInWatts` has been defined to use Watt as the base unit, so all values given in other magnitudes will be converted automatically.

However, the standard interfacing method to gather information about the platform is via custom queries, that is, queries specified in a DSL-like syntax collecting the necessary information. Often, it is important to find a certain element in the platform description tree. By using the `find` operation, the first matching EPD object, or, more precisely, the entire matching subtree, is returned and can be used for further queries. Otherwise, the Scala object `None` is returned. Of course, it also possible to return a list of all matching objects by using `findAll`. These lookup operators take predicates, according to which objects are evaluated. To ease specification, comparison operators for scalar values and ranges are provided. EPD objects support many other operations, such as group operations to find the minimum, maximum, average or median values of certain attributes. As the returned result usually is an EPD subtree, our interface also offers a variety of tree manipulation operations: Trees can be folded or unfolded, which means that quantities are either expressed using the numeric value and a single object, or are duplicated the corresponding times. While the former is sufficient to compute the sum of values, e.g., the total memory available, the latter allows to further refine the selection and, for example, select only two cluster nodes.

In Figure 5, a custom query to get a list of all working units that have been defined in a system is depicted. As can be seen, we make heavy use of Scala's pattern matching features. Pattern matching is a compelling feature that provides a wealth of options: For instance, data structures may be matched by their type. In the example, whenever a data structure of type `WorkingUnit` is encountered when traversing the system tree, it is assigned to the variable `wu`. Then, the code on the right-hand side of the `=>` operator is executed. In this case, the working unit's id (if available) is prepended to the list of all available ids.

Using a combination of predefined and custom queries, we can gather the information needed to generate code tailored especially towards the system given in EPD. This can be simply achieved by specifying an EPD object as the data source in the

```
1 val workers =
2   predefinedQuery(epdTree, GetWorkingUnits)
3 val workers50 =
4   workers.filter(w => w.power > Watts(50))
5 workers50.foreach(System.out.println(_))
```

Fig. 6. Combination of predefined and custom query to output all working units consuming more than 50 watts

`predefinedQuery` or `lowLevelQuery` function, instead of the root `epdTree`. A simple example outputting all workers' names is depicted in Figure 6. Here, a predefined query is used to find all working units. Then, the returned list is filtered to only include units consuming more than 50 watts. Note that for proper comparison, the `Watt()` function has to be used to supply a unit to the numeric value. Finally, all remaining working units are printed to the user.

VI. Future Work

All of our current requirements can be modeled using EPD. Future developments of the language might bring support for the modeling of energy consumption not as a single fixed value, but as functions that depend on the actual operations involved, i.e., arithmetic and logic operations, and memory transfers, and hardware-dependent parameters. For the latter, the main problem is the gathering of reliable data. Let us take CPU power dissipation as an example. One part of the total energy consumption is power loss because of transistor leakage currents, i.e., loss of energy from charged capacitors. Among others, this is influenced by the capacitors' temperatures, which in turn are influenced by countless other factors.

Writing a detailed hardware description is cumbersome, even with appropriate templates for some components already available. Thus, we envision an automatic tool that runs on the target platform and is able to extract most of the information automatically. Of course, it is almost impossible to get all relevant data, e.g., in a cluster, when the program is run on a single node only. However, it would form a good starting point where a user or system operator just needs to fill in the missing pieces, such as the total node number. To continue that idea, a wizard-like program could try to gather as much information as possible step by step, asking the user to fill in crucial data that could not be collected automatically.

VII. Conclusions

In this paper, we have introduced our motivation for a detailed description of target platforms targeted by ExaStencils. After carefully checking previous works, we decided to design and implement our own description language and an accompanying run-time query interface. Description of platforms using EPD is split into abstract descriptions of components, and their instantiation to a concrete model. To provide an easy to understand and use, yet powerful interface to query the platform description, we created an interface building upon Scala's object-functional properties. Thus, we can describe all platforms of interest to Project ExaStencils, and gather information required to apply appropriate optimizations.

VIII. Acknowledgments

This work is supported by the German Research Foundation (DFG), as part of the Priority Programme 1648 "Software for Exascale Computing" in project ExaStencils under contract number TE 163/17. We thank Sven Wille for his contributions to the EPD implementation.

References

[1] M. Christen, O. Schenk, and H. Burkhart. "PATUS: A Code Generation and Autotuning Framework for Parallel Iterative Stencil Computations on Modern Microarchitectures." In: *Proc. 25th IEEE Int'l Parallel & Distributed Processing Symposium (IPDPS)*. IEEE. May 2011, pp. 676–687.

[2] M. Fowler. *Domain Specific Languages*. 1st. Addison-Wesley Professional, 2010.

[3] D. Grünewald and C. Simmendinger. "The GASPI API specification and its implementation GPI 2.0." In: *Proc. 7th Int'l Conference on PGAS Programming Models*. Vol. 243. Edinburgh, UK, 2013, pp. 243–248.

[4] W. Hackbusch. *Multi-Grid Methods and Applications*. Springer-Verlag, 1985.

[5] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, and D. Fey. "HPX: A Task Based Programming Model in a Global Address Space." In: *Proc. 8th Int'l Conference on Partitioned Global Address Space Programming Models*. PGAS '14. Eugene, OR, USA: ACM, 2014, 6:1–6:11.

[6] C. Kessler, L. Li, A. Atalar, and A. Dobre. "An Extensible Platform Description Language Supporting Retargetable Toolchains and Adaptive Execution." In: *Proc. 44th Int'l Conference on Parallel Processing Workshops (ICPPW)*. Beijing, China: IEEE, 2015, pp. 51–60.

[7] C. Lengauer, S. Apel, M. Bolten, A. Größlinger, F. Hannig, H. Köstler, U. Rüde, J. Teich, A. Grebhahn, S. Kronawitter, S. Kuckuk, H. Rittich, and C. Schmitt. "ExaStencils: Advanced Stencil-Code Engineering." In: *Euro-Par 2014: Parallel Processing Workshops*. (Porto, Portugal). Vol. 8806. Lecture Notes in Computer Science. Springer, Aug. 25–29, 2014, pp. 553–564.

[8] I. Malavolta, P. Lago, H. Muccini, P. Pelliccione, and A. Tang. *Architectural Languages Today*. Nov. 2017. URL: http://www.di.univaq.it/malavolta/al/.

[9] S. Marlow, ed. *Haskell 2010 Language Report*. July 2010.

[10] N. Maruyama, T. Nomura, K. Sato, and S. Matsuoka. "Physis: An Implicitly Parallel Programming Model for Stencil Computations on Large-scale GPU-Accelerated Supercomputers." In: *Proc. 2011 Int'l Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE. Nov. 2011, 11:1–11:12.

[11] M. Mernik, J. Heering, and A. M. Sloane. "When and How to Develop Domain-specific Languages." In: *ACM Computing Surveys* 37.4 (Dec. 2005), pp. 316–344.

[12] P. Mishra and N. Dutt. *Processor Description Languages*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008.

[13] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala*. 2nd ed. artima, 2011.

[14] D. Perry. *VHDL*. Computer Engineering Series. Mcgraw-Hill Book Comp., 1991.

[15] M. Püschel, J. M. F. Moura, J. R. Johnson, D. A. Padua, M. M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. "SPIRAL: Code Generation for DSP Transforms." In: *Proc. IEEE* 93.2 (2005), pp. 232–275.

[16] P. Rawat, M. Kong, T. Henretty, J. Holewinski, K. Stock, L.-N. Pouchet, J. Ramanujam, A. Rountev, and P. Sadayappan. "SDSLc: A Multi-target Domain-specific Compiler for Stencil Computations." In: *Proc. 5th Int'l Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing (WOLFHPC)*. ACM, Nov. 2015, 6:1–6:10.

[17] M. Sandrieser, S. Benkner, and S. Pllana. "Using Explicit Platform Descriptions to Support Programming of Heterogeneous Many-core Systems." In: *Parallel Comput.* 38.1-2 (Jan. 2012), pp. 52–65.

[18] C. Schmitt, S. Kuckuk, F. Hannig, H. Köstler, and J. Teich. "ExaSlang: A Domain-Specific Language for Highly Scalable Multigrid Solvers." In: *Proc. Int. Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing (WOLFHPC)*. IEEE Computer Society, 2014, pp. 42–51.

[19] C. Schmitt, S. Kuckuk, F. Hannig, J. Teich, H. Köstler, U. Rüde, and C. Lengauer. "Systems of Partial Differential Equations in ExaSlang." In: *Software for Exascale Computing – SPPEXA 2013–2015*. Ed. by H.-J. Bungartz, P. Neumann, and W. E. Nagel. Vol. 113. Lecture Notes in Computational Science and Engineering (LNCSE). Springer, pp. 47–67.

[20] C. Schmitt, S. Kuckuk, H. Köstler, F. Hannig, and J. Teich. "An Evaluation of Domain-Specific Language Technologies for Code Generation." In: *Proc. 14th Int'l Conf. on Computational Science and its Applications (ICCSA)*. IEEE Computer Society, July 2014, pp. 18–26.

[21] D. Thomas and P. Moorby. *The Verilog Hardware Description Language*. Springer Science & Business Media, 2008.

[22] U. Trottenberg, C. W. Oosterlee, and A. Schüller. *Multigrid*. Academic Press, 2001.