Proc. 2015 44th International Conference on Parallel Processing Workshops (ICPP-EMS15), Beijing, 1-4 Sep. 2015. ©IEEE.

DOI: 10.1109/ICPPW.2015.17

XPDL: Extensible Platform Description Language to Support Energy Modeling and Optimization

Christoph Kessler*, Lu Li*, Aras Atalar† and Alin Dobre‡
*Linköping University, Linköping, Sweden. Email: firstname.lastname@liu.se
†Chalmers Technical University, Gothenburg, Sweden. Email: aaras@chalmers.se
‡Movidius Ltd, Dublin, Ireland. Email: firstname.lastname@movidius.com

***Abstract*—We present XPDL, a modular, extensible platform description language for heterogeneous multicore systems and clusters. XPDL specifications provide platform metadata about hardware and installed system software that are relevant for the adaptive static and dynamic optimization of application programs and system settings for improved performance and energy efficiency. XPDL is based on XML and uses hyperlinks to create distributed libraries of platform metadata specifications. We also provide first components of a retargetable toolchain that browses and processes XPDL specifications, and generates driver code for microbenchmarking to bootstrap empirical performance and energy models at deployment time. A C++ based API enables convenient introspection of platform models, even at run-time, which allows for adaptive dynamic program optimizations such as tuned selection of implementation variants.**

I. Introduction

The EU FP7 project EXCESS (www.excess-project.eu) aims at providing a generic framework for system-wide energy optimization. In particular, its ambition is to finally have a *retargetable* optimization framework that can handle a significant variety of systems, ranging from embedded systems such as the Movidius MV1.53 card via CPU and GPU servers up to clusters of GPU-equipped compute nodes. For retargetability, the framework must be parameterized in a formal description of the (optimization-relevant) properties of the given platform. A *platform* should be considered in the most general sense, i.e., it includes not only the hardware with its various components, but also installed system software such as libraries, programming models, compilers etc. Accordingly, a platform description language should allow to express both hardware and software entities and relations between them. Platform description languages must be formal enough to be processed automatically (by tools reading or modifying platform information) but should ideally also be readable and editable by humans, possibly with the help of tool support.

A wide variety of platform modeling languages, often referred to as architecture description languages, have been proposed in the literature, at various levels of granularity, detail, and purpose, often for single-processor architectures. Some are intended as a source language for hardware synthesis or for simulator generation, others are geared towards automatic generation of program development tool chains including compilers, assemblers and linkers. See Section V for a short survey of related work and references.

In the EU FP7 project PEPPHER, which was a predecessor of EXCESS focusing on performance rather than energy optimization, a platform description language called PDL (PEPPHER platform description language) [1] has been developed and was used in the PEPPHER composition tool [2] for conditional composition [3]. The PDL design contains a number of limitations, which we will elaborate on in Section II in order to motivate why we decided not to adopt PDL in EXCESS but to develop a new design for a more modular platform description language for EXCESS, called XPDL.

In this paper we present the design of XPDL, a modular, extensible platform description language for heterogeneous multicore systems and clusters. For pragmatic reasons (tool support), we adopt XML as primary syntax for XPDL, but would like to emphasize that this syntactic appearance is not the key point for its applicability. Some unique features of XPDL include its support for modularity and the use of hyperlinks to create distributed libraries of platform metadata specifications, the inclusion of metadata for microbenchmarking code that can be used to automatically derive parameters of performance and energy models at deployment time, and the possibility to introspect (parts of) the platform model in a running application, which allows for customized, adaptive dynamic program optimizations such as tuned selection of implementation variants. We also provide first components of a retargetable toolchain that browses and processes XPDL specifications.

The remainder of this paper is organized as follows. After discussing PDL and its limitations in Section II, we introduce the XPDL design in Section III and show how it can be used to build platform models of target systems, ranging from HPC clusters down to embedded processors and their components. Due to lack of space, extensive example models for concrete systems are omitted and can be found in a technical report [4]. The XPDL toolchain and its current implementation status are described in Section IV. Section V discusses related work, and Section VI concludes.

II. A review of PEPPHER PDL

PDL, described by Sandrieser et al. [1], has been developed in the EU FP7 project PEPPHER (2010-2012, www.peppher.eu), as a XML-based platform description language for single-node heterogeneous systems, to be used by tools to guide performance prediction, automated selection of implementation variants of annotated application software components (”PEPPHER components”), or task mapping. In particular, it establishes naming conventions for the entities in a system such that they can be referred to by symbolic names. Beyond that, it is also intended to express so-called *platform patterns*, reusable templates for platform organization that could be instantiated to a complete platform specification.

PDL models the main architectural blocks and the hierarchical execution relationship in heterogeneous systems. All other entities (e.g., installed software) are modeled as free-form properties in the form of key-value pairs.

The *main architectural blocks*, which distinguish between *types* and *instances*, include
– *Processing Units* (PU), describing the processing elements in a heterogeneous machine;
– *Memory regions*, specifying data storage facilities (such as global/shared main memory, device memories); and
– *Interconnect*.
For PDL example specifications see [1] or [4, Sec. 4.1].

A. Control Relation

The overall structure of these hardware components descriptions in a PDL specification is, however, not organized from a hardware perspective (i.e., the structural organization of the hardware components in the system) but follows the programmer perspective by formalizing the *control relation* between processing units as a logic tree consisting of so-called Master, Hybrid and Worker PU. There must be one *Master PU*, a feature-rich general purpose PU that marks a possible starting point for execution of a user program, thus acting as the root of the control hierarchy; there are a number of *Worker PU*s that are specialized processing units (such as GPUs) that cannot themselves launch computations on other PUs and thus act as leaves of the control hierarchy; and *Hybrid PU*s that can act both as master and worker PU and thus form inner nodes in the control hierarchy. A system might not contain all three types of PUs (i.e., control roles) simultaneously. For instance, a standard multicore server without accelerators has no workers, and the Cell/B.E., if used stand-alone (i.e., not coupled with a host computer acting as Master), has no hybrid PUs.

In practice, which PU is or could be host and which one is a device depends not only on the hardware but mainly on the programming model used, which might be defined by the compiler, runtime system and/or further system layers. The PDL control relation usually reflects the system's native low-level programming model used (such as CUDA).

The actual workings of the launching of computation modeled by the control relation cannot be explicitly specified in PDL. In practice, this aspect is typically hardcoded in the underlying operating system scheduler or the heterogeneous runtime system, making the specification of a unique, specific Master PU questionable, for instance in a dual-CPU server.
Discussion The motivation of emphasizing the control role in PDL was to be able to define abstract platform patterns that can be mapped to concrete PUs. However, we find that using the control relation as the overarching structure of a platform description is not a good idea, especially as the control relationship may not always be fixed or even accessible to the programmer but might, for instance, be managed and hardcoded in the underlying heterogeneous runtime system.

In particular, from the EXCESS project point of view, it makes more sense to adopt a hardware-structural organization of platform descriptions, because power consumption and temperature metrics and measurement values naturally can be attributed to coarse-grain hardware blocks, such as CPU, memory or devices, but not to software roles. Most often, the software roles are implicitly given by the hardware blocks and/or the underlying heterogeneous runtime system, hence a separate role specification is usually not necessary.

Hence, we advocate a language structure that follows the hardware composition of the main architectural blocks, that is also more aware of the concrete architectural block type (e.g., that some hardware component is actually a GPU), and that allows to optionally model control relations separately (referencing the involved hardware entities) for complex systems where the control relation cannot be inferred automatically from the hardware entities alone. This should still allow the definition of abstract platform (i.e., generic control hierarchy) patterns, but rather as a secondary aspect to a more architecture oriented structural specification. Where appropriate, the control relation could also be specified as a feature of the runtime system rather than the architectural structure itself.

In the PEPPHER software stack, the mechanisms and constraints of launching tasks to other PUs are managed entirely by the runtime system (StarPU [5]); as a consequence, the control relation information in PDL specifications was not needed and not used by the PEPPHER tools. If suitably complemented by control code specification, it might rather become relevant in scenarios of generating code that runs directly on the bare hardware without a managing runtime system layer in between.

B. Interconnect Specification

The interconnect specification in PDL is intended to model communication facilities between two or more PUs, in a style similar to xADML [6] inter-cluster communication specifiers. In PEPPHER this information was not used, as the communication to and from an accelerator is managed completely within the PEPPHER runtime system, i.e., StarPU.

C. Properties Concept

The possibility of specifying arbitrary, unstructured platform properties (e.g., installed software) as *key-value pairs*[1] provides a very flexible and convenient extension mechanism in PDL, as long as the user(s) keep(s) control over the spelling and interpretation of introduced property names. Properties can also be used as placeholders if the actual values are not known yet at meta-data definition time or can change dynamically, e.g. due to different possible system configurations. The existence and, where existing, values of specified properties can be looked up by a basic query language.

Discussion Properties allow to extend PDL specifications beyond the defined tags and attributes given by the fixed PDL language syntax, in an ad-hoc way. This is highly flexible but can also lead to inconsistencies and confusion due to lack of standardization of naming conventions for properties.

Properties in PDL can be mandatory or optional. We think that mandatory properties should better be modeled as predefined XML tags or attributes, to allow for static checking. For instance, a property within a CPU descriptor such as `x86_MAX_CLOCK_FREQUENCY` [1] should better be specified as a predefined attribute.

Using Platform Descriptions for Conditional Composition In our recent work on *conditional composition* [3] we used PDL specifications to guide the dynamic selection of implementation variants of PEPPHER components. In the *PEPPHER composition tool* [2], which builds an adaptive executable program for a PEPPHER application from its annotated multi-variant components, the target system's PDL specification is parsed and stored in an internal representation, which can be inspected at composition time to check for selectability constraints that depend on static property values. In order to also enable constraints that involve dynamic properties or property values, the composition tool also writes the PDL representation to a file that is loaded by the PEPPHER application's executable code on startup into a run-time data structure. This data structure thus holds the platform metadata which the composition code can introspect via a C++ API, i.e. read access property values in the evaluation of constraints that guide selectability of implementation variants. In a case study with a single sparse matrix vector multiply component in [3], we used this feature to let each CPU and GPU implementation variant specify its specific constraints on availability of specific libraries (such as sparse BLAS libraries) in the target system, and to add selection constraints based on the density of nonzero elements, leading to an overall performance improvement.

D. Modularity Issues

Another limitation of PDL is its semi-modular structure of system specifications. While it is generally possible to decompose any XML specification into multiple submodules (files), PDL does not specifically encourage such modularity and tends to produce monolithic system descriptions, which limits the reuse of specifications of platform subcomponents.

III. XPDL Design Principles

In contrast to the PDL default scenario of a single monolithic platform descriptor file, XPDL provides a modular design by default that promotes reuse of submodels and thus avoids replication of contents with the resulting risk for inconsistencies. XPDL defines a hierarchy of submodels, organized in model libraries containing separate models for entities such as

- CPUs, cores, caches,
- Memory,
- GPUs and other accelerators with own device memory,
- Interconnects, such as inter-node networks (e.g. Infiniband), intra-node networks (e.g. PCIe) and on-chip networks, and
- System software installed.

In principle, a XPDL descriptor is a machine-readable data sheet of the corresponding hardware or system software entity, containing information relevant for performance and energy optimization. The descriptors for the various types of CPUs, memory etc. are XPDL descriptor modules (`.xpdl` files) placed in a distributed model repository: XPDL models can be stored locally (retrieved via the model search path), but may, ideally, even be provided for download e.g. at hardware manufacturer web sites. Additionally, the model repository contains (source) code of microbenchmarks referenced from the descriptors.

In addition to the models describing hardware and system software components, there are (top-level) models for complete systems, which are (like their physical counterparts) composed from third-party provided hardware and software components, and reference the corresponding XPDL model descriptors by name to include these submodels.

Alternative Views of XPDL Models Generally, XPDL offers multiple views: XML (used in the examples in this paper), UML (see [4]), and C++ (as used for the internal and runtime representation of models). These views only differ in syntax but are semantically equivalent, and are (basically) convertible to each other.

A. Basic Features of XPDL

We distinguish between *meta-models* (classes describing model element *types*) and *concrete models* of the concrete hardware components (i.e., *instances*) in the modeled target system, as several concrete models can share the same meta-model. For example, several concrete CPUs in computer systems can share the same CPU type definition.

[1] Both keys and values are strings in PDL.

In principle, each XPDL model of a (reusable) hardware component shall be specified separately in its own descriptor file, with the extension `.xpdl`. For pragmatic reasons, it is sometimes more convenient to specify sub-component models in-line, which is still possible in XPDL. Reusing and referencing submodels is done by referencing them by a unique name, so they can easily be retrieved in the same model or in the XPDL model repository. Depending on whether one includes a submodel at meta-level or base-level, the naming and referencing use different attribute names. To specify an identifier of a model element, the attribute `name` is used for a meta-model, and the attribute `id` is for a model. The strings used as `name` and `id` should be unique across the XPDL repository for reference non-ambiguity, and naming is only necessary if there is a need to be referenced. The attribute `type` is used in both base-level and meta models for referencing to a specific meta-model.

Using the `extends` attribute, XPDL also supports (multiple) *inheritance* to encourage more structuring and increased reuse by defining attributes and subcomponents common to a family of types (e.g., GPU types) in a supertype. The inheriting type may overscribe attribute values.

The XML element `group` can be used to group any elements. If the attribute `quantity` is used in a `group` element (as in Listing 1), then the group is implicitly homogeneous. In such cases, to facilitate identifier specification, attributes `prefix` and `quantity` can be used together to assign an `id` to the group member elements automatically. For example, if we specify a group with prefix `core` and quantity `4`, then the identifiers of the group members are assigned as `core0`, `core1`, `core2` and `core3`.

For a metric such as static_power, if specified as an attribute, its unit should also be specified, in *metric*`_unit` form such as `static_power_unit` for `static_power`. As an exception, the unit for the metric `size` is implicitly specified as `unit`.

Installed software is given by `installed` tags; also these refer to separate descriptors, located in the software directory.

The `<properties>` tag refers to other possibly relevant properties that are not modeled separately by own descriptors but specified flexibly by key-value pairs. This escape mechanism (which avoids having to create new tags for such properties and allows ad-hoc extensions of a specification) was also suggested in PDL [1].

Not all entities are modeled explicitly. For instance, the motherboard (which also contributes some base energy cost) may not be modeled explicitly; its static energy share will be derived and associated with the node when calculating static and dynamic energy costs for each (hardware) component in the system.

B. Hardware Component Modeling

In the following we show examples of XPDL descriptions of hardware components. Note that these examples are simplified for brevity. More complete specifications for some of the EXCESS systems are given in [4].

Processor Core Modeling Consider a typical example of a quad-core CPU architecture where L1 cache is private, L3 is shared and L2 is shared by 2 cores. Listing 1 shows the corresponding XPDL model. The use of attribute `name` in the `cpu` element indicates that this specifies a name of this meta-model in XPDL, and the attribute `quantity` in the `group` element shows that the group is homogeneous, consisting of identical elements whose number is defined with the attribute `quantity`. Furthermore, the `prefix` can define a name prefix along with `quantity`, which automatically assigns identifiers to all group members that are built by concatenating the prefix string with each group member's unique rank ranging from 0 to the group size minus one.

The sharing of memory is given implicitly by the hierarchical scoping in XPDL. In the example, the L2 cache is in the same scope as a group of two cores, thus it is shared by those two cores.

```
<cpu name="Intel_Xeon_E5_2630L">
  <group prefix="core_group" quantity="2">
    <group prefix="core" quantity=2>
      <!--  Embedded definition -->
      <core frequency="2" frequency_unit="GHz" />
      <cache name="L1" size="32" unit="KiB" />
    </group>
    <cache name="L2" size="256" unit="KiB" />
  </group>
  <cache name="L3" size="15" unit="MiB" />
  <power_model type="power_model_E5_2630L" />
</cpu>
```

Listing 1. An example meta-model for a specific Xeon processor

Although the element `core` should rather be defined in a separate sub-metamodel (file) referenced from element `cpu` for better reuse, we choose here for illustration purposes to embed its definition into the CPU meta-model. Embedding subcomponent definitions gives us the flexibility to control the granularity of modeling, either coarse-grained or fine-grained. The same situation also applies to the cache hierarchy.

```
<!-- Descriptor file ShaveL2.xpdl -->
<cache name="ShaveL2" size="128" unit="KiB" sets="2"
       replacement="LRU" write_policy="copyback" />

<!-- Descriptor file DDR3_16G.xpdl -->
<memory name="DDR3_16G" type="DDR3"
  size="16" unit="GB"
  static_power="4" static_power_unit="W" />
```

Listing 2. Two example meta-models for memory modules

Memory Module Modeling Listing 2 shows example models for different memory components, in different files.

Interconnect Modeling The tag `interconnect` is used to denote different kinds of interconnect technologies, e.g. PCIe, QPI, Infiniband etc. Specifically, in the PCIe example of Listing 3, the channels for upload and download are modeled separately, since the energy and time cost for the two channels might be different [7].

```
<!-- Descriptor file pcie3.xpdl: -->
<interconnect name="pcie3">
 <channel name="up_link"
   max_bandwidth="6" max_bandwidth_unit="GiB/s"
   time_offset_per_message="?"
   time_offset_per_message_unit="ns"
   energy_per_byte="8" energy_per_byte_unit="pJ"
   energy_offset_per_message="?"
   energy_offset_per_message_unit="pJ" />
 <channel name="down_link" ... />
</interconnect>

<!-- Descriptor file spi1.xpdl: -->
<interconnect name="spi...">
  ...
</interconnect>
```

Listing 3. Example meta-models for some interconnection networks

Device Modeling Listing 4 shows a concrete model for a specific Myriad-equipped server (host PC with a Movidius MV153 development board containing a Myriad1 DSP processor), thus its name is specified with `id` instead of `name`. This Myriad server uses several interconnections to connect the host server to Myriad1, e.g. SPI and USB. For an instantiation of any kind of interconnect, the connection information must also be specified, e.g. using the `head` and `tail` attributes for directed communication links.

```
<system id="myriad_server">
  ...
  <socket>
   <cpu id="myriad_host" type="Xeon1"
        role="master"/>
  </socket>
  <device id="mv153board" type="Movidius_MV153" />
  <interconnects>
   <interconnect id="connect1" type="SPI"
      head="myriad_host" tail="mv153board" />
   <interconnect id="connect2" type="usb_2.0"
      head="myriad_host" tail="mv153board" />
   <interconnect id="connect3" type="hdmi"
      head="myriad_host" tail="mv153board" />
   <interconnect id="connect4" type="JTAG"
      head="myriad_host" tail="mv153board" />
  </interconnects>
  ...
</system>
```

Listing 4. A concrete model for a Myriad-equipped server

The device with the Myriad1 processor on it is a Movidius MV153 card, whose meta-model named `Movidius_MV153` is specified in Listing 5 and which is `type`-referenced from the Myriad server model.

```
<device name="Movidius_MV153">
  <socket>
    <cpu type="Movidius_Myriad1"
         frequency="180" frequency_unit="MHz" />
  </socket>
</device>
```

Listing 5. Example meta-model for Movidius MV153 board

The MV153 model in Listing 5 in turn refers to another meta-model named `Myriad1` which models the Myriad1 processor, see Listing 6.

```
<cpu name="Movidius_Myriad1">
 <core id="Leon" type="Sparc_V8" endian="BE" >
  <cache name="Leon_IC" size="4" unit="kB"
         sets="1" replacement="LRU" />
  <cache name="Leon_DC" size="4" unit="kB"
         sets="1" replacement="LRU"
         write_policy="writethrough" />
 </core>
 <group prefix="shave" quantity="8">
   <core type="Myriad1_Shave" endian="LE" />
     <cache name="Shave_DC" size="1" unit="kB"
            sets="1" replacement="LRU"
            write_policy="copyback" />
   </core>
 </group>
 <cache name="ShaveL2" size="128" unit="kB" sets="2"
        replacement="LRU" write_policy="copyback" />
 <memory name="Movidius_CMX" type="CMX"
         size="1" unit="MB" slices="8" endian="LE"/>
 <memory name="LRAM" type="SRAM"
         size="32" unit="kB" endian="BE" />
 <memory name="DDR" type="LPDDR"
         size="64" unit="MB" endian="LE" />
</cpu>
```

Listing 6. Example meta-model for Movidius Myriad1 CPU

```
<system id="liu_gpu_server">
  <socket>
    <cpu id="gpu_host" type="Intel_Xeon_E5_2630L"/>
  </socket>
  <device id="gpu1" type="Nvidia_K20c" />
  <interconnects>
    <interconnect id="connection1" type="pcie3"
                  head="gpu_host" tail="gpu1" />
  </interconnects>
</system>
```

Listing 7. A concrete model for a GPU server

GPU modeling is shown in Listings 7–10. The K20c GPU (Listing 9) inherits most of the descriptor contents from its supertype `Nvidia_Kepler` (Listing 8) representing a family of similar GPU types. The 64KB shared memory space in each shared-memory multiprocessor (SM) on Kepler GPUs can be partitioned among L1 cache and shared memory in three different configurations (16+48, 32+32, 48+16 KB). This configurability is modeled by defining constants (`const`), formal parameters (`param`) and constraints, see Listing 8. In contrast, a concrete K20c GPU instance as in Listing 10 uses one fixed configuration that overrides the generic scenario inherited from the metamodel. Note that some attributes of `K20c` are inherited from the Nvidia_Kepler supertype, while the K20c model sets some uninitialized parameters like global memory size (`gmsz`), and overwrites e.g. the inherited default value of the attribute `compute_capability`.

```
<device name="Nvidia_Kepler" extends="Nvidia_GPU"
        role="worker">
 <compute_capability="3.0" />
 <const name="shmtotalsize"... size="64" unit="KB"/>
 <param name="L1size" configurable="true"
        type="msize" range="16, 32, 64" unit="KB"/>
 <param name="shmsize" configurable="true"
        type="msize" range="16, 32, 64" unit="KB"/>
 <param name="num_SM" type="integer"/>
 <param name="coresperSM" type="integer"/>
 <param name="cfrq" type="frequency" />
 <param name="gmsz" type="msize" />
 <constraints>
   <constraint expr=
        "L1size + shmsize == shmtotalsize" />
 </constraints>
 <group name="SMs" quantity="num_SM">
   <group name="SM">
     <group quantity="coresperSM">
       <core type="..." frequency="cfrq" />
     </group>
     <cache name="L1" size="L1size" />
     <memory name="shm" size="shmsize" />
   </group>
 </group>
 <memory type="global" size="gmsz" />
 ...
 <programming_model type="cuda6.0,...,opencl"/>
</device>
```

Listing 8. Example meta-model for Nvidia Kepler GPUs

```
<device name="Nvidia_K20c" extends="Nvidia_Kepler">
 <compute_capability="3.5" />
 <param name="num_SM" value="13" />
 <param name="coresperSM" value="192" />
 <param name="cfrq" frequency="706" ...unit="MHz"/>
 <param name="gmsz" size="5" unit="GB" />
 ...
</device>
```

Listing 9. Example meta-model for Nvidia GPU K20c

```
<device id="gpu1" type="Nvidia_K20c">
 <!-- fixed configuration: -->
 <param name="L1size" size="32" unit="KB" />
 <param name="shmsize" size="32" unit="KB" />
 ...
</device>
```

Listing 10. Example model for a concrete Nvidia GPU K20c

Cluster Modeling Listing 11 shows a concrete cluster model in XPDL. The cluster has 4 nodes each equipped with 2 CPUs and 2 different GPUs. Within each node, the GPUs are attached with PCIe3 interconnect, while an Infiniband switch is used to connect different nodes to each other.

```
<system id="XScluster">
  <cluster>
    <group prefix="n" quantity="4">
      <node>
        <group id="cpu1">
          <socket>
            <cpu id="PE0" type="Intel_Xeon_..." />
          </socket>
          <socket>
            <cpu id="PE1" type="Intel_Xeon_..." />
          </socket>
        </group>
        <group prefix="main_mem" quantity="4">
          <memory type="DDR3_4G" />
        </group>
        <device id="gpu1" type="Nvidia_K20c" />
        <device id="gpu2" type="Nvidia_K40c" />
        <interconnects>
          <interconnect id="conn1" type="pcie3"
                        head="cpu1" tail="gpu1" />
          <interconnect id="conn2" type="pcie3"
                        head="cpu1" tail="gpu2" />
        </interconnects>
      </node>
    </group>
    <interconnects>
      <interconnect id="conn3" type="infiniband1"
                    head="n1" tail="n2" />
      <interconnect id="conn4" type="infiniband1"
                    head="n2" tail="n3" />
     ...
    </interconnects>
  </cluster>
  <software>
    <hostOS id="linux1" type="Linux_..." />
    <installed type="CUDA_6.0"
               path="/ext/local/cuda6.0/" />
    <installed type="CUBLAS_..." path="..." />
    <installed type="StarPU_1.0" path="..." />
  </software>
  <properties>
    <property name="ExternalPowerMeter" type="..."
              command="myscript.sh" />
  </properties>
</system>
```

Listing 11. Example of a concrete cluster machine

C. Power Modeling

Power modeling consists in modeling power domains, power states with transitions and referencing to microbenchmarks for power benchmarking. A power model (instance) is referred to from the concrete model of the processor, GPU etc.

Power domains or *power islands* are groups of cores etc. that need to be switched together in power state transitions. Every hardware subcomponent not explicitly declared as (part of) a (separate) power domain in a XPDL power domain specification is considered part of the default (main) power domain of the system. For the default power domain there is only one power state that cannot be switched off and on by software, i.e., there are no power state transitions.

For each explicitly defined power domain, XPDL allows to specify the possible *power states* which are the states of a finite state machine (the *power state machine*) that abstract the available discrete DVFS and shutdown levels, often referred to as P states and C states in the processor documentation, specified along with their static energy levels (derived by microbenchmarking or specified in-line as a constant value where known). A power state machine has power states and transitions, and must model all possible transitions (switchings) between states that the programmer can initiate, along with their concrete overhead costs in switching time and energy.

The dynamic power depends, among other factors, on the instruction type [7] and is thus specified for each instruction type or derived automatically by a specific microbenchmark that is referred to from the power model for each instruction.

With these specifications, the processor's energy model can be bootstrapped at system deployment time automatically by running the microbenchmarks to derive the unspecified entries in the power model where necessary.

```
<power_domains name="Myriad1_power_domains">
  <!-- this island is the main island -->
  <!-- and cannot be turned off        -->
  <power_domain name="main_pd"
                enableSwitchOff="false">
    <core type="Leon" />
  </power_domain>
  <group name="Shave_pds" quantity="8">
    <power_domain name="Shave_pd">
      <core type="Myriad1_Shave" />
    </power_domain>
  </group>
  <!-- this island can only be turned off      -->
  <!-- if all the Shave cores are switched off -->
  <power_domain name="CMX_pd"
                switchoffCondition="Shave_pds off">
    <memory type="CMX" />
  </power_domain>
</power_domains>
```

Listing 12. Example meta-model for power domains of Movidius Myriad1

```
<power_state_machine name="power_state_machine1"
                     power_domain="xyCPU_core_pd">
  <power_states>
    <power_state name="P1" frequency="1.2"
                 frequency_unit="GHz"
                 power="20" power_unit="W" />
    <power_state name="P2" frequency="1.6" ... />
    <power_state name="P3" frequency="2.0" ... />
  </power_states>
  <transitions>
   <transition head="P2" tail="P1"
               time="1" time_unit="us"
               energy="2" energy_unit="nJ"/>
   <transition head="P3" tail="P2" ... />
   <transition head="P1" tail="P3" ... />
  </transitions>
</power_state_machine>
```

Listing 13. Example meta-model for a power state machine of a pseudo-CPU

Listing 12 shows an example of a power domain meta-model. It consists of one power domain for the Leon core, eight power domains for each Shave core, etc. The setting for attribute `enableSwitchOff` specifies that the power domain for the Leon core can not be switched off. The attribute `switchoffCondition` specifies that power domain `CMX_pd` can only be switched off if the group of power domains `Shave_pds` (all Shave cores) is switched off.

Listing 13 shows an example of a power state machine for a power domain `xyCPU_core_pd` in some CPU, with the applicable power states and transitions by DVFS.

```
<instructions name="x86_base_isa"
              mb="mb_x86_base_1" >
 <inst name="fmul"
       energy="?" energy_unit="pJ" mb="fm1"/>
 <inst name="fadd"
       energy="?" energy_unit="pJ" mb="fa1"/>
 ...
 <inst name="divsd">
   ...
   <data frequency="2.8"
         energy="18.625" energy_unit="nJ"/>
   <data frequency="2.9"
         energy="19.573" energy_unit="nJ"/>
   ...
   <data frequency="3.4"
         energy="21.023" energy_unit="nJ"/>
 </inst>
</instructions>
```

Listing 14. Example meta-model for instruction energy cost

Instruction Energy The instruction set of a processor is modeled including references to corresponding microbenchmarks that allow to derive the dynamic energy cost for each instruction automatically at deployment time. See Listing 14 for an example. For some instructions, concrete values may be given, here as a function / value table depending on frequency, which was experimentally confirmed. Otherwise, the energy entry is set to a placeholder (`?`) stating that the concrete energy cost is not directly available and needs to be derived by microbenchmarking. On request, microbenchmarking can also be applied to instructions with given energy cost and will then override the specified values.

Microbenchmarking An example specification of a microbenchmark suite is shown in Listing 15. It refers to a directory containing a microbenchmark for every instruction of interest and a script that builds and runs the microbenchmark to populate the energy cost entries.

```
<microbenchmarks id="mb_x86_base_1"
                 instruction_set="x86_base_isa"
                 path="/usr/local/micr/src"
                 command="mbscript.sh">
 <microbenchmark id="fa1" type="fadd" file="fadd.c"
                 cflags="-O0" lflags="..." />
 <microbenchmark id="mo1" type="mov" file="mov.c"
                 cflags="-O0" lflags="..." />
 ...
</microbenchmarks>
```

Listing 15. An example model for instruction energy cost

A *power model* thus consists of a description of its power domains, their power state machines, and of the microbenchmarks with deployment information.

D. Hierarchical Energy Modeling

A concrete system model forms a tree hierarchy, where model elements such as `socket`, `node` and `cluster` form inner nodes and others like `gpu`, `cache` etc. may form leaves that contain no further modeling elements as explicitly described hardware sub-components.

Every node in such a system model tree (e.g., of type cpu, socket, device, gpu, memory, node, interconnect, cluster, or system) has explicitly or implicitly defined attributes such as `static_power`. These attributes are either directly given for a node or derived (synthesized). Directly given attribute values are either specified in-line (e.g., if it is a known constant) or derived automatically at system deployment time by specifying a reference to a microbenchmark. Synthesized attributes[2] can be calculated by applying a rule combining attribute values of the node's children in the model tree, such as adding up static power values over the direct hardware subcomponents of the node.

IV. XPDL Toolchain and Runtime Query API

The *XPDL processing tool* to be developed in subsequent work runs statically to build a run-time data structure based on the XPDL descriptor files. It browses the XPDL model repository for all required XPDL files recursively referenced in a concrete model tree, parses them, generates an intermediate representation of the composed model in C++, generates microbenchmarking driver code, invokes runs of microbenchmarks where required to derive attributes with unspecified values, filters out uninteresting values, performs static analysis of the model (for instance, downgrading bandwidth of interconnections where applicable as the effective bandwidth should be determined by the slowest hardware components involved in a communication link), and builds a light-weight run-time data structure for the composed model that is finally written into a file. The XPDL processing tool should be configurable, thus the filtering rules for uninteresting values and static analysis / model elicitation rules can be tailored. Then the EXCESS run-time system or the generated composition code can use the XPDL Runtime Query API described further below to load the run-time data structure file at program startup time, in order to query platform information dynamically.

One important requirement of the XPDL query API is to provide platform information to the upper optimization layers in the EXCESS framework. Such relevant hardware properties include whether a specific type of processor is available as required for a specific software component variant (conditional composition [3]), or what the expected communication time or the energy cost to use an accelerator is, etc.

At top level, we distinguish between two types of computer systems: single-node and multiple-node computers. A single-node computer has a basic master-slave structure and a single operating system; a multiple-node computer has several nodes.

The *XPDL run-time query API* provides run-time introspection of the model structure and parameters, which can be used for platform-aware dynamic optimizations [3]. It consists of the following four categories of functions:

1) *Initialization of the XPDL run-time query library.* For example, function `int xpdl_init(char *filename)` initializes the XPDL run-time query environment and loads the file with the run-time model data structure as generated by the XPDL toolchain.
2) *Functionality for browsing the model tree.* These functions look up inner elements of a model element and return a pointer to an inner model tree object or array of model tree objects respectively if existing, and NULL otherwise.
3) *Functions for looking up attributes of model elements*, which are likewise generated getter functions.
4) *Model analysis functions for derived attributes*, such as counting the number of cores or of CUDA devices in a model subtree or adding up static power contributions in a model subtree. These getters can be generated automatically from the rules that have been specified to calculate these derived attributes from existing attributes.

We map basically every element in XPDL to a run-time model class in C++. Then the XPDL Query API consists of the (generated) set of all getter functions for the member variables of each C++ class. As also object references can be queried, these allow to browse the whole model object tree at runtime.

C++ classes (as opposed to structs in C) are a natural choice because they allow for inheritance (as in UML and XML-XPDL) which facilitates the modeling of families of architectural blocks, and because we do not foresee constraints in the query language choice limiting it to C, because the XPDL query API is intended to be used on CPU (i.e., general purpose cores linked to main memory) only.

The XPDL model processing tool generates for all attributes Query API functions as getter functions for the corresponding class member variables, such as m.`get_id()` to retrieve the value of the `id` attribute for model object m.

The major part of the XPDL (run-time) query API (namely the C++ classes corresponding to model element types, with getters and setters for attribute values and model navigation support) is generated automatically from the central `xpdl.xsd` schema specification, which contains the core metamodel of XPDL. C++ class names are derived from

[2] Note the analogy to attribute grammars in compiler construction.

name attributes, getter and setter names are based on the declared attribute names etc. As the core XPDL schema definition is shared (to be made available for download on our web server), it will be easy to consistently update the Query API to future versions of XPDL. In contrast, the model analysis functions for derived attributes (such as aggregating the overall number of cores in a system etc.) will not be generated from the schema but need to be implemented manually and can be included by inheritance.

Implementation Status Currently the XML schema, parser (based on Xerces) and intermediate representation for XPDL are completed, and the benchmarking driver generator and runtime model generator are under development. When finished, the XPDL toolchain will be made publically available as open-source software at www.excess-project.eu[3].

V. Related Work

Architecture description languages (ADL) for describing hardware architecture[4] have been developed and used mainly in the embedded systems domain since more than two decades, mostly for modeling single-processor designs. Hardware ADLs allow to model, to some degree of detail, either or both the *structure* (i.e., the hardware subcomponents of an architecture and how they are connected) and the *behavior* (i.e., the set of instructions and their effect on system state and resources). Depending on their design and level of detail, such languages can serve as input for different purposes: for hardware synthesis and production e.g. as ASIC or FPGA, for generating (parts of) a simulator for a processor design, or for generating (parts of) the program development toolchain (compiler, assembler, linker). Examples of such ADLs include Mimola, nML, LISA, EXPRESSION, HMDES, ISDL, see e.g. [8] for a survey and classification. In particular, languages used by retargetable code generators (such as HMDES, LISA, ISDL and xADML [6]) need to model not only the main architectural blocks such as functional units and register sets, but also the complete instruction set including resource reservation table and pipeline structure with operand read and write timing information for each instruction, such that a optimizing generic or retargetable code generator (performing instruction selection, resource allocation, scheduling, register allocation etc.) can be parameterized or generated, respectively; see also Kessler [9] for a survey of issues in (retargetable) code generation. Note that such ADLs differ from the traditional *hardware description languages* (HDLs) such as VHDL and Verilog which are mainly used for low-level hardware synthesis, not for toolchain generation because they lack high-level structuring and abstraction required from a toolchain's point of view. However, a HDL model could be generated from a sufficiently detailed ADL model.

Architecture description languages for multiprocessors have become more popular in the last decade, partly due to the proliferation of complex multicore designs even in the embedded domain, but also because portable programming frameworks for heterogeneous multicore systems such as Sequoia [10] for Cell/B.E. and PEPPHER [2], [11] for GPU-based systems respectively, need to be parameterized in the relevant properties of the target architecture in order to generate correct and optimized code. Sequoia [10] includes a simple language for specifying the memory hierarchy (memory modules and their connections to each other and to processing units) in heterogeneous multicore systems with explicitly managed memory modules, such as Cell/B.E.

MAML [12] is a structural, XML-based ADL for modeling, simulating and evaluating multidimensional, massively parallel processor arrays.

Hwloc (Hardware Locality) [13] is a software package that detects and represents the hardware resources visible to the machine's operating system in a tree-like hierarchy modeling processing components (cluster nodes, sockets, cores, hardware threads, accelerators), memory units (DRAM, shared caches) and I/O devices (network interfaces). Like XPDL, its main purpose is to provide structured information about available hardware components, their locality to each other and further properties, to the upper layers of system and application software in a portable way.

ALMA-ADL [14] developed in the EU FP7 ALMA project is an architecture description language to generate hierarchically structured hardware descriptions for MPSoC platforms to support parallelization, mapping and code generation from high-level application languages such as MATLAB or Scilab. For the syntax, it uses its own markup language, which is extended with variables, conditions and loop constructs for compact specifications, where the loop construct is similar to the group construct in XPDL.

HPP-DL [15] is a platform description language developed in the EU FP7 REPARA project to support static and dynamic scheduling of software kernels to heterogeneous platforms for optimization of performance and energy efficiency. Its syntax is based on JSON rather than XML. HPP-DL provides predefined, typed main architectural blocks such as CPUs, GPUs, memory units, DSP boards and buses with their attributes, similarly to the corresponding XPDL classes. In comparison to XPDL, the current specification of HPP-DL [15] does not include support for modeling of power states, dynamic energy costs, system software, distributed specifications, runtime model access or automatic microbenchmarking.

[3] For further information on XPDL and download, see also http://www.ida.liu.se/labs/pelab/xpdl

[4] Note that the same term and acronym is also used for software architecture description languages, which allow to formally express a software system's high-level design in terms of coarse-grained software components and their interconnections, and from which tools can generate platform-specific glue code automatically to deploy the system on a given system software platform. In this section we only refer to Hardware ADLs.

What ADLs generally omit is information about system software such as operating system, runtime system and libraries, because such information is of interest rather for higher-level tools and software layers, as for composing annotated multi-variant components in PEPPHER and EXCESS. As discussed earlier, PDL [1] models software entities only implicitly and ad-hoc via free-form key-value properties. None of the ADLs considered puts major emphasis on modeling of energy and energy-affecting properties. To the best of our knowledge, XPDL is the first ADL for heterogeneous multicore systems that provides high-level modeling support also for system software entities, and that has extensive support for modeling energy and energy-affecting factors such as power domains and power state machines.

VI. Conclusion

We proposed XPDL, a portable and extensible platform description language for modeling performance and energy relevant parameters of computing systems in a structured but modular and distributed way. It supports retargetable toolchains for energy modeling and optimization, and allows application and system code to introspect its own execution environment with its properties and parameters, such as number and type of cores and memory units, cache sizes, available voltage and clock frequency levels, power domains, power states and transitions, etc., but also installed system software (programming models, runtime system, libraries, compilers, ...). We observe that, beyond our own work, such functionality is needed also in other projects in the computing systems community, and we thus hope that this work will contribute towards a standardized platform description language promoting retargetability and platform-aware execution.

Acknowledgment

This research was partially funded by EU FP7 project EXCESS (www.excess-project.eu) and SeRC (www.e-science.se) project OpCoReS.

We thank Dmitry Khabi from HLRS Stuttgart for suggestions on cluster modeling. We thank Jörg Keller from FernUniv. Hagen, Anders Gidenstam from Chalmers and all EXCESS project members for comments on this work. We also thank Ming-Jie Yang for improvements of the current XPDL toolchain prototype.

References

[1] M. Sandrieser, S. Benkner, and S. Pllana, "Using explicit platform descriptions to support programming of heterogeneous many-core systems," *Parallel Computing*, vol. 38, no. 1-2, pp. 52–65, 2012.

[2] U. Dastgeer, L. Li, and C. Kessler, "The PEPPHER composition tool: Performance-aware composition for GPU-based systems," *Computing*, vol. 96, no. 12, pp. 1195–1211, Dec. 2014.

[3] U. Dastgeer and C. W. Kessler, "Conditional component composition for GPU-based systems," in *Proc. MULTIPROG-2014 wksh. at HiPEAC'14 conf., Vienna, Austria*, Jan. 2014.

[4] C. Kessler, L. Li, U. Dastgeer, A. Gidenstam, and A. Atalar, "D1.2 initial specification of energy, platform and component modelling framework," EU FP7 project EXCESS (611183), Tech. Rep., Aug. 2014.

[5] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures," *Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009*, vol. 23, pp. 187–198, Feb. 2011.

[6] A. Bednarski, "Integrated optimal code generation for Digital Signal Processors," Ph.D. dissertation, Linköping Univ., 2006.

[7] C. Kessler, L. Li, U. Dastgeer, P. Tsigas, A. Gidenstam, P. Renaud-Goud, I. Walulya, A. Atalar, D. Moloney, P. H. Hoai, and V. Tran, "D1.1 Early validation of system-wide energy compositionality and affecting factors on the EXCESS platforms," Project Deliverable, EU FP7 project EXCESS, www.excess-project.eu, Apr. 2014.

[8] P. Mishra and N. Dutt, "Architecture description languages for programmable embedded systems," *IEE Proc.-Comput. Digit. Tech.*, vol. 152, no. 3, pp. 285–297, May 2005.

[9] C. Kessler, "Compiling for VLIW DSPs," in *Handbook of Signal Processing Systems, 2nd ed.*, S. Bhattacharyya, E. Deprettere, R. Leupers, and J. Takala, Eds. Springer, Sep. 2013.

[10] K. Fatahalian, D. R. Horn, T. J. Knight, L. Leem, M. Houston, J. Y. Park, M. Erez, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan, "Sequoia: Programming the memory hierarchy," in *ACM/IEEE Supercomputing*, 2006, p. 83.

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

[12] A. Kupriyanov, F. Hannig, D. Kissler, J. Teich, R. Schaffer, and R. Merker, "An architecture description language for massively parallel processor architectures," in *GI/ITG/GMM-Workshop - Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Sytemen*, Feb. 2006, pp. 11–20.

[13] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst, "hwloc: A generic framework for managing hardware affinities in HPC applications," in *Proc. 18th Euromicro Conf. on Par., Distr. and Netw.-based Proc. (PDP)*. IEEE CS, 2010, pp. 180–186.

[14] T. Stripf, O. Oey, T. Bruckschloegl, R. Koenig, J. Becker, G. Goulas, P. Alefragis, N. Voros, J. Potman, K. Sunesen, S. Derrien, and O. Sentieys, "A Compilation- and Simulation-Oriented Architecture Description Language for Multicore Systems," *Proc. 2012 IEEE 15th Int. Conf. on Computational Science and Engineering*, Nicosia, Dec. 2012, pp. 383–390.

[15] L. Sanchez (ed.), "D3.1: Target Platform Description Specification," EU FP7 REPARA project deliverable report, http://repara-project.eu, Apr. 2014.