Chapter 2
Resource-Driven Modelling for Managing Model Fidelity

Ashur Rafiev, Andrey Mokhov, Fei Xia, Alexei Iliasov, Rem Gensh, Ali Aalsaud, Alexander Romanovsky, and Alex Yakovlev

2.1 Introduction

Systems with large scale concurrency and complexity, e.g. computation systems built upon architectures with multiple and increasingly many processing cores with heterogeneity among the components, are becoming more popular and commonplace [6]. The hardware motivations are clear, as concurrency scaling can help delay the potential saturation of Moore's Law with current and future CMOS technology and better use the opportunities provided by the technology scaling. In this environment, software designs are increasingly focused towards greater concurrency and mapping to such many-core hardware [16].

If we regard elements of computation, such as software, hardware, energy, time, etc. as resources, then computation itself can be regarded as behaviours of the entire resource space. Existing modelling approaches usually include some degree of representation of resources [5]. Functional units such as transistors, gates, CPUs, memory, software threads, etc. need to be represented in functional models, and non-functional parameters including power, time, temperature, etc. also need to be represented in models that are used to study non-functional behaviours of systems. Existing modelling methods in the literature therefore embed certain degrees of resource representation. However, a modelling method that is entirely based on representing resources and their dependencies has, to our knowledge, not yet been investigated. Being able to reason about resources and their interdependency during computation directly, on the other hand, should be advantageous for studying the

A. Rafiev (✉) • A. Mokhov • F. Xia • A. Iliasov • R. Gensh • A. Aalsaud • A. Romanovsky • A. Yakovlev
Newcastle University, Newcastle upon Tyne, UK
e-mail: ashur.rafiev@ncl.ac.uk; andrey.mokhov@ncl.ac.uk; fei.xia@ncl.ac.uk; alexei.iliasov@ncl.ac.uk; r.gensh@ncl.ac.uk; a.m.m.aalsaud@ncl.ac.uk; alexander.romanovsky@ncl.ac.uk; alex.yakovlev@ncl.ac.uk

A. Molnos, C. Fabre (eds.), *Model-Implementation Fidelity in Cyber Physical System Design*, DOI 10.1007/978-3-319-47307-9_2

more complex systems of today and the future. For instance, power consumption has become a crucial limiting factor for the continued expansion of the world's computing capabilities and power is one of the most straightforward resources [11].

Functional resources, such as hardware and software in large complex systems, tend to form hierarchical structures, for instance, the levels of detail in hardware include the entire spectrum from transistors to gates to function blocks to entire CPUs to multiple CPUs with supporting logic, memory, etc. For system designers, software (e.g. applications), operating systems and the platforms on which these are run also form natural design layers with clear boundaries between the layers. Such structures are usually conveniently modelled with traditional hierarchical modelling methods, with the modelling levels of abstraction corresponding to these system layers of concern [15].

This is, however, not always optimal for analysis, design and runtime management. In most cases these require the modelling of particular parameters and the "modelling fidelity" [21] should, ideally, be determined by the parameter(s) under study [28]. For instance, when a part of a system makes a crucial contribution to the power consumption of the entire system and small changes may have a significant effect, it pays to study it in detail, i.e. at some lower layer of abstraction. On the other hand, to moderate the modelling, analysis and design effort, and potentially runtime overhead for models that need to be used in runtime, other less significant parts of the system should be studied at higher levels of abstraction. When this "centre of gravity" of system operation concerning power can dynamically move around the system, traditional hierarchical modelling methods are ill positioned for efficient representation.

Hierarchical methods, because of their complexity, are usually less straightforward to use than flat representations. Petri nets [3], which exemplify flat modelling methods, have extremely simple semantics and offer conveniences in reasoning, proofing and other aspects of analysis, a quality shared by other flat modelling methods. But when the modelling needs span multiple layers in a hierarchy it becomes somewhat difficult to adopt flat methods as study tools.

In this chapter, we present a systematic resource-driven approach to modelling that emphasises resources and dependencies between resources. The distinction between static design-time modelling which focus on types or classes of resources and run-time dynamic modelling which focuses on actual instances of resources will be made, with the proposed approach covering both issues. The semantics and other theoretical details will be presented, and derived modelling techniques, including the simulation tool ArchOn, will be used to solve real-world problems. For instance, scalable simulations and finding the optimal scaling factor for homogeneous many-core systems considering performance, energy and reliability (PER) trade-offs will be shown to be potential application areas of the method. This resource-driven focus will further lead to the presentation of a method of achieving resource parameter-proportional fidelity in models through analysing abstraction hierarchies and using cross-layer cuts. And this will be demonstrated through solving real PER and system task scheduling trade-offs in heterogeneous multi-core systems.

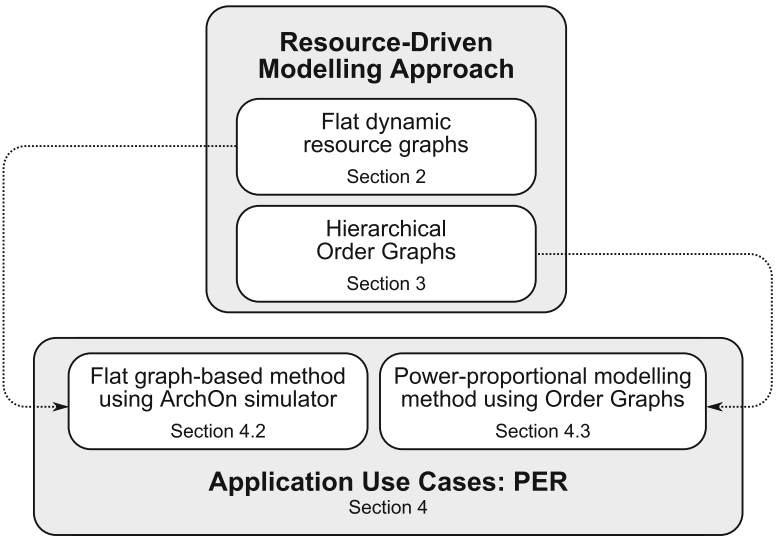

Fig. 2.1 The organisation of the chapter

The rest of this chapter is organised as shown in Fig. 2.1. Section 2.2 introduces the resource-driven modelling approach in a form of static and dynamic resource graphs, and also defines the notions of implementability and resource quantification. Section 2.3 describes Order Graphs (OG)—a hierarchical modelling formalism, and the concept of cross-layer cuts. In Sect. 2.4, methods derived from these theories are used solve the real-life problem of performance, energy and reliability interplay in systems. Section 2.5 concludes the chapter.

2.2 Resource-Driven Modelling

The central subject of our method is the study of a computational platform comprising a number of diverse resources and the way resources may be handled in order to realise a computation [24]. A resource is in this case an indivisible element required by the system in order to change its state, and it is defined by its function and availability in relation to this transition. With the word "resources" we make the point that we do not exclude computation, communication, or other facilities, e.g. energy and time.

We propose to represent a system with a relation graph (R, D), consisting of a set of vertices R and a set of edges $D \subseteq R \times R$. Each vertex $r \in R$ represents a single resource and each edge $d = \langle r_1, r_2 \rangle \in D$ represents a dependency between two resources $r_1, r_2 \in R$. Modelling different types of resources may be achieved by labelling the graph, as illustrated in Fig. 2.2a.

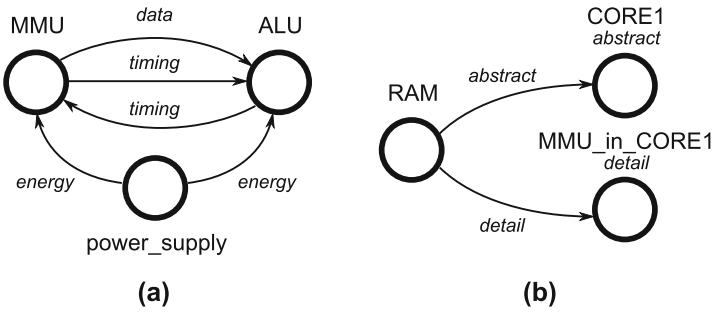

Fig. 2.2 Examples of using flat labelled graphs to reason about diverse resource types and dependencies (**a**), and different levels of abstraction (**b**)

Organising systems, both practically and conceptually, as hierarchies is a popular way of thinking and engineering. The practical motivation for this is manageability. This is the "natural" way for humans to reason about, design and organise most of our systems. In Sect. 2.3 we emphasise the hierarchy-based cross-layer aspects of our work, while this section focuses on the flat graph models.

Figure 2.2b demonstrates that the flat labelled graph approach can, in fact, facilitate the cross-layer way of thinking as long as the behaviour of different layers of abstraction is coherent. A label can be viewed as a condition that includes or excludes an edge or a vertex, giving a graph *projection* onto that label. The complexity of the system can be dealt with using projections of the resource graphs. With resources as diverse as a software instruction or a single hardware gate, within the same single graph executed in a transition, we could reason about different parts of the system at different abstraction layers. This helps a designer focus their attention on any particular details of a system they want, and build a system either top down or bottom up or with mixed-level components at different stages of development.

At the same time, this does not prevent designers to isolate concerns and concentrate on some layers only. For instance, all resources in one transition could be elements of the same layer, or a software engineer could arrange complex low-level software resources for detailed study with coarse-grain hardware resources provided by hardware colleagues (which are not the specific target of concern) in the same transition.

2.2.1 System Design and Implementation

This section introduces the basic concepts of the resource-driven modelling and draws the boundary between the static and dynamic resource representations of a system. It is essential to understand that the knowledge of a system (its definition or design) exists independently of its implementation. In order for the system to be

realised, we need to map this knowledge onto a set of resources—an architecture. Thence, if the architecture contains enough resources of the right type, the system implementation or implementations may emerge.

An unconstrained *architecture* $\mathcal{A}^+ = (R, T^r, \tau^r)$ is the set of discrete resources R and the set of resource types T^r, where $\tau^r : R \to T^r$ is the resource type assignment function. Such an architecture is called unconstrained because it does not put any restriction on how the resources may interact. Constrained architectures will be introduced later in Sect. 2.2.4. For now, we see an architecture as a "soup" of resources, where every resource is equally accessible.

The *system design* $\mathcal{S} = (V, E, T^v, \tau^v)$ is a graph, where V is the set of vertices, $E \subseteq V \times V$ is the set of edges (dependencies), T^v is the set of vertex types, and $\tau_v : V \to T^v$ is the type assignment for vertices. The type of some edge $\langle v_1, v_2 \rangle \in E$ is a tuple $\langle \tau^v(v_1), \tau^v(v_2) \rangle$.

In order for the system to be implemented, the vertices must be instantiated with resources and the edges must become active resource dependencies.

For some design $\mathcal{S}$ and an architecture $\mathcal{A}^+$, an *instantiation* of vertices V on a set of resources R is a partial function $q_i : V \nrightarrow R$, such that for any $v \in V, r \in R$ the statement $\langle v, r \rangle \in q_i \Rightarrow \tau^v(v) = \tau^r(r)$ holds true. In other words, resource r can instantiate vertex v if their types match. Relation q_i being a partial function means that for a single instantiation of V each vertex can be instantiated no more than once, however there can be multiple instantiations: $Q = \{q_0, q_1, \ldots\}$. Let $R_i = \mathrm{ran} q_i$ be the set of resources involved in the instantiation q_i. For any pair of instantiations $q_i \in Q, q_j \in Q, i \neq j$ and any resource $r \in R$, it is required that $r \in R_i \Rightarrow r \notin R_j$, i.e. any resource cannot belong to multiple instantiations at the same time.

Let $D \subseteq R_i \times R_i$ be the set of active resource dependencies, w.r.t. the instantiation q_i. D is constrained by E, i.e. for every $\langle r_1, r_2 \rangle \in D$ there must exist $\langle v_1, v_2 \rangle \in E$, such that $q_i(v_1) = r_1$ and $q_i(v_2) = r_2$. Also, following from the properties of vertex instantiation, any edge in E can have at most one related element in D.

Thus, an *implementation* of the design $\mathcal{S}$ on the architecture $\mathcal{A}^+$ is defined as $S_i = (\mathcal{S}, \mathcal{A}^+, q_i, D)$, where q_i is a vertex instantiation and D is a set of active resource dependencies. The design $\mathcal{S}$ can be mapped onto an architecture giving zero or more implementations (Fig. 2.3):

$$\mathcal{S} \mapsto \mathcal{A}^+ = \{S_0, S_1, \ldots\}. \tag{2.1}$$

Lastly, an implementation is *complete* if $\mathrm{dom} q_i = V$ and for any $\langle v_1, v_2 \rangle \in E$ there exist $\langle q_i(v_1), q_i(v_2) \rangle \in D$, i.e. every vertex and every edge of the system design graph are realised and active. The architecture may have more resources than it is required for the system implementation S_i, but it must have at least the required number of resources and dependencies in order to complete S_i. When it is not possible to form any complete implementations on the architecture $\mathcal{A}^+$ for the system design $\mathcal{S}$, the design is called *non-implementable* on the given architecture. For example, the design is surely non-implementable if it does not share common types with an architecture: $T^v \cap T^r = \emptyset$.

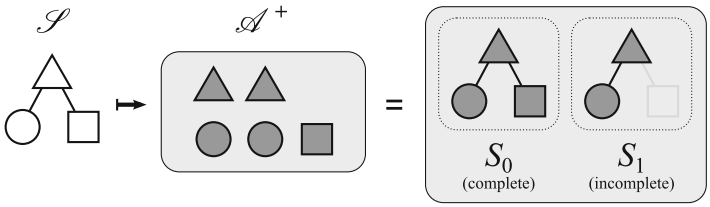

Fig. 2.3 System design $\mathcal{S}$ is mapped onto an architecture $\mathcal{A}^+$ giving system implementations $\{S_0, S_1, \ldots\}$, some or all of which may be incomplete

One can notice that vertex instantiations as well as active dependencies can have multiple solutions for each mapping $\mathcal{S} \mapsto \mathcal{A}^+$. In fact, the result depends on the order in which the elements of q_i and D have been added. That is why we consider this mapping as a non-deterministic discrete process.

2.2.2 Dynamic Systems and Architectures

In many real-life systems the dependencies between resources do not have to be maintained all the time in order for the system to function normally. In fact, for some systems the functionality requires switching dependencies on and off. For example, let's consider a FIFO connection consisting of reader, writer and a buffer. Reader and writer both depend on the buffer, so the system design contains edges $\langle \text{reader}, \text{buffer} \rangle$ and $\langle \text{writer}, \text{buffer} \rangle$. However, at runtime writer and reader accessing the buffer at the same time will cause deadlock, so these dependencies cannot be active at the same time.

In the previous section we mentioned that the mapping $\mathcal{S} \mapsto \mathcal{A}^+$ is a discrete process. Now we can go further by allowing elements of q_i and D to be not just added, but also removed from the implementation. Thus, the state of the dynamic set $q_i \times D$ encodes the state of the implementation S_i. Each state of a dynamic implementation S_i is called a *configuration*. We consider four possible transitions between configurations: instantiation of a vertex, releasing of a resource (as opposed to instantiation), activation of a dependency and deactivation of a dependency:

Instantiate vertex $v \in V$: if v is currently not instantiated in q_i and there is a resource $r \in R$ of the same type as the vertex v and this resource is not used in any other instantiations, add $\langle v, r \rangle$ to q_i.
Release vertex $v \in V$: if v is instantiated in q_i and resource $q_i(v)$ is not used in any active dependency, then remove v from q_i.
Activate dependency $\langle v_1, v_2 \rangle \in E$: if vertices v_1 and v_2 are instantiated in q_i but dependency $\langle q_i(v_1), q_i(v_2) \rangle$ is not active, activate it.
Deactivate dependency $\langle v_1, v_2 \rangle \in E$: if vertices v_1 and v_2 are instantiated in q_i and dependency $\langle q_i(v_1), q_i(v_2) \rangle$ is active, deactivate it.

On the same limited set of resources it is possible to have more working dynamic systems than static, because the dynamic systems can use resources "in turns" grabbing and then releasing them, e.g. as a CPU core activating computing resources according to the instruction being executed [20]. Sharing in the static systems is not possible.

It is important to note that, although an implementation S_i of the system is dynamic, the design of the system $\mathcal{S}$ remains static by definition and includes all vertices that can be instantiated and all dependencies that can be active.

Similarly to dynamic implementations, we can introduce dynamic architectures, where the set of resource R is dynamic, i.e. individual resources of an architecture may appear and disappear during runtime. The closest example is dark silicon [14], which exploits powering on and off different regions of the electronic system. Resources being excluded from R may also model malfunction of these resources. In this case, if some resource used in an active dependency in S_i leaves the architecture, then the system implementation S_i fails.

We illustrate the above definitions by a small example shown in Fig. 2.4. There are three resources A, B and C and two possible resource dependencies $ab = (A, B)$ and $ac = (A, C)$. When the system is fully shut down, no resources and no resource dependencies are required and can therefore be powered off to save energy, as illustrated by the empty box at the top of the diagram. The system can function normally in one of two modes as determined by a Boolean variable *mode*: when $mode = 0$ resources A and B as well as the dependency ab must be active, and

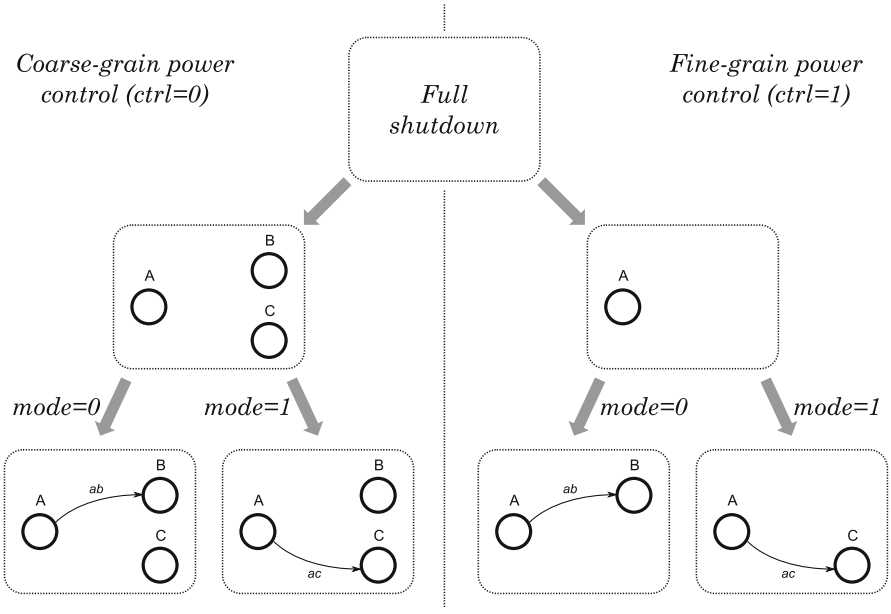

Fig. 2.4 An example of a dynamic resource graph with completeness condition (2.2)

when $mode = 1$ resources A and C as well as the dependency ac must be active. Let us now consider two possible power control strategies. The *coarse-grain control*, activated when $ctrl = 0$, powers on all resources regardless of the current mode, while the *fine-grain control*, activated when $ctrl = 1$, power on the resources on demand according to the current mode. Figure 2.4 covers all possible situations with the four boxes at the bottom of the diagram corresponding to the four possible complete implementations. One can derive the following *completeness condition* which captures all situations in a compact form:

$$A \wedge (\overline{mode} \wedge ab \wedge B \ \vee \ mode \wedge ac \wedge C). \tag{2.2}$$

Note that the condition does not depend on the chosen power control strategy *ctrl*, because it does not influence the completeness property. See [19] for a systematic approach to the derivation of such completeness conditions.

2.2.3 Resources Quantification and Reward Functions

In very large systems, representing each resource with a node leads to large increases in model sizes. An unconstrained architecture, defined in Sect. 2.2.1, differentiates the resources by their types, so it makes sense to group same-type resources under a single node with an added scalar value representing the quantity.

For some resource $r \in R$, its *quantity* $\omega(r) \in \mathbb{Z}^+$ is the number of the resource's instances. For some dependency $\langle r_1, r_2 \rangle \in D$, its *multiplicity* $\omega(\langle r_1, r_2 \rangle) \in \mathbb{Z}^+$ means that the number $\omega(\langle r_1, r_2 \rangle)$ of resource instances r_2 is dependent on the same number of resource instances r_1; $\omega(\langle r_1, r_2 \rangle) \leq \omega(r_1)$ and $\omega(\langle r_1, r_2 \rangle) \leq \omega(r_2)$.

Figure 2.5 gives an example of using resource quantification. The system design shown in Fig. 2.5a represents a task t running on a core c using some scheduler $s = \langle q_i(t), q_i(c) \rangle$. We map this design on the architecture shown in Fig. 2.5b, which consists of n cores $c_0, \ldots, c_{n-1}$ and a number of task instances. Task instances are

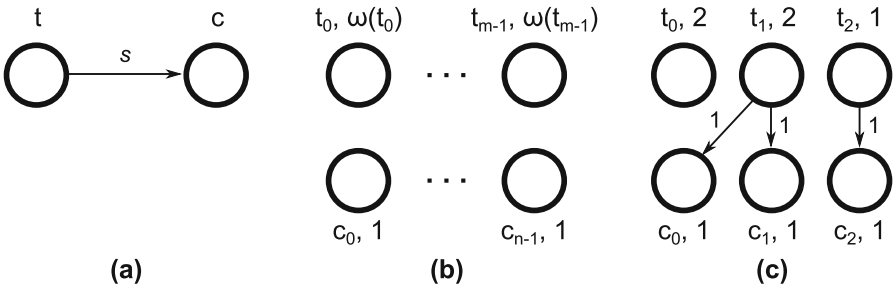

Fig. 2.5 Resource quantification example: (**a**) the system design representing a task t scheduled on a core c, (**b**) the system architecture consisting of n cores and m types of tasks, (**c**) a possible configuration for $n = 3, m = 3$

grouped by their type $t_0, \ldots, t_{m-1}$, m types in total; $\omega\left(t_j\right) \in \mathbb{Z}^+$ is the number of task instances of a particular type t_j, $0 \leq j < m$. Figure 2.5c shows a possible configuration for $n = 3, m = 3$. The system has two tasks of type 1 ($\omega\left(t_1\right) = 2$), thus the node t_1 can have two outgoing connections. Each core, in turn, has a quantifier of 1 and can connect to no more than one task, thus t_0 cannot be scheduled due to the lack of core resources.

In addition to quantifying discrete resources, we can model continuous resources by quantising them in a way that some amount of a continuous resource is represented with an integer ω. A dynamic architecture can also include an inflow and outflow of resource quanta.

An alternative approach, inspired by Markov analysis [17], is using reward functions. For a given configuration of a resource graph, the reward function computes some numeric value representing an instantaneous quantity of a continuous resource. For example, this could be an instantaneous power consumption in a system, and a time-integral of this reward would give the total amount of energy consumed. Notably, in many cases, reward functions can be computed locally on dependency arcs and adjacent resources.

2.2.4 Constrained Architectures

In general, an architecture is defined as $\mathcal{A} = (R, L, T^r, \tau^r)$, where R is the set of discrete resources, T^r is the set of resource types, $\tau^r : R \to T^r$ is the resource type assignment function. The *constraint* $L \subseteq R \times R$ is the set of allowed resource dependencies, so for any pair of resources $r_1 \in R, r_2 \in R$ the statement $\langle r_1, r_2 \rangle \in D \Rightarrow \langle r_1, r_2 \rangle \in L$ must be true, where D is a set of active dependencies of some system implementation $S_i = (\mathcal{S}, \mathcal{A}, q_i, D)$. Thus, an architecture is also a resource graph, and an unconstrained architecture $\mathcal{A}^+$ is a complete graph: $L = R \times R$. The process of mapping the design onto an architecture, $\mathcal{S} \mapsto \mathcal{A}$, is the process of mapping one graph onto another graph.

An illustrative example for constrained architectures is many-core processors. Every core in such a system contains a similar set of resources, like registers and ALUs, but the resources of one core cannot directly access the resources of another core. Instead, we need to connect them via the network component of the system. This can be modelled using architecture constraint.

To conclude Sect. 2.2, the presented resource-driven modelling approach allows the capturing of static and dynamic knowledge of a system being implemented on a given architecture. Both the system design and the architecture are represented with resource graphs, and the behaviour is realised in a transitional semantic. Quantitative modelling of the system resources is also possible using quantised resources or by defining reward functions.

By the definition of resource graphs, anything can be considered a resource. Can we say that the edges of a graph are also resources? It is actually true, especially for hierarchical systems and architectures. This contradiction is explained and solved by Order Graphs in the next section.

2.3 Hierarchical Modelling in Order Graphs

The following discussion revisits the definition of a hierarchy as a sequence of model transformations, which thereafter is applied to graph models leading to Order Graphs. The latter combines the notions of resource modelling with the hierarchical representation of system layers.

2.3.1 Introducing Hierarchies

An underlying approach for having adjustable fidelity in the models relies on different levels of abstraction. Naturally, these layers have to be consistent with each other, however the very definition of consistency may vary from model to model and depend on the system properties that need to be preserved.

A common way to define a model of a system is to represent it as a set tuple $M = (E_1, E_2, \ldots E_n)$, where each set E_k contains system elements of a particular type, e.g. vertices, edges, labels, etc. We can also generalise these to a single type—"system elements", $\mathcal{E}$—so $E_1 \subset \mathcal{E}, E_2 \subset \mathcal{E}, \ldots$. Thus, we can have a type-agnostic representation of a model: $\mathcal{M} = E_0 \cup E_1 \cup \cdots \cup E_n$.

Let M_a and M_b be some system models with corresponding sets of system elements $\mathcal{M}_a, \mathcal{M}_b$, and some relation between these elements $\gamma \subseteq \mathcal{M}_a \times \mathcal{M}_b$. Given a boolean predicate Φ, such that

$$\Phi : \mathbb{P}(M_a) \times \mathbb{P}(M_b) \times \mathbb{P}(\mathcal{M}_a \times \mathcal{M}_b) \to \{0, 1\}, \tag{2.3}$$

the relation γ is called a *consistency relation* between models M_a and M_b under the predicate Φ if $\Phi(M_a, M_b, \gamma) = 1$. Φ is called the *rule set*, and for convenience can be specified as a conjunction of smaller predicates of the same type (2.3).

The predicate Φ is called *strongly consistent* if it requires γ to be a total surjective relation, i.e. for every element in $\mathcal{M}_a$ there must be at least one related element in $\mathcal{M}_b$, and for every element in $\mathcal{M}_b$ there must be at least one related element in $\mathcal{M}_a$. In this case, γ is called a *transformation*; transformations are further denoted as $\gamma = \mathcal{M}_a \vdash \mathcal{M}_b$ (or $\gamma = M_a \vdash M_b$ since $\mathcal{M}_a, \mathcal{M}_b$ are derived from M_a and M_b).

Let $\{\ldots, M^{(k-1)}, M^{(k)}, M^{(k+1)}, \ldots\}$ be an infinite or finite set of models of the same system, where each $M^{(k)}$ models the system in a specific level of details. An *abstraction hierarchy* is a total order of models where any two adjacent models

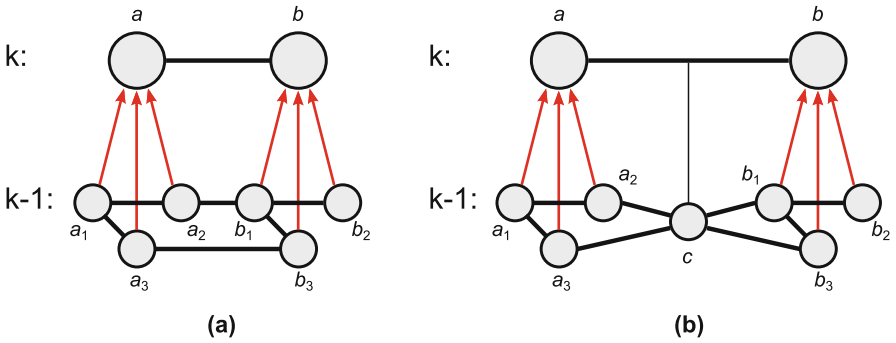

Fig. 2.6 Conventional hierarchy representation (**a**) compared to Order Graphs (**b**); k is the higher level of abstraction and $k-1$ is the lower level

form a transformation $\gamma_k = M^{(k)} \vdash M^{(k+1)}$ under a given strongly consistent predicate Φ_k, and the size of models monotonically decreases (or increases) with k:

$$\mathcal{H} = \cdots \vdash M^{(k-1)} \vdash M^{(k)} \vdash M^{(k+1)} \vdash \cdots \tag{2.4}$$

Each $M^{(k)}$ is kth level of abstraction, also called *order* k.

A hierarchy is called *homogeneous* if it uses the same rule set Φ for all its consistency relations; this implies that $\mathbb{P}\left(M^{(k)}\right) = \mathbb{P}\left(M^{(k+1)}\right)$ for all k.

Every hierarchy contains both horizontal and vertical knowledge: each abstraction layer $M^{(k)}$ is a horizontal view of the system, while the set of relations $\{\ldots, \gamma_k, \gamma_{k+1}, \ldots\}$ stores the information on how different layers interlink. Notions of horizontality and verticality can be found in [10].

Figure 2.6a shows the conventional approach to hierarchical graphs, which is based on clustering and uses tree structures [15]. Each node of a higher layer zooms into a subgraph in a lower layer. Consequently, an edge between two nodes becomes multiple edges between the corresponding subgraphs. The notation used in the diagram is based on Zoom Structures [10]. A convenient way to display graph hierarchies is zoom views, showing verticality and horizontality with vertical and horizontal arcs, respectively. The following is a redefinition of hierarchical graphs in the terms presented in Sect. 2.3.1.

A *Hierarchical graph* is a homogeneous hierarchy, such that, each kth order is a graph $G^{(k)} = (V, E)$, where V is the set of vertices and $E \subseteq V \times V$ is the set of edges; and all consistency relations in this hierarchy are defined as follows:

Inclusion function represents vertex clustering by relating multiple vertices in the lower order to a single vertex in a higher order.

Supplementary inclusion function ensures that all edges within a cluster are also included, i.e. if two vertices in the lower order relate to the same vertex in the higher order, any edge connecting them is automatically related to the same high-level vertex.

Edge grouping function groups edges connecting vertex clusters: an edge in the lower order connects vertices iff there is an edge in the higher order connecting related high-order vertices.

A more formal definition of these rules can be found in [25]. The inclusion function can be chosen arbitrarily, and from it, the other two uniquely describe the edges in the hierarchical graph.

The most important property of the rule set defined above is that it preserves all paths in the graph during the mapping. In other words, for any vertices $v_1, v_2 \in V$ and related vertices $v_1', v_2' \in V'$, if there exists a path between v_1 and v_2 in $G^{(k)}$, there will be a path between v_1' and v_2' in $G^{(k+1)}$, and vice versa:

$$\forall v_1, v_2 \in V, v_1', v_2' \in V' : \gamma_v(v_1) = v_1' \wedge \gamma_v(v_2) = v_2' \Rightarrow \left(P(v_1, v_2) \Leftrightarrow P\left(v_1', v_2'\right)\right), \tag{2.5}$$

where $P(x, y)$ is a function that is true iff there is a path between x and y. This property ensures that the dependencies between resources are consistent throughout the hierarchy.

2.3.2 Order Graphs

Section 2.2 suggests that an edge in a resource graph can be a resource (a node). As an example, let's imagine that Fig. 2.6a models a network interaction, where a is a server and b is a client. On the very abstract level we do not care about the structure of the network, we just need to know that the client and the server are connected somehow, thus we model this entire system as the client and the server connected directly with a single dependency. However, in a more detailed model we can no longer ignore the network protocols and have to introduce it at least as a single resource node as shown in Fig. 2.6b.

A distinct property of the proposed Order Graph (OG) modelling method is that a high-order edge relates to a node at the next lower order. In this case we say that the node *supports* an edge, while in fact this is the same entity viewed from the different abstraction levels. In real-life systems, any dependency is always supported by a resource of some kind, and this "fractal" structure goes down to the smallest details, including atoms and below. We may not want to include all these in the model, and this is pragmatically solved by saying that an edge is either supported by a resource at the lower layer or stays an edge like in conventional hierarchical graphs.

An *Order Graph* is a homogeneous hierarchy, such that, each kth order is a graph $G^{(k)} = (V, E)$, where V is the set of vertices and $E \subseteq V \times V$ is the set of edges; and all consistency relations in this hierarchy are defined as follows:

Inclusion, **supplementary inclusion**, and **edge grouping** are defined as in Sect. 2.3.1.

Support function is a one-to-one mapping of some vertices onto some of the edges of a higher order graph. The first rule on this function tells that we can map a vertex in the lower order to some edge $\langle v_1, v_2 \rangle$ in the higher order iff this vertex is connected to at least one vertex related to v_1 and at least one vertex related to v_2. In addition, all vertices adjacent to v must be related either to v_1 or v_2. Finally, the same vertex cannot be used in a vertex-to-vertex and a vertex-to-edge relation; and the same higher order edge cannot be used in an edge-to-edge and a vertex-to-edge relation.
Supplementary support function groups all edges adjacent to v into the same higher order edge.

These rules are formally defined in [25]. OGs preserve paths in the same way as (2.5) shows for hierarchical graphs.

2.3.3 *Cross-Layer Cuts*

In the approach presented in this chapter, the analysis of the system is performed on a flat model, not the entire hierarchy. The actual benefit of using hierarchies in this case is in the possibility to obtain a flat model (or models) by cutting the hierarchy not horizontally but across multiple layers. The level of details is selected per element of the system, which gives high control on adjusting the precision of the obtained models, ultimately leading to the best sized models for the given fidelity requirement.

An *elementary transformation* is the minimum set of changes that may happen between two graphs without violating the rule set of OGs. Thus, OGs have the following types of elementary transformations, shown in Fig. 2.7:

Inclusion: Vertices and edges of the lower order are mapped into a single vertex in the higher order. Figure 2.7a shows vertices a_1, a_2, a_3, and edge e_1 being mapped into vertex a; relation $\langle e_1, a \rangle$ is implied and not drawn. This elementary transformation also appears in conventional hierarchical graphs.

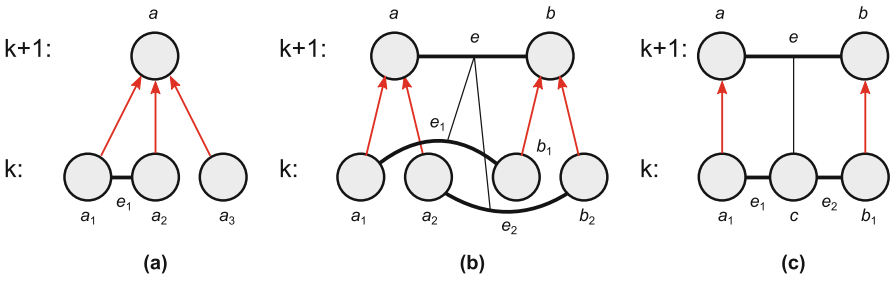

Fig. 2.7 Elementary transformations in Order Graphs and their notation: (**a**) inclusion, (**b**) edge grouping, (**c**) support

Edge grouping: Edges of the lower order are mapped into a single edge in the higher order. Figure 2.7b shows edges e_1, e_2 being mapped into edge e. The relations are drawn as thin black lines to be differentiated from vertex-to-vertex relations. This elementary transformation also appears in conventional hierarchical graphs.

Support: One vertex is mapped into one edge in the higher order. Figure 2.7c shows vertex c being mapped into edge e; relations $\langle e_1, e\rangle, \langle e_2, e\rangle$ are implied and not drawn. This elementary transformation is unique to OGs.

Any transformation $\gamma = G^{(k)} \vdash G^{(k+1)}$ in OG can be represented with a sequence of elementary transformations $\gamma = \gamma_1 \circ \cdots \circ \gamma_n$, or:

$$G^{(k)} \vdash G^{(x_1)} \vdash \cdots \vdash G^{(x_n)} \vdash G^{(k+1)}. \tag{2.6}$$

For two consecutive orders $G^{(k)}, G^{(k+1)}$ of an OG, a *cross-layer cut* $G^{(x)}$ between order k and order $(k+1)$ is a graph, such that $G^{(k)} \vdash G^{(x)} \vdash G^{(k+1)}$ under the same rule set, and $G^{(x)}$ is partially equal to $G^{(k)}$ and $G^{(k+1)}$.

Figure 2.8 explains the above definition. Let $\gamma_a = G^{(k)} \vdash G^{(x)}$ and $\gamma_b = G^{(x)} \vdash G^{(k+1)}$. From $\gamma = G^{(k)} \vdash G^{(k+1)}$ and $G^{(k)} \vdash G^{(x)} \vdash G^{(k+1)}$, it follows that $\gamma = \gamma_a \circ \gamma_b$. Then, $G^{(x)}$ can be split in three parts: $G^{(x)} = g_a \cup g_b \cup g_i$, where g_i is the part that is not changed by γ, so $g_i \subseteq G^{(k)}, g_i \subseteq G^{(k+1)}$; $g_a \subseteq G^{(k)}$ is the part not changed by γ_a, and $g_b \subseteq G^{(k+1)}$ is the part not changed by γ_b. Thus, $G^{(x)}$ contains parts equal to subgraphs of $G^{(k)}$ (namely, g_a and g_i) and subgraphs of $G^{(k+1)}$ (g_b and g_i).

An example of a cross-layer cut can be found in Fig. 2.9.

Making a cut through more than two layers—from $G^{(k)}$ to some $G^{(k+b)}$—can be done iteratively. Firstly, obtain a cut between $G^{(k)}, G^{(k+1)}$, so $G^{(k)} \vdash G^{(x_1)} \vdash G^{(k+1)}$. Then, obtain a cut $G^{(x_2)}$ between newly created $G^{(x_1)}$ and $G^{(k+2)}$, which may now contain parts from $G^{(k)}, G^{(k+1)}$ and $G^{(k+2)}$. Repeat the process until the final cut $G^{(x_{b-1})} \vdash G^{(x_b)} \vdash G^{(k+b)}$ is found.

Cross-layer cuts are models of the same system and are consistent with the layers in the corresponding OG and preserve the connectivity property. The choice, which cut is the most appropriate, depends on the application. Section 2.4.3 presents the use case of parametric-proportional approach to optimise the model size for modelling system power.

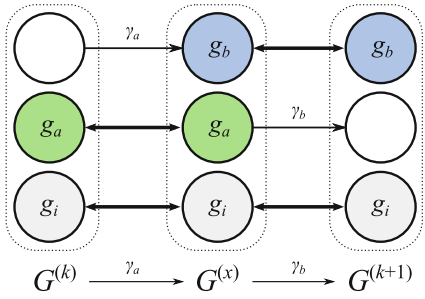

Fig. 2.8 Cross-layer cut $G^{(x)}$ explained

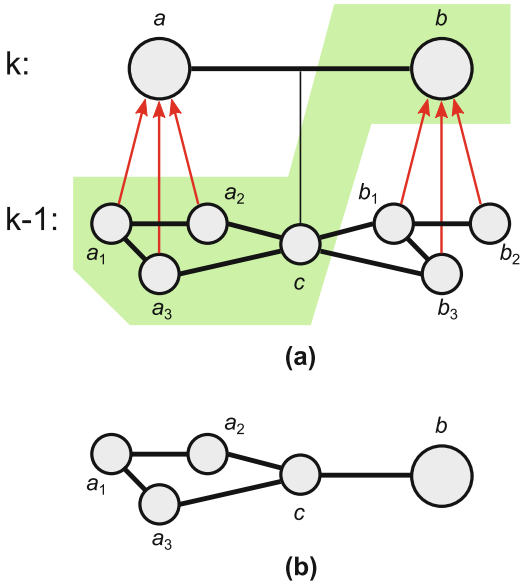

Fig. 2.9 Cross-layer cut example from Fig. 2.6b showing (**a**) the cut and (**b**) resulting flat graph

2.4 Case Studies

This section describes cases of application for our modelling methods. These cases focus on example parameters that can be regarded as resources and we motivate their studies with real applications.

2.4.1 Studying the Performance, Energy and Reliability Trade-Offs of Scalable Systems

In digital VLSI systems, a higher supply voltage (V_{dd}) usually allows a higher operating (clock) frequency and hence a higher throughput, but at the cost of higher power consumption. For reliable operation, there is a limited space within the V_{dd} vs frequency space in which the system can operate, as shown in Fig. 2.10.

Figure 2.10 illustrates the number of limits within which a system can reliably function. There may be a minimum requirement for throughput, which translates to a lower frequency bound a system must achieve. Otherwise the system is not regarded as reliable because it cannot deliver the required performance. There is usually a data reliability requirement which means that the system must operate at a V_{dd} higher than some lower bound because reducing the V_{dd} further the data representation stops being reliably digital and start having unacceptable susceptibility to noise. The VLSI (usually CMOS) technology's fundamental characteristic between its switching speed and V_{dd} means that a circuit cannot be run at a frequency higher

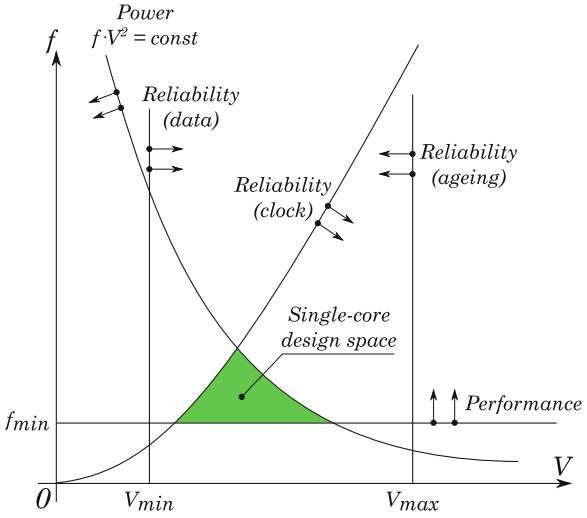

Fig. 2.10 The region of reliable operation

than an upper bound at a particular V_{dd}, in order not to risk combinational logic not completing before the next clock pulse. Any CMOS technology will have a specific highest possible V_{dd} under which it functions correctly. And finally, system power may be limited by some higher bound because of issues like battery life, thermal dissipation, energy efficiency requirements, etc.

Dynamic power is known to be related to switching activity (and through which to system frequency), switching swing voltage (and through which to system V_{dd}) and switching element capacitance (and through which to system size/area—which is a constant before hardware scaling). Lumping all the constants together we can say that power is related to frequency and V_{dd} in the following manner:

$$P = cF \cdot V_{dd}^2, \tag{2.7}$$

where c is a constant, P is the power and F is the frequency. In this section, we explore the issue of core scaling (increasing or decreasing the degree of parallelisation) with an assumption of perfect scaling. Multiplied hardware operating at the same frequency will provide multiplied throughput and require a multiplied amount of power based on the same constant multiplier. Non-zero scaling overheads will be investigated in Sect. 2.4.2. In perfect scaling with a scaling factor of n, the constant c is scaled in the same way, i.e.

$$c = nc_1, \tag{2.8}$$

where c_1 is the c for the hardware before scaling (e.g. a single core). In general, scaling with a factor of n will give a new c which is a factor of n of the unscaled c.

For each core in a new scaled set-up, the available power is also changed by a factor of $1/n$. Considering these factors, for each core, Eq. (2.7) now becomes

$$P_n = cF \cdot V_{dd}^2/n, \tag{2.9}$$

and the overall system power equation with n cores stays the same as (2.7).

Parallelism may be used as a way of increasing throughput without increasing system power dissipation, effectively enlarging the reliable operation region of a system. For instance, if a software application or set of applications can be parallelised and distributed to multiple cores, it will be possible to scale the V_{dd} and the frequency of each core down, while using the multiple cores to improve the overall throughput. Because the dynamic power of CMOS systems is related to V_{dd} and frequency according to Eq. (2.9), reducing both V_{dd} and frequency together reduces power much faster than frequency is reduced, leading to the power-performance advantage of using multiple cores.

Figure 2.11 shows the potential of using parallelisation scaling to reduce power and/or improving performance. The constant power and safe frequency vs V_{dd} operation curves are obtained from experimenting with a real CMOS system [4]. From Fig. 2.11 it can be observed that with a parallelisation scaling factor of 16 and for the constant power budget $P_{\max}$, it is possible to operate at a lower V_{dd} of 0.55 V instead of the nominal 1.2 V, and achieve a nearly four-times overall throughput improvement. On the other hand, if the requirement of throughput is unchanged, it is possible to scale the parallelisation up by the factor of 4, reduce the V_{dd} down to 0.55 V, and use only 25 % of $P_{\max}$ of power.

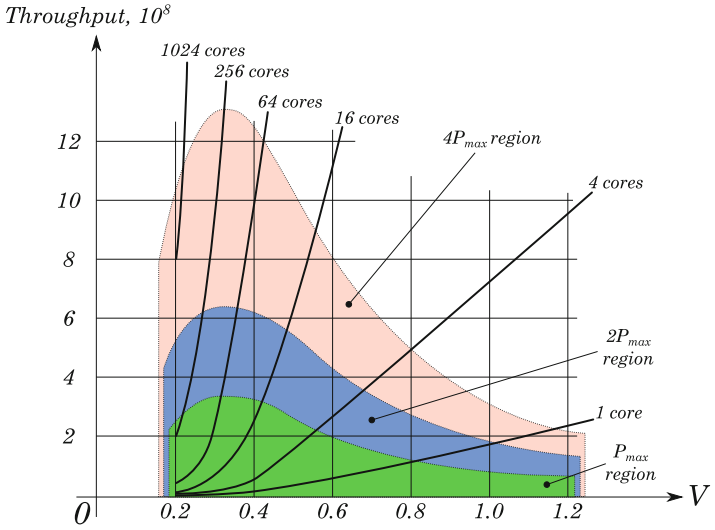

Fig. 2.11 SRAM constant max power curve and scaling lines

Reasoning about these trade-offs can be done in an ad-hoc manner on a per-system or per-circuit bases using characterisation data. However, for system designers it would be much more convenient if this can be done through system-independent models. Resource-driven modelling provides such opportunities as explained in the following sections.

2.4.2 *Exploring Concurrency in Many-Core Systems*

This section presents the example of using flat dynamic resource graphs, presented in Sect. 2.2, to implement a scalable hardware–software co-simulation for exploring concurrency. The analysis of physical parameters is done via resource access counting. The simulator is flexible towards the hardware architecture and facilitates controllable model fidelity by combining resources from different levels of abstraction. The systematic way of determining the choice of the abstraction levels based on fidelity requirements will be presented in Sect. 2.4.3.

2.4.2.1 Architecture-Open (ArchOn) Simulator

The method to supply resource graphs to the simulation software has been derived from our hardware vision. We view the simulator modules as connected via the connectivity fabric, and the simulator input parser works as a router. Table 2.1 shows some commands for this "router", which provide step-by-step graph configurations as well as explicit invocations of resource state transitions. Applying this method to sparsely connected graphs with many vertices gives more compact specifications than traditionally used adjacency matrices. We call it the *graph assembly language*.

Figure 2.12 shows a virtual communication-based hardware architecture that could potentially emulate most real-world systems. This type of architecture is called transport-triggered architecture [7]. It hasn't become popular in general purpose microprocessors, but it appeared attractive for our purposes. Assuming

Table 2.1 Some commands of graph assembly language

Command	Description
U [a] $=$ *value*	Set resource *a* state to *value*
a $\rightarrow$ b	Set a dependency between resources *a* and *b*
a $\xrightarrow{x}$ b	Set a labelled dependency between resources
a $\nrightarrow$ b	Unset a dependency
$G = \emptyset$	Clear all dependencies
go!	"Execute" graph: fire all resource state transitions
go to X	Continue assembly from label X (jump)
if condition **go to** X	Conditional jump

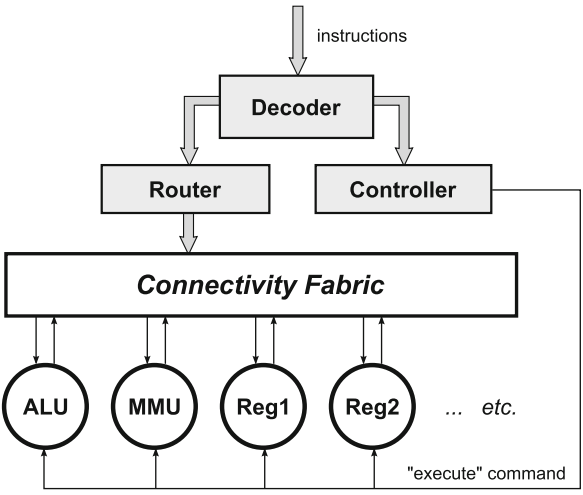

Fig. 2.12 A virtual hardware resembling dynamic resource graphs

that the target system has an instruction set, its software can be recompiled into the connectivity fabric routing commands. The process of executing such software would have alternating phases of configuring the connectivity fabric and executing modules.

2.4.2.2 Benchmark Results

In this section we demonstrate the application of the ArchOn framework to modelling PER trade-offs in a many-core processor. As our example we take the basic computational step in the 3×3 matrix convolution that is used in most image processing applications [23]. Given two 3×3 matrices A and B the goal is to multiply them element-wise and sum up the results, denoted by $A \boxtimes B$:

$$\begin{pmatrix} x_1 & x_2 & x_3 \\ x_4 & x_5 & x_6 \\ x_7 & x_8 & x_9 \end{pmatrix} \boxtimes \begin{pmatrix} y_1 & y_2 & y_3 \\ y_4 & y_5 & y_6 \\ y_7 & y_8 & y_9 \end{pmatrix} = \sum_{1 \le k \le 9} x_k y_k. \tag{2.10}$$

Usually, one of the matrices is a 3×3 sub-matrix of an image being processed and the other matrix, called a *kernel* or a *mask*, represents the required image transformation, e.g., sharpening or edge detection. This step is applied to all 3×3 sub-matrices of a given image, each time producing a value for a pixel in the resulting image. This is an embarrassingly parallelisable computation task: one can cut an image into pieces and process them in parallel on different cores. However, the memory access is still a bottleneck, which will define the system's scalability.

As a many-core test platform we used a simplified ARM architecture. The mainstream ARM processors to date consist of up to 8 cores, however there are

concrete plans to increase the number of cores to 16, 32, and beyond [2, 12]. In ArchOn we ran the benchmark on up to 64 cores, however it was possible to use even more as the simulation time scales linearly with the number of cores. The difference between multi-core and single core architectures for the simulator is in the restrictions on certain connections between the resources belonging to different cores, as described in Sect. 2.2.4.

Algorithm 1 ARM instruction `MLA r8,r9,r10,r8` in graph assembly language

$G = \emptyset$
r9 $\xrightarrow{n}$ mul
r10 $\xrightarrow{m}$ mul
go!
$G = \emptyset$
mul $\xrightarrow{n}$ alu_add
r8 $\xrightarrow{m}$ alu_add
go!
$G = \emptyset$
alu_add $\rightarrow$ r8
go!

The convolution filter software is written in ARM assembly language. Here, a 256×256 image is divided between processing cores, each working on a separate set of pixels (with single pixel wide overlaps). Each pixel is a 32-bit integer representing grey-scale colour. For every ARM instruction we derive a resource evolution and translate it into graph assembly language. This is a routine task since all instructions follow a common pattern. The process of translation can be done automatically. An example instruction is shown in Algorithm 1.

The nature of this simulation requires appropriate functional behaviour resources, thus the computation resources have to be modelled down to the ALU level. On the other hand, the memory access does not require exact modelling of all levels of cache (the behaviour of cache is typically non-deterministic), and can be approximated to the set number of access "modes". Since shared memory would become the bottleneck while scaling to many-cores, we added control over the "criticality" of this resource, so the program can be executed in three different modes: (1) simultaneous read and write access to the memory is allowed, (2) simultaneous read is allowed, but only one writer is allowed at a time, (3) all memory access is exclusive and must be done sequentially. With this example we start exploring non-ideal concurrency scaling and non-zero overheads.

By Amdahl's law [1], the theoretical speed-up that can be achieved by executing a given algorithm on a system capable of executing n threads is

$$\frac{T(1)}{T(n)} = \frac{1}{s_s + \frac{s_p}{n}}, \tag{2.11}$$

Table 2.2 Execution time (in cycles) versus the number of cores running for different memory access models

N cores	Mode 1: multiple read multiple write	Mode 2: multiple read single write	Mode 3: single read single write
1	26607635	26607635	26607635
2	13303827	13303947	**19660819**
3	8938515	8938755	**19660819**
4	6651923	**7864625**	**19660819**
5	5404691	**7864625**	**19660819**
6	4469267	**7864625**	**19660819**

where $T(n)$ is the time an algorithm takes to finish when being executed on n cores, and $s_s \in [0, 1]$ is the fraction of the algorithm that is strictly serial, $s_p = 1 - s_s$ is the fraction of the algorithm that runs in parallel.

For our algorithm s_s and s_p are not known in advance, and actually depend on the memory mode and the number of cores running, i.e. are not constants. ArchOn time estimation enables analysis of this factor. Table 2.2 gives the estimates for execution time. From (2.11) we can find s_p, which will be an estimate of parallelisation for our example. In Mode 1 the scaling is nearly perfect, $s_p \approx 9.999999$, however in Mode 3 the memory becomes such a narrow bottleneck that there is no performance gain for more than two cores (performance cap is shown in bold). The most illustrative example is Mode 2, when multiple cores are allowed to simultaneously read, but forbidden to simultaneously write to the memory. The performance cap is reached at four cores, and s_p varies from 9.96 at two cores to 0.4 at four cores and decreasing.

The main goal of this section is to use ArchOn simulation to draw PER diagrams, described in Sect. 2.4.1. The diagrams can be drawn using the ARM power models, which, however, are typically unavailable. ARM power characterisation from measurements using a prototyping board requires high effort, as shown in Sect. 2.4.3.1. Moreover, the commercially available systems usually do not allow near-threshold and sub-threshold operation. In order to explore PER into these regions, we substitute the power profile with that of an asynchronous SRAM controller [4], which qualitatively reflects the behavior of any CMOS combinational logic in terms of PER. Thus, a hypothetical proprietary ARM core with a wide voltage range should display similar PER relations.

Figure 2.13a illustrates perfect scaling (memory Mode 1) with applied power limit of 2 mW. Figure 2.13b shows the PER diagram for the same system with the same power budget after applying actual metrics for scalability to many cores in Mode 2. The diagram considers only integer numbers of cores, hence the performance line looks jagged. Please note that the line for four cores in Fig. 2.13b is lower than in Fig. 2.13a due to imperfect scaling. The data for this graph is computed automatically. One can see that the performance cap is clearly reflected in the power limit.

Such diagrams can be used in a runtime management system in order to predict the best voltage and the number of cores for a particular software with regard to the

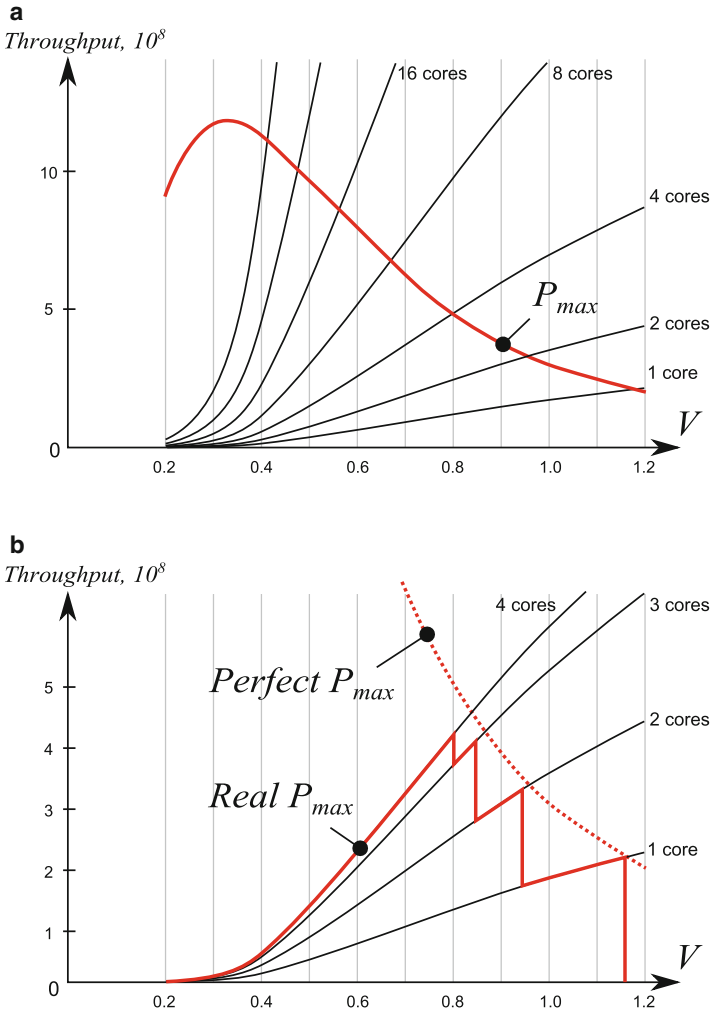

Fig. 2.13 Computed power limit for perfect scaling (**a**) and for an actual scaling to many cores in the simulated example (**b**)

power restrictions. In our example, for memory Mode 2, if the system is limited to 2 mW, the best number of cores is 4 running at 0.8 V.

2.4.3 *Power-Proportional Modelling of Heterogeneous Systems*

This section presents the method of managing fidelity based on power-proportionality metric and cross-layer cuts. Section 2.3 gives the theoretical foundation for this method.

The method can be applied to resource graph simulations in ArchOn. However, in contrast to ArchOn's deterministic simulations, the example method presented in this section uses stochastic modelling and Markov rewards for quantitative analysis of system power consumption. The reward functions are still determined by the resource graphs, as described in Sect. 2.2.3. Since state-space exploration-based analysis has exponential complexity, choosing the right size of the model becomes crucial.

2.4.3.1 Platform Description

The experimental platform used in this section is Odroid XU3 [22].

The main component of Odroid XU3 is the 28 nm 8-core Application Processor Exynos 5422. This System-on-Chip is based on the ARM big.LITTLE architecture [13] and consists of a high performance Cortex-A15 quad-core processor block, a low power Cortex-A7 quad-core block, a Mali-T628 GPU and 2GB LPDDR3 DRAM. The board contains four real-time current sensors allowing the measurement of power consumption on the four separate power domains: A7, A15, GPU and DRAM. Because of the system's heterogeneity and suppled power measurement facilities, the Odroid XU3 is arguably one of the best off-the-shelf heterogeneous platforms for power analysis.

For each domain, the supply voltage and clock frequency can be tuned through a number of pre-set pairs of values. The performance-oriented A15 quad core block can scale its frequencies from 200 to 2000 MHz, whilst the low-power A7 block has a frequency range from 200 to 1400 MHz. Core 0 in the A7 domain has an additional speciality of running the OS kernel and drivers, and it cannot be switched off. We avoid using this core for stress tests and benchmarks to reduce the impact from the OS on the measurements.

There are compatible Linux and Android distributions available for Odroid; in our examples, we used Ubuntu 14.04.

In our characterisation experiments, firstly the above parameters were measured without any additional workload, with only the OS running. Then the same parameters were measured for each core with application threads running. We experimented with the typical Linux stress task, i.e. running square root calculations repeatedly, and in addition, other computations including logarithm calculations and the four arithmetic operations. We also covered different levels of workload.

Another important experiment is the measurement of the same parameters with some of the cores in each block disabled. Odroid allows from one to four of the A15 to be disabled and from one to three of the A7 to be disabled.

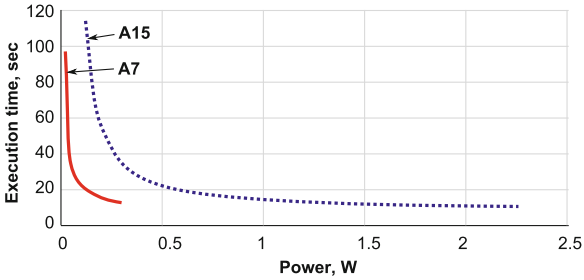

Fig. 2.14 Measured power in relation to the time required to complete a fixed amount of computation. The same performance requires higher power consumption from A15 than A7

In these experiments, it was observed that an A15 consumes four times or more power than an A7 when both are running at the same frequency, and up to an order of magnitude more power when both are running at the same voltage. Figure 2.14 shows the relationship between power consumption and the execution time for the two types of cores on the average running a range of different types of tasks.

These radically different performance and power figures, and their complex relations to the different tasks being executed in a core, validate the approach promoted in this paper. For instance, when certain tasks are mapped to the A7 block, because of the relatively light power demand of these cores we may be able to afford to model such processing with less fidelity, i.e. using a more probabilistic model and/or using a more structurally fuzzy model. For instance, when the A15 block is also running, it may be a good idea to not represent individual A7 cores but to cover the entire A7 block as a meta-core with a single model.

2.4.3.2 Platform Model

In this case study we focus on modelling power consumption of the platform. Two major contributors are task affinities (which task runs on which core) and voltage-frequency pairs.

Figure 2.15 shows the OG model of the system. At the higher levels of abstraction, the system is represented as a set of tasks running on a platform, which in turn consists of a computation component and a power component. The computation resource is provided by A7 and A15 cores, which appear in the lower orders, and the power resource is divided into four power domains, as described in Sect. 2.4.3.1.

For clarity, some of the horizontal edges on this diagram are hidden: every core is actually connected to the corresponding V_{dd} tree and to the task node, etc. Every $\langle$Task, Core X$\rangle$ edge at order 0 represents scheduling of a task onto a certain core. The next order groups these edges into $\langle$Task, A7 cores$\rangle$ and $\langle$Task, A15 cores$\rangle$, respectively. Similarly, every core is connected to the corresponding "V_{dd} tree" resource in the power domain. "V_{dd} tree" resource included in "A15 power domain" supports $\langle$A15 cores, A15 power domain$\rangle$ dependency; the same is true for A7

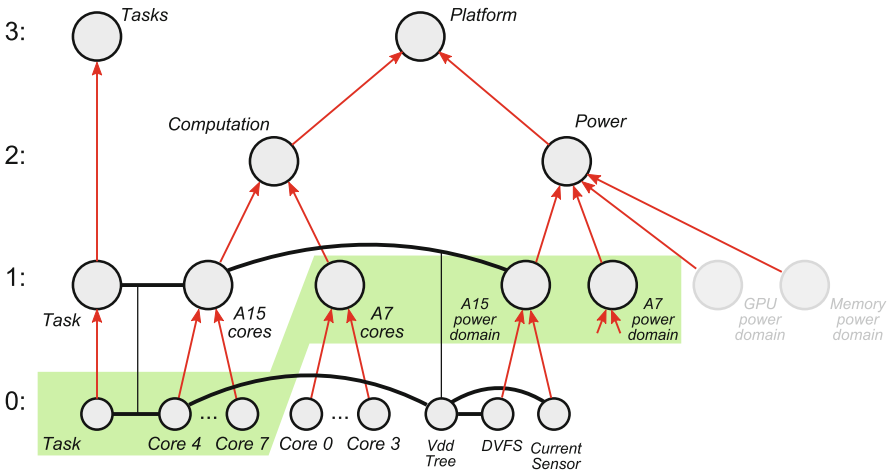

Fig. 2.15 Order Graph model of running tasks on Odriod XU3 platform (some horizontal dependencies are omitted). The *shaded area* marks the cut shown in Fig. 2.17

Cores. In this example, the interactions with "GPU power domain" domain and "Memory power domain" are not captured during the experiments, therefore these resources are removed from the model in the following discussion.

The above OG model represents the structural (static) knowledge of the system. In order to model power, this section uses Stochastic Activity Networks (SANs) [26]. SANs are an extension to Generalised Stochastic Petri Nets (GSPNs) and a more expressive representation language. SANs inherit from Petri nets the basic elements: places, transitions and tokens. In general, a system state is a marking, which intuitively is a configuration of token to place distribution (the number of tokens in each place). A system's dynamic progression consists of its moving from one state to another. This is represented in Petri nets as the firing of a transition. A transition may fire when every one of its input place is marked with at least one token and firing a transition results in a marking modification to a new state, where every input place has its marking reduced by one and every output place has its marking increased by one.

The SANs formalism provides a general way of specifying the enabling of an activity (SANs extension of a Petri net transition), a general way of specifying a completion (firing) rule, a method of representing zero-time events (hence including deterministic as well as stochastic behaviours), a method of representing probabilistic choice in addition to probabilistic delay provided by GSPNs. All of these are based on extensions of the Petri net rules described above. The SANs formalism also provides state-dependent reward values and general delay distributions on activities. For instance, SANs markings can be coded with rewards relating to such parameters as power consumption.

A crucial issue to be modelled for this system, when we talk about system power consumption, is processing, i.e. the execution of threads/tasks in the cores. The fundamental processing element model is shown in Fig. 2.16a. Here the place

Capacity represents the unused capacity of a processing element (e.g. a core), and the place *Processing* represents the current number of tasks being executed in the core. If it is a single core, the sum of tokens of these two places represents the pipeline depth or multi-threading capability of the core. If there are multiple cores in this model, the sum of tokens in the *Capacity* and *Processing* places represents the entire block's multi-threading capability. There is a direct relation between the marking in this model and the resource quantifiers in the graph shown in Fig. 2.5:

$$M\left(Tasks_j\right) = \omega\left(t_j\right) - \sum_i \omega\left(s_{ij}\right),$$

$$M\left(Capacity_i\right) = \omega\left(c_i\right) - \sum_j \omega\left(s_{ij}\right),$$

$$M\left(Processing_{ij}\right) = \omega\left(s_{ij}\right).$$

In the model in Fig. 2.16a, the initial marking, as shown, represents multi-tasking capacity of three and no active processing. The *Spawn* activity is a stochastic activity with a given firing rate and distribution, and because it does not have input places, it is always enabled and can generate tokens into the *Tasks* place. When both *Tasks* and *Capacity* places are marked, the *Start* transition becomes enabled. This is instantaneous transition, thus it fires deterministically and adds a token into *Processing*, whilst removing one token each from *Capacity* and *Tasks*. The input gate of the *Start* transition may contain any additional logic controlling the firing depending on the global of the net. The stochastic transition *Finish* represents the end of processing of a task and returns the token to *Capacity*. The rate of *Finish* represents the throughput related to a single task.

The power consumption is modelled by assigning rewards to markings in appropriate places. For instance, each token in *Processing* can have a reward value corresponding to the power consumption of processing a single task in this core (or cores). A token in *Capacity* can have a reward value associated with the idle power.

Different levels of fidelity are possible with this representation. For instance, the degree of probabilistic vs. deterministic can be tuned for a more or less fuzzy representation. We may decide to model part of a core (i.e. a multiplier), an entire

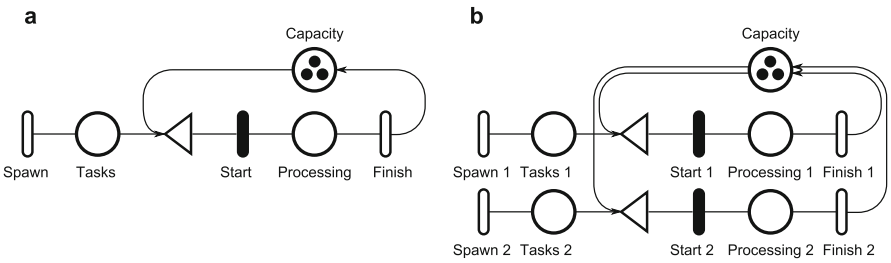

Fig. 2.16 SANs models for task execution

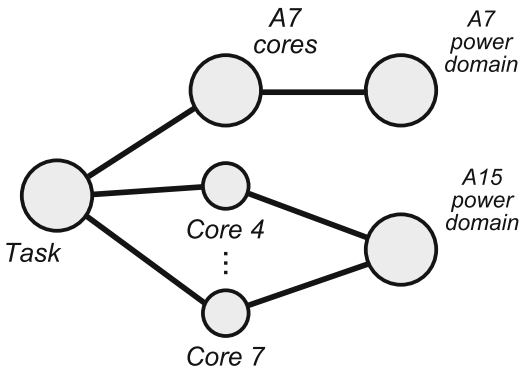

Fig. 2.17 Proposed cross-layer cut for power-proportional modelling

single core, a core-block, or the entire Odroid chip with one of these sub-nets. When setting up a more detailed model with higher fidelity, we may need to distinguish how a processing element behaves with different types of tasks, as our experiments showed that the Odroid cores consume different amounts of power when dealing with different tasks. The model in Fig. 2.16b allows this differentiation by assigning different rewards to *Processing* places corresponding to different tasks. With more fuzzy representations, such issues may be covered by probabilities.

Once a cut has been determined using the OG model, a flat SANs model covering the entire system can be made with different levels of fidelity for different parts. This will be a flat model with power-proportional fidelity and effort.

2.4.3.3 Power-Proportional Model Sizes

Based on experimental data from the Odroid, presented earlier in Sect. 2.4.3.1, for a certain modelling fidelity we may need to represent each A15 core with a model of the type in Fig. 2.16b, with multiple types of tasks—e.g., CPU-heavy and memory-heavy, and many levels of DVFS and workload resolutions. For the same level of fidelity, we can represent the entire A7 block with a single sub-net of the type in Fig. 2.16a without task, DVFS and workload differentiation.

The corresponding OG cut is shown as a flat graph in Fig. 2.17 and also marked with a shaded area in Fig. 2.15. Power-proportional cuts through the model space usually result in models whose sizes are optimal for studying power, in the sense that the resolution or fidelity of power as a parameter is constant through the model. In other words, a power-proportional cut for a specific power representational fidelity gives the smallest possible model for that degree of fidelity. Other representations away from this cut will inevitably result in certain parts showing an unnecessarily higher degree of fidelity leading, usually, to higher degrees of representational complexity.

This approach can be expressed with a metric (p/ec), where p is the power consumption (or any other parameter in study) of a resource, e is the modelling

error produced locally by this resource's sub-model and c is the cost to compute this sub-model. The most power-proportional cut has minimal variance of this metric across all its elements. Typically, reducing the error in the model increases its computational cost, hence the term ec within the same modelling technique can be approximated to a constant. This gives the direct proportionality metric p meaning that each resource in the final cross-layer model should have as approximately the same as possible power consumption as the other resources.

One of the generally accepted metrics of model size and therefore modelling effort and the effort of using models is the size of the state space of the model. For instance, one of the envisaged applications of our modelling method is the design and analysis of runtime parameter management algorithms or machines, e.g. runtime power management for mobile and embedded systems. For such management or control schemes, more sophistication is usually needed to achieve better results. Naïve examples that are widely available in the public and commercial domains, such as such Linux/Android power governors as `ondemand`, usually assume very simplistic plant models and rely on feedbacks to achieve some degree of effectiveness, which is almost never optimal. More sophisticated algorithms such as those based on learning and those providing a degree of adaptation can almost always provide better results than the standard governors, but inevitably require better plant models. On the other hand, most computer system control algorithm designers are most comfortable with thinking of the plant as a state machine. And many types of parameter management algorithms, e.g. learning and model adaptive schemes, depend on a state space representation of the plant being available [8, 9, 18, 27]. The size of the state space of a model, therefore, is directly relevant for this type of model usage.

The example architecture of the type seen in the big.LITTLE Exynos chip featured in the Odroid system consists of N power domains. The kth power domain, $0 \leq k < N$, has d_k DVFS points (pre-set pairs of V_{dd} and clock frequency values) and c_k processing cores of the same type. For such a system and for power studies, the fundamental state element is 'a particular core in a particular power domain is running a particular type of task at a certain workload'. Usually the parameter representational fidelity requirement dictates the granularity of workload representation, which should be constant within each individual power domain, as there is no intra-domain core heterogeneity. This leads the jth core in the kth domain having w_{kj} workload points and t_{kj} types of tasks, where $0 \leq j < c_k$. With no core heterogeneity within a power domain, it is convenient to differentiate workloads and task types into the same numbers of points for all cores, i.e. w_k and t_k. The size of the state space S of a cut model for such a system as the controlled plant, of the type described in the previous sections, is therefore

$$|S| = \prod_{k=0}^{N-1} d_k \left(w_k t_k\right)^{ck} . \tag{2.12}$$

For the Odroid's ARM cores, reducing the representation of the A7 cores to a single execution sub-net model, as in Fig. 2.17, effectively changing c_{A7} from 4 to 1, produces $(w_{A7}t_{A7})^3$ times reduction of the model state space. Reducing the DVFS resolution of the A7 cores to equal power fidelity of the A15 cores produces a further state space reduction.

There are 20 DVFS points for the A15 power domain and 15 for the A7 power domain in the Odroid. Maintaining the same power fidelity, a maximum of five, not 15, DVFS bands are needed for the A7 domain in the model. Assuming a workload resolution of five $(0, 25, \ldots, 100\,\%)$ and task type resolution of two (CPU- and memory-heavy), a $3 \times (5 \times 2)^3 = 3000$ times reduction of the state space (new state space size = 1/3000 or 0.033 % of the original) can be obtained by using a power-proportional cut. This kind of state space reduction may result in qualitative differences in the sophistication of the runtime management scheme given any constant overhead budget for the management, or it can be used to reduce the management overhead whilst maintaining the same degree of management effectiveness.

This modelling method provides more opportunities for runtime tuning and adaptation because cuts may be allowed to dynamically change during runtime. For instance, if it is found that no A15 core is active and the entire A15 block is shut down, power fidelity may be improved by representing the A7s individually by adopting a different cut. This can be necessary because an A15 total shutdown may indicate that the system is running in low power or even survival modes and during these modes what are regarded as small amounts of power during normal operation become significant. This should lead to a higher degree of fidelity in representing the quantity of power in the runtime model. On the other hand, if the A15 total shutdown is purely a result of workload demand reduction, the previous coarse A7 block cut should be entirely satisfactory. The facility of layer-crossing cuts provides additional flexibility for model adaptation.

2.5 Conclusions

This chapter presents a general systematic approach to model complexity control through managing model fidelity. The approach is based on resource-driven modelling and includes two concrete methods. Resource graphs represent the static information of computation systems as sets of resources and their cross dependencies, and the dynamic behaviour of these systems as evolution steps of resources and dependencies. This method allows the straightforward emphasis of elements and issues that can be regarded as resources and their interplay in system models, making it easy for designers to reason about them. Issues such as the level of representational fidelity of parameters can be managed through resource definitions leading to the scalability of models as well as the straightforward reasoning of the scalability of systems themselves.

The second method presented within this resource-driven framework is the use of cross-layer cuts to achieve parameter-proportional fidelity in hierarchical models. In this a previously presented formalism Order Graphs is shown to be effective, and techniques relevant to the derivation of parameter-proportional cuts are presented. We propose a metric for rationalising parameter fidelity across complex models within this context.

The entire approach is tested and validated by a number of application cases, with systems ranging from homogeneous many-core to heterogeneous multi-core, and properties ranging from performance, energy/power and reliability. Models are derived for design-time explorations, and both static and dynamic analysis are made. For analysis, the usefulness of such models is demonstrated through both simulation studies and state space analysis.

Acknowledgements This work is supported by EPSRC grant EP/K034448/1.

References

1. G.M. Amdahl, Validity of the single processor approach to achieving large scale computing capabilities, in *Proceedings of the Spring Joint Computer Conference, AFIPS'67* (Spring) (ACM, New York, 1967), pp. 483–485
2. ARM. http://www.arm.com, 2015
3. G. Balbo, Introduction to generalized stochastic petri nets. *Formal Methods for Performance Evaluation*. Lecture Notes in Computer Science, vol. 4486 (Springer, Berlin, 2007), pp. 83–131
4. A. Baz, D. Shang, F. Xia, A. Yakovlev, Self-timed SRAM for energy harvesting systems. J. Low Power Electron. **7**(2), 274–284 (2011)
5. A. Beyranvand Nejad, A. Molnos, K. Goossens, A unified execution model for multiple computation models of streaming applications on a composable MPSoC. J. Syst. Archit. **59**(10), 1032–1046 (2013)
6. S. Borkar, Thousand core chips: a technology perspective, in *Proceedings of the 44th Annual Design Automation Conference, DAC'07* (ACM, New York, 2007), pp. 746–749
7. H. Corporaal, Design of transport triggered architectures, in *Proceedings on Design Automation of High Performance VLSI Systems* (1994), pp. 130–135
8. A. Das, R.A. Shafik, G.V. Merrett, B.M. Al-Hashimi, A. Kumar, B. Veeravalli, Reinforcement learning-based inter- and intra-application thermal optimization for lifetime improvement of multicore systems, in *Proceedings of the 51st Annual Design Automation Conference*, DAC'14 (ACM, San Francisco, 2014), pp. 1–6
9. A.K. Das, R.A. Shafik, G.V. Merrett, B.M. Hashimi, A. Kumar, B. Veeravalli, Workload uncertainty characterization and adaptive frequency scaling for energy minimization of embedded systems, in *Proceedings of the Conference on DATE'15*, March 2015
10. A. Ehrenfeucht, G. Rozenberg, Zoom structures and reaction systems yield exploration systems. Int. J. Found. Comput. Sci. **25**, 275–306 (2014)
11. H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, D. Burger, Dark silicon and the end of multicore scaling, in *Proceedings of the 38th Annual International Symposium on Computer Architecture, ISCA'11* (ACM, New York, 2011), pp. 365–376
12. EZchip. http://www.tilera.com, 2015
13. P. Greenhalgh, *big.LITTLE Processing with ARM Cortex-A15 & Cortex-A7 – Improving Energy Efficiency in High-Performance Mobile Platforms*. ARM, 2011. White Paper

14. N. Hardavellas, M. Ferdman, B. Falsafi, A. Ailamaki, Toward dark silicon in servers. IEEE Micro **31**(4), 6–15 (2011)
15. B. Kumar, E.S. Davidson, Computer system design using a hierarchical approach to performance evaluation. Commun. ACM **23**(9), 511–521 (1980)
16. Y. Lhuillier, M. Ojail, A. Guerre, J.-M. Philippe, K. Ben Chehida, F. Thabet, C. Andriamisaina, C. Jaber, R. David, HARS: a hardware-assisted runtime software for embedded many-core architectures. ACM Trans. Embed. Comput. Syst. **13**(3s), 102:1–102:25 (2014)
17. Q.-L. Li, Markov reward processes. *Constructive Computation in Stochastic Models with Applications* (Springer, Berlin, 2010), pp. 526–573
18. L.A. Maeda-Nunez, A.K. Das, R.A. Shafik, G.V. Merrett, B. Al-Hashimi, PoGo: an application-specific adaptive energy minimisation approach for embedded systems, in *HiPEAC Workshop on Energy Efficiency with Heterogenous Computing (EEHCO). HiPEAC*, January 2015
19. A. Mokhov, Conditional partial order graphs. PhD thesis, University of Newcastle upon Tyne, School of Electrical, Electronic and Computer Engineering, 2009
20. A. Mokhov, A. Iliasov, D. Sokolov, M. Rykunov, A. Yakovlev, A. Romanovsky, Synthesis of processor instruction sets from high-level ISA specifications. IEEE Trans. Comput. **63**(6), 1552–1566 (2014)
21. A. Nouri, M. Bozga, A. Molnos, A. Legay, S. Bensalem, Building faithful high-level models and performance evaluation of manycore embedded systems, in *In Proceedings of 12th ACM/IEEE International Conference on Methods and Models for System Design, MEMOCODE*, 2014
22. Odroid XU3. http://www.hardkernel.com/main/products, 2013
23. M. Petrou, C, Petrou, *Image Processing: The Fundamentals* (Wiley, Chichester, 2010)
24. A. Rafiev, A. Iliasov, A. Romanovsky, A. Mokhov, F. Xia, A. Yakovlev, Studying the interplay of concurrency, performance, energy and reliability with ArchOn – an architecture-open resource-driven cross-layer modelling framework, in *Proceedings of International Conference on ACSD*, 2014
25. A. Rafiev, F. Xia, A. Iliasov, R. Gensh, A.M.M. Aalsaud, A. Romanovsky, A. Yakovlev, Order graphs and cross-layer parametric significance-driven modelling, in *Proceedings of International Conference on ACSD*, 2015
26. W.H. Sanders, J.F. Meyer, Stochastic activity networks: formal definitions and concepts, in *Lectures on Formal Methods and Performance Analysis* (Springer, Berlin, 2001), pp. 315–343
27. A. Suardi, S. Longo, E.C. Kerrigan, G.A. Constantinides, Robust explicit MPC design under finite precision arithmetic, in *Proceedings of IFAC*, 2014
28. B. Wang, Y. Xu, R. Rosales, R. Hasholzner, M. Glaß, J. Teich, End-to-end power estimation for heterogeneous cellular LTE SoCs in early design phases, in *2014 24th International Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, Sept 2014, pp. 1–8

http://www.springer.com/978-3-319-47306-2

Model-Implementation Fidelity in Cyber Physical System Design

Molnos, A.; Fabre, C. (Eds.)

2017, XII, 236 p. 126 illus., 87 illus. in color., Hardcover
ISBN: 978-3-319-47306-2