Reproducible Evaluation of System Efficiency with a Model of Architecture: From Theory to Practice

Maxime Pelcat, Alexandre Mercat, Karol Desnos, Luca Maggiani, Yanzhou Liu, Julien Heulot, Jean-François Nezan, Wassim Hamidouche, Daniel Menard, Shuvra S Bhattacharyya

Reproducible Evaluation of System Efficiency with a Model of Architecture: From Theory to Practice

Maxime Pelcat*†, Alexandre Mercat*, Karol Desnos*, Luca Maggiani‡†, Yanzhou Liu§, Julien Heulot*, Jean-François Nezan*, Wassim Hamidouche*, Daniel Ménard* and Shuvra S. Bhattacharyya§¶

*INSA Rennes, IETR, UBL, CNRS UMR 6164, 20 av. Buttes de Coësmes, 35708, Rennes, France
†Institut Pascal, CNRS UMR 6602, 4 imp. Blaise Pascal, 63178, Aubière, France
‡Scuola Superiore Sant'Anna, Piazza Martiri della Libertà, 33, 56127, Pisa, Italy
§University of Maryland, College Park 20742, USA
¶Tampere University of Technology, Korkeakoulunkatu 10, 33720 Tampere, Finland

***Abstract*—Current trends in high performance and embedded computing include design of increasingly complex hardware architectures with high parallelism, heterogeneous processing elements and non-uniform communication resources. In order to take hardware and software design decisions, early evaluations of the system non-functional properties are needed. These evaluations of system efficiency require Electronic System-Level (ESL) information on both the algorithms and the architecture.**

Contrary to algorithm models for which a major body of work has been conducted on defining formal Models of Computation (MoCs), architecture models from the literature are mostly empirical models from which reproducible experimentation requires the accompanying software. In this paper, a precise definition of a *Model of Architecture (MoA)* is proposed that focuses on reproducibility and abstraction and removes the overlap previously existing between the notions of MoA and MoC. A first MoA, called the Linear System-Level Architecture Model (LSLA), is presented. To demonstrate the generic nature of the proposed new architecture modeling concepts, we show that the LSLA Model can be integrated flexibly with different MoCs.

LSLA is then used to model the energy consumption of a State-of-the-Art Multiprocessor System-on-Chip (MPSoC) when running an application described using the Synchronous Dataflow (SDF) MoC. A method to automatically learn LSLA model parameters from platform measurements is introduced. Despite the high complexity of the underlying hardware and software, a simple LSLA model is demonstrated to estimate the energy consumption of the MPSoC with a fidelity of 86%.

***Index Terms*—modeling, architecture, hardware/software co-design, performance optimization, design space exploration, system on chip, multiprocessor SoC, power modeling and estimation.**

I. Introduction

In the 1990s, models of parallel computation such as the ones over-viewed by Maggs et al. in [1] were designed to comprehensively represent a system including hardware and software-related features. Since the early 2000s, rapid prototyping initiatives like the Algorithm-Architecture Matching (AAA) methodology [2] have fostered the separation of algorithm and architecture models in order to automate the Design Space Exploration (DSE). Separation of concerns plays a major part in mitigating the design complexity of systems. In particular, the design productivity of Cyber-Physical Systems (CPS), hampered by intricate hardware, application, and external constraints [3], calls for innovative model-based methods.

Several levels of abstraction exist to model a hardware architecture, ranging from the transistor model level to logic gate level, register transfer level, and transaction level. The unprecedented complexity of current systems, embedding billions of transistors, has led to the creation of a higher level of abstraction named Electronic System-Level (ESL) [4]. ESL methods empower designers to perform early analysis and DSE through coarse grain modeling. The added value of ESL methods is testified by company products such as SLX Explorer from Silexica [5] or Pareon from Vector Fabrics [6] whose aims include providing early system efficiency figures.

At the ESL level, the system is decomposed into a behavioral model, expressed with a MoC, and an architecture description, expressed with an MoA [4]. We have proposed in [7] the first precise definition of an MoA removing the existing overlap between the concepts of an MoA and a MoC. Moreover, the LSLA MoA has been introduced in [7] and shown to be the only model fully respecting the proposed definition of an MoA, i.e. the only architecture model capable of providing a reproducible computation of an abstract efficiency cost from an application model respecting a MoC. One may note a difference between system performance and system efficiency. In computer science, performance is often a synonym of throughput [8], [9]. However, system design requires decisions based on many non-functional costs such as memory, energy, latency, or area. In order to evaluate non-functional costs, an MoA must represent the internal behavior of an architecture at a high level of abstraction while offering an evaluation accurate enough to take early design decisions.

Contributions: As an extension of [7], this paper puts MoAs into practice for modeling the energy consumption of an MPSoC. After defining the concepts of MoA and LSLA, the paper covers two new aspects of MoAs: first, a new method is introduced to learn an MoA from measurements of a studied platform; then, the LSLA MoA is shown to predict the energy consumption of a modern MPSoC executing a complex application. This paper demonstrates that, additionally to their formal interest, LSLA, and more generally MoAs, can be applied in practice to evaluate the efficiency of a system.

The paper is organized as follows: Sections II and III introduce the context and related work of MoAs. Then, the LSLA MoA is defined in Section IV and its cost computation mechanism is demonstrated. In Section V, a method is proposed to learn an LSLA model from platform measurements. Finally, the method is applied in Section VI to model the energy consumption of an MPSoC executing a SDF application.

II. The Context of Model-Based Design

MoAs complement the work on MoCs in providing precise semantics for the second input of the Y-chart design method [10]. The Y-chart separates the description of an application from the one of an architecture, as illustrated in Figure 1 where algorithm descriptions, conforming to a precise MoC are combined with architecture descriptions conforming to an MoA. The objective of this paper is to sketch the contours of MoAs as the architectural counterparts of MoCs. This section introduces the MoCs used in Section IV-B to demonstrate the cost computing capability of the proposed LSLA MoA.

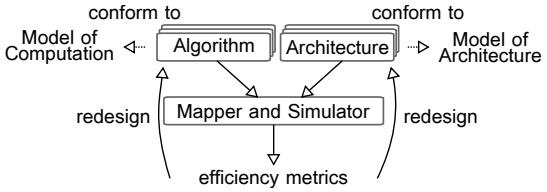


Fig. 1: MoC and MoA in the Y-chart [10].

Many MoCs have been designed to represent the behavior of a system. The Ptolemy II project [11] has a considerable influence in promoting MoCs with precise semantics. Different families of MoCs exist such as finite state machines, process networks, Petri nets, synchronous MoCs and functional MoCs [12]. The LSLA MoA discussed in this paper is demonstrated with both dataflow MoCs and the Bulk Synchronous Parallel (BSP) MoC for their capacity to represent parallel computation. Section II-A presents a static and a dynamic dataflow models while Section II-B introduces the BSP MoC.

A. Dataflow Models of Computation (MoCs)

A dataflow MoC represents an application behavior with a graph where vertices, named actors, represent computation and exchange data through First In, First Out data queues (FIFOs). The unitary exchanged data is called a data token. Computation is triggered when data present on the input FIFOs of an actor respects a set of conditions called *firing rules*. Dataflow MoCs constitute an important class of MoCs targeting the modeling of streaming applications. Dozens of different dataflow MoCs have been explored [13] and this diversity of MoCs demonstrates the benefit of precise semantics and reduced model complexity. To draw a parallel between MoCs and MoAs, the SDF, EIDF and CFDF dataflow MoCs are presented in the next sections.

a) Synchronous Dataflow (SDF): SDF [14] is the most commonly used dataflow MoC [15]. SDF has a limited expressivity and an extended analyzability. Production and consumption token rates set by firing rules are fixed scalars. Static analysis is applied on an SDF graph to determine whether or not fundamental consistency and schedulability properties hold. Such properties, when they are satisfied, ensure that an SDF graph can be implemented with deadlock-free execution and FIFO memory boundedness. An SDF graph (Fig. 2) is defined as $G = \langle A, F \rangle$ where A is the set of actors, and F is the set of FIFOs. For an SDF actor, a positive-integer-valued data rate is specified for each port by the function $rate : P^{in}_{data} \cup P^{out}_{data} \rightarrow \mathbb{N}^*$ where $\mathbb{N}^*$ is the set of strictly positive natural numbers, P^{in}_{data} is the set of all input ports for an actor and P^{out}_{data} is the set of all output ports for an actor. These rates correspond to the fixed firing rules of an SDF actor. A delay $d : F \rightarrow \mathbb{N}$, where $\mathbb{N}$ is the set of natural numbers, is set for each FIFO $f \in F$, corresponding to a number of tokens present initially.

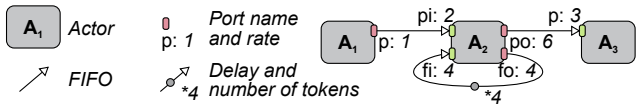


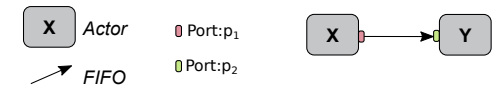
Fig. 2: Example of an SDF Graph.

If an SDF graph is consistent and schedulable, a fixed sequence of actor firings, called *graph iteration*, can be repeated indefinitely to execute the graph, and there is a well defined concept of a minimal sequence for achieving an indefinite execution with bounded memory. The notion of graph iteration is used to compute the cost of mapping an SDF algorithm model on an LSLA architecture model in Section IV-B1.

b) The Enable-Invoke Dataflow (EIDF) and Core Functional Dataflow (CFDF) MoCs: EIDF is a highly expressive form of dataflow MoC that is useful for implementing and analyzing a wide variety of specialized dataflow MoCs [16]. While specialized models such as SDF are useful for exploiting specific characteristics of targeted application domains (e.g., see [17]), the more flexibly-oriented MoC EIDF is useful for integrating and interfacing different forms of dataflow, and providing tool support that spans heterogeneous applications, subsystems, or platforms. In EIDF, the behavior of an actor is decomposed into a set of mutually exclusive actor *modes* such that each actor firing operates according to a mode, and at the end of each actor firing, the actor determines a *next mode set* specifying the set of possible modes according to which the next actor firing can execute. The production or consumption rate for each port is constant for a given actor mode. However, the dataflow behavior for the same port may differ for different actor modes, which allows for specifying dynamic dataflow behavior. An EIDF graph is defined as $G = \langle A, F \rangle$ and notations used to denote actors, FIFOs, and data ports are identical to these defined in the SDF MoC.

This paper uses a restricted form of EIDF called *core functional dataflow* (*CFDF*) (Fig. 3a). CFDF requires that the next mode set that emerges from any actor firing contain *exactly one* mode [18]. This restriction ensures execution

determinacy. The unique element (actor mode) within the next mode set of a CFDF actor firing is referred to as the *next mode* associated with the firing. Dataflow attributes of a CFDF actor can be characterized by a CFDF *dataflow table* (Figures 3b and 3c). The rows of the table correspond to the different actor modes, and the columns to the different actor ports. Given a CFDF actor A, we denote the dataflow table for A by T_A. If m is a mode of A and p is an input port of A, then $T_A[m][p] = -\kappa(m,p)$, where $\kappa(m,p)$ denotes the number of tokens consumed from p in mode m. Similarly, if q is an output port of A, then $T_A[m][q] = \rho(m,q)$, where $\rho(m,q)$ represents the number of tokens produced onto q in mode m. $\kappa(m,p)$ and $\rho(m,q)$ are constant values.



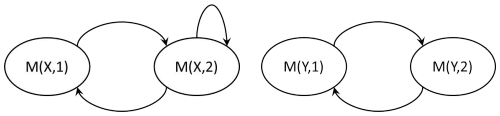
(a) Example of a CFDF graph.

Mode	p_1
1	1
2	2

(b) Dataflow table for actor X.

Mode	p_2
1	-1
2	-4

(c) Dataflow table for actor Y.



(d) Mode transition graph for actor X.

(e) Mode transition graph for actor Y.

Fig. 3: Dataflow attributes of an example CFDF graph.

Mode transition behavior for a CFDF actor can be represented by a *mode transition graph* (Figures 3d and 3e). Given a CFDF actor A, the mode transition graph for A, denoted $MTG(A)$ is a directed graph in which the vertices are in one-to-one correspondence with the modes of A. The edge set of $MTG(A)$ can be expressed as $\{(x,y) \in V_A \times V_A \mid y \in \mu_A(x)\}$, where V_A represents the set of vertices in $MTG(A)$, and $\mu_A(x)$ is the set of possible next modes for actor x. While production and consumption rates for CFDF actor modes cannot be data-dependent, the next mode can be data-dependent, and therefore, $\mu_A(x)$ can in general have any positive number of elements up to the number of modes in A.

The combination of CFDF and LSLA to compute an implementation efficiency will be discussed in Section IV-B2.

B. The Bulk Synchronous Parallel MoC

Another example of an MoC for parallel computation is the Bulk Synchronous Parallel (BSP) [19] MoC. BSP splits up an application into several phases called supersteps. A BSP computation is composed of a set of components A called agents in this paper to distinguish them from the Processing Elements (PEs) in an MoA. Each agent $\gamma \in \Gamma$ has its own memory. An agent γ can access the memory of another agent δ through a remote access (message) $r(\gamma,\delta)$ via a so-called *router*. The computation execution happens in a series of supersteps indexed by $\sigma \in \mathbb{N}$ and consisting of processing efforts, remote accesses and a global synchronization $s(\sigma)$. An example of a BSP algorithm model is illustrated in Fig. 4.

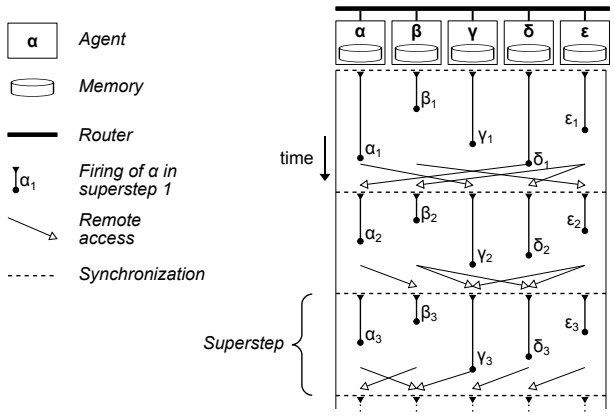


Fig. 4: Example of a BSP Representation.

Each agent γ executes the processing effort γ_σ during the superstep σ. The processing effort γ_σ requires a time $w(\gamma_\sigma) \in \mathbb{N}$ to be processed. During the superstep σ, an agent sends or receives at most $h_\sigma \in \mathbb{N}^*$ remote accesses, each access transferring one atomic data from one agent to another. A barrier synchronization follows each superstep, ensuring global temporal coherency before starting the superstep $\sigma+1$.

BSP provides time performance evaluation for a superstep. A lower bound for the time of a superstep is computed by:

$$T_\sigma = \max_{0 \le \gamma < card(\Gamma)} w(\gamma_\sigma) + h_\sigma \times g + s \qquad (1)$$

where $card(\Gamma)$ is the number of agents, g is the time to execute one atomic remote transfer, and s is a fixed time cost associated to the synchronization. A superstep has a discrete length $n \times L$ with $n \in \mathbb{N}$ and L the minimal period of synchronization. The smaller L is chosen, the closer from the lower bound T_σ the superstep time results.

This cost computation is limited to the latency efficiency metric and assumes that communication costs for an agent are additive. The combination of the BSP MoC and the LSLA MoA will be explained in Section IV-B3, extending BSP cost computation mechanisms.

C. Benefits Offered by MoCs

Each one of the previous MoCs is characterized by a specific set of properties such as their expressiveness, dynamicity, analyzability, or their decidability. Depending on the complexity and constraints of the modeled application, a simple SDF representation or a more complex EIDF or BSP representation can be chosen. MoCs offer abstract representations of applications at different levels of abstraction. They can be used for early system studies or system functional verification. MoCs simplify the study of a system and, since they do not depend on a particular syntax, they offer interoperability to the tools manipulating them.

MoCs, by nature, do not carry hardware related information such as resource limitations and hardware efficiency. In this paper, we use the concept of MoA to complement MoCs in the process of design space exploration.

III. Definition of an MoA and Related Work

A. Definition of Models of Architecture (MoAs)

The main goal of an MoA is to offer mathematically-formulated, reproducible ways to evaluate at an ESL level the efficiency of design decisions. Reproducibility means that the model alone, without an associated implementation, is sufficient to reproduce the cost computation. Following this objective, we introduced a new definition of MoAs [7]:

*Definition 1: A **Model of Architecture (MoA)** is an abstract efficiency model of a system architecture that provides a unique, reproducible cost computation, unequivocally assessing a hardware efficiency cost when processing an application described with a specified MoC.*

An MoA does not need to reflect the real hardware architecture of the system. It aims to represent its efficiency at a coarse granularity. As an example, a complete cluster of processors in a many-core architecture may be represented by a single Processing Element (PE) in its MoA representation, hiding the internal structure of this PE. Hardware architecture models at ESL level that have been proposed in the literature do not comply with Definition 1 because they do not state a cost computation procedure. These models, qualified as *quasi-MoAs* in the rest of the paper, do not guarantee reproducible cost computation. At the ESL level, operating system and middleware may be abstracted together, as demonstrated on an example in Section VI. MoAs can be used at all stages of the system design process, from early steps (e.g. to define how many hardware coprocessors are necessary) to late steps (e.g. to optimize runtime scheduling). An MoA should be as independent as possible from algorithm-related concerns. For this purpose, application activity is defined in the next section as an interface between a MoC and its executing MoA.

B. Application Activity as an Interface between MoC and MoA

As introduced in [7], the notion of application activity is necessary to ensure the separation of MoC and MoA.

Definition 2: Application activity $\mathcal{A}$ corresponds to the amount of processing and communication necessary for accomplishing the requirements of an application. Application activity is composed of processing and communication tokens.

Definition 3: A quantum q is the smallest unit of application activity. There are two types of quanta: processing quantum q_P and communication quantum q_C.

Two distinct processing quanta are equivalent, thus represent the same amount of activity. Processing and communication quanta do not share the same unit of measurement. As an example, in a system with a unique clock and Byte addressable memory, 1 cycle of processing can be chosen as the processing quantum and 1 Byte as the communication quantum.

Definition 4: A token $\tau \in T_P \cup T_C$ is a non-divisible unit of application activity, composed of a number of quanta. The function $size : T_P \cup T_C \rightarrow \mathbb{N}$ associates to each token the number of quanta composing the token. There are two types of tokens: processing tokens $\tau_P \in T_P$ and communication tokens $\tau_C \in T_C$.

The activity $\mathcal{A}$ of an application is defined as the set:

$$\mathcal{A} = \{\mathcal{T_P}, \mathcal{T_C}\} \tag{2}$$

where $T_P = \{\tau_P^1, \tau_P^2, \tau_P^3...\}$ is the set of processing tokens composing the application processing, and $T_C = \{\tau_C^1, \tau_C^2, \tau_C^3...\}$ is the set of communication tokens composing the application communication. An example of a processing token can be a run-to-completion task with static activity. The task is composed of N processing quanta (e.g. N cycles). An example of a communication token is a data message. The token is composed of M communication quanta (e.g. M Bytes). Using the two levels of granularity of a token and a quantum, an MoA can reflect the cost of managing a quantum and the overhead of managing a token composed of several quanta. To be computed, application activity may require the definition of a time scope and input data. These concerns are discussed in Section VI-C4.

The activity definition in its present form is sufficient as a basis for LSLA. Activity is generic to several families of MoCs, as will be demonstrated in Section IV-B.

C. Related Work on MoAs

The concept of MoA is evoked in [20] where it is defined as "a formal representation of the operational semantics of networks of functional blocks describing architectures". This definition allows the concepts of MoC and MoA to overlap. As an example, an SDF graph representing a fully specialized system may be considered as a MoC because it formalizes the application. It may also be considered as an MoA because it fully complies with the definition from [20]. This is in contrast to the orthogonalization between MoC and MoA representations that is supported in our proposed modeling framework. Table I references architecture models of abstract heterogeneous parallel architectures. A general idea of the level of abstraction of each model is given, as well as some properties.

Model	Abstr-action	Distributed Memory	Obj-ectives	Reprodu-cible cost
HVP [21]	- - -	no	time	no
UML Marte [22]	- -	yes	multiple	no
AADL [23]	-	yes	time	yes
[24]	-	yes	multiple	no
[25]	+	yes	multiple	yes
[26]	+	yes	time	no
[27]	++	yes	multiple	no
S-LAM [28]	++	yes	time	no
LSLA	+++	yes	abstract	yes

TABLE I: Properties of different state of the art architecture models.

High-level Virtual Platform (HVP) [21] is a virtual platform based on SystemC. The MoC that can be coexplored by HVP is the *Communicating Processes* one [29]. The HVP platform virtually executes tasks and defines task automata for managing the internal behaviour of application tasks over time. A virtual platform differs from an MoA, as it builds a functional platform rather than a formal model.

The Architecture Analysis and Design Language (AADL) language [23] defines a syntax to describe both software and

hardware components in a system with an objective of time simulation. In contrast to this approach, MoAs offer abstract features for describing hardware architectures and delegate responsibility for modeling algorithms to MoCs.

UML MARTE [22] is a system modeling standard offering a holistic approach encompassing all aspects of real-time embedded systems. The standard consists of UML classes and stereotypes. MARTE does not standardize how a cost should be derived from the specified amount of hardware resources and non-functional properties. Conversely, MoAs focus on abstract cost computation and can be used in the context of MARTE.

Castrillón and Leupers define in [24] a *quasi-MoA* that represents an architecture with a graph G of PEs where each edge interconnecting a pair of PEs is associated to a Communication Primitive (CP). A CP is an application programming interface that is used to communicate among *tasks*. PEs and CPs have sets of properties. The model does not comply with Definition 1 because it does not specify how a cost should be computed from these properties.

In [26], Grandpierre and Sorel define a quasi-MoA for message passing and shared memory data transfer simulations of heterogeneous platforms. Memory sizes and bandwidths are taken into account in the model. This model is also considered as a quasi-MoA because cost computation is not specified.

The System-Level Architecture Model (S-LAM) [28] quasi-MoA focuses on timing properties of a distributed system and defines communication enablers such as Random Access Memory (RAM) and Direct Memory Access (DMA). S-LAM is focused on time modeling and does not provide a reproducible cost computation procedure.

In [25], Kianzad and Bhattacharyya present the CHARMED co-synthesis framework and its architecture model. The CHARMED framework aims at optimizing multiple system parameters represented in Pareto fronts. The model is composed of a set of PEs and Communication Resources (CRs). Each PE has a vector of attributes representing the area and price of the processor, the size of data and instruction memories, and the idle power consumption. Each CR also has an attribute vector including the power consumption per each unit of data, the idle power consumption, and the worst case transmission rate. This model constitutes, to our knowledge, the closest model to the concept of MoA as stated by Definition 1. However, it does not abstract the computed cost, limiting the model to the defined metrics.

Some architecture description languages have been voluntarily omitted because they operate at a different level of abstraction than MoAs. For instance, VHDL is a language to model a hardware behavior but its extreme versatility does not orientate the designer towards a specific Model of Architecture. In the next section, the LSLA MoA is explained. This model provides simple semantics for computing an abstract cost from the mapping of an application described with a precise MoC.

IV. The LSLA Model of Architecture

A. LSLA Definition

LSLA composing elements are illustrated in Fig. 5. An LSLA model is composed of Processing Elements, Communication Nodes and Links. LSLA is linear because the computed cost is a linear combination of the costs of its components.

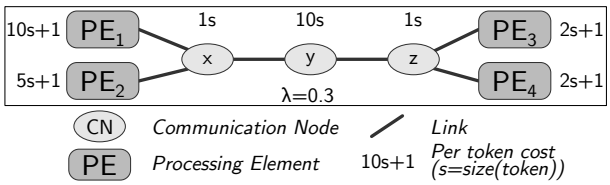


Fig. 5: LSLA MoA semantics elements.

Definition 5: The Linear System-Level Architecture Model (LSLA) is a Model of Architecture (MoA) that consists of an undirected graph $\Lambda = (P, C, L, cost, \lambda)$ where:

- P is the set of architecture Processing Elements (PEs). A PE is an abstract processing facility. A processing token t_P must be mapped to a PE $p \in P$ to be executed.
- C is the set of architecture Communication Nodes (CNs). A communication token t_C must be mapped to a CN $c \in C$ to be executed.
- $L = \{(n_i, n_j) | n_i \in C, n_j \in C \cup P\}$ is a set of undirected links connecting either two CNs or one CN and one PE. A link models the capacity of a CN to communicate tokens to/from a PE or to/from another CN.
- $cost$ is a function associating a cost to different elements in the model. The cost unit is specific to the non-functional property being modeled. It is in nJ in the energy-centered study of Section VI. Formally, the generic unit is denoted ν.
- $\lambda \in \mathbb{R}$ is a Lagrangian coefficient setting the Computation to Communication Cost Ratio (CCCR), i.e. the cost of a single communication quantum relative to the cost of a single processing quantum.

On the example displayed in Fig. 5, PE_{1-4} represent Processing Elements (PEs) while x, y and z are Communication Nodes (CNs). As an MoA, LSLA provides reproducible cost computation when the activity $\mathcal{A}$ of an application is *mapped* onto the architecture. The cost related to the management of a token τ by a PE or a CN n is defined by:

$$\begin{array}{rcccl} cost & : & T_P \cup T_C \times P \cup C & \rightarrow & \mathbb{R} \\ & & \tau, n & \mapsto & \alpha_n.size(\tau) + \beta_n, \\ & & \alpha_n \in \mathbb{R}, \beta_n \in \mathbb{R} & & \end{array} \quad (3)$$

where α_n is the fixed cost of a quantum when executed on n and β_n is the fixed overhead of a token when executed on n. For example, in the use case developed Section VI, α_n and β_n are respectively expressed in $energy/quantum$ and $energy/token$, as the cost unit ν represents energy. A token communicated between two PEs connected with a chain of CNs $\Gamma = \{x, y, z...\}$ is reproduced $card(\Gamma)$ times and each occurrence of the token is mapped to 1 element of Γ. This procedure is explained on different examples in Section IV-B. In following figures representing LSLA architectures, the size of a token $size(\tau)$ is abbreviated into s and the affine equations near CNs and PEs (e.g. $10s + 1$) represent the cost computation related to Equation 3 with $\alpha_n = 10$ and $\beta_n = 1$.

A token not communicated between two PEs, i.e. internal to one PE, does not cause any cost. The cost of the execution of application activity $\mathcal{A}$ on an LSLA graph Λ is defined as:

$$cost(\mathcal{A},\Lambda) = \begin{array}{l} \sum_{\tau \in T_P} cost(\tau, map(\tau)) + \\ \lambda \sum_{\tau \in T_C} cost(\tau, map(\tau)) \end{array} \quad (4)$$

where $map : T_P \cup T_C \rightarrow P \cup C$ is a surjective function returning the mapping of each token on one of the architecture elements.

B. Computing the cost of an application execution on an LSLA architecture

While CNs with high cost in a LSLA model (such as y in Fig. 6) represent *bottlenecks* in the architecture, i.e. communication media with low data rates, PEs with high cost (such as PE_1 in Fig. 6) represent processing facilities with limited processing efficiency. For example, the LSLA model at the bottom of Figure 6 may represent a set of two processors $\{PE_1, PE_2\}$ and $\{PE_3, PE_4\}$ where $\{PE_1, PE_2\}$ has a core PE_1 and a coprocessor PE_2, and $\{PE_3, PE_4\}$ is a homogeneous bi-core processor with high efficiency (cost of 2 for each firing). PE_2 is almost twice as efficient as PE_1 (cost of $5s + 1$ instead of $10s + 1$ for each token). $\{PE_1, PE_2\}$ and $\{PE_3, PE_4\}$ are communicating through a link that has one tenth of the efficiency of internal $\{PE_1, PE_2\}$ and $\{PE_3, PE_4\}$ communications (cost of 10ν instead of 1ν for each token). The next sections illustrate the cost computation provided by LSLA when combined with SDF, CFDF, and BSP MoCs.

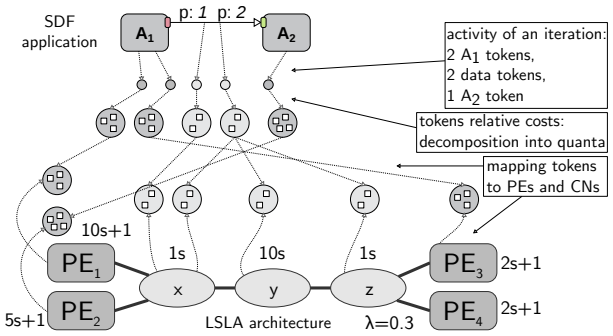


Fig. 6: Computing the cost of executing an SDF graph on an LSLA architecture. The obtained cost for 1 iteration is 31 + 21 + 0.3 (2 + 2 + 20 + 2) + 7 = 66.8 ν (Equation 4).

1) Computing the cost of an SDF application execution on an LSLA architecture: The cost computation mechanism of the LSLA MoA is illustrated by an example in Fig. 6 combining an SDF application model with 2 actors A_1 and A_2 and an LSLA architecture model with 4 PEs PE_{1-4} and 3 CNs x, y and z (Section IV-A). Each actor firing during the studied graph iteration is transformed into one processing token. Each dataflow token transmitted during one iteration is transformed into one communication token. A token is embedding several quanta (Section IV-A), allowing a designer to describe heterogeneous tokens to represent firings and messages of different sizes.

Activity computation consists first in choosing a measurable metric for each token and quantum representing its computational or communication burden. If time is chosen as the metric for computation activity, a computation token corresponding to an actor completed in $4ms$ can for instance be associated with 4 quanta and the unit for all computation quanta is then the millisecond. Another actor completed in $3ms$ would receive 3 quanta. If memory is chosen for communication activity, a message of 2MBytes can be represented by 2 quanta and the unit for all computation quanta is then the MByte. In this particular example, λ is expressed in ms/MBytes. Activity computation is illustrated on a use case in Section VI-C4.

In Fig. 6, each firing of actor A_1 is associated with a cost of 3 quanta and each firing of actor A_2 is associated to a cost of 4 quanta. Communication tokens represent 2 quanta each. The natural scope for the cost computation of a couple (SDF, LSLA), provided that the SDF graph is consistent, is one SDF graph iteration (Section II-A).

Each processing token is mapped to one PE. Any heuristic or manual mapping method can be used to choose the appropriate mapping. Communication tokens are "routed" to the CNs connecting their producer and consumer PEs. For instance, the second communication token in Fig. 6 is generating 3 tokens mapped to x, y, and z because the data is carried from PE_3 to PE_2. It is the duty of the mapping process to verify that a link $l \in L$ exists between the elements that constitute a communication route. The resulting cost, computed from Equations 3 and 4, is 66.8ν. This cost is reproducible and abstract, making LSLA an MoA.

2) Computing the efficiency of a CFDF execution on an LSLA architecture: For dynamic dataflow models, such as CFDF, a simulation-based integration is a natural way to apply MoA-driven cost computation since there is in general no standard, abstract notion of an application iteration — i.e., no notion that plays a similar role as the periodic schedules (and their associated repetitions vectors) of consistent SDF graphs.

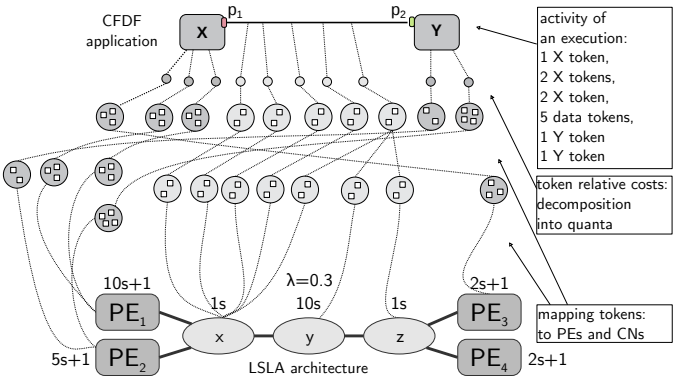


Fig. 7: Computing the cost of executing a CFDF graph on an LSLA architecture. The obtained cost for the chosen simulation scope is $62 + 32 + 9.6 + 7 = 110.6\nu$.

Fig. 7 illustrates an example of execution of a CFDF dataflow graph on an LSLA architecture. We define the cost of execution of actor X in mode $M(X, 1)$ to be 3 quanta and

the cost of execution of actor X in mode $M(X,2)$ to also be 3 quanta. Similarly, the cost of execution of actor Y in mode $M(Y,1)$ is 2 quanta and the cost of execution of actor Y in mode $M(Y,2)$ is 4 quanta. These choices represent additional information associated with the CFDF MoC.

The cost of communication tokens on the FIFO is set to 2 quanta. We can then compute a cost for every PE and CN. There are 2 actor tokens mapped to PE PE_1. Each of them has 3 quanta. The cost for PE PE_1 is $2\times(3\times10+1)=62\nu$. There are 2 actor tokens mapped to PE PE_2. They represent 2 and 4 quanta respectively. The cost for PE PE_2 is $1\times(2\times5+1)+1\times(4\times5+1)=32\nu$. There is 1 actor token mapped to PE PE_3. It represents 3 quanta. The cost for PE PE_3 is $1\times(3\times2+1)=7\nu$. There is no actor token mapped to PE PE_4. Therefore, the cost for PE PE_4 is 0ν. There are 5 communication tokens mapped to CN x. Each of them has 2 quanta. Therefore, the cost for CN x is $5\times(2\times1)=10\nu$. There is 1 data token mapped to CN y. It has 2 quanta. Therefore, the cost for CN y is $1\times(2\times10)=20\nu$. There is 1 data token mapped to CN z. It has 2 quanta. Therefore, the cost of z is $1\times(2\times1)=2\nu$. Since a multiplication by $\lambda=0.3$ brings the cost of communication tokens to the processing domain, the total cost for communication would be $0.3\times(10+20+2)=9.6\nu$. Therefore, the obtained cost is the summation of all PEs' cost and CNs' cost, which in this example sums up to $62+32+9.6+7=110.6\nu$.

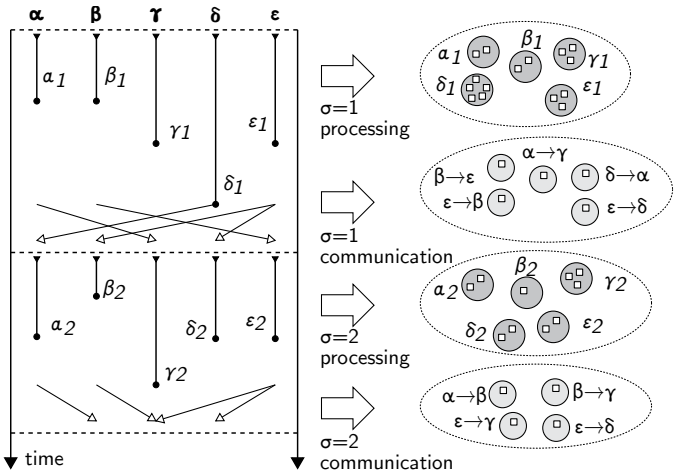


Fig. 8: Extracting the activity of a BSP model.

3) Computing the efficiency of a BSP execution on an LSLA architecture: Figures 8 and 9 illustrate the cost computation of the execution of a BSP algorithm on an LSLA architecture. Figure 8 displays the extraction of the activity, consisting of processing and communication tokens, from the BSP description. Each processing effort α_σ is transformed into one processing token consisting of $w(\alpha_\sigma)$ quanta (Section II-B) and each remote access is transformed into one communication token of one quantum. This size of one quantum is chosen because the BSP model considers atomic remote accesses.

Figure 9 shows the mapping and pooling of tokens, consisting on associating tokens to PEs and CNs and replicating communication tokens to route the communications. Agents α and β are mapped on core PE_2, agent γ is mapped on core PE_1, agent ϵ is mapped on core PE_3 and agent δ is mapped on core PE_4. The global cost is computed as the sum of the cost of each token on its PE or CN. The communication token $\alpha\rightarrow\beta$ is ignored because it is communicating a token between two agents mapped on the same PE and such a communication is supposed to have no specific cost in LSLA, because there is no remote access. An abstract cost of 144.6ν

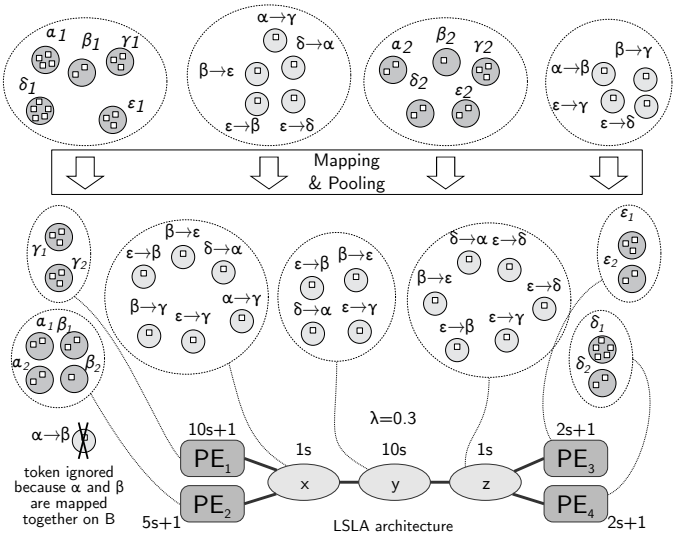


Fig. 9: Computing the cost of executing the BSP model in Figure 8 on an LSLA architecture. The obtained cost is 31 + 31 + 11 + 11 + 11 + 6 + 0.3 (6 + 40 + 6) + 7 + 5 + 11 + 5 = 144.6 ν (Equation 4).

is obtained for this couple (BSP, LSLA) and, as for SDF and CFDF, this cost is reproducible as long as the activity extraction from the BSP model follows the same conventions. Compared to using BSP alone, combining it with LSLA helps studying the cost of mapping several agents on a single PE, exploiting parallel slackness to balance activity between PEs.

In previous sections, the cost computation mechanisms of LSLA have been demonstrated on static SDF dataflow, dynamic CFDF dataflow and BSP MoCs. This generic and reproducible cost computation makes LSLA an MoA.

4) Discussion on LSLA cost computation: The cost computed by LSLA and resulting from communication and processing is linear w.r.t. the number of tokens (Equation 4). This cost can represent an energy, an area, a price, an amount of memory, etc., depending on the purpose of the architecture model. As a simple MoA, LSLA is not aware of the schedule (i.e. order of execution) of tokens. This is a limitation of LSLA introduced in exchange for model simplicity. Compared to models presented in Section III-C, LSLA is the only model abstracting the computed cost type.

Previous sections have defined the notion of Model of Architecture (MoA) and introduced an MoA named LSLA. In the next sections, a method is proposed to infer the parameters of an MoA from platform measurements. The method is then applied and evaluated by modeling the energy consumption of a multicore embedded processor using the LSLA MoA.

V. Learning an LSLA Model from Platform Measurements

This section introduces a method to learn parameters of LSLA from hardware measurements of the MoA-modeled cost. The method being based on algebra, the next section presents an algebraic representation of an LSLA model.

A. Algebraic Expression of costs in an LSLA Model

Let us consider an LSLA model with fixed topology, i.e. the sets P, C and L of respectively PEs, CNs and Links are fixed. The parameters α_n and β_n are initially unknown and will be learnt from measurements of the modeled non-functional property on the platform (e.g. energy). The parameters of an LSLA MoA are gathered in a vector $\boldsymbol{m}$ of size 2η such that:

$$\boldsymbol{m} = (\alpha_n, \forall n \in P \cup C; \beta_n, \forall n \in P \cup C). \tag{5}$$

The size of 2η is due to the concatenation of token- and quanta-related parameters. An arbitrary order is thus chosen for PEs and CNs and the per-quantum costs α_n and per-token costs β_n are concatenated in a unique vector.

B. Applying Parameter Estimation to LSLA Model Inference

Parameter estimation [30] consists of solving an inverse problem to learn the parameters of a model from real-life measurements. In the case of LSLA, the relationship between activity and cost is assumed to be linear and the inverse problem is solved by a linear regression. A series of measured cost $\boldsymbol{d}$ can be ideally expressed as the result of the following forward problem:

$$\boldsymbol{d} = \boldsymbol{G}\boldsymbol{m} + \epsilon, \tag{6}$$

where $\boldsymbol{d} = (d_1, ..., d_M)^T$ is a set of M cost samples (e.g. energy samples), $\boldsymbol{m}$ is the vector of 2η costs defined in Eq. 5, ϵ is the measurements noise resulting in the error vector $\epsilon = (\epsilon_1, ..., \epsilon_m)$, and each line $\boldsymbol{G_k} \in \boldsymbol{G}$ corresponds to an activity vector containing the number of quanta and tokens mapped to the corresponding PEs or CNs for a sample d_k. G_k can be decomposed into:

$$\begin{aligned}\boldsymbol{G_k} = (&\sum size(\tau), \forall \tau \in M_k(n_1);\\ &\sum size(\tau), \forall \tau \in M_k(n_2); ...; \sum size(\tau), \forall \tau \in M_k(n_\eta);\\ &card(M_k(n_1)); card(M_k(n_2)); ...; card(M_k(n_\eta)))\end{aligned} \tag{7}$$

where $M_k : P \cup C \rightarrow T_P \cup T_C$ is the mapping function for experiment k that associates to each PE or CN the set of tokens executed by this component. $card$ refers to the cardinality of the considered set, i.e. the number of tokens while the sum of sizes return the number of quanta. In Eq. 7, the LSLA Lagrangian coefficient λ has been fixed to 1. λ affects problem conditioning by balancing communication and processing costs. Choosing $\lambda = 1$ is possible if the user can choose the quanta units to balance these costs. In the experimental setup developed in the rest of this paper, λ is fixed to $1\ ns/Byte$, assuming communications at around $1\ GBytes/s$. An advantage for the demonstration is that LSLA α coefficients for communications can be directly interpreted as $J/Byte$. A more precise determination of λ could be obtained by setting it as the average communication speed, learnt by benchmarking the platform. The number of *communication* quanta in $\boldsymbol{G_k}$ would then be multiplied by λ for each CN in Eq. 7.

In order to obtain reliable parameter values, the system is overdetermined by performing more measurements than there are parameters in the model, i.e. $M >> 2\eta$. Furthermore, the error vector ϵ is assumed as random variable with zero mean μ_ϵ and constant standard deviation σ_ϵ among samples. From the forward problem in Equation 6, we can derive the Ordinary Least Square (OLS) solution to the inverse problem [30]:

$$\boldsymbol{m_{L2}} = (\boldsymbol{G^T G})^{-1} \boldsymbol{G^T d}. \tag{8}$$

This equation performs the training of the model. $\boldsymbol{m_{L2}}$ is thus a set of parameters α_n and β_n, deduced from measurements d, that can be entered in the LSLA model. For a new system activity $\boldsymbol{G'}$, cost evaluation is computed with:

$$\boldsymbol{d^{LSLA}} = \boldsymbol{G' m_{L2}}. \tag{9}$$

This equation performs the prediction of the cost based on the LSLA model and on the application activity. The residual error of the prediction can be evaluated as follows:

$$\epsilon_m = \boldsymbol{d_m^{LSLA}} - \boldsymbol{d}_m \quad m = 1, ..., M \tag{10}$$

where the error term is expressed as the deviation between measures and the trained model. Such residuals represent the measures' variability that is not considered in the regression model (e.g., correlated side-effect among measures) [31]. In section VI-D2, the impact of the error term ϵ on the trained model is empirically evaluated. In the next section, parameter inference is put into practice for predicting the energy consumption of an MPSoC.

VI. Experimental Evaluation with the LSLA MoA of the Energy Consumption in an MPSoC

A. Objective of the Study and Modeled Hardware Architecture

We intend to model with LSLA the dynamic energy consumption when executing an application, modeled with SDF, on an MPSoC running at full speed where the number of cores reserved for the application is tuned. The motivation for this study lies in the hypothesis that dynamic energy consumption depends additively on application activity.

The modeled architecture is an Exynos 5422 processor from Samsung. This processor is integrated in an Odroid-XU3 platform that offers real-time power consumption measurements of the cores and memory. The Exynos 5422 processor embeds 8 ARM cores in a big.LITTLE configuration. Four of the cores are of type Cortex-A7 and form an *A7 cluster* sharing a clock with frequency up to 1.4GHz. The four remaining cores are of type Cortex-A15 and form an *A15 cluster* with frequency up to 2GHz. An external Dynamic Random Access Memory (DRAM) of 2GBytes is connected as a Package on Package (PoP). A Linux Ubuntu Symmetric Multiprocessing (SMP) operating system is running on the platform. Four Texas

Instruments INA231 power sensors measure the instantaneous power of the A7 cluster, the A15 cluster, the Graphics Processing Unit (GPU) and the external DRAM memory. The energy consumed by the GPU is left out of the scope of the paper but its modeling with an MoA constitutes a promising extension. Power values are read from an I^2C driver. A lightweight script runs in parallel to the measured program, forces the processor to run at full speed and reports current and voltage at $10Hz$ during program execution. This data is exported into files to be processed offline. In our experiments, the power measurements from the A7 and A15 clusters and the memory are summed up and used as the energy consumption vector d.

B. Choosing the LSLA topology

We consider a fixed target platform from which a model is learnt. While the parameters set on PEs and CNs are learnt, their number and topology are chosen, based on a prior knowledge of the hardware features. This type of model is qualified as a "hybrid combination of mechanistic and empirical modeling" in [32]. Mechanistic choices are made "from a basic understanding of the mechanics of the modeled system" while empirical modeling corresponds to the set of trained parameters. A method is introduced hereunder to perform the mechanistic choices. The hardware being characterized is assumed to preexist the study. The method is decomposed into:

1) the number of coarse-grain PEs to consider in the study (cores, coprocessors, GPUs...) is determined. One PE is instantiated in the model per considered platform PE,
2) the hardware communication features of the platform for inter-core communication are located (including shared bus, DMA, shared memory, cache coherency management, etc.). If several PEs share the same communication hardware feature, one CN is allocated on the model, connected to all the cores sharing this feature,
3) if communication hardware features, already modeled by CNs, are themselves communicating through "higher-level" hardware communication, a new CN is created and connected to their corresponding CNs,
4) step 3 is repeated until the graph is connected.

Applying this method to the experimental setup, the 8 PEs corresponding to the Exynos 5422 processor cores are first instantiated. Then, as each cluster is connected by a shared memory supporting hardware cache coherency, A7 and A15 clusters are each associated to a CN connecting the 4 cores of the CN. Finally, The ARM ACE (AXI Coherency Extension) higher-level cache coherency protocol, connecting the two clusters, is associated to a CN named ICC. The resulting model is shown at the bottom of Fig. 10.

After this mechanistic model creation, the model may be simplified to reduce its number of parameters. First, two connected CNs may be merged if 1) the set of tokens crossing both CNs is forecast or measured to be equivalent, or 2) one of the 2 CNs is forecast or measured to strongly dominate the other in terms of cost. Moreover, equivalently performing PEs can be merged to simplify the model. Such a model simplification is experimented in Section VI-D7.

C. Experimental Setup

1) Software Tools: Fig. 10 summarizes the experimental setup used to train and test the LSLA MoA of an Exynos 5422 processor from energy measurements. The PREESM dataflow framework [33] is used to generate code for different SDF configurations of a stereo matching application from a Parameterized and Interfaced Synchronous Dataflow (PiSDF) executable specification. PiSDF [34] is an extension to SDF that introduces a hierarchy of composable elements, as well as static and dynamic parameters influencing token production and consumption. The motivation for using a PiSDF description is that, by fixing various values for application parameters, different functional SDF applications are obtained. Once the parameters of the application are fixed, PREESM generates an executable SDF graph that feeds a multicore mapper and scheduler. Mapping and scheduling are automatically computed, based on the list scheduling algorithm of [35]. PREESM then generates a self-timed multicore code for the application that runs on the target platform. The internal code of the actors is manually written in C code. PREESM manages the inter-core communication and allocates the application buffers statically in the *.bss* segment of the executable. PREESM generates one thread per target core and forces the thread to the corresponding core via affinities.

Communication between actors occurs through shared memory with cache coherency. Semaphores are instantiated to synchronize memory accesses. The whole procedure of mapping, scheduling and generating code with PREESM is scripted. For the current experiment, scripts have been developed to automate large numbers of code generations, compilations, application executions and energy measurements. An application activity exporter has also been added to PREESM that computes the activity for each core, from which α_n and β_n LSLA parameters are learnt. Finally, once its parameters have been learnt, the LSLA model of the platform can be used, together with application activity information, to predict the energy consumption of the platform.

2) Benchmarked Application: The stereo matching algorithm from [36], shown in its SDF form in Fig. 11, is used for the study. From a pair of views of the same scene, the stereo matching application computes a disparity map, corresponding to the depth of the scene for each pixel. The disparity is the distance in pixels between the representations of the same object in both views. Parameters can be customized such as the size of the input images, the number of tested disparities and the number of refinement iterations in the algorithm. These parameters allow for various configurations and application activities to be created. The tested configurations for this study are summarized in Tab. II. The size of the obtained SDF graph is stated, as well as its maximum speedup in latency if executed on a homogeneous architecture with an infinite number of Cortex-A7 cores and costless communication. The stereo matching application is open source and available at [37].

Below each actor in the SDF graph of Fig. 11 is a repetition factor indicating the number of actor executions during a graph iteration. This number is deduced from the data production and

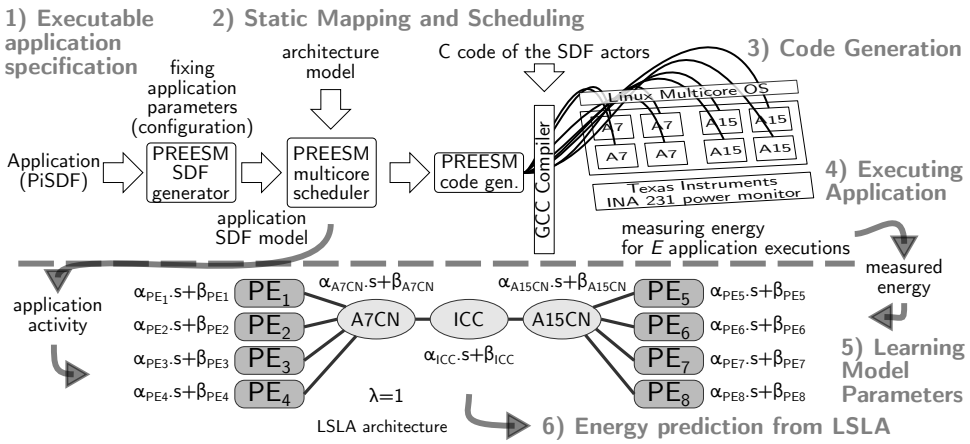


Fig. 10: Experimental setup for inferring the LSLA execution energy model of a Samsung Exynos 5422 MPSoC.

TABLE II: Configurations of the stereo matching application employed to assess the energy modelling.

Configuration ID	1	2	3	4	5	6
input image size	450×375				90×75	270×225
# disparities	30	2	15	60	60	60
# iterations	4	2	3	4	4	4
# of actors	177	67	134	297	317	317
total # of FIFOs	560	102	323	1040	1050	1050
max. speedup	6×	2.5×	4.7×	6.6×	6.5×	6.6×

consumption rates of actors. Two parameters are shown in the graph: *NbDisparities* represents the number of distinct values that can be found in the output disparity map, and *NbOffsets* is a parameter influencing the size of the pixel area considered for the pixel weight and aggregation calculus of the algorithm. *NbIterations* affects the computational load of actors.

The SDF graph contains 12 distinct actors: *ReadRGB* reads from a file the pixels of an image, *BrdX* is a broadcast actor. It duplicates on its output ports the data token consumed on its input port. It generates only pointer manipulations in the code. *GetLeft* gets the RGB left view of the stereo pair. *RGB2Gray* converts an RGB image into grayscale. *Census* produces an 8-bit signature for each pixel, obtained by comparing the pixel to its 8 neighbors: if the value of the neighbor is greater than the value of the pixel, the signature bit is set to 1, and otherwise to 0. *CostConstruction* is executed once per potential disparity level. By combining the two images and their census signatures, it produces for each pixel the cost of matching this pixel from the first image with the corresponding pixel in the second image shifted by a disparity level. *ComputeWeights* produces 3 weights for each pixel, using characteristics of neighboring pixels. *AggregateCosts* computes the matching cost of each pixel for a given disparity. *DisparitySelect* produces a disparity map by computing the disparity of the input cost map from the lowest matching cost for each pixel. *RoundBuffer* forwards the last disparity map consumed on its input port to its output port. *MedianFilter* applies a 3×3 pixels median filter to the input disparity map to smooth the results. The filter is data parallel and 15 occurrences of the actor are fired to process 15 slices in the image. Finally, *Display* writes the depth map in a file. The SDF description of the algorithm provides a high degree of parallelism since it is possible to execute in parallel the repetitions of the three most computationally intensive actors: *CostConstruction*, *AggregateCosts*, and *ComputeWeights*.

The generated application code is compiled by GCC with $-O3$ optimization. For each configuration, 255 different PE mappings are tested by enabling different subsets of the platform cores. PREESM schedules the application on the subsets with the objective of minimizing application latency.

3) Energy Measurements: Only the dynamic energy consumption is considered in this experiment. All the eight cores are activated and their frequency is fixed at their maximum. Thus, the static power, measured at $2.4362W$ in the given conditions, is subtracted from power samples. $\boldsymbol{d}$ in Eq. 6 is a vector of energy samples expressed in Joules. The energy of an application execution is measured by integrating the instantaneous power consumed by the A7 cluster, the A15 cluster and the memory during application execution time.

The unit being measured and analyzed is one execution of the application, from the beginning of the retrieval of 2 images to the end of the production of a depth map. By varying application parameters and the set of authorized cores, a population of executions is built, modeled and analyzed.

4) Application Activity: The activity of the application must be expressed in terms of tokens and quanta (Section III-B). The stereo matching application is represented by a static SDF graph and the computational loads of its actors do not depend on input data. Its application activity does thus not depend on input data. For supporting a more dynamic application with data-dependent loads and topology, the CFDF MoC could be used (Section IV-B2), and a time scope, as well as training input data, representative of application data, would be necessary to compute activity.

In the code generated by PREESM, each PE runs a loop that processes a schedule of actors and the different PEs are synchronized by blocking messages. Several possibilities arise when choosing the format of tokens and quanta. Using PREESM information, the number of computational tokens on a given PE is set to the number of actor firings onto this PE and the number of communication tokens is the number of messages between actors. Time computational quanta in nanoseconds are used, corresponding to the execution time of the actor on the considered core. They are measured by repeating actor execution and running the C *clock()* function to

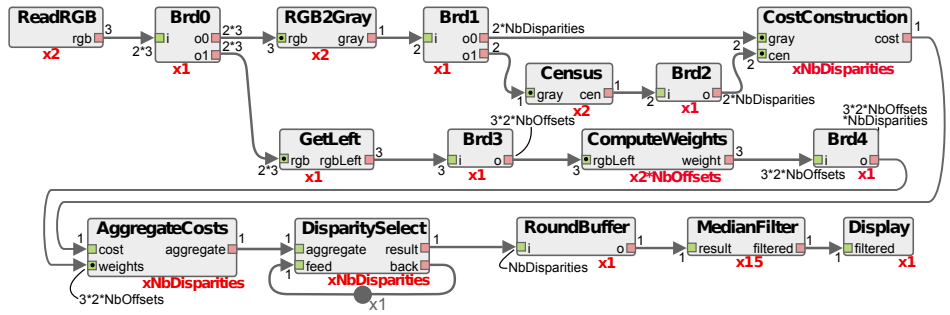


Fig. 11: Illustration of the stereo matching application graph. The number of duplications of each actor is specified. All rates are implicitly multiplied by the picture size.

retrieve timings. This operation is automated in the PREESM tool. As an example, the timings of actors for application configuration 4 are shown in Tab. III. Communication quanta correspond to the size of exchanged messages (in Bytes).

Instead of time quanta, the per-actor computational energy could be used. Each computation quantum could correspond, for instance, to $1mJ$ of energy to execute the actor on the considered core. Such an approach requires each actor to be characterized in energy. Activity focuses on particular aspects of a design while ignoring others. For instance, application-related GPU and cache activities are not modeled in the chosen application activity and they are also ignored in the MoA. As the energy of cores is measured independently from the energy of the GPU, the model can ignore its presence. However, the multiport caches with hardware coherency management are being measured and their activity depends on the data flowing between cores. In the built model, the energy of managing a message by cache coherency is assumed to be affine. A more sophisticated MoA could be developed to precise simulation.

TABLE III: Time quanta (in us) per actor type and core type for configuration 4.

Actor name	time on Cortex-A7	time on Cortex-A15	$\frac{t_{A7}}{t_{A15}}$
$ReadRGB$	$1,813$	719	$2.5\times$
$RGB2Gray$	$6,682$	$2,459$	$2.7\times$
$Census$	$6,846$	$2,320$	$3.0\times$
$ComputeWeights$	$85,265$	$32,251$	$2.6\times$
$CostConstruction$	$13,240$	$2,698$	$4.9\times$
$AggregateCosts$	$76,262$	$29,052$	$2.6\times$
$disparitySelect$	$6,192$	$1,128$	$5.5\times$
$Median_Filter$	$4,923$	$2,555$	$1.9\times$
$Display$	$131,638$	$100,411$	$1.3\times$

D. Experimental Results

1) Measuring Computational Dynamic Energy: Each of the six application configurations from Tab. II are scheduled with each of the 255 possible mapping patterns in the Odroid architecture, resulting in $M = 1530$ energy measurements. Having $M = 1530$ measurements for $2\eta = 22$ parameters, the constraint $M >> 2\eta$ stated in Section V-B is respected. The mapping pattern refers to a binary-composed integer representing the currently used subset of cores (1 for PE_1, 2 for PE_2, 3 for PE_1 + PE_2, 4 for PE_3, etc.).

To ensure reliable measures, application iteration is repeated from 10 to 100 times for each measurement. All energy measurements are repeated 10 times to obtain the energy standard deviation. As illustrated in Fig. 12, the average standard deviation of measurements is moderate ($0.21J$, or 2.4%). This low variation shows that energy consumption is stable for a given application activity and motivates for MoA modeling. For each configuration, the first measurements on the left (in a dashed circle on Fig. 12) show less energy than the rest of the measurements of their application configuration on their right. This is due to the fact that PEs 1 to 4 are Cortex-A7 cores and these cores are more energy efficient than Cortex-A15 cores. These samples use only Cortex-A7 cores and, as a consequence, show more energy efficiency. One may note in the third column of Tab. III that the energy efficiency of Cortex-A7 cores comes at the price of a significantly lower speed.

2) Learning the Energy Model with LSLA: Following the experimental setup depicted in Fig. 10 and the learning method from Section V, an LSLA model is inferred from the energy measurements of previous section and from the application activity provided by PREESM.

The learning curve is drawn in Figure 13 to evaluate the test error ϵ_{te} of the model as a function of the number of training points. The measured energy samples are split into two parts: a training set containing between 1 sample and 80% of the samples (1224 samples), and a test set with the remaining 20% of the samples (306 samples). The samples of the training set are randomly chosen. Fig. 13 displays the training root-mean-square (RMS) error and the test RMS error as the number of training samples rises.

The training error ϵ_{tr} is calculated over the training dataset while the test error ϵ_{te} is calculated over the test dataset. The RMS deviations are computed as follows: $RMS_{te} = \sqrt{E\{\epsilon_{te}^2\}}$ and $RMS_{tr} = \sqrt{E\{\epsilon_{tr}^2\}}$.

The model reasonably fits data, as test error lowers rapidly when the number of training samples grows and reaches a plateau at about 150 training samples before stabilizing at $RMS_{te} = 1.37J$. The training error rises until $RMS_{tr} = 1.21J$, showing that, as expected, the model does not capture the entire physical sources of energy consumption, but the rising rate of the training error lowers with the number of training samples.

3) Discussion on the LSLA Model Parameters: In the next experiments, the model is trained over $M = 1224$ samples and the test set is fixed to 306 samples. The data vector d of Eq. 6 is of length 1224, the matrix G is of size 1224×22 and the model

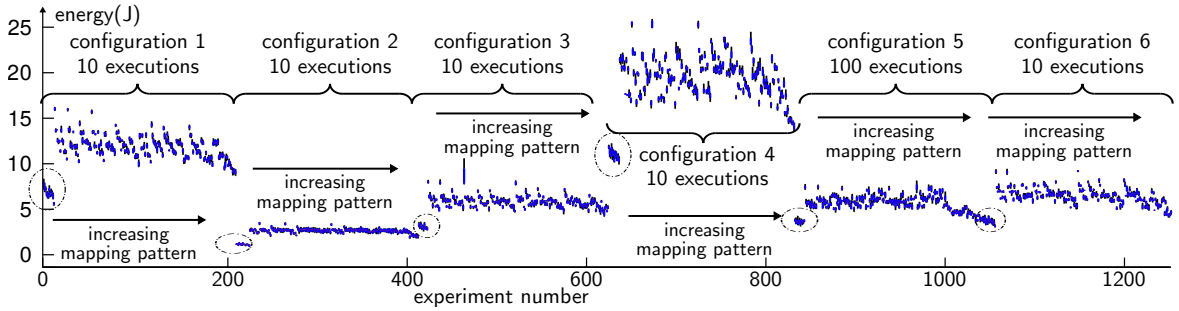


Fig. 12: Training set composed of processor energy measurements. The dynamic of measurements is displayed.

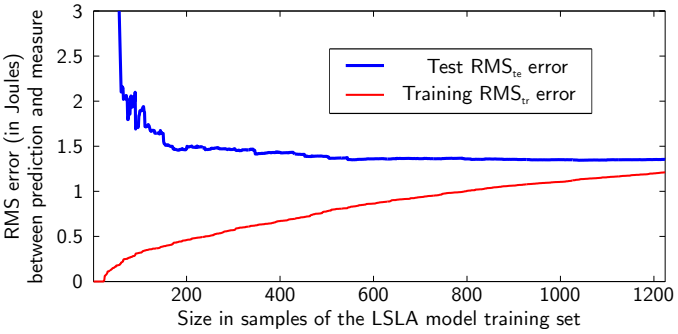


Fig. 13: LSLA dynamic energy model learning curve for a fixed test set of 306 samples and a variable training set of 0 to 1224 samples.

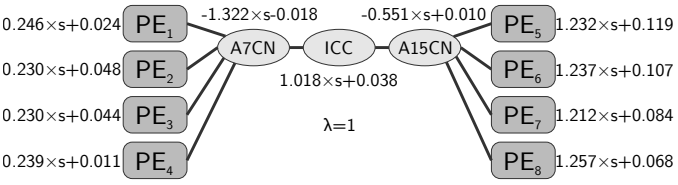


Fig. 14: LSLA dynamic energy model inferred from energy measurements with computational quanta in ns, communication quanta in Bytes, and energy data vector d in nJ. PE_{1-4} are Cortex-A7 cores and PE_{5-8} are Cortex-A15 cores.

vector m is of length 22. The values of the obtained parameters are displayed in Fig. 14. The solid line in Fig. 15 corresponds to the energy predicted with the model from Fig. 14 on the test set. Points correspond to energy samples. The full model offers an energy assessment with a RMS_{te} of $1.37J$, corresponding to an average error of 16%.

Easily explainable parameters in Fig. 14 are α_{PE_1} to α_{PE_8} because they translate into average core execution dynamic power, in $nJ/ns = W$. PEs 1 to 4 have an average dynamic power of $236mW$ and PEs 5 to 8 have an average dynamic power of $1.23W$. These values are credible and correspond to the average dynamic powers of a Cortex-A7 core (PEs 1 to 4) and of a Cortex-A15 core (PEs 5 to 8) running at full speed.

One may observe in Fig. 15 that the last energy samples of each configuration are lower than their prediction with LSLA. This effect can be explained by the intra-cluster parallelism that reduces the execution time of the application without increasing as much the instantaneous power. This intra-cluster parallelism tends to decrease the dynamic energy. This effect

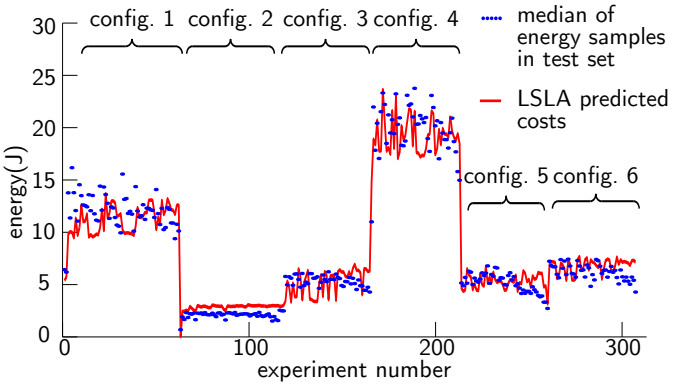


Fig. 15: Comparing the LSLA predicted cost and the median of the corresponding energy measurements in test set.

is partly captured by the learnt negative costs on internal cluster communication quanta $\alpha_{A7CN} = -1.322nJ/Byte$ and $\alpha_{A15CN} = -0.551nJ/Byte$ because more parallelism in a cluster leads in general to more communication in this cluster. However, the amount of communication in a cluster is not fully correlated with the load balancing inside this cluster, leading to errors. The per-quantum cost of ICC $\alpha_{ICC} = 1.018nJ/Byte$ is positive but, as a token flowing through ICC also flows through $A7CN$ and $A15CN$, each inter-cluster exchanged quantum finally costs $1.018 - 1.322 - 0.551 = -0.855nJ/Byte$. As a consequence, the energy gain obtained by parallelizing over the whole processor dominates the energy cost of the communication.

The LSLA model from Fig. 14 does not model the mere hardware. Instead, it represents hardware together with its operating system, the PREESM scheduler and the communication and synchronization library. For example, PREESM tends to favor A15 cores because PREESM optimizes the schedule for latency and, because A15 cores are much faster than Cortex-A7 cores, the demand placed on them is greater. An A15 core is less energy efficient than a Cortex-A7 so the scheduling choices will tend to raise the consumed dynamic energy.

While the average error of the model is substantial, the built LSLA model is characterized by an extreme simplicity, the implementation of the cost computation being reduced to 22 multiplications and 21 additions. Moreover, neither application code nor architecture hardware with low-level representation are needed to compute this model cost. Only a MoC and an MoA are needed, as well as a well defined activity inference method.

TABLE IV: Average and standard deviation of trained LSLA parameters α_n and β_n when the training set is varied.

PE/CN	PE1	PE2	PE3	PE4	PE5	PE6
α_n	0.246	0.230	0.230	0.238	1.239	1.238
$\sigma(\alpha_n)$	2.9%	3.0%	2.7%	3.0%	0.7%	0.7%
β_n	0.027	0.048	0.046	0.012	0.119	0.107
$\sigma(\beta_n)$	45.7%	27.3%	23.9%	82.2%	7.1%	6.8%
PE/CN	PE7	PE8	A7CN	A15CN	ICC	
α_n	1.213	1.258	-1.324	-0.552	1.018	
$\sigma(\alpha_n)$	0.8%	0.6%	1.8%	7.7%	4.6%	
β_n	0.083	0.068	-0.018	0.010	0.038	
$\sigma(\beta_n)$	9.2%	13.1%	27.9%	63.1%	16.8%	

4) Discussion on the Trained LSLA Model Stability: In this section, the stability of the trained LSLA model is tested to account for outliers in training data. To this end, 100 training sets of size 1224 samples are randomly chosen among available data, the rest serving as test set. The standard deviations of parameters $\sigma(\alpha_n)$ and $\sigma(\beta_n)$ in the LSLA model, caused by training set modifications, are reported in Tab. IV. They show that, by far, not all parameters are equivalent in stability. While parameters α_n (applied to quanta) all have moderate standard deviations under 5% (except for $A15CN$ with 7.7%), showing a rather precise determination, parameters β_n (applied to tokens) have in average standard deviations of 30%. This difference shows that the most stable information relevant for energy estimation lies in the number of quanta (in this case, in the execution time of actors). The number of tokens (number of executed actors) is less reliably related to energy.

5) Discussion on the Trained LSLA Model Accuracy: The RMS_{te} prediction error of 16% is provoked by a vast amount of non-modeled factors, including the variable per-actor average power, the application specific scheduling gaps, the memory management, etc. As an example of a non-modeled factor, by using LSLA with time quanta to predict energy, the present analysis assumes the power consumed by a core to be equivalent for each executed actor. However, it is not the case in reality. From low-power (memory-intensive) to high-power (compute-intensive) actors in the stereo matching application, the difference of power consumption is $+55\%$ on A7 cores and $+102\%$ on A15 cores.

As a consequence of non-modeled factors, the learned model presents two weaknesses:

1) Within the presented experimental setup, the model loses its accuracy on applications and configurations that do not appear at all in the training set. The model thus strongly depends on the application and configurations used for its training.
2) One can also observe on Fig. 15 that the local energy variations within a configuration are not precisely captured by the model that has been chosen to cover several application configurations.

As a first solution to these two limitations, a different LSLA model could be learnt for each application type and configuration and the model could be switched at runtime based on application type and configuration. Experiments show that such a strategy improves the accuracy. A tradeoff is then possible between a compact but imprecise unique model and a more precise model set based on configuration- and application-related switching.

Another solution to these limitations consists of creating a new MoA, feeding the activity with more information on actors and communication, and feeding the MoA with more information on hardware. For instance, as the instantaneous power consumption of cores is highly dependent on the current actor, labelling activity tokens with actor types would make it possible for the new MoA to apply different scaling factors based on actor type, making the same MoA applicable to more applicative cases.

The fidelity of an LSLA model is certainly more important than its average error. The next section discusses the fidelity of the inferred LSLA energy model.

6) Fidelity of the LSLA Energy Model: Model fidelity, as presented in [38], refers to the probability, for a couple of data d_i and d_j, that the order of the simulated costs d_i^{LSLA} and d_j^{LSLA} matches the order of the measured costs. The fidelity f of the LSLA energy model is formally defined by

$$f = \frac{2}{M(M-1)} \sum_{i=1}^{M-1} \sum_{j=i+1}^{M} f_{ij}, \tag{11}$$

where M is the number of measurements and

$$f_{ij} = \begin{cases} 1 \text{ if } sgn(d_i^{LSLA} - d_j^{LSLA}) = sgn(d_i - d_j) \\ 0 \text{ otherwise} \end{cases}, \tag{12}$$

with d_i^{LSLA} and d_i respectively the ith LSLA-evaluated and measured energy, and

$$sgn(x) = \begin{cases} (-1) \text{ if } (x < 0) \\ 0 \text{ if } x = 0 \\ 1 \text{ if } x > 0 \end{cases}. \tag{13}$$

The fidelity of the inferred LSLA model for the considered problem is of more than 86%, suggesting that the model can be used for taking energy-based decisions at a system level. As for accuracy, a good fidelity is obtained on condition that the application types and configurations are sufficiently similar between the training and the test set. Solutions, already discussed in Section VI-D5, constitute future research directions.

Fidelity is illustrated by Fig. 16 where measurements have been sorted in ascending order and are displayed together with their LSLA prediction.

7) Simplifying a LSLA Model: As explained in Section VI-B, different LSLA topologies can be used to represent a single platform and metric, for example by merging PEs and CNs. Each cluster of the Exynos 5422 processor having homogeneous cores, a simplified model of the platform has been experimented where PEs of one cluster are undifferentiated. As a consequence, only 2 PEs are retained that each fuse the 4 PEs of one cluster. By doing so, we remove the cost of intra-cluster communication because the new model does not differentiate intra-core communication from intra-cluster communication. The results on the same training and test sets of using the simplified model instead of the original one show a limited degratation of RMS_{tr} ($1.32J$ instead of $1.21J$) and RMS_{te}

($1.49J$ instead of $1.21J$) and a very slight degradation of fidelity (85.8% instead of 86.1%). Such a simplification is thus adequate and reduces cost computation to 6 multiplications and 5 additions.

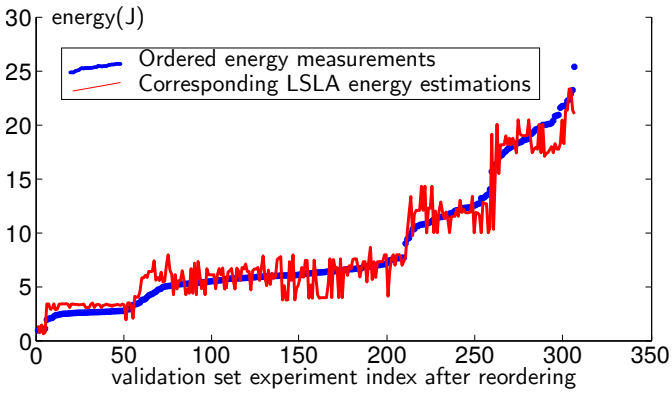


Fig. 16: Test set sorted in ascending order and their corresponding LSLA predictions.

VII. Conclusion

In this paper, a precise definition of a *Model of Architecture (MoA)* has been proposed that makes cost abstraction and computational reproducibility the main features of an MoA. The Linear System-Level Architecture Model (LSLA) MoA has then been defined, compared to the state of the art of architecture models and studied both theoretically and on a use case. LSLA is the first model fully complying the proposed definition of an MoA. LSLA represents hardware performance with a linear model, summing the influences of processing and communication on system efficiency. LSLA has been demonstrated on an example to predict the dynamic energy of an MPSoC executing a complex SDF application with a fidelity of 86%. Additionally, a method for learning the LSLA parameters from hardware measurements has been introduced, automating the creation of the model.

LSLA opens new perspectives in building system-level architecture models that provide reproducible prediction fidelity for a limited complexity. A vast amount of potential extensions exist, including the study of other non-functional properties, systems-of-systems models, and memory hierarchy models.

Acknowledgment

This work is supported by the CERBERO H2020 Project.

References

[1] B. M. Maggs, L. R. Matheson, and R. E. Tarjan, "Models of parallel computation: A survey and synthesis," in *Proceedings of the HICSS Conference*. IEEE, 1995.

[2] T. Grandpierre and Y. Sorel, "From algorithm and architecture specifications to automatic generation of distributed real-time executives: a seamless flow of graphs transformations," in *Proceedings of MEMOCODE Conference*. IEEE, 2003.

[3] M. Masin, F. Palumbo *et al.*, "Cross-layer design of reconfigurable cyber-physical systems," in *Proceedings of the DATE Conference*. IEEE, 2017.

[4] A. Gerstlauer, C. Haubelt *et al.*, "Electronic system-level synthesis methodologies," *IEEE Trans. Computer Aided Des.*, 2009.

[5] *Silexica - https://silexica.com/ (accessed 12/2015).*

[6] *Vector Fabrics - https://www.vectorfabrics.com/ (accessed 12/2015).*

[7] M. Pelcat, K. Desnos *et al.*, "Models of Architecture: Reproducible Efficiency Evaluation for Signal Processing Systems," in *Proceedings of the SiPS Workshop*, Dallas, United States, 2016.

[8] F. Baccelli, G. Cohen *et al.*, *Synchronization and linearity: an algebra for discrete event systems*. John Wiley & Sons, 1992.

[9] N. Bambha, V. Kianzad *et al.*, "Intermediate representations for design automation of multiprocessor DSP systems," *Design Automation for Embedded Systems*, 2002.

[10] B. Kienhuis, E. Deprettere *et al.*, "An approach for quantitative analysis of application-specific dataflow architectures," in *Proceedings of the ASAP Conference*. IEEE, 1997.

[11] J. Eker, J. W. Janneck *et al.*, "Taming heterogeneity-the ptolemy approach," *Proceedings of the IEEE*, 2003.

[12] M. Pelcat, S. Aridhi *et al.*, *Physical Layer Multi-Core Prototyping: A Dataflow-Based Approach for LTE eNodeB*. Springer, 2012.

[13] H. Yviquel, "From dataflow-based video coding tools to dedicated embedded multi-core platforms," Ph.D. dissertation, Rennes 1, 2013.

[14] E. A. Lee and D. G. Messerschmitt, "Synchronous data flow," *Proceedings of the IEEE*, vol. 75, no. 9, 1987.

[15] E. A. Lee and T. M. Parks, "Dataflow process networks," *Proceedings of the IEEE*, vol. 83, no. 5, 1995.

[16] W. Plishker, N. Sane *et al.*, "Heterogeneous design in functional DIF," in *Proceedings of the SAMOS Workshop*, 2008.

[17] S. S. Bhattacharyya, E. Deprettere *et al.*, Eds., *Handbook of Signal Processing Systems*, 2nd ed. Springer, 2013.

[18] W. Plishker, N. Sane *et al.*, "Functional DIF for rapid prototyping," in *Proceedings of the RSP Symposium*, June 2008.

[19] L. G. Valiant, "A bridging model for parallel computation," *Communications of the ACM*, vol. 33, no. 8, pp. 103–111, 1990.

[20] B. Kienhuis, E. F. Deprettere *et al.*, "A methodology to design programmable embedded systems," in *Embedded processor design challenges*. Springer, 2002.

[21] J. Ceng, W. Sheng *et al.*, "A high-level virtual platform for early MPSoC software development," in *Proceedings of CODES+ISSS Conference*. ACM, 2009.

[22] OMG, "A UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems," OMG, Standard, June 2011.

[23] P. H. Feiler, D. P. Gluch, and J. J. Hudak, "The architecture analysis & design language (AADL): An introduction," CMU, Tech. Rep., 2006.

[24] J. Castrillon Mazo and R. Leupers, *Programming Heterogeneous MPSoCs*. Springer, 2014.

[25] V. Kianzad and S. S. Bhattacharyya, "CHARMED: A multi-objective co-synthesis framework for multi-mode embedded systems," in *Proceedings of the ASAP Conference*. IEEE, 2004.

[26] T. Grandpierre and Y. Sorel, "Un nouveau modèle générique d'architecture hétérogène pour la méthodologie AAA," *JFAAA*, 2002.

[27] E. Raffin, C. Wolinski *et al.*, "Scheduling, binding and routing system for a run-time reconfigurable operator based multimedia architecture," in *Proceedings of the DASIP Conference*. IEEE, 2010.

[28] M. Pelcat, J.-F. Nezan *et al.*, "A system-level architecture model for rapid prototyping of heterogeneous multicore embedded systems," in *Proceedings of the DASIP conference*, 2009.

[29] A. Donlin, "Transaction level modeling: flows and use models," in *Proceedings of the CODES+ISS Conference*. ACM, 2004.

[30] R. C. Aster, B. Borchers, and C. H. Thurber, *Parameter estimation and inverse problems*. Academic Press, 2011, vol. 90.

[31] D. C. Montgomery, E. A. Peck, and G. G. Vining, *Introduction to linear regression analysis*. John Wiley & Sons, 2015.

[32] A. D. Pimentel, "Exploring exploration: A tutorial introduction to embedded systems design space exploration," *IEEE Design & Test*, 2017.

[33] M. Pelcat, K. Desnos *et al.*, "PREESM: A dataflow-based rapid prototyping framework for simplifying multicore dsp programming," in *Proceedings of the EDERC Conference*, 2014.

[34] K. Desnos, M. Pelcat *et al.*, "PiMM: Parameterized and interfaced dataflow meta-model for MPSoCs runtime reconfiguration," in *Proceedings of the SAMOS Workshop*, 2013.

[35] Y.-K. Kwok, "High-performance algorithms for compile-time scheduling of parallel processors," *Ph. D. thesis*, 1997.

[36] A. Mercat, J.-F. Nezan *et al.*, "Implementation of a stereo matching algorithm onto a manycore embedded system," in *Proceedings of the ISCAS Symposium*. IEEE, 2014.

[37] K. Desnos and J. Zhang, "PREESM project - stereo matching - svn://svn.code.sf.net/p/preesm/code/trunk/tests/stereo," Dec. 2013.

[38] N. K. Bambha and S. S. Bhattacharyya, "A joint power/performance optimization algorithm for multiprocessor systems using a period graph construct," in *Proceedings of the 13th international symposium on System synthesis*. IEEE Computer Society, 2000.