Models of Architecture: Application to ESL Model-Based Energy Consumption Estimation

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

Models of Architecture: Application to ESL Model-Based Energy Consumption Estimation

Technical Report PREESM/2017-02TR01, 2017

Maxime Pelcat[1,3], Alexandre Mercat[1], Karol Desnos[1],
Luca Maggiani[2,3], Yanzhou Liu[4], Julien Heulot[1],
Jean-François Nezan[1], Wassim Hamidouche[1], Daniel Ménard[1], and
Shuvra S. Bhattacharyya[4,5]

[1]IETR/INSA Rennes, France
[2]Scuola Superiore Sant'Anna, Pisa, Italy
[3]Institut Pascal, Clermont-Ferrand, France
[4]University of Maryland, College Park, USA
[5]Tampere University of Technology, Tampere, Finland

February 10, 2017

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 report, 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%.

1 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).

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) [3]. ESL methods empower designers to perform early analysis and DSE through coarse grain modeling. The added value of ESL methods is demonstrated by company products such as Poly-Platform from PolyCore Software, Inc. [4], SLX Explorer from Silexica [5] or Pareon from Vector Fabrics [6] whose objectives 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 [3]. 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, throughput, 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.

As an extension of [7], this report puts MoAs into practice for modeling the energy consumption of an MPSoC. After defining the concepts of MoA and LSLA, the report covers two new aspects of MoAs: first, a new method is introduced to learn an MoA from physical measurements of a studied platform; then, the LSLA MoA is shown to efficiently predict the energy consumption of a modern MPSoC when executing a complex algorithm. This report 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 report is organized as follows: Section 2 introduces the context and related work of MoAs. Then, the LSLA MoA is defined in Section 3 and its efficiency cost computation mechanism is demonstrated with 3 different MoCs. In Section 4, a method is proposed to learn an LSLA model from platform measurements. Finally, the method is applied in Section 5 to model the energy consumption of an MPSoC while executing an SDF application, before concluding in Section 6.

2 Context and Related Work

2.1 Related Work on MoCs

MoAs complement the work on MoCs in providing precise semantics for the second input of the Y-chart [10] design method. 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. As illustrated in Figure 1, the objective of this report is to sketch the contours of MoAs as the architectural counterparts of MoCs. This section introduces a few MoCs that will be used in Section 3.2 to demonstrate the cost computing capabilities of the proposed LSLA MoA.

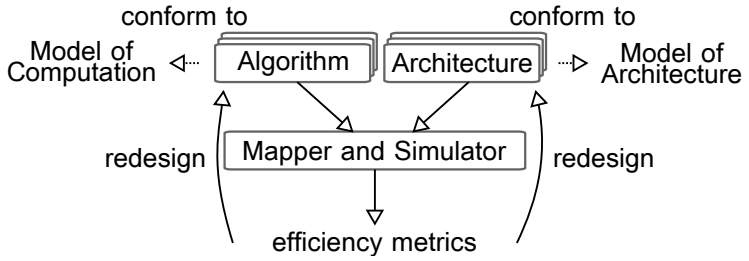


Figure 1: MoC and MoA in the Y-chart [10].

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 report is demonstrated with both dataflow MoCs and the Bulk Synchronous Parallel (BSP) MoC for their capacity to represent parallel computation. Section 2.1 presents a static and a dynamic dataflow models while Section 2.1 introduces the BSP MoC.

Dataflow 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 the data present on the input FIFOs of an actor respect 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, examples of dataflow MoCs, namely SDF, EIDF and CFDF, are presented.

The Synchronous Dataflow (SDF) MoC 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 in an SDF graph and for that reason, SDF is commonly called a static dataflow MoC.

Static analysis can be 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 (Figure 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 fixed, positive-integer-valued data rate is specified for each port by the function $rate : P_{data}^{in} \cup P_{data}^{out} \rightarrow \mathbb{N}^*$ where $\mathbb{N}^*$ is the set of strictly positive natural numbers, P_{data}^{in} is the set of all input ports for an actor and P_{data}^{out} 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 all natural numbers, is set for each FIFO $f \in F$, corresponding to a number of tokens initially present in the FIFO.

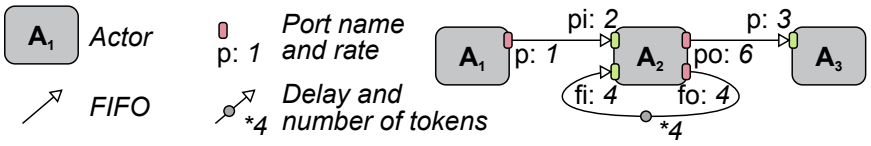


Figure 2: Example of an SDF Graph.

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 will be used to compute the cost of mapping an SDF algorithm model on an LSLA architecture model in Section 3.2.

The Enable-Invoke Dataflow (EIDF) and Core Functional Dataflow (CFDF) MoCs EIDF is a highly expressive form of dataflow MoC that is useful as a common basis for representing, 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.

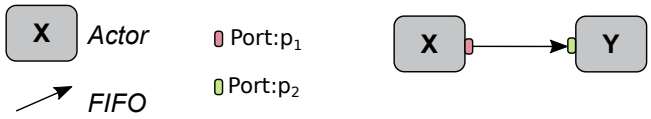
In EIDF, the behavior of an actor is decomposed into a set of mutually exclusive actor *modes* such that each actor firing operates according to a given mode, and at the end of each actor firing, the actor determines a set called the *next mode set*, which specifies the set of possible modes according to which the next actor firing can execute. The dataflow behavior (production or consumption rate) for each actor port is constant for a given actor mode. However, the dataflow behavior for the same port can be different for different modes of the same actor, which allows for specification of dynamic dataflow behavior. EIDF can be qualified as a dynamic dataflow MoC.

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.

In this report, we focus on a restricted form of EIDF called *core functional*

dataflow (*CFDF*) (Figure 3a). CFDF requires that the next mode set that emerges from any actor firing contain *exactly one* mode [18]. This restriction is important in ensuring 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 dataflow table correspond to the different actor modes, and the columns correspond 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.



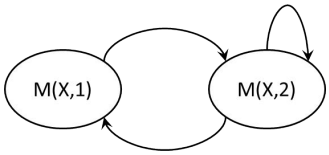
(a) Example of a CFDF graph.

Mode	p_1
1	1
2	2

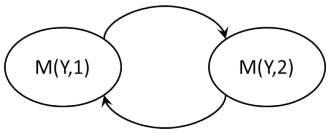
(b) Dataflow table for actor X.

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.

Figure 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 3.2.

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 report 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 Figure 4.

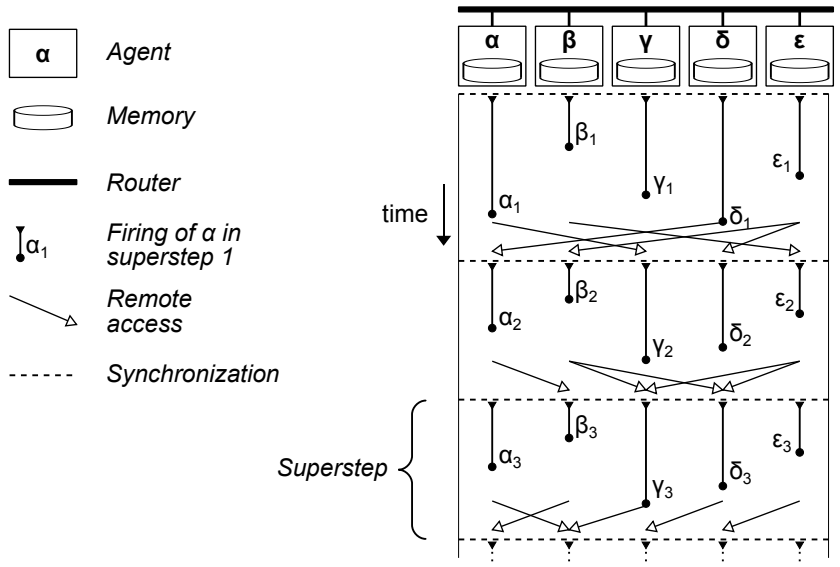


Figure 4: Example of a BSP Representation.

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:

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

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 3.2. Combining the BSP MoC with different MoAs opens opportunities of agent merging and of computating different efficiency metrics when compared to using BSP alone.

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 report, we use the concept of MoA to complement MoCs in the process of design space exploration.

2.2 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.*

As explained by Box and Draper [20], "all models are wrong, but some are useful". An MoA does not need to reflect the real hardware architecture of the system. It only aims to represent its efficiency at a coarse grain. 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 report, do not guarantee reproducible cost computation. At ESL level, operating system and middleware may be abstracted together, as demonstrated on an example in Section 5. 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.

2.3 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 *The application activity $\mathcal{A}$ corresponds to the amount of processing and communication necessary for accomplishing the requirements of the application. The 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 message in a message passing system. 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. The activity definition in its present form is sufficient as a basis for LSLAs. Activity is generic to several families of MoCs, as will be demonstrated in Section 3.2.

2.4 Related Work on MoAs

The concept of MoA is evoked in [21] where it is defined as "a formal representation of the operational semantics of networks of functional blocks describing architectures". This definition is very broad, and 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 [21]. This is in contrast to the orthogonalization between MoC

and MoA representations that is supported in our proposed modeling framework. Moreover, contrary to the definition from [21], our proposed definition of MoA does not compel the MoA to match the internal structure of the hardware architecture, as long as the generated cost is of interest.

Table 1 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	Reproduc-ible cost
HVP [22]	- - -	no	time	no
UML Marte [23]	- -	yes	multiple	no
AADL [24]	-	yes	multiple	no
[25]	-	yes	multiple	no
[26]	+	yes	multiple	yes
[27]	+	yes	time	no
[28]	++	yes	multiple	no
S-LAM [29]	++	yes	time	no
LSLA	+++	yes	abstract	yes

Table 1: Properties of different state of the art architecture models.

High-level Virtual Platform (HVP) [22] defines an evolutionary virtual platform based on SystemC. The corresponding MoC that can be coexplored by HVP is the *Communicating Processes* one [30]. The HVP platform virtually executes tasks and defines task automata for managing the internal behaviour of application tasks over time. HVP targets Symmetric Multiprocessing (SMP) processor architectures and is focused on time simulation. A virtual platform differs from an MoA, as it builds a functional platform rather than a formal model.

UML Marte [23] is a system modeling standard offering a holistic approach encompassing all aspects of real-time embedded systems. The standard consists in Unified Modeling Language (UML) classes and stereotypes. As a specification language, UML Marte does not standardize how a cost should be derived from the specified amount of hardware resources and non-functional properties. MoAs focus on abstract cost computation and can be used in the context of UML Marte.

The Architecture Analysis and Design Language (AADL) language [24] defines a syntax and semantics to describe both software and hardware components in a system. The language constructs match logical and physical features of the described system such as threads and processes for software and bus and memory for hardware. In contrast to this approach, MoAs offer abstract features for describing hardware architectures and delegate responsibility for modeling algorithms to MoCs.

Castrillón and Leupers define in [25] a *quasi-MoA* (Section 2.2) that divides an architecture into PEs. Each PE has a specific Processor Type (PT) defined by $PT = (CM^{PT}, X^{PT}, V^{PT})$ where CM^{PT} is a set of functions associating costs to PEs, X^{PT} is a set of PE attributes such as context switch time of resource limitations, and V^{PT} is a set of variables such as the processor scheduling policy. A graph G of PEs is defined where each edge interconnecting a pair of PEs is associated to a Communication Primitive (CP). A CP is a software application programming interface that is used to communicate among *tasks*. A CP has its

own cost model CM^{CP} associating different costs to communication volumes. It also refers to a Communication Resource (CR), i.e. a hardware module used to implement the communication that knows the number of channels and the amount of available memory in the module. The model does not comply with Definition 1 because it does not specify any cost computation procedure from the data provided in the model.

In [26], 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 (CR). Each PE has a vector of attributes $PE_{attr} = [\alpha, \kappa, \mu_d, \mu_i, \rho_{idle}]^T$ where α denotes the area of the processor, κ denotes the price of the processor, μ_d denotes the size of data memory, μi denotes the instruction memory size and ρ_{idle} denotes the idle power consumption of the processor. Each CR also has an attribute vector: $CR_{attr} = [\rho, \rho_{idle}, \theta]^T$ where ρ denotes the average power consumption per each unit of data to be transferred, ρ_{idle} idle denotes idle power consumption and θ denotes the worst case transmission rate or speed per each unit of data. 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.

In [27], Grandpierre and Sorel define a graph-based 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.

In [28], Raffin et. al, describe a quasi-MoA for evaluating the performance of a Coarse Grain Reconfigurable Architectures (CGRAs). The model is customized for the ROMA processor and based on a graph representing PEs, memories, a network connecting PEs to memories, and a network interconnecting PEs. The model contains memory sizes, network topologies and data transfer latencies. The objective of the model is to provide early estimations of the necessary resources to execute a dataflow application.

The System-Level Architecture Model (S-LAM) [29] 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.

Some architecture description languages have been voluntarily omitted because they operate at a different level of abstraction than MoAs. For instance, behavioral 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.

3 The LSLA Model of Architecture

3.1 LSLA Definition

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

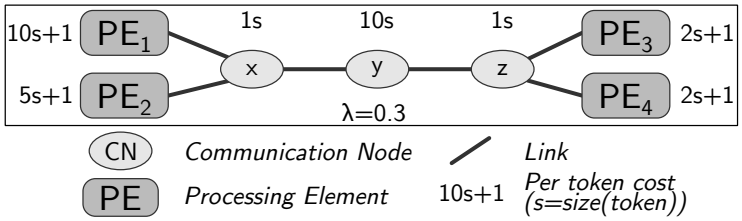


Figure 5: LSLA MoA semantics elements.

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 5. Formally, the generic unit is denoted* ν*.*

On the example displayed in Figure 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 5, 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 3.2. 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 $\alpha_n = 1$.

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:

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

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.

- *$\lambda \in \mathbb{R}$ 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.*

In the next section, the cost computation of LSLA is demonstrated on different MoCs.

3.2 Computing the cost of an application execution on an LSLA architecture

The next sections illustrate the cost computation provided by LSLA when combined with SDF, CFDF, and BSP MoCs (Section 2.1). This combination follows the Y-chart displayed in Figure 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 Figure 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 3.1). 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 3.1), allowing a designer to describe heterogeneous tokens to represent firings and messages of different sizes. In Figure 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 2.1).

Each processing token is mapped to one PE. Communication tokens are "routed" to the CNs connecting their producer and consumer PEs. For instance, the second communication token in Figure 6 is generating 3 tokens mapped to x, y, and z because the data is carried from C to B. It is the responsibility

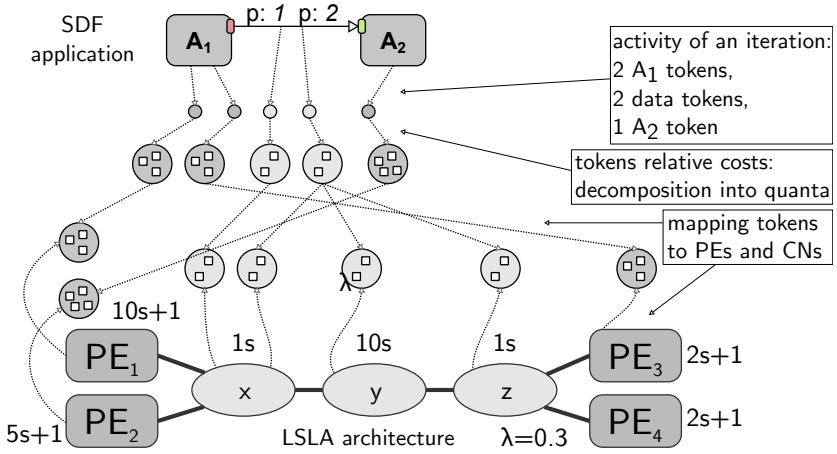


Figure 6: Computing the cost of executing an SDF graph on an LSLA architecture. The obtained cost for 1 iteration is 31 + 21 + 0.3 (2 + 2 + 20 + 2) + 7 = 66.8 ν (Equation 4).

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.

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.

Figure 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 $2 \times (3 \times 10 + 1) = 62\nu$. There are 2 actor tokens mapped to PE PE_2. They represent 2 and 4 quanta respectively. The cost for PE PE_2 is $1 \times (2 \times 5 + 1) + 1 \times (4 \times 5 + 1) = 32\nu$. There is 1 actor token mapped to PE PE_3. It represents 3 quanta. The cost for PE PE_3 is $1 \times (3 \times 2 + 1) = 7\nu$. 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 $5 \times (2 \times 1) = 10\nu$. There is 1 data token mapped to CN y. It has 2 quanta. Therefore, the cost for CN y is $1 \times (2 \times 10) = 20\nu$. There is 1 data token

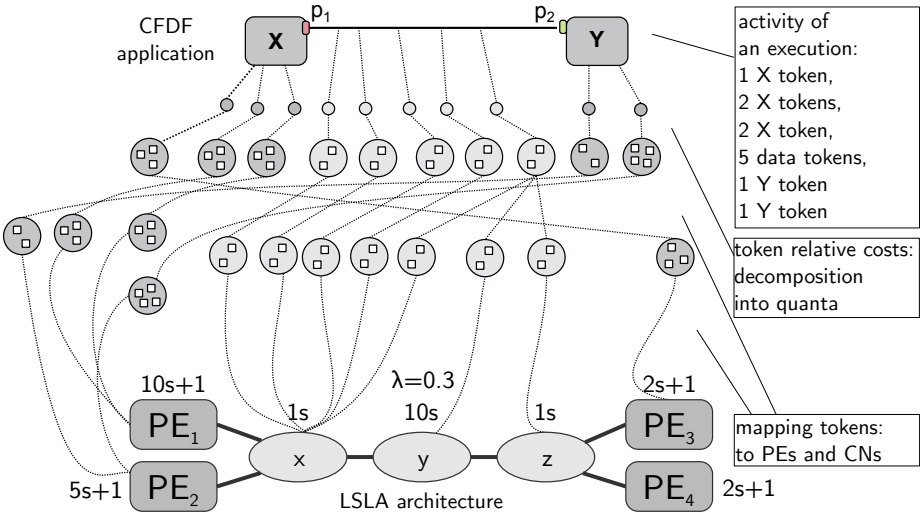


Figure 7: Computing the cost of executing a CFDF graph on an LSLA architecture. The obtained cost for the chosen simulation scope is $62+32+9.6+7 = 110.6\nu$.

mapped to CN z. It has 2 quanta. Therefore, the cost of z is $1 \times (2 \times 1) = 2\nu$. 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 $62 + 32 + 9.6 + 7 = 110.6\nu$.

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 2.1) 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 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. When compared to using BSP alone, combining BSP and LSLA helps studying the cost of mapping several agents on a single PE, exploiting parallel slackness to balance activity between

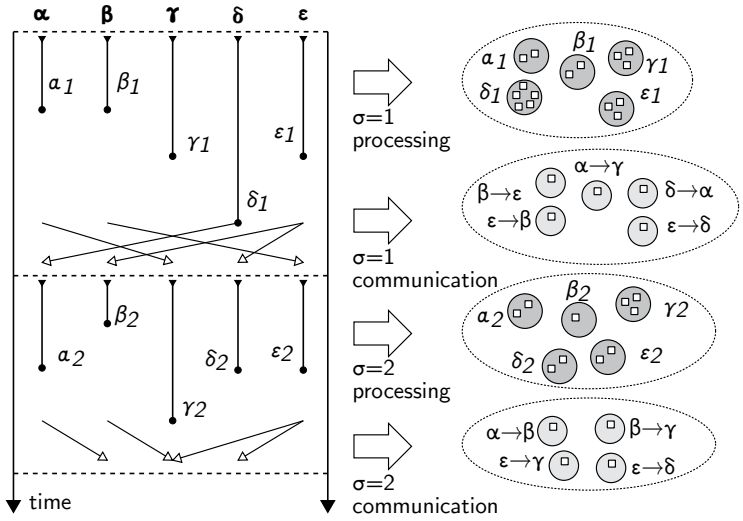


Figure 8: Extracting the activity of a BSP model.

PEs.

In the previous sections, the cost computation mechanisms of LSLA have been demonstrated on static SDF dataflow, dynamic CFDF dataflow and BSP MoCs. The generic and reproducible cost computation of LSLA makes LSLA a Model of Architecture.

Discussion on LSLA cost computation

While CNs with high cost (such as y in Figure 9) represent *bottlenecks* in the architecture, i.e. communication media with low data rates, PEs with high cost (such as PE_1 in Figure 9) represent processing facilities with limited processing efficiency.

For example, the LSLA model at the bottom of Figure 9 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 cost computed by LSLA and resulting from communication and processing is linear w.r.t. the number of communication and processing tokens (Equation 4). This cost can represent a processing time, a diminution in the throughput, an area, a price, an energy, a power, an amount of memory, etc., depending on the purpose of the architecture model.

LSLA differs from the model in [25] in the sense that PEs are not directly linked by edges but rather through communication nodes that can be chained to represent complex PE interconnects. Moreover, LSLA is simpler than the model in [25] and the cost of the internal communication in a PE is taken into account in [25] while it is not taken into account in LSLA. In [26], all different

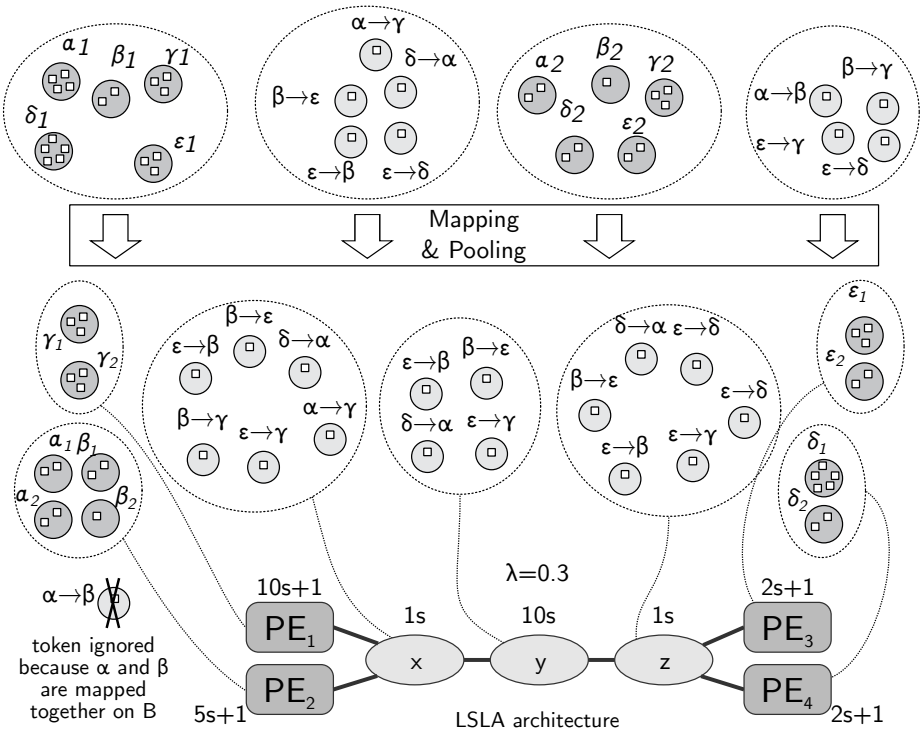


Figure 9: Computing the cost of executing the BSP model in Figure 8 on an LSLA architecture. The obtained cost is 31 + 31 + 11 + 11 + 11 + 6 + 0.3 (6 + 40 + 6) + 7 + 5 + 11 + 5 = 144.6 ν (Equation 4).

costs are represented in a single model whereas LSLA defines a single abstract cost model and several models of the same hardware can be combined for multi-objective optimization. Compared to all models presented in Section 2.4, LSLA is the only model that abstracts the type of the computed implementation cost.

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 automatically 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.

4 Learning an LSLA Model from Platform Measurements

This section introduces a method to learn the parameters of an LSLA MoA 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.

4.1 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 Processing Elements, Communication Nodes 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 consumption). The coefficient λ being a tunable parameter to obtain similar orders of magnitude for computation and communication tokens, it is originally fixed to 1 in this experiment. 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.

4.2 Applying Parameter Estimation to LSLA Model Inference

Parameter estimation [31] 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 supposed 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:

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

where $\boldsymbol{d} = (d_1, ..., d_M)^T$ is a set of M cost measurements (e.g. energy measurements), $\boldsymbol{m}$ is the vector of 2η costs defined in Equation 5, 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 measurement 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 order to obtain reliable parameter values, the system is overdetermined by performing more measurements than there are parameters in the model, i.e. $M >> 2\eta$. From the forward problem in Equation 6, we can derive the least square solution to the inverse problem [31]:

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

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:

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

This equation performs the prediction of the cost based on the LSLA model and on the application activity. In the next section, parameter inference is put into practice for predicting the energy consumption of an MPSoC.

5 Experimental Evaluation with the LSLA MoA of the Energy Consumption in a Samsung Exynos 5422 Processor

In this section, the use of the LSLA MoA to model a system at an ESL level of abstraction is demonstrated on an example.

5.1 Objective of the Study and Modeled Hardware Architecture

We intend to model with LSLA the dynamic energy consumption spent to execute 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 the activity of the system that is a direct consequence of the application activity.

The modeled architecture is an Exynos 5422 processor from Samsung. This processor is integrated in an Odroid-XU3 platform that provides 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 sharing a clock 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 SMP operating system is running on the platform.

Four INA231 power sensors from Texas Instruments measure the instantaneous power of the A7 cluster, the A15 cluster, the Graphics Processing Unit (GPU) and the external DRAM memory. The power values are reported to the processor and 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 the 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. The energy consumed by the GPU is ignored.

Initially, the MoA is chosen to mimic the hardware architecture. An MoA with 8 processing elements is selected (as shown at the bottom of Figure 10) and the processing elements are linked with three communication nodes: $A7CN$ providing communication internally to the Cortex-A7 cluster, $A15CN$ providing

communication internally to the Cortex-A15 cluster and ICC (for Inter-Cluster Communication) transporting messages between the two clusters. One may note that with the given configuration, any token flowing through CN ICC also flows through $A7CN$ and $A15CN$. All cores are cache coherent with 512kB of L2 cache in the A7 cluster and 2MB of L2 cache in the A15 cluster. Inter-cluster communication is transparent and managed by a hardware block called Cache Coherent Interconnect (CCI). LSLA does not model caches. Instead, it models the cost of transmitting a message (a token), by an affine model of the activity cost ($\alpha_n \times s + \beta_n$).

5.2 Experimental Setup

Software Tools

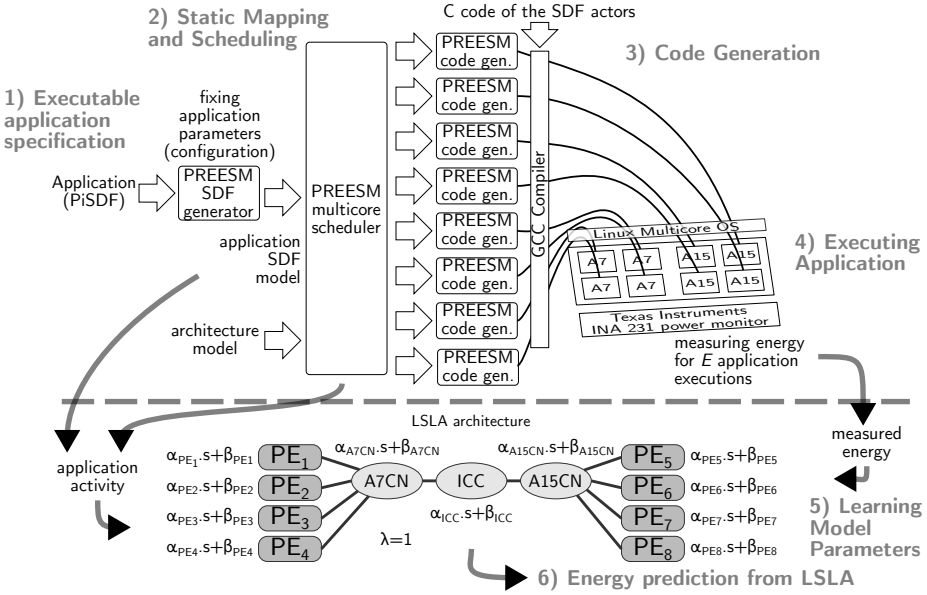


Figure 10: Experimental setup for inferring the LSLA execution energy model of a Samsung Exynos 5422 MPSoC.

Figure 10 summarizes the experimental setup used to build and assess the LSLA MoA of an Exynos 5422 processor from energy measurements. The PREESM dataflow framework [32] 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 [33] is an extension to SDF that introduces a hierarchy of composable elements, as well as static and dynamic parameters influencing token production and consumption. In the present work, dynamic parameters are not used. The motivation for using a PiSDF description is that, by fixing various values for the application parameters, different functional SDF applications are obtained. Once the parameters of the application are fixed, PREESM generates a (possibly large) executable SDF graph that feeds a multicore mapper and scheduler. Mapping and scheduling are static. PREESM then generates a self-timed multicore code

Table 2: Configurations of the stereo matching application employed to assess the energy modelling.

Configuration number	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×

for the application that can run 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 when running over an operating system. PREESM generates one thread per target core and forces the thread to the corresponding core via affinities.

Communication between actors occurs through shared memory and cache coherency between different threads. Semaphores are instantiated to synchronize memory accesses. The whole procedure of mapping, scheduling and generating code with PREESM can either be manually launched or 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 and feeds it to Matlab for learning the α_n and β_n LSLA parameters. 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.

Benchmarked Application

The benchmarked application is a stereo matching code described in [34] and illustrated in its SDF form in Figure 11. 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 corresponds to 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 Table 2. 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 [35].

Below each actor in the SDF graph illustration of Figure 11 is a repetition factor indicating the number of executions of this actor during an iteration of

the graph. This number of executions is deduced from the data production and 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:

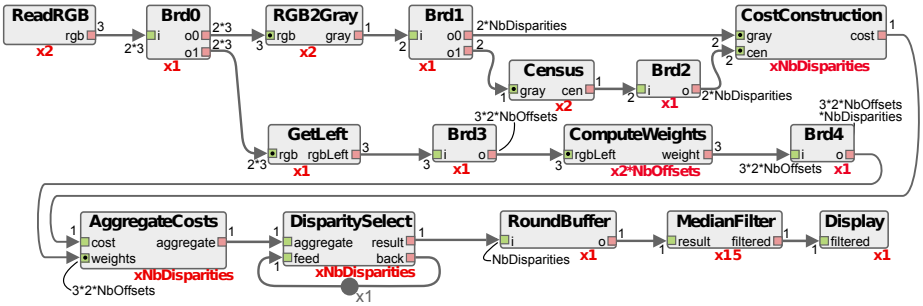


Figure 11: Illustration of the stereo matching application graph. The number of duplications of each actor is specified. All rates are implicitly multiplied by the picture size.

- *ReadRGB* produces the 3 color components of an input image by reading a stream or a file.
- *BrdX* is a broadcast actor. Its only purpose is to duplicate 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 of an image. This signature is obtained by comparing each pixel to its 8 neighbors: if the value of the neighbor is greater than the value of the pixel, the corresponding signature bit is set to 1; otherwise, it is set to 0.
- *CostConstruction* is executed once per potential disparity level. By combining the two images and their census signatures, it produces a value for each pixel that corresponds to 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. Computations are based on an iterative method that is executed *NbOffsets* times.

- *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.
- *Display* writes the resulting 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 mapping configurations are tested by enabling a different subset of the platform cores. PREESM schedules the application on the chosen subset with the objective of minimizing the application latency.

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 measurements. $\boldsymbol{d}$ in Equation 6 is a vector of energy measurements expressed in Joules. The energy of an application execution on the system is measured by integrating the instantaneous power consumed by the A7 cluster, the A15 cluster and the memory during application execution time. The power is sampled at 10 Hz.

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 the parameters of the application and the mapping constraints (choice of authorized cores), a population of executions is built, modeled and analyzed.

Application Activity

In order to quantify the activity of the application onto the architecture, this activity must be expressed in terms of tokens and quanta (Section 2.3).

In the self-timed code generated by PREESM, each PE runs a loop that processes a schedule of actors and the different PEs are synchronized by blocking communications of messages with different sizes. Using internal information from PREESM, the number of computational tokens on a given Processing Element (PE) is computed and equals the number of actor firings onto this PE. The number of communication tokens is the number of messages exchanged over a communication node.

In terms of computational quanta, time quanta in nanoseconds are used. A number of time quanta for an actor when running on a core of type $y \in$

Table 3: Time quanta (in ns) per actor type and core type for configuration 4.

Actor name	time on Cortex-A7	time on Cortex-A15	$\frac{t_{A7}}{t_{A15}}$
$ReadRGB$	$1,813,475$	$719,318$	$2.5\times$
$RGB2Gray$	$6,682,186$	$2,459,756$	$2.7\times$
$Census$	$6,846,640$	$2,320,574$	$3.0\times$
$ComputeWeights$	$85,265,027$	$32,251,999$	$2.6\times$
$CostConstruction$	$13,240,986$	$2,698,685$	$4.9\times$
$AggregateCosts$	$76,262,390$	$29,052,299$	$2.6\times$
$disparitySelect$	$6,192,992$	$1,128,882$	$5.5\times$
$Median_Filter$	$4,923,746$	$2,555,114$	$1.9\times$
$Display$	$131,638,086$	$100,411,424$	$1.3\times$

$\{Cortex\text{–}A7, Cortex\text{–}A15\}$ corresponds to the time needed to execute independently each actor on a core of type y. This time (in ns) is measured by repeating the actor and executing the C *clock()* function to retrieve timings. This operation is automated in the PREESM tool with the ARM code generator. As an example, the timings of actors for application configuration 4 are shown in Table 3. Communication quanta correspond to the amount of transfered data (in Bytes) over a given CN.

5.3 Experimental Results

Measuring Computational Dynamic Energy

Each of the six application configurations from Table 2 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 4.2 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.).

In order to integrate enough power samples to obtain a precise energy measure, the graph iteration of the application is repeated 10 times for each measurement, except for configuration 5 that executes much faster and necessitates 100 executions to provide reliable energy measurements. The measured energy is then divided by the number of graph iterations All the energy measurements are moreover repeated 10 times to obtain the standard deviation of the measurements. The ranges of measured values, displayed as vertical lines in Figure 12, give an indication of the limited measurement fluctuations. The average standard deviation of measurements is 0.21 Joules, representing a coefficient of variation of 2.4%. This relatively low variation is important because it shows that the energy consumption is stable for a given application activity and motivates for its modeling. For each configuration, the first 15 measurements on the left (in a dashed circle on Figure 12) show less dynamic 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 much more energy efficient than Cortex-A15 cores. The first 15 measurements (up to

pattern 00001111b) use only Cortex-A7 cores and, as a consequence, show more energy-efficiency. This fact will be reflected in the LSLA models constructed in next sections. One may note in the third column of Table 3 that the energy efficiency of Cortex-A7 cores comes at the price of a significantly lower speed.

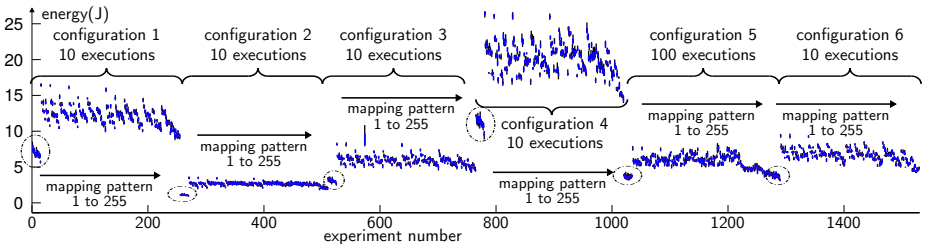


Figure 12: Range of the dynamic energy measurements based on 10 measurements and excluding outliers, i.e. values more than 1.5× the interquartile range away from the first and third quartile.

Learning the Energy Model with LSLA

Following the experimental setup depicted in Figure 10 and the learning method from Section 4, 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 validation error of the model as a function of the number of training points. The measured energy data is split into two parts: a training set containing between 1 sample and 80% of the samples (1224 samples), and a validation set with the remaining 20% of the samples (306 samples). The samples of the training set are randomly chosen. Figure 13 displays the training root-mean-square (RMS) error and the validation RMS error as the number of training samples rises. The training error is calculated over the training dataset while the validation error is calculated over the validation dataset.

The model fitting appears to be reasonable, as validation error lowers rapidly when the number of training samples grows and reaches a plateau at about 150 training samples corresponding to $1.45J$ of error on average. The training error rises continuously with the number of training elements, 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.

Discussion on the LSLA Model Parameters

The model is now trained over the whole $M = 1530$ energy measures. The data horizontal vector d of Equation 6 is of size 1530, the matrix G is of size 1530×22 and the model vector m is of size 22. The values of the obtained parameters are displayed in the model of Figure 14. The solid line in Figure 15 corresponds to the energy predicted with the model from Figure 14. The points correspond to energy measurements. The full model offers an energy assessment with an average root-mean-square error of 1.35 Joules, corresponding to an average error of 15.6%. The model performs better when the graph includes a large number

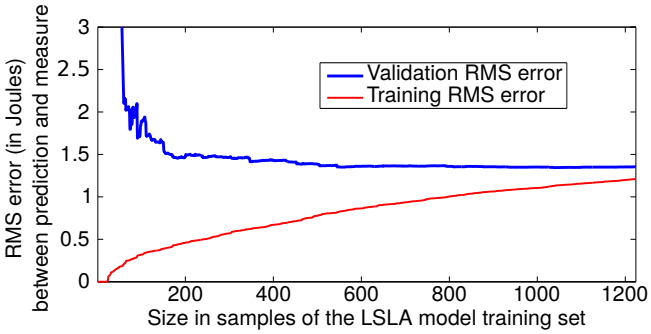


Figure 13: LSLA dynamic energy model learning curve.

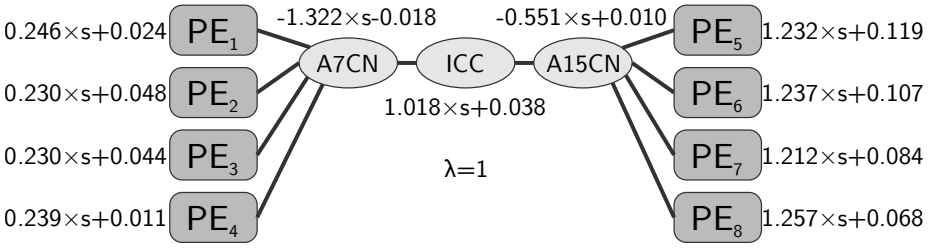


Figure 14: LSLA dynamic energy model inferred from energy measurements with computational quanta in ns, communication quanta in Bytes, and energy data vector d in nJ. PE_{1-4} are Cortex-A7 cores and PE_{5-8} are Cortex-A15 cores.

of actors and FIFOs, such as in configuration 4 where the average energy error is of 9.1%.

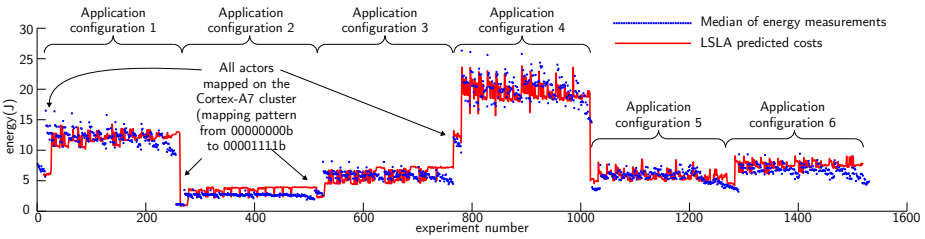


Figure 15: Comparing the LSLA predicted cost and the median of the corresponding energy measurements.

Easily explainable parameters in Figure 14 are α_{PE_1} to α_{PE_8} because they directly translate into average core execution dynamic power, in $nJ/ns = W$. Data from the table shows that PEs 1 to 4 have an average dynamic power of $236mW \pm 4\%$ and PEs 5 to 8 have an average dynamic power of $1.23W \pm 2\%$. 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 Figure 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 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 some assessment 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. This phenomenon motivates for research on a new, more precise MoA especially capable of capturing the effects of memory caches.

The LSLA model from Figure 14 does not represent the internal behavior of the mere architecture. Instead, it represents the architecture together with its operating system, the PREESM scheduler and the communication and synchronization library. For example, PREESM tends to favor Cortex-A15 cores because PREESM optimizes the schedule for latency and, because Cortex-A15 cores are much faster than Cortex-A7 cores, the demand placed on them is greater. A Cortex-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 a set of additions and no more than 22 multiplications. Moreover, neither application code nor architecture hardware of 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.

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.

Fidelity of the LSLA Energy Model

Model fidelity, as presented in [36], 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

$$f = \frac{2}{M(M-1)} \sum_{i=1}^{M-1} \sum_{j=i+1}^{M} f_{ij}, \tag{10}$$

where M is the number of measurements and

$$f_{ij} = \begin{cases} 1 \text{ if } sgn(d_i^{LSLA} - d_j^{LSLA}) = sgn(d_i - d_j) \\ 0 \text{ otherwise} \end{cases}, \tag{11}$$

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

and

$$sgn(x) = \begin{cases} (-1) \text{ if } (x < 0) \\ 0 \text{ if } x = 0 \\ 1 \text{ if } x > 0 \end{cases} . \tag{12}$$

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

The difference between the prediction and the measurements is provoked by a vast amount of factors, including cache non-deterministic behaviour, shared memory access arbitration, Linux scheduler decisions, background tasks, energy measurement sampling effect, etc. Considering this hardware and software complexity, the fidelity obtained by LSLA prediction is remarkable.

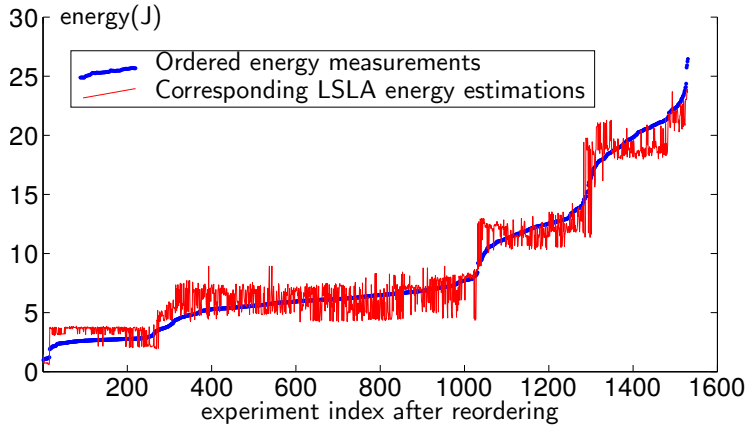


Figure 16: Energy Measurements Sorted in Ascending Order and their Corresponding LSLA Predictions.

6 Conclusion

In this report, 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. The example developed in this study is focused on dynamic energy modeling with LSLA for a given operating frequency. A vast amount of potential extensions exist, including static energy modeling, memory hierarchy modeling, and the study of large systems of systems.

Bibliography

[1] B. M. Maggs, L. R. Matheson, and R. E. Tarjan, "Models of parallel computation: A survey and synthesis," in *System Sciences, 1995. Proceedings of the Twenty-Eighth Hawaii International Conference on*. 1995, vol. 2, pp. 61–70, IEEE.

[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 *Formal Methods and Models for Co-Design, 2003. MEMOCODE'03. Proceedings. First ACM and IEEE International Conference on*. 2003, pp. 123–132, IEEE.

[3] A. Gerstlauer, C. Haubelt, A. D. Pimentel, T. P. Stefanov, D. D. Gajski, and J. Teich, "Electronic system-level synthesis methodologies," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, no. 10, pp. 1517–1530, 2009.

[4] *PolyCore Software, Inc. - http://polycoresoftware.com/ (accessed 12/2015).*

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

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

[7] M. Pelcat, K. Desnos, L. Maggiani, Y. Liu, J. Heulot, J.-F. Nezan, and S. S. Bhattacharyya, "Models of Architecture: Reproducible Efficiency Evaluation for Signal Processing Systems," in *Proceedings of the 2016 IEEE International Workshop on Signal Processing Systems*, Dallas, United States, 2016.

[8] F. Baccelli, G. Cohen, G. Jan Olsder, and J.-P. Quadrat, *Synchronization and linearity: an algebra for discrete event systems*, John Wiley & Sons Ltd, 1992.

[9] N. Bambha, V. Kianzad, M. Khandelia, and S. S. Bhattacharyya, "Intermediate representations for design automation of multiprocessor DSP systems," *Design Automation for Embedded Systems*, vol. 7, no. 4, pp. 307–323, 2002.

[10] B. Kienhuis, E. Deprettere, K. Vissers, and P. van der Wolf, "An approach for quantitative analysis of application-specific dataflow architectures," in *Application-Specific Systems, Architectures and Processors, 1997. Proceedings., IEEE International Conference on*. 1997, pp. 338–349, IEEE.

[11] J. Eker, J. W Janneck, E. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, Y. Xiong, et al., "Taming heterogeneity-the ptolemy approach," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 127–144, 2003.

[12] M. Pelcat, S. Aridhi, J. Piat, and J.-F. Nezan, *Physical Layer Multi-Core Prototyping: A Dataflow-Based Approach for LTE eNodeB*, vol. 171, Springer, 2012.

[13] H. Yviquel, *From dataflow-based video coding tools to dedicated embedded multi-core platforms*, Ph.D. thesis, 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, M. Kiemb, and S. S. Bhattacharyya, "Heterogeneous design in functional DIF," in *Proceedings of the International Workshop on Systems, Architectures, Modeling, and Simulation*, Samos, Greece, July 2008, pp. 157–166.

[17] S. S. Bhattacharyya, E. Deprettere, R. Leupers, and J. Takala, Eds., *Handbook of Signal Processing Systems*, Springer, second edition, 2013.

[18] W. Plishker, N. Sane, M. Kiemb, K. Anand, and S. S. Bhattacharyya, "Functional DIF for rapid prototyping," in *Proceedings of the International Symposium on Rapid System Prototyping*, Monterey, California, June 2008, pp. 17–23.

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

[20] G. E. Box and N. R. Draper, *Empirical model-building and response surfaces*, vol. 424, Wiley New York, 1987.

[21] B. Kienhuis, E. F. Deprettere, P. Van Der Wolf, and K. Vissers, "A methodology to design programmable embedded systems," in *Embedded processor design challenges*. Springer, 2002, pp. 18–37.

[22] J. Ceng, W. Sheng, J. Castrillon, A. Stulova, R. Leupers, G. Ascheid, and H. Meyr, "A high-level virtual platform for early MPSoC software development," in *Proceedings of the 7th IEEE/ACM international conference on Hardware/software codesign and system synthesis*. 2009, pp. 11–20, ACM.

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

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

[25] J. Castrillon Mazo and R. Leupers, *Programming Heterogeneous MPSoCs*, Springer International Publishing, Cham, 2014.

[26] V. Kianzad and S. S. Bhattacharyya, "CHARMED: A multi-objective co-synthesis framework for multi-mode embedded systems," in *Application-Specific Systems, Architectures and Processors, 2004. Proceedings. 15th IEEE International Conference on*. 2004, pp. 28–40, IEEE.

[27] T. Grandpierre and Y. Sorel, "Un nouveau modèle générique d'architecture hétérogène pour la méthodologie AAA," *Actes JFAAA'02*, 2002.

[28] E. Raffin, C. Wolinski, F. Charot, K. Kuchcinski, S. Guyetant, S. Chevobbe, and E. Casseau, "Scheduling, binding and routing system for a run-time reconfigurable operator based multimedia architecture," in *Design and Architectures for Signal and Image Processing (DASIP), 2010 Conference on*. 2010, pp. 168–175, IEEE.

[29] M. Pelcat, J.-F. Nezan, J. Piat, Jerome Croizer, and S. Aridhi, "A system-level architecture model for rapid prototyping of heterogeneous multicore embedded systems," in *Proceedings of DASIP conference*, 2009.

[30] A. Donlin, "Transaction level modeling: flows and use models," in *Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*. 2004, pp. 75–80, ACM.

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

[32] M. Pelcat, K. Desnos, J. Heulot, C. Guy, J.-F. Nezan, and S. Aridhi, "PREESM: A dataflow-based rapid prototyping framework for simplifying multicore dsp programming," in *Education and Research Conference (EDERC), 2014 6th European Embedded Design in*, Sept 2014, pp. 36–40.

[33] K. Desnos, M. Pelcat, J.-F. Nezan, S. S. Bhattacharyya, and S. Aridhi, "PiMM: Parameterized and interfaced dataflow meta-model for MPSoCs runtime reconfiguration," in *SAMOS XIII*, 2013.

[34] A. Mercat, J.-F. Nezan, D. Menard, and J. Zhang, "Implementation of a stereo matching algorithm onto a manycore embedded system," in *2014 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2014, pp. 1296–1299.

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

[36] 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, pp. 91–97.