

For an Efficient Execution of Data Intensive SoCs (Position Paper)

Abdoulaye Gamatié

LIFL - CNRS UMR 8022 / INRIA Lille Nord Europe,
Parc de la Haute Borne, Bât A, 40 avenue Halley, 59650 Villeneuve d'Ascq Cedex, France

1 Design of data-intensive embedded systems

Modern embedded systems tend to be more and more sophisticated with the integration of multiple functionalities in the same system, often implemented on a single chip, termed *system-on-chip* (SoC). Today, they have become very attractive in the consumer electronics domain (cellular phones, television, camera, GPS automotive navigation system, etc.), which shows a rapid economic growth. The counter-part of this success is the crucial need to deal with their increasing design complexity and their other important requirements such as safety, quality of service (QoS) and performance. All these aspects play a critical role in the product competitiveness. In particular, since a large part of these products is data (or computation) intensive, they require powerful architectures in order to be supplied with the necessary execution performances. With the huge number of transistors in chips today, the implementation of parallel architectures on a chip is possible, making *multiprocessor system-on-chip* (MPSoCs) mainstream for embedded systems with intensive parallel computations. MPSoCs offer high computational performances, while reducing power consumption. They consist of several processing and memory elements, interconnected by an on-chip dedicated structure (e.g. Tile64 of Tileria - www.tileria.com).

MPSoC-based design of embedded systems requires adequate methodologies in order to reduce the complexity of design space exploration and to increase the productivity of developers. A solution consists in considering high-level models and automatic transformations that refine such models into lower level ones for various purposes. These models must be rich enough to enable the expression of concurrency (for an efficient exploitation of the parallelism and scheduling) and the evaluation of performances (for design space exploration). They must also have a clear formal semantics, usable for a trustworthy system validation.

2 A model-driven design approach for MPSoCs

We propose a model-driven design approach according to which the different parts of data intensive SoCs (software and hardware) can be designed, simulated at various abstraction levels, analyzed and synthesized on hardware [3].

This approach is particularly suitable for application domains such as multimedia processing and detection systems (sonar, radar) in which computations are most often composed of a regular application of filters to multidimensional data structures. For instance, in multi-resolution televisions, image size reduction consists of a uniform application of a downscaling filter to 2-dimensional arrays representing images [2]. Due to the high parallelism degree of these applications, dealing with concurrency is not easy. We consider a separation of concerns allowing for the modeling of: *i*) the functional behavior of data intensive SoCs via the specification of pure data dependencies; the resulting dependency graph features all possible execution scenarios; *ii*) the refinement of this graph with environment and platform constraints for execution.

Data dependency specification. Given a system, the model of its functional behavior consists of the description of data dependencies between multidimensional arrays [2]. Such dependencies are expressed in a modular way according to the following grammar, where the notations $x : X$ and $\{X\}$ respectively mean X is the type of x and a set of elements of type X :

<i>Component</i>	::= <i>Interface</i> ; <i>Body</i>	(r1)
<i>Interface</i>	::= $i, o : \{Port\}$	(r2)
<i>Port</i>	::= $id; type; shape$	(r3)
<i>Body</i>	::= $Body^h \mid Body^r \mid Body^e$	(r4)
<i>Body^e</i>	::= <i>some function</i>	(r5)
<i>Body^r</i>	::= $t_i, t_o : \{Tiler\}; \{s_r; Component\}; \{Ird\}$	(r6)
<i>Ird</i>	::= $Connexion; d; c_p$	(r7)
<i>Connexion</i>	::= $p_i, p_o : Port$	(r8)
<i>Tiler</i>	::= $Connexion; (F; o; P)$	(r9)
<i>Body^h</i>	::= $\{Component\}; \{Connexion\}$	(r10)

All components have the same global structure, as described in rule (r1): an *interface* (rule (r2)) specifying the input and output ports (rule (r3)), respectively represented by i and o ; and a *body* (rule (r4)), defining the data dependency between the input and output ports of the interface.

An *elementary component* E , characterized by rule (r5), consists of a simple function. A *repetitive component* R , characterized by rule (r6), expresses a data-parallelism where *Component* denotes the function to be replicated (or repeated) on different data subsets from arrays. The resulting *instances* of *Component* are assumed to be independent and schedulable following any order. Thus, the execution concurrency of R is minimally constrained by the specified data dependencies. In rule (r6), the attribute s_r denotes the *repetition space*, which gives the number of *Component* instances. Each instance consumes and produces sub-arrays with the same shape, called *patterns* or *tiles*. Tiles are constructed via *tilers* (rule (r9)). A tiler extracts (resp. stores) patterns from (resp. in) an array based on the following information: F as *fitting* matrix, describing how array elements fill patterns; o as *origin* of the *reference pattern*; and P as *paving* matrix, specifying how patterns cover an array. Sometimes, the computation of component instances may depend on other instances in the same repetition. This typically happens when computing a discrete integration where successively, each instance adds an input value to the previous result

computed by another instance, until the final result. This is referred to as *inter-repetition dependency*. Finally, a *hierarchical component*, characterized by rule (r10), is defined as an acyclic dependency graph of component. It expresses task parallelism. All these concepts have been adopted in the UML standard profile MARTE, dedicated to the Modeling and Analysis of Real-time and Embedded systems [3]. The data dependency specification fully captures the functional behavior of a system. It is constrained by neither the system environment nor the execution platforms. So, it can be seen as an abstraction of all possible execution scenarios of a system. Typical properties such as single assignment of absence or causality cycles can be checked on this model without worrying about the behavior combinatorics induced by its execution concurrency.

Platform and environment property specifications. In order to enable design space exploration, we need to refine our abstract model with additional information under the form of environment and execution platform constraints.

Automata have been used to define execution modes that express how different components specifying data dependencies interact [4]. The semantics of these automata is that of synchronous mode automata. For the sake of simplicity, the control part is assumed to be separated from the data part. In such a way, we avoid the potential scalability issues arising when using model-checkers since the amount of the manipulated data is very large. An alternative choice consists in using the *abstract clocks* [5, 1] of synchronous languages, instead of automata. Indeed, by associating components with activation clocks and by specifying relations between such clocks to indicate when the corresponding data dependencies are valid and combine, several environment and execution platform constraints can be described. An abstract clock expresses event occurrences at some logical instants. The resulting set of instants is totally ordered and defines the rate of a clock. A monoprocessor execution of a set of components can be modeled by considering a single “master” clock for the whole set. Then, the clock of each component is simply a subsampling of this master clock. A multiprocessor execution leads to multiple clocks that are related only when the associated components interact. Therefore, the concurrency can be clearly described and analyzed with suitable tools such as the compilers of synchronous languages in our case. In particular, we address synchronizability issues between the clocks of communicating components so as to ensure some QoS properties.

3 Position: clocks efficiently deal with concurrency

From the viewpoint of our previous proposition to deal with environment and platform constraints on pure data dependency specifications, we believe that abstract clocks (borrowed from synchronous languages) are more powerful than automata. Indeed, clocks offer a best compromise in terms of model size and preciseness in the expression of concurrency in data intensive systems.

Nevertheless, the considered abstract clocks are not completely satisfactory to enable the modeling of all non functional constraints. As a matter of fact, such clocks deal with computation rates, which are mainly related to the temporal dimension. In our target applications where data are manipulated under the form of multidimensional arrays, one also needs to capture information related to the spatial dimension. The spatial organization of data has sometimes an important impact on the concurrency during executions. For instance, let us consider a component c_1 that produces some image line by line, consumed by another component c_2 column by column. Then, one can observe that c_2 cannot start executing before c_1 has produced as many lines as elements a column contains. This amount of data determines the minimal degree of pipelined execution of c_1 and c_2 . This simple example shows that in order to efficiently address the concurrency and to improve the performances in data intensive embedded systems that manipulate multidimensional data structures, one needs to reason on both time and space dimensions. As a preliminary tentative, we believe that the abstract clocks defined in synchronous languages can serve as a basis to define the suitable multidimensional clocks enhanced with data structure shape information. Then, further clock definitions such as vector clocks [6] could be also considered. The existing clock and shape analysis techniques are interesting ingredients for the definition of an adequate verification approach to ensure an efficient and correct concurrency. More generally, we strongly believe that abstract interpretation techniques combined with static analysis would greatly help to find the best representation on which one can adequately reason and infer optimized execution models.

References

1. Adolf Abdallah and Abdoulaye Gamatié and Jean-Luc Dekeyser. MARTE-based Design of a Multimedia Application and Formal Analysis. In *Forum on specification and design languages (FDL'08)*, Stuttgart, Germany, September 2008.
2. Pierre Boulet. Formal Semantics of Array-OL, a Domain Specific Language for Intensive Multidimensional Signal Processing. Research report, INRIA, France, March 2008. available online at <http://hal.inria.fr/inria-00261178/fr>.
3. Abdoulaye Gamatié, Sébastien Le Beux, Éric Piel, Anne Etien, Rabie Ben Atitallah, and Philippe Marquet Jean-Luc Dekeyser. A Model Driven Design Framework for High Performance Embedded Systems. Research report 6614, INRIA - France, August 2008.
4. Abdoulaye Gamatié, Éric Rutten, and Huafeng Yu. A model for the mixed-design of data-intensive and control-oriented embedded systems. Research report 6589, INRIA, France, July 2008.
5. Abdoulaye Gamatié, Éric Rutten, Huafeng Yu, Pierre Boulet, and Jean-Luc Dekeyser. Synchronous Modeling and Analysis of Data Intensive Applications. *Eurasip Journal of Embedded Systems*, 2008, 2008. Article ID 561863, 22 pages.
6. Friedemann Mattern. Virtual time and global states of distributed systems. In M. Cosnard, editor, *Proc. Workshop on Parallel and Distributed Algorithms*, pages 215–226, North-Holland / Elsevier, 1989.