HAL
open science

Rigorous System Level Modeling and Analysis of Mixed HW/SW Systems

Paraskevas Bourgos, Ananda Basu, Saddek Bensalem, Marius Bozga, Joseph Sifakis, Kai Huang

To cite this version:

Paraskevas Bourgos, Ananda Basu, Saddek Bensalem, Marius Bozga, Joseph Sifakis, et al.. Rigorous System Level Modeling and Analysis of Mixed HW/SW Systems. 9th IEEE/ACM International Conference on Formal Methods and Models for Codesign, MEMOCODE 2011, Jul 2011, Cambridge, United Kingdom. pp.11-20, 10.1109/MEMCOD.2011.5970506 . hal-00722402

HAL Id: hal-00722402

https://hal.science/hal-00722402v1

Submitted on 1 Aug 2012

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Rigorous System Level Modeling and Analysis of Mixed HW/SW Systems

P. Bourgos, A. Basu, M. Bozga, S. Bensalem, J. Sifakis
UJF-Grenoble 1 / CNRS, VERIMAG UMR 5104
Grenoble, F-38041, France
{bourgos, basu, bozga, bensalem, sifakis}@imag.fr

K. Huang
Institute of VLSI Design
Zhejiang University, China
huangk@vlsi.zju.edu.cn

***Abstract*—A grand challenge in complex embedded systems design is developing methods and tools for modeling and analyzing the behavior of an application software running on multicore or distributed platforms. We propose a rigorous method and a tool chain that allows to obtain a faithful model representing the behavior of a mixed hardware/software system from a model of its application software and a model of its underlying hardware architecture. The system model can be simulated and analyzed for validation of both functional and extra-functional properties. The tool chain uses DOL (Distributed Operation Layer [1]) as the frontend for specifying the application software and hardware architecture, and BIP (Behavior Interaction Priority [2]) as the modeling and analysis framework. It is illustrated through the construction of system models of MJPEG and MPEG2 decoder applications running on MPARM, a multicore architecture.**

I. Introduction

Performance of embedded applications strongly depends on features of the underlying hardware platform. In contrast to performance of application software running on a single core, getting the maximum throughput out of multicore processors demands application software to be designed taking parallelism into account from scratch. This is needed to catch up with the fast growth of computing capacity due to the foreseeable exponential increase of physical parallelism. But programming, testing and verifying parallel software with currently existing tools is notoriously hard, even for experts. There are no rigorous techniques for deriving global model of a given system from models of its application software and its execution platform.

Application software must be programmed for performance, in a platform independent way, exhibiting all potential parallelism. Its implementation must deal with mapping the specified application-level parallelism onto platform-level (threads, cores, processors) on an as-needed/as-available basis. Actually, this mapping would need to be adapted dynamically as applications must scale up or down according to the available resources of the execution platform. Moreover, efficiency and correctness are not the only concerns. Programmer productivity, that is, the programmer's ability to design correct software that gathers the maximum performance out of an arbitrary multicore platform with ease should not be neglected [3].

Achieving these goals requires a design flow based on a single semantic model. The design flow must be able to generate rigorous models of mixed hardware/software systems, suitable for analysis, design space exploration and automatic code generation. The main contribution of this paper is deriving a rigorous system model combining the application software and the architecture, which can be the basis for multiple objectives, such as functional verification, performance evaluation and code generation for target architectures.

We propose a system construction method that is both rigorous and allows a fine analysis of system dynamics. It is rigorous because it is based on formal models, have precise semantics and thus can be analyzed by using formal techniques. A system model is derived by progressively integrating constraints induced on an application software model by the underlying hardware architecture model. Both models are described in BIP [2], which is a formal component based modeling framework. In contrast to ad hoc modeling approaches, the system model is obtained from a BIP model of the application software and a description of the hardware architecture, by application of source-to-source transformations that are correct-by-construction [4]. The final generated model is a mixed software-hardware model which provides the capability using a single model to simulate and apply formal verification techniques on it using the BIP framework.

Metro II [5] is a platform-based design framework and provides a simulation backend based on SystemC. Octopus [6] allows design space exploration by stochastic simulation of task graphs. Both have connections to formal verification tools based on model checking. Most of the frameworks for mixed HW/SW systems are based on SystemC [7] as a language for modeling at various levels of abstractions. Various tools and associated design methodologies emerged

The research leading to these results has received funding from the European Community's Seventh Framework Programme [FP7/2007-2013] under grant agreement no 248776 (PRO3D) and from ARTEMIS JU grant agreement ARTEMIS-2009-1-100230 (SMECY)

e.g., SystemCoDesigner [8], Spade [9], Sesame [10] to cite only a few. All these focus and facilitate the construction of executable simulation models which, while being claimed cycle-accurate, do not rely on a formal foundation. For instance, such models cannot be used to check formally the correctness of the constructed system. There have been attempts on providing formal semantics to System-C models using tools like LusSy [11], however, they remain difficult to use mainly because of the limited expressiveness of the target formalism compared with a general purpose language.

One of the main needs for rigorous system model is performance evaluation. Simulation based methods use ad-hoc executable system models such as [12] or models in SystemC [7], [13]. The latter provide cycle-accurate results, but are not adequate for thorough exploration of hardware architecture dynamics and its effects on software execution. Furthermore, long simulation time is a major drawback. Trace-based co-simulation is used in Spade [9], Sesame [10]. There exist much faster techniques that work on abstract system models e.g., Real Time Calculus [14] and SymTA/S [15]. They use formal analytical models representing a system as a network of nodes exchanging streams. The dynamics of the execution platform is characterized by execution times. Nonetheless, these techniques allow only estimation of pessimistic worst-case measures (delays, buffer sizes, etc) and moreover, they require an abstract model of the application software. Building these abstract models represents a significant modeling effort and, if done through a manual process, the results are not guaranteed to be accurate. Similar drawbacks exists for performance analysis techniques based on Timed-Automata [16], [17]. These can be used for modeling and solving scheduling problems. An approach combining simulation and analytic models is presented in [18], where simulation results can be propagated to analytic models and vice versa through well defined interfaces.

The paper is structured as follows. Section II presents the method and the main steps in the design flow, with a brief overview of the BIP framework and associated toolbox. The generation of the system model follows in section III. Section IV describes the performance estimation technique applied on the system model. Finally, experimental results are provided in section V. In section VI we conclude and discuss future work directions.

II. Design Flow

The flow of our method is illustrated in Figure 1. The method takes three inputs: (i) the application software, (ii) the hardware architecture and (iii) the mapping. We consider application software defined using the Kahn process network model [19]. They consists of a set of deterministic processes communicating through FIFO channels by executing atomic read/write operations. The behavior of each process is a sequential program. We consider hardware architectures described as interconnections of computational and communication devices such as processors, buses and memories. Finally, we consider mappings that associate application software elements to hardware architecture, that is, processes to processors and FIFO channels to memories.

In this paper, we will focus on the generation of the system model. We will also describe one of its utilities, i.e., performance evaluation. The first stage of the method is the construction of the *system model* in BIP. The system model represents the application mapped on the hardware architecture. The system model is obtained by the three following steps:

1) the construction of a BIP model by automatic translation from the application software,
2) the construction of a BIP model by automatic translation from the hardware architecture,
3) the construction of the system model by source-to-source transformation of the previous two models and their composition according to the mapping.

The second stage of the method is performance evaluation realized on the system model. We provide a simulation-based technique allowing the accurate estimation of real-time characteristics (response times, delays, latencies, throughputs, etc.) and particular indicators about the use of resources (bus conflicts, memory conflicts, etc.).

The performance evaluation method combines native (BIP) simulation of the system model with online code profiling on the target hardware architecture. That is, the (simulated) processing time required by the application code is computed during simulation, on demand, using the application object code for the target architecture and the processor weight table. The later provides the raw execution times for elementary (assembler) instructions.

The method is completely automated and has been implemented in a tool. The tool uses as inputs Distributed Operation Layer (DOL) [1] specifications, that is, the application software, the hardware architecture and the mapping are described using the concrete formalisms available in the DOL framework. The method is realized using the BIP framework [2], [20], [21] and the associated toolbox[1]. The BIP language is a notation which allows complex systems to be built by coordinating the behavior of a set of atomic components. The behavior is described as automata or Petri nets extended with data and functions described in C/C++. Transitions are labelled with ports (action names), guards (enabling conditions on the state of a component) as well as functions (computations on local data). The description of coordination between components is layered. It consists of interactions and priorities that characterizes the overall architecture of a component. Their combination confers BIP strong expressiveness that cannot be matched by other languages [20]. BIP has clean operational semantics that

[1] http://www-verimag.imag.fr/Download.html

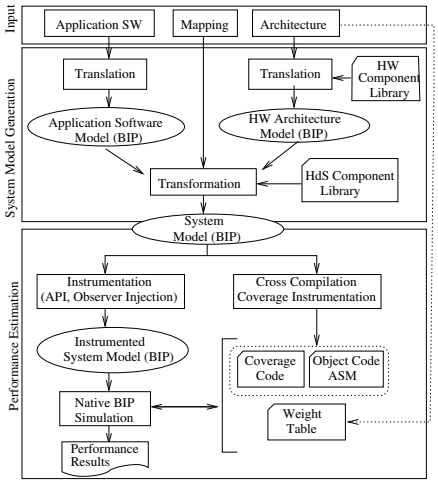

Figure 1. System Model Construction and Performance Evaluation

describe the behavior of a composite component as the composition of the behaviors of its atomic components. This allows a direct relation between the underlying semantic model (transition systems) and its implementation.

III. Deriving System Model

The construction of the system model in BIP from the input DOL specification [1] is done in three steps, as described in the following subsections.

A. Construction of Application Software Model in BIP

An application software in DOL [1] is a process network that consists of three basic entities: *SW-Process*, *SW-Channel*, and *SW-Connection*, organized as described by the following abstract grammar:

$$
\begin{array}{rcl}
\textit{Appl-Software} & ::= & \textit{SW-Process}^{+} \,.\, \textit{SW-Channel}^{+} \,.\, \textit{SW-Conn}^{+} \\
\textit{SW-Process} & ::= & \textit{SW-InPort}^{*} \,.\, \textit{SW-OutPort}^{*} \,.\, \textit{SW-Behavior} \\
\textit{SW-Channel} & ::= & \textit{SW-RecvPort} \,.\, \textit{SW-SendPort} \,.\, \textit{SW-Channel-Behav} \\
\textit{SW-Conn} & ::= & \textit{SW-Read-Conn} \mid \textit{SW-Write-Conn} \\
\textit{SW-Write-Conn} & ::= & \textit{SW-OutPort} \,.\, \textit{SW-RecvPort} \\
\textit{SW-Read-Conn} & ::= & \textit{SW-SendPort} \,.\, \textit{SW-InPort} \\
\textit{SW-Behavior} & ::= & \textit{a-C-program} \\
\textit{SW-Channel-Behav} & ::= & \textit{FIFO-Param}^{+}
\end{array}
$$

Each software process P has input ports $P.InPort_i$, output ports $P.OutPort_j$ and behavior *P.Behavior*. Each channel C has a single input port *C.RecvPort* and a single output port *C.SendPort*. A write connection between output port j of a process P and a channel C is a pair ($P.OutPort_j$, *C.RecvPort*). A read connection between input port i of process P and a channel C is a pair (*C.SendPort*, $P.InPort_i$). We assume that ports of channels are uniquely associated with ports of processes and vice versa.

Process behavior is described using C programs with a particular structure (see figure 3 for a concrete example). In general, the behavior of a process P is defined by an initial call of the *P_init()* function followed by an endless loop calling the *P_fire()* function. Communication is realized by using two particular primitives, namely *write* and *read* for respectively sending and receiving data to software channels. A *read* operation reads data from an input port, and a *write* operation writes data to an output port. The code may also call another special primitive, namely *detach*, in order to terminate the execution of the process.

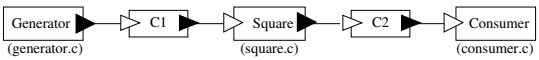

Figure 2. An application software

Example 1: An example process network is shown in figure 2. It has three SW-processes (generator, square and consumer), connected through two SW-channels (C1 and C2). The generator produces an integer and sends it to square, which squares it and send it to the consumer which prints the result. The description of square process is shown in figure 3. It defines the data structure for the process state, the function square_init() to initialize the process state and the function square_fire() to define the cyclic behavior of the process. The square process uses integer variables index and len. The function square_fire defines a floating variable i, which holds the value read from the port IN. On every call of square_fire, it reads a value for i, squares it, writes it to the port OUT and increments the counter index. The process terminates when index reaches len.

```
#define IN  1
#define OUT 2
typedef struct _local_states {
    int index;
    int len;
} Square_State;
void square_init(Process *p) {
    p->local->index = 0;
    p->local->len = LENGTH;
}
int square_fire(Process *p) {
    float i;
    if (p->local->index < p->local->len) {
        read((void*)IN, &i, sizeof(float), p);
        i = i*i;
        write((void*)OUT, &i, sizeof(float), p);
     p->local->index++;
    }
    else {
        detach(p);
        return -1;
    }
    return 0;
}
```

Figure 3. C code fragment of the *square* process

The construction of the application software model in BIP is structural: every process and every channel are independently translated to atomic components in BIP and then connected according to their connections in the process network.

1) Translation of Software Processes into BIP:

The translation converts every software process to an atomic component in BIP. Each port is defined as a port in the atomic component. Data structures defined in the C functions are used as data in the atomic component. Control locations correspond to invocation of *read/write* primitives for which synchronization is required. Transitions are labeled by the port name associated with the primitives. Computation statements are added as actions of the transitions.

The translation requires the extraction of a control-flow graph from the C code. It starts by parsing the process code into an intermediate, annotated abstract syntax tree (AST). The translation to BIP is then completed in two steps. In the first step, the interaction points in the AST are identified, that is, each call to a *read/write* primitive is registered as an interaction point. The second step involves the construction of an explicit control flow graph and its representation as a finite state automaton extended with data in BIP. For every interaction point, a control location is created. An outgoing transition is added from this location, labeled by the port used in the *read/write* call. The transition models the primitive call and requires synchronization with a software channel.

Statements other than *read/write* calls are added as actions to the existing transitions. Let us notice that any functions that contain *read/write* calls (either directly or through nested calls) are *inlined* in the BIP automaton. Consequently, our translation is restricted to programs without communication calls occurring within recursive functions. Additional restrictions are, namely: no use of global variable; and no *goto* statement.

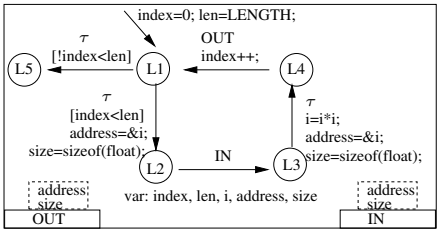

Figure 4. The model of the *square* process as an atomic BIP component

Example 2: Figure 4 shows the translation of the square process into an atomic component in BIP. The generated BIP component has ports IN, OUT, control locations L1, . . . L5 and variables index, len and i. Additional variables size and address are associated as parameters of the ports. Transitions are labeled by IN, OUT and τ, denoting an internal transition. At L2, it awaits synchronization through IN corresponding to the read primitive call. At L4 it awaits synchronization through OUT corresponding the write primitive call. At L1, internal transitions with guard model the conditional (if) statement. Exit of the process on a detach is modeled by the deadlocked location L5.

2) Translation of Software Channels into BIP:

Every software channel is translated into a predefined BIP atomic component, as shown in figure 5. It has ports *recvPort* and *sendPort*, and a single control location *L1*. It contains an array of data *buff* parametrized by size N. The variable x associated with *recvPort* gets the received value which is inserted into *buff*. The variable y associated with *sendPort* contains the value to be read next. The FIFO policy is implemented by using two indices i and j, for respectively insertion/deletion into/from the (circular) buffer *buff*.

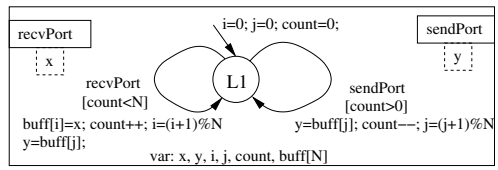

Figure 5. SW-channel (FIFO) in BIP

3) Translation of Connections into BIP:

Every connection in the application software is translated into a BIP connector which strongly synchronizes the corresponding ports. Connectors provide the transfer of data implementing the *read* and *write* operations. A connector implementing *write* transfers data from a process to a channel, whereas the one implementing *read* transfers data from a channel to a process.

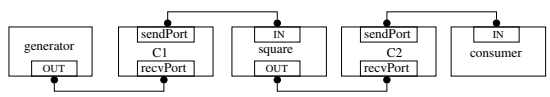

Figure 6. Application software model in BIP

Example 3: The figure 6 provides the complete BIP model obtained from the application example given in figure 2. It consists of the BIP component generator sending data to square and consumer by using channels C1 and C2 respectively.

B. Construction of Hardware Architecture Model in BIP

A hardware architecture consists of computational resources interconnected according to communication paths. Resources are used for computation (processors, memories) or for communication (buses). Communication paths define the connections between computational resources. More formally, we consider the family of hardware architectures that can be represented in DOL [1] and are abstracted by the following grammar:

$$
\begin{array}{rcl}
\textit{HW-Arch} & ::= & \textit{HW-Resource}^{+} \ . \ \textit{HW-Comm-Path}^{+} \\
\textit{HW-Resource} & ::= & \textit{HW-Processor} \mid \textit{HW-Memory} \mid \textit{HW-Bus} \\
\textit{HW-Comm-Path} & ::= & \textit{HW-Read-Path} \ . \ \textit{HW-Write-Path} \\
\textit{HW-Read-Path} & ::= & \textit{HW-Memory} \ . \ \textit{HW-Bus}^{+} \ . \ \textit{HW-Processor} \\
\textit{HW-Write-Path} & ::= & \textit{HW-Processor} \ . \ \textit{HW-Bus}^{+} \ . \ \textit{HW-Memory}
\end{array}
$$

Example 4: An example of a multi-core hardware architecture is shown in figure 7. It contains two identical tiles

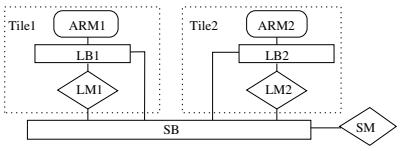

Figure 7. A multi-core hardware architecture with two ARM tiles

and a shared memory (SM) connected via a shared bus (SB). Each tile $i = 1, 2$, contains an ARM processor (ARMi) connected to the local memory (LMi) via a local bus (LBi). The local memory of each tile is also connected to the shared bus. We consider the following three communication paths, ordered (write, read) as follows:

WP1 =	*ARM1.LB1.LM1*	*RP1 =*	*LM1.LB1.ARM1*
WP2 =	*ARM1.LB1.SB.SM*	*RP2 =*	*SM.SB.LB2.ARM2*
WP3 =	*ARM2.LB2.LM2*	*RP3 =*	*LM2.LB2.ARM2*

The BIP model constructed from the hardware architecture represents explicitly, in an operational manner, the interconnect between the different resources as defined by the communication paths. This model is organized as a collection of bus, processor and memory components. Nonetheless, let us notice that, the processor and memory components are just empty, placeholder components. We introduce them in the BIP model of the hardware architecture only for the sake of clarity. They will be filled during the next step, that is, the construction of the system model.

Every bus component is concretely defined as a scheduled collection of communication path fragments. That is, for every read/write path going on a bus, we consider the path fragment defined by three atomic components, respectively:

- the *MasterInterface* (MI) component, which controls the access of the communication path on the bus and initiates the read/write operation. Depending on its position on the path, the master component receives data either from some software processes executing inside the processor or from the previous path segment.
- the *VirtualLink* (VL) component, which models effectively the transfer of data over the bus, from the master once it gets access to the bus, towards the slave.
- the *SlaveInterface* (SI) component, which acts like a buffer and is needed to connect further either to the next path fragment or to some FIFO buffers on the memory, depending on the position of the bus on the path.

All the paths segments going over the same bus must share its transport capabilities according to some predefined bus policy. The scheduling can be of one of fixed-priority, round-robin or TDMA. We model it explicitly by using a *HW-Bus-Scheduler* component, which interacts with all the master interface components and ensures exclusive access for transmission of data, according to the policy selected. The *HW-Bus-Scheduler* acts as an arbiter to resolve the bus access conflicts.

All these components are predefined and belong to the BIP hardware library. They have identical interfaces for the transport of data, respectively ports *RR/WR* (*Read/Write-Request*), *RA/WA* (*Read/Write-Acknowledge*) to connect with upper components, and *RB/WB* (*Read/Write-Begin*), *RE/WE* (*Read/Write-End*) to connect with lower components on the path. In addition, the *MI* components use ports *ACQ* (*Acquire*) and *REL* (*Release*) to interact with the bus scheduler.

Finally, let us also notice that all these components are *timed* BIP components [2]. The *VirtualLink* components model the latency of the buffer. The *Master/SlaveInterface* components observe the time progress and can be used for observation purposes, as explained later in section IV.

Example 5: The BIP model of the local bus LB1 of example 4 is shown in figure 8. It implements the two write paths WP1, WP2 and the read path RP1.

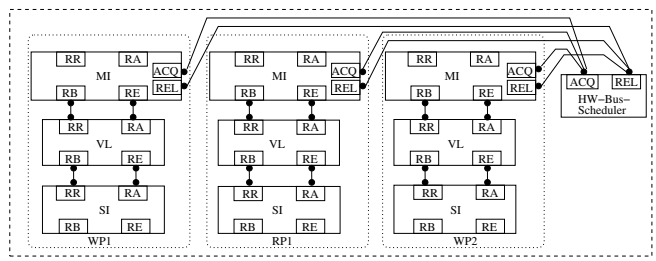

Figure 8. The BIP Model of the *LB1* bus

Every connection is realized using BIP connectors which strongly synchronize the corresponding ports. The behavior of the connector implements the transfer of data, its address and size between the successive components, corresponding to the *write* and *read* operations.

Example 6: Figure 13 shows the BIP hardware model of the 2-Tile ARM architecture of example 4. Communication paths between the processors and the memories are implemented using the previously defined set of bus components.

C. Construction of the System Model in BIP

Given the BIP models of respectively the application software and hardware architecture, the construction of the BIP system model is completed in two steps:

1) transformation of components in the BIP application model, namely decomposing the *SW-Channels* into data buffers and read/write FIFO access routines, and consequently breaking the atomicity of the *read/write* operations in *SW-Processes*.
2) allocation of the transformed processes and FIFO routines on hardware processors and respectively data buffers on hardware memories according to the mapping, and eventually filling up the processor and memory placeholder components.

Formally, the BIP system model conforms to the following abstract grammar:

System-Model	::=	*HW-Processor*$^{+}$. *HW-Memory*$^{+}$. *HW-Bus*$^{+}$
HW-Processor	::=	*SW-Process*$^{(t)+}$. *HdS*$^{+}$. *HW-Cpu-Scheduler* . *SW-Conn*$^{+}$
HdS	::=	*FIFO-Read* | *FIFO-Write*
SW-Conn	::=	*SW-Process-HdS* | *SW-Process-HW-Cpu-Scheduler*
	|	*HdS-HW-Cpu-Scheduler*
HW-Memory	::=	*FIFO-Buffer*$^{+}$

1) Transformation of the BIP Application Model:

In order to deploy the application software on the architecture, we need a low level implementation model for the *SW-Channels* where the control and the data are dissociated and moreover, the *read/write* operations are no longer atomic.

Splitting software channels: Every *SW-Channel* in the application software is *replaced* by a composition of *FIFO-Write*, *FIFO-Read* and a *FIFO-Buffer* atomic components (figure 9). The two former components represent the control part of the software channel, that is, the hardware dependent software routines implementing the *read/write* operations. The latter component simply represents the buffer of data.

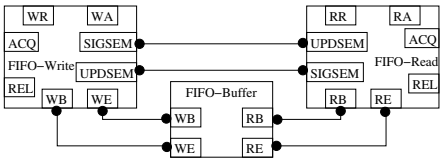

Figure 9. Low-level implementation BIP model for software channels

All the three components *FIFO-Read*, *FIFO-Write*, *FIFO-Buffer* are predefined BIP components and belong to the BIP hardware dependent software library. The *FIFO-Read* component, illustrated in figure 10, implements the *read* operation on channels. It has the ports *RR* (*Read-Request*), *RA* (*Read-Acknowledge*) for its interaction with a software process *read* operation, and ports *RB* (*Read-Begin*), *RE* (*Read-End*) for its interaction with the buffer. The *FIFO-Write* component implements the *write* action in a similar manner.

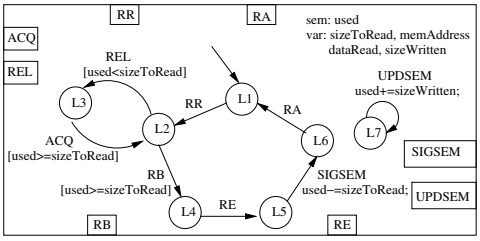

Figure 10. *FIFO-Read* component

Let us notice that the two routines, *FIFO-Write* and *FIFO-Read*, require extra synchronization with each other in order to maintain a coherent value for the used space within the buffer. This is realized by using strong synchronization between two control ports, *SIGSEM* and *UPDSEM*. Moreover, they also use the ports *REL* and *ACQ* for interaction with the processor scheduler. These ports are used to release (resp. acquire) the processor whenever the *read/write* operation is suspended (resp. resumed) due to lack (resp. presence) of available data (or available space) in the buffer.

The *FIFO-Buffer* represents a *passive* component modeling the data storage. It has ports *WB*, *WE* and *RB*, *RE* for writing and reading respectively. The ports for writing (resp. reading) synchronizes with the *FIFO-Write* (resp. *FIFO-Read*) component.

We can prove that the proposed model is a correct implementation of the *SW-Channel*. That is, the composition is a refined model of the *SW-Channel* which fully preserves the input/output behavior of the software channel.

Transformation of software processes: The splitting of *SW-Channels* as described before requires the transformation of the software processes as well.

The first transformation consists in *breaking atomicity of write and read operations*. Every transition involving an input/output port *X* is split into two transitions, labeled by fresh ports, respectively *XB* (i.e., *X-begin*) and *XE* (i.e., *X-end*). This is obtained by adding new control locations for each read/write operations in the behavior of the process.

The second transformation, completely orthogonal to the first one, consists in *adding interactions with the processor scheduler*. This transformation is needed since several processes, together with their associated FIFO access routines, are potentially mapped on the same hardware processor and must use it in mutual exclusion. The ports *ACQ* and *REL* are added for interaction with the processor scheduler. The port *ACQ* is used for acquiring and *REL* is for releasing the processor. A process acquires the processor at the start of its behavior. It releases the processor on its termination.

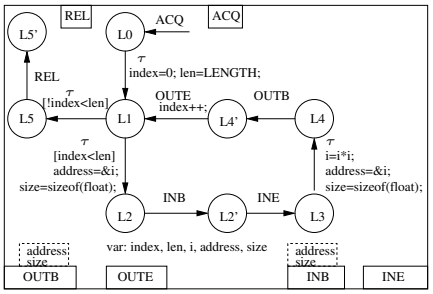

Figure 11. The transformed BIP model for the *square* process

Example 7: The transformed behavior of the square process from figure 4 is provided in figure 11.

Let us mention that, the transformed model is a correct implementation of the initial model constructed from the application software. That is, it can be formally proven that the input/output behavior of every process is fully preserved by the transformation above.

2) Allocation according to mapping:

Given an *Application-Software* and a *Hardware-Architecture*, a mapping *Map* associates software processes

to hardware processors and software channels to memories, formally:

$$\begin{array}{rcl} \textit{Mapping} & ::= & \textit{Mapping-Item}^{+} \\ \textit{Mapping-Item} & ::= & \textit{SW-Process} \mapsto \textit{HW-Processor} \\ & \mid & \textit{SW-Channel} \mapsto \textit{HW-Memory} \end{array}$$

A mapping must be *consistent*. That is, for every *write-connection* from process P to channel C in the application software, if the mapping associates P on processor H and C on memory M, there must exist a *write-path* of the form $H\ Bus_1 \ldots Bus_n\ M$ in the hardware architecture. Similarly, for every *read-connection* from channel C to process P, there must exist a *read-path* of the form $M\ Bus'_1 \ldots Bus'_m\ H$.

Example 8: For our example, we consider the following consistent mapping:

$$\begin{array}{rclrcl} \textit{generator} & \mapsto & \textit{ARM1} & \textit{C1} & \mapsto & \textit{LM1} \\ \textit{square} & \mapsto & \textit{ARM1} & \textit{C2} & \mapsto & \textit{SM} \\ \textit{consumer} & \mapsto & \textit{ARM2} & & & \end{array}$$

The construction of the system model is completed as follows. For every hardware processor, we consider the composition of all transformed software processes mapped on it, together with all the FIFO routines required to access the FIFO buffers. These components are connected as defined by the transformed software model. Additionally, the composition includes a *HW-CPU-Scheduler* component which ensures mutual exclusion for execution on the processor.

Example 9: The structure of the ARM1 processor is shown in figure 12. It contains the generator and square processes together with their associated FIFO routines respectively, the FIFO-Write for writing on C1, the FIFO-Read for reading from C1 and the FIFO-Write for writing on C2.

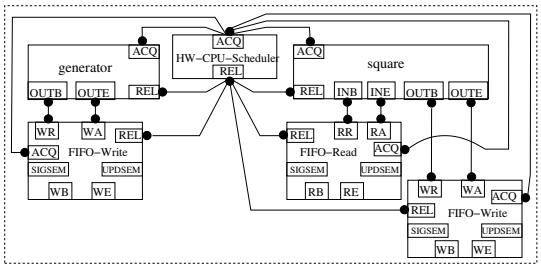

Figure 12. The BIP Model of the HW Processor *ARM1*

Moreover, for every memory component, we consider the union of all the FIFO buffers mapped onto it according to the mapping. Let us remark that no scheduling is done here: all the operations requiring access to memory are controlled by the processor and the bus, the memories being simple *passive* components, with no behavior.

Finally, the direct connections between the FIFO routines and the FIFO buffers which exist in the transformed software model are replaced by connections over the associated hardware communication paths. For example, the request/acknowledge connectors between a FIFO routine and the FIFO buffer (FB) are replaced by (i) request/acknowledge connectors from the FIFO routine to the master interface of the first bus of the associated hardware path and (ii) request/acknowledge connectors from the slave interface of the last bus of the path to the FIFO buffer.

We assume a high cache hit rate for the local variables of the processes mapped on a processor, and hence we do not model explicitly the allocation of process data in the memory. The memory is used only to model inter process data communications through the software FIFOs.

The system model can be seen as a refined implementation of the transformed BIP model of the application software according to hardware constraints. In fact, direct communication between components within the application software model have been replaced by multi-hop communication using hardware communication paths, along different buses. Moreover, mutual exclusion constraints are enforced between components running on the same hardware processors. These transformations do not impact the input/output behavior of the application. This can be proved by establishing a trace equivalence between the input and the transformed model. Nevertheless, the transformations reveal all the non-functional constraints the hardware architecture put on the execution due to contention for bus and memory access, bus access and transfer latencies, contention for processor, etc. These constraints are mandatory for an accurate performance evaluation of the application mapped on the hardware architecture.

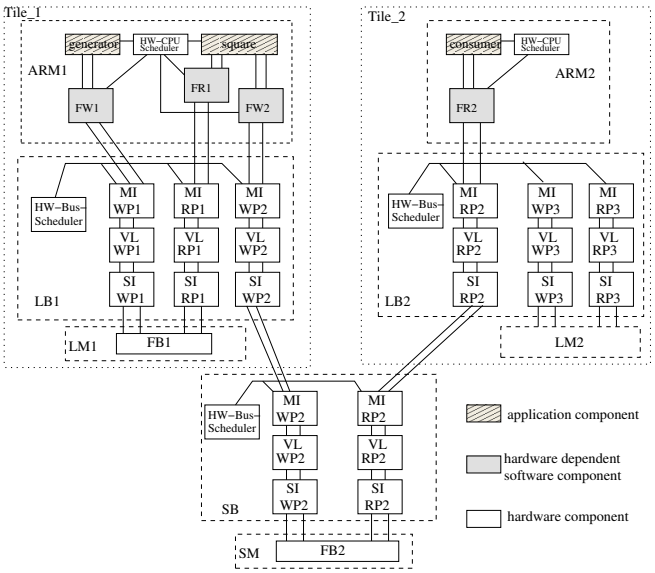

Figure 13. The BIP system model of generator-square-consumer application software mapped into 2-tile ARM hardware architecture

Example 10: Figure 13 shows the complete system model obtained for the mapping of the software application given in figure 6 to the hardware architecture of example 4 according

to the mapping from example 8.

IV. Performance Estimation on System Model

We provide an infrastructure for performance estimation of the system model based on native BIP simulation. The process is dynamic and based on fine granular analysis of code generated for the target platform, using weight table profiling, as shown in figure 1.

A. Instrumenting the System Model

The system model is instrumented with the profiling API, embedded in the behavior of the SW-Processes. Every block of code, except the read/write calls, is instrumented by inserting profiling function calls at its start and at its end. These calls invoke the profiler which provides accurate execution times.

The instrumented BIP system model is used as such by the BIP tool-chain for compilation and execution using BIP native simulator. On execution, the profiler is invoked , which dynamically estimates the computation time of the current block of code of the SW-Processes. The estimated execution time is recorded by dedicated observers for delay measurements.

The observers added in the system model are timed BIP components and monitor both the computation and the communication delays. The communication latencies of the buses and memories are also recorded by separate sets of observers, considering the conflicts arising in the use of the buses and the memories.

B. Weight Table Profiling

We use standard tools for cross-compilation and coverage profiling of the source code for SW-Processes, generated from the system model using the BIP tool-chain. The source code is cross-compiled to generate the object code (assembly) for the target processor. The source code is also instrumented for coverage analysis. The profiler is parameterized by a weight-table, which characterizes the time of executing each elementary instruction on the target HW-Processor. The object code, instrumented sources and weight-table are used by the profiler dynamically during the simulation to estimate the execution time of transitions within processes.

V. Experiments

The method described in section III has been implemented in a tool [2]. It consists of two parts, the *frontend* that transforms the input specification into a system model, and the *backend* for performance estimation on the system model. The *frontend* uses an open source C parser called codegen [3] to parse C files that describe the behavior of the DOL processes into an intermediate model. This, along with the description of the hardware architecture and mapping information (XML description) is transformed into the system model. The *backend* uses gcov as a profiling tool for code coverage, and arm-rtems-g++ cross compiler for assembly code generation for ARM processors. The weight-table conforms to the ARM7 data sheet [4].

We experimented the method on two applications: MJPEG [22] and MPEG-2 [1], [22] described in subsections V-A and V-B respectively. We used the multi-processor ARM (MPARM [5]) with five tiles as the target architecture (a two tile MPARM is illustrated in figure 7). For the hardware model in BIP, we assumed all the local memories as SRAM with an access time of 2 cycles. The shared memory is a DRAM with an access time of 6 cycles. All CPU frequencies are assumed to be 200MHz. Communication paths are defined between all five processors using shared and local memories.

A. MJPEG Decoder

The MJPEG decoder application software reads a sequence of MJPEG frames and displays the decompressed video frames. The process network of the application is illustrated in figure 14. It contains five processes *SplitStream* (*SS*), *SplitFrame* (*SF*), *IqzigzagIDCT* (*IDCT*), *MergeFrame* (*MF*) and *MergeStream* (*MS*), and nine communication sw channels *C1*, ..., *C9*.

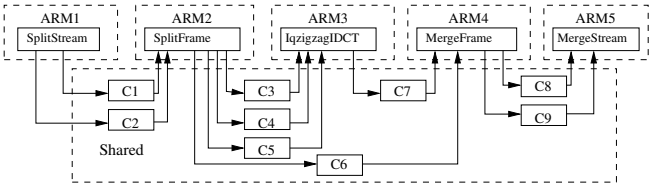

Figure 14. MJPEG Decoder application and a mapping

	$ARM1$	$ARM2$	$ARM3$	$ARM4$	$ARM5$
1	all				
2	SS, SF, IQ	MF, MS			
3	SS, SF	IQ, MF, MS			
4	SS, SF	IQ	MF, MS		
5	SS, MS	SF	IQ	MF	
6	SS	SF	IQ	MF	MS
7	SS, SF	IQ	MF, MS		
8	SS	SF	IQ	MF	MS

	$Shared$	$LM1$	$LM2$	$LM3$	$LM4$
1		all			
2	C6, C7	C1, C2, C3, C4, C5	C8, C9		
3	C3, C4, C5, C6	C1, C2	C7, C8, C9		
4	C3, C4, C5, C6, C7	C1, C2		C8, C9	
5	all				
6	all				
7	C6, C7	C1, C2, C3, C4, C5		C8, C9	
8		C1, C2	C3, C4, C5, C6	C7	C8, C9

Table I
Mapping Description of the processes and the sw channels

[2] http://www-verimag.imag.fr/BIP-System-Designer.html
[3] http://think.ow2.org
[4] http://www.datasheetarchive.com/ARM7-datasheet.html
[5] http://www-micrel.deis.unibo.it/sitonew/research/mparm.html

We experimented with eight different mappings to analyze their effect on the total computation and communication time for decoding a frame. The process and the sw channel mappings are illustrated on table I.

For the mappings described above, a system model contains about 50 BIP atomic components and 220 BIP connectors, and consists of approximately 6K lines of BIP code, generating around 19.5K lines of C code for simulation.

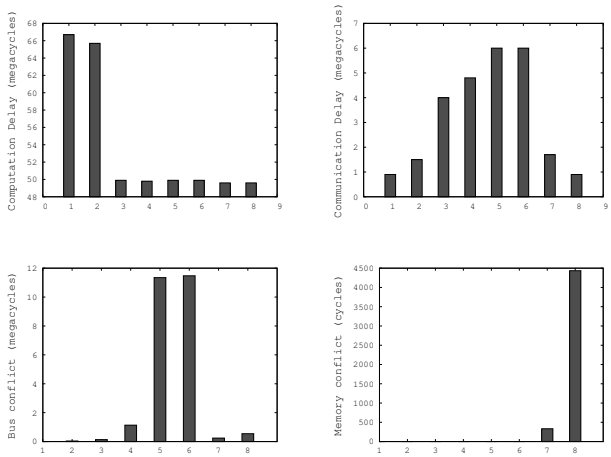

Figure 15. Mjpeg Performance Analysis Results

The total computation and communication delays for decoding a frame for different mappings are shown in figure 15. Mapping (1) produces the worst computation time as all processes are mapped to a single processor. Mapping (2) uses two processors, still the performance does not improve much. But (3) gives much better performance as the computation load is balanced. The other mappings can not produce better performance as the load can not be further distributed, even if more processors are used. The communication overhead is reduced if we map more channels to the local memories of the processors. The bus and memory access conflicts are shown in figure 15. As more channels are mapped to the local memory, the shared bus contention is reduced. However, this might increase the local memory contention, as shown for (8).

B. MPEG2 Decoder

The MPEG2 decoder application decodes a set of moving pictures and associated audio information. We used an application case study where there are seven processes *DispatchGops* (*DG*), *DispatchMb* (*DM*), *DispatchBlocks* (*DB*), *TransformBlock* (*TB*), *CollectBlocks* (*CB*), *CollectMb* (*CM*) and *CollectGops* (*CG*) and six software channels *C1*, ..., *C6*. The process and the sw channel mappings are illustrated on table II.

For the MPEG-2 case study the BIP System Model contains about 90 BIP atomic components, 340 BIP connectors and 30K lines of BIP code generating approximately 100K

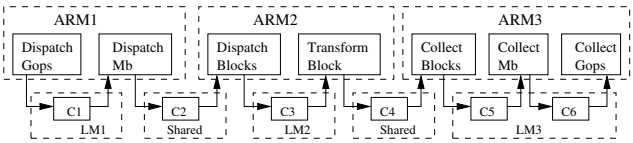

Figure 16. MPEG-2 Decoder application and a mapping

	$ARM1$	$ARM2$	$ARM3$	$ARM4$	$ARM5$	
1	all					
2	DG, DM, DB, TB	CB, CM, CG				
3	DG, DM	DB, TB	CB, CM, CG			
4	DG	DM, DB	TB	CB, CM, CG		
5	DG	DM, DB	TB	CB, CM	CG	
6	DG, DM	DB	TB	CB	CM, CG	
7	DG	DM, DB	TB	CB, CM	CG	

	$Shared$	$LM1$	$LM2$	$LM3$	$LM4$	$LM5$
1		all				
2	C4	C1, C2, C3	C5, C6			
3	C2, C4	C1	C3	C5, C6		
4	C1, C3, C4		C2		C5, C6	
5	C1, C3, C4, C6		C2		C5	
6	C2, C3, C4, C5	C1				C5
7		C1	C2, C3	C4	C5, C6	

Table II
MAPPING DESCRIPTION OF THE PROCESSES AND THE SW CHANNELS

lines of C code. The total computation and communication delays for decoding 5 frames for different mappings are shown in figure 17. The MPEG-2 process network is characterized as computationally intensive. The more we distribute the computational load to different CPUs, the smaller is the computational delay. Since the SW-channels are few, there is small difference in the communication delays between the different mappings, except for mapping (1) where all processes and SW-channels are mapped on a single tile. However, as we distribute the processes into more tiles, the communication delay increases and more bus conflicts occur. The best throughput is achieved in Mapping (7) due to the usage of five CPUs and their local memories.

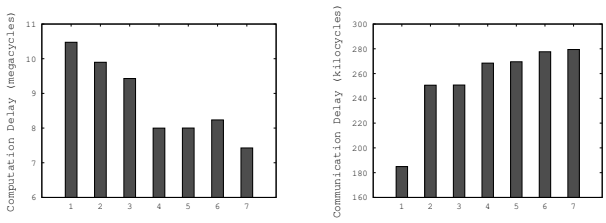

Figure 17. Mpeg-2 Performance Analysis Results

VI. CONCLUSION

The presented method allows generation of a correct-by-construction model of a mixed hardware/software system from application software, a description of the hardware architecture and a mapping. The method is completely automated and supported by BIP tools. The system model is obtained by refining the application software model and composing it with the hardware architecture model. The

composition is defined by the mapping. BIP instruments the incremental construction of the models. Its expressiveness allows the integration of architecture constraints into the application model without suffering complexity explosion.

The method clearly separates software and hardware design issues. It is also parameterized by design choices related to resource management such as scheduling policies, memory size and execution times. This allows mastering the complexity and appreciation of the impact of each parameter on system behavior.

When the generated system model is adequately instrumented with execution times, it can be used for performance analysis and design space exploration. Experimental results show the feasibility of the system model for fine granular analysis of the effects of architecture and mapping constraints on the system behavior. The method is tractable and allows design space exploration to determine optimal solutions.

Future work includes extension to other programming models for the application software and richer hardware architecture models that includes DMA (Direct Memory Access) Controller, Bus Bridge and Network on Chip communication. Moreover, we plan to include statistical model checking on system models consisting of multiple applications running on complex multicore architectures for performance analysis, as in [23].

References

[1] L. Thiele, I. Bacivarov, W. Haid, and K. Huang, "Mapping applications to tiled multiprocessor embedded systems," in *ACSD*. IEEE Computer Society, 2007, pp. 29–40.

[2] A. Basu, M. Bozga, and J. Sifakis, "Modeling Heterogeneous Real-time Components in BIP," in *SEFM*, 2006, pp. 3–12.

[3] K. Asanovic et al., "The landscape of parallel computing research: A view from berkeley," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2006-183, Dec 2006.

[4] B. Bonakdarpour, M. Bozga, M. Jaber, J. Quilbeuf, and J. Sifakis, "From high-level component-based models to distributed implementations," in *EMSOFT*, 2010.

[5] D. Abhijit et al., "A next-generation design framework for platform-based design," in *DVCon 2007*, February 2007.

[6] B. Twan et al., "Model-driven design-space exploration for embedded systems: The octopus toolset," in *ISoLA (1)*, 2010, pp. 90–105.

[7] T. Grtker, S. Liao, G. Martin, and S. Swan, *System Design with SystemC*. Kluwer Academic Publishers, 2002.

[8] C. Haubelt, T. Schlichter, J. Keinert, and M. Meredith, "Systemcodesigner: automatic design space exploration and rapid prototyping from behavioral models," in *DAC*, 2008, pp. 580–585.

[9] P. Lieverse, T. Stefanov, P. van der Wolf, and E. Deprettere, "System level design with SPADE: an M-JPEG case study," *ICCAD*, pp. 31–38, 2001.

[10] C. Erbas, A. D. Pimentel, M. Thompson, and S. Polstra, "A framework for system-level modeling and simulation of embedded systems architectures," *EURASIP J. Embedded Syst.*, vol. 2007, pp. 2–2, 2007.

[11] M. Moy, F. Maraninchi, and L. Maillet-Contoz, "Lussy: A toolbox for the analysis of systems-on-a-chip at the transactional level," in *ACSD*, 2005, pp. 26–35.

[12] B. Kienhuis, E. F. Deprettere, K. A. Vissers, and P. van der Wolf, "An approach for quantitative analysis of application-specific dataflow architectures," in *ASAP*, 1997, pp. 338–349.

[13] I. Moussa, T. Grellier, and G. Nguyen, "Exploring sw performance using soc transaction-level modeling," in *DATE*, 2003, pp. 20 120–20 125.

[14] L. Thiele, S. Chakraborty, and M. Naedele, "Real-time calculus for scheduling hard real-time systems," in *ISCAS*, vol. 4, no. March. IEEE, 2002, pp. 101–104.

[15] R. Henia et al., "System-level performance analysis - the SymTA/S approach," in *IEE Proceedings Computers and Digital Techniques*, vol. 152, no. 2, 2005, pp. 148–166.

[16] R. B. Salah, M. Bozga, and O. Maler, "Compositional timing analysis," in *EMSOFT*, 2009, pp. 39–48.

[17] Y. Abdeddaim, E. Asarin, and O. Maler, "Scheduling with timed automata," *Theoretical Computer Science*, vol. 354, pp. 272–300, 2006.

[18] S. Künzli, F. Poletti, L. Benini, and L. Thiele, "Combining simulation and formal methods for system-level performance analysis," in *DATE*, 2006, pp. 236–241.

[19] G. Kahn, "The semantics of a simple language for parallel programming," in *Information processing*, J. L. Rosenfeld, Ed. Stockholm, Sweden: North Holland, Amsterdam, Aug 1974, pp. 471–475.

[20] S. Bliudze and J. Sifakis, "A Notion of Glue Expressiveness for Component-Based Systems," in *CONCUR*, ser. LNCS, vol. 5201, 2008, pp. 508–522.

[21] A. Basu, S. Bensalem, M. Bozga, J. Combaz, M. Jaber, T.-H. Nguyen, and J. Sifakis, "Rigorous component-based design using the BIP framework," *IEEE Software, Special Edition – Software Components beyond Programming – from Routines to Services*, June 2011, to appear.

[22] K. Huang, "Coupling MPARM with DOL," ETH Zurich, Technical Report, Nov 2009.

[23] A. Basu, S. Bensalem, M. Bozga, B. Caillaud, B. Delahaye, and A. Legay, "Statistical abstraction and model-checking of large heterogeneous systems," in *FMOODS/FORTE*, 2010, pp. 32–46.