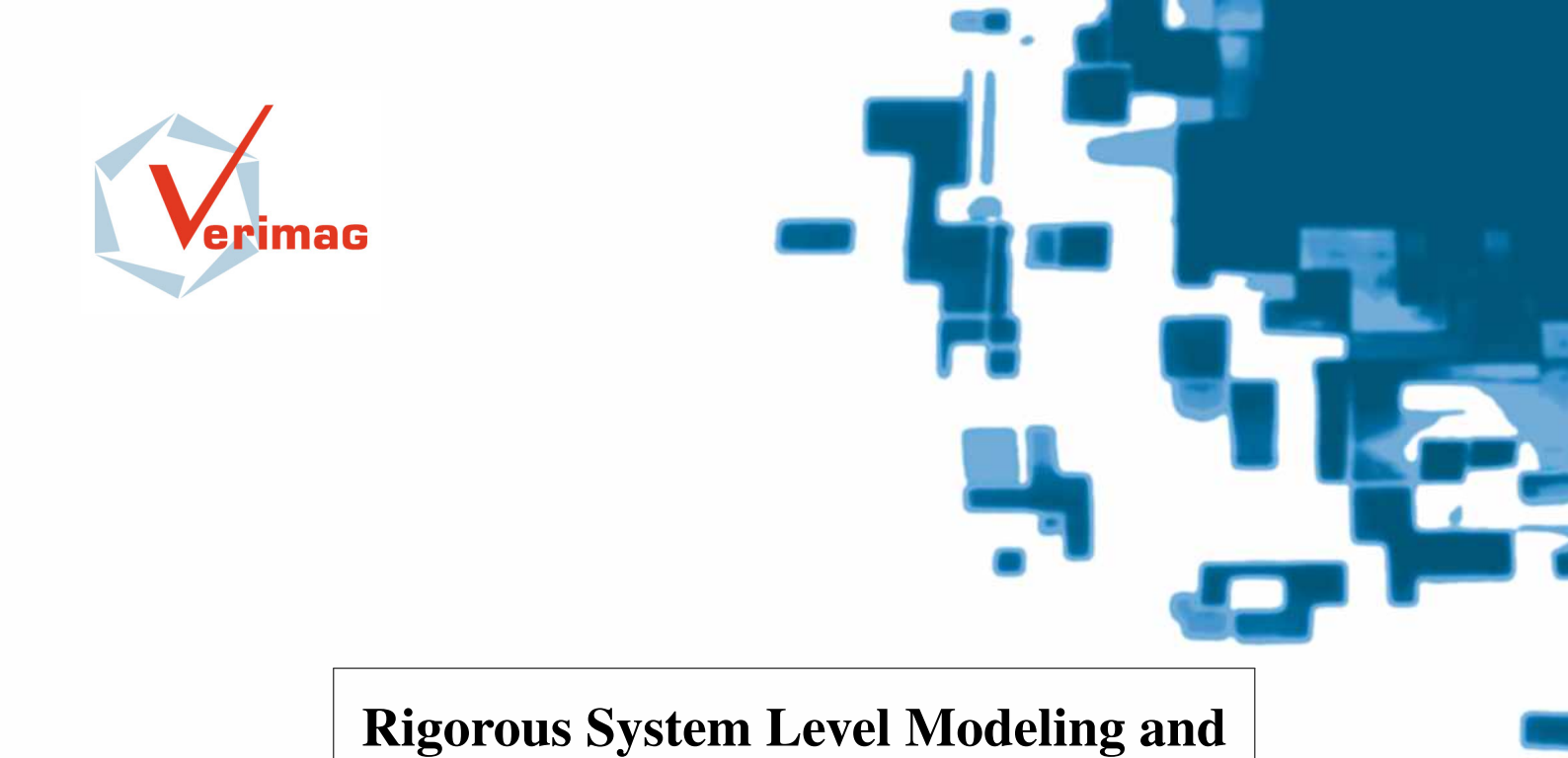

Rigorous System Level Modeling and Analysis of Mixed HW/SW Systems

P. Bourgos, A. Basu, M. Bozga, S. Bensalem, J. Sifakis, K. Huang

Verimag Research Report n^o TR-2011-5

15-03-2011

Unité Mixte de Recherche 5104 CNRS - INPG - UJF
Centre Equation
2, avenue de VIGNATE
F-38610 GIERES
tel : +33 456 52 03 40
fax : +33 456 52 03 50
http://www-verimag.imag.fr

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

P. Bourgos, A. Basu, M. Bozga, S. Bensalem, J. Sifakis, K. Huang

15-03-2011

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 a given hardware architecture. For application software running on multicore or distributed platforms, rigorous performance analysis techniques are essential for determining optimal implementations with respect to resource management criteria. 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. It also provides a basis for performance evaluation and automated code generation for target architectures. The method has been implemented as a tool chain that uses DOL (Distributed Operation Layer [23]) as the frontend for specifying the application software and hardware architecture, and BIP (Behavior Interaction Priority [6]) 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.

Keywords: System Level Design, Mixed HW/SW Systems, Mapping, Hardware Constraints, Simulation, Performance Evaluation

Reviewers:

Notes: 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)

How to cite this report:

```
@techreport {TR-2011-5,
  title = {Rigorous System Level Modeling and Analysis of Mixed HW/SW Systems},
  author = {P. Bourgos, A. Basu, M. Bozga, S. Bensalem, J. Sifakis, K. Huang},
  institution = {{Verimag} Research Report},
  number = {TR-2011-5},
  year = {2011}
}
```

1 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 require 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 by the underlying hardware architecture. Both models are described in BIP [6], 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 [8]. 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.

Most of the frameworks for mixed HW/SW systems are based on SystemC [10] as a language for modeling at various levels of abstractions. Various tools and associated design methodologies emerged e.g., SystemCoDesigner [11], Spade [18], Sesame [9] 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 such as SpecC [21], or using tools like Pinapa [20], however, they remain marginal and difficult to use mainly because of the complexity of some of the SystemC components (i.e., simulator) and their dependencies on C++.

One of the main needs for rigorous system model is performance evaluation. Simulation based methods use ad-hoc executable system models e.g., models in SystemC [10,19]. They 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 [18], Sesame [9]. There exist much faster techniques that work on abstract system models e.g., Real Time Calculus [24] and SymTA/S [12]. 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 represent a significant modelling effort and, if done through a manual process, the results are not guaranteed accurate. Similar drawbacks exists for performance analysis techniques based on Timed-Automata [22, 16, 2, 13]. These can be used for modeling and solving scheduling problems. An approach combining simulation and analytic models is presented in [17], where simulation results can be propagated to analytic models and vice versa through well defined interfaces.

The paper is structured as follows. Section 2 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 3. Section 4 describes the performance estimation technique applied on the system model. Finally, experimental results are provided in section 5. In section 6 we conclude and discuss future work directions.

2 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 [15]. 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) [23] 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 [6, 7] and the associated toolbox[1].

The BIP Component Framework

Our method is entirely supported by the BIP language and its associated toolset and design flow [5]. The BIP language is a notation which allows building complex systems by coordinating the behavior of a set of atomic components. 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. The first layer describes the interactions between components by using connectors. An interaction is a set of strongly synchronized ports. It is labelled with guards (enabling conditions) and data transfer functions (data exchange) between interacting components. The second layer

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

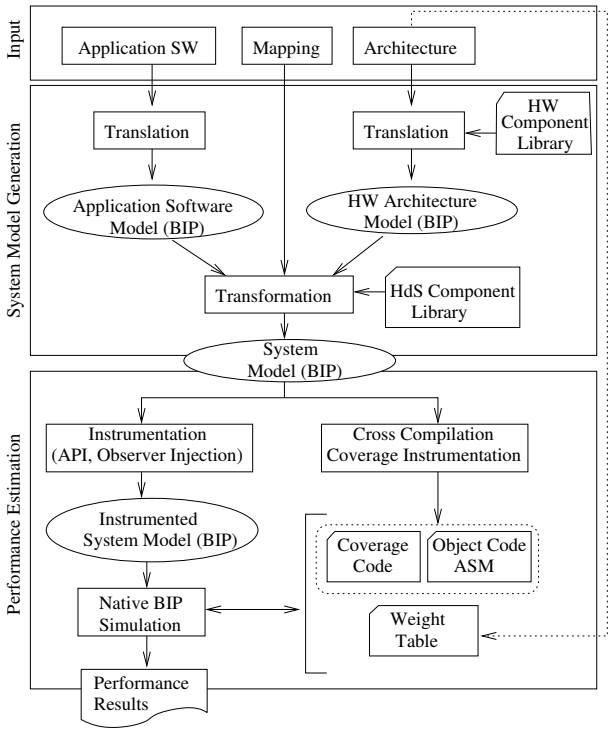

Figure 1: The Flow for System Model Construction and Performance Evaluation

describes dynamic priorities between interactions and is used to express scheduling policies. The combination of interactions and priorities characterizes the overall architecture of a component. It confers BIP strong expressiveness that cannot be matched by other languages [7]. BIP has clean operational semantics that 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.

3 Deriving System Model

We use the DOL framework [23] as frontend to describe the input specification. The input specification includes the application software, the hardware architecture and the mapping. The construction of the system model in BIP from the input DOL specification is done in three steps, as described in the following subsections.

3.1 Construction of Application Software Model in BIP

An application software in DOL [23] 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:

$$
\begin{array}{rcl}
\textit{Application-Software} & ::= & \textit{SW-Process}^{+} \;.\; \textit{SW-Channel}^{+} \;.\; \textit{SW-Connection}^{+} \\
\textit{SW-Process} & ::= & (\textit{SW-InPort})^{*} \;.\; (\textit{SW-OutPort})^{*} \;.\; \textit{SW-Behavior} \\
\textit{SW-Channel} & ::= & \textit{SW-RecvPort} \;.\; \textit{SW-SendPort} \;.\; \textit{SW-Channel-Behavior} \\
\textit{SW-Connection} & ::= & \textit{SW-Read-Connection} \\
 & | & \textit{SW-Write-Connection} \\
\textit{SW-Write-Connection} & ::= & \textit{SW-OutPort} \;.\; \textit{SW-RecvPort} \\
\textit{SW-Read-Connection} & ::= & \textit{SW-SendPort} \;.\; \textit{SW-InPort} \\
\textit{SW-Behavior} & ::= & \textit{a-C-program} \\
\textit{SW-Channel-Behavior} & ::= & \textit{FIFO-Param}^{+}
\end{array}
$$

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 behaviour is described using C programs with a particular structure (see figure 3 for a concrete example). In general, the behaviour 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.

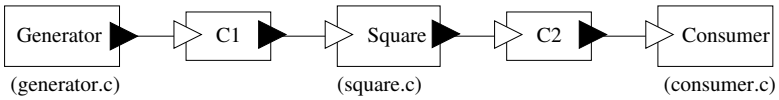

Figure 2: An application software

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*.

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.

3.1.1 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

```
#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

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.

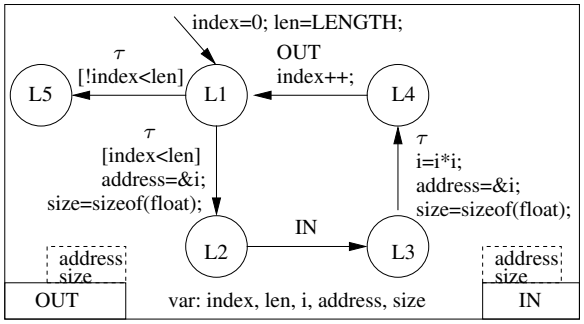

Figure 4: The model of the *square* process as an atomic BIP component

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*.

3.1.2 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*.

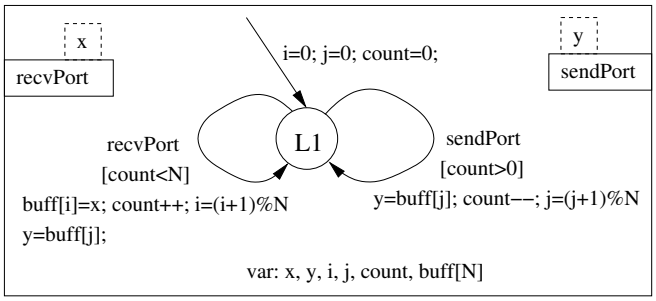

Figure 5: SW-channel (FIFO) in BIP

3.1.3 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.

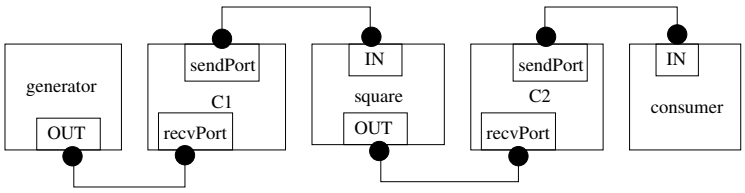

Figure 6: Application software model in BIP

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

3.2 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). Com-

munication paths define the connections between computational resources. More formally, we consider the family of hardware architectures that can be represented in DOL [23] and are abstracted by the following grammar:

$$\begin{aligned}
\textit{Hardware-Architecture} &::= \textit{HW-Resource}^{+} \,.\, \textit{HW-Comm-Path}^{+} \\
\textit{HW-Resource} &::= \textit{HW-Processor} \mid \textit{HW-Memory} \mid \textit{HW-Bus} \\
\textit{HW-Comm-Path} &::= \textit{HW-Read-Path} \,.\, \textit{HW-Write-Path} \\
\textit{HW-Read-Path} &::= \textit{HW-Memory} \,.\, \textit{HW-Bus}^{+} \,.\, \textit{HW-Processor} \\
\textit{HW-Write-Path} &::= \textit{HW-Processor} \,.\, \textit{HW-Bus}^{+} \,.\, \textit{HW-Memory}
\end{aligned}$$

A hardware architecture can be equally viewed as a graph with three kinds of nodes (processor, memory and bus) and edges defined according to the communication paths.

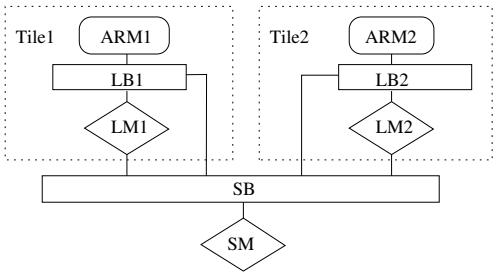

Figure 7: A multi-core hardware architecture with two ARM tiles

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

All these components are predefined and belongs to the BIP hardware library. They have identical interfaces for the tranport 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 [6]. 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 4.

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.

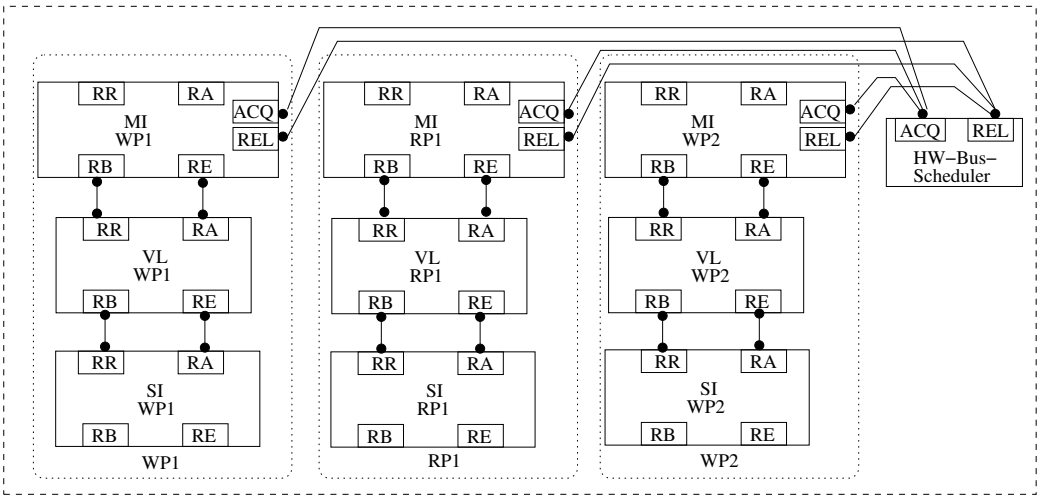

Figure 8: The BIP Model of the *LB1* bus

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 successessive components, corresponding to the *write* and *read* operations.

Example 6 Figure 9 illustrates the BIP hardware model of the 2-Tile ARM arhitecture of example 4. Communication paths between the processors and the memories are implemented using the previously defined set of Bus components.

3.3 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.

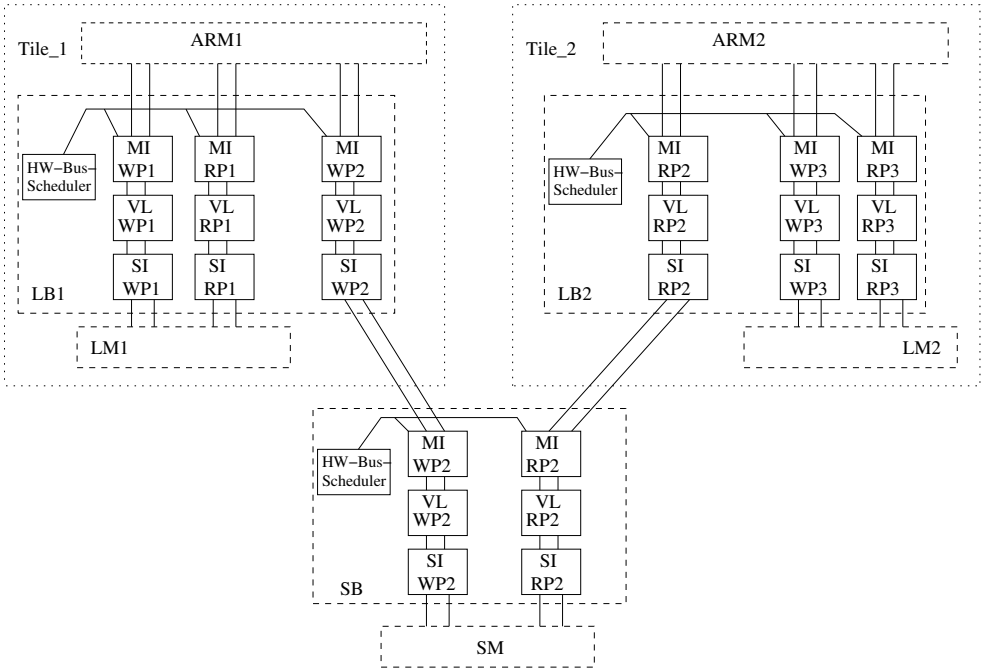

Figure 9: The BIP model of the two ARM hardware architecture

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-Routine*$^+$.
		HW-Cpu-Scheduler . *SW-Connection*$^+$
HdS-Routine	::=	*FIFO-Read* | *FIFO-Write*
SW-Connection	::=	*SW-Process-HdS-Routine*
	|	*SW-Process-SW-Scheduler*
	|	*HdS-Routine-SW-Scheduler*
HW-Memory	::=	*FIFO-Buffer*$^+$

3.3.1 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. 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.

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 11, 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.

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

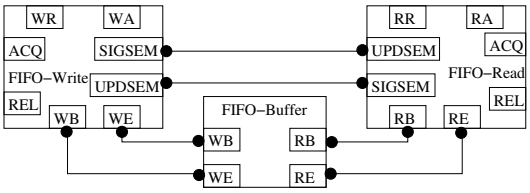

Figure 10: Low-level implementation BIP model for software channels

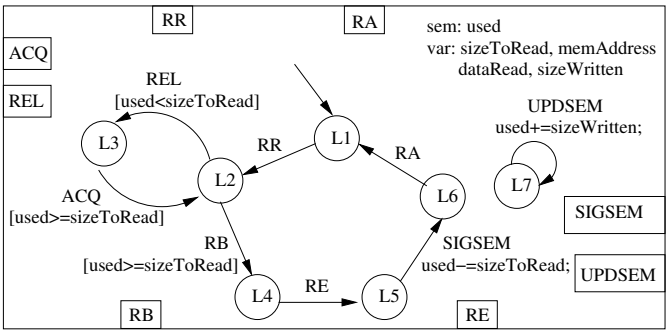

Figure 11: *FIFO-Read* component

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 behaviour of the software channel.

Transformation of software processes

The splitting of *SW-Channels* as described before require 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 process 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.

Example 7 The transformed behavior of the *square* process from figure 4 is provided in figure 12.

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.

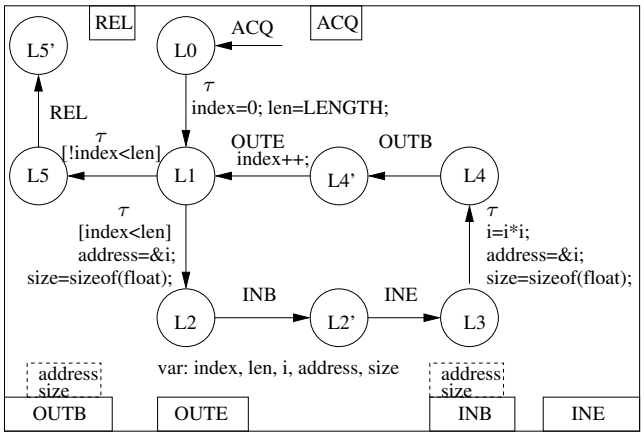

Figure 12: The transformed BIP model for the *square* process

3.3.2 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:

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

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:

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

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 13. 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*. The first one is used by the *generator* whereas the last two are used by the *square*.

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 at processor and bus, the memories being simple *passive* components, with no behaviour.

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.

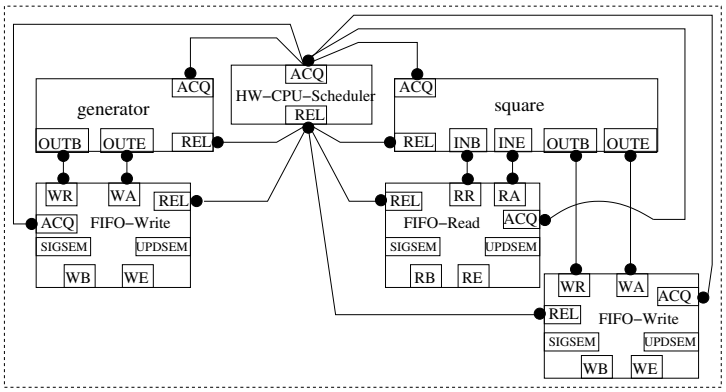

Figure 13: The BIP Model of the HW Processor *ARM1*

We assume high cache 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 communications 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 transformation does not impact the input/output behavior of the application, that is, the functionality of the application model within the system model is fully preserved. Nevertheless, they 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.

Example 10 The figure 14 shows the complete system model obtained for the mapping of the software application given in figure 6 to the hardware architecture given in figure 9 according to the mapping from the example 8.

4 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. It is used to obtain accurate execution times for the code of processes on the target platform. The method is described in the following subsections.

4.1 Instrumenting the System Model

The system model is instrumented with the profiling API. The API calls are embedded in the behavior of the SW-Processes. Every block of code, except for the read/write calls are instrumented by inserting profiling function calls at the start and at the end of the executable block of code associated with the transition. These calls invoke the profiler and are used to get 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 through the use of profiling API, which dynamically estimates the computation time of the current block of code of the SW-Processes. The API propagates the estimated execution time to the SW-Process, which is recorded by dedicated observers for computation delay measurements.

The observers added in the system model are timed BIP components and monitor both the computation and the communication delay results. The communication latencies of the buses and memories are

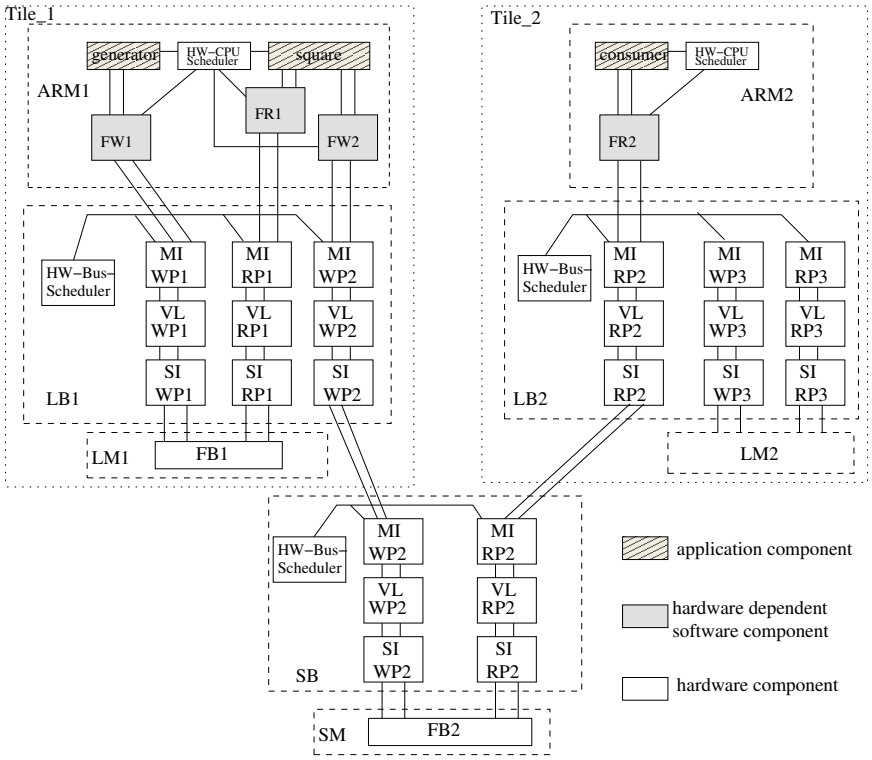

Figure 14: The BIP system model of generator-square-consumer application software mapped into 2-tile ARM hardware architecture

recorded by separate sets of observers. We also have observers for measuring conflicts in the use of bus and memories.

4.2 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.

The profiler is implemented as a runtime process with an API, that provides routines for: 1) notifying the portion of the code to be profiled, and 2) sending back the estimation times.

5 Experiments

The method described in section 3 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. DOL is used as an input specification framework to describe the application software, the hardware architecture as well as the mapping.

[2] http://www-verimag.imag.fr/BIP-System-Designer.html

The *frontend* uses an open source C parser called codegen [1] 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 is created by considering the instruction set from the ARM7 data sheet [3].

We experimented the method on two applications: MJPEG [14] and MPEG-2 [23, 14] described in subsections 5.1 and 5.2 respectively. We used the multi-processor ARM (MPARM [4]) 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.

5.1 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 15. It contains five processes *SplitStream* (*SS*), *SplitFrame* (*SF*), *IqzigzagIDCT* (*IDCT*), *MergeFrame* (*MF*) and *MergeStream* (*MS*), and nine communication sw channels *C1*, . . . , *C9*.

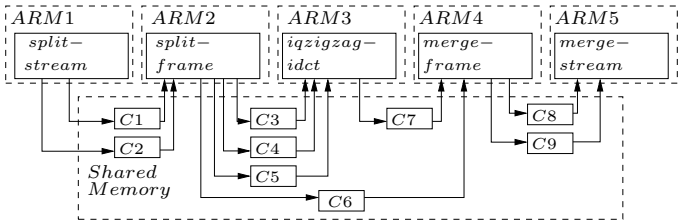

Figure 15: MJPEG Decoder application and a mapping

	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 1: Mapping Description of the processes and the sw channels

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 the illustrated on table 1.

[3] http://www.datasheetarchive.com/ARM7-datasheet.html
[4] http://www-micrel.deis.unibo.it/sitonew/research/mparm.html

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.

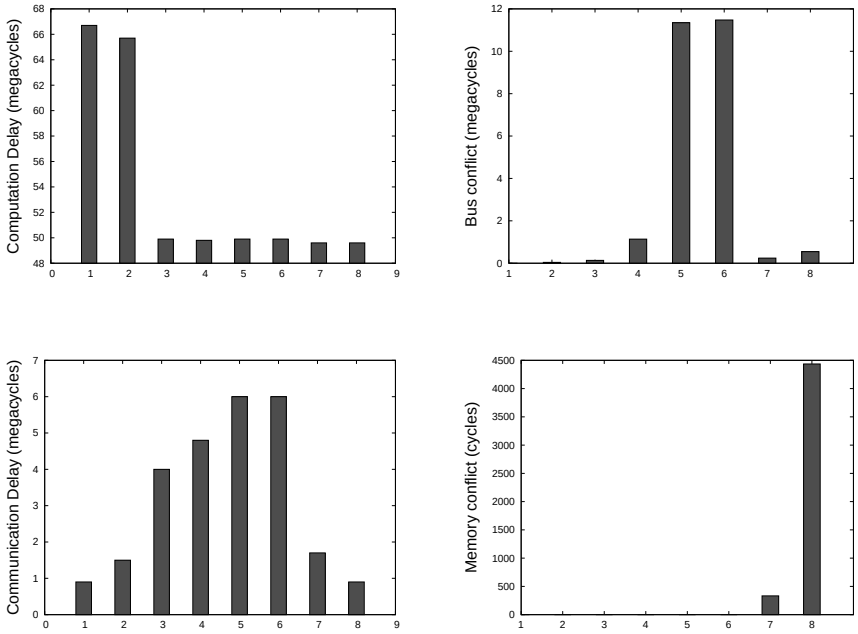

Figure 16: Mjpeg Performance Analysis Results

The total computation and communication delays for decoding a frame for different mappings are shown in figure 16. 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 16. 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).

5.2 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 *dispatch_gops* (*DG*), *dispatch_mb* (*DM*), *dispatch_blocks* (*DB*), *transform_block* (*TB*), *collect_blocks* (*CB*), *collect_mb* (*CM*) and *collect_gops* (*CG*) and six software channels $C1, \ldots, C6$. The process and the sw channel mappings are illustrated on table 2.

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

The total computation, communication and throughput delays for decoding 5 frames for different mappings are shown in figure 18. The MPEG-2 process network is characterized as computationally intensive. Thus, the more we distribute the computational load to different CPUs the smaller the computational delay is. Since the sw channels are few, there is small difference regarding the communication delay between mappings, except for Mapping (1) where all processes and channels are mapped on a single tile. However, the more we distribute the process network into more tiles, the greater the communication delay becomes and the more frequently bus conflicts occur.

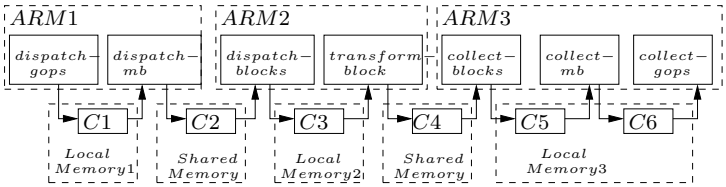

Figure 17: MPEG-2 Decoder application and a mapping

	$ARM1$	$ARM2$	$ARM3$	$ARM4$	$ARM5$	
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 2: Mapping Description of the processes and the sw channels

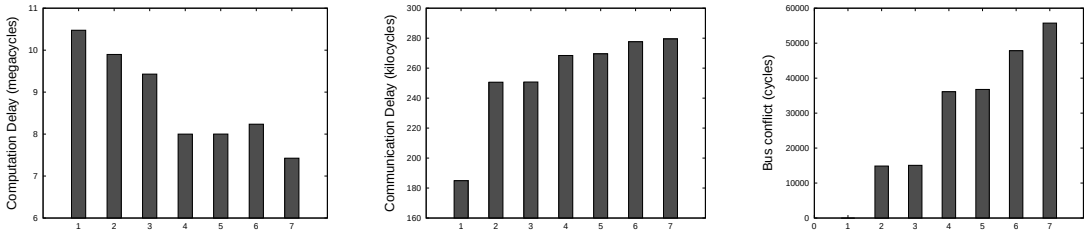

Figure 18: Mpeg-2 Performance Analysis Results

6 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 instrumentates incremental construction of the models. Its expressiveness allows the integration of architecture constraints into the application model without suffering complexity explosion. DOL is used as a front-end.

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. Another issue is the optimization of our performance estimation method. We can enrich the system model with performance parameters through an initial calibration step, instead of dynamic analysis of the target code, to fasten the simulation. 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 [4].

References

[1] http://think.ow2.org.

[2] Yasmina Abdeddaim, Eugene Asarin, and Oded Maler. Scheduling with timed automata, 2006.

[3] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[4] Ananda Basu, Saddek Bensalem, Marius Bozga, Benoît Caillaud, Benoît Delahaye, and Axel Legay. Statistical abstraction and model-checking of large heterogeneous systems. In *FMOODS/FORTE*, pages 32–46, 2010.

[5] Ananda Basu, Saddek Bensalem, Marius Bozga, Jacques Combaz, Mohamad Jaber, Thanh-Hung Nguyen, and Joseph 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.

[6] Ananda Basu, Marius Bozga, and Joseph Sifakis. Modeling Heterogeneous Real-time Components in BIP. pages 3–12. Ieee, 2006.

[7] S. Bliudze and J. Sifakis. A Notion of Glue Expressiveness for Component-Based Systems. In *Concurrency Theory CONCUR'08 Proceedings*, volume 5201 of *LNCS*, pages 508–522. Springer, 2008.

[8] Borzoo Bonakdarpour, Marius Bozga, Mohamad Jaber, Jean Quilbeuf, and Joseph Sifakis. From high-level component-based models to distributed implementations. In *EMSOFT*, 2010.

[9] Cagkan Erbas, Andy D. Pimentel, Mark Thompson, and Simon Polstra. A framework for system-level modeling and simulation of embedded systems architectures. *EURASIP J. Embedded Syst.*, 2007:2–2, 2007.

[10] Thorsten Grtker, Stan Liao, Grant Martin, and Stuart Swan. *System Design with SystemC*. Kluwer Academic Publishers, 2002.

[11] Christian Haubelt, Thomas Schlichter, Joachim Keinert, and Mike Meredith. Systemcodesigner: automatic design space exploration and rapid prototyping from behavioral models. In *DAC*, pages 580–585, New York, NY, USA, 2008. ACM.

[12] Rafik Henia, Arne Hamann, Marek Jersak, Razvan Racu, Kai Richter, and Rolf Ernst. System level performance analysis the SymTA/S approach, 2005.

[13] Anders Hessel, Kim G Larsen, Brian Nielsen, Paul Pettersson, and Arne Skou. Time-optimal Real-Time Test Case Generation using U PPAAL, 2004.

[14] Kai Huang. Coupling MPARM with DOL. Technical report, ETH Zurich, Nov 2009.

[15] G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information processing*, pages 471–475, Stockholm, Sweden, Aug 1974. North Holland, Amsterdam.

[16] Hermann Kopetz and Güenther Bauer. The time-triggered architecture, 2003.

[17] Simon Künzli, Francesco Poletti, Luca Benini, and Lothar Thiele. Combining simulation and formal methods for system-level performance analysis. In *DATE '06: Proceedings of the conference on Design, automation and test in Europe*, pages 236–241, 3001 Leuven, Belgium, Belgium, 2006. European Design and Automation Association.

[18] P. Lieverse, T. Stefanov, P. van der Wolf, and E. Deprettere. System level design with SPADE: an M-JPEG case study. *IEEE/ACM International Conference on Computer Aided Design. ICCAD 2001. IEEE/ACM Digest of Technical Papers (Cat. No.01CH37281)*, pages 31–38, 2001.

[19] Imed Moussa, Thierry Grellier, and Giang Nguyen. Exploring sw performance using soc transaction-level modeling. In *DATE*, page 20120, Washington, DC, USA, 2003. IEEE Computer Society.

[20] Matthieu Moy, Florence Maraninchi, and Laurent Maillet-Contoz. Pinapa: an extraction tool for systemc descriptions of systems-on-a-chip. In *Proceedings of the 5th ACM international conference on Embedded software*, EMSOFT '05, pages 317–324, 2005.

[21] Wolfgang Mueller, Rainer Dömer, and Andreas Gerstlauer. The formal execution semantics of SpecC. *Proceedings of the 15th international symposium on System Synthesis ISSS 02*, page 150, 2002.

[22] Ramzi Ben Salah, Marius Bozga, and Oded Maler. Compositional timing analysis. In *EMSOFT*, pages 39–48, 2009.

[23] Lothar Thiele, Iuliana Bacivarov, Wolfgang Haid, and Kai Huang. Mapping applications to tiled multiprocessor embedded systems. In *ACSD '07: Proceedings of the Seventh International Conference on Application of Concurrency to System Design*, pages 29–40, Washington, DC, USA, 2007. IEEE Computer Society.

[24] Lothar Thiele, Samarjit Chakraborty, and Martin Naedele. Real-time calculus for scheduling hard real-time systems, 2002.