Coupling MPARM with DOL

Kai Huang, Wolfgang Haid, Iuliana Bacivarov, Lothar Thiele

Abstract. This report summarizes the work of coupling the MPARM cycle-accurate multi-processor simulator with the Distributed Operation Layer (DOL) system-level MPSoC development framework. The main contribution of this work is the runtime environment which enables the execution of the DOL applications on top of the MPARM simulator. The runtime environment is automatically generated in a correct-by-construction manner from the DOL specifications by a software synthesis procedure.

1 Introduction

Multi-Processor System-on-Chip (MPSoC) becomes prevalent for modern embedded system design, which provides high computation power, lower power dissipation, and low cost. Nevertheless, how to program an MPSoC is still an open question. The Distributed Operation Layer (DOL) [10] framework proposes a specific way to answer this question, i.e., adopting the Kahn process network [5] as the model of computation and C/C++ plus XML as the programming language.

Within the DOL framework, an MPSoC is specified separately into application, architecture, and mapping, following the orthogonalization of concerns methodology [6]. The DOL application is specified in an architecture-independent manner for the potential design space exploration of heterogeneous architectures. Therefore, to execute a DOL application on top of a given architecture, a run-time environment is requested to provide architecture-dependent services, such as multi-processing of each processor core and the inter-process communication.

In this work, we couple the DOL framework with the MPARM [1] cycle-accurate simulator such that an application specified within the DOL can efficiently execute on top of the MPARM platform in a correct-by-construction manner. The MPARM is a cycle-accurate simulator which consists of variable number of ARM cores connected via a shared AMBA bus. In this report, we focus on the run-time environment built on top of the MPARM simulator. We develop a multi-processing implementation based on the RTEMS [9] operating system. For the inter-process communication, four mechanisms are introduced to exploit the potential of the MPARM architecture.

The rest of the report is structured as follows: An overview of the DOL framework and MPARM are presented in Section 2 and 3. The details of the run-time environment are explained in Section 4. We present experiment results in Section 5 and conclude the report in Section 6.

2 DOL Programming Framework

The DOL follows the well-known Y-chart design methodology [**?**], separating the modeling of a system into three parts, i.e., the application, the architecture, and the mapping. A screenshot of a simple model in a graphical front-end is depicted in Fig. 1.

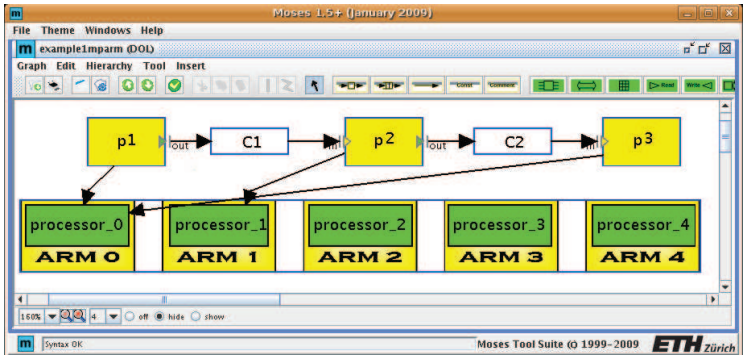


Fig. 1: Screenshot of the DOL graphical front-end, showing the mapping of a simple producer-consumer application onto MPARM.

In this section, we shortly introduce the major features of the DOL programming environment. For detailed information, we refer to [2].

2.1 Application Programming Model

To model the application, we adopt the Kahn process network (KPN) model of computation [5] that assumes a network of concurrent autonomous *processes* communicating in a point-to-point fashion via FIFO *software channels* with blocking read and non-blocking write semantics. The process network model of computation is well suited to model streaming applications because it directly exposes the available data and functional parallelism. In addition, the explicit differentiation between computation and communication allows to separately analyze their influence on the target systems. Furthermore, the determinism of KPN, i.e., the result of the computation is independent on the timing, allows to separate the functionality of the application from the underlying architectures. Finally, the KPN model is rather general such that most of the popular analysis models, e.g. SDF and Mark graph, can be derived by imposing certain restrictions. An example of a three-process network can be seen at the upper part of Fig. 1.

To specify the application, a process-oriented programming paradigm is designed in which the structure of the application and the source code are separated

as well. The structure of the process network is syntactically represented in a customized XML format. An example is shown in Fig. 2 which represents the process network in Fig. 1.

```
1  <variable value="3" name="N" />
2
3  <process name="P1">
4      <port type="output" name="10" />
5      <source type="c" location="generator.c" />
6  </process>
7
8  <iterator variable="i" range="N">
9      <process name="P2">
10         <append function="i" />
11         <port type="input" name="0" />
12         <port type="output" name="1" />
13         <source type="c" location="square.c" />
14     </process>
15 </iterator>
16
17 <process name="P3">
18     <port type="input" name="100" />
19     <source type="c" location="consumer.c" />
20 </process>
21
22 <iterator variable="i" range="N + 1">
23     <sw_channel type="fifo" size="10" name="C2">
24         <append function="i" />
25         <port type="input" name="0" />
26         <port type="output" name="1" />
27     </sw_channel>
28 </iterator>
```

Fig. 2: A three-process process network where the `P2` can be scaled to `N` processes.

The functionality of an application is defined by the function of each individual process. To program processes, plain C/C++ is used whereby a set of coding rules needs to be respected, as the example in Fig. 3 shown. In particular, a process consists of an `init()` and a `fire()` procedure, as shown in Lines 22-26, 27-43, respectively. The `init()` procedure is called only once during the initialization of the application. Afterward, the `fire()` procedure is called repeatedly. For inter-process communication, dedicated communication primitives are defined. Blocking read and blocking write can be invoked by the `DOL_read()` and `DOL_write()` primitives, as shown in Lines 31 and 33, respectively.

2.2 Architecture and Mapping Modeling

The modeling of an architecture in DOL is at an abstract level, the granularity of which depends on the refinements that take place during the software synthesis for a specific target architecture. The DOL architecture specification consists of basic system components, their attributes, and the way they are connected, e.g., computational components like programmable processors and hardware IPs, or

```
21 ...
22 void P2_init(DOLProcess *p) {
23     p->local->index = 0;
24     p->local->len = LENGTH;
25 }
26
27 int P2_fire(DOLProcess *p) {
28     float i;
29
30     if (p->local->index < p->local->len) {
31         DOL_read((void*)PORT_IN, &i, sizeof(float), p);
32         i = i*i;
33         DOL_write((void*)PORT_OUT, &i, sizeof(float), p);
34         p->local->index++;
35     }
36
37     if (p->local->index >= p->local->len) {
38         DOL_detach(p);
39         return -1;
40     }
41
42     return 0;
43 }
```

Fig. 3: Code segment for the iterated `P2` process in Fig 2. Every instance of this process shares this same piece of source code.

communication components like buses and NoCs. Fig. 1, for instance, represents a very abstract MPARM architecture, where only the processors are shown. This specification could get more complex if further architectural features are considered during system synthesis, e.g. several communication possibilities, like CPU-driven transfers or transfers via a direct memory access (DMA) controller.

The mapping defines where and how the process network is executed on a target architecture. The mapping can be classified into two parts: the spatial domain which is referred to as binding and the temporal domain which is referred to as scheduling. In our framework, the binding defines a mapping of processes to processors and software channels to FIFO implementations. The scheduling defines the scheduling policy on each resource and the according parameters, e.g. time-division multiple access (TDMA) scheme and the associated slot length, fixed priority scheduling and the associated priorities, or static scheduling and the associated ordering.

Similar to the application specification, customized XML schemes are used for the architecture and mapping specifications.

2.3 Scalability

In order to facilitate the definition of larger structures, the components of a system in DOL environment can be specified in a scalable manner. For this purpose, an *iterator* element has been introduced in the DOL XML scheme, which allows to replicate not only a single component, e.g. processes, software channels, processors, but also entire structures in one or more dimensions. In

the example illustrated in Fig. 2, for instance, iterator elements, e.g. Line 8-15 and 22-28, are used to create a parameterized number of processes between the producer and the consumer process, and their corresponding interconnects.

In the case of a set of iterated processes, all processes share the same source code. Processes can, however, read their iteration index which can be used to adapt the behavior of a process depending on its location in the process network.

By using this parameterized constructions (i.e. with iterators), a system of DOL can be easily scale.

3 MPARM Platform

The MPARM [1] is a reconfigurable cycle-accurate simulator for a distributed memory architecture. The architecture of the MPARM is composed of a configurable number of identical ARM tiles connected by a shared bus. In addition, a shared memory and an optional *external* DMA device are attached to the shared bus. A block diagram of the MPARM architecture is shown in Fig. 4.

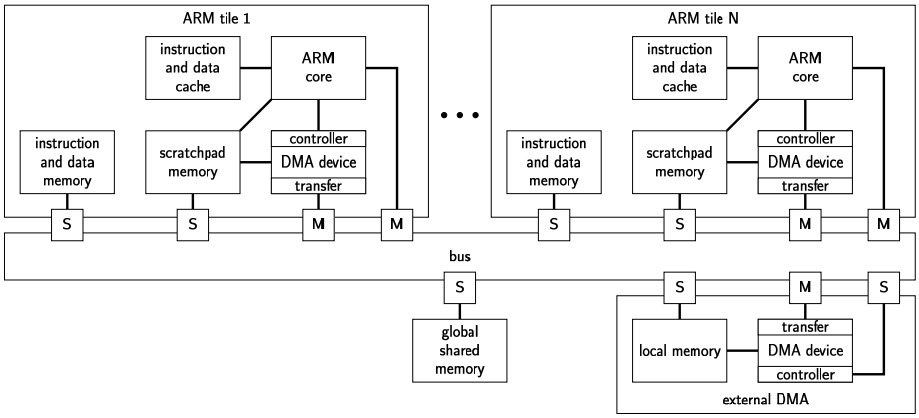


Fig. 4: Block diagram of MPARM architecture. The blocks labeled with "M" and "S", denote master and slave ports on the bus, respectively.

Each ARM tile contains an ARM core, a local instruction and data memory, a cache, a scratchpad memory, and an *internal* direct memory access (DMA) device. To access the local instruction and data memory, the ARM core needs to go via the shared bus. On the contrary, the cache and the scratchpad memory are tightly coupled within the ARM tile, the access of which does not trigger any shared bus traffic.

The MPARM platform supports the RTEMS (real-time executive for multiprocessor systems) [9] operating system, which offers multi-tasking and message passing between tasks. The scheduling is priority-based which supports both

preemption and non-preemption as well as time-slicing of tasks with the same priority.

Direct Memory Access The so-called internal DMA devices found on each tile can be used to transfer data from or to the (scratchpad) memory via a slave port of the shared bus. DMA transfers are set up by the ARM processor and do not require any further coordination of the ARM core while executing the data transfer.

In Fig. 4, one can see that the DMA device is split up into a control and a transfer module. The DMA control module has an internal memory where it is able to store so-called *objects*. Objects are entities which track blocks of data before, during, and after their transfer from one (scratchpad) memory to the other. To transfer data, an object needs to be acquired, filled with the required transfer information (including the size of the transfer, the source address, and the target address), and en-queued to the list of pending transfers. Every time the DMA transfer module is free, and this queue contains an entry, the DMA control module asks the DMA transfer module to start a new transaction. The object queue has a FIFO scheduling policy. Details about the DMA can be found in [8]

4 Runtime Environment

To execute a DOL application on the MPARM simulator, the processes and channels, i.e. the computation and the communication, of the process network need to be carefully designed to preserve properties of the model of computation. Applications programmed using the DOL adhere to the process network model of computation which enables a purely data-driven execution. A global clock or time base is thus not needed for scheduling of the application. Therefore, the architecture-dependent runtime environment only needs to provide the multi-processing for each core as well as the software channel implementation. In this work, we implement the multi-processing by means of the multi-tasking provided by the RTEMS operating system and software channel communication using scratchpad FIFOs. The software structure is shown in Fig. 5.

Specifically, to actually execute the DOL process network, the `fire` function of each DOL process is wrapped within a RTEMS task which can be scheduled by the RTEMS operating system. A wrapper task controls the actual execution of a process of the DOL process network. It repeatedly calls the `fire` function until the `DOL_detach` is met. The `DOL_read` and `DOL_write` API are refined as scratchpad FIFO read and write routines, respectively. Depending on the specification, a concrete FIFO implementation is chosen at compile time.

A software synthesis tool has been developed to generate the source code of the runtime environment from the standard DOL specifications, i.e., application, architecture, and mapping specifications. As shown in Fig. 6, the generated source code includes the RTEMS task wrappers for each process, the `main` file, and the `Makefile`. The generated code together with the RTEMS kernel is compiled and linked into a single binary which will be loaded into every tile. Note

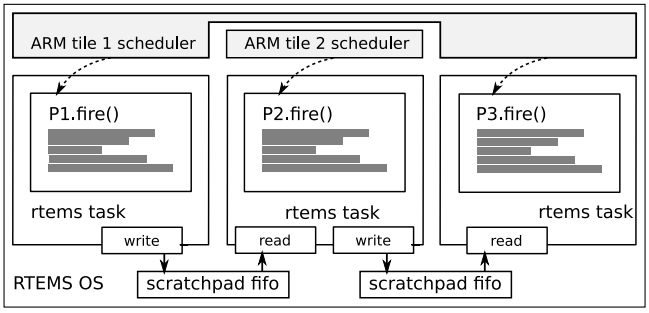


Fig. 5: The structural view of the DOL application running on top of the RTEMS OS.

that the different tiles are basically operating independently although they share the same binary. Nevertheless, the tiles will behave differently because the instantiation and initialization of DOL processes depend on the tile index in the generated `main` file.

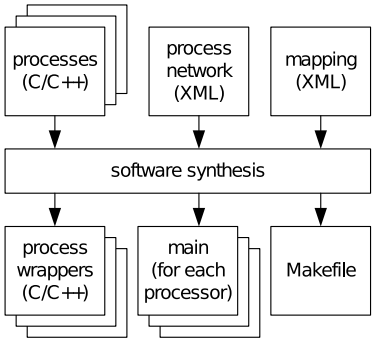


Fig. 6: Software synthesis for generating the source code for the runtime environment.

During the initialization of the system, one RTEMS task will be instantiated for each process in order to initialize the connected FIFO as well as the wrapper task. These initialization tasks have a higher priority than all wrapper tasks such that the process network can be guaranteed to start only after the initialization phase. After the initialization phase, this task will be deleted.

4.1 Inter-process Communication

In the runtime environment, four different mechanisms are provided as alternatives for FIFO communication of the DOL process network. They are differing with respective to the location of the FIFO buffer, as shown in Fig. 7. Besides the classical message passing approach where the FIFO buffer is allocated to

the shared memory as shown in Fig. 7(a), we present three other alternatives, i.e., local scratchpad FIFOs. In Fig. 7(b) and Fig. 7(c), the FIFO buffer is allocated into the local scratchpad memory of the sender and receiver processors, respectively. In Fig. 7(d), the FIFO buffer is split into two parts allocated to the local scratchpad of both the sender and receiver processors. Therefore, both the read and write FIFO routines copy data within an ARM tile. To transfer data between scratchpads of different tiles, a dedicated ARM tile is assigned to coordinate the transmission.

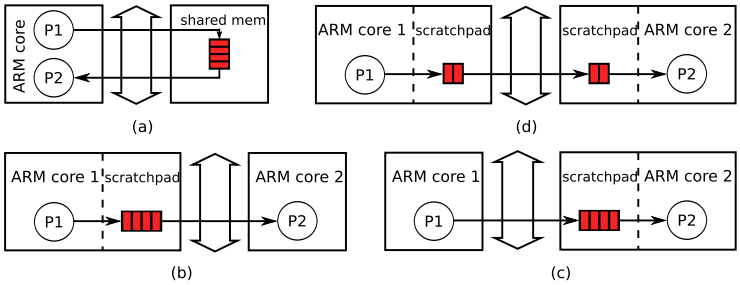
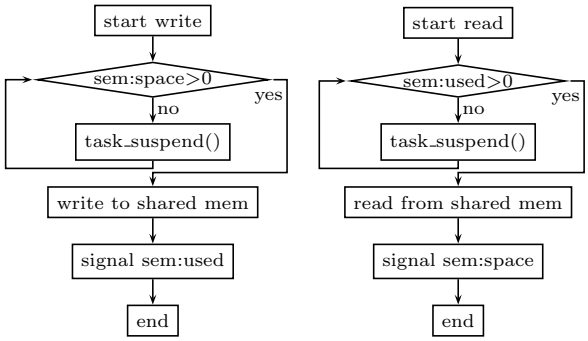


Fig. 7: Four different communication mechanisms: (a) Channel buffer in shared memory. (b) Channel buffer in sender processor. (c) Channel buffer in receiver processor. (d) Channel buffer is split into two, distributed in both sender and receiver processors.

The FIFO communication is packet-based. Each write/read operation transfers N packets where $N > 0$. The size of a packet and the length of a FIFO is set at compile time in the application specification.

A pair of semaphores is used for the access control of a FIFO, namely `sem:used` and `sem:space`. The `sem:space` semaphore located in the local scratchpad of the sender processor indicates how many packets a FIFO can still contain. Initially, it is set to the size of the software channel. Upon writing or reading a packet, it decreases and increases by one, respectively. When it is 0, the write routine will stall until it becomes larger than 0 again. On the contrary, the `sem:used` semaphore located at the local scratchpad of the receiver processor is used for the blocking read operation. Initially, it is set to 0. Upon writing or reading a packet, it increases and decreases by one, respectively. The flowcharts of the write/read routines for the four mechanisms are described below.

Shared memory FIFO. The flowcharts for the write and read routines of this approach are shown in Fig. 8. Since the shared memory is connected to a slave port of the shared bus, both the write and read routines will trigger data transfers on the shared bus. Therefore, they have to compete for the shared bus with other data-copy routines issued by other processes. Furthermore, the remote semaphore notification at the end of a data-copy operation will introduce additional shared bus traffic. In this case, the shared bus is the potential bottleneck of the system.



(a) Write operation (b) Read operation

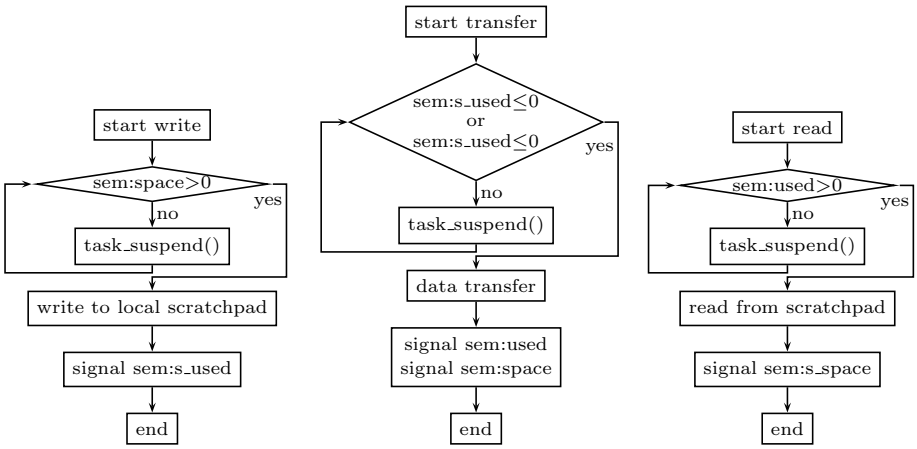
Fig. 8: The flowcharts of writing and reading one packet for the shared memory FIFO.

Local scratchpad FIFOs. The goal of using local scratchpad memory is to reduce the shared bus traffic. Two implementations are presented in which the FIFO buffer can be allocated to the local scratchpad memory of either the sender or receiver processors. In the former case, only the read routine issued by the receiver triggers data transfers on the shared bus. The write routine copies data from the cache to the scratchpad memory within one tile. In the latter case, the write routine copies data from local cache to the remote scratchpad memory in the receiver tile. The read routine is thereby a local behavior within a tile. The flowcharts of the write and read routines are similar to Fig. 8. We omit them here due to the similarity. By means of the local scratchpad FIFOs, half of the shared bus traffic for the data transfer is avoided, compared to the shared bus FIFO. Note that the remote semaphore notification still triggers shared bus traffic.

Split FIFO. Although the previous two local scratchpad FIFOs avoid half of the shared bus traffic, the traffic itself is not manageable in the sense that the shared bus arbiter schedules the traffic at the tile level. To coordinate the traffic at the software channel level, we introduce the third scratchpad FIFO mechanism. The FIFO buffer is split into two halves, located at the scratchpad memory of both the sender and receiver processors. The write and read routines now copy data to and from local scratchpads, respectively. A control process located at an isolated tile is responsible for coordinating the data transfer between these split FIFO buffers located in two different scratchpad memories. In this manner, the shared bus traffic is fully decoupled from the write and read routines. The write and read routines return immediately after data are copied to and from local scratchpad memory, respectively. In this manner, although the amount of traffic via the shared bus is still the same, the traffic itself can be orchestrated by the control process which can provide different kinds of scheduling polices at the software channel level.

The flowcharts for transmitting one packet for this FIFO are shown in Fig. 9. Beside the pair of semaphores `sem:space` and `sem:used`, two new semaphores,

namely `sem:s_used` and `sem:s_space`, are introduced for flow control of the split FIFO, indicating data availability in the sender buffer and the space in the receiver buffer. They are located in semaphore memory of the tile running the control process. Only when there is at least one data packet in the sender buffer (`sem:s_used` > 0) and there is space in the receiver buffer (`sem:s_space` > 0), the control process will conduct a packet transfer. To orchestrate the data transfer for different FIFOs, a packet-based round-robin policy is used to schedule among all software channels. Other kinds of priority-based or time-slicing based scheduling can be implemented without any difficulty.



(a) Write operation (b) Data transfer (c) Read operation

Fig. 9: The flow-charts of writing and reading one packet for the split scratchpad FIFO.

Using DMA. To actually transfer the data from/to a FIFO buffer, two approaches are provided. The first approach is direct memory copy by the ARM core. Since the memory address of the shared memory and all scratchpad memories are globally visible, each ARM core can directly access the shared memory as well as remote scratchpad memories located at other tiles. The second approach is the DMA transfer by using the internal DMA device of each tile. The DMA device can transfer data:

- from the cache to the local scratchpad within a tile, and vice verse;
- from local scratchpad memory to remote scratchpad memory located in another tile, and vice verse;
- from local cache to remote scratchpad memory.

The cases (a), (b), and (c) in Fig. 7 support both approaches. In the case of (d), since a dedicated ARM tile is assigned to coordinate the data transfer

and no process of the process network is attached, the DMA approach has no advantage and is thereby not supported. In Fig. 10, the flowchart of the write routine is depicted.

Note in the current implementation, write and read routines perform a busy wait to check whether the DMA operation finishes or not. The write and read routines themselves will not give up the CPU control unless a higher priority process becomes ready and preempts the current one.

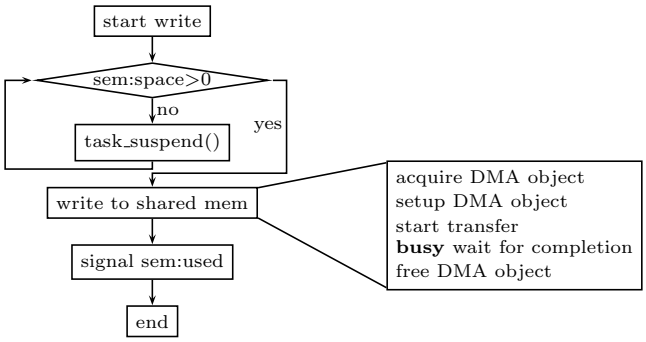


Fig. 10: Flowchart of the implementations of read operation using the DMA device.

5 Experiments

In principle, an application that can run through the DOL functional simulation can execute (correctly) in the MPARM. We conduct two case studies: a pipelined MPEG-2 decoder and a SDF [7] like MJPEG decode. The MPARM simulations execute on a 2 *GHz* AMD Athlon 2800+ Linux machine.

5.1 MPEG-2 Decoder

We present an MPEG-2 decoder in this section. The simulated MPEG-2 clip has a frame-rate of 25 *fps*, bit-rate of 8 *Mbps*, and resolution of 704×576 pixels. For our experiments, we use a video clip with a duration of just 4 frames. We chose the shared memory FIFO for software channel communication. The the frequency of the ARM cores and shared bus is set to 200 *MHz*.

We measure two different timings:

1. *Simulation runtime* (`Sim.`), corresponding to the time elapsed from the start of a simulation until it completes. This indicates how long the simulation needs (wall-clock time) .
2. *Estimated execution time* (`Est.`), indicating how fast an application executes on a target architecture, i.e., the time of decoding an the MPEG-2 video clip on the MPARM.

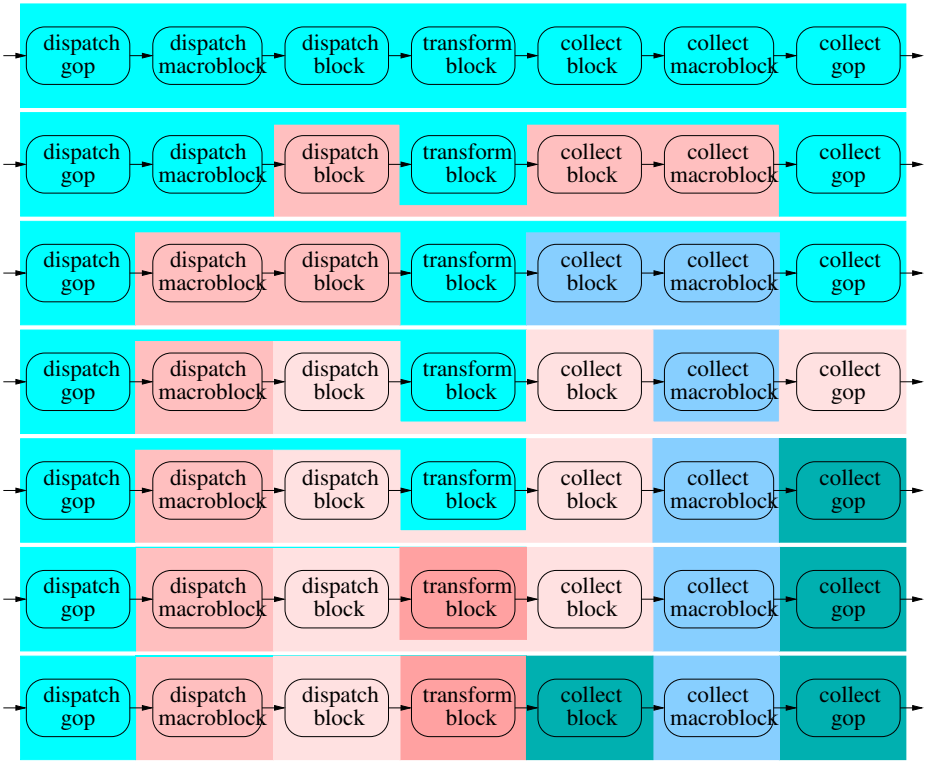


Fig. 11: 7 different mappings of the pipelined MPEG-2 decoder. From top to bottom, the process network is mapped to 1 until 7 different processors.

Fig.12 illustrates the simulation runtime and estimated execution time, respectively, for the mappings in Fig. 11. As the figure shown, a simulation of decoding 4 MPEG-2 frames takes at least hours. The simulation runtime increases linearly as more cores are simulated. We can also find out that the estimated execution time for all cases is far too slow with respect to the real-time requirement, i.e., 25 fps. The extremely slow execution time is due to:

- All data transfers between processes use FIFO communication, resulting in a triple buffering for any communication data – data in sender processor cache, FIFO buffer, and receiver processor cache. This is a typical problem of a process network system. Using a more efficient FIFO mechanism [**?**,**?**] could significantly reduce the runtime overhead.
- The major artifact of the runtime overhead comes from the packet-based FIFO communication. In our pipelined MPEG-2 decoder, the size of data transferred via a software channel is variable during the execution. As shown in Fig. 11, there is only one software channel between two processes, transmitting both control signal and data packets. However, the packet size for

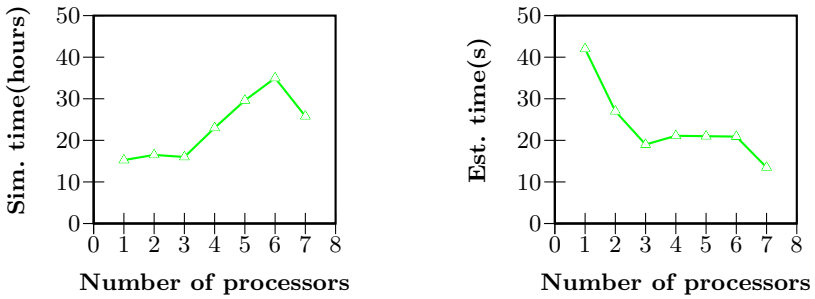


Fig. 12: Simulation runtime (left) and estimated execution time (right) for 7 different systems, with 1 up to 7 processors.

each FIFO has to be fixed at the compile time. Therefore, the greatest common factor of the sizes of all possible data will transfer through a software channel is used as the packet size of the corresponding FIFO, which in this case is 32 *bits*. This small packet size results in an inefficient communication that incurs a huge amount of packet transmission and remote semaphore notifications. Note this fact also slows down the simulation runtime.

5.2 MJPEG Decoder

In this section, we present another case study, i.e., a gray-scale MJPEG decoder. we show the impact of different FIFO implementations for a given mapping. The process network and the mapping are shown in Fig. 13. Comparing to the previous case, each software channel in this process network transfers data of fixed length which is known at compile time. Therefore, the packet size of a chosen FIFO is set to the actual length of the data, not 32 bits. We exploit the performance of different frequency settings. The video clip used consists of 31 frames with 320×240 pixels per frame.

The simulation results are shown in Table 1. Given the same frequency setting, the shared memory FIFO cases have the worst performance due to the double accesses of the shared bus. The receiver scratchpad FIFO performs best for all cases. The reason could be that the *push* semantic of the receiver scratchpad FIFO can better utilize the shared bus for this system. In the case of sender scratchpad FIFO, the request for data transfer is invoked when the receiver process asks for data. The actual transfer will be possibly deferred if the shared bus is currently busy. However, further investigation is needed to identify the potential of each FIFO mechanism.

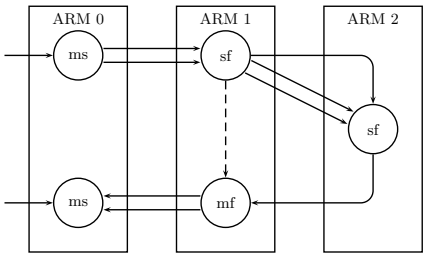


Fig. 13: The 3-ARM MJPEG system.

Table 1: Estimated runtimes (second) for different frequency combinations of the ARM processors and the shared bus.

	p200,b100	p200,b200	p200,b400
shared mem.	7.807	5.252	3.526
sender scratchpad	7.758	5.229	3.522
receiver scratchpad	7.439	5.171	3.376
split scratchpad	7.751	5.245	3.385

Another observation is that the execution time still does not fulfill the real-time requirement, i.e., 25 fps. However, there is still large space to improve the execution runtime. By a deeper probe of the system, we found out the workload of the three processor cores are not balanced. The workloads imposed on `ARM 1` and `ARM 2` are 1% and 50% of that on `ARM 3`. Large speedup can be expected if the process network is remapped in more load-balanced manner.

The third observation is the shared bus is indeed the bottleneck. As the table shown, varying the frequency of the shared bus significantly changes the system performance.

6 Conclusion

We have successfully coupled the DOL framework with the MPARM cycle-accurate simulator. We developed a software synthesis procedure to automatically generate the source code of the runtime environment for the MPARM in a correct-by-construction manner. The efforts to inspect new mappings are thereby minimal, and several mapping possibilities can easily be inspected.

References

1. L. Benini, D. Bertozzi, A. Bogliolo, F. Menichelli, and M. Olivieri. MPARM: Exploring the Multi-Processor SoC Design Space with SystemC. *VLSI Signal Processing Systems*, 41(2):169–182, Sept. 2005.
2. DOL distribution. `ToolDescription.pdf, cCodingStyle.pdf, pnExamples.pdf`.

3. W. Haid, L. Schor, K. Huang, I. Bacivarov, and L. Thiele. Efficient Execution of Kahn Process Networks on Multi-Processor Systems Using Protothreads and Windowed FIFOs. In *Proc. IEEE Workshop on Embedded Systems for Real-Time Multimedia (ESTIMedia)*, pages 35–44, Grenoble, France, Oct. 2009.
4. K. Huang, D. Grünert, and L. Thiele. Windowed FIFOs for FPGA-based Multiprocessor Systems. In *Proc. Int'l. Conf. on Application-Specific Systems, Architectures and Processors (ASAP)*, pages 36–41, Montreal, Canada, July 2007.
5. G. Kahn. The Semantics of a Simple Language for Parallel Programming. In *Proc. IFIP Congress*, North Holland Publishing Co, 1974.
6. K. Keutzer, S. Malik, A. R. Newton, J. M. Rabaey, and A. Sangiovanni-Vincentelli. System-Level design: Orthogonalization of Concerns and Platform-Based Design. *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, 19(12):1523–1543, Dec. 2000.
7. E. A. Lee and D. G. Messerschmitt. Synchronous Data Flow. *Proceedings of the IEEE*, 75(9):1235–1245, Sept. 1987.
8. MPARM distribution. `readme_dma.txt`.
9. RTEMS Steering Committee. RTEMS Home Page. `http://www.rtems.com/`.
10. L. Thiele, I. Bacivarov, W. Haid, and K. Huang. Mapping Applications to Tiled Multiprocessor Embedded Systems. In *Proc. Int'l Conf. on Application of Concurrency to System Design (ACSD)*, pages 29–40, Bratislava, Slovak Republic, July 2007.