INCREASING THE LEVEL OF ABSTRACTION IN FPGA-BASED DESIGNS

Martin Danek, Jiri Kadlec, Roman Bartosinski, Lukas Kohout

Department of Signal Processing,
Institute of Information Theory and Automation, UTIA AV CR
Pod Vodarenskou vezi 4, Praha 8, Czech Republic
email: {danek,kadlec,bartosr,kohoutl}@utia.cas.cz

ABSTRACT

Traditional design techniques for FPGAs are based on using hardware description languages, with functional and post-place-and-route simulation as a means to check design correctness and remove detected errors. With large complexity of things to be designed it is necessary to introduce new design approaches that will increase the level of abstraction while maintaining the necessary efficiency of a computation performed in hardware that we are used to today. This paper presents one such methodology that builds upon existing research in multithreading, object composability and encapsulation, partial runtime reconfiguration, and self adaptation. The methodology is based on currently available FPGA design tools. The efficiency of the methodology is evaluated on basic vector and matrix operations.

1. INTRODUCTION

In recent years we have seen the area of FPGA implementations broadening from simple, single-purpose applications "hard-wired" in a single configuration bitstream towards complex, variable, dynamically changing applications based either on a reprogrammable sequential von Neumann machine, or reconfigurable dataflow machine that uses partial runtime reconfiguration to change the circuit function.

Due to the advance of technology at present we can perceive partial runtime reconfiguration of FPGAs as another architectural feature ready to be used in real-world designs; both the FPGA silicon and design tools are available today. On the other hand, partial runtime reconfiguration itself increases the complexity of the design process, resulting in higher design costs. It is even expensive in terms of execution performance due to the relatively long reconfiguration latency in current FPGA chips; this is caused mainly by the fine grain nature of elements that must be described in the partial reconfiguration bitstream, and by a very limited architectural support of partial runtime reconfiguration in today's FPGAs, which are designed to work primarily with just a single context.

An alternative approach to changing circuit function during runtime is to use simple CPUs that control the function of data processing blocks; this approach has already been described in the literature [1], [2], [3], [4], [5], [6].

The challenge in this approach is to align the slow, sequential nature of CPU data processing (here we mean a very simple basic processor without advanced architectural features) with the parallel nature of FPGA data processing, often using pipelined function blocks with different pipeline latencies.

The advantage of using simple CPUs to change the circuit function is twofold: First, the CPU program usually works with a higher granularity than the FPGA configuration mechanism. Second, the human mind has been trained to think in a sequential rather than parallel manner. These two facts mean that the circuit function can be changed faster by CPU reprogramming than by direct FPGA reconfiguration (by downloading programs that have a smaller memory footprint compared to the corresponding configuration bitstreams), and that it should be easier for a designer to design programs for a CPU than to directly program the hardware in HDL.

To make the mixed CPU and dataflow approach usable in practical designs it is necessary to find a suitable organization of a hardware design in general so that we can take advantage of using the sequential CPU approach while not killing the data parallel processing inherent to FPGAs. This paper proposes one such organization implemented with currently available FPGA design tools.

The paper is organized as follows: Section 2 overviews the developments in mapping dataflow computations to CPUs. Section 3 analyzes gains offered by using simple CPUs in hardware design, and it describes a sample design organization based on simple CPUs and programmable datapaths. Section 4 presents performance evaluation, and Section 5 concludes the paper.

This research was partially supported by the European Commission under project Aether FP6-IST-027611.

978-1-4244-1961-6/08/$25.00 ©2008 IEEE.

2. DATAFLOW COMPUTING AND CPUS

Architectural features of modern CPUs that try to decrease execution stalls, usually due to operands not being ready in registers, include several variants of multithreading, where several computation threads without data dependencies are executed independently by the instruction issue (do not confuse with OS-level threads that usually represent much bigger parts of computation).

CPU multithreading [7] increases processor performance by executing the computation in several independent dataflow parts, preferably without any mutual data dependencies. In this approach an efficient computation is equivalent to a full usage of the processing pipeline. The major events that cause pipeline stall are accesses to external memories (represented mainly by cache misses) and misprediction of branches. The reasoning is that when a large number of threads are ready for processing, with zero latency on context switch, no pipeline stall will ever occur.

Since the above example is idealized, we can expect that in practice there will always be a certain percentage of pipeline stalls due to a limited number of threads ready for execution. **Microthreading** [8] is a technique that tries to increase the number of processing threads "by definition" - the compiler identifies program locations where context switch may occur, and it inserts explicit context switch instructions there (the context is defined only by a distinct program counter for each thread, CPU registers and memory space is shared).

From the point of view of a hardware designer the above approaches are not satisfactory to be used in embedded applications with FPGAs, since such applications contain a certain level of parallelism, where **dependencies can be determined statically** [6]. In this context multithreading does not guarantee efficient usage of computing resources in data streaming applications; these techniques cannot totally eliminate pipeline stalls, and the silicon overhead in terms of the necessary CPU function blocks is too high for limited-purpose applications [9]. The lower functional density of the CPU silicon can be justified only for general-purpose applications that are not usually implemented in FPGAs.

3. DESIGNING WITH SIMPLE CPUS

On the other hand, it is beneficial to use simple CPUs in the role of programmable finite state machines to control configurable datapaths, and to eliminate execution stalls by a definition of an explicit computation schedule.

We propose a design organization in two layers joined in a basic computing element: the lower level consists of **dataflow units (DFUs)** with a limited number of configurations implemented through multiplexers in their data paths. In the following text we will consider a general case of DFUs with pipelined floating-point operations with different latencies.

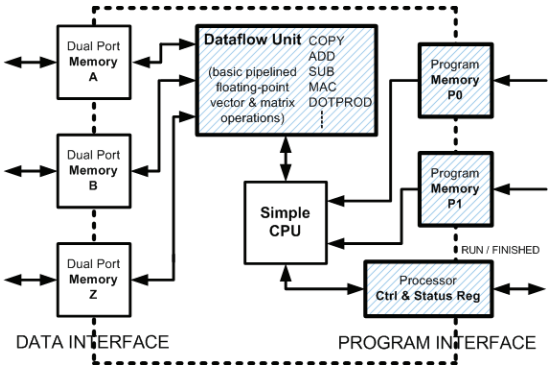

Fig. 1. The basic computing element (BCE): a dataflow unit with a simple control CPU. The dataflow unit can be reconfigured on the fly to suit different application domains (e.g. fixed-point vs. floating-point).

In general the computation of pipelined dataflow units can be split in three steps: pipeline initialization, data processing and wind-up. The ratio between the number of clock cycles of the data processing step and the initialization plus wind-up steps determines the efficiency of the pipeline usage, implying the data processing step should dominate the computation.

The DFUs are meant to implement basic mathematical operations, i.e. basic building blocks of the target computation. To increase the computation efficiency, when considering pipelined floating-point operations, the dataflow units should operate on batches of data. The advantage besides partly eliminating the initialization and wind-up is the possibility to mask communication latencies as we will see later.

The higher level consists of **simple CPUs (sCPUs)**, such as the PicoBlaze [10], where each sCPU controls one or more dataflow units. The role of these sCPUs is to encapsulate implementation details of the dataflow units, provide a design interface on a higher level of abstraction (atomic operations on data batches rather than dealing with individual clock cycles), provide an ability to specify sequences of operations performed on one batch of data (making use of data locality in the computation), and from the hardware point of view to eliminate the expensive wide multiplexers that would be necessary in pure hardware design with the full variability built directly in hardware.

The encapsulation of dataflow units enables to use DFUs of different complexity in terms of implemented operations with different area requirements since it is assumed that more specialized operations implemented in complex DFUs can be implemented as a sequence of generic operations in simple DFUs. Each case is represented by an sCPU program: one program will contain only a single instruction, while the other will contain an equivalent sequence of instructions. A typical example is a vector dot product operation that can be

calculated as an explicit sequence of scalar multiplications and additions.

3.1. Dataflow Unit Operations

The set of operations supported in the discussed dataflow unit was derived to speed up common DSP algorithms, based on vector or matrix operations. We have focused on the elementary problem of vector dot product and its use in matrix multiplication.

Matrix multiplication is defined as follows:

$$Z_{[i,j]} = A_{[i,k]}.B_{[k,j]} \quad (1a)$$

$$z_{i,j} = \sum_{r=1}^{k} a_{i,r}.b_{r,j} \quad (1b)$$

Equation (1b) can be computed as a sequence of basic multiply and add operations, or it can be mapped directly to a specific structure. This structure can be optimized either for individual dot-product operations, computing only one element of the resulting matrix at a time, ideal for long vectors, or it can be optimized for computing several elements of the resulting matrix at once, ideal for cases where several short vectors can be processed in one batch, such as when multiplying matrices with a small dimension k, see Eqs. (1a), (1b). In the discussed dataflow unit the former case is implemented by the DOTPROD operation, and the latter case by the MAC operation.

Important areas that use multiply-accumulate as the fundamental composite operator are mathematics - discrete convolution in Eq.(2a), used also in audio DSP - FIR, and image processing - pattern matching in Eq.(2b) [11].

$$(f * g)(n) = \sum_{n} f(n)g(m-n) \quad (2a)$$

$$R_e(s,t) = \sum_{m=0}^{M} \sum_{n=0}^{N} [I_1^{i,j}(m,n).I_2^{i,j}(m-s,n-t)] \quad (2b)$$

The discussed DFU uses the floating-point representation; the main reason is its flexibility compared to the fixed-point representation, i.e. no overflow effects and a wide dynamic range.

The DFU calculates results that are bit-exact identical to the results generated by an equivalent sequence of operations in the MicroBlaze configured with the standard hardware floating-point unit (note that in the MicroBlaze different results are obtained when floating-point operations are computed in the software floating-point library and in the hardware floating point unit that does not support denormalization).

3.2. Basic computing element

Figure 1 shows a block diagram of the described **basic computing element (BCE)**. Since the details of the internal organization have been described in [12], here we briefly mention just the key facts.

The BCE is generic; this means the user can specify the basic functions calculated in the dataflow unit to suit his application domain (fixed-point vs. floating-point operations, arbitrary precision, different libraries of basic operations e.g. for image or audio processing). In the discussed case the dataflow unit implements vector and matrix calculations in a single-precision floating-point representation, the operations used in the DFU unit are pipelined, not necessarily with the same pipeline latencies.

An example of a variable DFU discussed in the remaining text is shown in Figure 2. The following discussion considers a unit with two single-precision floating-point operations ($+$, $*$) generated in the Xilinx Core Generator. The pipeline latency for the $+$ operation is 3 clock cycles and for $*$ 4 clock cycles; the latencies have been selected to achieve system frequency of 100MHz, which is the limit dictated by the QinetiQ floating-point IP cores and the PicoBlaze used as the simple CPU to control the dataflow unit; this frequency is also compatible with a design featuring the MicroBlaze configured with a single-precision floating-point unit (a MicroBlaze option in Xilinx EDK) running at 100MHz.

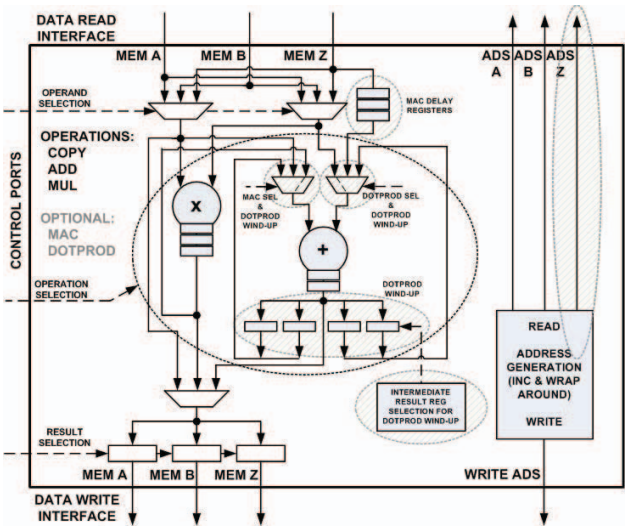

Fig. 2. A configurable dataflow unit with pipelined floating-point operations for vector and matrix processing.

In the following text we analyze the DFU in three flavours that differ in the complexity of the batch operations it supports. In the basic version *BCE_BASIC* the unit supports data transfers between any of the three data memories and batched vector-like addition and multiplication. A more pow-

Table 1. Resource requirements (XC4VSX35).

Feature	Slices	BRAM16	DSP48
MicroBlaze (MB)	1552	32	3
STD MB HW FPU	692	0	4
BCE_BASIC	1632	8	4
BCE_MAC	1904	8	4
BCE_MAC_DOTPROD	2176	8	4

erful version *BCE_MAC* implements an additional multiply-accumulate (MAC) operation that adds a multiple of two numbers stored in two of the data memories to a number stored in the third data memory, storing the result there. The most powerful version *BCE_MAC_DOTPROD* implements an additional dot product (DOTPROD) operation, used as the basic building block in many DSP calculations. The hatched parts in Fig. 2 correspond to design features implemented only in the MAC and DOTPROD versions of the DFU. The resource requirements of each of the three versions and their comparison to the MicroBlaze and its hardware floating-point unit are listed in Table 1.

The function of the computing element can be changed either via changing the contents of any of the program memories $P0$ or $P1$ (the memories can be swapped in one clock cycle), or by using partial runtime reconfiguration of the dataflow unit. Table 2 shows representative times for both cases; we assume that each program memory consists of 256 locations to be defined, and that the reconfigurable area taken by the whole basic computing element in its three variants is 24 by 48 CLBs, 30 by 64 CLBs, and 28 by 80 CLBs respectively, the area being utilized completely. In the limit case it is necessary to reload the whole program memory or to reconfigure the whole reconfigurable area to change the computation. It can be clearly seen that software-emulated reconfiguration via program memory change is about **400 times** faster.

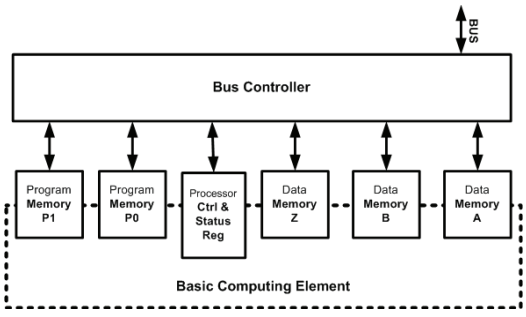

Fig. 3. User-level view of the basic computing element.

We assume that the partial runtime reconfiguration will be used only when redefining the application domain, and that in the majority of cases the function of the computing element will be modified by reloading the program memory.

Another important feature is the organization of the data memories A, B, and Z. In our sample case each memory is 1024 data words long, organized in four 256-word parts. This enables to mask communication latencies, provided the computation performed on the local data take longer than is needed to read back previous results and load a new data batch.

The basic computing element has been designed so as to hide unnecessary implementation details from the designer. The system perceives the whole BCE as an intelligent memory without any direct access to the BCE internals (see Fig. 3). The computation is started by a write to the BCE processor control register after all data have been transfered to the BCE data memories, and synchronization is performed by reading the BCE processor status register (an option is to interrupt the host CPU, e.g. the MicroBlaze). The BCE application programming interface (BCE API), currently implemented as C function calls, is identical for all the described flavours of the dataflow unit. In addition to the BCE organization discussed so far the BCE API also covers SIMD-like variants that we have also developed, where the BCE contains several dataflow units controlled by one sCPU; in such cases the designer uses additional function calls to transfer data to additional data memories that are served by the additional dataflow units, and the interaction with the simple CPU remains identical.

The protocol between the simple CPU and the dataflow unit consists of these steps: **1.** the sCPU requests the dataflow unit to send an identification of its capabilities (a vector of bit values set to 1 or 0 according to available features), **2.** the DFU answers the ID request, **3.** the sCPU commands the DFU to perform an operation on a batch of data values (in our setting a typical batch length is 1-256 data values) and monitors its completion (in the meantime the sCPU can perform other tasks such as data transfers between an external memory and the local data memories, in the discussed example implemented through a host CPU), **4.** the DFU acknowledges completion and halts.

3.3. User-level interface and system integration

The use of dual port memories makes it possible to hide implementation details beneath a standard memory access. User applications simply perceive the computing element as an intelligent memory that can perform operations on its data. The interface memories also isolate the internal architecture of the element from the comunication network; the computing elements can be connected either directly to a processor bus as shown in Figure 3 (this case will be discussed in the following text), or they can be connected in a specialized network topology constructed with respect to the flow of data in the application, as offered by the recently

Table 2. Change of function: partial runtime reconfiguration vs. reprogramming (XC4VSX35).

Variant	Slices	BRAM16	DSP48	Bitstream size	Duration		
					Theoretical	Measured	
						Standalone	Petalinux
	[1]	[1]	[1]	[B]	[μs]	[μs]	[μs]
BCE_BASIC	1632	8	4	151184	34831.81	34832.32	35168.00
BCE_MAC	1904	8	4	152556	35148.73	35149.24	35484.00
BCE_MAC_DOTPROD	2176	8	4	209960	48372.44	48372.95	48813.00
PB program load	Maximal values			4096	92.40	92.91	344.00

researched component approaches [13], [14].

4. PERFORMANCE EVALUATION

We have evaluated the core performance of the discussed basic computing element and the influence of latencies of data transfers between an external DDR memory and the dual-port data BlockRAMs connected to a BCE. The experimental system consisted of one BCE connected to the MicroBlaze CPU through the FSL interface (in general there can be as many as eight BCEs connected via the FSL to the MicroBlaze). The performance of the BCE has been evaluated as a speedup of basic vector and matrix operations with respect to identical operations executed in the similarly flexible MicroBlaze configured with the hardware floating-point unit; both the BCE and MicroBlaze operations are sequenced in the same way to achieve bit-exact results.

The floating-point add and multiply operations used in the dataflow unit have been generated in the Xilinx Core Generator. The latencies of the operations ($lat_{add} = 3$, $lat_{mul} = 4$) have been selected to be compatible with the operating frequency of a design with the MicroBlaze configured with the hardware floating-point unit.

The design has been implemented in the Xilinx ML402 starter kit with XC4VSX35-10, the whole design operating at 100MHz. We used the Xilinx System Generator, ISE and EDK in version 9.1; the main reason for using this version is the support of virtual platform simulation that we use for debugging the system, which is missing in EDK version 9.2. The MicroBlaze is v6.0b, with both Icache and Dcache configured to 16kB, running Petalinux kernel version 2.6 with 100Mbps Ethernet.

4.1. Sample problem

The selected sample problem is a parameterizable matrix multiplication, selected for its possibilities to test and evaluate all the implemented BCE operations.

$$Z_{[10,10]} = A_{[10,j]}.B_{[j,10]} \qquad (3a)$$
$$1 \leq j \leq 24$$

The dimensions of the matrices are dictated by the size of the data memories (each memory is divided into four parts, each containing 256 words). The dimensions of the result matrix are fixed at 10, while the inner dimensions of the input matrices change from 1 to 24.

4.2. Communication

Our experiments performed ten consecutive matrix multiplications using the following three basic communication schemes:

- **No communication:** Input data are transfered from the external DDR memory to the data BRAMs only once before all computation steps. Output data are transfered back to the DDR memory after all computation steps have finished.
- **Sequential communication:** Input data are transfered from the DDR memory to the data BRAMs before each computation. Output data are transfered back to the DDR memory after each computation step.
- **Masked communication:** Input and output data transfers overlap with the computation steps.

4.3. Results

Figure 4 shows the BCE performance for a system with a single basic computing element with a single dataflow unit.

In both figures the upper plot shows an upper bound of possible acceleration of the basic computing element since it uses data in place. The middle plot shows a more realistic lower bound of possible acceleration since computation steps are interleaved with communication steps (i.e. computation does not happen during communication). The lower plot shows the most realistic case with an optimized communication that overlaps with computation.

Each plot in Fig. 4 shows the computation speedup for all three versions of the dataflow unit as described in Table 1. These values can be used to trade off the available computing performance and the resources needed to implement

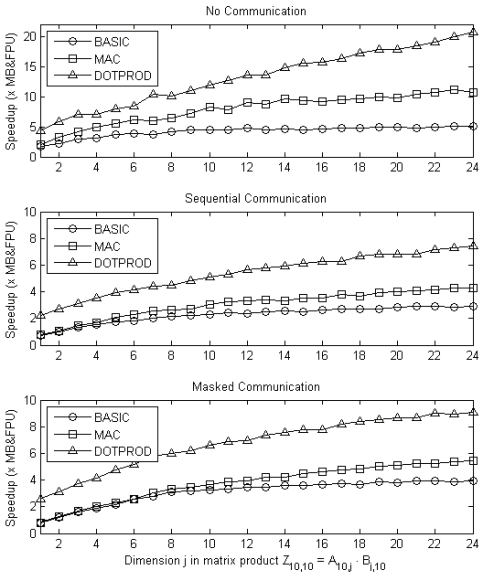

Fig. 4. Speedup of the BCE with a single dataflow unit over the MicroBlaze with the HW FPU computing the sample problem.

the operations. The application program commanding the BCEs, presently running in the MicroBlaze, remains identical for all three versions of the DFU since the BCE performs internal self-adaptation to different DFU configuration, self-adaptation managed by the simple CPU.

It can be seen that for the three versions of the dataflow unit (*BASIC*, *MAC*, *MAC_DOTPROD*) the realistic speedup with respect to the MicroBlaze with HW FPU is between 4 and 8 for the biggest matrices. The difference in speedups between the top plot and the middle and bottom plots is given by the data transfer latencies due to the FSL interfaces and the MicroBlaze running the Petalinux system processes.

Please note that in a usual setting only a minimal amount of data will be transfered between the external DDR memory and the internal data memories; this is given by the nature of signal processing algorithms used in embedded systems. The presented platform is intended for embedded rather than high-performance computing.

5. CONCLUSION

This paper has presented a methodology based on a basic computing element that increases the level of design abstraction of FPGA designs. The element uses a combination of a simple CPU with a configurable pipelined datapath to implement basic floating-point vector and matrix operations. The function of the element can be changed through replacing the CPU program as well as through partial runtime reconfiguration of the dataflow unit; implementation results for both features have been shown. Performance results have been presented for a matrix multiplication operation in a MicroBlaze based system; the speedup of the basic computing element over the pure MicroBlaze with its hardware floating-point unit is about eight.

6. REFERENCES

[1] M. Dickinson, "System-level design for FPGAs," *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on*, 2007, keynote speech.

[2] J. L. Hennessy and D. A. Patterson, *Computer Architecture; A Quantitative Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2003.

[3] P. G. de Massas, P. Amblard, and F. Pétrot, "On SPARC LEON-2 ISA extensions experiments for MPEG encoding acceleration," *VLSI Design*, vol. 2007, no. 1, pp. 1–10, 2007.

[4] J. Potter and W. Meilander, "Array processor supercomputers," *Proceedings of the IEEE*, vol. 77, no. 12, pp. 1896–1914, Dec 1989.

[5] S. Vassiliadis, S. Wong, G. Gaydadjiev, K. Bertels, G. Kuzmanov, and E. Panainte, "The MOLEN polymorphic processor," *Computers, IEEE Transactions on*, vol. 53, no. 11, pp. 1363–1375, Nov. 2004.

[6] R. Krashinsky, C. Batten, M. Hampton, S. Gerding, B. Pharris, J. Casper, and K. Asanovic, "The vector-thread architecture," *IEEE Micro*, vol. 24, no. 6, pp. 84–90, 2004.

[7] T. Ungerer, B. Robič, and J. Šilc, "A survey of processors with explicit multithreading," *ACM Comput. Surv.*, vol. 35, no. 1, pp. 29–63, 2003.

[8] C. Jesshope, "Scalable instruction-level parallelism," in *Computer Systems: Architectures, Modeling, and Simulation*. Springer Berlin / Heidelberg, 2004, pp. 383–392.

[9] N. Zhang and R. W. Brodersen, "The cost of flexibility in systems on a chip design for signal processing applications," 2002, Berkeley Technical Report.

[10] K. Chapman, "PicoBlaze 8-bit embedded microcontroller user guide," Xilinx, Tech. Rep., 2005.

[11] K. Underwood, "FPGAs vs. CPUs: trends in peak floating-point performance," in *FPGA '04: Proceedings of the 2004 ACM/SIGDA 12th International Symposium on Field-Programmable Gate Arrays*. New York, NY, USA: ACM, 2004, pp. 171–180.

[12] J. Kadlec, R. Bartosinski, and M. Danek, "Accelerating MicroBlaze floating point operations," *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on*, pp. 621–624, 27-29 Aug. 2007.

[13] C. Grelck, S.-B. Scholz, and A. Shafarenko, "Coordinating data parallel SAC programs with S-Net," *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pp. 1–8, 26-30 March 2007.

[14] J. W. Janneck, "Actors and their composition," *Formal Aspects of Computing*, vol. 15, no. 4, pp. 349–369, 2003.