A Novel Codesign Approach based on Distributed Virtual Machines

Christian Kreiner, Christian Steger, Egon Teiniker, Reinhold Weiss
Institute for Technical Informatics
Graz University of Technology, Austria
A-8010 Graz, Inffeldgasse 16, Austria
kreiner, steger, teiniker, rweiss@iti.tu-graz.ac.at

ABSTRACT

This paper describes a hardware/software codesign approach for the design of embedded systems based on digital signal processors and FPGAs. Our approach is based on distributed virtual machines for simulation and verification of the application on a Linux cluster and for running the application on different target architectures (DSPs, FPGAs) as well. The main focus is the description of the virtual machine, which was designed to make DSP applications portable across different platforms while maintaining optimal code.

1. INTRODUCTION

In recent years the interest in the problem of designing mixed hardware/software systems has increased due to growing system complexities. Hardware/software codesign environments are very important in order to shorten and improve the design process of DSP applications. To reach the required hard real-time behavior of such systems, the designers should be able to select between functions implemented in programmable DSPs or functions implemented in dedicated hardware (FPGAs). This choice is determined by requirements like performance, cost, power, weight and more. We have developed a hardware/software cosimulation environment for the verification of heterogeneous DSP applications [11].

1.1 Application

Realtime simulation gains more importance in the development of new technical products. Therefore, the overall project deals with a process coupled simulator for the verification of radio systems (VHF band from 118 to 400 MHz) for air traffic control. The realtime simulator supports multipath fast fading effects, propagation losses and additive white Gaussian noise development of an aeronautical radiochannel in realtime. To fulfill the realtime constraints combined hardware/software systems are necessary

2. RELATED WORK

Two main approaches for multilanguage verification functional correctness of the hardware and software work together [8]: the compositional and the cosimulation based approach. The compositional approach aims at integrating the partial specification of subsystems into a unified representation. This description is used for the verification of the entire behavior of the system. The cosimulation approach (e.g. [8, 11]) based on multilanguages consists in interconnecting the simulation environments associated to each of the partial specifications.

The main focus of this section deals with related projects based on virtual machines.

V. K. Madisetti [9] presents a virtual prototyping environment that is suitable for system design and legacy system reengineering. The application is composed of a set of generic instructions which are interpreted by a processor model. The instruction set also includes MPI primitives [6] to model the communication. The processor model comprises a set of configurable generic VHDL processes. This modular approach allows it to map communication and computation and control related design decisions very quickly.

Randall S. Janka [7] developed a new specification and design methodology which effectively allows the designer to evaluate candidate architectures and technologies before committing to a technology. This methodology is called MAGIC which stands for the 'methodology applying generation, integration, and continuity'. The application is specified on MATLAB/Simulink, the de facto lingua franca of algorithm developers. The Simulink model consists of blocks and links between these blocks. The blocks are transformed into VSIPL computation function calls, while the links are transformed into MPI function calls. VSIPL stands for 'vector, signal and image processing' and is an API which provides hundreds of functions. The C code generation is a functionality of Simulink's Real-Time Workshop (RTW).

Another quite different idea influencing this project is the use of virtual machines to keep hardware independence for heterogenous target architectures. The Java Virtual Machine (JVM) [13] is the Java component responsible for hardware and operating system independence and the small size of compiled code of the Java platform. The JVM does not assume any particular implementation technology, host hardware or operation system. It may be implemented in software, microcode or directly in silicon. The JVM knows nothing about the Java programming language, only about a particular binary format, the class–file format.

3. NOVEL HW/SW CODESIGN APPROACH

We apply a new HW/SW codesign approach (Figure 1) which allows us to simulate our designs on different abstraction levels. It is possible to consider results of former implementations in the current design flow. Designing starts with modeling the problem in Simulink. At this point it is already possible to simulate the dataflow graph [2]; thus we can find and remove functional errors. The dataflow graph consists of functional blocks and connections between the blocks. The dataflow graph is the highest level of abstraction, because only the application is modelled and no assumptions about the target architecture are made.

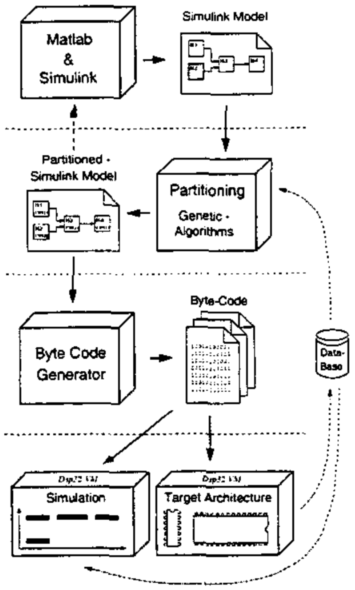

Figure 1: Design flow

The graph is partitioned depending on the target architecture [1]. In the partitioning process, computation time, required memory, area and power consumption of the blocks and also the costs of communication between the functional blocks are considered. We use genetic algorithms [14, 4] to achieve a suitable partitioning. The partitioned dataflow graph is translated into bytecode [3].

The next level of abstraction is the bytecode level, where the application is represented as a bytecode file for every CPU. This bytecode is interpreted by a virtual machine for 32bit DSP processors (Dsp32VM). For simulation the Dsp32VM also runs on a Linux network using the Message Passing Interface (MPI) [6] for communication between the computers (Figure 2). At this level, experiments with different target architectures and parameters (design space exploration) can be made. Also the functional correctness of the distributed application is verified.

Dsp32VM also runs on the target architecture (DSP, FPGA) and can interpret the same bytecode. Thus it is possible to simulate and visualize the program flow of the target architecture application already on the Linux network. Runtime, communication costs etc. for every target architecture are collected in a database and can be used for future partitioning and simulation tasks.

3.1 The virtual machine Dsp32VM

The virtual machine for 32bit DSP-CPUs (Dsp32VM) interprets a bytecode in which the application program is encoded. The interpreter fetches the bytecode and executes the instructions in the native code. The instruction set architecture and bytecode interpreter are optimized for the digital signal processing domain. That means that there are many complex instructions including vector and matrix operations. Figure 2 shows the architecture of the Dsp32VM consisting of:

Code Memory. Contains the whole bytecode (instructions and constants) and is loaded from the host computer at initialization time.

Data Memory. Also called heap, contains the variables of the application (scalars, vectors and matrices).

Fetch & Decode Unit. All instructions, are fetched and decoded by this unit; therefore, this unit must be implemented very carefully to avoid execution delays.

Execution Unit. This unit consists of subunits for every datatype (scalar integer / float, vector integer / float, matrix integer / float). It is possible to implement just a few of these subunits, for example, on FPGA devices. In this case, the bytecode ID carries the information about which subunit is used.

Registers. There are 32 registers on the Dsp32VM to speed up the bytecode execution. The execution unit provides a lot of one–word instructions which operate on registers.

Boot Unit. At initialization time the bytecode is loaded down from the host computer. The boot unit handles this job.

Timer Unit. Up to 8 timers can be used in one application to generate the sampling clock for the calculation of the dataflow graphs.

IO Unit. This unit consists of several input and output ports which are used for interfacing the target architecture.

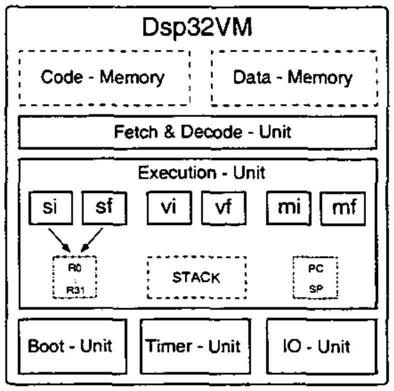

Figure 2: Architecture of Dsp32VM

3.2 Target architecture model

By applying a virtual machine on every CPU on the target board, we can model the target architecture as an architecture graph . This architecture graph $G_A = (V_A, E_A)$ consists of nodes V_A realized by virtual machines and edges E_A implemented as unidirectional communication channels.

The communication model (Figure 3) is based on a synchronous oneword communication. Sender task A writes data in a send queue and task B reads the data from a receive queue. Write time t_{SP}, read time t_{RG}, transfer time t_{TW} and buffer size are parameters which can be modified during simulation on the Linux network. The times for initialization of the buffers t_{SI} and t_{RI} are also adjustable.

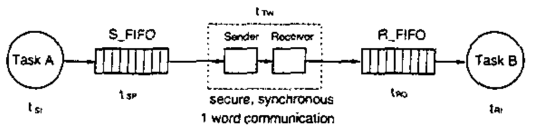

Figure 3: Communication channel

The instruction set includes commands for sending vectors and matrices. These instructions are based on the one-word communication model described above.

3.3 Bytecode structure

An application for the Dsp32VM consists of a variable number of 32bit words stored in a bytecode file. Figure 4 shows the structure of a bytecode application.

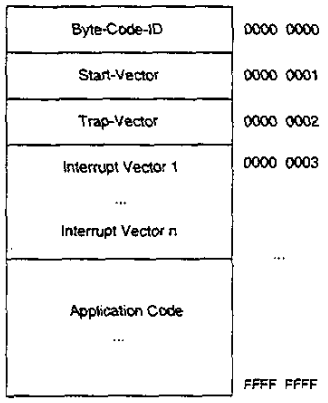

Figure 4: Bytecode structure

The bytecode file consists of a few header words and the real application code:

Bytecode ID. To recognize a bytecode file, the first word is an identifier (Figure 5). It contains the version and revision number of the virtual machine which are necessary for execution. The bytecode ID also includes a number of flags which indicate the execution units used for execution of the file.

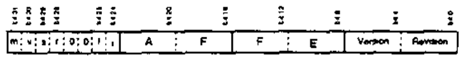

Figure 5: Bytecode identifier

Start Vector. The start vector contains the address of the first word of an application.

Trap Vector. In the trap vector the address of the trap handler within the application code is stored.

Interrupt Vectors. These vectors contain the addresses of all of the interrupt service functions.

Application Code. Here the real bytecode for application is stored. This area includes also the constants, trap- and interrupt-handler functions.

Booting the bytecode means that the Dsp32VM inspects the bytecode ID and if the Dsp32VM can execute the file, the bytecode will be loaded into the code memory. Then the program counter is set with the start vector and the execution unit starts interpretation.

3.4 Bytecode vs. native code machine

As shown in Figure 6, the bytecode machine contains an additional abstraction layer which maps the different target architectures on a uniform machine, the Dsp32VM. Thus the same bytecode can be interpreted on every target architecture running a Dsp32VM.

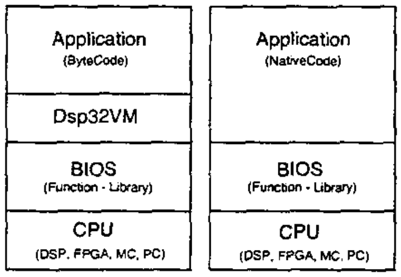

Figure 6: Bytecode vs. native code machine

We implemented the Dsp32VM on a Linux cluster, the TMS320C67 DSP from Texas Instruments and an Xilinx Virtex FPGA board. This environment allows us to develop an application on a Linux cluster using debugging and profiling tools for the Dsp32VM. The resulting bytecode is loaded down to the individual target architecture; thus, there is no need for recompilation of the application for DSP and FPGA devices.

However, there are unfortunately some disadvantages which are: i) interpreting the bytecode produces an overhead in execution time. Every instruction must be fetched, decoded and executed. As we use a hash-table to decode instructions, we need $O(1)$ time, independent of instruction set size. ii) the implementation of a Dsp32VM is very critical because poor implementation results in a bad performance for every application running on this virtual machine.

Despite the disadvantages described above the use of virtual machines is still very advantageous:

- The Dsp32VM layer provides a uniform architecture for every target platform (DSP, FPGA, MC, Workstation). In other words, every target board can be modelled on a uniform architecture graph $G_A = (V_A, E_A)$.
- To use a new CPU, we only have to implement a well specified virtual machine. All design tools and applications can be reused without recompilation.
- We simulate applications on a Linux cluster and tune the parameters of the architecture graph (number of CPUs, buffer size etc.) to find out the best target architecture for the application.
- For a fast execution of the application, the bytecode can be used as input for a native code generator which translates the bytecode into the native code for a particular CPU.

3.5 Instruction set

To reduce the overhead of interpretation, we provide very powerful instructions which operate on scalars, vectors and matrices. All instructions, in the virtual machine are implemented in the native code on a particular target CPU.

3.5.1 Datatypes

Because of the 32bit architecture of the virtual machine, the base datatypes are:

- 32bit signed integer
- 32bit floating point (IEEE 754)

In every memory location, one of the base datatypes (called scalars) can be stored. Based on these scalars we provide extended datatypes which are described below:

- One-dimensional arrays of scalars (vectors)
- Two-dimensional arrays of scalars (matrices)
- First in last out buffers (stacks)
- First in first out buffers (queues)
- Circular buffers

It is important to note that no type information is stored together with the base datatypes. The type of data value is determined by the instruction using it. Extended datatypes, of course, need additional information like size, read and write position this additional information is stored together with the data.

3.5.2 Instruction encoding

Encoding defines the structure of the instructions on the binary level. There are instructions from one to four words in length. The structure of the first instruction word is uniform for all instructions (figure 7).

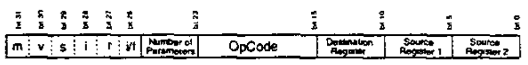

Figure 7: First instruction word

Prefix (bits 26 ... 31). The prefix contains information about the data type of arguments and the execution unit needed for executing the instruction.

Number of parameters (bits 23 ... 25). Number of parameters needed for execution of the instruction.

Opcode (bits 15 ... 22). The opcode defines the instruction and at the same time it is used as a hash-key. Using a hash table for encoding the instructions reduces the overhead of interpretation in the Dsp32VM.

Register number (bits 0 ... 14). There are three five bit groups which contain the numbers of one destination and two source registers. By doing this we can encode most of the instructions in one instruction word.

3.5.3 Library mechanism

To enhance the power of the Dsp32VM, we add a library mechanism which allows us to include native code operations within bytecode applications. The native code operations can be executed in a transparent way by using the `rExecute` instruction. The native code operation is specified by a unique 32 bit value which is used as a hashing key to access the operations in $O(1)$ time.

Thus we have the possibility to test new algorithms in the native code without having to change the instruction set of the virtual machine.

But the native library mechanism must be handled with care because native function calls lead to applications which are not portable without implementation of the used library operations on different target architectures. One way to reduce the risk of porting problems is to define a standard Dsp32VM library which is available on every target architecture.

3.6 HW/SW partitioning

To map a dataflow graph on a heterogeneous target architecture which consists of DSPs and FPGAs, it is necessary to partition the functional blocks so that design constraints, including the runtime constraints, are met and at the same time the system cost is minimized.

Our partitioning method uses genetic algorithms to solve this problem. Genetic algorithms are based on the principle of evolution. Different individuals are created, but only the fittest survive and increase. An individual is reduced to a chromosome represented by a binary string. In this context two major issues have to be considered: i) mapping of the partitioning problem on a binary string and ii) evaluation of this chromosome. For creation, mutation and crossover of the chromosome population, we use a Parallel Genetic Algorithm Library (PGA-Pack) [12]. The PGA-Pack is a C library which allows distributed computation on computer networks using MPI. More details in [10].

4. SIMULATION ON A LINUX CLUSTER

One of the key features of our approach is the possibility to simulate bytecode on a Linux cluster. The simulation results in a graphic representation of the runtime behavior of the application on the target architecture. The use of an abstract virtual machine allows us to simulate a heterogenous architecture (DSP, FPGA, MC) very well, because there is a clear mapping between instructions executed by the target architecture and the instructions simulated in the Linux cluster. Thus we can determine the execution time of every instruction on every target CPU by measuring. These execution times are stored in a database and will be used during simulation on the Linux cluster.

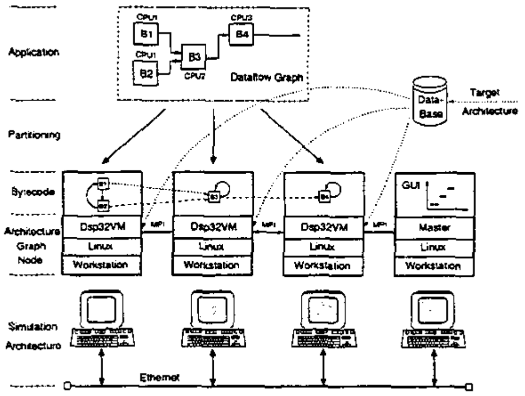

Figure 8: Simulation on a Linux cluster

As shown in figure 8, the simulation environment consists of $n+1$ workstations to simulate a target architecture of n CPUs and one

workstation is called the master which controls the simulation and provides the user interface. On every Linux workstation used for simulating, a CPU runs a virtual machine (Dsp32VM). For simulation, every Dsp32VM obtains the hardware information (execution times, memory size, buffer size etc.) from the database. The master loads the bytecode files into the code-memory of every Dsp32VM and starts the simulation. During simulation, logging information is generated and stored in a logfile.
We use the Message Passing Interface implementation MPICH [5] for all communication between the workstations. For logging we use the Multi-Processing Environment (MPE). The virtual machine and the master program are written in C++. Because of the object oriented software paradigm, this allows us to design the project as modular as possible. For example, by using polymorphism we can implement different I/O units for the Dsp32VM derived from an abstract I/O class. Another motivation to choose C++ was the fact that the MPICH and MPE libraries are written in C.
As explained above, the hardware parameters for a particular application can be adjusted to find the best target architecture. For example, you can replace a DSP by a FPGA and observe the improvement in execution time. You can also simulate the application on different components-of-the-shelf (COTS) boards modelled as an architecture graph. With this approach, we can significantly reduce time to market.

5. EXPERIMENTAL RESULTS

The following sections present the results of experiments and simulations with the novel codesign approach based on distributed virtual machines.

5.1 HW/SW partitioning

We used the mathematical model of a so-called "loading bridge" as a sample application and mapped it to a dataflow graph [10]. Table 1 shows the difference in execution times for the simulation in this example, depending on the target architecture. We have simulated the "loading bridge" application for a target architecture with one, two and four DSP processors (Texas Instruments TMS320C40). The results show that the speedup for two and four processors is not significant. This is a consequence of our scheduling method which works very well for functional blocks with nearly the same execution time, but it produces unsatisfactory results in other cases. This problem is solved by using a heterogenous target architecture. The most time consuming block (integrator) is implemented on a FPGA device to reduce its execution time from 27 to 10 cycles. As shown in table 1, the architecture with two DSP processors and one FPGA device is the best one.

target architecture	execution cycles
1 DSP (C40)	185
2 DSPs (C40)	173
4 DSPs (C40)	141
2 DSPs + 1 FPGA	118

Table 1: Execution cycles for different target architectures

5.2 DSP implementation

The second set of results is derived from an implementation of the Dsp32VM on a multi-DSP board of Blue Wave Systems. This board contains two TMS320C6701 floating point DSPs operating at 167 MHz, and it can be optionally extended with a Xilinx FPGA module. We have compared the execution times of bytecode and native code applications to find out the overhead of the bytecode interpreter. By using the standalone simulator and profiler *load6x* from Texas Instruments, we have obtained the following results:

Single instructions. Every Dsp32VM bytecode instruction is implemented by an individual C function. Depending on the number N of data words every instruction has to process, we define the execution time as:

$$F(N) = F_1 + N \cdot F_2$$

That means that the number of cycles needed to execute any bytecode instruction is given by the constants F_1 and F_2. These constants are a result of measuring at the target CPUs (for example the FIR filter instruction on the TMS320C6701 DSP: $F_1 = 134$ and $F_2 = 4$).

Bytecode interpreter. The interpreter fetches instructions, decodes them and calls the appropriate C functions. The number of cycles needed to interpret n instructions in the Dsp32VM is given by:

$$Cycles = C_1 + n \cdot C_2$$

Measurments with the multi-DSP board gave $C_1 = 17$ and $C_2 = 59$.

Bytecode application. Thus the number of cycles needed to execute an application can be then calculated by:

$$Cycles = C_1 + n \cdot C_2 + \sum_{i=0}^{n-1} F_{1i} + N_i \cdot F_{2i}$$

Upon definition of execution time for bytecode applications, we have compared them with native code programs. We compared bytecode with native C code and modified C code ("Native- Functions" where each operation is implemented as a particular C function). Table 2 contains the execution cycles for different numbers of multiplications with floating point variables. The results show that using simple scalar bytecode instructions like addition and multiplication is very expensive.

MUL N Elements	*Native-Code*	*Native-Functions*	*Byte-Code*
1	17	26	102
10	112	230	867
100	169	2120	8517

Table 2: Comparison of the bytecode and native code (simple functions)

On the other hand, using specific signal processing, vector or matrix instructions leads to good results especially for large N (table 3). For example, a FIR filter instruction with order $N = 150$ has an overhead of less than 20%.

N'th Order FIR filter	*Native-Functions*	*Byte-Code*
10	135	254
100	495	614
150	695	814

Table 3: Comparison of the bytecode and native code (complex functions)

5.3 FPGA implementation

An FPGA implementation of Dsp32VM on a Xilinx Virtex FPGA board is underway. In a first version, it implements a subset of the full Dsp32VM as defined in section 3.1. The supported command set comprises all program control statements and the scalar integer operations. Currently there is no exception mechanism for detecting unsupported commands at execution time. This requires the development tools to prohibit the use of such commands at compile time. The ultimate goal, however, is to support the entire command set in this implementation. All Dsp32VM addressing modes are supported: immediate, direct, register, and register indirect.

The virtual machine is implemented on the FPGA in a simple micro-programmed style. This allows to extend the supported command set with little modifications in the VHDL source code. Registers are implemented as a RTL (register transfer level) VHDL code. Code, data and stack are stored in SRAM, either internally or externally. Partitioning between the size of code/data memory and stack has to be done manually. Microcode is stored in the internal SRAM.

The particular bitcoding of the instructions is, of course, identical to the software implementation on DSPs. So it is possible to reuse the existing Dsp32VM design flow and development toolchain from source code down to Dsp32VM bytecode. The bytecode is then translated into a memory config file for Virtex' SRAM using a specialized tool specific for this hardware implementation. Finally, the resulting file can be loaded into the FPGA and executed.

5.4 Simulation results

As shown above, the number of execution cycles needed for a bytecode application can be exactly calculated, if the constants C_1, C_2, F_{1i} and F_{2i} are known. Due to the same architecture of the Dsp32VM on the target board and the Linux simulation cluster, we were able to simulate the execution of bytecode applications with a resolution down to single CPU cycles.

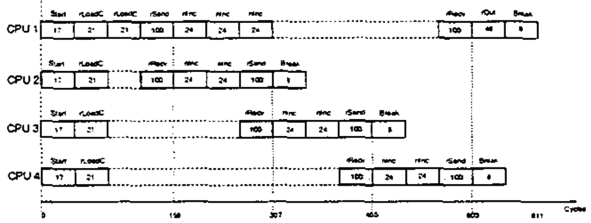

Figure 9: Simulation of synchronous communication

Figure 9 shows the simulation result of a simple application which transfers a single integer value between four CPUs in a synchronous way. Every involved CPU increments the received integer value and sends it to the next one.

Our experiments have shown that a typical signal processing application can be implemented in the bytecode with acceptable overhead, and the uniform Dsp32VM architecture on every CPU makes it possible to simulate heterogenous multiprocessor applications on a Linux cluster with minimal errors.

6. CONCLUSION AND FUTURE WORK

We have presented a new hardware/software codesign approach for the design of embedded systems based on digital signal processors and FPGAs. It is based on distributed DSP virtual machines for simulation and verification on a Linux cluster and also for running the application on different target architectures (DSPs, FPGAs). Our first experiences have shown the feasibility of this approach. Ongoing investigations will evaluate this framework by giving more complex examples.

Another usecase for the concept of virtual machines is the possibility to utilize simulators for education in hardware/software codesign. Students are able to develop, optimize applications and explore the design space for expensive multi-processor boards by using the Dsp32VM simulator environment.

7. REFERENCES

[1] Jakob Axelsson. *Three Search Strategies for Architecture Synthesis and Partitioning of Real-Time Systems*. Department of Computer and Information Science, Linkoeping University Sweden, 1996.

[2] Shuvra S. Bhattacharyya, Praveen K. Murthy, and Edward A. Lee. *Software Synthesis from Dataflow Graphs*. Kluwer Academic Publishers, 1996.

[3] Matthias K. Dalheimer. *Java Virtual Machine*. O'REILLY, 1997.

[4] D.Saha, R.S. Mitra, and Anupam Basu. *Hardware Software Partitioning using Genetic Algorithm*. Department of Computer Science & Engineering, Indian Institute of Technology, 1996.

[5] William Gropp and Ewing Lusk. Installation Guide to mpich, a Portable Implementation of MPI Version 1.2.1. Technical report, Argonne National Laboratory, University of Chicago, 2000.

[6] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI*. The MIT Press, 1999.

[7] Randall S. Jankaa and Linda M. Wills. A Novel Codesign Methodology for Real-Time Embedded COTS Multiprocessor-Based Signal Processing Systems. *CODES*, 2000.

[8] A.A. Jerraya, C.A.Valderrama, and et.al. Models and Languages for System-Level Specification and Design. In *NATO-ASI on System Level Synthesis, NATO Advanced Study Institute, Barga, Italy, Aug. 10–21*, 1998.

[9] M.H. Kahn and V.K. Madisetti. System Design and Re-Engineering Through Virtual Prototyping: A Temporal Model–Based Approach. *IEEE*, 1998.

[10] Christian Kreiner, Christian Steger, Egon Teiniker, and Reinhold Weiss. A HW/SW Codesign Framework based on Distributed DSP Virtual Machines. In *Proceedings of the Euromicro Symposium on Digital Systems Design (DSD 2001)*, Warsaw, Poland, 2001.

[11] Christian Kreiner, Christian Steger, and Reinhold Weiss. A Hardware/Software Cosimulation Environment for DSP Applications. In *Proceedings of the 25th Euromicro Conference*, volume 1, pages 492–495, Milan, Italy, September 1999. IEEE Computer Society.

[12] David Levine. *Users Guide to the PGAPack Parallel Genetic Algorithm Library*. Argonne National Laboratory, 1996.

[13] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Java Series, 1999.

[14] Ralf Niemann. *Hardware/Software Co-Design for Data Flow Dominated Embedded Systems*. Kluwer Academic Publishers, 1998.