A Distributed Computer Architecture for Qualitative Simulation Based on a Multi-DSP and FPGAs

Marco Platzner, Bernhard Rinner, Reinhold Weiß
Institute for Technical Informatics
Graz University of Technology, Austria
{marco,rinner,rweiss}@iti.tu-graz.ac.at

Abstract

The design of a specialized computer architecture for qualitative simulation is presented.

Our interest focuses on the hardware design of an application-specific computer architecture which is composed of programmable processors (digital signal processors TMS320C40) and application-specific integrated circuits (FPGAs). Two design strategies are considered to improve the performance. Primitive functions are hardware-implemented using FPGAs (software/hardware migration). More complex functions are mapped onto a multi processor system formed by TMS320C40. This computer architecture is designed for the well known algorithm for qualitative simulation - QSIM*[5].*

In this paper we present the design of a computer architecture for the constraint-check-function — a function of the QSIM *kernel.*

keywords:
TMS320C40, Multi-DSP, FPGA, application-specific computer architecture, QSIM, codesign

1 Introduction

In this paper we describe the design of a specialized computer architecture for applications in the field of qualitative simulation. A primary goal in designing special-purpose computer architectures is to improve the performance. Larger *(real)* problems can be solved more effectively on special-purpose architectures than on general-purpose architectures. High performance is also essential for most real-time applications. Specialized computer architectures and design methodologies in the area of artificial intelligence have emerged recently. Several case studies for computer architectures for intelligent systems are presented in [2].

We are implementing the special-purpose computer architecture by using programmable processors (TMS320C40) and application-specific integrated circuits (FPGAs). This flexible implementation is well suited for experimental evaluation of the design. Two strategies are considered in this computer architecture to improve the performance. Firstly, the runtime of frequently used primitive functions is reduced by hardware implementation. Software-implemented functions migrate to hardware-implemented ones. Software/hardware migration is one important aspect of the codesign paradigm. Examples can be found in [3][4]. Secondly, more complex functions are mapped onto a multiprocessor system. The parallelism of the application is exploited.

This application-specific computer architecture is developed for the well known algorithm for qualitative simulation QSIM [5]. In qualitative simulation physical systems are modeled on a higher level of abstraction than in other simulation paradigms, e.g. continuous simulation. Simulating continuous models requires extensive computation and results in a detailed description of the system behavior. Qualitative simulation leads to a less detailed description of possible behaviors. There are many applications where the detailed description is not necessary. Furthermore, a corresponding qualitative model is simulated more efficiently than a continuous one, due to a less accurate description.

This computer architecture for qualitative simulation will extend simulators which are already in use at our institute. These simulators are part of a *Distributed Real-Time Expert-System for Fault Diagnosis in Technical Processes* [12][13]. The already applied hybrid simulation technique, which combines discrete and continuous simulation, will be extended by the paradigm of qualitative simulation.

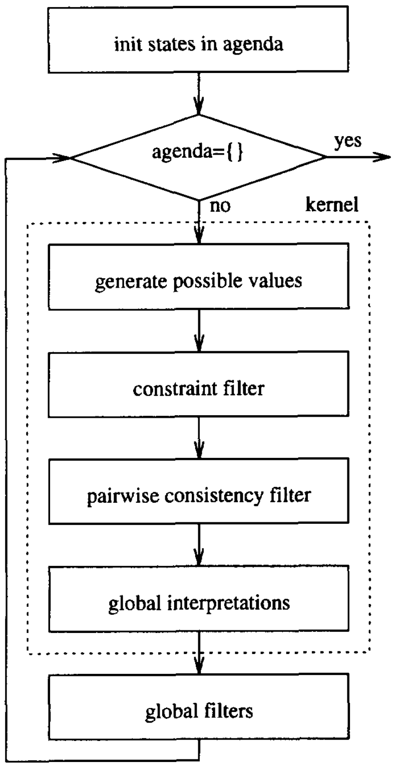

Figure 1: Flow chart of QSIM

2 Problem task

2.1 Algorithm QSIM

In this section we present a brief description of the algorithm QSIM. *Variables* and *constraints* are basic components of a qualitative model (constraint network). Variables represent system parameters (e.g. speed, temperature etc.) and constraints describe relations between system parameters. QSIM uses several types of constraints which represent arithmetic relations (ADD–, MULT–, D/DT–constraints) and functional dependencies (M^+–, M^-–constraints) between variables.

Figure 1 shows the flow chart of QSIM. *States* are stored in a global queue called agenda. A state in QSIM is defined as an assignment of values to all variables of the model. A state characterizes the system at a given time. In one simulation step (one loop cycle) all possible values for the next time step are determined. Qualitative simulation can predict several behaviors — contrary to continuous simulation. The simulation step is repeated until the agenda is empty or a time limit or state limit is exceeded. The individual steps of this procedure can be informally described as follows.

The first step generates the possible values for the next time step for all variables. The possible values in combination with the model description (constraint network) define a *constraint-satisfaction-problem (CSP)* [7]. The following three steps are used to solve a CSP (i.e. to determine all global consistent value assignments).

The constraint filter checks all value assignments for a given constraint. The pairwise consistency filter rejects inconsistencies between adjacent constraints. Constraints are adjacent if they share a variable. The last step in solving the CSP is the generation of global interpretations. A global interpretation is formed by each consistent combination of possible values for all variables. This step is achieved by a backtracking algorithm. Global filters reduce the set of new states which are added to the agenda. There are many global filters in QSIM. Some of them are necessary while most of them are optional extensions of QSIM.

2.2 Objectives

QSIM is implemented in LISP and its source code is public domain. The design of the presented application-specific computer architecture is based on QSIM-version NQ 2.0. Following system aspects strongly influence the design.

Execution speed Applications of qualitative simulation rely on the availability of fast qualitative simulators. The major objective of our project is to increase significantly the performance of current simulators.

Scalability A further important aspect of this computer architecture is the scalable design. A higher performance can be achieved by increasing the number of processing elements.

Real-time requirements The knowledge of guaranteed execution times is important for real-time applications of qualitative simulation. Current QSIM implementations allow *timed constraint reasoning.* After a given runtime the simulation process is aborted and states already in the agenda are excluded from further processing.

The designed computer architecture eases real-time applications for two reasons. Firstly, maximum execution times are reduced due to the improved performance. Secondly, analysis of maximum execution

Model	*Total*	*Kernel*	*Global Filters*	*Init*
BATHTUB	0.268 s	57.8 %	33.3 %	8.9 %
BOUNCING-BALL	2.842 s	58.1 %	34.0 %	7.9 %
SIMPLE-BALL	2.032 s	51.8 %	37.7 %	10.5 %
Q2-BATHTUB	0.748 s	26.5 %	70.7 %	2.8 %
TOASTER	1.695 s	46.7 %	48.7 %	6.6 %
HEART-REGULAT	3.827 s	76.7 %	21.8 %	1.5 %
STARLING	0.778 s	75.4 %	21.3 %	3.3 %
4-REACTIONS	569.506 s	92.1 %	5.0 %	2.9 %

Table 1: Runtime ratio of QSIM-functions measured on a TI-Explorer LISP workstation. The total runtime and the runtime ratios of kernel functions, global filters, and initial processing (Init) are shown. The used models are included in the QSIM package.

times is simplified. This is achieved by mapping iterations from time domain into space domain [8].

2.3 Kernel functions

The qualitative simulator QSIM is a very complex algorithm and has a lot of optional functions. Design considerations of this computer architecture are restricted to QSIM kernel functions. Kernel functions are essential in calculating one simulation step. They are shown in Figure 1. The kernel fulfills the basic functionality of qualitative simulation and it dominates the runtime of the overall algorithm [9]. Table 1 presents the runtime ratios of kernel functions, global filters, and initial processing.

Several model-based fault diagnosis and monitoring systems use qualitative simulation [6][1]. These systems normally do not require the functionality of the entire simulator — basically, QSIM kernel functions are sufficient.

3 QSIM computer architecture

Developing new computer architectures is a complex and challenging task. In this paper only the design of a part of the entire computer architecture is presented. We demonstrate this in more detail in the next section based on the *constraint-check-function* (CCF) — an essential function of the constraint filter. The limited complexity of the CCF enables a direct hardware implementation of this function. Software/hardware migration is applied to increase the performance.

The parallelism of the remaining kernel functions is exploited. The hierarchical structure of the kernel functions is mapped onto a hierarchical processor topology. Individual kernel functions are executed in parallel on several processors. Figure 2 represents an overview of the overall architecture.

4 Constraint-check-function

The constraint-check-function checks one tuple of the tuple set of a given constraint for consistency. A tuple set is the product space of all possible values for the considered constraint.

4.1 CCF analysis

Analysis shows that CCFs can generally be partitioned into two groups of sub-functions. Figure 3 presents this partitioning for the MULT–CCF. The first group consists of a fixed number of sub-functions. They take a tuple of possible values (pval_tuple) as input data and produce a boolean result. In case of the MULT–CCF these sub-functions are labeled *qdir*, *sign*, and *inf*. The second group is formed by several identical sub-functions, labeled *cvals[i]*. Each cval[i] sub-function takes the pval_tuple and one tuple of *corresponding values* as input data (labeled *cval_tuples*). The sub-functions of the second group return a boolean result. The cval_tuples are created dynamically outside the QSIM kernel. The number of cvals[i] sub-functions can increase monotonically during simulation. However, the creation of a cval_tuple is a very rare process compared with the number of CCF executions. All sub-functions of an individual CCF are mutually independent. The boolean results are combined to the total result by a logical AND–operation. *Short circuit evaluation* can be used to implement this operation. Whenever one sub-function returns a negative result, the entire calculation is aborted. A negative result is returned to the calling function.

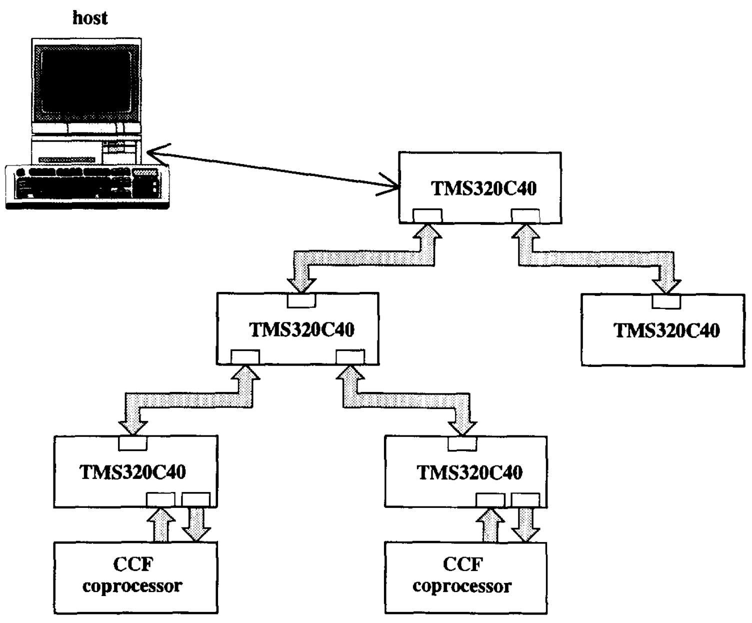

Figure 2: Example of the overall computer architecture. Five processing elements are connected in a tree-structure. Two processing elements are equipped with specialized CCF-coprocessors.

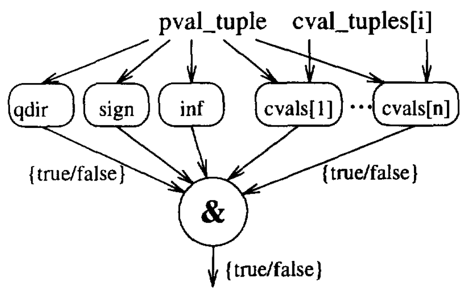

Figure 3: Sub-functions of the MULT–CCF.

Two methods of complexity estimation are used. A mathematical approach *(O–notation)* results in a quite trivial estimation. More suitable results can be provided by experimental measurements. We have implemented the CCFs in software on the target processor TMS320C40 and measured runtimes with different input data. Figure 4 presents the results of the runtime measurements of the MULT–CCF.

4.2 Design of a CCF coprocessor

Analysis brings up low complexity of the CCF subfunctions. This encourages a direct hardware implementation of the whole CCF. Hardware implementation also supports fast communication mechanisms between the sub-functions. This is required for an efficient exploitation of the parallelism. The hardware implemented CCF can be considered as application-specific coprocessor. Following points are of special interest for the design.

Number of cval[i] elements Two strategies for the implementation of cval[i] sub-functions are considered. Firstly, one sub-function is implemented in hardware and the cval_tuples are checked sequentially. The second alternative is to implement several cval[i] processing elements and to check the cval_tuples in parallel. The number of cval_tuples is an important parameter for design considerations. It limits the number of processing elements. Although the number of cval_tuples is generally unbounded, experience reveals that in most cases only a few cval_tuples are used. Results from tracing a QSIM system are presented in Figure 5. Cval_tuples of all constraints have been counted while several models have been simulated. Even with complex models at most four cval_tuples have been traced for a given constraint. A coprocessor with four cval[i] elements can calculate the CCFs of the traced models in one step.

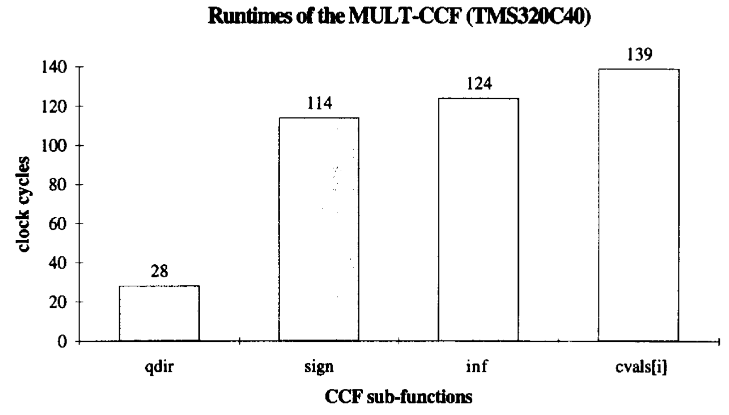

Figure 4: Runtimes (number of clock cycles) of CCF sub-functions measured on a TMS320C40. The labels *qdir*, *sign*, and *inf* represent sub-functions which take the pval_tuple as input data. The sub-function *cvals[i]* requires the pval_tuple and one cval_tuple as input data.

Customized memory architecture The memory which holds the cval_tuples is split into as many banks as cval[i] processing elements are implemented. This leads to a parallel memory access for all cval[i] elements. The disadvantage of this memory architecture is a more complex update procedure for the cval_tuples. Since the memory is updated outside the QSIM kernel, the complex update operation can be tolerated.

Coprocessor interface The interface between coprocessor and host processor is essential for the performance of the overall architecture. A high speed interface with low communication setup time is required. The amount of transferred data has to be minimized. For example, a better throughput can be achieved by packing several results into one message. This avoids redundancy in communication. However, the complexity of the coprocessor design is increased. Another technique to improve the throughput is to overlap computation and communication phases whenever possible. This can be supported by the use of buffer registers in the coprocessor.

4.3 Implementation and experimental evaluation

The design of the coprocessor is based on Xilinx XC40xx FPGAs to allow a fast and flexible development. The DSP TMS320C40 has been chosen as host processor. This DSP fulfills several requirements which are necessary for both, an efficient interface to the coprocessor and the implementation of a multi DSP system.

- Fast communication ports
- Independent communication channels
- Simultaneous I/O and CPU processing
- Multiprocessor support

Some special features of this DSP, like hardware multiplier and special addressing modes, are of minor interest to this project. Other simulators developed at the institute have also been implemented on the DSP TMS320C40 [10] [11].

A proposed computer architecture is presented in Figure 6. Data flow between host processor, FPGA, and cval_tuples memory is shown. Data transfer between DSP and CCF coprocessor is established via two separate communication ports. Unidirectional data transfer is preferred due to less communication setup time and less complex FPGA design.

Figure 7 defines three phases of CCF execution. There are two communication phases (A and C) and one computation phase (B). A sequential procedure for CCF execution is shown in Figure 7a. Increased throughput can be achieved by packing results into one communication (example in 7b) and by overlapping communication and computation phases (example in 7c).

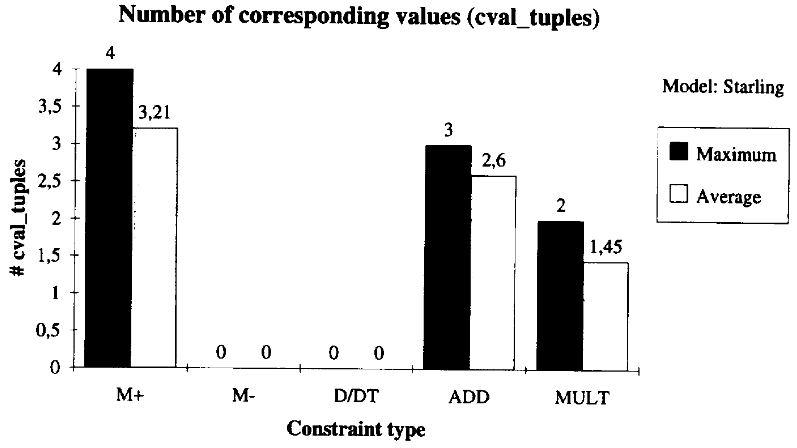

Figure 5: Average and maximum number of cval_tuples for individual types of constraints. The maximum number of cval_tuples does not exceed the moderate value 4, even in complex models.

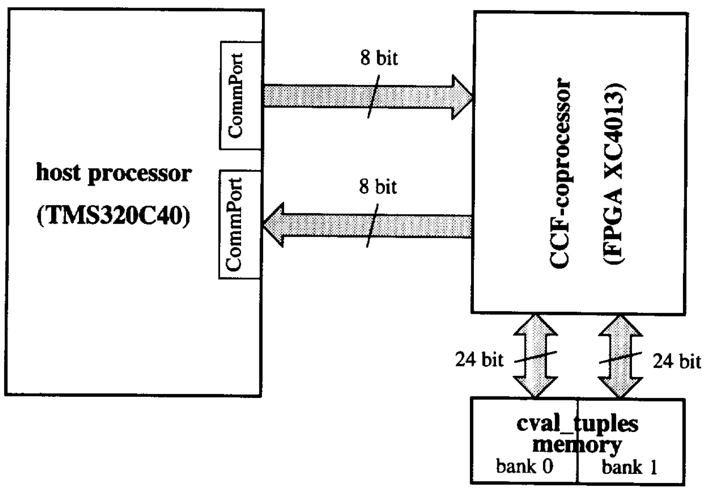

Figure 6: Example of a CCF-coprocessor system. Data paths between host processor, CCF-coprocessor, and cval_tuples memory are shown. The cval_tuples memory is partitioned into two banks.

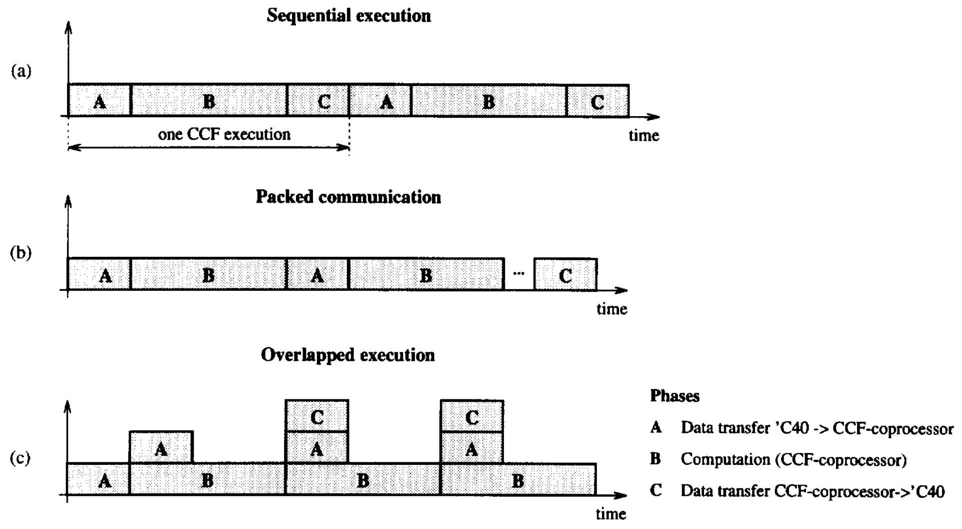

Figure 7: Phases of execution of the CCF-coprocessor system. Two communication phases (A, C) and one computation phase (B) are defined. A sequential execution is presented in (a). Two techniques for increasing the throughput are shown: Packing several results into one data transfer (b) and overlapping several phases (c).

Evaluation of this application-specific computer architecture is based on an experimental multiprocessor system consisting of several DSPs and FPGAs. A software implementation on a DSP is used as a reference system for the evaluation of the CCF coprocessor. We do not compare runtimes of this special-purpose system with runtimes of a LISP environment.

Two kinds of parameters are considered for evaluation. Firstly, the execution time of one CCF and secondly, the execution time of the *tuple filter*. The tuple filter checks the whole tuple set of a given constraint for consistency. It consists of 1 up to 64 CCF executions. Therefore, *execution time* measured in [sec] of one CCF and *throughput* measured in [CCFs/sec] are important parameters.

Development of the FPGA-based CCF coprocessor is in progress. The instruction set of the coprocessor is currently being defined. The sub-functions of the MULT–CCF coprocessor are being implemented. First preliminary results show that a MULT–CCF with one cval[i] processing element fits easily into one XC4013 FPGA. We will present experimental results in the near future.

5 Summary and future work

An application-specific computer architecture for the qualitative simulation algorithm QSIM is presented. Design considerations are mainly influenced by requirements for technical applications. The main objective is to improve the performance. This is achieved by two strategies — exploiting parallelism and software/hardware migration. These two strategies are illustrated by the example of the constraint-check-function — a function of the QSIM kernel. This function is analyzed, the computation and communication requirements are defined. An architecture consisting of a host processor and a coprocessor for CCFs is presented. Parameters for the experimental evaluation are discussed.

Implementation of several kernel functions is in progress (i.e. functions for CCF, constraint filter, and CSP). Computer architectures for these functions are independently developed and evaluated. The entire parallelism and best performance is exploited by combining these individual architectures.

References

[1] Daniel Dvorak and Benjamin Kuipers. Process Monitoring and Diagnosis: A Model-Based Approach. *IEEE Expert*, pages 67–74, June 1991.

[2] IEEE Computer, May 1992. Computer Architectures for Intelligent Systems.

[3] IEEE Computer, January 1994.

[4] IEEE Micro, August 1994. Pulling Together: Hardware/Software Codesign.

[5] Benjamin Kuipers. Qualitative Simulation. *Artificial Intelligence*, 29:289–338, 1986.

[6] Franz Lackinger and Wolfgang Nejdl. Diamon: A Model-Based Troubleshooter Based on Qualitative Reasoning. *IEEE Expert*, pages 33–40, February 1993.

[7] Alan K. Mackworth. Consistency in Networks of Relations. *Artificial Intelligence*, 8:99–118, 1977.

[8] Dan I. Moldovan. *Parallel processing from applications to systems.* Morgan Kaufmann Publishers, 1993.

[9] Bernhard Rinner. Konzepte zur Parallelisierung des qualitativen Simulators QSIM. Diplomarbeit, Institut für Technische Informatik, Technische Universität Graz, Oktober 1993.

[10] Christian Steger. Implementation of a Petri-Net Simulator on TMS320C40. In *Proceedings of the Hungarian Transputer Users Group's Workshop on Parallel Processing in Education*, pages 115–119, Miskolc, March 1993.

[11] Christian Steger, Marco Platzner, and Reinhold Weiß. Performance Measurements on a Multi-DSP Architecture with TMS320C40. In *International Conference on Signal Processing Applications & Technology*, Santa Clara, California, USA, September 1993.

[12] Reinhold Weiß, et al. Ein verteiltes Echtzeit-Expertensystem auf Transputerbasis zur Fehlerdiagnose in technischen Prozessen. In *Workshop über Parallelverarbeitung*, Graz, Austria, June 1990.

[13] Reinhold Weiß, et al. Design and Implementation of a Distributed Real-Time Expert-System for Fault Diagnosis in Modular Manufacturing Systems. In *Euromicro 1991*, pages 799–806, Vienna, September 1991.