ELSEVIER

Simulation Practice and Theory 5 (1997) 623–638

SIMULATION PRACTICE AND THEORY

# Parallel qualitative simulation

Marco Platzner, Bernhard Rinner *, Reinhold Weiss

*Institute for Technical Informatics, Graz University of Technology, Steyrergasse 17/IV, A-8010 Graz, Austria*

Received 1 May 1996; revised 26 February 1997

**Abstract**

Qualitative simulation is a rather new and challenging simulation paradigm. Its major strength is the prediction of all physically possible behaviors of a system given only weak and incomplete information about it. This strength is exploited more and more in applications like design, monitoring and fault diagnosis. However, the poor performance of current qualitative simulators complicates or even prevents their application in technical environments. This paper presents the development of a special-purpose computer architecture for the best-known qualitative simulator QSIM. Two design methods are applied to improve the performance. Complex functions are parallelized and mapped onto a multiprocessor system. Less complex functions are accelerated by software to hardware migration; they are executed on specialized coprocessors. © 1997 Elsevier Science B.V.

*Keywords:* Qualitative simulator QSIM; Special-purpose computer architecture; Multi-DSP TMS320C40; FPGA

## 1. Introduction

Qualitative simulation is a rather new and challenging simulation paradigm which belongs to the research area *qualitative reasoning* (*QR*). In qualitative simulation, physical systems are modeled on a higher level of abstraction than in other simulation paradigms, like in continuous simulation. In continuous simulation, the structural description of a physical system is modeled by a mathematical description in form of differential equations. Qualitative simulation relies on a further abstraction of these differential equations — the so-called *qualitative differential equations* (*QDEs*). Qualitative simulation requires neither a complete structural description of the physical system nor a fully specified initial state. The major strength of qualitative simulation is the prediction of all physically possible behaviors derivable from this incomplete knowledge. Additionally, qualitative simulation potentially predicts behaviors which are not physically possible. Hence, qualitative simulation is *sound* but

* E-mail: rinner@iti.tu-graz.ac.at.


*PII* S0928-4869(97)00008-6

*incomplete*. The qualitative simulation paradigm is mainly used in applications where a detailed description of a physical system is not required or even not known. Major application areas of qualitative simulation are design, monitoring, and fault diagnosis [2,7,17].

At the Graz University of Technology, a distributed expert system for fault diagnosis in technical processes is developed [16]. In this embedded system, several local expert systems are tightly coupled with a technical process (e.g., a modular production system) and are supervised by a global expert system. Fault diagnosis is achieved by model-based algorithms. When a deviation is detected between calculated and measured states, diagnosis is started in a local expert system. The calculated state of the technical process is derived from discrete and continuous simulators. The applied fault diagnosis methods will be complemented by a qualitative reasoning method. This means, that the already used hybrid simulation technique [14], which combines discrete and continuous simulation, has to be extended by a qualitative simulator for deriving system behavior. The widely-used algorithm QSIM was chosen for this task. QSIM has been developed by Kuipers [6] and is the best-known algorithm for qualitative simulation. In the past years, QSIM has been widely studied, applied and extended, both by the original developers and by researchers worldwide. However, QSIM has still some drawbacks which prevent it from a wider application in industrial environments. The main drawback is that current QSIM implementations lack in runtime performance. Models of only low to medium complexity can be simulated within reasonable execution time. There are two reasons for this. First, QSIM tends to generate a huge number of system behaviors during simulation. This can at least partially be avoided by improved filter algorithms. Research in this area is done by the Qualitative Reasoning Group at UT Austin. Second, QSIM is implemented in LISP and executed on general-purpose computers. None of the research work in the area of qualitative simulation thus far studies or analyzes the computational complexity of QSIM or presents an empirical study of QSIM's runtime behavior [1]. However, complexity and performance issues are extremely important for applying qualitative simulation in embedded systems. High performance qualitative simulators are required in these environments. Another drawback is that nothing is known about the real-time capabilities of QSIM. Investigations are necessary to decide how to integrate the AI task QSIM into real-time environments.

In this paper, we present the development of a special-purpose computer architecture for QSIM with the primary goal to increase the performance [11]. The design of this application-specific computer architecture is based on an analysis of the QSIM algorithm and on extensive experimental measurements taken from a QSIM implementation [13].

The remaining part of this paper is organized into following sections. Section 2 analyzes the algorithm and the runtime behavior of QSIM. In Section 3, the design of a multiprocessor and specialized coprocessors for QSIM is presented. Some details concerning the prototype implementation are also given in this section. Section 4 presents the experimental results for the QSIM multiprocessor and the specialized coprocessors. Section 5 concludes this paper with a brief discussion of the results.

## 2. QSIM algorithm and runtime analysis

In QSIM, models are described as qualitative differential equations or equivalently as *constraint-networks*, which consist of *variables* and *constraints*. Variables represent system parameters, e.g., speed or temperature. The values of qualitative variables are expressed by two parts, a *qualitative magnitude* (*qmag*) and a *qualitative direction* (*qdir*). Constraints describe relations between system parameters. QSIM uses several types of constraints which represent arithmetic relations (e.g., ADD-, MULT-, D/DT-constraints) and functional dependencies (e.g., $M^+$-, $M^-$-constraints) between variables.

Fig. 1 shows the flow-chart of QSIM. In QSIM, an assignment of values to all variables of the model defines a *state*. A state characterizes the modeled system at a given time. States are stored in a global data structure called agenda. The first function in the flow-chart of QSIM, *initial state processing*, generates all possible initial states consistent with the model and its initial values. In one simulation step, i.e., a loop cycle in Fig. 1, one state is read from the agenda and all successor states are determined. These successor states are written back to the agenda. This simulation step is repeated until the agenda is empty or a time limit or state limit is

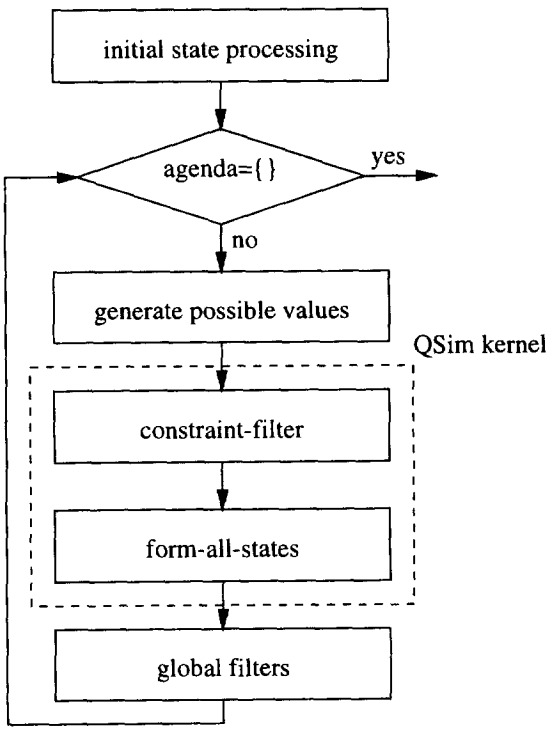


Fig. 1. Flow-chart of QSIM.

exceeded. The individual functions of a simulation step can be informally described as follows:

- Generate possible values. This function generates the possible values for all variables for the next time step. The number of transitions from one qualitative value to the next is limited. No variable can have more than four possible successor values [6].
- Constraint-filter. An assignment of possible values to all variables of a given constraint is called *tuple*. The task of the constraint-filter is to reject all tuples which do not satisfy the local consistency conditions in the constraint-network. This is achieved by calling a *tuple-filter* function for each constraint in the network. Additionally, the *Waltz-filter* discards tuples by detecting inconsistencies between adjacent constraints.
- Form-all-states. This function finds all consistent combinations of tuples in the constraint-network. Each consistent combination is a candidate for a successor state. Form-all-states is actually a backtracking algorithm to solve a *constraint satisfaction problem* (*CSP*) [9].
- Global filters. Each candidate state is checked by global filters. Candidate states which pass all global filters are written to the agenda. There are many global filters in QSIM. Some of them are necessary while many of them are optional extensions of QSIM.

The qualitative simulator QSIM is a very complex algorithm and has many optional features. The design considerations for our specialized computer architecture are restricted to the QSIM kernel functions constraint-filter and form-all-states. The kernel functions are essential in calculating one simulation step, and they normally dominate the overall runtime of QSIM. Furthermore, several model-based fault diagnosis and monitoring systems do not require the functionality of the whole simulator [2,7]. These systems are based on the kernel functions.

Fig. 2 presents an overview of the hierarchical structure of the QSIM kernel functions. The *constraint check functions* (*CCFs*) are primitive kernel functions and check the local consistency of individual tuples. For each constraint type there exists an individual CCF. The presented runtime ratios in Fig. 2 are extracted from various runtime measurements of a QSIM system implemented on a TI Explorer LISP workstation. This empirical runtime analysis was based on the simulation models included in the QSIM package. The complexity of these models ranges from QDEs with 3 variables and 3 constraints to QDEs with 28 variables and 21 constraints. The number of variables and constraints of these models is shown in Table 1. Additionally, the models RCS [5] (48 constraints, 52 variables), QSEA (21 constraints, 28 variables) and the artificially constructed model M1 (30 constraints, no variables specified) were used for the empirical runtime analysis and for the experimental evaluation of our QSIM computer architecture due to their high complexity. All models were simulated and the runtimes of the individual functions were measured. The runtime ratios represent an average of all simulated models. For most models, the kernel functions require more than 50% of the overall QSIM runtime. An important fact is that this percentage is positively correlated to the complexity of the model. Qualitative models for 'real-world' technical systems usually have

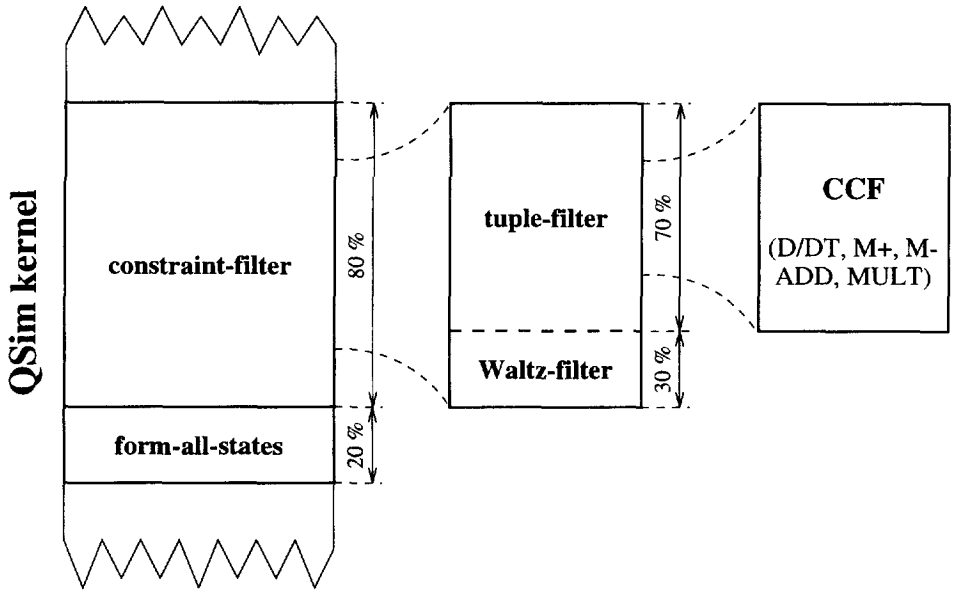


Fig. 2. Hierarchical structure and runtime analysis of the QSIM kernel. The runtimes are informally presented with regard to the runtime of the calling function.

Table 1
Simulation models from the QSIM package NQ 2.0 used for the empirical runtime analysis. The models Bouncing-Ball and Toaster have more than one QDE. These models describe physical systems with several modes of operation. QSIM switches between the QDEs depending on defined transition conditions

| Model | QDEs (number) | Variables (number) | Constraints (number) |
|---|---|---|---|
| Bathtub | 1 | 6 | 6 |
| Bouncing-Ball | 2 | 7 | 8 |
| | | 8 | 8 |
| Simple-Ball | 1 | 7 | 8 |
| Toaster | 4 | 4 | 4 |
| | | 10 | 10 |
| | | 5 | 5 |
| | | 3 | 3 |
| Heart | 1 | 28 | 21 |
| STLG | 1 | 17 | 18 |
| 4-reactions | 1 | 18 | 16 |

many constraints and variables [5,13]. For these models, kernel runtime ratios of up to 90% were observed. The empirical runtime analysis reveals that the tuple-filter and subsequently the CCFs dominate the kernel runtime. A further interesting fact is that although form-all-states is NP-complete, an exponential behavior of this function could not be experimentally observed [13]. The runtime of form-all-states increases with more complex models, however, the runtime ratio remains nearly constant, even with complex models it does not exceed 20% of the kernel runtime.

## 3. QSIM computer architecture

The specialized computer architecture presented in this paper improves the runtime of the QSIM kernel by two strategies. First, the parallelism in the complex kernel functions constraint-filter and form-all-states is exploited. These functions are parallelized and mapped onto a multiprocessor system. Second, the less complex CCFs are accelerated by software to hardware migration, i.e., the CCFs are implemented in hardware. In the following sections, design considerations for this specialized computer architecture and a prototype implementation are described.

### *3.1. Design of the QSIM kernel multiprocessor*

#### *3.1.1. Constraint-filter*

The data dependence graph of the constraint-filter is shown in Fig. 3. In this graph, the nodes represent the individual functions of the constraint-filter, and the arcs represent input and output data of these functions. Each tuple-filter $\mathbf{t\text{-}f}_i$ requires the set of possible values for all variables of constraint $i$, $pvals_i$. The output data of the functions $\mathbf{t\text{-}f}_i$ are the sets of locally consistent tuples, $tuples_i$. The Waltz-filter **W-f** requires all $tuples_i$ as input data and returns the restricted sets of tuples, $tuples'_i$ as output data. This data dependence analysis reveals a high parallelism within the constraint-filter. All tuple-filter functions are independent of each other and can be executed in parallel.

Fig. 4 presents the logical structure of the constraint-filter. It consists of a set of tasks and communication links. The tasks are partitioned into two groups — a *master* task and a set of *slave* (tuple-filter) tasks. The master task is responsible for the transmission of the input data to all tuple-filter tasks, the reception of the tuple-filters' results and the execution of the Waltz-filter. Since all tuple-filters are independent of each other, no communication between the slaves is required. The maximum degree of parallelism for the constraint-filter is the number of constraints $C$ which is given by the simulation model. Therefore, the logical structure consists of $C$ slaves.

The logical structure forms a *star* with the master as the central element. However, in a star structure the master becomes a bottleneck as the number of slaves increases. This limits the scalability of the computer architecture. To achieve a scalable architecture, our multiprocessor system is connected in a *wide tree* topology which is a compromise between logical structure and scalability. In a wide tree, each processing element has a constant node degree and, hence, a fixed number of communication

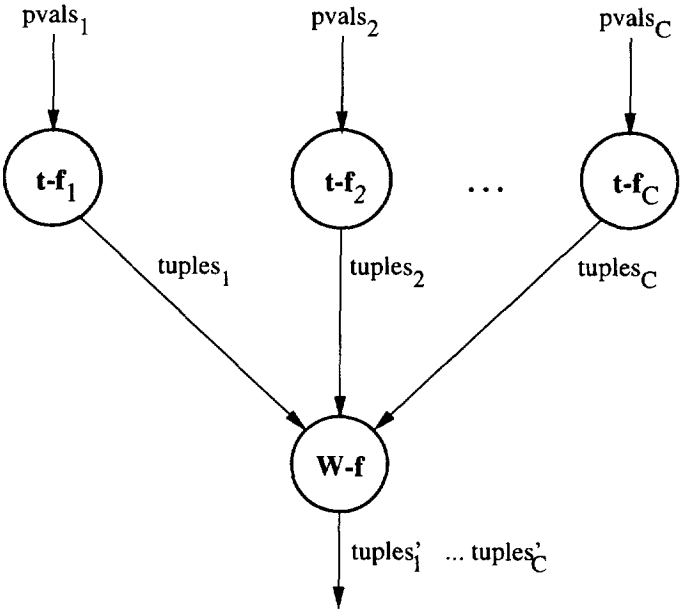


Fig. 3. Data dependence graph of the constraint-filter.

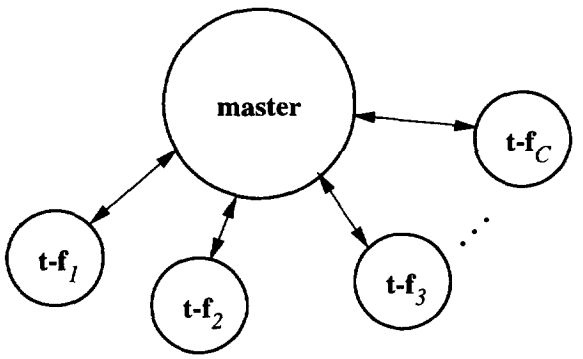


Fig. 4. Logical structure of the constraint-filter.

links. The root node of the tree corresponds to the master task; all other nodes correspond to the slaves of the logical structure.

The $C+1$ tasks of the logical structure are mapped onto the processing elements of the wide tree architecture by a scheduling algorithm. For the design of this scheduling algorithm the following issues must be considered:

- The scheduling of the slave tasks is performed at the beginning of the QSIM kernel execution for two reasons. First, the actual number of slave tasks is not known before the kernel execution. This is because QSIM may change the set of active constraints of the model during simulation. Second, also the input data for

the slave tasks are not known before the kernel execution. These input data are required to estimate the execution times of the individual slave tasks. These estimations are utilized by the scheduling algorithm.

- Only non-preemptive scheduling algorithms are considered, because all slave tasks are independent of each other.
- The execution time of the scheduling algorithm itself must be as short as possible because scheduling is a sequential part of the parallel constraint-filter implementation.

A *list scheduling* algorithm [4] is applied for the parallel implementation of the constraint-filter. This scheduling algorithm (i) uses the estimated execution times to improve the schedule, (ii) guarantees a worst-case completion time of all tasks and (iii) is only slightly slower than a simple task attraction scheduling algorithm.

### *3.1.2. Form-all-states*

The kernel function form-all-states solves a CSP by a backtracking algorithm. A big search space has to be processed by a depth-first search to find all solutions of the CSP. Contrary to the constraint-filter, there is no obvious parallelization given by the function hierarchy of form-all-states. For a parallel implementation of form-all-states, the CSP must be partitioned artificially. A *parallel-agent-based* (*PAB*) strategy [8] is used for the parallelization of the CSP in our QSIM architecture. The basic idea of PAB is to partition the overall search-space into smaller independent subspaces which can be solved by any sequential CSP algorithm. Therefore, the parallel form-all-states algorithm consists of three consecutive steps:

(1) Partitioning of the overall search space into independent subproblems.
(2) Solving the subproblems by a sequential CSP algorithm in parallel.
(3) Merging all partial results of the subproblems to the overall result.

The partitioning step is essential for the performance of the parallel implementation. A *variable-based partitioning* (*VBP*) heuristic [12] is used to partition the search space of QSIM CSPs.

The logical structure of the parallel form-all-states algorithm is similar to the logical structure of the constraint-filter. Due to the PAB strategy, also a master–slave structure is derived. The master task is responsible for the generation of the subproblems, their transmission to the slave tasks and the union of the partial results to the overall result. The maximum degree of parallelism is determined by the number of generated subproblems. The number of generated subproblems are influenced by the simulation model and the partitioning heuristic. The same architectural considerations as for the constraint-filter are valid for form-all-states.

Similar to the parallel constraint-filter, the tasks of the parallel form-all-states implementation are scheduled at the QSIM kernel execution time. However, due to the irregular behavior of the backtracking algorithm, the execution times of the slave tasks cannot be determined in advance. Just the worst-case execution times are known, and these times differ normally from the actual execution times by orders of magnitude. To balance the slave tasks as best as possible statically, *task attraction*

*scheduling* is applied. Whenever a processing element is idle, the next slave task is scheduled to this processing element.

### 3.2. Design of the CCF coprocessors

The CCFs are executed on specialized coprocessors. This section presents the coprocessor design on the example of the MULT-CCF coprocessor. The MULT-CCF is one of the most complex CCFs; CCFs for other constraint types are less complex, but very similar in structure.

Fig. 5 shows the dataflow diagram for the MULT-CCF. Input data are the three qualitative values $qval_1$, $qval_2$ and $qval_3$ and a list of corresponding value tuples, *cval_tuple*[*i*]. Corresponding value tuples are tuples which are known to be consistent for a particular constraint. In QSIM, corresponding value tuples can be part of the initial values or they can be created during simulation. Output data of the MULT-CCF is the boolean value *result*, which indicates whether the tuple of *qvals* is consistent or not.

An analysis of the MULT-CCF reveals, that this CCF can be partitioned into several subfunctions. The partitioned MULT-CCF is shown in Fig. 6. Subfunction SF1, *value check*, tests the signs and directions of change of the input values. Subfunction SF2, *infvalue check*, tests relations between infinite and zero input values. Subfunction SF3, *cval check*, tests the input values against all tuples from the list of corresponding value tuples. SF1 to SF3 form the functionality of the MULT-CCF. SF4 performs a logical AND operation on the partial results of SF1

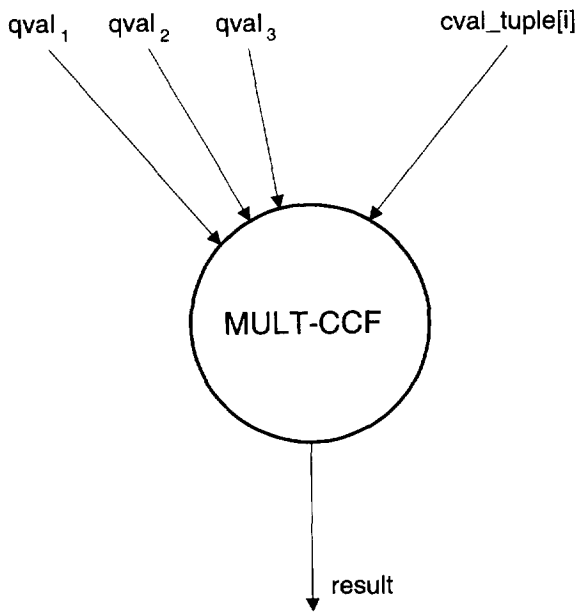


Fig. 5. Dataflow diagram of the MULT-CCF.

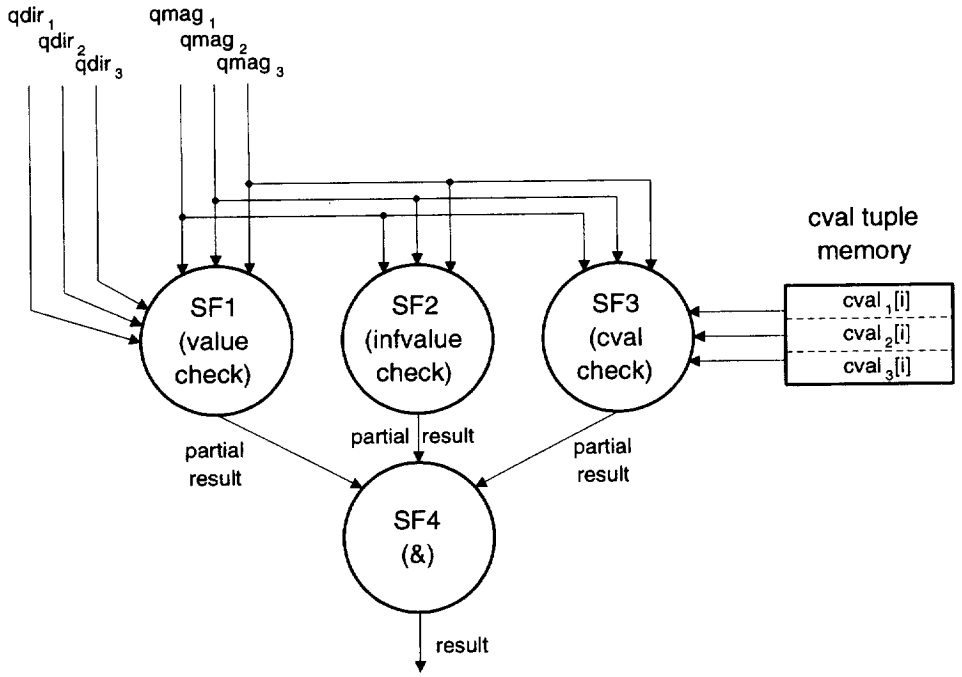


Fig. 6. Partitioned MULT-CCF. The corresponding value tuples are stored in a memory inside the MULT-CCF.

to SF3. The list of corresponding value tuples grows monotonically during simulation. New tuples of corresponding values are created by a global filter outside the QSIM kernel. This is a very rare process compared to the CCF executions. Therefore, the list of corresponding value tuples can be considered as static. This motivates to store the data *cval_tuple*[*i*] in an internal memory of the MULT-CCF. This is indicated in Fig. 6.

The specialized coprocessor implementing the functionality of the MULT-CCF is designed at the gate- and register-level to obtain maximum performance. The main features of the design are:

- Exploitation of parallelism. Parallelism is exploited at two levels. First, it can be taken from Fig. 6 that the subfunctions SF1, SF2 and SF3 are dataflow-independent. Therefore, they can be executed in parallel. SF3 actually implements an iteration over the list of corresponding value tuples. These iterations are also mutually dataflow-independent and can be executed in parallel. For this parallel execution of SF3 iterations, several architectural variants are proposed, including pipelining and array processing techniques [10]. SF4 is implemented similar to the *short-circuit-evaluation* in software, i.e., whenever one of the subfunctions SF1 to SF3 returns a negative result, SF4 aborts the entire calculation and generates the overall result. Second, the parallelism inside the subfunctions SF1 to SF3 is exploited by means

of wide application-specific data-paths and by transforming control-flow dominated code, i.e., nested case-statements, into table-lookups.

- Use of optimized data types. The required data types are optimized concerning the number of bits to represent the data type and the coding for an efficient execution of SF1 to SF4.
- Use of a customized memory architecture. The list of corresponding value tuples is stored in an internal memory of the coprocessor. This memory is split into three blocks (see Fig. 6) to access a whole tuple of corresponding values in one read cycle. Further, this customized memory operates in an auto-increment, circular addressing mode.

The coprocessor design contains functional blocks for SF1 to SF4, the internal memory, an I/O controller, and a function controller [3]. The I/O controller establishes communication to a host processor via two separate communication channels, which enables simultaneous input and output operations. The function controller decodes the instructions and controls the operation of all other functional blocks of the coprocessor. Three instructions are defined for the MULT-CCF coprocessor. Two instructions update the internal memory; the third instruction actually executes the MULT-CCF.

### *3.3. Prototype implementation*

A prototype of the overall heterogeneous multiprocessor architecture is shown in Fig. 7. The digital signal processor TMS320C40 was chosen as processing element because of its high I/O performance and its 6 independent communication channels [15]. Software is developed in 'C' under the distributed real-time operating system Virtuoso [18], which supports a portable and flexible software design. The specialized CCF coprocessors are implemented on field programmable gate arrays (FPGAs).

## 4. Experimental evaluation

### *4.1. QSIM kernel multiprocessor*

The experimental evaluation of the QSIM kernel multiprocessor is based on a comparison of the execution times of the sequential implementation, $t_{seq}$, and the parallel implementation using $n$ processing elements, $t_{par}(n)$. With these execution times, the speedup $S(n)=t_{seq}/t_{par}(n)$ can be determined. The execution times are measured on a prototype of the QSIM kernel multiprocessor using a 32 bit timer of the TMS320C40 with a resolution of 80 ns. The sequential implementation of the QSIM kernel is executed on the root node of the QSIM kernel multiprocessor. In the next two sections, the QSIM kernel multiprocessor is evaluated independently for both kernel functions constraint-filter and form-all-states.

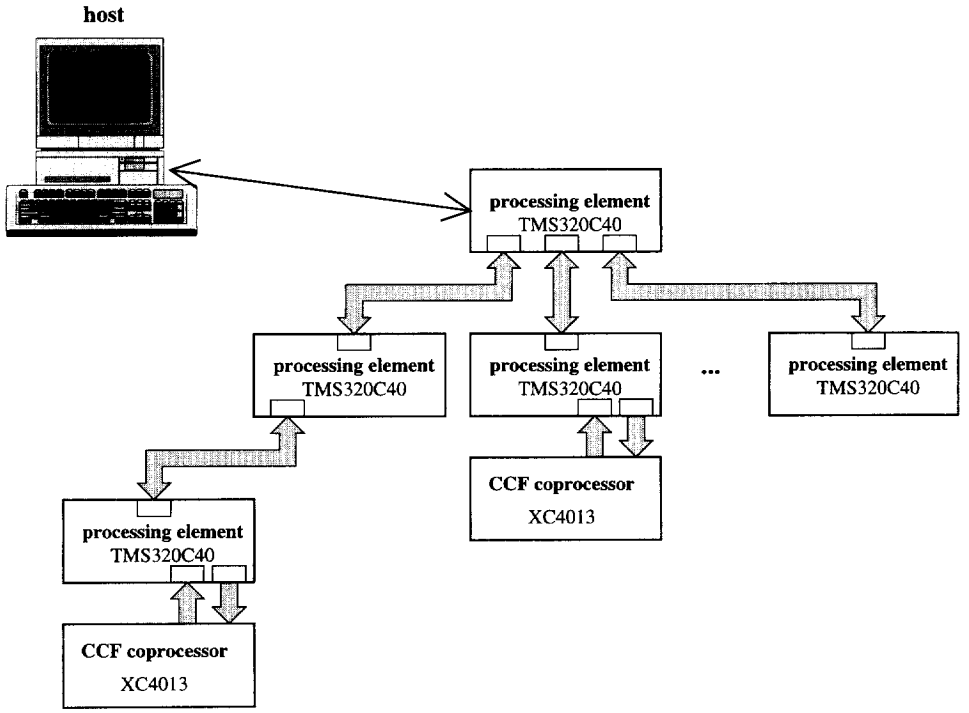


Fig. 7. Example of the overall architecture for the QSIM kernel multiprocessor. The processing elements are connected in a wide tree structure. Some processing elements are equipped with CCF-coprocessors.

#### *4.1.1. Constraint-filter*

The parallel implementation of the constraint-filter is evaluated using three different sets of input data. Two sets are derived from the QSIM models RCS (48 constraints) [5] and QSEA (21 constraints) and one set is constructed artificially (M1). In the data sets RCS and QSEA, the numbers and types of the tuple-filter tasks as well as the numbers of tuples which must be checked vary. The data set M1 consists of 30 tuple-filters of type MULT; each tuple-filter must check 64 tuples.

Fig. 8 presents the speedups of the parallel implementation of the constraint-filter using 1, 2 and 3 slave processing elements. The best speedup is achieved with input data set M1 because from all data sets M1 has the longest execution times of the individual tuple-filter tasks. At this complex data set, the overhead introduced by the parallel implementation, e.g., communication times, has only a small influence on the overall execution time.

#### *4.1.2. Form-all-states*

The parallel implementation of form-all-states is evaluated using CSPs derived from the simulation of the QSIM models RCS and QSEA. Fig. 9 presents the speedups of the parallel form-all-states implementation using up to 7 slave processors. Parallel execution of the RCS model reveals a superlinear speedup using one

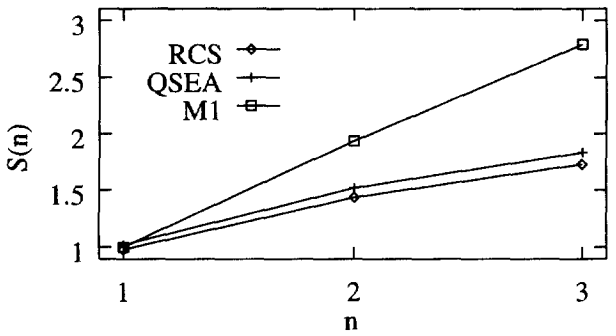


Fig. 8. Speedup $S(n)$ of the parallel implementation of the constraint-filter for the models RCS, QSEA and M1 using $n=1...3$ slave processors.

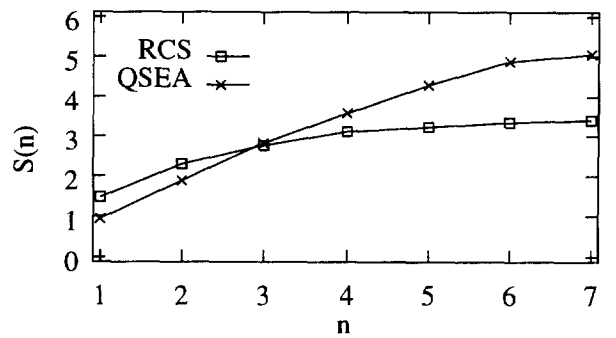


Fig. 9. Speedup $S(n)$ of the parallel implementation of form-all-states for the models RCS and QSEA using $n=1...7$ slave processors.

and two slave processors. This occurs because the partitioning algorithm discards many inconsistent subproblems, and the total execution time of the remaining consistent subproblems is smaller than the execution time of the unpartitioned problem.

### 4.2. CCF coprocessor

The experimental evaluation of the CCF coprocessors is based on a comparison of the execution times of the software CCF, $t_{sw}$, and the execution times of the pair host and coprocessor, $t_{hw}$, for the CCF. From these execution times, the overall speedup of the coprocessor, $S_{tot}=t_{sw}/t_{hw}$, is calculated. This overall speedup respects also the required communication between the host and the coprocessor. The coprocessor execution times are measured on a MULT-CCF coprocessor prototype with sequential execution of SF3 iterations. This coprocessor is implemented on an FPGA of type Xilinx XC4013 and is operated at a clock frequency of 15 MHz [3]. The measured execution times and the calculated speedups are subdivided into 6 cases according to the subfunction which causes termination of the MULT-CCF. For the

short-circuit-evaluation of the software CCF, the execution order SF1, SF2, SF3 is assumed. Following cases are differentiated:

case 1: execution of SF1
case 2: execution of SF1, SF2
case 3: execution of SF1, SF2, $SF3_1$
case 4: execution of SF1, SF2, $SF3_1$, $SF3_2$
case 5: execution of SF1, SF2, $SF3_1$, $SF3_2$, $SF3_3$
case 6: execution of SF1, SF2, $SF3_1$, $SF3_2$, $SF3_3$, $SF3_4$

For example, case 4 represents the sequential execution of SF1, SF2 and two iterations of SF3. The six cases above reflect the most likely situations. In Fig. 10 the overall speedup of the coprocessor, $S_{tot}$, is presented dependent on the execution cases.

## 5. Conclusion

In this paper we have presented the design and the prototype implementation of a specialized computer architecture for the qualitative simulator QSIM. The experimental results proved that a significant speedup can be achieved with this computer architecture. The performance of QSIM is improved by the two strategies paralleliza-

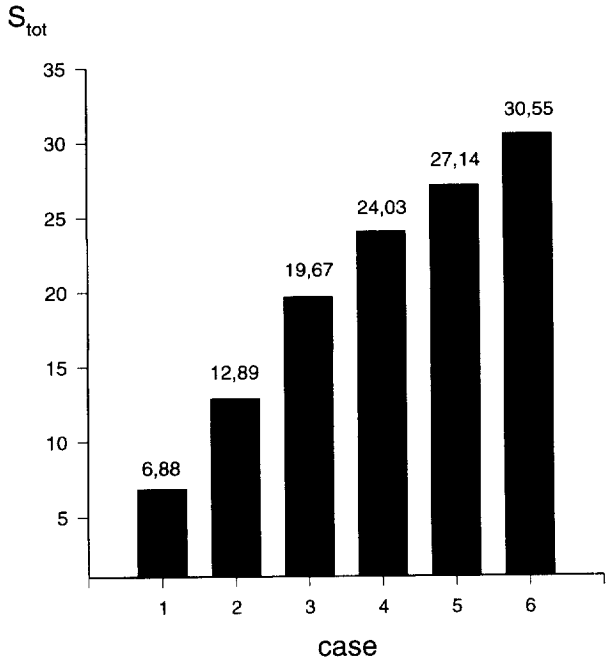


Fig. 10. Speedup $S_{tot}$ of the MULT-CCF coprocessor dependent on 6 different execution cases.

tion and software to hardware migration. With the first strategy, speedups of up to 2.79 using three slave processors for the constraint-filter and 5.06 using 7 slave processors for form-all-states are achieved. However, the actual speedup of the parallel constraint-filter depends on the number and execution times of the tuple-filters. For the parallel implementation of form-all-states, the actual speedup is limited mainly by the number of generated subproblems and the execution times of these subproblems. The second strategy results in specialized coprocessors with speedups of up to 30.55, even on an FPGA-based prototype.

In general, our computer architecture exploits the parallelism in the QSIM kernel functions. The degree of parallelism depends strongly on the input simulation model. More complex models, i.e., models with many constraints and variables, lead to higher degrees of parallelism. Due to the high complexities of models for 'real-world' problems, the efficiency of existing qualitative simulators is a significant barrier to their application. The presented special-purpose computer architecture will help to remove this barrier.

Further work will include (i) the more extensive evaluation of the overall computer architecture, (ii) the implementation of a larger heterogeneous multiprocessor system and (iii) the integration of the presented computer architecture into fault diagnosis applications.

## References

[1] E. Davis, An engaging exploration of QSIM and its extensions, IEEE Expert. 9 (6) (1994) 70–71. Book review

[2] D. Dvorak, B. Kuipers, Process monitoring and diagnosis: A model-based approach, IEEE Expert. 5 (3) (1991) 67–74.

[3] G. Friedl, M. Platzner, B. Rinner, A special-purpose coprocessor for qualitative simulation, in: Proc. of the 1st Int. EURO-PAR Conf., Stockholm, August 1995, pp. 695–698.

[4] R.I. Graham, Bounds on multiprocessing timing anomalies, SIAM J. Appl. Math. 17 (2) (1969) 416–429.

[5] H. Kay, A qualitative model of the space shuttle reaction control system, Tech. Rep. AI92-188, Artificial Intelligence Laboratory, University of Texas at Austin, 1992.

[6] B. Kuipers, Qualitative reasoning: Modeling and simulation with incomplete knowledge, Artificial Intelligence, MIT Press, 1994.

[7] F. Lackinger, W. Nejdl, Diamon: A model-based troubleshooter based on qualitative reasoning, IEEE Expert. 8 (1) (1993) 33–40.

[8] Q.P. Luo, P.G. Hendry, J.T. Buchanan, Strategies for distributed constraint satisfaction problems, in: Proc. 13th Int. DAI Workshop, Seattle, WA, 1994.

[9] A.K. Mackworth, Constraint satisfaction, in: S.C. Shapiro (Ed.), Encyclopedia of artificial intelligence, vol. 1, John Wiley&Sons, Inc., 1992, pp. 285–293.

[10] M. Platzner, Design, implementation and experimental evaluation of coprocessor architectures for fast qualitative simulation, Ph.D. thesis, Graz University of Technology, 1996.

[11] M. Platzner, B. Rinner, Improving performance of the qualitative simulator QSIM: Design and implementation of a specialized computer architecture, in: Proc. of the ISCA Int. Conf. on Parallel and Distributed Computing Systems, Orlando, 1995, pp. 494–501.

[12] M. Platzner, B. Rinner, R. Weiss, Exploiting parallelism in constraint satisfaction for qualitative simulation, J. UCS J. Univers. Comput. Sci. 1 (12) (1995) 811–820.

[13] B. Rinner, Konzepte zur Parallelisierung des qualitativen Simulators QSIM, M.Sc. thesis, Institute for Technical Informatics, Graz University of Technology, October 1993.
[14] P. Seifter, C. Steger, R. Weiss, A distributed simulation architecture used as a reference model in a distributed real-time expert-system, in: Proc. of the ISMM Int. Conf. on Parallel and Distributed Computing and Systems, Washington, 1991, pp. 68–72.
[15] R. Simar, P. Koeppen, J. Leach, S. Marshall, D. Francis, G. Mekras, J. Rosenstrauch, S. Anderson, Floating-Point Processors Join Forces in Parallel Processing Architectures, IEEE Micro, 1992, pp. 60–69.
[16] C. Steger, R. Weiss, A model-based real-time fault diagnosis system in technical processes, in: Proc. of the 10th Int. Conf. on Applications of Artificial Intelligence in Engineering, Udine, 1995, pp. 145–152.
[17] S. Subramanian, R.J. Mooney, Multiple-fault diagnosis using general qualitative models with behavioral modes, in: Int. Joint Conf. on Artificial Intelligence, 1995.
[18] E. Verhulst, Virtuoso: A virtual single processor programming system for distributed real-time applications, Microprocess. Microprogram. 40 (1994) 103–115.