Improving Performance of the Qualitative Simulator QSIM — Design and Implementation of a Specialized Computer Architecture[*]

Marco Platzner
Graz University of Technology
Institute for Technical Informatics
A-8010 Graz, Austria

Bernhard Rinner
Graz University of Technology
Institute for Technical Informatics
A-8010 Graz, Austria

Abstract

Qualitative simulation is a new and challenging simulation paradigm. QSim, the widely-used algorithm for qualitative simulation has been developed by Kuipers at UT Austin. A drawback of current QSim-implementations is poor execution speed. In our research project a special-purpose computer architecture for QSim is developed to increase the performance. Two approaches are considered to improve the performance. Complex functions are parallelized and mapped onto a multiprocessor system. Less complex functions are directly implemented in hardware. These functions are executed on specialized coprocessors. The prototype implementation of this computer architecture is based on digital signal processors TMS320C40 and field programmable gate arrays (Xilinx). This paper presents first experimental results of this research project.

keywords: special-purpose computer architecture, parallel qualitative simulation, multi-DSP TMS320C40, FPGA specialized coprocessor

1 Introduction

QSIM, the widely-used algorithm for qualitative simulation, has been developed by Kuipers [6]. Qualitative simulation is a new and challenging simulation paradigm. QSIM is mainly used in applications where a accurate description of a system is not required or even not known. Major areas of qualitative simulation applications are design, monitoring, and fault-diagnosis. A drawback of current QSIM implementations is poor execution speed. In our research project [10] [11] a special-purpose computer architecture for QSIM is developed to improve the performance.

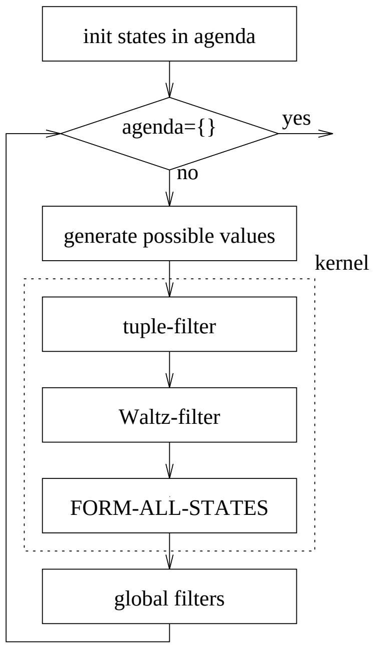

Figure 1: Flow chart of QSIM.

In qualitative simulation systems are modeled on a higher level of abstraction than in other simulation paradigms, like continuous simulation. *Variables* and *constraints* are basic components of a qualitative model. 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 varia-

[*]This project is partially supported by the Austrian National Science Foundation *Fonds zur Förderung der wissenschaftlichen Forschung* under grant number P10411-MAT.

bles 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. An assignment of possible values of all variables of a given constraint is called *tuple*. The tuple-filter rejects all tuples of an individual constraint which do not satisfy the conditions of this constraint. The Waltz-filter discards additional tuples by detecting inconsistencies between adjacent constraints. Constraints are adjacent if they share a variable. The final kernel function (FORM-ALL-STATES) finds consistent combinations of tuples of all constraints. 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 many of them are optional extensions of QSim.

This paper describes the current state of our research project and presents experimental results of first prototypes of components of the overall computer architecture. Chapter 2 gives an overview of QSim kernel functions and shows a runtime analysis. In Chapter 3 top-level kernel functions are analyzed and the speedup of a parallel version of FORM-ALL-STATES is estimated. Chapter 4 presents an analysis of low-level kernel functions and experimental results of an implemented coprocessor for these functions. Some conclusions are given in the final chapter.

2 Simulator Kernel

The qualitative simulator QSim is a very complex algorithm and has many optional features. Design considerations of this specialized computer architecture are restricted to kernel functions. Kernel functions are essential in calculating one simulation step, and they normally dominate the runtime of the complete simulator. Several model-based fault diagnosis and monitoring systems use qualitative simulation [1] [7]. These systems do not require the functionality of the whole simulator. However, QSim kernel functions are required.

Figure 2 presents an overview of the kernel functions. These functions are hierarchically structured and are analyzed in the following two chapters. The *constraint check functions (CCFs)* are primitive kernel

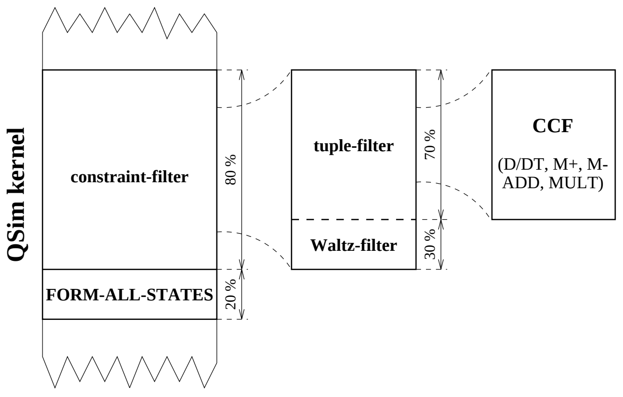

Figure 2: Runtime analysis of the kernel. Kernel functions are hierarchically structured and their runtimes are informally presented with regard to the runtime of the calling function.

functions but they dominate the overall kernel runtime. These functions are called by the *tuple-filter*. For each constraint of the input model one tuple-filter is required. The *constraint-filter* is generated by all tuple-filters and the *Waltz-filter*, which is used for efficiency reasons. The final kernel function is called *FORM-ALL-STATES*.

The presented runtime ratios in Figure 2 are extracted from various runtime measurements of a QSim system implemented on a TI Explorer LISP workstation. Many input 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 kernel functions require more than 50 % of the overall runtime. An important fact is that this percentage is positively correlated to the complexity of the model. Qualitative models for serious technical processes usually have many constraints and variables [5]. For these models kernel functions consume definitely more than 50 % of the overall runtime.

According to the complexity of the kernel functions different approaches to increase the performance are considered. Complex kernel functions (like constraint-filter and FORM-ALL-STATES) are parallelized and mapped onto a multiprocessor system. Less complex functions (CCFs) are HW-implemented using FPGAs. These runtime intensive functions are executed on specialized coprocessors.

3 QSim Multiprocessor

As depicted in Figure 2, the QSim kernel mainly consists of two consecutive functions — the *constraint-filter* and *FORM-ALL-STATES*. In this chapter we analyze these functions and present some implementation considerations and experimental results from parallelizing and mapping these functions onto a multiprocessor system.

3.1 Constraint-filter

The constraint-filter is generated by mutually independent functions (tuple-filters) and the Waltz-filter. The number of tuple-filters is variable and depends on the input simulation model. For each constraint of the input simulation model one tuple-filter has to be executed. Waltz-filtering can be considered as a preprocessing step for the successive function of the QSim kernel (FORM-ALL-STATES). It is used for efficiency reasons to reduce the search space for FORM-ALL-STATES as soon as possible. An even better improvement is achieved by interleaving the tuple-filter with the Waltz-filter. This is called incremental Waltz-filtering which possibly eliminates input data for unprocessed tuple-filters. The pseudocode of sequential and incremental Waltz-filtering is shown in Figure 3.

```
FOR all constraints $c_i$ DO        FOR all constraints $c_i$ DO
    tuple-filter($c_i$)                 tuple-filter($c_i$)
ENDFOR                                  waltz-filter()
waltz-filter()                      ENDFOR
```

Figure 3: Constraint-filter pseudocode with sequential (left) and incremental (right) Waltz-filtering.

Parallelization of tuple-filter with sequential Waltz-filtering is trivial. Tuple-filters are executed on individual processors. After all tuple-filter results have been received sequential Waltz-filtering is performed. Since the Waltz-filter requires global access to the results of the tuple-filters, incremental Waltz-filter disables parallelization of the tuple-filter. However, a variant of incremental Waltz-filtering can be used, if there are more tuple-filters than processing elements. Whenever results from tuple-filters are received, scheduling is stopped and Waltz-filtering is executed. After completion of the Waltz-filter scheduling is continued.

3.2 FORM-ALL-STATES

FORM-ALL-STATES is actually a backtracking algorithm to solve a *constraint satisfaction problem (CSP)* [9]. A big search space has to be processed with a depth-first search to find all solutions of the CSP. Many parallel algorithms for constraint satisfaction are known in literature. A classification of the most common parallel algorithms can be found in [8]. We use a *parallel-agent-based (PAB)* strategy for our application. The basic idea of PAB is to partition the overall search-space into smaller independent subspaces, which can be solved with any sequential CSP-algorithm. The overall result is formed by merging the results of the subspaces. Hence, the overall CSP is partitioned into independent smaller sub-CSPs.

Partitioning the complete search-space is essential to achieve good parallel performance. Due to redundancies in the subproblems, the overall runtime to solve all sub-CSPs can be longer than the runtime to solve the complete CSP. An efficient partitioning keeps this overall runtime small. It is also important to generate equally complex sub-CSPs. Big differences in the runtime of individual sub-CSPs can lead to poor load-balance. When tasks with long runtimes are scheduled last some processors can get idle.

3.3 Implementation and Experimental Results

Analysis of both kernel functions (constraint-filter and FORM-ALL-STATES) has brought up the same logical structure of the parallel algorithms. The tasks are logically connected in a *star* structure. The master task, which is responsible for distributing tasks and receiving results, is the center of the star. The multiprocessor system is implemented in a *wide tree* structure, which is a compromise between scalability and logical structure. Hence, the processing elements of the multiprocessor system should be equipped with many communication ports. This requirement was one reason for choosing the digital signal processor TMS320C40 as processing element. It is equipped with six independent high performance communication ports. Multiprocessor trees with up to five children can be built with this processor.

The number of tasks is not known at compile time. Thus, dynamic mapping and dynamic scheduling of these tasks is required. Implementation is based on the distributed real-time operating system *Virtuoso* [13]. Virtuoso eases a scalable design of the parallel algorithms and allows a flexible and portable implementation.

In this paper we present only experimental results from parallelizing FORM-ALL-STATES [12]. Several partitioning heuristics are evaluated and compared by a speedup estimation. The evaluation is based on

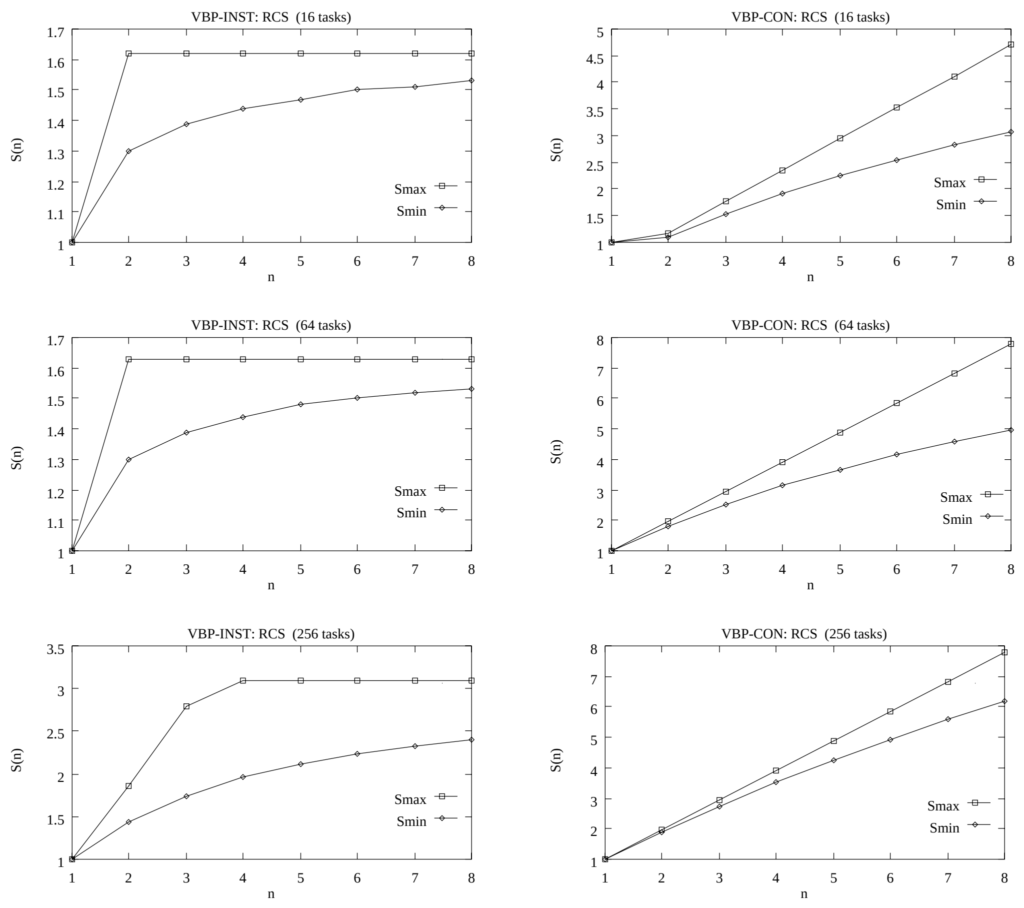

Figure 4: Speedup estimation for parallel FORM-ALL-STATES. Speedup limits (S_{max}, S_{min}) for two partitioning heuristics (VBP-INST and VBP-CON) are shown in the left and right column plots. The speedup limits for different numbers of partitioned sub-CSPs (16, 64, and 256 tasks) are shown. Especially for complex models, like RCS[5], VBP-CON performs better than VBP-INST.

many CSPs traced from QSim simulations. The partitioning heuristics are evaluated using these CSPs. The partitioned sub-CSPs are solved sequentially on a single TMS320C40, where runtimes are measured. The most interesting runtimes are the overall runtime t_o, which is the sum of the runtimes of all sub-CSPs, the maximum runtime of all sub-CSPs t_{max}, and the sequential runtime of the unpartitioned CSP t_{seq}.

These runtimes are used to estimate the speedup of the parallel algorithm. Communication times are not considered for this estimation and simple task attraction is assumed to schedule tasks to free processors. The speedup is defined as $S(n) = t_{seq}/t_{par}$, where t_{par} denotes the runtime using n processors. An upper limit (worst-case) and a lower limit (best-case) for t_{par} are given using t_o, t_{max}, and n. Worst-case runtime of the parallel algorithm can be given as

$$t_{par} = \frac{t_o - t_{max}}{n} + t_{max}$$

and best-case runtime can be estimated as

$$t_{par} = \begin{cases} \frac{t_o}{n} & if \quad n < \lceil \frac{t_o}{t_{max}} \rceil \\ t_{max} & otherwise \end{cases}$$

A comparison of two partitioning heuristics is presented in Figure 4. A detailed description of the partitioning heuristics can be found in [12]. The most successful partitioning heuristic is VBP (variable-based-partitioning) with the two variants VBP-INST and VBP-CON. This partitioning strategy is based on the variables of the input simulation model. The difference between the variants is the order in which variables are processed. It turns out that in most cases VBP-CON performs better than VBP-INST. VBP-CON results in a linear speedup for worst- and best-case estimation. A further interesting point is the number of generated sub-CSPs. Three cases are considered — the complete CSP is partitioned into at most 16, at most 64, and at most 256 sub-CSPs. The corresponding speedup limits are also presented in Figure 4. Speedup increases with the number of generated tasks. However, the more tasks are generated the more overall communication time is required and the speedup of highly partitioned CSPs can be lost. Best results are expected with VBP-CON and a medium number of tasks.

4 QSim Coprocessor

4.1 Tuple–filter

The tuple-filter is executed for each constraint of the simulation model. Its task is to check each combination of possible values (pvals) for the particular constraint and to discard the combination if it violates the rules of qualitative simulation. This check is done by the so called constraint check function (CCF).

Two important attributes of constraint types are *arity* and the existence of *corresponding values* (cvals). For example, the arity of the constraint types D/DT, M^+, and M^- is 2, the constraint types ADD and MULT are ternary. The maximum number of pvals per variable is limited by 4 [1]. This leads to at most 16 checks for binary constraints and to at most 64 checks for ternary constraints, respectively. Cvals are tuples which are known to be correct. Most constraint types have an associated set of cvals. This set can grow monotonically during simulation. However, the creation of a new cval-tuple is a very rare process compared to the execution of the CCF. Therefore, cvals are considered as constants rather than variables in the context of the tuple-filter.

Figure 5 shows the tuple-filter pseudocode for a ternary constraint. The indices i, j, and k scan over the pvals of the corresponding parameters, the constants i_{max}, j_{max}, and k_{max} are bounded by 4. Input data for the constraint check function are the possible values $p1(i)$, $p2(j)$, and $p3(k)$.

To improve the performance of the tuple-filter it is of utmost importance to accelerate the constraint check function. The following sections in this chapter present the analysis, hardware implementation, and experimental evaluation of one of the most complex check functions, the Mult–CCF. CCFs for other constraint types are very similar in structure, but less complex.

[1] Calculating initial states from an incomplete state description can lead to more the 4 pvals of an individual variable.

```
FOR $i = 1$ TO $i_{max}$ DO
    FOR $j = 1$ TO $j_{max}$ DO
        FOR $k = 1$ TO $k_{max}$ DO
            $result(i, j, k) = ccf(p1(i), p2(j), p3(k))$
        ENDFOR
    ENDFOR
ENDFOR
```

Figure 5: Tuple–filter pseudocode for a ternary constraint.

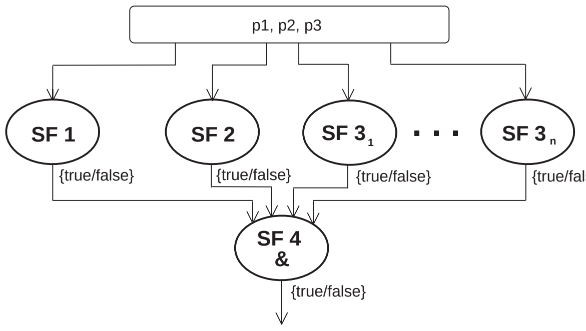

Figure 6: Dataflow diagram for the Mult–CCF.

4.2 Mult–CCF

Figure 6 presents the dataflow diagram for the Mult–CCF. The function is partitioned into four subfunctions, SF1 to SF4. Subfunction SF3 consists of n iterations, where n denotes the number of cvals. All subfunctions produce a boolean result.

Two important facts can be taken from Figure 6. First, the subfunctions SF1, SF2, and all iterations of SF3 are dataflow–independent. Therefore they can be executed in parallel. Second, subfunction SF4 can be implemented as short-circuit-evaluation, i.e. whenever one of the subfunctions SF1 to SF3 returns a negative result, the entire calculation is aborted. A negative result is returned to the tuple–filter, which discards the current pval combination.

4.3 Implementation and Experimental Results

The Mult–CCF was designed at the gate- and register level and implemented directly in hardware to obtain maximum execution speed [2]. Figure 7 shows the block diagram of the Mult–CCF coprocessor. Main

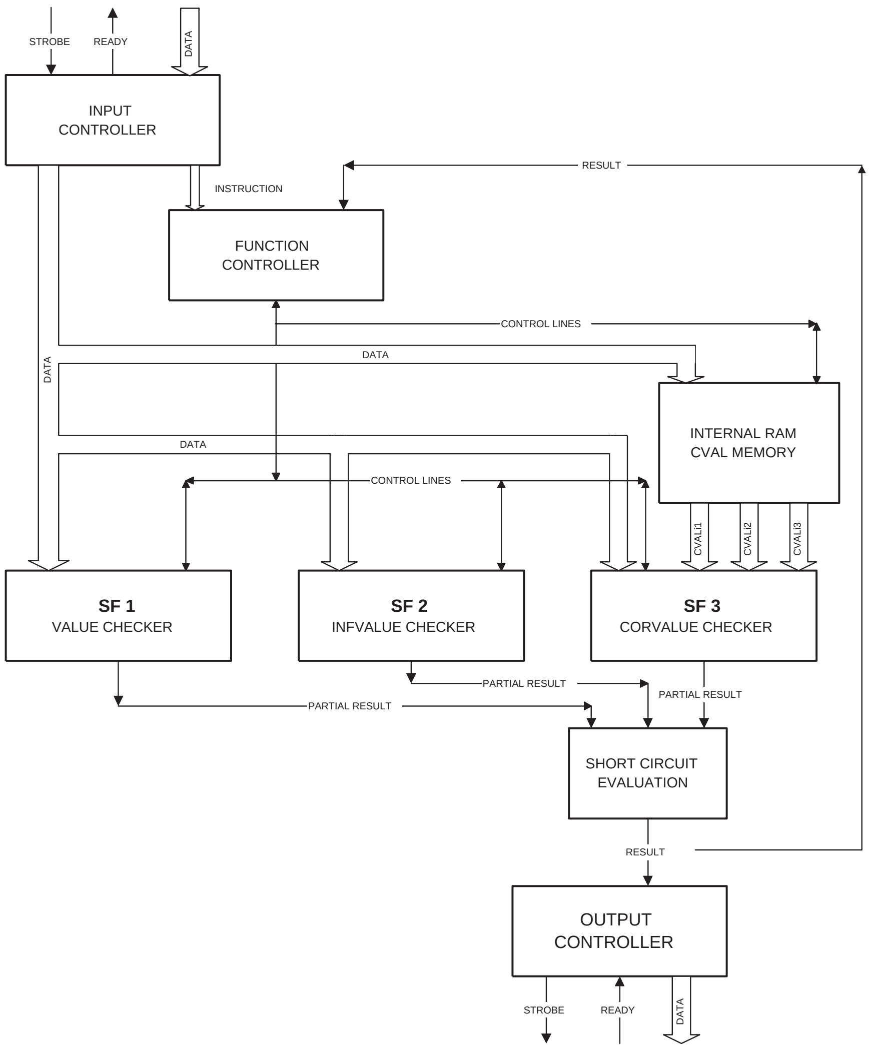

Figure 7: Block diagram of the Mult–CCF coprocessor. The input and output controller connect the coprocessor to the host processor via two 8 bit data buses and four handshake lines (2xSTROBE, 2xREADY). For simplification control lines between input/output controller and function controller are not shown in this diagram.

features of the design are (i) optimized data structures for the application QSim, (ii) operations using maximum parallelism, and (iii) customized memory architecture for parallel access.

The input and output controller establish communication to the host processor (digital signal processor TMS320C40) via two separate communication channels. These unidirectional connections ease the I/O controller design of the coprocessor and allow parallel input- and output–operations. The operands and the instruction code (2 bit) are packed into a 32 bit word for communication from host to the coprocessor. The function controller decodes the instruction, supplies the coprocessor's function blocks with data, and controls the operation. Also short-circuit-evaluation is handled by the function controller. The blocks SF1 to SF3 correspond directly to the subfunctions SF1 to SF3 in Figure 6. The iterations of SF3 are executed sequentially in this design. For the cvals a fast internal RAM is used. This memory is partitioned and allows access to a whole cval-tuple in one memory read–cycle. A detailed description of the design and implementation of the Mult–CCF coprocessor can be found in [3]. Three instructions are defined for the coprocessor:

- `EXECUTE_CFF`
- `RESET_CVAL_MEMORY`

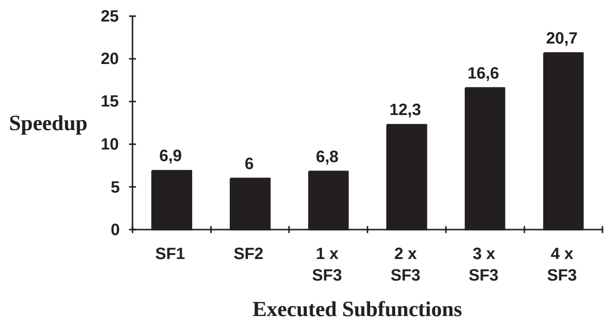

Figure 8: Speedup of the Mult–CCF coprocessor compared to a CCF implemented in software on the DSP TMS320C40.

- `LOAD_CVAL_MEMORY`

The most frequently used instruction is `EXECUTE_CCF`. The other two instructions establish operations with the internal memory of the coprocessor. The runtime for an `EXECUTE_CCF` operation can be divided into three phases: (i) communication from host to coprocessor (t_{in}), (ii) operation (t_{op}), and (iii) communication from coprocessor to host (t_{out}). The number of coprocessor clock cycles for $t_{in} = 7$, for $t_{out} = 8$, and for t_{op} the runtime depends on i, the number of cvals:

$$t_{op} = \begin{cases} 3 & for\ i < 2 \\ 2i+1 & for\ i \geq 2 \end{cases}$$

To evaluate the performance of the Mult-CCF coprocessor we compare it to a Mult–CCF software implementation on the host processor. We have to distinguish several cases according to the subfunction, which causes termination of the Mult–CCF. In Figure 8 six cases are shown. Computation is finished after SF1, SF2, or 1 to 4 iterations of SF3. Although the number of cvals is unbounded in general, more than 4 cvals are very unlikely [4]. Figure 8 shows a runtime improvement with the Mult–CCF coprocessor of up to factor 20.

The numbers in Figure 8 were measured on a coprocessor implementation in one FPGA (Xilinx XC4013) at a clock frequency of 15 MHz. The software reference was executed on a TMS320C40 running at a clock frequency of 32 MHz. The TMS320C40's instruction cycle time is two clock cycles. Therefore, the TMS320C40 and the CCF coprocessor are actually running at nearly same clock rates (16 MHz vs. 15 MHz). Hence, the given speedup evaluates directly our hardware design — no clock frequency transformation is necessary.

As the number of clock cycles for t_{in}, t_{op}, and t_{out} is almost equal, the three phases can be overlapped. This leads only to a minor increase of t_{op}. However, the runtime for the tuple-filter is further improved by a factor of up to 3 resulting in a maximum speedup for the tuple–filter of factor 60.

5 Conclusion, Further Work

We presented the design and implementation of a specialized computer architecture. We shortly introduced the qualitative simulator QSim and analyzed the simulator kernel. Runtime measurements were used to identify the most runtime intensive functions. It turned out that two approaches to increase performance should be used: Parallelization of complex functions and direct HW implementation of less complex but frequently used functions. These approaches were discussed in more detail in chapter 3 and chapter 4, respectively.

We already obtained first experimental results, some of them are presented in this paper. These first results are very important. They allow a rough performance estimation of the overall architecture. Obtaining experimental results on real simulation runs is the only useful evaluation method for this special computer architecture. This is due to following facts: (i) Since the algorithm QSim is very irregular and input–data sensitive, analytical performance predictions are not possible or at least very difficult. Predicting performance by simulation is too complex. (ii) The underlying hardware architecture can not easily be modeled at a satisfying level of detail.

Speedup factors as shown in chapter 3 and chapter 4 stress that a significant performance increase compared to QSim implementations on single–processor, general–purpose computers will be achieved.

Work, which partially is already going on and partially has to be done is divided into three sections:

QSim **multiprocessor** Implementation of the developed parallel functions constraint–filter and FORM-ALL-STATES on the multi–DSP system; evaluation of several alternatives concerning scheduling policies and partitioning heuristics

QSim **coprocessors** Implementation of coprocessors for other often used constraint types, e.g. ADD, M^+, M^-, D/DT

Overall architecture SW integration of the specialized coprocessors into the multi–DSP system; experimental evaluation of the QSIM machine with all features in its operational state

References

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

[2] Gerald Friedl, Marco Platzner, and Bernhard Rinner. A Special-Purpose Coprocessor for Qualitative Simulation. In *EURO-PAR'95 International Conference on Parallel Processing*, Stockholm, Sweden, August 1995.

[3] Gerald Friedl. Entwurf und FPGA-Implementierung eines Coprozessors für qualitative Simulation. Master's thesis, Institute for Technical Informatics, Graz University of Technology, 1995.

[4] Peter Hinterberger. QSim Tracedaten Filter. Technical report, Institute for Technical Informatics, Graz University of Technology, 1994.

[5] Herbert Kay. A qualitative model of the space shuttle reaction control system. Technical Report AI92-188, Artificial Intelligence Laboratory, University of Texas, September 1992.

[6] Benjamin Kuipers. *Qualitative Reasoning: Modeling and Simulation with Incomplete Knowledge.* Artificial Intelligence. MIT Press, 1994.

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

[8] Q.P. Luo, P.G. Hendry, and J.T. Buchanan. Strategies for Distributed Constraint Satisfaction Problems. In *Proceedings 13th International DAI Workshop*, Seattle, WA, 1994. DAI.

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

[10] Marco Platzner, Bernhard Rinner, and Reinhold Weiss. A Distributed Computer Architecture for Qualitative Simulation Based on a Multi-DSP and FPGAs. In *3rd Euromicro Workshop on Parallel and Distributed Processing*, pages 311–318, San Remo, January 1995. IEEE Computer Society Press.

[11] Marco Platzner, Bernhard Rinner, and Reinhold Weiss. Parallel Qualitative Simulation. In *EUROSIM'95 Simulation Congress*, Vienna, Austria, September 1995.

[12] Johannes Riedl. Parallele Algorithmen und Laufzeitmessungen für Constraint Satisfaction im qualitativen Simulator QSim. Master's thesis, Institute for Technical Informatics, Graz University of Technology, 1995.

[13] Eric Verhulst. Virtuoso: A virtual single processor programming system for distributed real-time applications. *Microprocessing and Microprogramming*, 40:103–115, 1994.