Toward Embedded Qualitative Simulation: *A Specialized Computer Architecture for QSim*

Marco Platzner, Swiss Federal Institute of Technology Zurich
Bernhard Rinner and Reinhold Weiss, Technical University Graz

THE SPECIALIZED COMPUTER ARCHITECTURE THESE AUTHORS HAVE PROTOTYPED DRAMATICALLY IMPROVES THE RUNTIME PERFORMANCE, SCALABILITY, AND PORTABILITY OF QSIM, A WELL-KNOWN QUALITATIVE SIMULATOR.

QUALITATIVE SIMULATION IS A key inference technique of model-based reasoning that has found use in such areas as monitoring, fault diagnosis, and design. To advance embedded applications of the qualitative simulator QSim, we have developed a special-purpose computer architecture designed to provide high performance, scalability, and increased portability to embedded computer platforms. To demonstrate our approach's suitability for embedded qualitative simulation, we have developed a prototype implementation on a heterogeneous multiprocessor system. As this article discusses, this prototype improves QSim's performance by two orders of magnitude.

Improving embedded qualitative simulation

After more than a decade of research, qualitative simulation is on the brink of being applied to real-world problems. Interest is shifting from pure research-oriented issues to more application-oriented ones.

Qualitative simulation involves deriving a dynamic physical system's behavior given only weak and incomplete information about it. Qualitative simulation entails modeling physical systems on a higher level of abstraction than with other simulation paradigms, such as continuous simulation, which model the physical system on a mathematical description in the form of differential equations. Qualitative simulation relies on a further abstraction of these differential equations—the qualitative differential equations. Qualitative simulation requires neither a complete structural description of the physical system nor a fully specified initial state. This technique excels at predicting all physically possible behaviors derivable from this incomplete knowledge.

Qualitative simulation plays an essential role in qualitative reasoning research and is one of the primary inference techniques in model-based reasoning. The goal here is to automate tasks that engineers, technicians, and scientists perform when understanding, designing, explaining, monitoring, and diagnosing physical systems. Many research projects demonstrate the applicability of qualitative simulation in these areas. Some of these research systems have been applied to real-world problems, but only a few have led to industrial applications or the development of commercial products (see the "Industrial applications of qualitative simulation" sidebar).

Embedding qualitative simulation. In a typical industrial application, the qualitative simulator is part of a set of software components that are coupled with a physical process by sensors and actuators. The computer system (including the software component "qualitative simulator") and the physical process (the environment) form a dedicated unit commonly called an *embedded system*. The qualitative simulator in such an application has various requirements. First, it must interact with other software components and the environment. Second, it must compute its results within a reasonable amount of time. Finally, because it is coupled to a physical process, it

1094-7167/00/$10.00 © 2000 IEEE

Industrial applications of qualitative simulation

Qualitative reasoning has been a very active research area since the early 1980s, and qualitative simulation has always played an important role in this area. Until recently, very little research in this area has led to industrial or commercial applications. In the last few years, however, more attention has gone to application-oriented research.[1–3]

Probably the most advanced industrial application of qualitative simulation for monitoring and diagnosis is the recently completed ESPRIT III project Tiger.[4] A condition-monitoring system for gas turbines that was developed as part of Tiger reasons in parallel at three different levels to meet different reasoning and performance requirements. At the top level, qualitative simulation predicts the turbine's behaviors at startup and in response to load changes. This *constrained-influences* approach is based on the combined use of causality and deep knowledge in terms of mathematical equations. Application sites include a large industrial turbine at Exxon Chemical in the UK and a small aircraft auxiliary power-unit turbine at Dassault Aviation in France.

An industrial application done in cooperation with Siemens[5] uses qualitative simulation for online diagnosis and monitoring of ballast-tank systems on ships and offshore platforms. The application's *prediction module* derives qualitative parameter values only for instantaneous time points to check consistency with parameter observations.

Research at the University of Wales in combination with Ford and Jaguar has led to the commercially available design-analysis tool Flame.[6] Designs must be analyzed for hazardous and safety-critical situations. Flame automatically generates a *failure-mode effect analysis* of electrical subsystems in cars. FMEA involves the investigation and assessment of the effects of all possible failure modes on a system. Work in this area is also done at the Technical University of Munich, Bosch, and Daimler.

Although none of these applications uses the qualitative simulator QSim, they have adopted many ideas from QSim. The European Network of Excellence MONET (see monet.aber.ac.uk) is an excellent starting point for further investigations of qualitative-simulation and qualitative-reasoning applications.

References

1. *Proc. First Int'l Workshop on Model-Based Systems and Qualitative Reasoning: Perspectives for Industrial Applications,* John Wiley & Sons, London, 1996.
2. R. Milne, J. Pastor, and L. Travé-Massuyés, "Qualitative Reasoning for Complex Systems and Their Control," *Proc. Workshop at the 16th Int'l Joint Conf. AI*, Morgan Kauffmann, San Francisco, 1999.
3. L. Travé-Massuyés and R. Milne, "Application-Oriented Qualitative Reasoning," *The Knowledge Eng. Rev.*, Vol. 10, No. 2, June 1995, pp. 181–204.
4. R. Milne et al., "TIGER: Real-Time Situation Assessment of Dynamic Systems," *Intelligent Systems Eng.*, Fall, 1994, pp. 103–124.
5. O. Dressler, "On-Line Diagnosis and Monitoring of Dynamic Systems Based on Qualitative Models and Dependency-Recording Diagnosis Engines," *Proc. 12th European Conf. AI,* John Wiley & Sons, London, 1996, pp. 481–485.
6. D.R. Pugh and N.A. Snooke, "Dynamic Analysis of Qualitative Circuits for Failure Mode and Effect Analysis," *Proc. Ann. Reliability and Maintainability Symp.,* IEEE Press, Piscataway, N.J., 1996, pp. 37–42.

must be able to handle noisy and erroneous data. We define such an interactive, efficient, and robust application of qualitative simulation as *embedded qualitative simulation*. Based on the tightness of the coupling between software components and the physical process, we can classify two categories:

- *Offline applications* have a loose coupling between the computer system and the environment. Data transfers offline between these systems, for example, through files or a user interface. Although the qualitative simulator's actual performance is important for the technique's acceptance, it is not vital for its functionality. Qualitative simulation serves as a tool that interacts with other tools, such as CAD systems. Interoperability, adaptability, and portability are the important implementation issues. Typical offline applications of qualitative simulation are design verification and failure-mode effect analysis (FMEA).
- *Online applications* have a tight coupling between the computer system and the environment. The computer system must be reactive; that is, it must compute its output data when input data is derived from the environment. Moreover, almost all online qualitative simulation applications require real-time behavior, where the timeliness of the computer system's results is vital for the system's functionality. Here, performance and—even more important—predictability play crucial roles. The computer system must react without fail to inputs from the environment within predefined time windows. Because real-time systems are also spatially tightly coupled with the physical process, resource limitations strongly influence their design. Often widely divergent criteria must be met, such as low power consumption, small size, high performance, and high reliability. Typical online qualitative simulation applications are monitoring and fault diagnosis. Although some monitoring and diagnosis applications use qualitative simulation, they lack reactive and real-time behavior.

Both categories will become increasingly important commercially, provided qualitative simulators drastically improve in their interoperability, robustness, and efficiency. Looking at these requirements, we feel that efficiency is the major issue. The qualitative-modeling paradigm itself supports robustness, which leads naturally to quite robust system representations. Interoperability requires that qualitative simulators become software components with well-defined interfaces, and portability to many different platforms. Trends in software engineering, platform independence, and component software support this requirement. Efficiency, however, will be the enabling factor for embedded qualitative simulation.

Efficient embedded qualitative simulation. Improving qualitative simulation's performance will require effort in the design of both efficient qualitative simulation methods and computer architectures that optimally support these methods. A great challenge is the combination of AI tasks with real-time behavior, a challenge the AI community faces more and more nowadays. The model-based configuration manager of NASA's DS-1 spacecraft[1] is an excellent example of such combination. Based on the taxonomy David Musliner and his colleagues have developed,[2] we can identify three approaches to real-time AI:

- embedding AI tasks into a real-time system,
- embedding real-time tasks into an AI system, and
- cooperating real-time and AI tasks.

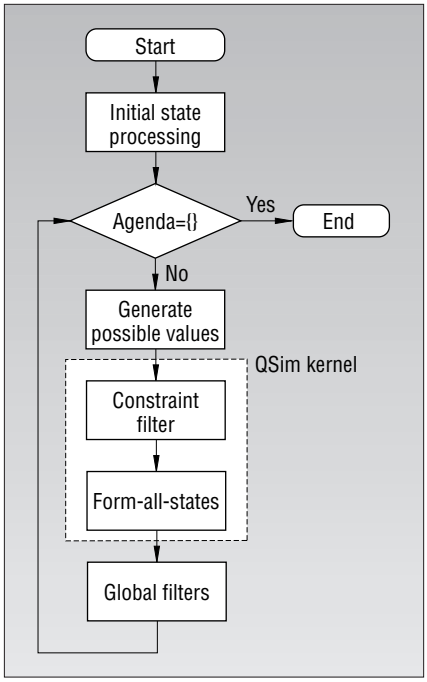

Figure 1. QSim flowchart.

In the last approach, neither the AI tasks nor parts of them are forced to run in real time; the AI tasks plan and schedule the real-time tasks. Therefore, this approach is not well suited for developing a real-time application with the AI task qualitative simulation. The first two approaches, however, lead directly to reasonable application scenarios.

Embedding qualitative simulation into a real-time system. Here, the qualitative simulator is forced to meet deadlines. The first of two methods to achieve this reduces the qualitative simulation's high execution time and its variance by

- constraining the qualitative-simulation algorithm's input—the model—at the price of decreased output quality,
- customizing the qualitative-simulation algorithm to a certain problem class (incorporating domain knowledge to simplify qualitative simulation), and
- supporting qualitative simulation by a specialized computer architecture, often denoted as performance engineering.[3]

Although these strategies will let us solve numerous important real-world problems, none makes an algorithm for qualitative simulation predictable in general. The second method to force qualitative simulation to meet deadlines is to design incremental or anytime algorithms.[4] This method makes the simulation task interruptible and at any time provides a useful—rather than the optimal—reaction.

Embedding real-time tasks into qualitative simulation. Here, most parts of the qualitative simulator remain unchanged. However, some functions that must have predictable execution times are integrated into the system as high-priority tasks. An example scenario for this approach is a monitoring and diagnosis system where a real-time task monitors the system state and, if it detects critical values of some parameters, triggers an alarm shutdown. Otherwise, the monitored system state passes to an AI task that diagnoses the system, but not in real time.

Specialized computer architecture. To advance embedded applications of qualitative simulation, we have developed a specialized computer architecture designed to improve the runtime performance, scalability, and portability of QSim.[5] Our approach improves runtime performance by parallelizing and mapping some QSim functions onto a multiprocessor system and migrating others from software to dedicated hardware.[6] This approach will broaden the application area for QSim in offline applications and facilitate qualitative simulation in online applications. In the presence of timing constraints, our approach will enable a wider range of real-world examples to be simulated in real-time, following the performance-engineering approach. Scalability means that we can adapt our computing system's performance to the problem complexity by adding more processing elements and dedicated hardware. We increased portability by implementing QSim in C, which is far more appropriate for porting the qualitative simulator to different embedded-processor platforms than the original Lisp implementation.

QSim algorithm

In QSim, models take the form of either qualitative differential equations (QDEs) or constraint networks, which consist of variables and constraints. Variables represent system parameters, such as velocity or temperature. Variable values 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 that represent arithmetic relations (such as ADD–, MULT–, and D/DT–constraints) and functional dependencies (such as the monotonic function constraints M^+ and M^-) between variables.

QSim predicts all possible behaviors of a physical system. A behavior is a sequence of states that represent one possible temporal evolution of the system. The generation of behaviors basically requires the solution of two different problems:

- *Generation of initial states*: Given a QDE and partial information about the initial state, determine all complete, consistent qualitative states.
- *Generation of successor states*: Given a QDE and a complete qualitative state, determine its immediate successor states.

Figure 1 shows the flowchart of the basic QSim algorithm. *Initial state processing* generates all complete, consistent initial states and stores them in an agenda for further processing. The algorithm generates each state's immediate successors in three successive steps:

- The algorithm determines the possible values of all variables for the next time-step (*generate possible values*).
- The *QSim kernel* generates all candidates for successor states.
- The algorithm applies *global filters* to test each candidate state for consistency with the behavior's other states—for example, to detect cycles.

States surviving all these checks are stored in the agenda. The generation of successor states continues until the algorithm has processed all states in the agenda or has reached a resource limit.

The QSim kernel consists of two consecutive functions, *constraint filter* and *form-all-states*, which are further hierarchically structured as Figure 2 shows. The constraint filter calls several *tuple-filter* functions. Each constraint in the QDE requires one tuple-filter function. These functions further divide into *constraint-check functions*. CCFs are primitive functions, and an individual CCF exists for each constraint type.

The QSim computer architecture

In developing our QSim computer architecture, we relied on two strategies: parallelizing and mapping more complex QSim kernel functions onto a multiprocessor and supporting small but runtime-intensive functions with specialized processing elements.

Combining both strategies produces the maximum performance gain.

QSim kernel multiprocessor. The generation of all candidate successor states in the QSim kernel is equivalent to finding all solutions of a constraint-satisfaction problem defined by the constraint network and the possible values of all variables.[7] Thus, we can also view the QSim kernel as a CSP solver. The kernel functions find the solutions by transforming the constraint network into its dual representation; that is, the network's constraints become the CSP's variables, and the CSP's constraints represent the constraint network's shared variables. From this viewpoint, the constraint filter achieves local consistency of the dual-constraint network—node and arc consistency—and form-all-states finds all globally consistent variable assignments (CSP solutions).

Parallel constraint filter. The constraint filter uses the possible values of all variables as input data and returns only combinations of these possible values that do not violate local consistency conditions. This filtering takes two consecutive steps:

- The tuple-filter function returns only tuples that are consistent within an individual constraint (node consistency).
- The Waltz filter function discards tuples that violate conditions between adjacent constraints (arc consistency).

The dataflow graph in Figure 3 reveals that all tuple-filter functions are independent and can execute in parallel. The number of constraints *C* of the QDE determines the degree of parallelism. The Waltz filter executes after all tuple-filter results have been received. All constraint-filter functions can group logically in a master/slave structure of tasks. For each QDE constraint, one slave task executes that constraint's tuple-filter function. The master task transmits the input data to all tuple-filter tasks, receives the tuple-filters' results, and executes the Waltz filter.

Although the logical structure implies the use of multiprocessors in a *star* topology with the master as the central node, we map the constraint-filter tasks onto a *wide-tree* topology. In a wide tree, each node can have more than two children and the master task corresponds to the root node. On the one hand, this is motivated because in a star structure the master becomes a bottleneck as the number of slaves increases. By using a wide tree, we ensure scalability. On the other hand, the wide tree leverages on a class of microprocessors having several fast point-to-point communication links on-chip that allow a simple and quick setup of multiprocessors. Using the wide-tree model, we can both easily map the logical task structure onto such a multiprocessor and optimally exploit its communication facilities.

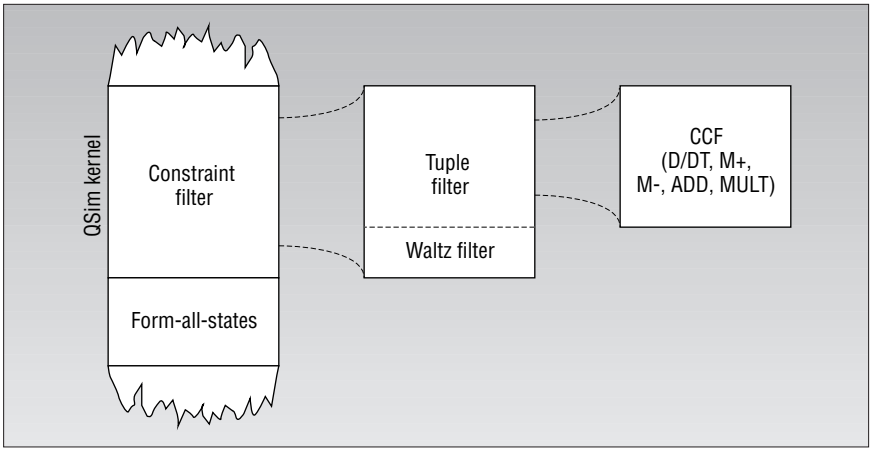

Figure 2. The hierarchical structure of the QSim functions.

Because the actual number of slave tasks and their input data is not known before kernel execution, our approach uses an online scheduling algorithm running on the root processor to map the slave tasks onto the remaining processing elements. We apply a list-scheduling algorithm that

- uses estimated task-execution times to improve the schedule,
- guarantees a worst-case completion time for all tasks, and
- is only slightly slower than simple task-attraction scheduling.

Parallel form-all-states. The kernel function form-all-states uses a backtracking algorithm to solve the CSP. To find all solutions of the CSP, a depth-first search must process a big search space—spanned by the tuples surviving the constraint-filter. Unlike the constraint filter, the function hierarchy offers no obvious parallelization. For a parallel implementation of form-all-states, the CSP must be partitioned. Our QSim architecture uses a *parallel-agent-based* partitioning strategy.[8] PAB divides the overall search space into smaller independent subspaces that any sequential CSP algorithm can solve in parallel. It achieves the partitioning by dividing the tuple sets of constraints into subsets. A subspace is thus given by a subset of tuples of some constraints and the complete tuple sets of all remaining constraints. Our variable-based partitioning algorithm[9] performs the partitioning. The degree of parallelism—the number of independent subspaces—depends on input data and cannot be determined in advance.

The parallel form-all-states algorithm's logical structure is similar to the constraint filter's logical structure. Given the PAB strategy, a master/slave structure also results. The master task generates and transmits subproblems to the slave tasks and merges the partial results to the overall result. The slave tasks execute a sequential CSP algorithm to find all solutions in the subspaces.

QSim kernel coprocessors. The tuple filter checks each constraint for node consistency by calling the CCF for each candidate tuple. Our analysis revealed that these CCFs are very runtime-consuming functions. Consequently, we have designed specialized processing elements to accelerate these CCFs and tuple filters for the most common constraint types (MULT, ADD, M^+, M^-, and D/DT). Our design realizes these processing

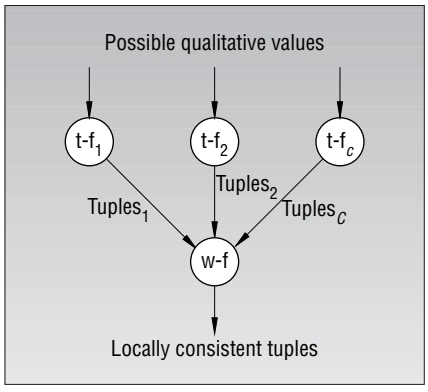

Figure 3. A dataflow graph of the constraint filter for a QDE with *C* constraints. t-f_i denotes the tuple-filter function for the constraint *i*, W-f the Waltz-filter, and $tuples_i$ the node-consistent tuples of constraint *i*.

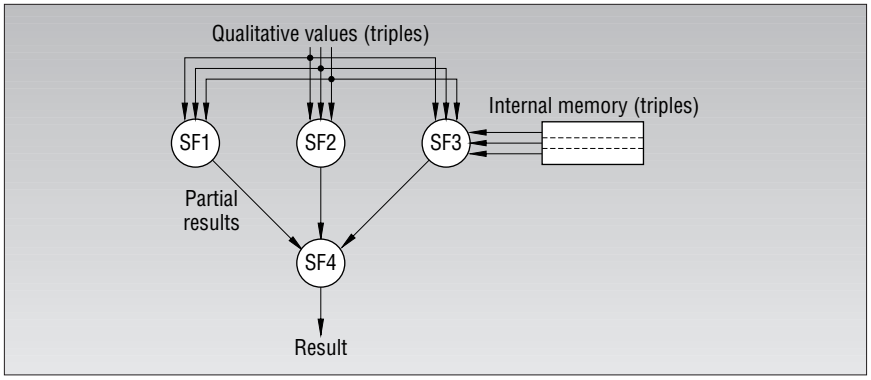

Figure 4. The MULT-CCF is partitioned into four subfunctions, SF1 to SF4.

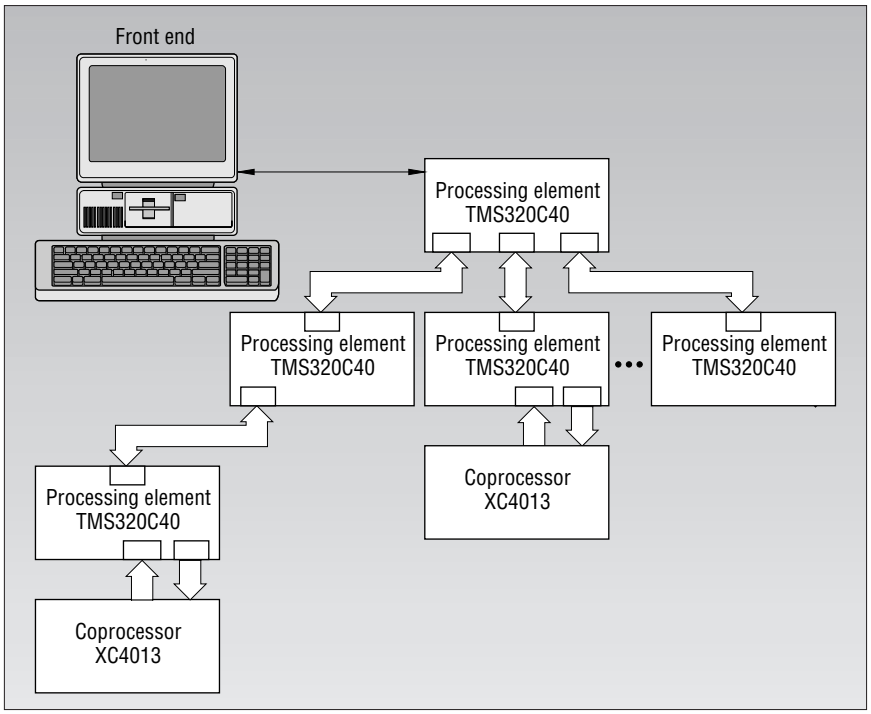

Figure 5. An example for the overall QSim computer architecture. The processing elements (DSP TMS320C40) connect in a wide-tree structure with up to five children per node. The processing elements can be equipped with coprocessors (Xilinx XC4013). The number of processing elements and coprocessors can vary owing to the scalable design of the QSim computer architecture.

elements as coprocessors attached to the multiprocessor processing elements.

As an example, Figure 4 shows the dataflow graph of the MULT-CCF, one of the most time-consuming CCFs. We can partition this CCF into four subfunctions, SF1 to SF4. The subfunctions SF1 to SF3 check whether various rules for qualitative multiplication hold with respect to the given input values. These subfunctions require triples of qualitative values as input data and return boolean results. SF3 forms an iteration over additionally required qualitative values, called the *corresponding values*, stored in the internal memory. SF4 performs a logical AND operation on the partial results of SF1 to SF3.

The main coprocessor design's features are

- *exploitation of parallelism*—the parallel execution of SF1, SF2, and the iterations of SF3,
- *use of optimized data types*—the number of bits and the coding scheme of the input values and the values stored in the coprocessor memory, and
- *use of customized memory architectures*—the internal organization and the access mode of the coprocessor memory.

Besides the functional blocks SF1 to SF4, the coprocessor contains a block of memory, an I/O controller, and a function controller. The I/O controller establishes communication to a host processor through two separate communication channels that enable simultaneous input and output operations. The function controller decodes the instructions and controls the operation of all other functional blocks of the coprocessor. The coprocessor architectures for other constraint types are similar. The CCFs for the M^+ and M^- are slightly less complex because SF2 is not required. The CCF for D/DT consists only of one subfunction; the iterative check for the corresponding values is also not required.

Prototype implementation and experimental results. Figure 5 shows an example of the overall heterogeneous multiprocessor architecture. We chose the digital signal processor TMS320C40 as the processing element because of its six independent communication channels and its high I/O performance. So we can build wide-tree structures of up to five children per node. We developed the software using the distributed real-time operating system Virtuoso, which supports a portable and flexible software design.

We prototyped a number of coprocessors on field-programmable gate arrays (FPGAs). Table 1 shows the hardware cost in terms of configurable logic blocks for the Xilinx XC4K family and the utilization of the XC4013. For the constraint types MULT, ADD, M^+, and M^-, we prototyped CCF coprocessors with an internal memory capacity of 16 triples. The coprocessors for MULT and ADD use sequential execution of the SF3 iterations, whereas the M+ and M– use 16 processing elements to evaluate the SF3 iterations in parallel. For the constraint type D/DT, we implemented a complete tuple-filter coprocessor that checks all 16 possible CCFs in a single step.

We based our experimental evaluation on a prototype consisting of one to seven TMS320C40 processors running at 50 MHz

Table 1. Coprocessor types with the required hardware cost in configurable logic blocks (CLBs) of the Xilinx XC4K family and utilization of the XC4013.

Coprocessor type	Hardware cost (CLBs)	Utilization of XC4013 (%)
CCF		
MULT	317	55
ADD	173	30
M^+	519	90
M^-	519	90
Tuple filter		
D/DT	137	24

and three XC4013 coprocessors each executing CCFs of one type. We ran QSim simulation models on the multiprocessor, varying the number of processing elements and tracing all kernel functions. Separately, we measured the execution times of all coprocessor types for all possible input data, which let us estimate the overall runtime of our architecture assuming full coprocessor support. Full coprocessor support is a realistic scenario because recent progress in FPGA technology now permits the integration of all coprocessor types shown in Table 1 into a single FPGA device.

The individual speedups resulting from parallelization and coprocessor support, S_{cf} S_{fas}, and S_{cop} let us determine the overall speedup of the QSim computer architecture S_{tot} compared to the runtime on a single processing element. The kernel's runtime is given by the sum of the runtimes of the two kernel functions constraint filter and form-all-states: $t_{seq} = t_{cf} + t_{fas}$. We define two ratios of kernel function runtimes, $\alpha = t_{cf}/t_{seq}$ and $\beta = t_{fas}/t_{seq}$. With these ratios, the QSim computer architecture's speedup takes the form

$$S_{tot} = \frac{1}{\frac{\alpha}{S_{cf} S_{cop}} + \frac{\beta}{S_{fas}}}.$$

Given this speedup formula, high overall speedup clearly can result only if both kernel functions are accelerated appropriately with respect to their runtime ratios. Depending on the solution of the two basic problems of QSim—the generation of initial states and the generation of successor states—we can identify two different classes of runtime ratios. Our empirical runtime analysis revealed that the generation of initial states results in a runtime ratio of approximately $\alpha = 0.1$ and $\beta = 0.9$. In this case, a high speedup for form-all-states is important. On the other hand, generation of successor states results in a runtime ratio of approximately $\alpha = 0.75$ and $\beta = 0.25$. Here, a high speedup for the constraint filter is essential; therefore, the influence of the coprocessors on the overall speedup increases considerably.

We evaluated the QSim computer architecture using the three QSim simulation models STLG (17 variables, 18 constraints), RCS (45 variables, 48 constraints), and QSEA (38 variables, 37 constraints).[5] Table 2 shows the kernel execution times of a complete simulation run for these models. This table includes t_{seq}, the execution time of the QSim computer architecture using one processing element; t_{mp}, the execution time for a seven-node multiprocessor; and t_{tot}, the execution time for a seven-node multiprocessor with full coprocessor support. Table 2 also presents the kernel execution times on a standard Lisp implementation of QSim t_{std}. For a comparison, we simulated the same models on a Sun Sparc10 running SunOS 5.5, compiling and executing the QSim source code using Allegro Common Lisp 4.3.1. We included timing functions into the Lisp source code to determine the runtime required for the kernel functions. We measured the ratios of the kernel runtime to the overall QSim runtime—including the global filters (see Figure 1)—as 70% for STLG, 93% for RCS, and 98% for QSEA.

Figure 6 compares the speedup factors of the QSim computer architecture to a standard QSIM implementation in Lisp. An acceleration factor of about three to eight results from migrating from a Lisp to a C implementation. This is quite remarkable because the Sparc processor is clocked much faster than the TMS320C40 processor and because the TMS320C40's special processor features are not exploited well due to the symbolic computation of QSim. Using the multiprocessor and coprocessor support improves the performance by another order of magnitude.

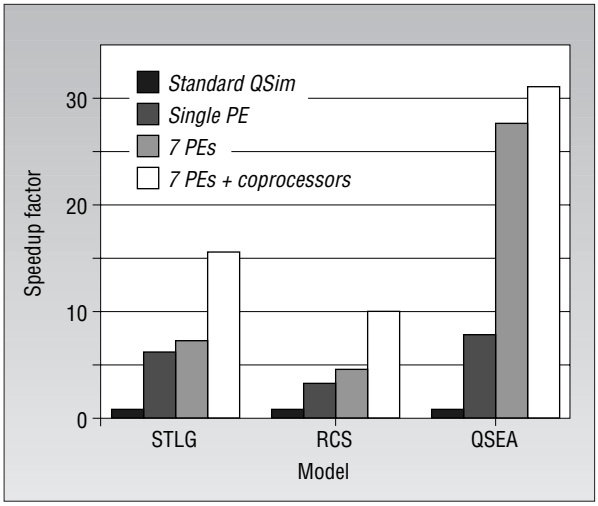

Figure 6. Speedup factors of the QSim computer architecture compared to a standard QSim implementation in Lisp.

Table 2. Comparison of the QSim computer architecture with the standard Lisp implementation of QSim. The runtimes of the standard Lisp implementation (t_{std}) were measured on a Sparc10 running Allegro Common Lisp. These runtimes are compared with the kernel runtimes on the QSim computer architecture with one processing element (t_{seq}), using a seven-node multiprocessor (t_{mp}) and this multiprocessor with full coprocessor support (t_{tot}).

Model	Standard QSim	QSim computer architecture		
	t_{std} [s]	t_{seq} [s]	t_{mp} [s]	t_{tot} [s]
STLG	0.19	0.03	0.026	0.012
RCS	4.05	1.21	0.917	0.403
QSEA	114.58	14.56	4.160	3.640

OUR WORK PROVIDES TWO MAIN benefits for QSim applications. First, using our QSim kernel coded in C improves the performance by at least an order of magnitude compared to a standard Lisp implementation on the same processor platform. The C kernel also improves the portability to different embedded target platforms and the coupling with many existing I/O subsystems for process control and monitoring. Second, deploying our QSim computer architecture results in a further performance improvement of about one order of magnitude. This specialized architecture is scalable: processors and dedicated hardware can be easily added to improve the performance even further, if the simulation model provides sufficient parallelism.

Developing this specialized architecture has given us insights that might be interesting for other AI applications as well. Parallelizing AI algorithms is very different than in scientific computing where parallelization is more commonly applied. AI algorithms mostly rely on symbolic computation, have an irregular runtime behavior, and offer only

low to medium data parallelism. We believe that the key to successful parallelization is

- the comprehensive analysis of the AI algorithm,
- the exploitation of parallelism at multiple levels of granularity, and
- the deployment of a multiprocessor system that best matches the multigranular parallel algorithm in its logical structure and its communication requirements.

Both offline and online applications will benefit from the improvement in QSim's performance. For offline applications, application developers can simulate more complex models in less time, which in turn will enable qualitative simulation to more widely accepted engineering tasks. Online applications that follow the performance-engineering approach will benefit by providing tighter bounds on the execution times for given models. However, performance engineering does not reduce the QSim algorithm's possibly exponential worst-case behavior.

The advances in performance, scalability, and portability of QSim are important first steps toward embedded qualitative simulation. Future efforts to reach this ambitious goal should focus on integrating qualitative simulation with established methods at the quantitative level to benefit from both worlds, improving qualitative simulation's real-time capabilities and deploying it in embedded real-world applications. ▪

Acknowledgments

This work took place at the Institute for Technical Informatics, Technical University Graz, and was supported by the Austrian Science Fund under grants P10411-MAT and J1429-MAT.

References

1. N. Muscettola et al., "Remote Agent: To Boldly Go Where No AI System Has Gone Before," *Artificial Intelligence*, Vol. 103, Nos. 1–2, Aug. 1998, pp. 5–47.
2. D.J. Musliner et al., "The Challenges of Real-Time AI," *Computer*, Vol. 28, No. 1, Jan. 1995, pp. 58–66.
3. T.P. Hamilton, "An Architecture for Real-Time Qualitative Reasoning," *Recent Advances in Qualitative Physics,* B. Faltings and P. Struss, eds., MIT Press, Cambridge, Mass., 1992, pp. 279–294.
4. A. Garvey and V. Lesser, "A Survey of Research in Deliberative Real-Time AI," *Real-Time Systems*, Vol. 6, No. 3, May 1994, pp. 317–347.
5. M. Platzner, B. Rinner, and R. Weiss, "Parallel Qualitative Simulation," *Simulation Practice and Theory,* Vol. 5, Nos. 7–8, Oct. 1997, pp. 523–538.
6. B. Kuipers, *Qualitative Reasoning: Modeling and Simulation with Incomplete Knowledge,* MIT Press, Cambridge, Mass., 1994.
7. A.K. Mackworth, "Constraint Satisfaction," *Encyclopedia of Artificial Intelligence, Vol. 1,* S.C. Shapiro, ed., John Wiley & Sons, New York, 1992, pp. 285–293.
8. Q.P. Luo, P.G. Hendry, and J.T. Buchanan, "Strategies for Distributed Constraint Satisfaction Problems," *Proc. 13th Int'l DAI Workshop,* 1994.
9. M. Platzner and B. Rinner, "Design and Implementation of a Parallel Constraint Satisfaction Algorithm," *Int'l J. Computers and Their Applications,* Vol. 5, No. 2, June 1998, pp. 106–116.

Marco Platzner is a senior researcher at the Swiss Federal Institute of Technology (ETH) Zurich where he works on reconfigurable computer architectures. His research interests include embedded systems, hardware and software codesign, and application-specific computing systems. He received his PhD and MSc in telematics from the Technical University Graz. He is a member of the IEEE, ACM, and TIV (Telematik Ingenieurverband, Austria). Contact him at the Computer Eng. and Networks Lab, Swiss Federal Inst. of Technology (ETH) Zurich, Gloriastrasse 35, 8092 Zurich, Switzerland; marco.platzner@computer.org.

Bernhard Rinner is a senior researcher at the Technical University Graz. His research interests include parallel processing, real-time AI, and embedded systems. He received his PhD and MSc in telematics from Technical University Graz. He is member of the IEEE and TIV. Contact him at the Inst. for Technical Informatics, Technical Univ. Graz, Inffeldgasse 16, A-8010 Graz, Austria; b.rinner@computer.org.

Reinhold Weiss is a professor of computer science (technical informatics) and the head of the Institute for Technical Informatics at the Technical University Graz. His research interests focus on embedded distributed real-time architectures with applications in factory automation, digital signal processing, and simulation. He received his Dipl.-Ing. and Dr.-Ing. in electrical engineering and his Dr.-Ing. habil. in real-time control from the Technical University of Munich. He is a member of the editorial board of *Computers and Their Applications*. He is a member of the IEEE, GI (Gesellschaft für Informatik, Germany), and ÖVE (Österreichischer Verein für Elektrotechnik, Austria). Contact him at the Inst. for Technical Informatics, Technical Univ. Graz, Inffeldgasse 16, A-8010 Graz, Austria; rweiss@iti.tu-graz.ac.at.