Dipl.–Ing. Bernhard Rinner

Design, Prototype Implementation and Experimental Evaluation of a Scalable Multiprocessor Architecture for Qualitative Simulation

Dissertation

vorgelegt an der Technischen Universität Graz

zur Erlangung des akademischen Grades
"Doktor der Technischen Wissenschaften"
(Dr. techn.)

durchgeführt am Institut für Technische Informatik
Vorstand: O. Univ.–Prof. Dipl.–Ing. Dr. techn. Reinhold Weiß

April 1996

Zusammenfassung

Die vorliegende Arbeit behandelt den Entwurf, die Implementierung und die experimentelle Bewertung eines skalierbaren Multiprozessors für schnelle qualitative Simulation. Das Ziel der Arbeit ist die Reduktion der Laufzeit des qualitativen Simulators QSIM.

In der qualitativen Simulation werden physikalische Systeme auf einer höheren Abstraktionsebene modelliert als in anderen Simulationsparadigmen, wie z.B. in der kontinuierliche Simulation. Ein wesentlicher Vorteil der qualitativen Simulation ist Darstellung und Verarbeitung mit unvollständigem Wissen. Qualitative Simulation benötigt weder eine vollständige, strukturelle Beschreibung des Systems noch einen eindeutig spezifizierten Anfangszustand. Alle physikalisch möglichen Verhalten, die von diesem unvollständig beschriebenen Anfangszustand ableitbar sind, werden vorhergesagt. Qualitative Simulation wird im technischen Bereich hauptsächlich in der Überwachung und der Diagnose von Prozessen eingesetzt.

QSIM ist der bekannteste qualitative Simulator. QSIM ist in der Programmiersprache Lisp implementiert und wird auf Universalrechnern ausgeführt. Ein wesentlicher Nachteil derzeitiger QSIM Implementierungen ist die hohe Laufzeit. Eine Verbesserung der Laufzeit ist eine wichtige Voraussetzung für einen breiten industriellen Einsatz von QSIM. In dieser Arbeit wird ein Spezialrechner für die Kernfunktionen von QSIM entwickelt.

Die Entwicklung dieses Spezialrechners basiert auf einer empirischen Laufzeitanalyse von QSIM Implementierungen. Abhängig von der Art der Berechnung der Zustände ergibt sich ein unterschiedliches Laufzeitverhältnis für die beiden Kernfunktionen *constraint-filter* und *form-all-states*. Bei der Berechnung von Nachfolgezuständen dominiert der constraint-filter die Laufzeit des QSIM Kernes. Bei der Berechnung von Anfangszuständen benötigt die Funktion form-all-states den größten Teil der Laufzeit. Beide Kernfunktionen werden durch Parallelisierung und Abbildung auf ein Multiprozessor System beschleunigt.

In der vorliegenden Arbeit werden die Kernfunktionen anhand ihrer Hierarchie analysiert und die Parallelisierbarkeit durch eine Datenabhängigkeitsanalyse untersucht. Diese Analyse ergibt für den constraint-filter, daß seine Teilfunktionen *tuple-filter* unabhängig voneinander sind und parallel ausgeführt werden können. Die Funktion form-all-states bestimmt alle Lösungen des sogenannten *constraint satisfaction Problems (CSP)*. Um eine parallele Ausführung von form-all-states zu erzielen, wird das CSP zur Laufzeit in unabhängige Teilprobleme partitioniert. Die Zuordnung von Tasks zu Prozessoren erfolgt mit einem on-line Schedulingverfahren, das ein statisches Schedulingverfahren verwendet. Die gesamte Rechnerarchitektur für den QSIM Kern ist ein Multiprozessor System mit verteiltem Speicher, welches im MIMD Modus betrieben wird. Die Prozessorelemente sind in einer breiten Baumstruktur miteinander verbunden, um eine skalierbare Rechnerarchitektur zu erzielen.

Der Prototyp der Rechnerarchitektur für die QSIM Kernfunktionen besteht aus digi-

talen Signalprozessoren vom Typ TMS320C40 (Texas Instruments). Die Software-Implementierung basiert auf dem verteilten Echtzeit Betriebssystem Virtuoso (Eonic Systems). In der experimentellen Bewertung wird die Laufzeit der parallelen Implementierung auf der Spezialrechnerarchitektur mit der Laufzeit einer sequentiellen Implementierung verglichen. Die sequentielle Implementierung der QSim Kernfunktionen wird ebenfalls auf einem Prozessorelement der Spezialarchitektur ausgeführt. Verschiedene QSim Simulationsmodelle werden für diesen Vergleich als Eingabedaten verwendet. Die experimentelle Bewertung zeigt, daß die Laufzeit der Kernfunktionen von QSim durch eine Spezialrechnerarchitektur signifikant reduziert werden kann.

Abstract

This dissertation presents the design, the prototype implementation and the experimental evaluation of a scalable multiprocessor for qualitative simulation. The main objective of this work is to improve the running time of the qualitative simulator QSIM.

In qualitative simulation, physical systems are modeled on a higher level of abstraction than in other simulation paradigms, like in continuous simulation. A major strength of qualitative simulation is that it can represent and reason with incomplete knowledge — qualitative simulation requires neither a complete structural description nor a fully specified initial state. All physically possible behaviors consistent with this incomplete description are predicted by qualitative simulation. In engineering, qualitative simulation is mainly applied in monitoring and diagnosis.

QSIM is the most prominent algorithm for qualitative simulation. QSIM is implemented in Lisp and executed on general-purpose computers. A drawback of current QSIM implementations is poor execution speed. This prevents QSIM from wider industrial application. In this thesis, a special-purpose computer architecture for the kernel of QSIM is developed.

The development of this special-purpose computer architecture is based on an extensive empirical running time analysis of QSIM. Depending on the mode of operation of QSIM, this analysis reveals different running time ratios for the two kernel functions *constraint-filter* and *form-all-states*. In generation of successor states, the constraint-filter dominates the overall running time of the kernel, whereas in initial state processing, the function form-all-states dominates the kernel running time. Both kernel functions are accelerated by parallelization and mapping onto a multiprocessor system.

In the presented thesis, the QSIM kernel is analyzed in a top-down manner, and the potential parallelism is detected by a data dependence analysis. For the constraint-filter, this analysis reveals that the *tuple-filter* functions are independent of each other and can be executed in parallel. The function form-all-states finds all solutions of a so-called *constraint satisfaction problem (CSP)*. To enable parallel execution of form-all-states, the CSP is partitioned into independent subproblems at run-time. Static on-line scheduling is used to schedule tasks to processing elements. The overall QSIM kernel computer architecture is a multiprocessor system with distributed memory and is operated in MIMD mode. The processing elements are connected in a wide-tree topology for scalability.

The prototype implementation of the QSIM kernel architecture is based on a multiprocessor system consisting of digital signal processors TMS320C40 (Texas Instruments). The software is implemented using the distributed real-time operating system Virtuoso (Eonic Systems). In the experimental evaluation, the running time of the QSIM kernel architecture is compared with the running time of a sequential implementation of the QSIM kernel executed on one processing element of the multiprocessor system. Several QSIM simulation models are used as input data for this comparison. The experimental evaluation proves that a significant speedup can be achieved with the prototype of the QSIM kernel architecture.

Acknowledgments

This dissertation was performed during my research assistantship at the Institute for Technical Informatics, Graz University of Technology.

I especially thank my advisor, Prof. Dr. Reinhold Weiß, for his continuous encouragement and support, and also for providing me with extraordinary research facilities. Special thanks go to the whole staff at the Institute for Technical Informatics for providing a pleasant and stimulating research environment. I am grateful to Prof. Benjamin Kuipers and all the collaborators of the qualitative reasoning group at the University of Texas at Austin for their kind reception and for their great support during my visit last summer. I am deeply indebted to all collaborators of the research project "Distributed Computer Architecture for Qualitative Simulation" who have contributed to the quality of this project through their work over the last years. I am very grateful to the Austrian National Science Foundation who has supported this research project under grant P10411-MAT for their financial assistance.

And finally, I thank my parents for their support and encouragement during my studies and, especially, Gundis for her patience and understanding.

Graz, April 1996 BERNHARD RINNER

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Model-Based Diagnostic Reasoning

Over the years, human experts have gained sufficient experience to solve a lot of problems efficiently. During the last years, much work has been done to build machines that imitate the description of human thinking and acting. Expert systems are computer programs which use knowledge and inference mechanisms to solve problems where normally expert knowledge is required. One important class of problems is *diagnostic reasoning*. Diagnostic reasoning can be seen as a classification problem, since it involves identifying current behavior with a set of known classes of behavior. It can also be considered as an abduction problem, concerned with generating plausible explanations for observations [KM93].

Diagnostic reasoning systems are important in many technical systems. As these systems — like electronic circuits, assembly lines or nuclear power plants — are becoming more complex, the need for automatic reasoning systems to support troubleshooting is increasing enormously.

Most diagnostic expert systems can be classified into the following categories [Pup87]:

Statistical approach. The statistical approach determines the probabilities of a diagnosis. This approach is mainly based on the *Bayes Theorem*. Due to the lack of causal inference, the statistical reasoning process has only a very limited ability for explanation.

Associative approach. Associative diagnosis systems are built by accumulating the experience of expert troubleshooters in the form of empirical associations. The associative approach became popular, in part, because it permitted easy construction of expert systems by encoding heuristic information in the form of *if–then* rules. However, a big problem is the knowledge accumulation of rule-based systems.

Model-based approach. Model-based diagnosis can be viewed as an interaction between prediction and observation. The knowledge is represented by different models for the *structure* and the *behavior* of the system. The predicted behavior is compared with the actual observation, producing discrepancies. Discrepancies then give rise to a possible diagnosis [Ham91]. A Model-based diagnosis covers a broader range of faults by viewing misbehavior as anything other than what the model predicts. This

approach also better captures the causal dependencies of the system than the other two approaches.

The key units of a model-based diagnostic expert system are the *models* which describe behavior and structure and the *inference mechanism* for predicting the behavior given the structure of the system. Reasoning with models can be broken down into two major subproblems [Kui93b]. *Model building* starts with a description of the physical situation and builds an appropriate simplified model. *Model simulation* starts with a model and predicts the possible behaviors consistent with the model. A system can be modeled at different levels of abstractions. These levels range from detailed numerical descriptions up to rather coarse and incomplete descriptions at a qualitative level. The selected abstraction level depends basically on the domain, the information available about the system and the inference mechanism which must correspond to the model description. In many cases, the behavior is predicted by simulators. Therefore, numerical, discrete or qualitative simulators are often applied. Sometimes system descriptions at different levels are combined to achieve diagnosis at an appropriate level of detail.

Model-based fault diagnosis is applied more and more nowadays. This reasoning technique is used in both static as well as dynamic systems. In dynamic systems, the parameters of the system change over time. Hence, the behavior also changes over time. Examples for model-based diagnosis in static systems are [dKW87] [Dav84]. Model-based diagnosis in dynamic systems is demonstrated in [DK91] [LN93] [Ng91] [SM95].

Besides diagnosis, major tasks of model-based reasoning are: monitoring, design, planning and explanation [Kui94].

1.2 Qualitative Reasoning

Qualitative reasoning (QR) is concerned with representing and reasoning with incomplete knowledge about physical systems. This reasoning technique is also based on a representation of the structure and the behavior of a system. However, the system is modeled on a *qualitative* level — only important distinctions are captured; details are ignored. Modeling on a qualitative level corresponds better with human reasoning than modeling on a numerical level. No one understands down to the last detail how a system works. A qualitative description is sufficient for reasoning.

In technical applications, the behaviors of dynamic systems are mostly predicted by numerical methods. The great difference between qualitative and numerical reasoning methods is that qualitative reasoning can deal with incomplete knowledge and the reasoning process is based on symbolic computation. Qualitative reasoning methods are preferred to numerical methods for the following reasons [Fis88]. First, simulation of complex numerical models requires an extremely high computing performance to be efficient. Simulation of qualitative models is less running time intensive. Second, reasoning with quantitative methods is not applicable in the following situations:

- Exact, numerical models are not available.
- The results of the reasoning process must be of a qualitative nature.
- The reasoning method must provide causal explanations for its behavioral predictions.

These situations correspond with the requirements of diagnosis in many domains. Therefore, qualitative methods are often applied in model-based diagnostic reasoning.

A lot of research has been done to apply qualitative descriptions of physical systems to problem solving. An excellent collection of work in the field of *qualitative reasoning about physical systems* can be found in [WdK90] [FS92].

1.2.1 Qualitative Reasoning Ontologies

This section briefly presents the three main ontologies in qualitative reasoning. Their modeling perspectives and their way of representing the results are discussed. This overview of the basic QR ontologies is based on [Wer94].

Device-Oriented Ontology

The device-centered approach [dKB84] concentrates on the description of physical devices. Systems are considered as more or less complex devices which consist of individual *components*, *conduits* and *terminals*. Conduits denote passive connections between components. Terminals establish the connection between components and conduits. Interconnected components constitute a larger component, which in turn may be connected to other components. Thus, the modeling approach is a hierarchical one. Furthermore, this approach respects two important principles of qualitative reasoning: *no-function-in-structure* and *locality*.

Both the modeling primitives, components and conduits, have associated sets of *confluences* which are multivariable linear equations. These equations describe the elementary behavior of the modeling primitives. The parameters of the confluences are mapped either to qualitative variables or qualitative derivations of variables. Real numbers are mapped to the set of qualitative values $\{-, 0, +\}$ by the sign operator. The qualitative derivation of a variable is defined by mapping the derivation of the corresponding real-valued variable to the set of qualitative values.

The simulation process in the device-oriented approach results in a *total envisionment*. A total envisionment describes the set of all possible states of the system and all possible transitions between these states. Therefore, a total envisionment represents all possible behaviors of the system. The total envisionment is generated by a combination of *constraint propagation* and *generate-and-test*.

Process-Oriented Ontology

The process-oriented approach [For84] represents a more abstract point of view. Systems are described by a set of *objects* and *processes*, which may change the state of the physical system, i.e., the states of the system's objects. The relationships between objects, also called individuals, are described by means of *variables* or *quantities*. Due to the concept of a process and the specific relationships between variables, this ontology — contrary to the device-centered approach — allows for modeling of causal influences.

Quantities are described by both amount (A) and derivative (D) which may have additional information about magnitude (m) and sign (s). This leads to four functions which map a real-valued variable x to symbolic values: $A_m[x], A_s[x], D_m[x]$ and $D_s[x]$. The description of the system's objects and the relationships between objects is called *individual*

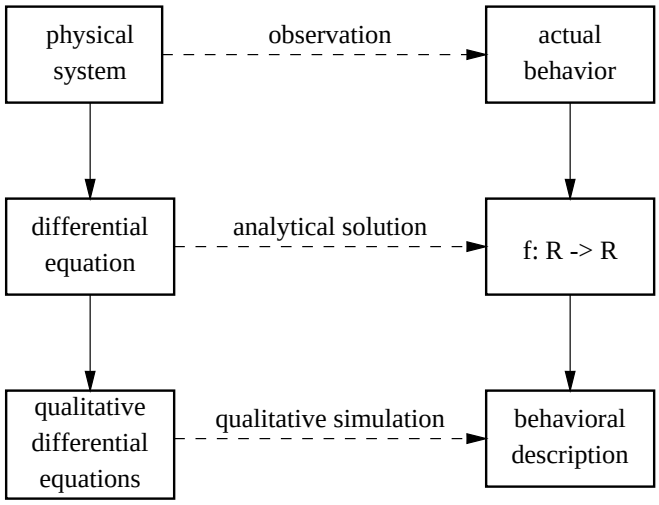


Figure 1.1: Abstraction of qualitative differential equations from differential equations.

view. Objects are changed by processes. Processes are specified by the attributes of individual views and additionally by *influences*. Influences describe how processes change quantities. They can be direct or indirect. A direct influence between quantities Q_1 and Q_2 is represented by $I^+(Q_1, Q_2)$ and $I^-(Q_1, Q_2)$. An indirect influence is denoted as $Q_1 \propto_{Q+} Q_2$ which means that Q_1 is qualitatively proportional to Q_2. The sign at the symbol $\propto_Q$ indicates whether this relationship is positive or negative.

The reasoning process in the process-oriented ontology is divided into four steps.

1. Create all possible processes and individual views.
2. Determine active processes. These are processes containing objects with quantities for which $D_s \neq 0$.
3. Resolve direct and indirect influences.
4. Analyze variables reaching limits and therefore changing predicates. This may lead to new processes and to individual views becoming active.

Constraint-Based Ontology

This approach [Kui94] is based on the well-known mathematical description of dynamic systems in form of differential equations. It starts from these equations and derives the so-called *qualitative differential equations (QDEs)*. This approach does not provide modeling primitives like components or processes as the other approaches; the focus of this approach is behavior generation. Therefore, this approach fits best into the notion of *qualitative simulation* [Kui84] [Kui86]. The algorithm QSIM is the most prominent implementation of this approach. Figure 1.1 presents the relationship between differential and qualitative differential equations as well as the concept of qualitative simulation.

In qualitative simulation the structure of a dynamic system is described by *variables* and *constraints*. Variables represent the system's parameters, which are continuous functions of time; constraints represent the relationships between variables. The qualitative

value of a variable is given by the pair $\langle qmag, qdir \rangle$ — the qualitative magnitude (qmag) and the qualitative direction (qdir). The real-valued quantity space of a variable is mapped to an ordered set of symbolic *landmark* values. The qualitative magnitude is expressed as a landmark value or an open interval between adjacent landmark values. The sign of the first derivative specifies the qualitative direction, which is given as INC (increasing), STD (steady), DEC (decreasing) or ING (ignore). Some qualitative constraints correspond to mathematical operations, like ADD, MULT or D/DT. Others express functional dependencies between variables. For example, $M^+(a, b)$ specifies that a is a monotonically increasing function of b.

Qualitative simulation generates all possible behaviors of a physical system given its structure and a description of the initial state. Contrary to the device-oriented approach, an *attainable envisionment* is generated. Qualitative simulation does not miss any real existing behavior. However, physically impossible behaviors *(spurious behaviors)* can be included in the solution. Therefore, QSIM is *sound* but *incomplete*.

1.2.2 Applications of Qualitative Reasoning Methods

Qualitative reasoning has been a very active research field of *artificial intelligence (AI)* for more than a decade [TM92] [Kui93a]. Techniques of qualitative reasoning have now become sufficiently advanced that they can be applied to real world problems.

Travé-Massuyès and Milne [TMM95] present a summary of more than 40 application oriented projects using QR techniques. These projects range from laboratory work of a research group which illustrates the potential application areas of QR up to commercially available products. Major application domains of qualitative reasoning are:

- *Continuous processes:* Projects related to continuous, often dynamic systems in process control, chemical engineering, power generation, etc.
- *Engineering:* Work related to the classical engineering fields such as mechanical engineering or manufacturing.
- *Ecology:* Projects related to natural ecological systems.
- *Electronic circuits:* Work relating to diagnosis of electronic circuits including logical and analog devices.
- *Business & commerce:* Work related to business management or financial trading.
- *Medicine:* Projects related to explanation of diagnoses in medicine.

The largest number of projects are in the traditional AI application areas of process control, engineering and medicine. However, there is also a significant number of projects in areas where precise numerical modeling is difficult, such as ecology and business. The dominant tasks in all projects are *monitoring and diagnosis*, *prediction and analysis* and *explanation*. Travé-Massuyès and Milne conclude their summary with:

> *"Over 30% of the projects are using a qualitative modeling formalism based on qualitative differential equations. This is partly because* QSIM *was made available for free very early, and partly because in the first papers, the approach*

was well grounded on mathematical foundations stated with a system theory flavor, which is familiar to the economics and engineering communities."

1.3 A Specialized Computer Architecture for QSIM

As presented in the previous section, the algorithm QSIM is used in many applications as a QR technique. QSIM is especially popular in applications which utilize monitoring and fault diagnosis in dynamic systems. The basic task in these model-based reasoning applications is the comparison of observations with the predicted behaviors of the structural model. Fault models are often used to represent faulty components of the system. Hence, the applied QR technique must only support the inference task. QSIM has its strength exactly in this area. On the other hand, when observations disagree with predictions, a hypothesized fault can be injected into the model so that the model predictions continue to track observations. This is the paradigm of *diagnosis as model modification* [DK91]. Since qualitative simulation does not deal with model building, the QSIM representation must be extended to be able to modify models. This is demonstrated in [LN93] [Ng91] [Dvo92].

There is a plan to integrate the qualitative simulation paradigm into the long-term research project *Distributed Real-Time Expert System for Fault Diagnosis in Technical Processes* of the Institute for Technical Informatics at the Graz University of Technology [GW90] [BGM+91] [SW94b] [SW95] [Ste95]. In this expert system, several local expert systems are tightly coupled with the technical process and are supervised by a global expert system. Fault-diagnosis is achieved by a model-based algorithm [Rei87]. The behavior of the technical process is derived from discrete and continuous simulators. This hybrid simulation technique [Sei92] will be extended by qualitative simulation for deriving the system behavior at a qualitative level.

However, QSIM has some drawbacks which prevent it from a wider application in industrial environments. One drawback is associated with the incompleteness of the simulation algorithm. A second drawback is concerned with computational complexity and current implementations of QSIM. These two drawbacks are described in more detail below:

- *Generation of many behaviors:*
  Qualitative simulation can result in the prediction of many different behaviors consistent with the model and the initial state description. All behaviors must be considered as alternatives for the behavior of the system and must therefore be compared with observations. The many different behaviors can be caused by underspecified models or initial states. Another reason is that the simulation algorithm is too weak to discard behaviors which are inconsistent. Especially, spurious behaviors can mislead the diagnosis algorithm. The large number of behaviors can be at least partially reduced by (i) improved filtering algorithms [KCMT91] [CK93], (ii) using semi-quantitative or numerical information [KK93] [BK92] and (iii) hierarchical structuring of the behaviors [Kui88].

- *Running time requirements:*
  QSIM as a research tool is implemented in Lisp and executed on general-purpose computers. The running time is not a major concern of these implementations.

However, performance is an important issue for applying QSIM in technical systems. In these embedded systems real-time requirements have to be fulfilled. Up to now, the computational complexity of QSIM has not been analyzed in detail and no comprehensive empirical study of the running time behavior of QSIM has been presented [Dav94]. Nothing is known about the real-time capabilities of QSIM and the integration of QSIM into real-time environments.

At the Institute for Technical Informatics, a research project has been started to develop a special-purpose computer architecture for the qualitative simulator QSIM [PR95a] [PR95b] [PRW95c] [PRW95a] [PRW95b] [FPR95] [Pla96]. The main objective of this project is to increase the running time performance of QSIM. Based on an extensive empirical analysis of the running time behavior, two strategies are supposed to improve the performance. First, QSIM functions are parallelized and mapped onto a multiprocessor system with distributed memory. Second, small but running time intensive functions are accelerated by a migration to hardware. These functions are executed on specialized coprocessors. Both strategies are combined to the overall computer architecture for QSIM. This heterogeneous computer architecture consists of a multiprocessor with embedded application-specific coprocessors.

This thesis describes the first strategy to improve the performance of QSIM. The second strategy is presented in [Pla96]. This thesis presents the parallelization of QSIM functions and the design of a scalable computer architecture for these functions. A prototype implementation demonstrates the feasibility of this design and results in a significant improvement of the running time compared to the sequential implementation.

1.4 Outline

Chapter 2 starts with a description of the basics for the algorithm QSIM. Section 2.2 presents an empirical running time analysis of QSIM. The problem task of this thesis is presented in Section 2.3. Section 2.4 reviews related work in the areas of (i) multiprocessor architectures for AI applications and (ii) parallel constraint satisfaction algorithms.

Chapter 3 presents the design of a scalable multiprocessor architecture for the QSIM kernel. It starts with a description of the design method in Section 3.1. Section 3.2 presents an analysis of the QSIM kernel. Sections 3.3 and 3.4 deal with the parallelization and the design of multiprocessor architectures for the kernel functions constraint-filter and form-all-states. Section 3.5 shows the overall multiprocessor architecture for the QSIM kernel.

Section 4.1 introduces the hardware and software platforms for the prototype implementation of the QSIM kernel multiprocessor. The multiprocessor implementation and the QSIM kernel interface are presented in Section 4.2. Sections 4.3 and 4.4 summarize and discuss the experimental results.

Chapter 5 concludes the presented thesis. The advantages and disadvantages of the prototype implementation are discussed and possible further work is presented.

Chapter 2

Problem Analysis and Related Work

2.1 Qualitative Simulator QSIM

Like any other simulator, the qualitative simulator QSIM requires a *model* and an *inference mechanism* to predict the behavior of a given system. The model is qualitatively abstracted from the physical system. In QSIM, this qualitative representation corresponds directly with the familiar elements of the theory of differential equations. Physical systems are modeled by qualitative differential equations (QDEs), which are actually differential equations but describe the system at a coarser level of abstraction than ordinary differential equations do.

This abstraction can be seen from the vertical arrows of Figure 1.1. The inference mechanism from qualitative structure to qualitative behavior is represented by the bottom vertical arrow of this figure. The following section presents basic definitions and a brief summary of the concept of qualitative simulation. A detailed summary is given in [Kui94].

2.1.1 Qualitative Representation of Behavior and Structure

Qualitative Behavior and Qualitative State

Physical systems are normally described by real-valued, continuous functions $f(t) : \Re \rightarrow \Re$. These functions have an infinite range. In qualitative simulation we are only interested in qualitatively distinct regions of the real-valued quantity space. The key intuition is that, although the world changes continuously, it changes *qualitatively* only at isolated points.

Qualitative variables, which represent time-varying quantities, are considered to range over the *extended* real number line, $\Re^*$, which includes the endpoints $-\infty$ and ∞. In order for qualitative reasoning to be possible, variables must be restricted to correspond to functions of time whose behavior is *reasonable*.

Definition 2.1 *Where* $[a, b] \subseteq \Re^*$, *the function* $f : [a, b] \rightarrow \Re^*$ *is a* reasonable *function over* $[a, b]$ *if*

1. f *is continuous on* $[a, b]$,

2. *f is continuously differentiable on (a, b),*

3. *f has only finitely many critical points in any bounded interval and*

4. *the one-sided limits $lim_{t \to a^+} f'(t)$ and $lim_{t \to b^-} f'(t)$ exist in $\Re$. Define $f'(a)$ and $f'(b)$ to be equal to these limits.*

A *quantity space* captures the intuition that, for reasonable functions at least, there are only a few qualitatively important *landmark values*.

Definition 2.2 *A* quantity space *is a finite, totally ordered set of symbols, the* landmark values $l_1 < l_2 < \cdots < l_k$.

Each landmark is a symbolic name, representing a particular value in $\Re^*$. A quantity space contains the landmarks $-\infty, 0$ and ∞. It must also contain a landmark value for each critical point of $f(t)$.

Time has the quantity space $t_0 < t_1 < \cdots < t_n < \infty$.

A qualitative variable v and its quantity space $l_1 < l_2 < \cdots < l_k$ define a qualitative description of the values of a function $f(t)$. At any time t, the qualitative value of $f(t)$ can be described in terms of its ordinal relationship with the landmarks in its quantity space and its direction of change. The qualitative values for a variable are the landmark values and the open intervals bounded by adjacent landmark values.

Definition 2.3 *The* qualitative value *of $f(t), QV(f, t)$, with respect to the quantity space $l_1 < l_2 < \cdots < l_k$ is the pair $\langle qmag, qdir \rangle$, where*

$$qmag = \begin{cases} l_j & \text{if } f(t) = l_j, \text{ a landmark value} \\ (l_j, l_{j+1}) & \text{if } f(t) \in (l_j, l_{j+1}) \end{cases}$$

$$qdir = \begin{cases} inc & \text{if } f'(t) > 0 \\ std & \text{if } f'(t) = 0 \\ dec & \text{if } f'(t) < 0. \end{cases}$$

Since the function $f(t)$ changes continuously, the qualitative value within an interval bounded by adjacent time-points remains the same. Hence, the qualitative description of the value of $f(t)$ changes only at discrete points.

Definition 2.4 *A time-point $t \in [a, b]$ is a* distinguished *or* landmark time-point *of f if t is a boundary element of the set $\{t \in [a, b] \mid f(t) = x$, where $x \in \Re^*$ is represented by a landmark value of $f\}$.*

For adjacent distinguished time-points t_i and t_{i+1}, define $QV(f, t_i, t_{i+1})$, the qualitative value of f on (t_i, t_{i+1}), to be $QV(f, t)$ for any $t \in (t_i, t_{i+1})$.

Definition 2.5 *The* qualitative behavior *of $f(t)$ is the sequence of qualitative values $QV(f, t_0), QV(f, t_0, t_1), QV(f, t_1), \ldots, QV(f, t_{n-1}), QV(f, t_n)$, alternating between qualitative values at time-points, and qualitative values on intervals between adjacent time-points.*

By means of the qualitative behavior and the qualitative value, the qualitative state of a system can be defined.

Definition 2.6 *A* system *is a set $F = \{f_1, \ldots, f_m\}$ of functions $f_i(t)$, each with its own set of landmarks and distinguished time-points. The distinguished time-points of a system F are the union of the distinguished time-points of the individual functions $f_i \in F$. The* qualitative state *of a system F of m functions is the m-tuple of individual qualitative states:*

$$\begin{aligned} QS(F, t_i) &= \langle QV(f_1, t_i), \ldots, QV(f_m, t_i)\rangle \\ QS(F, t_i, t_{i+1}) &= \langle QV(f_1, t_i, t_{i+1}), \ldots, QV(f_m, t_i, t_{i+1})\rangle \end{aligned}$$

Every state of the system F has a qualitative description $QS(F, t)$, but that description changes only at discrete landmark time-points, and remains constant on the open intervals between them. Thus, the concept of a "next state" of F is meaningful in qualitative simulation in contrast to continuous simulation.

Qualitative Constraints

The state of a system at a time t is described in terms of a set of variables $\{x, y, \ldots\}$. The relationships among these variables are expressed by qualitative constraints that hold for each t. There are several different types of constraints in QSIM. The most important ones can be enumerated as follows:

Mathematical Relations

$$\begin{aligned} ADD(x, y, z) &\equiv x(t) + y(t) = z(t) \\ MULT(x, y, z) &\equiv x(t) * y(t) = z(t) \\ D/DT(x, y) &\equiv \tfrac{d}{dt}x(t) = y(t) \end{aligned}$$

Functional Relations

$$\begin{aligned} M^+(x, y) &\equiv y(t) = f(x(t)) \wedge x'(t) > 0 \\ M^-(x, y) &\equiv y(t) = f(x(t)) \wedge x'(t) < 0 \end{aligned}$$

Operations, like addition and multiplication, over real numbers are *functions* that take two real values and return a third: $\Re \times \Re \rightarrow \Re$. Due to the ambiguity of these operations over qualitative values, qualitative constraints are represented as *relations*. These constraints take one more parameter than the corresponding functions and return a truth value, e.g., $ADD : \Re \times \Re \times \Re \rightarrow \{true, false\}$.

Qualitative Differential Equations (QDEs)

Constraints are key elements for abstracting a system described with *ordinary differential equations (ODEs)* into a qualitative description. This corresponding but weaker description is called *qualitative differential equation (QDE)*. "Weaker" means that any behavior that satisfies a ODE must satisfy the corresponding QDE, but not necessarily vice versa.

Given a ODE, it can be decomposed into an equivalent set of equations by introducing terms for each subexpression. When this process is completed, each equation can be mapped to a qualitative constraint. This abstraction is shown in the following ODE:

$$\frac{d^2u}{dt^2} - \frac{du}{dt} + \arctan ku = 0 \tag{2.1}$$

The subexpressions and the qualitative constraints are derived as follows:

subexpressions	*qualitative constraints*
$v_1 = du/dt$	$D/DT(u, v1)$
$v_2 = dv_1/dt$	$D/DT(v_1, v_2)$
$v_3 = ku$	$MULT(k, u, v_3)$
$v_4 = arctan(v_3)$	$M^+(v_3, v_4)$
$v_2 - v_1 + v_4 = 0$	$ADD(v_2, v_4, v1)$

The quantity spaces for the variables $\{u, v_1, \ldots, v_4, k\}$ are constructed by adding landmarks to the base quantity space $\{-\infty, 0, \infty\}$. These landmarks represent the values of constants as well as domain and range boundary values for monotonic functions. Each constraint is equivalent to the corresponding subexpression, with the exception of the M^+ constraint, which is less restrictive. The introduction of the monotonic function constraint requires that the corresponding subexpressions have a non-zero derivative on the interior of their domains.

This syntactic procedure for decomposing an ODE into subexpressions generates a unique set of constraints from a given ODE. However, different monotonic functions may be mapped to the same M^+ constraint. So multiple ODEs can be derived to the same QDE.

The QDE describes qualitatively the *structure* of a physical system. More formally, a QDE is defined as:

Definition 2.7 *A* qualitative differential equation (QDE) *is a tuple of four elements,* $\langle V, Q, C, T \rangle$*, each of which will be defined below.*

- V *is a set of* variables*, each of which is a reasonable function of time.*
- Q *is a set of* quantity spaces*, one for each variable in* V*.*
- C *is a set of* constraints*, applying to the variables in* V*. Each variable must appear in some constraint.*
- T *is a set of* transitions*, which are rules defining the boundary of the domain of applicability of the QDE.*

Corresponding Values

Corresponding values are tuples of landmark values that the variables in some constraint can take on at the same time. They provide a mutual constraint between the meaning of a QSIM constraint and the meaning of landmark values in the quantity spaces of the variables. In the case of an ADD constraint, a corresponding value triple (p, q, r) provides a constraint $p + q = r$ on the real numbers that the landmarks p, q, and r can stand for. In the case of incompletely specified monotonic function constraints such as M^+ and M^-, a corresponding value pair (p, q) provides a mutual constraint on the possible values of p and q, and on the shape of the functional relation.

Corresponding values provide additional information about specific constraints. This helps to eliminate certain combinations of possible values of the constraint's variables.

2.1.2 Deriving Behavior from Structure

The task of simulation is to predict the behavior of a system, given only its structure and a description of the initial state. In qualitative simulation, a behavior is a sequence of states that represents the temporal evolution of the system. This sequence alternates between states representing time-points and states representing time-intervals (compare Definitions 2.5 and 2.6).

Qualitative simulation starts with a QDE and a description of the initial state, and predicts one or more possible behaviors.

$$QDE \wedge \ QState(t_0) \rightarrow \{Behavior_1, \ldots, Behavior_m\} \tag{2.2}$$

This generation of possible behaviors requires basically the solution of two different problems.

1. Given a QDE and partial information about the initial state, determine all complete, consistent qualitative states $QState(t_0)$.

2. Given a QDE and a qualitative state, determine its possible immediate successors.

$$\begin{aligned} QState(t_i) &\rightarrow \{QState_1(t_i, t_{i+1}), \ldots, QState_{n_i}(t_i, t_{i+1})\} \\ QState(t_i, t_{i+1}) &\rightarrow \{QState_1(t_{i+1}), \ldots, QState_{m_i}(t_{i+1})\} \end{aligned}$$

The basic algorithm of qualitative simulation is shown in Figure 2.1. *Initial state processing* generates all states consistent with the QDE and the partial description of the initial state. These states are stored in an agenda for further processing. The immediate successors of each state are generated by three successive steps. First, the possible qualitative values of all variables are determined for the next time-point or the next time-interval *(generate possible values)*. Then, all candidates for successor states are generated using these possible values and the QDE *(*QSIM *kernel)*. Finally, *global filters* are applied to test for consistency of individual successors or entire behaviors. Consistent successor states are also stored in the agenda. The generation of successor states is continued until all states have been processed or a resource limit has been exceeded.

Immediate Successors of a State

The generation of the immediate successors of a state is based on a simple intuition. Since a qualitative variable and its derivative are continuous functions of time, there are only a few possible transitions from one qualitative value to the next. This intuition is formalized to a set of successor rules which determines all possible qualitative values for the successor states. These successor rules are directly derived from the Intermediate Value and Mean Value Theorems from the differential calculus.

The successor of a qualitative value is not uniquely determined. However, a variable has never more than four possible successor values, and in many cases the successor value is unique. A detailed description of the successor rules can be found in [Kui94].

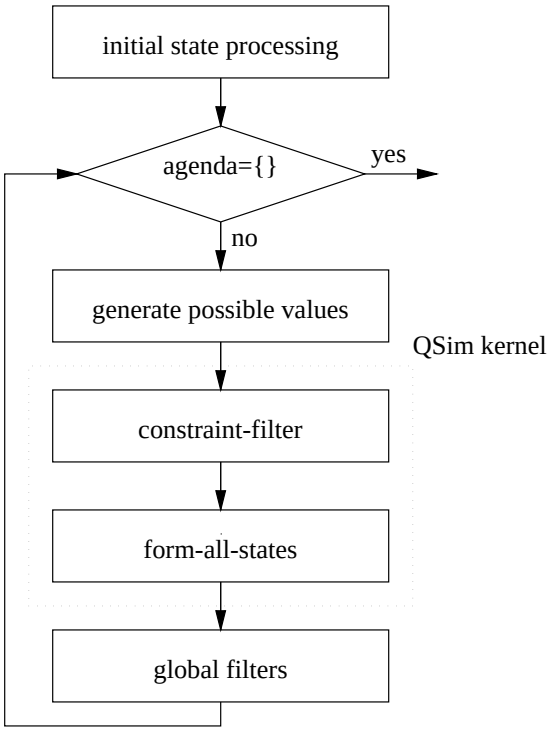


Figure 2.1: Flow chart of QSIM.

Generation of Successor States

QSIM generates a tree of states from one or more initial states linked by the immediate-successor relation. A path from the root to a leave represents one behavior of the system.

In most cases, we are interested in the prediction of all possible behaviors. However, there is no guarantee that the agenda will ever get empty. There are known sources of intractable branching that can generate an infinite number of infinite long behaviors. In these cases, QSIM will run forever [Kui86].

Intractable branching is a result of the generation of too many — possibly spurious — successor states. Thus, to avoid or reduce intractable branching, successor candidates must be filtered for consistency. These filters should discard as many spurious behaviors as possible.

Global Filters

Each candidate for a successor state is passed through a sequence of global filters. Global filters infer new information about that candidate state which was not visible to the candidate generation in the QSIM kernel. Two important cases are considered.

- Some global filters infer the inconsistency of a particular state and discard this state from further processing.
- Other global filters transform the qualitative description into a more specific one without loss of validity or generality — for example by creating new landmarks or

identifying new corresponding value tuples.

Some global filters are necessary for QSIM, while many of them are optional extensions to reduce branching or to infer additional information.

State-based filters use only information from the current state description or from the immediate predecessor state.

- The *no change* filter deletes a time-point state at which no variable changes its qualitative value.
- The *infinite values* filter identifies states which violate constraints on possible combinations of finite and infinite values. Different constraints must be checked if the time-point represented by the current state represents $t = \infty$ or $t < \infty$.
- The *quiescence* filter recognizes states where all directions of change are *std.*
- The *make new landmark* filter generates a new landmark value when a new critical value is detected.
- The *make new corresponding values* filter creates a new corresponding value tuple when all variables of a constraint take on landmark values at a time-point.
- The *filter for transitions* identifies limits of the current QDE by comparing qualitative values with transition rules. Whenever these rules are satisfied, the operating region is changed, i.e., qualitative values are change discontinuously and/or the QDE is changed.
- The *higher-order derivative* filter derives additional constraints for higher-order derivatives to eliminate spurious branches *(chatter)* from critical points.

Behavior-based filters consider global information about the entire behavior terminating in the current state.

- The *check for cycles* filter detects a cyclic behavior by comparing the current state with its predecessors in the behavior tree. Such states are excluded from further processing.
- The *non-intersection* filter eliminates behaviors with trajectories in a phase-plane representation that are intersecting themselves.
- The *global energy* filter detects behaviors which violate the conservation of energy.
- The *quantitative range-reasoning* filter processes quantitative information added to the qualitative description of the system. This information is added to the landmark values and other terms in the behavior description. This is propagated to derive inconsistencies.

2.1.3 QSIM Kernel

Constraint Satisfaction

The task of the QSIM kernel is to find all candidates for successor states given the QDE and all possible qualitative values for all variables. This can be formulated more generally as a *constraint satisfaction problem (CSP).* Constraint satisfaction is a well-known term for a variety of techniques for AI and related disciplines. Further information on constraint-satisfaction can be found in [Mac77] [Mac92].

Definition 2.8 *A* constraint-satisfaction problem (CSP) *is defined as a triple* $\langle V, D, P\rangle$ *consisting of*

- *a set* $V = \{v_1, \ldots, v_n\}$ *of variables,*
- *a set* $D = \{D_1, \ldots, D_n\}$ *of domains, such that each* D_i *is a set of values for the variable* v_i *and*
- *a set* $P = \{P_1, \ldots, P_m\}$ *of constraint relations, where each* P_j *refers to some subset of the variables* V*. The constraint relations determine subsets of the Cartesian product of the domains of the variables involved.*

An assignment for a unique domain value to each member of some subset of variables is called instantiation*. An instantiation is said to be legal or locally consistent if it does not violate any of the (relevant) constraints. A legal instantiation of all variables* V *is called a solution of the CSP.*

In qualitative simulation, a constraint-satisfaction problem $\langle V, D, P\rangle$ can be formulated easily from the QDE.

- The variables $v_i \in V$ are the variables of the QDE.
- The domain D_i for a variable v_i is the set of possible qualitative values for v_i. This set is constructed by the initial value information or by the successor rules. The domain D_i is always finite — in the generation of successor states the cardinality of the domain is limited by 4.
- The set P of constraints corresponds to the constraints of the QDE.

A CSP can also be represented as a graph. Figure 2.2 presents the constraint graph of the QDE transformed from Equation 2.1. This representation of QSIM QDEs is dual to the original formulation in [Mac77]. This is due to the fact that QSIM constraints are not restricted to arity$\leq$2. In the QSIM graph, the constraints are the nodes, and the variables are the edges.

Thus, the basic inference mechanism in QSIM can be viewed as generating and solving appropriate constraint-satisfaction problems. *Generate-and-test* is a straightforward algorithm to solve any CSP with finite domains. However, this algorithm is intractable, since the number of tests is the product space of the domains. If there are V variables, each with a domain of d, the number of tests is d^V.

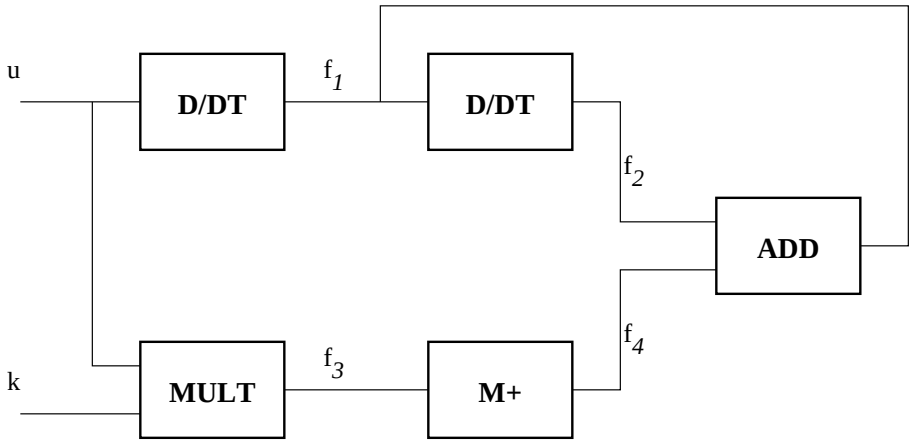


Figure 2.2: Representation of a CSP as constraint network.

Backtracking algorithms systematically explore D by sequentially instantiating the variables in some order. Thus, individual constraint violations can eliminate entire subspaces of the product space, rather than just a single test. The worst-case performance is still exponential, but typical performance is vastly improved.

Another family of algorithms is used to improve the performance of backtracking. These are known as *node-, arc-* and *path-consistency* algorithms [MF85]. The consistency algorithms serve as preprocessing steps for backtracking to eliminate local inconsistencies, and hence reduce the search space for the backtracking step. Node consistency considers p-tuples of variables associated to individual constraints of arity p, thus removing those that violate the constraints. Arc consistency ensures that tuples assigned to adjacent nodes assign the same value to shared variables. In QSIM, path consistency is currently not exploited.

The Kernel Functions

In this section, the QSIM kernel functions are described in more detail. The description is based on the pseudo code of the kernel functions. This pseudo code is derived from the original Lisp source code of the QSIM package. There are some minor modifications to make the pseudo code more readable. First, some basic functions, which are called from the kernel functions, are defined.

$\texttt{ccf}_{con}\texttt{(p}_1\texttt{, p}_2\texttt{, p}_3\texttt{)}$ is the so-called *constraint check function* for an individual constraint. It checks if the p-tuple of possible qualitative values is consistent with the constraint `con`. The constraint check function returns a truth value. Each parameter of the function corresponds to one variable of the constraint. Therefore, the constraint for the shown check function has an arity of 3. The check functions require access to the corresponding value tuples of the constraints. The corresponding value tuples are not passed via parameters to the check functions.

`qval(var)` returns the current qualitative value of the variable `var`. During the execution of the kernel, the variables can be instantiated with different qualitative values.

```
1    procedure cfilter()
2    begin
3      for all constraints con do
4        tuple-filter(con)
5        if (waltz-filter(con) = FALSE) then
6          return FALSE
7        endif
8      endfor
9      sort all constraints using a boundary-sort heuristic
10     form-all-states(first-constraint())
11   end
```

Figure 2.3: Pseudo code for the function cfilter.

`tuple-qval(tuple, var)` returns the qualitative value of the variable `var` in the tuple `tuple`.

`pvals(var)` returns the current set of possible qualitative values of the variable `var`. Due to filtering, the number of possible values can decrease during simulation.

`first-constraint()` `next-constraint(con)` `last-constraint()`. In general, the constraints can be processed by the backtracking algorithm in an arbitrary order. In QSIM, the constraints are ordered by a special heuristic to improve the backtracking. These functions return specific constraints corresponding to their order.

In QSIM, successor candidates are generated by the cfilter function. Hence, the cfilter solves the CSP determined by the QDE and the possible values. The cfilter algorithm is shown in Figure 2.3. Basically, it consists of three consecutive functions. The tuple-filter checks each tuple of possible values against the conditions of an individual constraint. The tuple-filter is applied to all constraints, hence, it is a node-consistency algorithm. The Waltz-filter ensures the arc-consistency of the constraint network. Finally, form-all-states finds all solutions for the CSP by a simple backtracking algorithm. These three kernel functions are described in more detail below:

`tuple-filter`. The tuple-filter algorithm for a constraint of arity 3 is presented in Figure 2.4. All tuples — in this case triples — of possible values are generated by 3 nested loops. Each tuple is tested for local consistency by the constraint check function $\texttt{ccf}_{con}$ (line 7). The constraint check function returns TRUE when the tuple satisfies the conditions of the individual constraint. Each type of constraint has its own constraint check function due to its different conditions. All consistent tuples are added to the tuple set of the individual constraint (line 8).

`Waltz-filter`. The Waltz-filter checks all tuple sets of adjacent constraints — all constraints which share a variable — for arc-consistency. Actually, the QSIM implementation shown in Figure 2.5 does not correspond to the algorithm of Waltz [Wal75], it is the algorithm AC–3 of Mackworth [Mac77]. In the Lisp code, the Waltz-filter is implemented recursively. However, the code in Figure 2.5 presents an implementation using a global stack. First, all constraints which have already been tuple-filtered

```
1    procedure tuple-filter(con)
2    begin
3      tuples ← {}
4      for all pvals p_i of variable v_i of con do
5        for all pvals p_j of variable v_j of con do
6          for all pvals p_k of variable v_k of con do
7            if (ccf_con(p_i, p_j, p_k) = TRUE) then
8              tuples ← tuples ∪ (p_i, p_j, p_k)
9            endif
10         endfor
11       endfor
12     endfor
13   end
```

Figure 2.4: Pseudo code for the tuple-filter.

are pushed onto the stack (line 3–5). Then all tuple sets of adjacent constraints are successively checked. If a value of the shared variable is not in the actual domain of this variable, this tuple is discarded (line 11–13). If all tuples of an adjacent constraint have been deleted, the Waltz-filter is aborted (lines 15–17).

Due to the deletion of tuples, the number of possible values can decrease during filtering. For simplicity, the update of the set of possible values after the deletion of a tuple is not shown in the pseudo code. The reduced domain may also affect adjacent constraints of this constraint. Thus, adjacent constraints are pushed onto the stack (lines 18–20). The filtering process is continued until the stack is empty.

`form-all-states.` The function form-all-states is presented in Figure 2.6. It is a simple backtracking algorithm which processes all constraints in an arbitrary order. Whenever this search algorithm reaches the last constraint, a solution of the CSP is found (lines 3–6). Backtracking performs a depth-first search. If the variables' values of a tuple do not differ from the values of already instantiated variables (line 10–18), then form-all-states is called recursively with the next constraint as parameter (lines 19–21). Whenever all tuples of an individual constraint violate the values of already instantiated variables, backtracking occurs. After checking one tuple, the values of all variables from that constraint must be restored to their old values (line 22).

The order in which form-all-states processes the constraints has a great influence on the performance. QSIM uses a *boundary-sort* heuristic to sort the constraints. This heuristic first selects all constraints with one tuple. If no constraint has a tuple set with one element, the constraint with the smallest tuple set is chosen. Then, all constraints directly connected to the already selected constraints are considered; the constraint with the smallest tuple set is selected next. This step is repeated until all constraints are processed. The order in which the constraints are selected determines the order of the constraints for the backtracking algorithm.

In the cfilter, the tuple-filter and the Waltz-filter are interleaved for increased efficiency. Thus, the Waltz-filter is only applied to already tuple-filtered constraints of the entire

```
1   procedure waltz-filter()
2   begin
3     for all constraints con already filtered do
4       push(con)
5     endfor
6     while (stack ≠ {})
7       con ← pop()
8       for all variables var of constraint con do
9         for all attached constraints acon of con do
10          for all tuples t of constraint con do
11            if (tuple-qval(t, var) ∉ pvals(var)) then
12              delete t from con
13            endif
14          endfor
15          if (acon has no more tuples) then
16            return FALSE
17          endif
18          if at least one pval of var has been deleted then
19            push(acon)
20          endif
21        endfor
22      endfor
23    endwhile
24    return TRUE
25  end
```

Figure 2.5: Pseudo code for the Waltz-filter.

```
1   procedure form-all-states(con)
2   begin
3     if (con = last-constraint()) then
4       generate-solution()
5       return
6     endif
7     save qvals of all variables of con
8     for all tuples t of con do
9       consistent ← TRUE
10      for all variables var of con do
11        if qval(var) is not instantiated then
12          qval(var) ← qval(tuple-qval(t, var))
13        else
14          if (qval(var) ≠ qval(tuple-qval(t, var))) then
15            consistent ← FALSE
16          endif
17        endif
18      endfor
19      if (consistent = TRUE) then
20        form-all-states(next-constraint(con))
21      endif
22      restore qvals of all variables of con
23    endfor
24  end
```

Figure 2.6: Pseudo code for the function form-all-states.

function	*elementary operations*	*complexity*
generate possible values	dV	$O(V)$
tuple-filter	$d^a tC$	$O(tC)$
Waltz-filter	$ka^2d^ad^2C^2$	$O(kC^2)$
form-all-states	d^V	$O(d^V)$
global filters	tV	$O(tV)$

Table 2.1: Algorithmic complexity of one iteration step in QSIM.

constraint network. Due to this *incremental* Waltz-filtering, domains of variables can be reduced. Thus, constraints which share these variables but have not yet been tuple-filtered, do have smaller tuple sets than those without incremental Waltz-filtering. The corresponding constraint check functions have to be called less often.

2.1.4 Estimation of the Algorithmic Complexity

This section presents an estimation of the algorithmic complexity of QSIM. The number of generated behaviors and the number of generated states strongly depend on the simulation model and the initial state description. Hence, the following complexity estimation considers only *one* loop iteration of the basic QSIM algorithm (Figure 2.1 — i.e., the generation of successor states).

The algorithmic complexity of QSIM can be estimated as follows. Suppose, there are V variables and C constraints given in the QDE. Each constraint c_i has arity a_i, and each variable v_i is attached to k_i constraints. The maximum values of these numbers are $a = max\{a_1, \ldots, a_C\}$ and $k = max\{k_1, \ldots, k_V\}$. Furthermore, d specifies the maximum number of possible values of all variables and t the length of the longest behavior in the behavior tree. Table 2.1 presents an overview of the algorithmic complexity of the most important QSIM functions. The maximum number of elementary operations and the complexities are shown. For this complexity estimation, the maximum cardinality of the domain $d = 4$ and the maximum arity of the constraints $a = 3$[1]. The constraint network is considered as a sparse graph, hence $\sum_{i=1}^{V} k_i = O(C)$.

- In *generate possible values*, at most d possible values are assigned to each variable of the QDE. Hence, dV elementary operations are required.

- There are at most d^a tuples which must be checked for consistency by the constraint check functions. The number of elementary operations in the constraint check functions is linear with the number of corresponding value tuples. Corresponding value tuples also grow linearly with the length t of the behavior. For all *tuple-filters*, $d^a tC$ elementary checks are necessary.

- In the *Waltz-filter*, the tuple sets of each constraint on the stack must be checked for consistency. For checking one individual tuple set with the set of possible values, $d^a d$ comparisons are required. For each constraint, the tuple sets of ak adjacent

[1] There are so-called *multivariate* constraints which have an arbitrary number of variables. Multivariate constraints are not considered in this estimation.

constraints are considered. The number of constraints pushed onto the stack can be estimated as follows. Whenever the sets of possible values decrease, all attached constraints are pushed. Hence, the total number of "attached" constraints is given by $\sum_{i=1}^{V} k_i = \sum_{j=1}^{C} a_j \leq aC$. Each variable can decrease its domain at the most by d times. Thus, at most daC constraints are pushed onto the stack, and the overall number of comparisons for *one* Waltz-filter call is estimated $daC \times akd^a d = ka^2 d^a d^2 C$. Finally, the incremental Waltz-filter is executed after each tuple-filter, and at the most $ka^2 d^a d^2 C^2$ comparisons are required.

- Form-all-states finds all solutions of the CSP by a simple backtracking algorithm. In general, finding a solution of a CSP is *NP-complete*[MH86]. The number of solutions can also be exponential with the number of variables V. Hence, the worst-case complexity of form-all-states is given by $O(d^V)$.

- The most expensive global filter is *check for cycles* which compares all previous states of the behavior with the actual state. Hence, tV comparisons are required.

2.2 Empirical Running Time Analysis

The analysis of the algorithmic complexity provides worst-case limits for the running time of individual algorithms with regard to their problem size. These worst-case running times need not correspond with the actual running time behavior of the algorithm. Thus, a pure theoretical analysis may be misleading for the design of a special-purpose computer architecture. On the other hand, an empirical study describes the running time behavior under *real* conditions, and *special-case* or *average-case* running times are determined. The empirical analysis provides a profile of the running times of the considered functions of QSIM. The profile enables an estimation of the effect of running time improvements of individual functions on the total running time. Furthermore, an empirical analysis is extremely important because the running time behavior of QSIM depends strongly on input data — the QDE and the initial state description.

An empirical study of the behavior of QSIM is also very important for the application of QSIM in *real* systems. In real applications, running time becomes a major requirement. The effects of the various extensions or the dependence of the running time on the problem characteristic are also easier to derive empirically than theoretically.

2.2.1 Generation of Successor States

The running times of QSIM functions were measured on a QSIM implementation installed on a Texas Instruments Explorer Lisp-workstation. For this purpose, the QSIM software was instrumented by timing functions. During simulation, these timing functions recorded the required running times for each considered function. When a function is called several times, the individual running times are summed up. Various simulation models from the QSIM package were simulated to provide input data with different characteristics and complexity. The number of variables and constraints of these models varies from 3 to 28. Table 2.2 shows an overview of the considered simulation models. The number of variables and constraints is given for each QDE. A few models consist of more QDEs, each

model	*# QDEs*	*# variables*	*# constraints*
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

Table 2.2: Complexity of the simulation models considered for the running time measurements.

valid for a specific operation region. The top 4 models in Table 2.2 represent simple models whereas the models HEART, STLG and 4-REACTIONS represent medium complex models. The BATHTUB model is simulated using 5 different initial state descriptions. With BATHTUB-1 to BATHTUB-3 only one initial state can be determined. Simulation of BATHTUB-4 and BATHTUB-5 results in more initial states. The models were simulated without any extension of QSIM. Basically, only the following global filters were executed: *no change, infinite values, quiescent state, new landmarks, new corresponding value tuples, operation region transitions,* and *cyclic behavior.* A brief description of the simulation models can be found in [Rin93].

Table 2.3 presents the running time behavior of the entire QSIM algorithm. The individual functions correspond to the flow chart of QSIM in Figure 2.1. The influence of initial state processing *(init)* and generation of possible values *(pvals)* is rather small. The cfilter function dominates the running time of QSIM. This observation is especially valid for complex models, where cfilter requires more than 65 % of the total running time. In the model TOASTER, the global filters have the greatest influence on the total running time. This is caused by an increased filtering effort for operation region transitions. The running time ratio for initial state processing remains nearly constant for all BATHTUB models. Due to the incomplete description of the initial state in the models BATHTUB-4 and BATHTUB-5, simulation of these models results in 9 and 30 states, respectively, compared to 3 states for the other BATHTUB models. Hence, the running time for initial state processing increases with respect to the running time for successor generation of *one* state.

Table 2.4 shows the ratio of the running times for the cfilter function during the generation of successor states. The tuple-filter dominates the total running time of the cfilter — in each model, more than 50 % of the running time is required by the tuple-filter.

2.2.2 Initial State Processing

In initial state processing, the set of possible values is not restricted to 4 elements. The quantity space limits the domain of a variable. Given n landmark values, there are $3(2n-1)$

model	*overall*	*init*	*pvals*	*cfilter*	*global filters*
BATHTUB-1	0.268 s	8.9 %	5.2 %	39.3 %	42.3 %
BATHTUB-2	0.284 s	8.5 %	4.9 %	43.7 %	40.0 %
BATHTUB-3	0.201 s	9.4 %	7.0 %	52.4 %	28.2 %
BATHTUB-4	1.07 s	10.4 %	5.2 %	41.0 %	40.7 %
BATHTUB-5	3.485 s	10.3 %	5.6 %	45.1 %	35.8 %
BOUNCING-BALL	2.842 s	7.9 %	4.6 %	42.2 %	38.2 %
SIMPLE-BALL	2.032 s	10.5 %	4.2 %	44.9 %	40.3 %
TOASTER	1.695 s	6.6 %	4.5 %	26.9 %	56.9 %
HEART	3.827 s	1.5 %	3.4 %	69.3 %	23.9 %
STLG	0.778 s	3.3 %	5.1 %	67.2 %	23.3 %
4-REACTIONS	569.506 s	2.9 %	2.5 %	83.0 %	8.4 %

Table 2.3: Running time ratio of the overall QSIM algorithm.

model	*tuple-filter*	*Waltz-filter*	*form-all-states*
BATHTUB-1	59.4 %	21.0 %	19.6 %
BATHTUB-2	64.6 %	17.7 %	17.7 %
BATHTUB-3	64.3 %	24.3 %	11.4 %
BATHTUB-4	62.5 %	22.1 %	15.4 %
BATHTUB-5	59.7 %	25.2 %	15.1 %
BOUNCING-BALL	73.2 %	19.5 %	7.3 %
SIMPLE-BALL	74.6 %	18.2 %	7.2 %
TOASTER	60.7 %	27.3 %	12.0 %
HEART	75.3 %	17.4 %	7.3 %
STLG	71.1 %	21.4 %	7.5 %
4-REACTIONS	67.5 %	11.0 %	21.5 %

Table 2.4: Running time ratio of the cfilter function at generation of successor states.

model and state	*overall*	*tuple-filter*	*Waltz-filter*	*form-all-states*
BATHTUB-5 0	0.0362 s	32.2 %	4.6 %	63.2 %
HEART 2	0.1852 s	4.7 %	1.0 %	94.4 %
HEART 3	0.6869 s	3.6 %	0.7 %	95.7 %
HEART 17	0.4538 s	5.5 %	1.1 %	93.4 %
HEART 39	0.6928 s	3.6 %	0.7 %	95.7 %
HEART 53	0.7311 s	3.4 %	0.7 %	95.9 %
HEART 67	0.6239 s	4.0 %	0.8 %	95.2 %
HEART 81	0.6809 s	3.7 %	0.7 %	95.6 %
HEART 91	0.1860 s	5.0 %	1.0 %	94.0 %

Table 2.5: Running time ratio of the cfilter function at initial state processing.

possible qualitative values — $n-1$ additional intervals between the landmarks, and 3 qdirs for each landmark and each interval. Thus, the overall search space for the CSP is much bigger than in the successor state generation. Initial state processing does not only occur in the beginning of the simulation process. Whenever the operation region is changed, all initial states of the new QDE must be determined. Furthermore, there are some extensions in QSIM which perform initial state processing, too. These extensions are used to avoid *chatter* [CK93].

The measurement of initial state processing differs from the measurement of the generation of successor states. The running times were measured only for one cfilter call, and the cfilter function was implemented in C running on a digital signal processor TMS320C40. Despite these differences, the running time behavior of the cfilter remains nearly the same compared to the running times of the Lisp implementation. Table 2.5 presents the running time ratios of the kernel functions. The running time ratios are given for the initial state processing of BATHTUB-5 and some cfilter calls for the HEART model. The HEART model was simulated using extensions to avoid chatter.

In initial state processing, the running time behavior differs completely from the behavior of generating successor states. Due to the bigger search-space of the CSP, form-all-states dominates the running time of the QSIM kernel. In all states of the HEART model, form-all-states requires more than 90 % of the kernel running time. The influence of the Waltz-filter can be neglected.

Summary of the Empirical Running Time Analysis

The results of the empirical running time analysis can be summarized as follows:

- The QSIM kernel dominates the running time of the overall algorithm. Thus, to improve the performance of QSIM, the QSIM kernel must be accelerated.

- In generation of successor states, the tuple-filter dominates the running time of the QSIM kernel. An exponential behavior of the NP-complete function form-all-states could not be observed experimentally.

- The situation is completely different for initial state processing. In these cases, form-all-states is the most running time intensive function of the kernel. Acceleration of form-all-states is extremely important to improve the performance of the QSIM kernel.

2.3 Problem Task

The objective of this thesis can be specified in more detail based on the results of the empirical running time analysis in Section 2.2 and the coarse description of the overall research project in Section 1.3. This thesis deals with the development of a scalable multiprocessor for the kernel of the qualitative simulator QSIM. This development is divided into three project phases: design, prototype implementation and experimental evaluation.

Design

- The special-purpose computer architecture is designed for both kernel functions constraint-filter and form-all-states. The main objective of the design is to minimize the execution time of the QSIM kernel. This objective is achieved by parallelization of the kernel functions. Optimizations of the sequential code of QSIM are not considered in this thesis.
- The specialized computer architecture is scalable. This means that the number of processing elements can vary, and the performance increases with the number of processing elements.
- The specialized computer architecture does not depend on the structure of the input simulation model (QDE). Therefore, all QSIM models — if they do not exceed any system resources — can be executed on this computer architecture.
- The design considers specialized coprocessors for constraint check functions and tuple-filter [Pla96]. These coprocessors accelerate the execution of the corresponding kernel functions.

Prototype Implementation

- A prototype of the multiprocessor is implemented on an appropriate software and hardware platform. The implementation platform supports the scalable design.
- In the prototype, only tuple-filters are implemented for the constraint types D/DT, M^+, M^-, ADD and MULT.

Experimental Evaluation

- The experimental evaluation is based on the measurement of the execution times on the prototype of the specialized multiprocessor.
- The execution times are compared with the execution times of the sequential kernel functions running on one processing element of the multiprocessor system. The execution times on the multiprocessor system are not compared with the running times on a Lisp-workstation.

2.4 Related Work

This section reviews related work of this thesis in *multiprocessor architectures for AI applications* and *parallel constraint satisfaction algorithms*. As presented in [Pla96], the research project "A Specialized Computer Architecture for Qualitative Simulation" is the first work known so far which improves the running time of QSIM by means of a specialized computer architecture. There is also very little work done in this field considering the computational complexity of qualitative reasoning applications and improving performance by developing specialized computer architectures. It seems that computational complexity does not (yet) strongly concern the QR research [Dav94]. For the wider AI

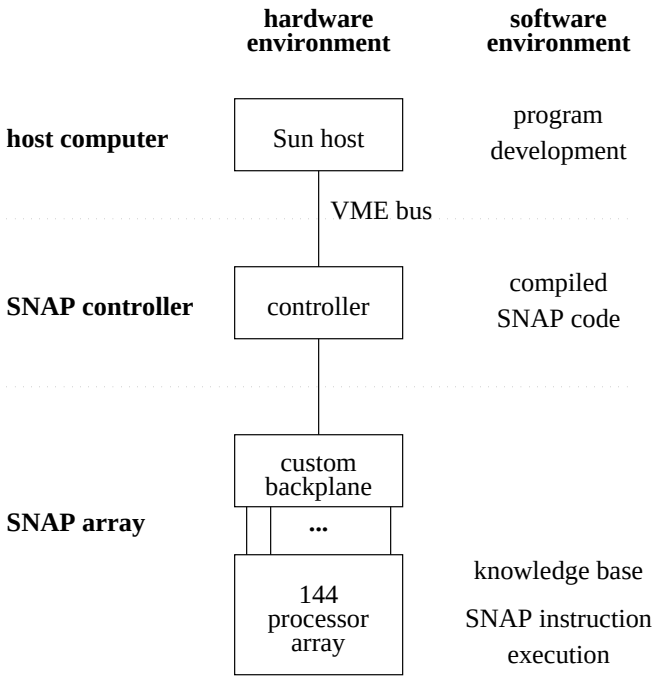


Figure 2.7: System overview of SNAP.

area, there are some projects which deal with performance improvements by design of specialized computer architectures.

On the other hand, there is a lot of work known which deals with improving the efficiency of CSP algorithms. Some of these projects use advanced tree-search techniques [KvB95]; others utilize improved preprocessing steps [MH86] and regroup the constraint network into hierarchical structures to avoid backtracking [DP89]. However, the objective of the presented thesis is to focus on work which uses parallel or distributed strategies for constraint satisfaction.

2.4.1 Multiprocessor Architectures for AI Applications

Semantic Network Array Processor (SNAP)

The *semantic network array processor (SNAP)* is a specially designed computer architecture for knowledge representation and reasoning which uses the marker-propagation paradigm. This special-purpose computer architecture is well documented [CM94] [DM93] [MLLC92] [Mol93] [MLL92]. Semantic networks are frequently used to represent and process structural knowledge. They consist of *nodes* which represent concepts within a domain, and *links* which show relationships between nodes. Each node is also assigned to a *color* to indicate the concept or class which it belongs to. Semantic networks represent the knowledge base of the reasoning system which allows the transfer of information between concepts in the domain. The dynamic inference mechanism which move information around is implemented by means of markers. Markers are data patterns associated with each node.

The primary objective of SNAP is to provide real-time *natural language understanding*

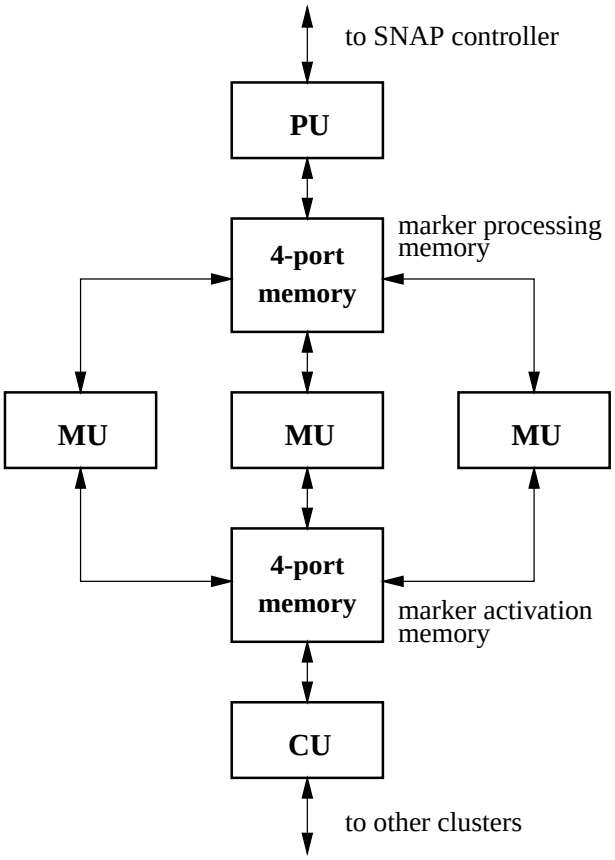


Figure 2.8: Block diagram of a SNAP cluster.

(NLU) while supporting a vocabulary of a few thousand words. Furthermore, this architecture provides the functionality for coding, executing and analyzing marker-propagation algorithms. To reduce the development time, only off-the-shelf components are used.

The architecture was developed by analyzing several marker-propagation programs which mainly consist of three phases: configuration, propagation and accumulation phase. Propagation operations consume most of the overall processing time. SNAP reduces propagation time by exploiting two types of parallelism. *Intra-propagation* parallelism is derived from processing a single propagation operation. Here, a number of nodes is activated simultaneously. These nodes perform the same instructions in a data-parallel manner. *Inter-propagation* parallelism exists between individual propagation operations, since there are no data-dependencies in the markers. To exploit both kinds of parallelism, aspects of SIMD and MIMD processing are employed.

Figure 2.7 presents a system overview of SNAP. This system consists of an *array* of 144 processing elements to store and process the semantic network, a *controller* which manages the array and a Sun *host* for the user-interface. The array is organized as 32 tightly-coupled clusters with four to five processing elements each. Communication occurs over a high-speed backplane. Each cluster consists of functional units for program control, marker processing and external communication. Each functional unit is implemented using a microprocessor chip with local memory. Figure 2.8 shows a block diagram of a SNAP

cluster with five functional units. The processing unit (PU) decodes the SNAP instructions and acts as the master of the cluster. The PU is further connected to the SNAP controller via a global bus. The role of the marker units (MU) is to process markers and search the knowledge base. The communication unit (CU) provides an interface between clusters via a 4-ary hypercube interconnection. Data transfer between functional units is established via two 4-port memories. The marker processing memory is the shared memory for marker propagation within the cluster. If a destination node of the marker processing is outside the cluster, then a message is placed in the marker activation memory.

Comparison

- Both projects deal with the design, implementation and experimental evaluation of a special-purpose computer architecture to improve the performance of specific AI applications. Although the applications are different, the design method is similar. The design is based on an extensive theoretical and empirical analyses of the algorithm. However, in marker-propagation the dominant inherent data-parallelism leads to a SIMD architecture. On the other hand, qualitative simulation offers only little to medium data-parallelism. SIMD processing is not appropriate to exploit the parallelism.

- Both prototype implementations use digital signal processors (DSP) as processing elements. The reasons for choosing DSPs are quite different. Speech and NLU applications require high-speed floating-point arithmetic to compute probabilities of competing hypotheses. Hence, the Texas Instruments TMS320C30 was selected because of its single-cycle 32-bit multiplication along with its on-chip support for arbitration of multiport memories. QSIM is based on symbolic computation — floating-point arithmetic is not required. In the presented thesis, the DSP TMS320C40 was chosen mainly due to its high number of independent communication channels and its high I/O performance.

Other Research Projects in 'Multiprocessor Architectures for AI Applications'

In research there are many other projects where multiprocessor systems are developed in order to speed up AI applications. These systems are specially designed for tasks in symbolic computation like searching, pattern matching and natural language processing. Some representative projects are:

- In [OMT94], a heterogeneous architecture is presented which exploits the parallelism of a hybrid problem solving system on different levels of granularity. The problem solving system consists of a rule-based expert system and an artificial neural network. The architecture is organized in three processing layers. The top layer is a MIMD computer consisting of transputers. The rule-based expert system is executed on this MIMD computer. The middle layer is made of a network of FPGAs. The bottom layer is constructed by digital signal processors which efficiently support vector processing algorithms required by the artificial neural network.

- The IMX2 processor [Hig94] is designed for speech-to-speech applications. This multiprocessor system is based on parallel associative processors and is operated in

SIMD mode. The IXM2 is actually a coprocessor under control of a host SparcStation. This system consists of 64 associative processors (transputers) which are equipped with associative memory.

- Robinson [Rob92] presents a specialized architecture to speed up pattern matching applications. This system is also based on associative memories (*pattern addressable memory PAM)* and is operated in SIMD mode.

2.4.2 Parallel Constraint Satisfaction Algorithms

Luo, Hendry and Buchanan [LHB94] have classified the most common parallel CSP algorithms as *distributed-agent-based (DAB)*, *parallel-agent-based (PAB)* and *function-agent-based (FAB)*. Different strategies may involve different control structures, problem spaces and communication methods. Important features of these strategies can be summarized as follows:

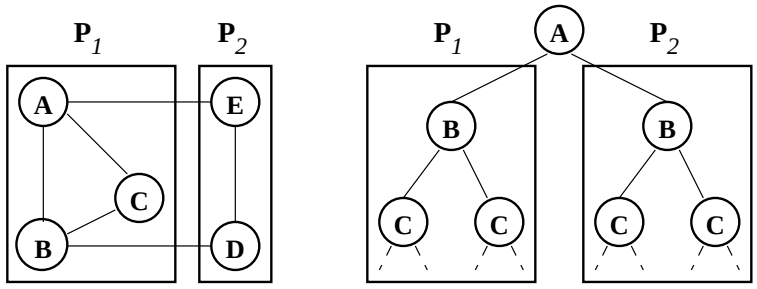
DAB. In the *distributed-agent-based* strategy, the problem is distributed based on the variables. Each processor is in charge of one or more variables and their domains. A variable is controlled by only one processor. The complete search space is shared among the processors. During the search, the processors have to communicate because of the constraints between distributed variables. For DAB algorithms, control mechanisms used to resolve conflicts can be centralized or decentralized. The control mechanism may introduce a lot of communication overhead.

PAB. In the *parallel-agent-based* strategy, the problem is distributed based on the domains of the variables. Each processor solves a part of the complete search space. Each partial search space involves all variables and is independent of other search spaces. Therefore, each processor solves a unique CSP, and no communication between processors is necessary. PAB can directly use any sequential CSP algorithm, needs little communication and can use established global heuristic search strategies.

FAB. The *function-agent-based* strategy exploits the control-parallelism of constraint satisfaction. Individual tasks, which are repeatedly performed during search, can be assigned to specific processors. If search is taking place on one processor using a sequential algorithm, then spare processors can be used to perform the actual checking in parallel. For example, the domain filtering step of a forward checking algorithm can be performed in parallel. This method is only suitable for shared-memory machines, where the data of a parent process can be seen and manipulated by its children.

Figure 2.9 presents the two most important parallelization strategies DAB and PAB. The inherent communication of the DAB strategy (a) can be seen from the edges (constraints) connecting variables located at different processors. The independent subspaces of the PAB strategy are shown in (b). Due to the reduced domain sizes of some variables, the individual subspaces are smaller than the overall problem space.

From the great number of parallel CSP algorithms, closely related work is described in some detail and compared to the work presented in this thesis. There are also some papers dealing with the parallelization of the preprocessing steps of constraint satisfactions. They



(a) Distributed-agent-based (b) Parallel-agent-based

Figure 2.9: Parallelization strategies for CSPs. In the DAB strategy (a), the CSP is partitioned into two subproblems based on the variables (A $\cdots$ E). In the PAB strategy (b), independent subspaces of the overall search space of the CSP are generated based on the domains of the variables.

range from a massively parallel implementation based on VLSI [CS92] to more theoretical papers [Kas90] [ZM93].

- Burg [Bur90a] uses a PAB strategy to solve the CSP. The search space is divided among a set of processors each executing a sequential forward checking (FC) algorithm. The search space is partitioned into as many subspaces as processors are available. Due to static load balancing, no communication between processors is required. An extension is also given [Bur90b] which utilizes dynamic load balancing. This technique introduces communication among the processors.

  Although the same parallelization strategy (PAB) is applied, Burg's work differs from the parallel CSP algorithm presented in this thesis. As the used multiprocessor system in [Bur90a] has a number of processors that is a power of two, the partitioning strategy is quite simple. The domains of one or more variables are split into two equal parts. The subspaces can be statically mapped onto the processing elements. In the architecture of the presented thesis, there can be an arbitrary number of processors. Furthermore, the complete search space is not divided into a fixed number of subspaces — the actual number of subspaces is not known before the partitioning algorithm is completed. On-line scheduling is applied to map subspaces onto processors.

- In [LY95a] [LY95b], a CSP is also parallelized by a PAB strategy, and a forward checking algorithm is executed on each processor. The partitioning strategy is different to that in the previous work. The search space is divided into d partitions, where d is the domain size of the first variable — only one variable is considered. The subspaces are statically mapped onto the processing elements. Furthermore, a load balancing technique uses a simple probabilistic analysis to estimate the amount of search required in each subspace. The parallel algorithm is evaluated by simulation on a single processor.

  The greatest restriction of this approach is the limited number of subspaces. The number of subspaces and, hence, processors is limited by the domain size of the variables. For qualitative simulation, this means that at most four subspaces are

generated. Static load balancing becomes also irrelevant for such a small number of subspaces.

- Rao and Kumar [RK93] evaluate the efficiency of parallel backtracking algorithms. They present analytical models and experimental results on the average case behavior of two parallel backtracking algorithms — simple backtracking and ordered backtracking. The difference between these two algorithms is that ordered backtracking uses a heuristic for ordering the successors of an expanded node. It may also use a heuristic to prune nodes of the search space. The search space is partitioned into disjoint parts by splitting the domains of the variables. Dynamic load balancing is applied to reduce processor idle times.

  Rao and Kumar state that the average speedup obtained with simple backtracking is linear when the distribution of solutions is uniform and superlinear when the distribution of solutions is non-uniform. For heuristic backtracking, the average speedup is at least linear. These theoretical results are validated by experimental results. However, these results do not directly apply to constraint satisfaction in qualitative simulation. In the evaluation of Rao and Kumar, only *one* solution has to be found by the backtracking algorithm. In qualitative simulation, *all* solutions of the CSP are required. Hence, all subspaces must be completely processed. If the search space is not pruned during partitioning, at the most linear speedup can be expected.

Chapter 3

Design of a Scalable Multiprocessor Architecture

3.1 Design Method

3.1.1 Parallelization and Design of Specialized Computer Architectures

Parallelization is a widely-used strategy to accelerate application software. To exploit the parallelism, the algorithms are mostly mapped onto general-purpose multiprocessor systems. There is often a mismatch between the algorithmic requirements and the properties of the multiprocessor system. This mismatch can happen for a lot of reasons: the topology of the multiprocessor system does not correspond with the structure of the parallel algorithm, or the functionality of the processing elements does not match with the functionality of the sequential tasks of the algorithm.

The presented thesis deals with both the parallelization of the QSIM kernel and the design of a specialized computer architecture for this algorithm. These two steps are not independent of each other — results from parallelization influence the design of the computer architecture and vice versa. The next sections present important steps for parallelization and architecture design with regard to the QSIM kernel — (i) the level of granularity for the parallelism detection, (ii) the mode of operation for the multiprocessor system and (iii) the scheduling strategies for the mapping of tasks to processors.

Parallelism Detection

The way to detect parallelism is to partition an algorithm into tasks and to study data dependencies between these tasks. Parallelism detection through the study of dependencies is valid for all levels of granularity. In the following sections, sequential units of an algorithm are denoted as *tasks* independent of the granularity. Dependencies occur due to input variables I and output variables O of tasks. For the two tasks T_1 and T_2 to be parallel, it is sufficient that *(Bernstein's conditions)*:

$$\begin{array}{lcll} I_2 \cap O_1 & = & \emptyset & \text{data-flow dependence} \\ I_1 \cap O_2 & = & \emptyset & \text{data antidependence} \\ O_1 \cap O_2 & = & \emptyset & \text{output dependence} \end{array} \tag{3.1}$$

Data dependencies between tasks are often represented in a *data dependence graph*. In this graph, nodes represent tasks and arcs represent dependencies. Data antidependencies and output dependencies can be removed by program transformations [Mol93]. Therefore, tasks which are not connected by arcs representing data-flow dependencies can be executed in parallel.

In the presented thesis, the data dependency analysis is performed in a top-down manner. The functions of the QSIM kernel are the tasks for the top-level analysis. Consequently, the analysis is gradually refined. An important question for a top-down analysis is when to stop. For the constraint-filter, the bottom-level of the analysis is given by the functionality of the tuple-filter and the constraint check function, respectively. In the overall architecture, these functions are executed on specialized coprocessors. The situation is quite different for the kernel function form-all-states. Here, tasks are not generated based on the function hierarchy. The overall function is artificially partitioned into tasks by a search space partitioning at run-time.

The level of granularity is often determined by the number of instructions of the tasks [Hwa93]. The number of instructions for these functions depends strongly on input data and cannot be exactly determined by a complexity analysis. However, a rough estimation reveals a medium level of complexity.

MIMD Processor with Distributed Memory

For efficient implementation of the parallelism detected by the data dependency analysis, the computer architecture must match with the requirements of the parallel algorithm as well as possible. Analysis of the QSIM kernel reveals a predominant *control-parallelism* compared to little *data-parallelism*. Different tasks execute different instructions at the same time. Thus, each processing element must be equipped with its own control unit. The multiprocessor system is operated in MIMD mode. MIMD multiprocessor systems can communicate via shared memory or message passing. While shared memory systems offer short latencies due to the tight coupling, they have only reduced scalability. The task granularity and the scalability requirement lead to a MIMD multiprocessor system with distributed memory.

The sequential tasks can be accelerated by performance improvements of the individual processing elements. The low-level parallelism within the tasks can be exploited by advanced processor and memory architectures, like super-scalar processors and multistage memories. Processors with specially designed instruction sets also reduce the execution time. Compiler technology influences the performance on the processing element, too. However, all these aspects which accelerate the execution time on a single processing element are not considered in the presented thesis.

Scheduling

Scheduling is the assignment of tasks to processors [SSNB95]. This assignment which is called *schedule* satisfies the conditions of the tasks (e.g., precedence relations, timing constraints) and of the target architecture (e.g., number of processors, communication times). The objective is to find an optimal schedule, with regard to a specific optimality criteria. A common optimality criteria is to minimize the maximal completion time of the tasks. In general, scheduling is a $NP-complete$ problem. Heuristics are often applied

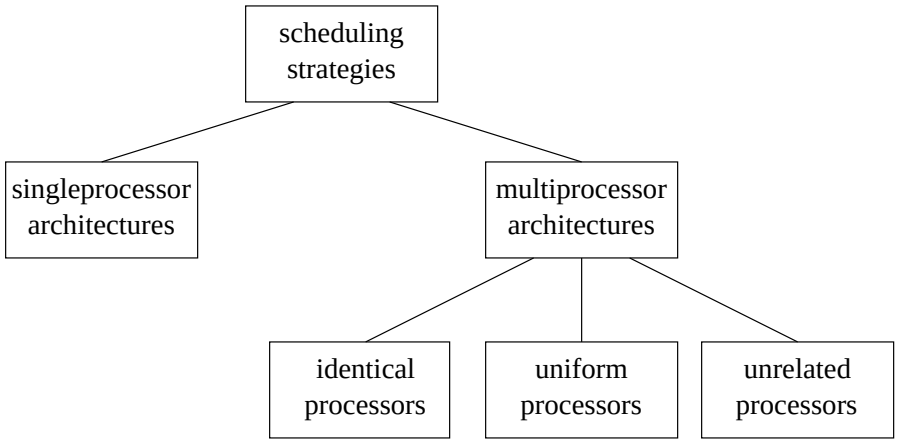


Figure 3.1: Classification of scheduling strategies based on target architectures.

to make scheduling tractable. However, the achieved schedules are often suboptimal. The scheduling problem has so many dimensions that there is no accepted taxonomy [SSNB95]. The most important issues of scheduling required for the presented thesis are presented below.

The scheduling strategy can be classified as *static* or *dynamic*. In static scheduling, the scheduling algorithm has complete knowledge of the tasks and their constraints. Tasks are assigned to processors before run-time. Hence, the static scheduling algorithm produces a single schedule that is fixed for all time. A dynamic scheduling algorithm has complete knowledge of currently active tasks, but new tasks activations not known to the scheduler when it schedules the current set may arrive. Therefore, the schedule changes over time. In static scheduling, tasks can be assigned to processors *off-line* or *on-line*. Off-line schedules are determined before run-time, whereas on-line schedules are determined by the scheduler at run-time. Scheduling also includes information regarding the time when a task will start to execute on a processor. From this point of view, scheduling strategies can be divided into *preemptive* and *nonpreemptive*. In a preemptive environment, tasks may be halted before completion by another task that requires service. In general, preemptive strategies can generate more efficient schedules than those that are not preemptive. However, an increased overhead occurs in the preemptive case due to task switching, which includes saving processor states.

Figure 3.1 presents a classification of scheduling strategies based on different target architectures. A basic distinction between scheduling strategies is whether the target architecture is a singleprocessor or a multiprocessor system. Multiprocessor systems are classified in (i) *identical* processors, where individual tasks have the same execution time on all processors, (ii) *uniform processors*, which are identical processors running at different clock speeds and (iii) *unrelated processors*, where individual tasks have completely different execution times on different processors. The ratio of execution times on distinct processors differs for each task.

A task must be executed on the appropriate processor. A *process* manages the execution of a task and the communication of input and output data. Tasks of the same type can be executed by the same process. Tasks of the QSim kernel are not fixed before

run-time. Both, the number of tasks and the properties of the tasks may change during the execution period. However, all tasks which are generated during kernel execution are known, and the properties of these tasks do not change. Therefore, *static* scheduling strategies can be applied to map tasks onto processors at *run-time*. To keep the overhead small, only nonpreemptive scheduling strategies are considered in this thesis.

3.1.2 Scalability

Scalability studies determine the degree of matching between a computer architecture and an application algorithm. Scalability must be studied for a given algorithm and architecture pair. A certain architecture can be very efficient for one algorithm but bad for another.

An intuitive but restrictive definition of scalability is based on the *system efficiency* $E(s,n) = \frac{S(s,n)}{n}$ [Hwa93], where s denotes the problem size; the number of processors is given by n. In general, the best possible efficiency is 1, implying that the best speedup is linear, $S(s,n) = n$. Therefore, a computer architecture is scalable for a given algorithm, if the efficiency $E(s,n) = 1$ for any number of processors and any problem size.

Nussbaum and Agrawal [NA91] have given a scalability definition based on a theoretical PRAM model. In this definition, the speedup of the real machine is compared with the speedup on an ideal realization on a PRAM, thus ignoring all communication overhead. The *isoefficiency function* [KR87] is another concept for defining the scalability. With a fixed problem size (or workload), the efficiency decreases as the number of processors increases. The reason is that the overhead increases with n. With a fixed number of processors, the overhead grows slower than the workload. Thus, the efficiency increases with the increasing problem size for a fixed-size machine. Therefore, a constant efficiency can be maintained, if the workload is allowed to grow properly with increasing machine size. The smaller the workload growth rates, the more scalable is the architecture and algorithm pair.

Hwang [Hwa93] addresses three different objectives for designing scalable computer architectures.

Machine-size scalability. A machine-size scalable computer architecture is designed to have a scaling range number of resource components. The objective is to achieve linearly increased performance with incremental expansion.

Generation scalability. Generation scalability deals with the problem that hardware and software technology should be scalable. If software or hardware components are replaced by new generation components, the computer architecture should also perform well.

Problem-size scalability. A problem-size scalable computer architecture should be able to perform well as the problem size increases. The problem size can be scaled to be sufficiently large in order to operate efficiently on a computer with a given granularity.

The presented thesis addresses mainly the machine-size scalability and to some extent the problem-size scalability. Generation scalability is not considered in this thesis.

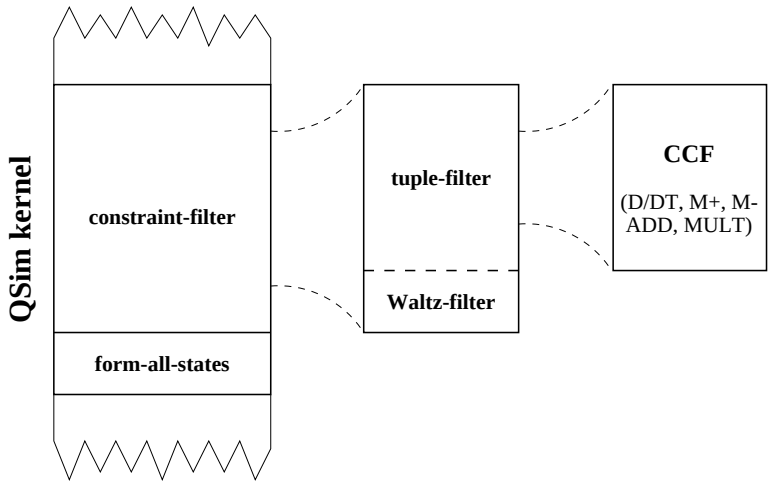


Figure 3.2: Function hierarchy of the QSIM kernel.

kernel function	*input variables*	*output variables*
tuple-filter	`pvals`	`tuples`
Waltz-filter	`pvals, tuples`	`pvals, tuples`
form-all-states	`tuples`	`CSP-solutions`

Table 3.1: Input and output variables of the QSIM kernel functions.

3.2 Analysis of the QSIM Kernel

The QSIM kernel consists of several hierarchically structured functions, as described in Section 2.1.3. Figure 3.2 presents the function hierarchy of the QSIM kernel. This graphical representation corresponds to the pseudo code of the kernel functions in Figure 2.3 and Figure 2.4, respectively. The kernel consists of two basic functions: the constraint-filter, which is the loop over all tuple-filter calls and the subsequent Waltz-filter, and the backtracking function form-all-states. The tuple-filter is further constructed by the constraint-check-functions (CCFs). There exists an individual CCF for each constraint type — like ADD, MULT and D/DT.

A first data dependency analysis is based on these kernel functions and starts with the identification of input and output variables. Table 3.1 shows the input and output variables of the kernel functions. Input variables for the tuple-filter are the sets of possible values `pvals` for the attached variables of the constraint. The tuple-filter produces the tuple set `tuples` for the constraint. The Waltz-filter requires all tuple sets of already filtered constraints and their corresponding set of possible values. Due to filtering, both tuples and possible values can be deleted — both are input and output variables, too. Finally, form-all-states requires the tuple sets of all constraints and generates all solutions of the CSP `CSP-solution`. Hence, input and output variables are the tuples and the set of CSP-solutions, respectively.

The data dependence graph in Figure 3.3 presents the relations between the kernel functions using input and output variables from Table 3.1. One iteration of the constraint-filter — the successive calls to the tuple-filter **t-f**$_i$ and to the Waltz-filter **W-f**$_i$ — is

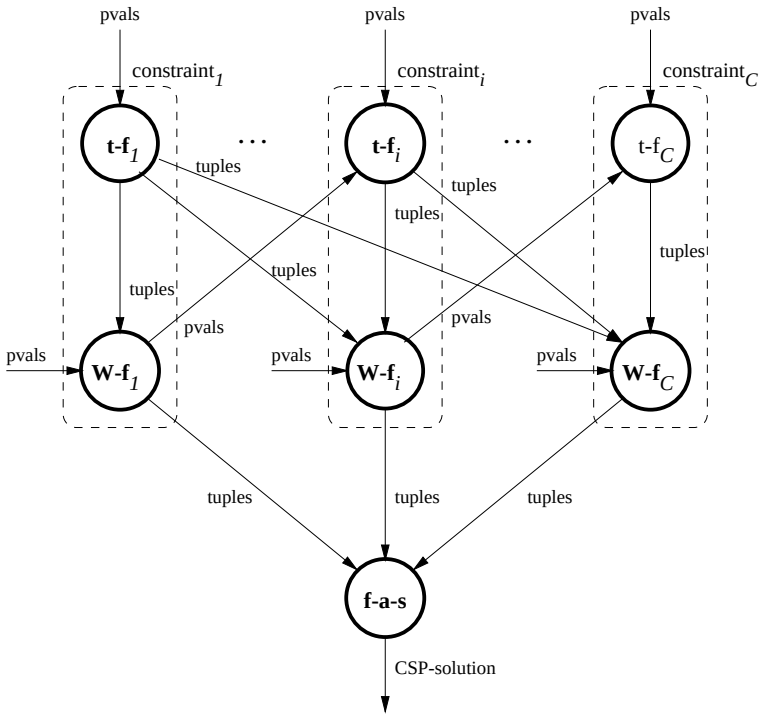


Figure 3.3: Data dependence graph of the QSIM kernel. The circles represent the tasks tuple-filter **t-f**$_i$, Waltz-filter **W-f**$_i$ and form-all-states **f-a-s**. The dashed boxes indicate one iteration of the constraint-filter, i.e., one call to the tuple-filter for constraint i and the subsequent call to the Waltz-filter.

represented by a dashed box. There is a high dependency between these constraint-filter functions. Furthermore, form-all-states requires the results of all Waltz-filter functions, which implies that constraint-filter and form-all-states must be executed successively.

To improve the performance of the kernel, the two successive functions, constraint-filter and form-all-states, must be improved. This leads to specialized multiprocessor architectures for both functions. Section 3.3 presents a multiprocessor for the constraint-filter. In Section 3.4, the parallelization of form-all-states is discussed and a specialized multiprocessor is presented. Finally, Section 3.5 combines these two approaches and presents an overall multiprocessor architecture for the QSIM kernel.

3.3 Constraint-Filter Multiprocessor

3.3.1 Analysis of the Constraint-Filter

The data dependency analysis of the QSIM kernel reveals a high dependency within the constraint-filter (Figure 3.3). The dependency between the Waltz-filter and the tuple-filter is caused by the set of possible values (pvals) and prevents the parallel execution of the tuple-filter tasks. The incremental Waltz-filter deletes pvals of variables as soon as possible to reduce the number of tuples which must be checked for consistency by the tuple-filter. The set of pvals of some variables can become smaller by the Waltz-filter call. Therefore, constraints which are connected to these variables but have not yet been tuple-filtered have a smaller number of tuples which must be checked than they would have without the previous Waltz-filter call.

Incremental Waltz-filtering is not necessary for the functionality of the QSIM kernel. It is basically a technique to improve the performance of the sequential algorithm. If the Waltz-filter is called *once* after all tuple-filter calls, the constraint network is still arc consistent. This technique is called *sequential* Waltz-filtering. Figure 3.5 compares the pseudo codes for the constraint-filter with incremental Waltz-filter (a) and with sequential Waltz-filter (b). This modification changes the data dependence graph of the constraint-filter. Figure 3.4 presents the data dependence graph of the constraint-filter with sequential Waltz-filtering. The data dependency between the Waltz-filter and the tuple-filter via the possible values `pvals` has disappeared. Thus, the tuple-filter tasks are independent from each other and can be executed in parallel.

Sequential Waltz-filtering influences the running time behavior of the constraint-filter. There is a trade-off between the running times of the tuple-filter and the Waltz-filter. First, there is more work to do for the tuple-filter due to the increased number of tuples that must be checked. On the other hand, the Waltz-filter is called just once. The worst-case complexity is reduced from $O(kC^2)$ for incremental Waltz-filtering to $O(kC)$ for sequential Waltz-filtering, where C denotes the number of constraints.

The influence of sequential Waltz-filtering on the total number of tuples has been empirically evaluated in [Rin93] [Hin94]. There is only a small increase of the total number of tuples compared to incremental Waltz-filtering, and the running time of the tuple-filter is only slightly increased. Therefore, it is reasonable to parallelize the constraint-filter based on a sequential Waltz-filter.

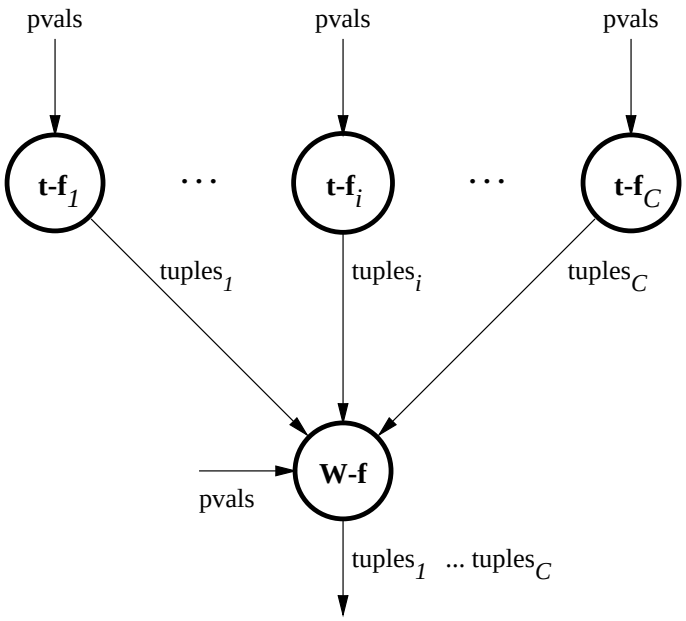


Figure 3.4: Data dependence graph of the constraint-filter with sequential Waltz-filter.

```
for all constraints con do
  tuple-filter(con)
  waltz-filter(con)
endfor
```

(a)

```
for all constraints con do
   tuple-filter(con)
endfor
waltz-filter()
```

(b)

Figure 3.5: Pseudo code for the constraint-filter with incremental (a) and sequential (b) Waltz-filter.

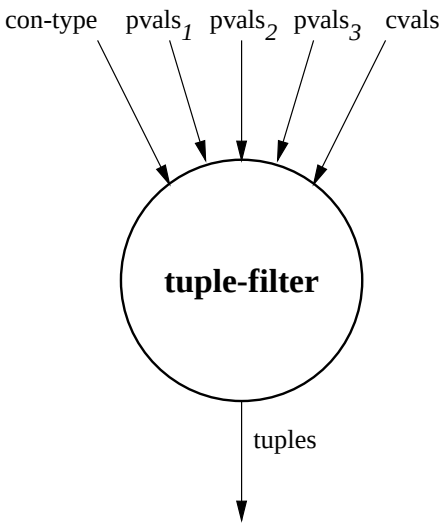


Figure 3.6: Data flow diagram of the tuple-filter.

3.3.2 Tuple-Filter Process

The tuple-filter process executes the basic sequential units of the parallel implementation of the constraint-filter — the tuple-filter tasks. In the pseudo code, the tuple-filter has only one input parameter — the constraint identifier (con). However, the tuple-filter requires access to more data. Figure 3.6 shows the data flow diagram of the tuple-filter function for a ternary constraint. Input data can be separated into two groups. The first group contains $pvals_1$, $pvals_2$, $pvals_3$ and *con-type*. The sets of possible values for the three variables of the constraint are represented by $pvals_1$, $pvals_2$ and $pvals_3$. The constraint type (e.g., ADD, MULT) is represented by *con-type*. The constraint type determines the constraint check function which is used for the consistency check of each tuple. The second group is formed by the set of corresponding value (cvals) tuples. Corresponding value tuples are generated outside the QSIM kernel. This generation is a very rare process compared to the number of tuple-filter calls.

To reduce the data transfer, the set of cvals tuples is stored within the tuple-filter process. Furthermore, the cvals tuples of all constraints together with their constraint types are also stored within the tuple-filter task. The storage of information for all constraints enables a transparent implementation of the tuple-filter task. Hence, the task can process all constraints in the same way. Figure 3.7 presents the data flow diagram of the tuple-filter task with internal data structures. The tuple-filter requires only the constraint identifier *con-id* and the sets of possible values $pvals_1$, $pvals_2$ and $pvals_3$ as input data. To get access to the constraint type and cvals tuples, the tuple-filter scans through its internal data structures for the entry with the same *con-id*. The internal data structures are initialized before simulation and updated during simulation by access commands. These commands operate from outside the QSIM kernel and are defined as follows:

- `TF-ADD-CONS con-id con-type`
  This command adds a new entry for the constraint *con-id* with the type *con-type* to the internal table.

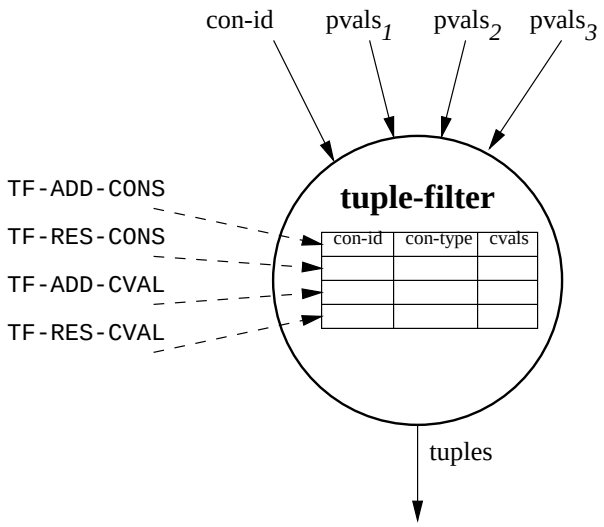


Figure 3.7: Data flow diagram of the tuple-filter task with internal data structures.

- `TF-RES-CONS`
  This command clears the internal table. All stored information is deleted.
- `TF-ADD-CVAL con-id cval`
  This command adds one corresponding value tuple *cval* to the entry of the constraint *con-id*.
- `TF-RES-CVAL con-id`
  This command resets the set of cvals tuples of the constraint *con-id*.

Update of the Internal Data Structures

The internal data structures are normally updated (i) before simulation with the constraint identifier, constraint types and initial cvals of all constraints and (ii) during simulation with newly generated cvals. However, there are some cases where additional manipulation of the internal data structures is required.

Region transitions. The set of active constraints can change at region transitions. All constraints stored in the tuple-filter must be deleted and data of all new constraints must be stored with `TF-ADD-CONS` and `TF-ADD-CVAL` commands.

States in different behaviors. The set of cvals tuples is dependent on the state in the behavior tree. Different cvals tuples for the same constraint can be generated and added to the set of cvals tuples in states on different paths in the behavior tree. This is shown in Figure 3.8. In the presented behavior tree, different cvals tuples for the same constraint are generated in state 2 and 3. Thus, all states in the corresponding subtrees of the behavior tree have different sets of cvals tuples. QSIM processes the state in the agenda in a breadth-first manner. Whenever a state of a different subtree is processed, the cvals set in the data structures must be updated. There can be a lot of update operations that just switch between the set of cvals

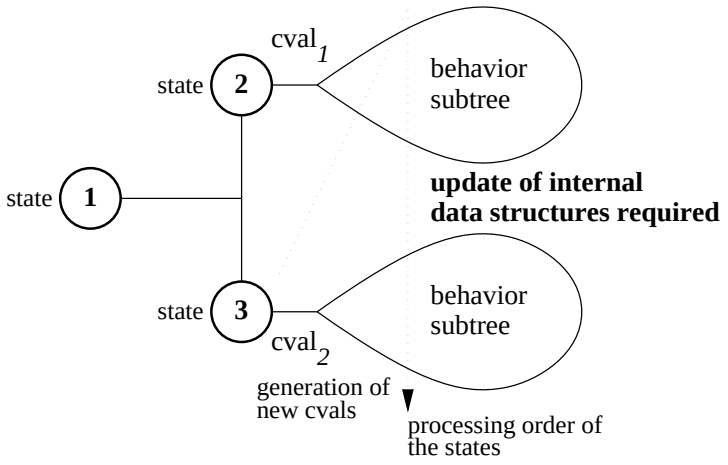


Figure 3.8: Generation of different cvals tuples in states on different paths in the behavior tree.

tuples in different behaviors. These update operations require the deletion of all cvals for some constraints and the storage of the new cvals for these constraints by the `TF-RES-CVAL` and the `TF-ADD-CVAL` commands. To overcome these updates, the set of cvals tuples for different behaviors can be stored in the tuple-filter. Then, the constraint identifier must be extended by an additional identifier for the behavior.

Decompaction of the landmark codes. Landmark values are represented as binary coded numbers with a fixed number of bits. Due to a nonuniform distribution of the assignments of landmark values to binary numbers, insertion of new landmarks can be blocked and the set of landmark values must be reassigned *(decompaction)* [Pla96]. If the landmark values of a variable are reassigned, the cvals set of all attached constraints must also be updated.

All these updates of the internal data structures are rare processes and are operated from outside the QSIM kernel.

Representation of Input and Output Data

The tuple-filter requires *con-id* and all possible values $pvals_1$, $pvals_2$ and $pvals_3$ as input data and returns the set of consistent tuples as output data. The total number of possible values is given by $P = i + j + k$, where i, j and k denote the cardinality of $pvals_1$, $pvals_2$ and $pvals_3$. To reduce the data transfer, the actual set of consistent tuples is coded as a binary number, where each bit represents whether a given tuple is consistent or not. This binary number is called `tf-res`. Thus, $T = ijk$ bits are required to represent the entire tuple space

$$pvals_3 \times pvals_2 \times pvals_1$$

of a constraint. The tuple-filter must only be slightly modified to encode the tuple set in `tf-res`. Figure 3.9 presents this modified tuple-filter. According to the result of the constraint check function (CCF), the corresponding bit in `tf-res` is set or reset (lines 7–11). `tf-res` is processed from the least significant bit (LSB) to the most significant

```
 1    procedure tuple-filter(con)
 2    begin
 3      bitno ← 0
 4      for all pvals p3 of pvals3 do
 5        for all pvals p2 of pvals2 do
 6          for all pvals p1 of pvals1 do
 7            if (ccf_con(p1, p2, p3) = TRUE) then
 8              tf-res[bitno] ← 1
 9            else
10              tf-res[bitno] ← 0
11            endif
12            bitno ← bitno + 1
13          endfor
14        endfor
15      endfor
16   end
```

Figure 3.9: Pseudo code for the modified tuple-filter.

bit (MSB). Thus, the counter for the bit position (`bitno`) is incremented by 1 in each iteration (line 12).

The set of consistent tuples is reconstructed from the encoded binary number as follows. For a given bit position *pos*, the actual tuple is defined as

$$\begin{aligned} ind_1 &= pos \ mod \ i \\ ind_2 &= (pos \ div \ i) \ mod \ j \\ ind_3 &= pos \ mod \ ij, \end{aligned} \tag{3.2}$$

where ind_1, ind_2 and ind_3 denote the indices of the possible values in the corresponding sets $pvals_1$, $pvals_2$ and $pvals_3$.

The transfer of input and output data is established via two commands. These commands operate within the QSIM kernel and are defined as follows:

- `TF-EXE-CONS con-id` $p_{1,1} \ldots p_{1,i}\ p_{2,1} \ldots p_{2,j}\ p_{3,1} \ldots p_{3,k}$
  This command sends the constraint identifier and $i + j + k$ possible values to the tuple-filter.

- `TF-RESULT con-id tf-res`
  This command returns the constraint identifier and the encoded tuple set from the tuple-filter. ijk bits are required to represent the tuple set.

3.3.3 Architectural Considerations

Figure 3.10 presents the logical structure of the constraint-filter. This logical structure is directly derived from the data dependency analysis of Section 3.3.1. It consists of a set of processes and communication links. The processes are partitioned into two groups — a *master* process and a set of *slave* (tuple-filter) processes. The master process is responsible for the transmission of all tasks to the slaves, the reception of the task's results

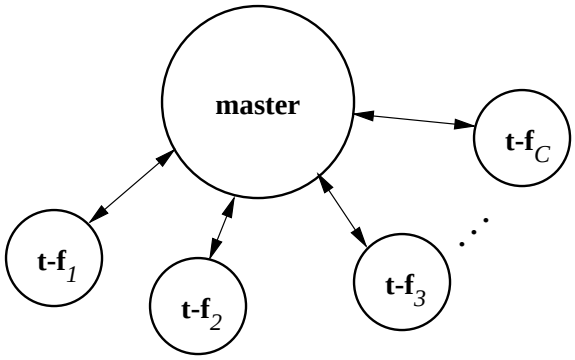


Figure 3.10: Logical structure of the constraint-filter.

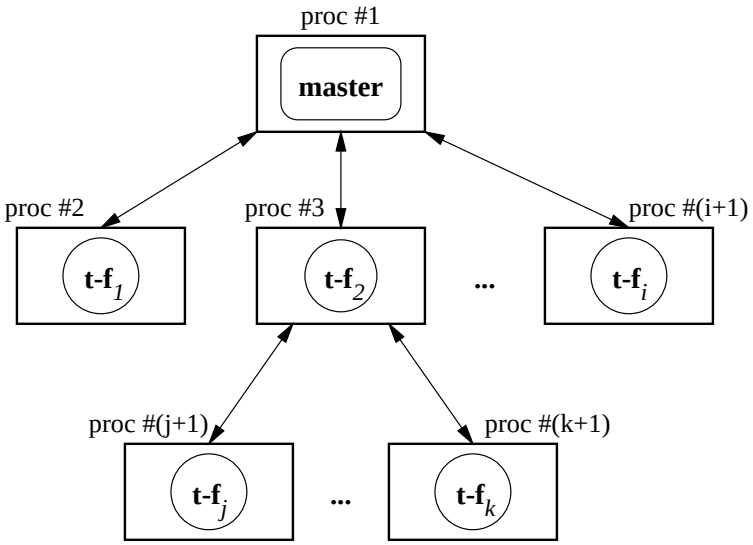


Figure 3.11: Topology of the constraint-filter multiprocessor.

from all slaves and the execution of the Waltz-filter. The slave processes must execute the individual tuple-filter tasks. Since all tuple-filter tasks 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. Therefore, the logical structure consists of C slaves.

The logical structure forms a *star* with the master as the central process. However, in a star structure the master becomes a bottleneck as the number of slaves increases. This limits the scalability of the system. To achieve a scalable architecture a *wide tree* topology is used as a compromise between logical structure and scalability. A wide tree topology for the constraint-filter is shown in Figure 3.11. In a wide tree, each processing element has a constant node degree and, hence, a fixed number of direct communication links. A wide tree with p children has a node degree of $p+1$. Thus, $p+1$ communication links are required. The number of nodes for a l level wide tree with p children at each node is given by $\sum_{i=0}^{l} p^i$. The root node of the tree corresponds to the master process; all other nodes correspond to the slaves of the logical structure. Not all nodes are directly connected to

```
1    procedure LS-I()
2    begin
3      for all processors i do
4        sum_i ← 0
5        tasklist_i ← ∅
6      endfor
7      for all tasks j do
8        find processor i with minimal sum_i + p_ij
9        sum_i ← sum_i + p_ij
10       add {j} to tasklist_i
11     endfor
12   end
```

Figure 3.12: Pseudo code for the list scheduling algorithm LS-I.

the root. Therefore, some tasks must be routed via intermediate nodes to their target nodes.

3.3.4 Scheduling

All tuple-filter tasks must be assigned to processing elements at run-time. The set of tasks is only known at the beginning of the QSim kernel, although the number of tasks is given by the number of constraints C. This is because the operating region can change during simulation and a new set of constraints can become active. Furthermore, input data of the individual tasks change from one kernel execution to the next. The scheduling algorithm must utilize the properties caused by the different input data. The applied scheduling algorithm must fulfill several important requirements. First, the execution time of the scheduling algorithm must be as short as possible because scheduling is a sequential part of the parallel implementation. Second, as presented in Section 3.1.1, only nonpreemptive scheduling is considered in this thesis. Thus, the generated schedule must result in a balanced workload to achieve a high performance. Finally, the tuple-filter processes can be implemented on different processing elements. Some of these can be equipped with specialized coprocessors (CCF- or tuple-filter coprocessors) to accelerate the running time of the tuple-filter. The scheduling algorithm should be able to handle these cases — i.e., to schedule tasks on identical or unrelated processors.

Scheduling Algorithms

List scheduling algorithms [Gra69] offer a reasonable compromise between the execution time of the scheduling algorithm and deviation of the generated schedules from the optimal schedules. Three different list scheduling algorithms, LS-I, LS-II and LS-III, are considered for the parallel implementation of the constraint-filter. Furthermore, all algorithms require the task execution times of all tasks on all processors. These times are represented by the matrix $\mathbf{P}$, where $\mathtt{p}_{ij}$ denotes the execution time of task j on processor i. In the following section, the three scheduling algorithms are described briefly:

```
1   procedure LS-II()
2   begin
3     for all tasks j do
4       find b_j = min_{1≤i≤n}{p_ij}
5       mark task j as unassigned
6     endfor
7     for all tasks j do
8       for all processors i do
9         e_ij ← b_j / p_ij
10      endfor
11    endfor
12    for all processors i do
13      create a tasklist sorted in nondecreasing order of e_ij
14      sum_i ← 0
15      assign processor i as active
16    endfor
17    while (not all tasks are assigned) do
18      find a processor i such that sum_i is minimal among active processors
19      find the next unassigned task j on processor i's tasklist
20      if ((∄ j) ∨ ( e_ij < 1/√n)) then
21        mark processor j as inactive
22      else
23        assign task j to processor i
24        mark task j as being assigned
25        sum_i ← sum_i + p_ij
26      endif
27    endwhile
28  end
```

Figure 3.13: Pseudo code for the list scheduling algorithm LS-II.

algorithm	*complexity*	*performance ratio*
LS-I	$O(Cn)$	n
LS-II	$O(Cn \log C)$	$\frac{5}{2}\sqrt{n}$
LS-III	$O(Cn \log C)$	$(1+\sqrt{2})\sqrt{n}$

Table 3.2: Comparison of complexity and performance ratio of LS-I, LS-II and LS-III.

LS-I. This algorithm uses the simplest scheduling heuristic and is based on [IK77]. The pseudo code for LS-I is shown in Figure 3.12. In this algorithm, the assignment of tasks is based on the actual processing time of each processor, $\mathtt{sum}_i$. In the initial step, the processing times and tasklists of all processors are reset (lines 3–6). Each task is assigned to the processor with the minimal sum of processing time and task execution time (line 8). Then the corresponding processing time and tasklist is updated (lines 9–10). The procedure is repeated until all tasks are assigned.

LS-II. Figure 3.13 presents the pseudo code for the list scheduling algorithm LS-II. This algorithm was introduced by [DJ81]. The task assignment is based on the *efficiency* $\mathtt{e}_{ij}$ of each task on each processor. The efficiency of a task j on processor i is defined as the ratio of the minimum execution time of task j over all processors to the execution time of task j on processor i ($\mathtt{e}_{ij} \leq 1$). In the preprocessing steps, the minimum task execution times are determined (lines 3–6), the efficiencies are calculated (lines 7–11) and all tasklists, sorted in nondecreasing order of their efficiency, are created. Furthermore, all tasks are marked as unassigned, and all processors are assigned as active. Tasks are assigned to the processor with the shortest processing time (line 18). The next unassigned task in the list is only assigned to this processor, if the task has a high efficiency. If the efficiency drops below a limit, then the processor is deactivated (line 21). In contrast to LS-I, LS-II can be applied as an on-line scheduler as long as the efficiencies are known even if the task execution times $\mathtt{p}_{ij}$ are not known in advance. The only place where processing times are needed is in determing which processor is the next to be assigned with a task (line 18). If this decision is made at run-time — i.e., after a processor completes all tasks already assigned to it — then the task execution times are not needed while the assignment is being made.

LS-III. This algorithm is a modification of LS-II. The ordering criteria for the tasklists is extended in LS-III. If the efficiency of two tasks is equal, $\mathtt{e}_{ik} = \mathtt{e}_{il}$, then task k is ordered before task l if $\mathtt{p}_{ik} \geq \mathtt{p}_{ij}$. Hence, tasks with the same efficiency are ordered by their execution times.

The worst-case behavior of a scheduling algorithm is often determined by the *performance ratio*. The performance ratio is defined as the ratio of the worst-case finish time of the scheduled tasks and the finish time of the optimal schedule. Table 3.2 summarizes the algorithmic complexities and the performance ratios of the three scheduling algorithms [IK77] [DJ81]. LS-I has a smaller algorithmic complexity than LS-II and LS-III, respectively. However, the performance ratio is linear with the number of processors n, whereas the performance ratios of LS-II and LS-III are proportional with $\sqrt{n}$.

Estimation of Task Execution Times

The execution times of the individual tuple-filter tasks are not known before their execution. Therefore, the execution times must be estimated by an appropriate estimation function. List scheduling requires accurate estimations of the execution times to generate good schedules. Especially LS-I cannot adjust inexact estimations during task execution. LS-II and LS-III are able to adjust inaccuracies due to on-line scheduling. In LS-II and LS-III, calculation of the efficiencies is based on the estimated execution times. Thus, inaccurate execution times result in inaccurate efficiencies. On the other hand, the estimation must be performed at run-time. The estimation function should be computed efficiently.

The execution time of an individual tuple-filter task i is estimated as

$$\tilde{t}_i = T_i \cdot \bar{t}_{CCFk} + t_{offs}, \tag{3.3}$$

where T_i is the number of tuples which must be checked, and $\bar{t}_{CCFk}$ denotes the average execution time of one tuple check by the appropriate call of the CCF for constraint type k. The overall offset of the tuple-filter task is represented by t_{offs}. If the scheduling algorithm assigns tasks to unrelated processors, the estimation function in Equation 3.3 is slightly modified to

$$\tilde{t}_{i,j} = T_i \cdot S_{j,k} \cdot \bar{t}_{CCFk} + t_{offs}. \tag{3.4}$$

The execution time of task i on processor j is determined by T_i, the speedup factor $S_{j,k}$ of the CCF k on processor j, $\bar{t}_{CCFk}$ and t_{offs}. The execution time of one CCF call depends on the constraint type, input data and the implementation of the coprocessor. Thus, $S_{j,k}$, $\bar{t}_{CCFk}$ and t_{offs} must be determined empirically.

Depending on whether the tuple-filter processes are implemented on identical or unrelated processors, the estimation functions of Equation 3.3 and Equation 3.4 are used to determine the execution times $\mathtt{p}_{ij}$.

3.3.5 Speedup Considerations

This section discusses an analytical evaluation of the execution time of a parallel tuple-filter execution. The three different scheduling algorithms are compared by a worst-case estimation.

The speedup $S(n)$ of the parallel tuple-filter is defined as

$$S_{tf}(n) = \frac{t_{tf-seq}}{t_{tf-par}(n)}, \tag{3.5}$$

where n denotes the number of processing elements. The sequential running time of all tuple-filters is given by the sum over the execution times of all tuple-filter calls t_{tf-i}

$$t_{tf-seq} = \sum_{i=1}^{C} t_{tf-i}. \tag{3.6}$$

On the other hand, the parallel execution time is determined by the execution time of the scheduling algorithm t_{tf-sch}, the execution time of the tuple-filter tasks t_{tf-exe} on n

processors and the overhead time t_{tf-oh} which is caused by communication and additional instructions.

$$t_{tf-par}(n) = t_{tf-sch}(n) + t_{tf-exe}(n) + t_{tf-oh}(n) \tag{3.7}$$

All these execution times depend on the number of processors. Although the scheduler is a sequential algorithm, its execution time t_{tf-sch} is dependent on the number of processors. Furthermore, the execution time of all tuple-filter tasks t_{tf-exe} depends on the generated schedule and the implementation of the tuple-filter processes (with or without coprocessor support). Finally, t_{tf-oh} includes all additional overhead of the parallel implementation which is dependent on the communication performance and the multiprocessor topology.

The evaluation of the scheduling algorithm cannot only be based on the performance ratio r, because the execution time of the algorithm t_{tf-sch} is also included in the overall parallel execution time. Therefore, the sum of t_{tf-exe} and t_{tf-sch} should be as small as possible for a given scheduling algorithm. A worst-case estimation for t_{tf-sch} and t_{tf-exe} is given by the algorithmic complexity and the performance ratio, respectively. The execution time of the scheduling algorithm can be estimated as

$$t_{tf-sch}(n) \leq k_i f_i(n), \tag{3.8}$$

where k_i represents an arbitrary constant and f_i represents the complexity for the scheduling algorithm i. If all tuple-filters are implemented without coprocessor support, the limits for t_{tf-exe} are given as

$$\frac{1}{n} t_{tf-seq} \;=\; \frac{1}{n} \sum_{i=1}^{n} t_{tf-i} \;\leq\; t_{opt}(n) \;\leq\; t_{tf-exe}(n) \;\leq\; r_l(n) t_{opt}(n). \tag{3.9}$$

The lower limit of the execution time is given by the arithmetic mean of the sequential execution time over n processors. A tighter lower bound is the execution time of the optimal schedule for n processors, t_{opt}. The upper bound is determined by the performance ratio $r_l(n)$ of the scheduling algorithm l.

By combining Equations 3.8 and 3.9 together with the complexities and performance ratios of Table 3.2, the worst-case estimation of $t_{tf-sch} + t_{tf-exe}$ is defined as:

$$t_{tf-sch}(n) + t_{tf-exe}(n) = \begin{cases} t_{LS1}(n) = k_1 C n + n t_{opt} & \text{for LS-I} \\ t_{LS2}(n) = k_2 C n \log C + \frac{5}{2}\sqrt{n} t_{opt} & \text{for LS-II} \\ t_{LS3}(n) = k_3 C n \log C + (1+\sqrt{2})\sqrt{n} t_{opt} & \text{for LS-III} \end{cases} \tag{3.10}$$

The worst-case behavior of the scheduling algorithms can be compared by the estimations of Equation 3.10. For the comparison of LS-I and LS-II, the difference of the corresponding estimations in Equation 3.10 must be determined. This results in the following expression:

$$t_{LS2}(n) - t_{LS1}(n) = (k_2 \log C - k_1) C n + (\frac{5}{2}\sqrt{n} - n) t_{opt} \tag{3.11}$$

If this expression is positive, the worst-case behavior of LS-I is better than the worst-case behavior of LS-II, and vice versa. The expression consists of two terms. If both terms are positive or both terms are negative, the overall expression is positive or negative. This results in two conditions for the comparison of LS-I and LS-II.

$$\begin{array}{lcll} (n < \frac{25}{4}) & \wedge & (C > 2^{k_1/k_2}) & \Rightarrow \quad t_{LS1} < t_{LS2} \\ (n > \frac{25}{4}) & \wedge & (C < 2^{k_1/k_2}) & \Rightarrow \quad t_{LS1} > t_{LS2} \end{array} \tag{3.12}$$

Similar conditions can be derived by comparing LS-I and LS-III.

$$\begin{array}{lcll} (n < (3+2\sqrt{2})) & \wedge & (C > 2^{k_1/k_3}) & \Rightarrow \quad t_{LS1} < t_{LS3} \\ (n > (3+2\sqrt{2})) & \wedge & (C < 2^{k_1/k_3}) & \Rightarrow \quad t_{LS1} > t_{LS3} \end{array} \tag{3.13}$$

Equations 3.12 and 3.13 define conditions for comparing the worst-case behavior of the three scheduling algorithms LS-I, LS-II and LS-III. This comparison is based on the number of processors n and the number of tasks C. However, these conditions allow only a limited comparison due to the following reasons. First, the scheduling algorithms can only be compared if both terms of Expression 3.11 are either positive or negative. Second, the constants k_1, k_2 and k_3 must be known for the worst-case comparison — but these constants depend on the actual implementation. Finally, the performance ratios define only upper limits for the execution times of the parallel tuple-filters. Nothing can be said about the actual execution time of a given problem or the average execution time of a set of problems. Therefore, a detailed comparison is only possible by an experimental evaluation of the scheduling algorithms in a parallel implementation of the tuple-filter.

3.4 Parallelization of Form-All-States

3.4.1 Parallelization Strategy

The final function of the QSIM kernel is form-all-states. This backtracking function solves the CSP, defined by the QDE and the sets of possible values. 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 into smaller subproblems. The CSP in QSIM has some specific properties which makes it different from many other CSPs. These properties also influence the parallel implementation.

- The backtracking algorithm solves the CSP by finding the solutions of the dual constraint network. Therefore, the overall search space is given as $T = tuples_1 \times \cdots \times tuples_C$. The search is based on the tuple sets of the individual constraints.
- QSIM requires *all* solutions of the CSP.
- For successor state generation, the domain of the variables is limited by 4 — thus, the maximum number of tuples per constraint is given by 16 and 64, respectively. For initial state processing, the domain is only limited by the quantity space — many more tuples per constraint are possible.
- The QSIM kernel is often executed successively with the same QDE. The constraint network of the CSP remains the same, only possible values are changed.

characteristics	*distributed constraint satisfaction strategies*		
	DAB	*PAB*	*FAB*
preferred problems	naturally distributed	tightly coupled	tightly coupled
algorithm design	specially designed	any sequential	any sequential
memory type	shared/distributed	shared/distributed	shared
communication cost	medium/high	lower	n/a
load balancing	poor/fair	good	good
scalability	poor	fair/good	reasonable
termination detection	difficult	easy	easy
find a solution	poor	fair/good	good
find more solutions	poor	good/excellent	fair

Table 3.3: Characteristics of basic distributed CSP strategies [LHB94].

These properties of the QSIM CSPs ease the selection of a specific parallelization strategy. Luo, Hendry and Buchanan [LHB94] present a characterization of basic distributed constraint satisfaction strategies. Table 3.3 summaries the characteristics of these strategies. In general, the parallel-agent-based (PAB) strategy is very versatile and has good characteristics. Especially for important properties for the parallelization of the QSIM kernel — like *memory type, scalability* and *find more solutions* — a PAB strategy is more suitable than the other strategies. Due to these reasons, the QSIM CSP and, hence, the function form-all-states are parallelized using a PAB strategy. Several questions arise when using this parallelization strategy:

1. How should the overall search space be partitioned into smaller independent subspaces to achieve a balanced workload and a high performance?

2. Which sequential algorithm should be used to solve the subspaces?

3. How should the tasks (subspaces) be scheduled to the processors ?

In the following sections, these questions are considered in more detail.

3.4.2 Partitioning of the Search Space

In a parallel-agent-based strategy, the overall search space is partitioned into independent subspaces based on the domains of the variables of the CSP. Form-all-states finds the solution of the dual constraint network of the CSP. Therefore, the constraints represent the variables of the constraint network, and the tuple sets correspond to the domains. The tuple sets must be split for the partitioning. A subspace is defined as

$$P_i = tuples_{1,i} \times \cdots \times tuples_{C,i}, \tag{3.14}$$

where $tuples_{k,i}$ represents subset i of $tuples_k$. For a valid partitioning, the union of all p subspaces must be the overall search space of the CSP. Hence, the following equation must hold:

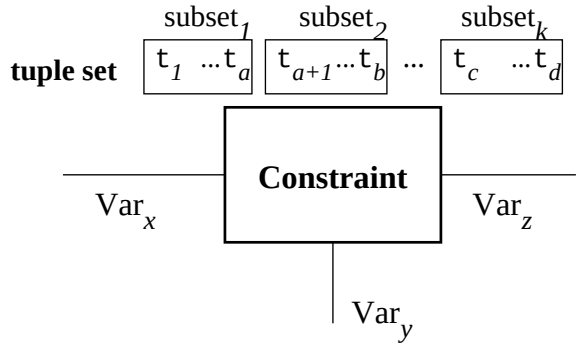


Figure 3.14: Constraint-based partitioning.

$$\bigcup_{i=1}^{p} P_i = tuples_1 \times \cdots \times tuples_C = T \tag{3.15}$$

Two methods, each dividing the tuple sets of the constraints, are described in the following sections. The first method is directly based on the tuple set of a constraint. The second method starts from the domain of a shared variable and partitions the tuple sets of adjacent constraints.

Constraint-Based Partitioning (CBP)

An obvious partitioning method is based on the tuple set of the constraints. The tuple set of an individual constraint is divided into k disjunct subsets.

$$tuples_i = tuples_{i,1} \cup \cdots \cup tuples_{i,k} \tag{3.16}$$

Figure 3.14 presents the constraint-based partitioning method graphically. By partitioning the tuple set into k subsets, k subproblems are generated. To generate more subproblems than the number of tuples of the constraint, the partitioning must be extended to other constraints. All combinations of subsets from different constraints must be searched for solutions. No subproblems can be discarded from further processing. For example, if three tuple sets are partitioned into k_1, k_2 and k_3 subsets, $k_1 k_2 k_3$ subproblems must be searched for solutions.

Variable-Based Partitioning (VBP)

A more intelligent partitioning method is based on the domains of the variables. The tuple sets of adjacent constraints are not independent of each other. The tuple sets depend on the domains of the shared variables. The domain of a variable is divided into k subdomains. This induces a partitioning of the tuple sets of all attached constraints. Each individual subset consists only of tuples with the same value of the shared variable as in the corresponding subdomain.

Figure 3.15 shows the variable-based partitioning method. Dividing the domain of variable V into k subdomains results in k subsets of the tuple sets of all attached constraints. The advantage of VBP is that many subproblems can be discarded from further

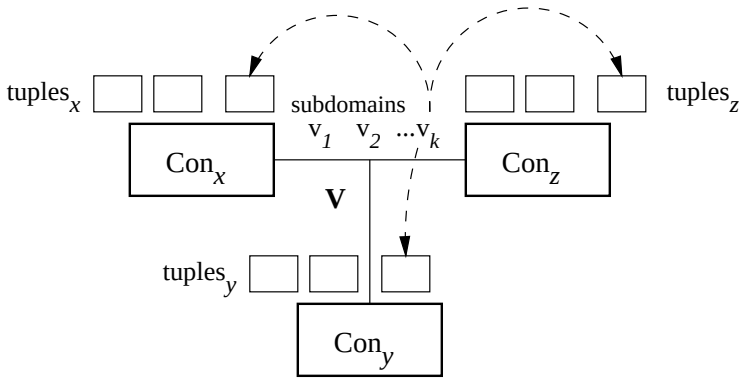


Figure 3.15: Variable-based partitioning. The partitioning of the domain of a variable induces a partitioning of the tuple sets of all attached constraints.

processing. Only subproblems which are generated from subsets of tuples with the same subdomain of the variable must be considered. Subproblems which are derived from different subdomains violate the arc consistency condition. There cannot be any solution in these subproblems. This results in a great reduction of subproblems that must be processed. If k subdomains are generated of a variable with a attached constraints, only k subproblems must be searched for solutions. In contrast, if each tuple set of a constraints is divided by the CBP method, a^k subproblems must be searched for solutions. To generate more subproblems than the ordinality of the domain of one variable, partitioning is extended to other variables.

VBP consists of two procedures. The procedure `vbp` divides recursively the complete search space into an appropriate number of subspaces. `generate-subproblem` checks the subspaces for local consistency and generates the corresponding tuple sets for valid subspaces. The pseudo code for the VBP method is presented in Figure 3.16 and Figure 3.17. At most `maxsub` subproblems can be generated by this algorithm. This is caused for two reasons. First, `maxsub` subspaces are checked by `generate-subproblem`, only if the product of the cardinality of all considered variables is equal to the number of required subproblems ($\prod |pvals_i| =$ `maxsub`). In all other cases, less than `maxsub` subspaces are checked. Second, the generation of subproblems discards subspaces, which violate local consistency conditions.

In the algorithm, the subdomains of the considered variables are stored in the global array `actpval`. The corresponding index to this array is determined by the function `index(con,var)`. In this array, there exists an entry for each variable for all constraints. A subdomain is generated for each possible value of a considered variable (lines 8–13 of `vbp`). The possible values are stored in `actpval` for all attached constraints (lines 9–11) and `vbp` is called recursively with the next variable and the increased number of subspaces as parameters (line 12). These two parameters are used for the termination of the recursive calls (line 3). The number of *generated* subproblems is stored in the variable `nsub` (line 5). In `generate-subproblem`, the tuple set of each constraint is filtered using `actpval` (line 7–11). If no tuple of a constraint survives, filtering is aborted and FALSE is returned (lines 17–20). Each consistent tuple is stored to generate the subproblem during filtering (line 14).

```
 1    procedure vbp(var, n)
 2    begin
 3      if ((var = last-var()) ∨ (n > maxsub)) then
 4        if (generate-subproblem() = TRUE) then
 5          nsub ← nsub + 1
 6        endif
 7      else
 8        for all pvals pv of pvals(var) do
 9          for all attached constraints con of var do
10            actpval[index(con,var)] ← pv
11          endfor
12          vbp(next-var(var), n*|pvals(var)|)
13        endfor
14      endif
15    end
```

Figure 3.16: Pseudo code for the variable-based partitioning method (VBP).

```
 1    procedure generate-subproblem()
 2    begin
 3      for all constraints con do
 4        tuple-found ← FALSE
 5        for all tuples tup of con do
 6          tuple-ok ← TRUE
 7          for all variables var of tup do
 8            if (tuple-qval(tup,var) ≠ actpval[index(con,var)]) then
 9              tuple-ok ← FALSE
10            endif
11          endfor
12          if (tuple-ok = TRUE) then
13            tuple-found ← TRUE
14            save tuple for subproblem
15          endif
16        endfor
17        if (tuple-found = FALSE) then
18          discard subproblem
19          return FALSE
20        endif
21      endfor
22      return TRUE
23    end
```

Figure 3.17: Pseudo code for the procedure generate-subproblem.

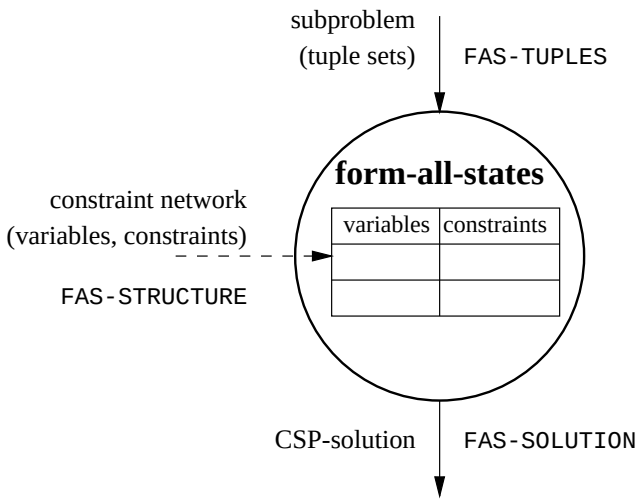


Figure 3.18: Data flow diagram of form-all-states.

The order in which the variables are processed influences the number of generated subproblems and their quality. To access the first, next and last variable, the functions `first-var()`, `next-var(var)` and `last-var()` are used. Therefore, the first call to `vbp` is done with `first-var()` and `1` as actual parameters. In this project, the order of the variables is based on four different heuristics.

VBP-INST. This heuristic processes the variables in the same order as the sequential algorithm instantiates the variables.

VBP-CON. The number of attached constraints of a given variable defines the order of this heuristic. The domains of variables which are shared by many constraints are partitioned first. Thus, many tuple sets are divided into subsets.

VBP-DOM. The cardinalities of the domains determine the order of the variables. The tuple sets of the attached constraints are divided into many subsets.

VBP-TUP. The order of variables is based on the total number of tuples of all attached constraints. This heuristic divides the largest tuple sets first.

3.4.3 Form-All-States Process

Using the PAB strategy for parallelization, independent subproblems must be solved by a sequential algorithm. The subproblems are complete CSPs but with a smaller search space than the unpartitioned CSP. The subproblems are transfered to a form-all-states process which computes the solutions by a sequential CSP algorithm. All solutions are returned from the form-all-states process. The form-all-states process requires all data which determines the CSP of the subproblem. As already defined in Section 2.1.3, a CSP is given by the set of constraints, the set of variables and — due to the dual representation in QSim — the sets of tuples. To reduce communication, input data for the form-all-states process is divided into two groups. The first group contains the set of variables and the set of constraints. These sets represent the constraint network. For all subproblems of one

command	*communication effort*
FAS-STRUCTURE	$O(C)$
FAS-TUPLES	$O(d^3C)$
FAS-SOLUTION	$O(V)$

Table 3.4: Communication effort for the form-all-state process.

QSIM kernel call, the constraint network remains the same, and, furthermore, the QSIM kernel is often executed successively with the same constraint network. The sets of tuples represent the second group. These sets define the search space, and they are different for each subproblem.

Figure 3.18 presents the data flow diagram of the form-all-states process. First, the constraint network is stored in the process by the FAS-STRUCTURE command. This can be seen as an initialization step for the form-all-states process. Then, subproblems for the same constraint network are transfered to the process via the FAS-TUPLES command only. The results of the subproblems are returned by the FAS-SOLUTION command.

Table 3.4 presents the communication effort for all commands of the form-all-states process. Since the arity of the constraints is limited by 3, at most three words are required to represent the variables of one constraint. Hence, $O(C)$ words are required to represent the constraint network. There are at most d^3 tuples for one constraint. The subspace of an individual subproblem can be transfered with $O(d^3C)$ data words by the FAS-TUPLES command. Finally, a solution of a subproblem is given by one value for each variable. In general, there can be d^V solutions in a subproblem. However, in QSIM the number of solutions is normally very small. Thus, the number of data words required for FAS-SOLUTION is estimated by $O(V)$.

In QSIM form-all-states uses a simple backtracking algorithm to solve the CSP. Many improvements of this simple search algorithm are known [Pro93]. These improved algorithms prune many more subtrees of the search space than simple backtracking. Especially for complex CSPs, the execution time is strongly reduced. These algorithms can also be used to improve the execution time of form-all-states for some complex CSPs [Rie95]. However, the main focus of this work is the parallelization of the QSIM kernel. Therefore, the subproblems are solved by the same simple backtracking algorithm as in the original algorithm form-all-states to allow a fair evaluation of the parallel implementation.

3.4.4 Architectural Considerations and Scheduling

The logical structure of the parallel algorithm for form-all-states is similar to the logical structure of the constraint-filter. Due to the PAB strategy, a master/slave structure is also derived. The master process is responsible for the generation of the subproblems, their transmission to the slave processes and the union of the partial results to the overall result. The slave processes must solve the subproblems and were described in the previous section. The maximum degree of parallelism is determined by the partitioning algorithm. Thus, the logical structure consists of nsub slave processes. The same architectural considerations for the constraint-filter are valid for form-all-states. To design a scalable architecture, the multiprocessor system is connected in a wide tree topology.

On-line scheduling is required for the parallel implementation of form-all-states, be-

cause the tasks (subproblems) are only known at run-time. To reduce the sequential part of the algorithm, scheduling time must be as short as possible. Besides these similarities to the constraint-filter, there are major differences in the requirements of the scheduling strategy. Due to the irregular behavior of the backtracking algorithm, the execution times of the tasks cannot be determined in advance. The actual number of instructions are not known before execution. Just the worst-case is known, but this number differs normally from the actual number by orders of magnitude. The slave processes are implemented on identical processing elements. Therefore, the execution time of the same task is independent of the processing element. Less information is available about the tasks for the scheduling algorithm. To balance the tasks as best as possible statically, *task attraction* scheduling is applied. Whenever a processor is idle, the next task is scheduled to this processor.

3.4.5 Evaluation and Speedup Limits

This section discusses the speedup limits of the parallel implementation of form-all-states. This evaluation is based on worst-case and best-case execution times of the parallel implementation. The speedup of form-all-states is given as the ratio of the sequential execution time $t_{fas-seq}$ and the parallel execution time $t_{fas-par}$ using n processors.

$$S_{fas}(n) = \frac{t_{fas-seq}}{t_{fas-par}(n)} \tag{3.17}$$

The parallel execution time depends on the execution time of the partitioning algorithm $t_{fas-part}$, the execution time of all subproblems on n processors $t_{fas-exe}$ and the overhead t_{fas-oh}. Hence, it is the sum

$$t_{fas-par}(n) = t_{fas-part}(n) + t_{fas-exe}(n) + t_{fas-oh}(n). \tag{3.18}$$

To derive speedup limits the parallel execution time $t_{fas-par}$ is analyzed in more detail. Partitioning of the complete search space is essential for an efficient parallel algorithm. Since the parallel implementation uses no dynamic load balancing technique, generated subproblems are executed by one processing element without interruption. Bad workload distribution cannot be balanced by additional partitioning of subproblems already running on a processing element. Therefore, the partitioning method should generate subproblems with equal workload. Due to redundancies in the independent subproblems, the total workload of all subproblems can be greater than the workload of the unpartitioned search space. Thus, an efficient partitioning method keeps the total workload small and generates equally sized subproblems.

The workload corresponds to the number of instructions required to solve the subproblem [Hwa93]. The execution time of the subproblem is used as a measure for the workload. The partitioning methods are evaluated using the sequential execution times of the subproblems. With these execution times, speedup limits can be derived. The

following execution times are required for this evaluation.

$t_{fas-seq}$ execution time of the unpartitioned problem

$t_{fas-tot}$ $\sum_{i=1}^{p} t_i$ total execution time of all p subproblems

$t_{fas-max}$ $max_{i=1}^{p}\{t_i\}$ execution time of the subproblem with maximum workload (3.19)

For speedup estimation, no communication times are considered, and simple task attraction is assumed to schedule subproblems to idle processors. The execution time of the partitioning method and the overhead are also neglected ($t_{fas-part} = 0$ and $t_{fas-oh} = 0$).

Worst-case and best-case speedups are determined by the maximum and minimum parallel execution time $t_{fas-exe}$. The parallel execution time of a given partitioning is determined by the order in which the subproblems are executed. The worst-case order is given if the subproblem with the maximum workload is scheduled last and all other subproblems are equally distributed among the processors. The worst-case execution time $t_{wc}(n)$ using n processors is given as

$$t_{wc}(n) = \frac{t_{fas-tot} - t_{fas-max}}{n} + t_{fas-max}. \tag{3.20}$$

The best-case execution time $t_{bc}(n)$ is determined as follows. If the number of processors is smaller than $\lceil \frac{t_{fas-tot}}{t_{fas-max}} \rceil$, all subproblems are equally distributed among the processors. Otherwise the parallel execution time is limited by $t_{fas-max}$. More formally, the best-case execution time is defined as

$$t_{bc}(n) = \begin{cases} \frac{t_{fas-tot}}{n} & \text{if } n < \lceil \frac{t_{fas-tot}}{t_{fas-max}} \rceil \\ t_{fas-max} & \text{otherwise.} \end{cases} \tag{3.21}$$

Worst-case and best-case parallel execution times determine the limits for the speedup. Minimum and maximum speedup ($S_{min}(n)$ and $S_{max}(n)$) are defined as

$$S_{min}(n) = \frac{t_{fas-seq}}{t_{wc}(n)} \tag{3.22}$$

$$S_{max}(n) = \frac{t_{fas-seq}}{t_{bc}(n)}. \tag{3.23}$$

A comparison of the partitioning methods based on maximum and minimum speedup is presented in Section 4.3.

3.5 Overall Architecture for the QSim Kernel

3.5.1 QSim Kernel Multiprocessor

Design considerations for both kernel functions constraint-filter and form-all-states reveal the same logical structure and, hence, the same topology of the multiprocessor system. Therefore, the same multiprocessor architecture is used for the parallel implementation of both kernel functions.

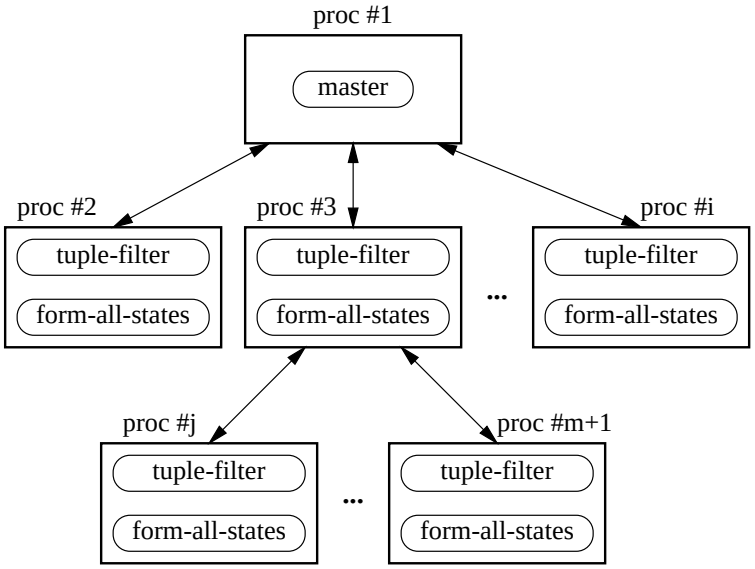


Figure 3.19: Topology of the overall QSIM kernel multiprocessor. The multiprocessor system consists of one root and m slave processing elements.

Figure 3.19 presents the topology of the overall QSIM kernel multiprocessor. The multiprocessor system is connected in a wide tree topology. A tuple-filter and a form-all-states process are located on each slave processing element. Thus, each slave processor has the same functionality and is able to execute all tasks generated by the master process. Constraint-filter and form-all-states are executed successively. Therefore, the master process in the root processor is basically a combination of the master processes of the constraint-filter and form-all-states. Independent of the kernel function, the master process is responsible for (i) the generation of the appropriate tasks (subproblems), (ii) the scheduling of all tasks and (iii) the generation of the total result from the partial results of the individual tasks.

The number of slave processors m is the basic parameter of this multiprocessor architecture. It limits the number of tasks which can be executed in parallel.

3.5.2 Coprocessor Support

In the overall architecture, specialized coprocessors are used to improve the performance of the QSIM kernel multiprocessor additionally. The tuple-filter processes are implemented by the support of CCF- or tuple-filter coprocessors [Pla96]. This section describes the required modifications for the overall multiprocessor system.

To manage the execution of a CCF or a tuple-filter function on the coprocessor, the tuple-filter process must be extended. The coprocessor interface is responsible for the communication to the coprocessor. This coprocessor interface hides all details concerning the coupling method as well as the coprocessor instruction set and provides transparent communication to the coprocessor. Figure 3.20 presents the embedding of a coprocessor to the tuple-filter process.

Nearly all QSIM coprocessor types require the corresponding value tuples for the ex-

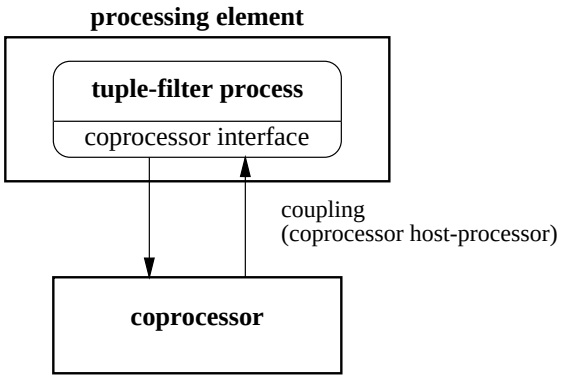


Figure 3.20: Embedding of a coprocessor into a tuple-filter process. The coprocessor interface manages the data transfer to and from the coprocessor using the instruction set of the coprocessor.

ecution of the CCF. Therefore, the corresponding value tuples must be stored in the coprocessor before the CCF or tuple-filter function is executed. Depending on the size of the internal memory of the coprocessor, a similar technique as in the tuple-filter process can be used to reduce the data transfer to the coprocessor. The sets of corresponding value tuples of several or all constraints can be stored within the coprocessor memory. The same conditions for the update of the internal data structures of the tuple-filter process are also valid for the update coprocessor memory (compare Section 3.3.2).

From the point of view of the master process, the execution time is the only difference between a tuple-filter process with and a tuple-filter process without coprocessor support. Therefore, the task execution times are determined with the estimation function from Equation 3.4, and the speedup factors of the processing elements $S_{j,k}$ must correspond to the actual multiprocessor system.

There are many different possibilities to construct the multiprocessor system supported by coprocessors. The actual implementation depends on the capabilities of the processing elements as well as on the number and type of coprocessors. The best performance is achieved if all processing elements are equipped with coprocessors for all types of constraints. However, it is not possible to equip the processing elements with so many coprocessors. The actual number is restricted by the coupling method and the capabilities of the processing element, i.e., the number of communication links. Therefore, to achieve a good performance the number and type of the attached coprocessors should match with the constraints of the simulation model.

3.5.3 Overall Speedup

Parallelization and the utilization of specialized coprocessors effect the performance of the QSIM kernel multiprocessor. In this section, the overall performance of the QSIM kernel multiprocessor is compared to the performance of the sequential implementation. This comparison is based on the individual speedups achieved for the parallelization of constraint-filter S_{cf} and form-all-states S_{fas}, respectively. Furthermore, the performance improvement due to coprocessor support, S_{ccf}, is also applied. These individual speedups

allow the calculation of the overall speedup for the parallel kernel implementation $\bar{S}_{par}$ and the total speedup $\bar{S}_{tot}$ for the additional coprocessor support.

First of all, the sequential execution time of the QSIM kernel is given by the sum of the execution times of the individual kernel functions

$$t_{seq} = t_{cf-seq} + t_{fas-seq}. \tag{3.24}$$

The influence of the individual speedups is determined by the execution time ratios of these two functions. Therefore, these ratios are defined as

$$\alpha = \frac{t_{cf-seq}}{t_{seq}} \quad \text{and} \quad \beta = \frac{t_{fas-seq}}{t_{seq}}. \tag{3.25}$$

The execution time of the parallel kernel implementation t_{par} can be expressed by these ratios and the individual speedups of the kernel functions

$$t_{par} = (\frac{\alpha}{S_{cf}} + \frac{\beta}{S_{fas}})t_{seq}. \tag{3.26}$$

With the execution time of the parallel implementation t_{par}, the speedup of the parallel kernel implementation $\bar{S}_{par}$ can be determined

$$\bar{S}_{par} = \frac{t_{seq}}{t_{par}} = \frac{1}{\frac{\alpha}{S_{cf}} + \frac{\beta}{S_{fas}}}. \tag{3.27}$$

$\bar{S}_{par}$ represents the speedup which is achieved by the QSIM kernel multiprocessor. If the QSIM kernel multiprocessor is supported by coprocessors, the execution time of the constraint-filter is additionally reduced by the factor S_{ccf}. Therefore, the total speedup $\bar{S}_{tot}$ of the QSIM kernel multiprocessor with coprocessor support is given as

$$\bar{S}_{tot} = \frac{1}{\frac{\alpha}{S_{cf}S_{ccf}} + \frac{\beta}{S_{fas}}}. \tag{3.28}$$

Given the speedup formulas in Equations 3.27 and 3.28, it is clear that a high overall speedup can only be achieved if both kernel functions constraint-filter and form-all-states are accelerated appropriately. Otherwise, the execution time of one kernel function will limit the total speedup. From this point of view it is also clear that a sequential kernel implementation with coprocessor support has only a limited total speedup. As presented in Section 2.2, the initial state processing and generation of successor states result in completely different execution time ratios α and β. The influence of these two cases on the total speedup is presented in the following:

Initial state processing. The empirical running time analysis reveals execution time ratios of approximately $\alpha = 0.1$ and $\beta = 0.9$. Therefore, a high speedup for form-all-states is important. The constraint-filter improvement has a minor influence on the total speedup. If the constraint-filter is executed sequentially a maximum total speedup $S_{tot} = 10$ can be achieved.

Generation of successor states. The situation is quite different for successor state generation. In this case, the empirical running time analysis reveals execution time ratios of approximately $\alpha = 0.75$ and $\beta = 0.25$. A high speedup for the constraint-filter is essential and, therefore, the influence of the coprocessors on the total speedup increases.

Discussion

The total speedup $\bar{S}_{tot}$ in Equation 3.28 is based on the individual speedups achieved by parallelization and coprocessor support. Many parameters influence these individual speedups and the exact values cannot be determined analytically. In general, only limits for the speedup can be given. Even if the actual speedup for a given multiprocessor system and a given kernel call is known exactly, the situation can change completely for the next kernel call which results in a completely different speedup.

Therefore, Equation 3.28 can be used to estimate limits for the achievable speedup of the QSIM kernel multiprocessor and the influence of improvements of individual speedups on the total speedup.

Chapter 4

Prototype Implementation and Experimental Evaluation

4.1 Prototype Platform

The multiprocessor system for the QSIM kernel is implemented on an appropriate platform. This section describes the hardware and software platforms — i.e., a *multi-DSP architecture* based on TMS320C40 processors and the distributed operating system *Virtuoso* — and demonstrates the suitability of these components.

4.1.1 Multi–DSP Architecture

The digital signal processor (DSP) TMS320C40 from Texas Instruments is used as a processing element for the prototype implementation of the multiprocessor architecture. This 32-bit floating-point processor was designed for application in the area of digital signal processing. The TMS320C40 is also used as a building block for multiprocessor systems with shared or distributed memory. Multi-DSP systems based on TMS320C40 are successfully applied for the parallel implementation of algorithms in the area of digital signal processing [Mat95] [VDC95] [HP95] [HW95]. Multi-TMS320C40 systems are also suitable for other high-performance applications [BGH+95]. These systems have a well-balanced ratio of communication and computation performance. This allows an efficient exploitation of parallelism at rather low levels of granularity. Many parallel algorithms are developed and implemented on these multi-DSP systems [SW94a] [SSW95] [Hra94] [Pam95].

On the other hand, the qualitative simulator QSIM has a rather distinct algorithm characteristic which distinguishes QSIM from typical DSP algorithms. Typical signal processing algorithms show a high inherent data parallelism. These algorithms are mainly based on extensive numerical computation, and their execution times are hardly influenced by input data. The QSIM algorithm can be characterized as follows:

- Symbolic computation
- Low to medium data parallelism
- High input sensitivity

Despite these different algorithm characteristics, the DSP TMS320C40 was chosen as the processing element due to the following reasons:

Multiprocessing capabilities. The TMS320C40 offers excellent capabilities for the development of a multiprocessor system. Multiprocessor systems can be constructed in arbitrary topologies by simply connecting TMS320C40 processors via their communication ports.

6 independent communication ports. The TMS320C40 is equipped with six independent bidirectional communication ports. Therefore, a wide tree topology with up to five children can be constructed.

High I/O performance. All communication ports can work in parallel, and each communication port has a data transfer rate of 20 MByte/s.

Parallel I/O and CPU operation. The six DMA channels of the DMA coprocessor can be used to perform data transfer between the communication ports and memory *(split mode)*. Each DMA channel supplies one of the six communication ports. In this mode, all communication ports and the CPU operate in parallel.

Coprocessor interface. In the prototype implementation, the tuple-filter and CCF coprocessors are loosely coupled to the TMS320C40 via two communication ports. Hence, the specialized coprocessors can be attached to any processor node of the multiprocessor.

The suitability of a multi-TMS320C40 system for prototype implementation of the specialized multiprocessor architecture for QSIM is demonstrated in [PR95a].

4.1.2 Distributed Operating System Virtuoso

The Software implementation is based on the distributed real-time operating system Virtuoso [Ver94]. This operating system is specially designed for multi-DSP architectures and allows a flexible, and to some extent, portable implementation. Furthermore, this multi-tasking operating system supports a scalable software design. This corresponds with the scalability requirement of the multiprocessor architecture. On the other hand, an operating system introduces some overhead. Therefore, software development based on an operating system is a compromise between efficiency and flexibility. Virtuoso has four different programming levels with different flexibility and efficiency. The programmer can choose the appropriate level to achieve the desired performance.

The parallel QSIM kernel is implemented on the highest programming level of Virtuoso, the *micro-kernel*. At this level, the application is written in the programming language ANSI–C. Micro-kernel objects can be accessed via C function calls. Some features of this level are important for the implementation of the prototype.

Task object. The high level programming model is based on the concept of micro-kernel objects. The main objects are the tasks which execute arbitrary functions and have dynamic priorities. The processes of the parallel QSIM kernel algorithm are implemented as Virtuoso tasks.

Task synchronization and communication. Tasks coordinate by using three types of objects: semaphores, mailboxes and FIFO queues. Task synchronization is achieved by semaphore objects. Mailboxes offer a synchronized and flexible data transfer between tasks. A variable number of data words can be sent from a task to a mailbox. The receiver task is determined by the message header attached to the transfered data words. FIFO queues establish a fast and asynchronous data transfer between two tasks.

Host communication. A PC serves as a host for the multi-DSP system of the prototype implementation. Virtuoso offers a versatile and convenient interface to the host system. Each processor node can communicate to the host via a standard C I/O-interface. Text, file and graphics I/O with the host is established via standard C functions.

Virtual single processor model. The Virtuoso programming system provides the same functionality by way of a virtual single processor model independent of the number of interconnected processors that are actually being used. Therefore, software can be developed and tested independently of the final multi-DSP architecture on one DSP or even on a PC. A Virtuoso library for PCs is also available.

Implementation of efficient parallel algorithms based on Virtuoso is demonstrated by several projects [SSW95] [Ste95].

4.2 Prototype Implementation

This section presents the implementation of the multiprocessor architecture for the QSim kernel and the test environment for the evaluation.

4.2.1 QSim Kernel Multiprocessor

A short summary of the most important issues of the multiprocessor implementation for both kernel functions, the constraint-filter and form-all-states, is given. A detailed description can be found in [Kau96].

Software was written in ANSI C using the operating system Virtuoso version 3.09. The source code was compiled by the Texas Instruments TMS320 compiler and linker tools version 4.50. The multi-DSP system was built using Transtech ISA–bus motherboards. Each board is equipped with up to four TMS320C40 TIM modules which have up to 8 MByte local memory.

Figure 4.1 presents an example of the overall architecture consisting of four TIM modules as processing elements. Each processing element is equipped with one DSP TMS320C40 and local memory. The processing elements are connected via the communication ports in a tree structure with three children. Communication to the host processor is established via a dedicated communication port of the root processor. Each slave processor is equipped with a CCF coprocessor via two communication ports.

The process mapping is shown in Figure 4.2. There are three processes (Virtuoso tasks) mapped onto the root processor: `QSim-kernel`, `fas-results` and `stdio`. The main process of the root processor is `QSim-kernel`. It unites the functionality of both

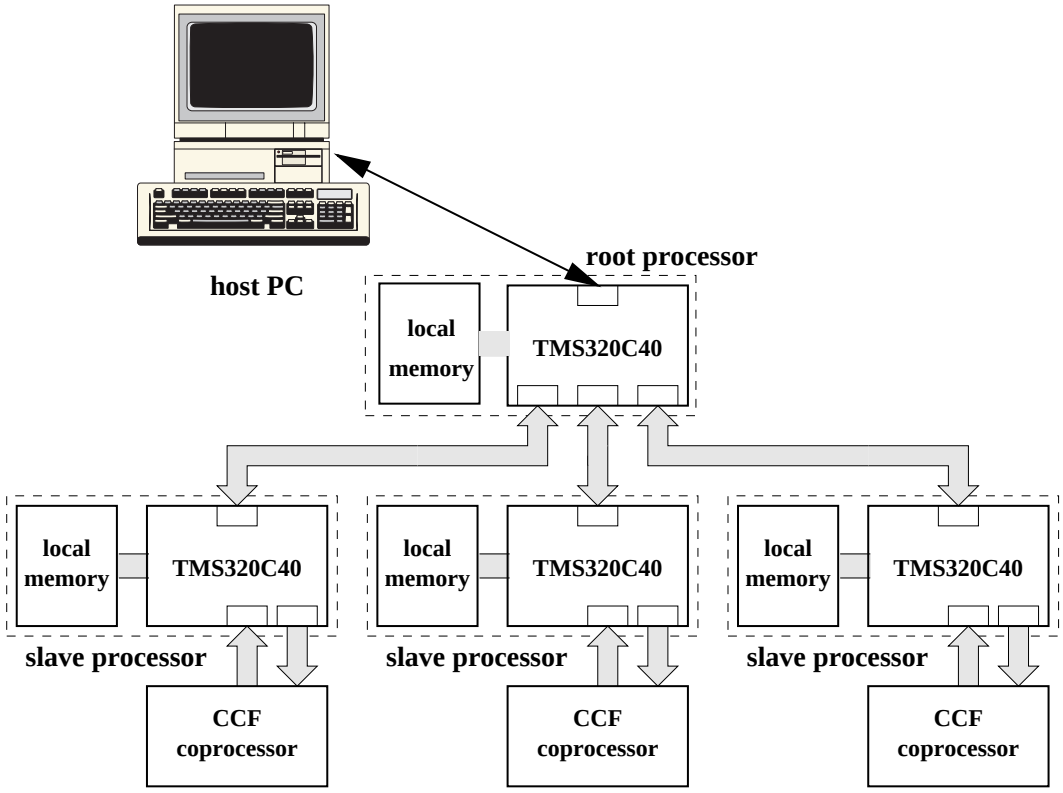


Figure 4.1: Example of the overall architecture for the QSim multiprocessor system.

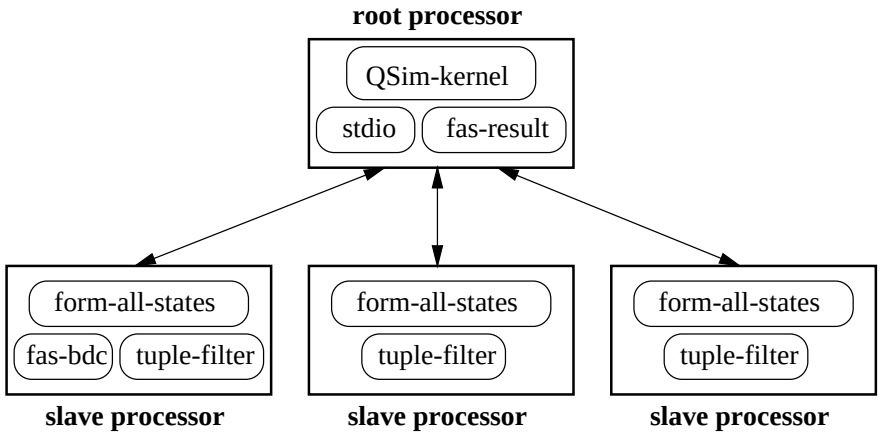


Figure 4.2: Process mapping. The tasks of the parallel implementation are executed on the corresponding processes — i.e., tuple-filter and form-all-states tasks.

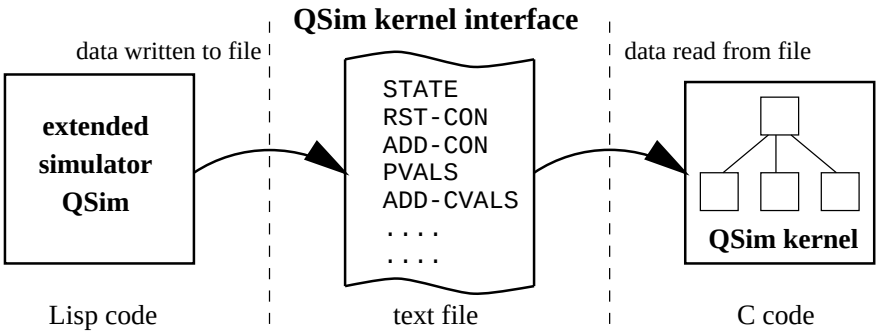


Figure 4.3: The QSIM kernel interface enables independent execution of the kernel and serves as a test environment for the evaluation.

master processes — of the constraint-filter and form-all-states, respectively. In this process, one part of the QSIM kernel interface (Section 4.2.2) is also implemented. `stdio` is a process of the Virtuoso operating system and manages the communication to the host PC. A `tuple-filter` and a `form-all-states` process is located on each slave processor. To reduce the load of the `QSim-kernel` process during execution of form-all-states, a `fas-result` process is mapped onto the root processor and a `fas-bdc` process is mapped onto an arbitrary slave processor. The `fas-bdc` process broadcasts data concerning the constraint network to all slave processors (`FAS-STRUCTURE` command). Communication to the other slave processors is established via the root processor. However, due to DMA transfer on the root processor, the CPU execution at the root processor is not interrupted. On the other hand, partial results from the `form-all-states` processes are collected by the `fas-result` process at the root processor. Only the overall result is transmitted to the `QSim-kernel` process.

Interprocess communication is established via Virtuoso mailboxes. Thus, data between processes are transfered synchronously. The execution times of the tuple-filter tasks are rather small — especially with coprocessor support — compared to the communication times with mailboxes. Therefore, to reduce data transfer times, the data transfer to the `tuple-filter` processes is directly performed via the communication ports of the processor. Direct data transfer reduces the communication times to a factor of up to 20 for small data sizes [Kau96].

4.2.2 QSIM Kernel Interface

The QSIM kernel is executed on the specialized multiprocessor system. To ease the experimental evaluation, an interface is required between the entire simulator QSIM and the kernel running on the multiprocessor system. With an appropriate interface, the kernel can be independently executed with original data from QSIM simulations. In the current implementation, data between QSIM and the kernel is transmitted using a file [Rin95].

Figure 4.3 presents the QSIM kernel interface. This interface is implemented in two parts. The qualitative simulator QSIM is extended to write all relevant data for an independent kernel execution to a file. This file serves as input for the kernel execution on the specialized multiprocessor. Therefore, the QSIM kernel must be extended to read the data from the file and update the data structures. An independent kernel execution

requires data from all active constraints, from all possible values and from all corresponding values. All this information is written to the file by the extended simulator QSIM before the kernel is executed on the specialized multiprocessor. To reduce the data transfer between QSIM and the independent kernel, only the differences of the constraints and corresponding values between the previous and the current kernel calls are written to the file.

The kernel interface allows also an easy way to test the functionality of the independent QSIM kernel. Since the result of the kernel is written to the file by the extended simulator QSIM, the results of the independent kernel can be easily compared with the original QSIM results.

4.3 Experimental Results

This section presents results of measurements taken from the prototype of the QSIM kernel multiprocessor. Each processing element of the prototype is operated at a clock frequency of 50 MHz which results in an instruction cycle time of 40 ns for the TMS320C40 processor. All execution times are measured on the root processor using a 32-bit timer of the TMS320C40. This HW timer has a resolution of two instruction cycle ticks of the processor — hence 80 ns. The singleprocessor algorithms are also executed and measured on the root processor.

The experimental results are divided corresponding to the kernel functions into the following groups: QSIM *kernel architecture without coprocessor support* and QSIM *kernel architecture with coprocessor support.*

4.3.1 QSIM Kernel Architecture without Coprocessor Support

The prototype of the QSIM kernel architecture without coprocessor support is evaluated using input data from QSIM simulation models. These models are called STLG, RCS and QSEA. The STLG model was briefly described in Section 2.2. The model RCS represents the *reaction control system* of a space shuttle [Kay92] and has 48 constraints and 45 variables. The model QSEA corresponds to the HEART model described in Section 2.2, but simulation of the QSEA model requires a lot of initial state processing.

Constraint-Filter

Parallel Tuple-Filter. Since the tuple-filter processes are implemented without coprocessors, individual tuple-filter tasks have the same execution time on all slave processors. The efficiency of all tasks is equal ($\mathsf{e}_{ij} = 1\ \forall i,j$). Therefore, scheduling algorithm LS-I is used for the evaluation of the parallel tuple-filter. Table 4.1 presents the execution times and the speedups measured for an individual state during simulation of the QSIM models. These individual states represent the maximum speedup for each model. Table 4.1 shows the sequential execution time t_{tf-seq}, the time required for the scheduling algorithm t_{tf-sch} and the task execution time $t_{tf-exe-oh}$. This task execution time includes also the overhead from the parallel implementation. The total execution time of the parallel implementation is given by the sum of t_{tf-sch} and $t_{tf-exe-oh}$. The execution times of the parallel implementation were measured using 1, 2 and 3 slave processors. With one

model/state	t_{tf-seq}	*#slaves*	t_{tf-sch}	$t_{tf-exe-oh}$	S_{tf}
STLG/5	4.798 ms	1	0.043 ms	4.585 ms	1.04
		2	0.065 ms	3.204 ms	1.47
		3	0.076 ms	2.457 ms	1.89
RCS/2	6.874 ms	1	0.104 ms	6.940 ms	0.98
		2	0.158 ms	4.641 ms	1.43
		3	0.186 ms	3.813 ms	1.72
QSEA/59	8.823 ms	1	0.089 ms	8.549 ms	1.02
		2	0.137 ms	5.683 ms	1.52
		3	0.160 ms	4.646 ms	1.84

Table 4.1: Execution times and maximum speedups of the parallel tuple-filter.

model	t_{tf-seq}	*#slaves*	t_{tf-sch}	$t_{tf-exe-oh}$	S_{tf}
STLG	2.082 ms	1	0.043 ms	2.081 ms	0.98
		2	0.068 ms	1.526 ms	1.31
		3	0.079 ms	1.272 ms	1.54
RCS	5.355 ms	1	0.107 ms	5.565 ms	0.94
		2	0.165 ms	3.846 ms	1.34
		3	0.193 ms	3.356 ms	1.51
QSEA	6.566 ms	1	0.106 ms	6.819 ms	0.95
		2	0.164 ms	4.768 ms	1.33
		3	0.197 ms	4.009 ms	1.56

Table 4.2: Execution times and average speedups of the parallel tuple-filter.

slave processor, a speedup greater than 1 is observed for the models STLG and QSEA. This is caused by a slightly improved initialization step of the parallel tuple-filter process compared to the sequential implementation.

Table 4.2 also presents the execution times and the speedups of the parallel tuple-filter implementation. However, this table shows the average execution times. Simulation of the models STLG, RCS and QSEA results in 7, 60 and 92 states, respectively. All execution times of the individual states are summed up and the average execution times are calculated. Therefore, the average speedup of the parallel tuple-filter is determined. Figure 4.4 presents graphically the maximum and average speedup for the models STLG, RCS and QSEA. The speedup of the tuple-filter is shown for 1, 2 and 3 slave processors.

Form-All-States

From the great amount of experimental data [Rie95] [Kau96], the following sections present the most interesting results. First, the partitioning methods are compared and the speedup limits are given. Second, the VBP algorithm is investigated in more detail, i.e., the number of generated subproblems of the partitioning algorithm is shown. Finally, the execution times and speedups of the parallel implementation are presented.

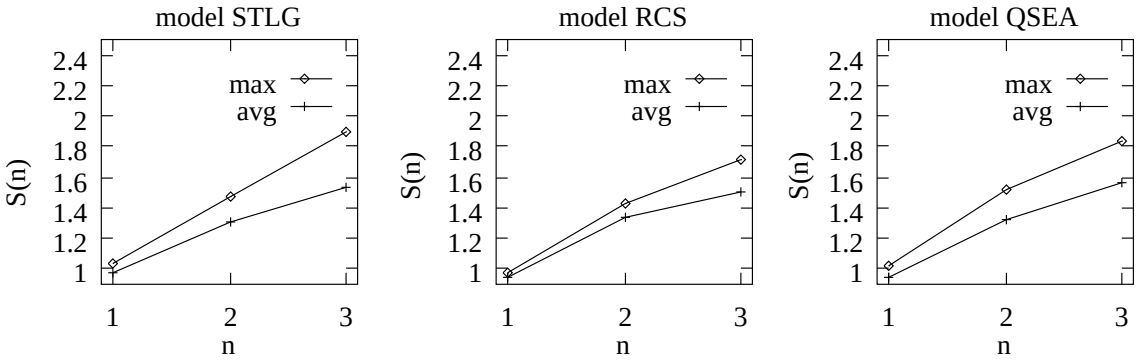


Figure 4.4: Maximum and average speedup of the parallel tuple-filter for the models STLG, RCS and QSEA using n slave processors.

		16 subtasks		*64 subtasks*		*256 subtasks*	
	model	$t_{fas-tot}$	$t_{fas-max}$	$t_{fas-tot}$	$t_{fas-max}$	$t_{fas-tot}$	$t_{fas-max}$
CBP	STLG	9.09	3.21	13.51	3.19	37.77	3.09
	RCS	741.97	450.93	763.48	442.79	947.65	231.14

Table 4.3: Constraint-based partitioning. The execution times of the partitioned subproblems $t_{fas-tot}$ and $t_{fas-max}$ are given in ms.

Partitioning Method. The two partitioning methods CBP and VBP are evaluated using the execution times $t_{fas-seq}$, $t_{fas-tot}$ and $t_{fas-max}$ from Equation 3.19 in Section 3.4.5. The execution times are measured with the QSIM models STLG and RCS. The execution times $t_{fas-tot}$ and $t_{fas-max}$ are presented in Table 4.3 and Table 4.4 for the partitioning methods CBP and VBP, respectively. Table 4.4 shows these execution times for the different heuristics of the VBP method. A further interesting point is the influence of the number of generated subproblems on $t_{fas-tot}$ and $t_{fas-max}$. Three cases are considered — the CSP is partitioned into at most 16, 64 and 256 subproblems. The corresponding execution times are also presented in Table 4.3 and Table 4.4. For CBP, the execution times of the unpartitioned problem $t_{fas-seq}$ are 6.12 ms for STLG and 726 ms for RCS. For VBP, these execution times are 6.83 ms for STLG and 805.63 ms for RCS. The small increase of $t_{fas-seq}$ compared to CBP is due to different memory mappings on the target processor TMS320C40.

Due to the exploitation of the dependencies between adjacent constraints, the VBP method achieves better results than the CBP method. Especially, the great increase of the total execution time $t_{fas-tot}$ and the execution time of the maximum subproblem $t_{fas-max}$ lead to poor parallel performance with CBP. VBP generates shorter maximum subproblems, and in some cases $t_{fas-tot}$ is shorter than the execution time of the unpartitioned problem.

A comparison of the speedup limits for the heuristics VBP-INST and VBP-CON is shown in Figure 4.5. The speedups are presented for up to 8 processors. The heuristic VBP-CON is used to partition the CSP for the evaluation of the parallel form-all-states implementation.

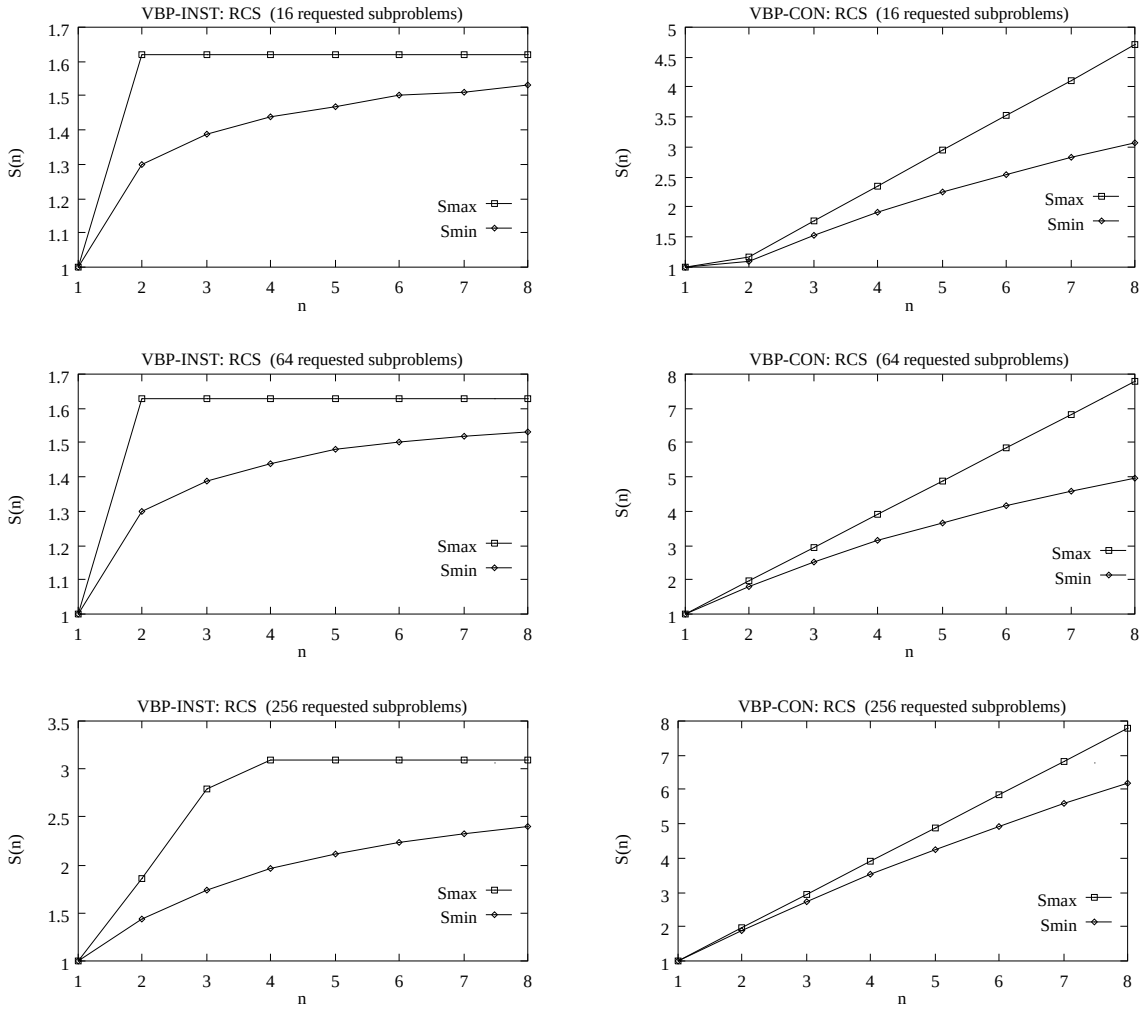


Figure 4.5: Speedup limits of VBP for the RCS model. Minimum speedup S_{min} and maximum speedup S_{max} for the partitioning heuristics VBP-INST and VBP-CON are shown in the left and right column plots.

		16 subtasks		*64 subtasks*		*256 subtasks*	
	model	$t_{fas-tot}$	$t_{fas-max}$	$t_{fas-tot}$	$t_{fas-max}$	$t_{fas-tot}$	$t_{fas-max}$
VBP-INST	STLG	6.96	3.49	6.55	3.28	6.34	3.18
	RCS	739.48	497.93	746.44	495.45	865.42	259.95
VBP-CON	STLG	6.02	3.29	5.60	1.99	9.40	1.53
	RCS	1370.53	104.29	826.66	67.86	825.96	30.71
VBP-DOM	STLG	34.96	3.15	24.73	1.99	18.12	1.70
	RCS	3932.79	265.16	15119.29	252.02	4739.37	244.82
VBP-TUP	STLG	7.65	2.15	5.60	1.99	9.40	1.53
	RCS	1401.59	103.31	779.47	51.08	780.95	44.95

Table 4.4: Variable-based partitioning. The execution times of the partitioned subproblems $t_{fas-tot}$ and $t_{fas-max}$ are given in ms.

	STLG/2		*QSEA/53*	
`maxsub`	`nsub`	$t_{fas-part}$	`nsub`	$t_{fas-part}$
1	1	0.136 ms	1	0.495 ms
2	1	0.810 ms	1	1.557 ms
4	2	0.972 ms	1	1.538 ms
8	2	1.280 ms	5	2.449 ms
16	2	1.284 ms	13	4.000 ms
32	4	2.584 ms	29	7.318 ms

Table 4.5: Number of requested `maxsub` and generated `nsub` subproblems using VBP-CON.

Number of generated Subproblems. Table 4.5 presents the number of generated subproblems `nsub` and the execution times required by the partitioning algorithm $t_{fas-part}$ using the partitioning method VBP-CON. The number of requested subproblems `maxsub` ranges from 1 to 32. The states which result in the greatest number of generated subproblems are shown for both QSIM models STLG and QSEA. The CSP of the QSEA model is partitioned into many subproblems — nearly the requested number is achieved. This is caused by the rather complex CSP of this state; many tuples have survived the tuple- and the Waltz-filter. The situation is quite different for the CSP of the STLG model. The number of generated subproblems is rather small. Many subspaces of the CSP are discarded by the `generate-subproblem` algorithm.

The first row of Table 4.5 with `maxsub` = 1 presents the time which is required for the initialization of the data structures of the parallel implementation — all tuples of the CSP are just copied without any checking. Hence, this execution time depends basically on the total number of tuples of the CSP. If more subproblems are requested, checking of tuple values occurs and the partitioning time increases.

Parallel Form-All-States. The speedup of the parallel form-all-states implementation is evaluated with the QSIM models STLG, RCS and QSEA. Table 4.6 presents the execution times and the speedups measured for an individual state during simulation of the

model/state	$t_{fas-seq}$	*#slaves*	$t_{fas-part}$	$t_{fas-exe-oh}$	S_{fas}
STLG/2	4.095 ms	1	0.808 ms	1.052 ms	2.20
		2	0.975 ms	1.003 ms	2.07
		3	1.062 ms	1.017 ms	1.97
		4	1.190 ms	1.034 ms	1.84
		5	1.333 ms	1.122 ms	1.67
		6	1.461 ms	1.116 ms	1.59
		7	1.568 ms	1.146 ms	1.51
RCS/26	47.372 ms	1	5.565 ms	26.481 ms	1.48
		2	5.696 ms	15.056 ms	2.28
		3	5.836 ms	11.212 ms	2.78
		4	5.960 ms	9.233 ms	3.12
		5	6.097 ms	8.676 ms	3.21
		6	6.240 ms	7.789 ms	3.38
		7	6.376 ms	7.450 ms	3.43
QSEA/53	1298.144 ms	1	7.050 ms	1320.989 ms	0.98
		2	7.180 ms	679.616 ms	1.89
		3	7.318 ms	448.253 ms	2.85
		4	7.437 ms	352,474 ms	3.61
		5	7.718 ms	294.201 ms	4.30
		6	7.724 ms	259.381 ms	4.86
		7	7.842 ms	248.492 ms	5.06

Table 4.6: Execution times and maximum speedups of the parallel implementation of form-all-states.

models. These individual states represent the maximum speedup for each model. Table 4.6 shows the sequential execution time $t_{fas-seq}$, the time required for the partitioning algorithm VBP-CON $t_{fas-part}$ and the task execution time $t_{fas-exe-oh}$. This task execution time includes also the overhead from the parallel implementation. The execution times of the parallel algorithm were measured using 1 to 7 slave processors, and 32 subproblems were requested from the partitioning algorithm.

Parallel execution of the STLG state reveals an interesting behavior. First, superlinear speedup is observed. This occurs because the partitioning algorithm discards many inconsistent subproblems. The total execution time of the remaining consistent subproblems is smaller than the execution time of the unpartitioned problem ($t_{fas-tot} < t_{fas-seg}$). Second, the speedup decreases with the number of slave processors. This occurs because only one subproblem is generated by the partitioning algorithm. The overhead increases with the number of slave processors. Therefore, the speedup decreases. A speedup greater than 1 using one slave processor is also observed for the RCS model. In this case, the partitioning algorithm discards also many inconsistent subproblems.

Table 4.7 presents the average execution times and the average speedups of the parallel form-all-states implementation for the models STLG, RCS and QSEA. Figure 4.6 presents graphically the maximum and average speedups for these models. The speedup of the parallel form-all-states implementation is shown for 1 to 7 slave processors.

model	$t_{fas-seq}$	*#slaves*	$t_{fas-part}$	$t_{fas-exe-oh}$	S_{fas}
STLG	2.348 ms	1	1.476 ms	3.125 ms	0.51
		2	1.612 ms	2.017 ms	0.65
		3	1.737 ms	1.937 ms	0.64
		4	1.865 ms	1.579 ms	0.68
		5	1.997 ms	1.601 ms	0.65
		6	2.120 ms	1.609 ms	0.63
		7	2.255 ms	1.648 ms	0.60
RCS	14.856 ms	1	3.933 ms	20.394 ms	0.61
		2	4.072 ms	12.280 ms	0.91
		3	4.205 ms	10.960 ms	0.98
		4	4.334 ms	10.347 ms	1.01
		5	4.475 ms	10.288 ms	1.01
		6	4.607 ms	10.157 ms	1.01
		7	4.747 ms	10.107 ms	1.00
QSEA	151.657 ms	1	4.389 ms	203.920 ms	0.73
		2	4.514 ms	100.820 ms	1.44
		3	4.656 ms	68.840 ms	2.06
		4	4.786 ms	52.844 ms	2.63
		5	4.929 ms	44.235 ms	3.08
		6	5.050 ms	38.131 ms	3.51
		7	5.187 ms	34.576 ms	3.81

Table 4.7: Execution times and average speedups of the parallel implementation of form-all-states.

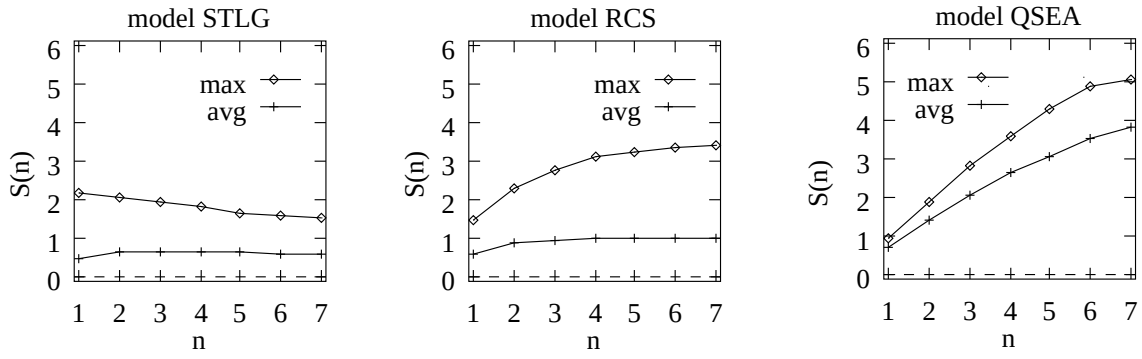


Figure 4.6: Maximum and average speedup of the parallel implementation of form-all-states for the models STLG, RCS and QSEA using n slave processors.

model/state	t_{seq}	*#slaves*	t_{par}	S_{par}
RCS/2	9.131 ms	1	9.301 ms	0.98
		2	7.057 ms	1.29
		3	6.256 ms	1.46
QSEA/53	1322.584 ms	1	1355.289 ms	0.98
		2	707.242 ms	1.87
		3	472.224 ms	2.80

Table 4.8: Execution times and maximum speedups of the QSIM kernel architecture.

model	t_{seq}	*#slaves*	t_{par}	S_{par}
RCS	8.824 ms	1	9.140 ms	0.97
		2	7.479 ms	1.18
		3	7.018 ms	1.26
QSEA	158.223 ms	1	215.233 ms	0.74
		2	110.266 ms	1.43
		3	77.701 ms	2.04

Table 4.9: Execution times and average speedups of the QSIM kernel architecture.

QSIM Kernel

The speedup of the overall QSIM kernel architecture is evaluated by the models RCS and QSEA. Table 4.8 presents the execution times and the speedups measured for an individual state during simulation of these models. These individual states represent the maximum speedup for each model. This table shows the sequential execution time of the kernel t_{seq} — including the time required for tuple-filter, Waltz-filter and form-all-states — the parallel execution time t_{par} and the achieved speedup of the parallel implementation S_{par}. The parallel execution times were measured using 1, 2 and 3 slave processors.

Table 4.9 presents the average execution times and the average speedups of the parallel kernel implementation for the models RCS and QSEA. Figure 4.7 presents graphically the maximum and average speedups for these models. The speedup of the parallel form-all-states implementation is shown for 1, 2 and 3 slave processors.

4.3.2 QSIM Kernel Architecture with Coprocessor Support

The coprocessors do not influence the execution time of the parallel implementation of form-all-states. Therefore, the effect of the coprocessor support is experimentally evaluated only for the constraint-filter.

There are many different possibilities to attach the coprocessors to the slave processors. In general, if the order of the slave processors is not distinguished, i.e., which coprocessor is attached to which slave processor, and there are k different types of coprocessors and n slave processors, the number of different assignments is given by

$$\left(\begin{array}{c} n+k-1 \\ n \end{array} \right). \tag{4.1}$$

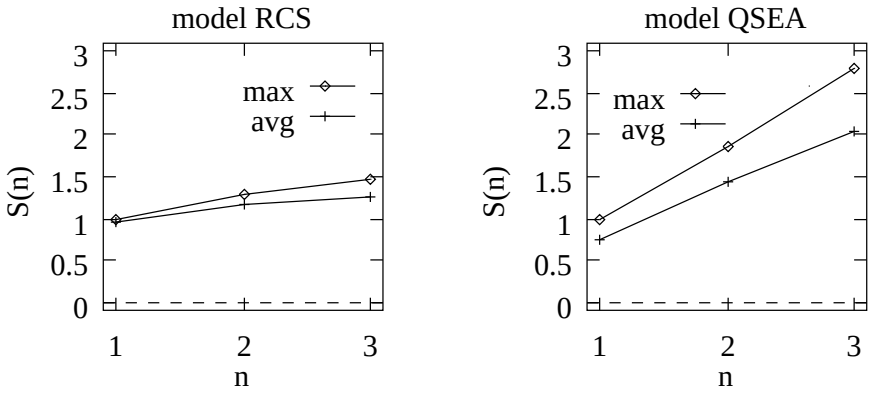


Figure 4.7: Maximum and average speedup of the QSim kernel for the models RCS and QSEA using n slave processors.

coprocessor assignment	*used coprocessors*
A1	none
A2	2 ADD, 1 MULT
A3	1 ADD, 2 MULT

Table 4.10: Coprocessor assignments for the experimental evaluation. Each slave processor is equipped with none (A1) or exactly one coprocessor (A2, A3).

For example, for 5 coprocessor types and 3 slave processors, 35 different assignments are possible. This number increases by orders of magnitude if more than one coprocessor type can be attached to one slave processor, or if the order of the slave processors is distinguished. It is not feasible to evaluate experimentally each coprocessor assignment. Therefore, three representative coprocessor assignments using three slave processors were chosen for the experimental evaluation. Table 4.10 presents these assignments. ADD and MULT coprocessor types were chosen because these coprocessors have the greatest execution time improvements compared to the software implementation [Pla96].

Constraint-Filter

The effect of the coprocessor support on the execution time of the constraint-filter is evaluated with three different input data sets. These data sets are artificially constructed and represent worst-case (M1), average-case (M2) and best-case (M3) for the sequential execution time of the tuple-filter. Input data set M1 has the longest execution times for the tuple-filter tasks, whereas data set M3 has the shortest execution times. The execution times of all tuple-filter tasks are between the execution times from any task of M3 and the execution times from any task of M1. Each data set is constructed of 30 tuple-filter calls. Input data set M1 consists of 30 MULT tuple-filter calls. Each tuple-filter has to check 64 tuples. On the contrary to M1, each tuple-filter of M3 only has to check one tuple of a D/DT constraint. Finally, M2 consists of 30 tuple-filter calls of different types and with a different number of tuples.

	LS-I		*LS-II*		*LS-III*	
coproc. ass.	t_{tf-sch}	t_{tf-exe}	t_{tf-sch}	t_{tf-exe}	t_{tf-sch}	t_{tf-exe}
A1	0.123 ms	2.087 ms	0.729 ms	2.041 ms	1.300 ms	1.753 ms
A2	0.125 ms	0.661 ms	0.702 ms	0.649 ms	1.272 ms	0.633 ms
A3	0.143 ms	0.604 ms	0.701 ms	0.562 ms	1.273 ms	0.587 ms

Table 4.11: Comparison of the scheduling algorithms LS-I, LS-II and LS-III.

Scheduling Algorithms. Table 4.11 compares the three different scheduling algorithms for the constraint-filter LS-I, LS-II and LS-III, respectively. The input data set M2 is used for this comparison. For each coprocessor assignment, the execution time of the scheduling algorithm t_{tf-sch} and the execution time of all tuple-filter tasks t_{tf-exe} are shown. The tuple-filter tasks are executed on three slave processors. The overall execution time of the tuple-filter tasks decreases with the advanced scheduling algorithms LS-II and LS-III. However, the execution time for the scheduling algorithm t_{tf-sch} is increased more than t_{tf-exe} is reduced. Scheduling algorithm LS-I achieves the best speedup for these coprocessor assignments. Therefore, algorithm LS-I is used to schedule tuple-filter tasks for the evaluation of the parallel tuple-filter prototype with coprocessor support.

Parallel Tuple-Filter with Coprocessor Support. The speedup of the parallel tuple-filter with coprocessor support is evaluated using the data sets M1, M2 and M3. These constructed data sets were used because the required number of CCF coprocessors was not available. However, the exact coprocessor execution times of all tasks from M1, M2 and M3 were known. The knowledge of these execution times allows the experimental evaluation of the parallel tuple-filter with coprocessor support. Whenever a tuple-filter task is executed on a slave processor "equipped" with a corresponding coprocessor, the slave processor waits until the known execution time of this task has been expired. With this procedure, the evaluation of the exact timing behavior of the tuple-filter with coprocessor support is possible, and the corresponding speedups can be determined.

Table 4.12 presents the execution times and the speedups for the parallel tuple-filter with coprocessor support. The parallel execution time t_{tf-par} is measured with the three coprocessor assignments A1, A2 and A3 using 3 slave processors. The speedups for the individual input data sets and coprocessor assignments are also shown in Figure 4.8. Data sets M1 and M3 represent the maximum and minimum execution times of the tuple-filter tasks. These execution times differ by two orders of magnitude. Therefore, the influence of the overhead is much greater for M3 than for M1. This results in a rather low speedup for M3. In this case, the sequential implementation is actually faster than the parallel implementation with coprocessor support. This low speedup is nearly constant for all coprocessor assignments, because only tuple-filter tasks of type D/DT are included in the data set M3, and no D/DT coprocessor is used in the coprocessor assignments A1, A2 and A3. On the other hand, parallel execution of M1 with coprocessor assignment A3 is approximately 20 times faster than the sequential execution. Figure 4.8 presents graphically the speedups of the parallel tuple-filter with coprocessor support using the coprocessor assignments A1, A2 and A3 for the data sets M1, M2 and M3.

		A1		A2		A3	
data set	t_{tf-seq}	t_{tf-par}	S_{tf}	t_{tf-par}	S_{tf}	t_{tf-par}	S_{tf}
M1	85.795 ms	30.772 ms	2.79	6.427 ms	13.35	4.469 ms	19.20
M2	5.708 ms	3.074 ms	1.86	1.631 ms	3.50	1.611 ms	3.54
M3	0.965 ms	1.239 ms	0.78	1.201 ms	0.80	1.220 ms	0.79

Table 4.12: Execution times and speedups of the parallel tuple-filter with coprocessor support using three slave processors.

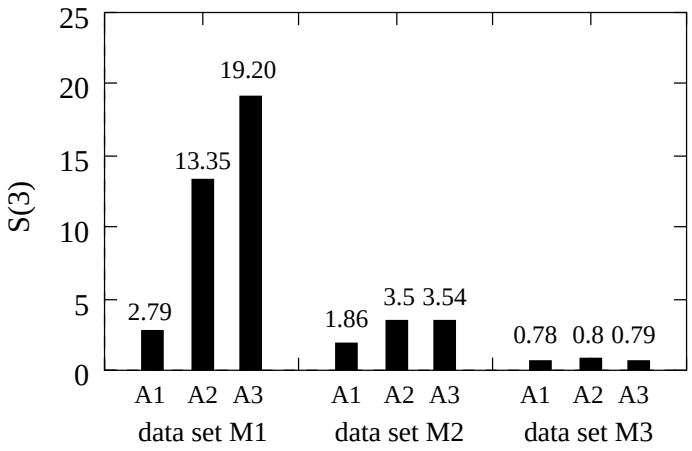


Figure 4.8: Speedup of the parallel tuple-filter with coprocessor support using three slave processors.

model	α	β	$\bar{S}_{par}$	$S_{par}(3)$
RCS_{max}	0.78	0.12	1.87	1.46
RCS_{avg}	0.78	0.12	1.40	1.26
QSEA_{max}	0.05	0.95	2.77	2.80
QSEA_{avg}	0.05	0.95	2.02	2.04

Table 4.13: Estimated overall speedup $\bar{S}_{par}$ and measured overall speedup S_{par} of the QSIM kernel architecture for the maximum- and average-case of the models RCS and QSEA using three slave processors.

4.4 Discussion

The experimental results proved that parallel execution of the QSIM kernel on a specialized computer architecture achieves a significant speedup compared to the sequential implementation. This section discusses some implications of the experimental results in more detail, i.e., concerning (i) the overall speedup, (ii) the scheduling algorithms, (iii) the conditional parallel execution of form-all-states, (iv) the simulation models for technical applications and (v) the integration of QSIM into real-time applications.

Overall Speedup

The measured overall speedup S_{par} of the QSIM kernel architecture is compared with the overall speedup estimation $\bar{S}_{par}$ from Equation 3.27 in Section 3.5.3. The overall speedup was measured with the models RCS and QSEA. Therefore, the execution time ratios α and β for these models must be determined. These ratios are given by the empirical running time analysis of QSIM. The speedups of the kernel functions, constraint-filter and form-all-states, are given by the experimental evaluation.

Table 4.13 presents the comparison of the estimated with the measured overall speedup. For the model QSEA, there is only a small deviation between the estimated and measured speedup. For the model RCS, the difference between the measured and the estimated speedup is caused by the influence of the execution time of the Waltz-filter. This execution time is not considered in the speedup of the constraint-filter. For the estimation, the speedup of the parallel tuple-filter is used.

In the experimental evaluation, at most three slave processors are used to determine the speedups of the QSIM kernel architecture. As presented in Table 4.13, only a moderate speedup is achieved for the model RCS. In this model, there are many states with only a small number of tuples. A parallel execution of these states — especially for form-all-states — results in a poor performance. Therefore, the overall performance is only moderate. However, the performance can be improved by the coprocessor support. As shown in Figure 4.8, the speedup of the parallel tuple-filter is strongly increased by the coprocessor support for the data sets M1 and M2. Furthermore, the utilization of more slave processors will also result in an additional performance improvement of the QSIM kernel architecture.

The speedup of the parallel implementation for form-all-states is evaluated with up to seven slave processors. Due to the great execution time ratio of form-all-states for the model QSEA ($\beta = 0.95$), improvements on the speedup of the parallel implementation of

	$QSEA_{max}$		$QSEA_{avg}$	
n_{fas}	$\bar{S}_{par}$	S_{par}	$\bar{S}_{par}$	S_{par}
4	3.44	3.51	2.54	2.59
5	4.03	4.15	2.93	2.96
6	4.49	4.66	3.30	3.34
7	4.65	4.84	3.55	3.61

Table 4.14: Estimated overall speedup $\bar{S}_{par}$ and measured overall speedup S_{par} of the QSIM kernel architecture for the maximum- and average-case of the model QSEA using three slave processors for the parallel tuple-filter and 4 to 7 slave processors for the parallel implementation of form-all-states n_{fas}.

form-all-states have a great influence on the overall speedup S_{par}. This can be seen from Table 4.14. This table presents a comparison of the estimated overall speedup $\bar{S}_{par}$ with the measured overall speedup S_{par} for the model QSEA. For this comparison, three slave processors are used for the parallel tuple-filter and 4 to 7 slave processors are used for the parallel implementation of form-all-states. The maximum speedup and the average speedup are shown in Table 4.14. For this model, an overall speedup of 4.84 is achieved with seven slave processors for form-all-states.

Scheduling Algorithms

The comparison of the three scheduling algorithms LS-I, LS-II and LS-III reveals that in all considered cases LS-I performs better than LS-II and LS-III. For LS-II and LS-III, the effect of the improved execution time of all tasks t_{tf-exe} is compensated by the increased execution time of the scheduling algorithm t_{tf-sch}. A reason for this behavior is the small number of slave processors used in the experimental evaluation. With three slave processors, even the performance ratio of LS-I is better than the performance ratio of LS-II and LS-III, respectively. For LS-II and LS-III, the performance ratios become smaller than the performance ratio of LS-I for more than 6 slave processors and 5 slave processors, respectively (compare Table 3.2 in Section 3.3.4). Therefore, a better performance of the scheduling algorithms LS-II and LS-III in comparison to LS-I is expected for a greater number of slave processors.

Conditional Parallel Execution of Form-All-States

In the experimental evaluation, there are some states in all considered QSIM models where the parallel execution of form-all-states results in poor performance. For these states, the parallel implementation is actually slower than the sequential implementation. One reason for this behavior is that the complexity of the CSP changes from one state to the next during simulation. Especially, the number of consistent tuples changes. The number of tuples has a great influence on the partitioning algorithm. If the CSP has only a few tuples, the CSP is hard to partition into many subproblems by the VBP algorithm. The few subproblems can result in an unbalanced workload. Moreover, if each constraint has only one tuple, the CSP cannot be partitioned. On the other hand, if every constraint has many tuples, the CSP can be partitioned into more subproblems. This results in a

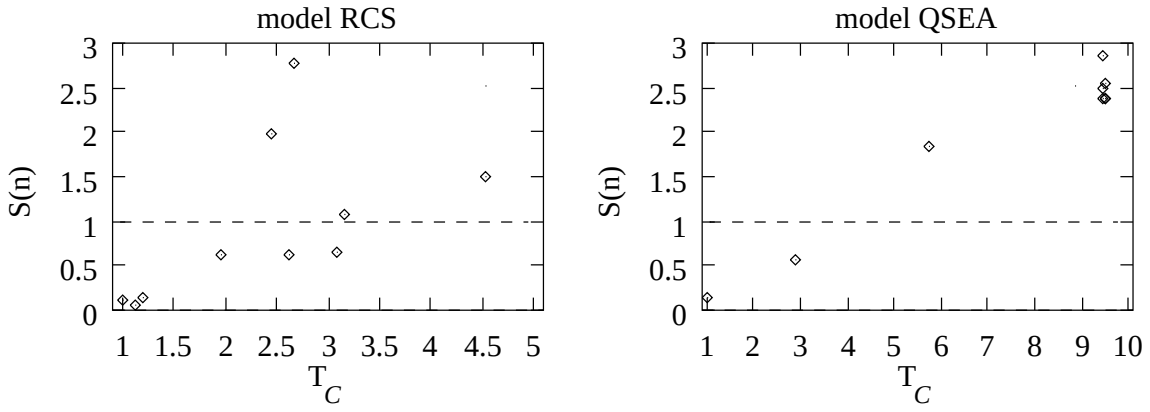


Figure 4.9: Achieved speedup over the average number of tuples per constraint T_C.

better parallel performance. Another reason for poor parallel performance is caused by the execution time of the partitioning algorithm. This algorithm is executed sequentially. Especially for small CSPs with short sequential execution times, the execution time of the partitioning algorithm limits the speedup.

To improve the overall performance, it is worth considering a conditional parallel execution of form-all-states. States which achieve a poor parallel performance are executed sequentially; all other states are executed in parallel. Therefore, a criterion at run-time is required to decide whether sequential or parallel execution is preferable. A possible approach for such a criterion is the *average number of tuples per constraint* T_C over all constraints of the QDE. With some qualification, states with a greater T_C achieve a better speedup than states with a smaller T_C. Therefore, states with T_C above a certain limit should be executed in parallel. Figure 4.9 plots the achieved speedup using three slave processors over the average number of tuples per constraint T_C for all states of the models RCS and QSEA. For the model QSEA, only two states with $T_C = 1$ and $T_C = 2.88$ result in a speedup $S(n) < 1$. All other states with $T_C > 2.88$ achieve a speedup $S(n) > 1$. The situation differs slightly for the model RCS. There are two states which achieve a speedup $S(n) < 1$, although states with smaller T_C have a speedup $S(n) > 1$.

As presented in Figure 4.9, there are some states where a decision only based on T_C can be misleading. Therefore, this criterion must be expanded to achieve better results. One possibility is to include additional information about the model, like the number of variables and the number of constraints. Another way to improve this criterion is to exploit information from the QSIM execution, i.e., whether QSIM executes initial state processing or generation of successor states.

QSIM Models for Technical Systems

The empirical running time analysis of QSIM as well as the experimental evaluation of the QSIM kernel architecture are mainly based on models from the QSIM source code package. Most of these models represent simple to medium complex dynamic systems. In general, the QSIM kernel architecture performs better with complex models than with simple models. The influence of the overhead of the parallel implementation is smaller with complex models than with simple models.

However, for the application of QSIM in technical systems, qualitative models of these systems are required. The description of "real-world" technical systems on even a qualitative level leads to large and possibly hierarchically structured qualitative models. It is expected that these models have a higher complexity than the considered models of the experimental evaluation of the QSIM kernel architecture. More variables and constraints are required to model these technical systems appropriately. Furthermore, applying QSIM in monitoring and diagnosis systems will result in QSIM kernel calls with incompletely described states. The monitoring or diagnosis algorithm does not know the qualitative values of each variable of the model. Therefore, several alternatives for the qualitative value of the variable must be considered. This results in a lot of possible values for the variable and, hence, a lot of tuples. The models and states have a higher inherent parallelism, and a better performance of the QSIM kernel architecture is expected.

Integration of QSIM into Real-Time Applications

The QSIM kernel is often used as a basic operation in QR applications like monitoring and diagnosis in dynamic systems [LN93] [SM95]. To apply these techniques in real-time environments, guaranteed limits for the execution times must be known. The *predictability* of execution times rather than the performance becomes extremely important [KV94]. The algorithmic complexity analysis estimates the worst-case execution time of an algorithm with regard to its problem size. However, exact limits cannot be derived only by a complexity analysis; absolute times can only be derived in combination with an actual implementation. The execution times of the "elementary operations" of the complexity analysis must be known.

The execution times of the QSIM kernel functions strongly depend on the input data. In the following sections the individual kernel functions are briefly investigated with regard to their execution times.

Constraint check functions (CCFs). The execution time of a CCF depends on the constraint type, the set of cvals tuples and the tuple which must be checked. In the CCF, the tuple is checked by subsequent calls to several subfunctions, and the number of subfunctions is dependent on the number of cvals tuples. The overall result of the CCF is calculated by a *short circuit evaluation* of all subfunctions. Whenever one subfunction returns a negative overall result, the calculation is aborted and a negative result is returned. Therefore, the longest execution time is required, if an overall positive result is returned; all subfunctions must be executed.

Tuple-filter. The execution time of the tuple-filter was already estimated in Section 3.3.4. It is basically dependent on the constraint type and the number of tuples. Therefore, the longest execution time is required, if the maximum number of tuples must be checked. For successor state generation, this number is given by 64.

Form-all-states. The maximum execution time of form-all-states can only be estimated by the algorithmic complexity which is $O(d^V)$. However, even for a model with a moderate number of variables, this results in a huge number of elementary operations. For example, for a model with 20 variables, the maximum number of elementary operations is given by 4^{20} which is approximately 10^{12}.

Therefore, to predict the execution time limits for the QSIM kernel a lot of information is required, and especially for form-all-states, the limit is not feasible for practical applications. Furthermore, this information is only available during the simulation of an actual model. But the maximum execution times are often required in advance — they should be derived from the model, basically the set of constraints and the set of variables. The situation is even more complicated for the prediction of the execution time for a complete simulation of a model. The number of states is not known in advance, and data which are required for the determination of the execution times of kernel functions change from one state to the next.

The lack of information about guaranteed execution times is a general problem when methods from AI are applied in real-time systems. Traditionally, AI systems have been developed without much attention to the resource limitations of real-time systems. The execution time of the AI method is a major limitation. In early systems, AI methods have been tested and shown to operate quickly enough to meet deadlines for test scenarios. However, complex behavioral interactions and domain variations beyond the set of tested scenarios may lead to violations of the deadlines [Sta88].

In general, the application of QSIM in real-time environments by satisfying guaranteed execution time limits for each kernel call is not feasible. Therefore, other ways of integrating qualitative simulation into real-time systems must be considered. Musliner et al. [Mus95] present three principal ways to combine real-time systems and AI techniques to so-called *real-time AI* systems.

- *Embedding AI in real-time*
  The goal of this approach is to be "intelligent *in* real-time", so that the system's reasoning capacity is applied to each decision before its deadline. The fundamental problem with this approach is that AI tasks are generally ill-suited to real-time scheduling due to their unpredictable execution times. This problem can be solved, if the AI tasks are incremental, interruptible algorithms. *Any-time* algorithms [DB88] can be interrupted before any deadline, and partial results can be used. These results can be refined and completed at a later time.

- *Embedding real-time in AI*
  This alternative approach essentially assumes that the overall system typically performs AI techniques, but that, under some circumstances, these techniques will be short-circuited in favor of a real-time action.

- *Cooperating real-time and AI*
  The third approach is based on separate real-time and AI subsystems. In the cooperative approach, the AI subsystem is isolated from the real-time environment by the real-time subsystem. The AI subsystem can interact with the real-time subsystem in various ways. On the one hand, higher-level AI-processes can only adjust parameters for the real-time system, whereas on the other hand AI components can design complete real-time control plans.

Chapter 5

Conclusions and Further Work

The presented thesis deals with the design, the prototype implementation and the experimental evaluation of a scalable multiprocessor system for the kernel of the qualitative simulator QSIM. This special-purpose computer architecture is the first work known so far which improves the execution time of QSIM by means of parallelization and mapping onto a specialized computer architecture. The prototype implementation of this approach reveals some advantages (+) and some disadvantages (-) which are listed below:

+ The prototype implementation for both kernel functions, constraint-filter and form-all-states, achieves a significant execution time improvement and speedup compared to a sequential implementation.

+ The scalability of the QSIM kernel architecture is demonstrated by a prototype implementation.

- An execution time improvement is not achieved for all states and/or models. This is caused by the overhead of the parallel implementation and — especially for form-all-states — by the execution time of the sequential partitioning algorithm.

- The exact execution times of the QSIM kernel cannot be determined in advance. This is due to the strong input data dependency of QSIM and the NP-completeness of the algorithm form-all-states. Therefore, feasible guaranteed execution times of the QSIM kernel architecture cannot be given for the simulation of individual states and/or models.

Further work on this project is presented for two areas, (i) improvement of the QSIM kernel architecture and (ii) application of the QSIM kernel architecture.

Improvement of the QSIM Kernel Architecture

- Partitioning algorithms for CSPs should be improved to generate more subproblems and to reduce the execution time of the partitioning algorithm.

- Criteria for deciding whether form-all-states is executed sequentially or in parallel should be investigated in more detail.

- Larger multiprocessor systems should be implemented to determine the limits of the achievable speedup. The multiprocessor system should also be equipped with coprocessors.

- Important parameters of the computer architecture, like the number of processing elements, the number and type of coprocessors and the topology, should be investigated with regard to QSIM simulation models. The determination of the optimal architectural parameters should be possible by an (automatic) analysis of the simulation model.

Application of the QSIM Kernel Architecture

- The QSIM kernel architecture should be integrated into applications of qualitative simulation, like monitoring and diagnosis of technical processes, to demonstrate the feasibility of this special-purpose computer architecture in *real* systems. The real-time capabilities of the QSIM kernel architecture should be investigated in more detail.

Bibliography

[BGH+95] Eugen Brenner, Robert Ginthör, Robert Hranitzky, Marco Platzner, Bernhard Rinner, Christian Steger, and Reinhold Weiss. High–Performance Simulators Based on Multi–TMS320C40. In *Proceedings of the Fifth Annual Texas Instruments TMS320 Educators Conference*, Houston, USA, August 1995.

[BGM+91] Eugen Brenner, Jörg Grabner, Michael Moosburger, Günther Otschko, Karlheinz Schlögl, Jinhua Song, Christian Steger, and Reinhold Weiss. Design and Implementation of a Distributed Real–Time Expert–System for Fault Diagnosis in Modular Manufacturing Systems. In *Proceedings of Euromicro 1991*, pages 799–806, Vienna, Austria, September 1991.

[BK92] Daniel Berleant and Benjamin Kuipers. Qualitative-Numeric Simulation with Q3. In Boi Faltings and Peter Struss, editors, *Recent Advances in Qualitative Physics*, chapter 1, pages 3–16. MIT Press Cambridge, Massachusetts, 1992.

[Bur90a] Bernard Burg. Parallel Forward Checking: First part. Technical Report TR-594, Institute for New Generation Computer Technology, September 1990.

[Bur90b] Bernard Burg. Parallel Forward Checking: Second part. Technical Report TR-595, Institute for New Generation Computer Technology, September 1990.

[CK93] Daniel J. Clancy and Benjamin Kuipers. Behavior Abstraction for Tractable Simulation. In *Working Papers from the Seventh International Workshop on Qualitative Reasoning about Physical Systems (QR-93)*, Orcas Island, Washington, 1993.

[CM94] Minhwa Chung and Dan Moldovan. Applying Parallel Processing to Natural-Language Processing. *IEEE Expert*, 9(1):36–44, February 1994.

[CS92] Paul R. Cooper and Michael J. Swain. Arc consistency: parallelism and domain dependence. *Artificial Intelligence*, 58:207–235, 1992.

[Dav84] Randall Davis. Diagnostic Reasoning Based on Structure and Behavior. *Artificial Intelligence*, 24:347–410, 1984.

[Dav94] Ernest Davis. An engaging exploration of QSIM and its extensions. *IEEE Expert*, 9(6):70–71, December 1994. book review.

[DB88] T. Dean and M. Boddy. An Analysis of Time-Dependent Planning. In *Proceedings of the 7th National Conference on Artificial Intelligence*, pages 49–54, St.Paul, Minnesota, 1988.

[DJ81] Ernest Davis and Jeffrey M. Jaffe. Algorithms for Scheduling Tasks on Unrelated Processors. *Journal of the Association for Computing Machinery*, 28(4):721–736, October 1981.

[DK91] Daniel Dvorak and Benjamin Kuipers. Process Monitoring and Diagnosis: A Model-Based Approach. *IEEE Expert*, 5(3):67–74, June 1991.

[dKB84] Johan de Kleer and John Seely Brown. A Qualitative Physics Based on Confluences. In Daniel G. Bobrow, editor, *Qualitative Reasoning about Physical Systems*, pages 169–203. Elsevier Sience Publishers B.V., Amsterdam, Netherlands, 1984.

[dKW87] Johan de Kleer and Brian C. Williams. Diagnosing Multiple Faults. *Artificial Intelligence*, 32:97–130, 1987.

[DM93] Ronald F. DeMara and Dan I. Moldovan. The SNAP–1 Parallel AI Prototype. *IEEE Transactions on Parallel and Distributed Systems*, 4(8):841–854, August 1993.

[DP89] Rina Dechter and Judea Pearl. Tree Clustering for Constraint Networks. *Artificial Intelligence*, 38:353–366, 1989.

[Dvo92] Daniel Luis Dvorak. *Monitoring and Diagnosis of Continuous Dynamic Systems using Semiquantitative Simulation.* PhD thesis, University of Texas at Austin, 1992.

[Fis88] Paul A. Fishwick. Qualitative simulation: Fundamental concepts and issues. *AI and Simulation*, pages 25–31, 1988.

[For84] Kenneth D. Forbus. Qualitative Process Theory. In Daniel G. Bobrow, editor, *Qualitative Reasoning about Physical Systems*, pages 85–168. Elsevier Sience Publishers B.V., Amsterdam, Netherlands, 1984.

[FPR95] Gerald Friedl, Marco Platzner, and Bernhard Rinner. A Special–Purpose Coprocessor for Qualitative Simulation. In *Proceedings of the First International EURO-PAR Conference*, pages 695–698, Stockholm, Sweden, August 1995.

[FS92] Boi Faltings and Peter Struss, editors. *Recent Advances in Qualitative Physics.* MIT Press, 1992.

[Gra69] R. I. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, 17(2):416–429, March 1969.

[GW90] Jörg Grabner and Reinhold Weiss. A Hierarchical Petri Net Model Used as a Knowledge Source in Distributed Expert Systems. In *Proceedings of the International Symposium on Computational Intelligence*, Milano, Italy, 1990.

[Ham91] Walter C. Hamscher. Modeling digital circuits for troubleshooting. *Artificial Intelligence*, 51:223–271, 1991.

[Hig94] Tesuya Higuchi et al. The IMX2 Parallel Associative Processor for AI. *IEEE Computer*, 27(11):53–63, November 1994.

[Hin94] Peter Hinterberger. QSim Tracedaten Filter. Technical report, Institute for Technical Informatics, Graz University of Technology, 1994.

[HP95] Robert Hranitzky and Marco Platzner. Design and Implementation of Adaptive Digital Filters on a Multi–TMS320C40 System. In *Proceedings of the International Conference on Signal Processing Applications & Technology*, pages 526–530, Boston, USA, October 1995.

[Hra94] Robert Hranitzky. Entwurf und Implementierung adaptiver Filter auf dem Multi–DSP System PPDS. Master's thesis, Institute for Technical Informatics, Graz University of Technology, February 1994.

[HW95] Robert Hranitzky and Reinhold Weiss. Simulation von drahtloser ungerichteter Infrarot-Nachrichtenübertragung auf Multi–DSP–Architekturen. In *Proceedings of DSP Deutschland '95*, pages 261–269, Munich, Germany, September 1995.

[Hwa93] Kai Hwang. *Advanced Computer Architecture: Parallelism, Scalability, Programmability*. McGraw-Hill, 1993.

[IK77] Oscar H. Ibarra and Chul E. Kim. Heuristic Algorithms for Scheduling Independent Tasks on Nonidentical Processors. *Journal of the Association for Computing Machinery*, 24(2):280–289, April 1977.

[Kas90] Simon Kasif. On the Parallel Complexity of Discrete Relaxation in Constraint Satisfaction Networks. *Artificial Intelligence*, 45:275–286, 1990.

[Kau96] Diethard Kaufmann. Parallele Implementierung des qualitativen Simulators QSIM auf einer Multi–DSP–Architektur. Master's thesis, Institute for Technical Informatics, Graz University of Technology, March 1996.

[Kay92] Herbert Kay. A qualitative model of the space shuttle reaction control system. Technical Report AI92-188, Artificial Intelligence Laboratory, University of Texas, September 1992.

[KCMT91] Benjamin J. Kuipers, Charles Chiu, David T. Dalle Molle, and D. R. Throop. Higher-order derivative constraints in qualitative simulation. *Artificial Intelligence*, 51:343–379, 1991.

[KK93] Herbert Kay and Benjamin Kuipers. Numerical Behavior Envelopes for Qualitative Models. In *Proceedings of the Eleventh National Conference on Artificial Intelligence (AAAI-93)*, pages 606–613, Cambridge, MA, 1993. AAAI/MIT Press.

[KM93] Mark A. Kramer and Richard S. H. Mah. Model-based monitoring. In *Second Conference on Foundations of Computer Aided Process Operations*, pages 1–24, Crested Butte, CO, 1993.

[KR87] Vipin Kumar and Nageshwara Rao. Parallel Depth-First Search, Part II: Analysis. *International Journal on Parallel Programming*, 16(6):501–519, 1987.

[Kui84] Benjamin Kuipers. Commonsense Reasoning about Causality: Deriving Behavior from Structure. In Daniel G. Bobrow, editor, *Qualitative Reasoning about Physical Systems*, pages 169–203. Elsevier Sience Publishers B.V., Amsterdam, Netherlands, 1984.

[Kui86] Benjamin Kuipers. Qualitative Simulation. *Artificial Intelligence*, 29:289–338, 1986.

[Kui88] Benjamin J. Kuipers. Qualitative simulation using time-scale abstraction. *International Journal Artificial Intelligence in Engineering*, 3(4):185–191, 1988.

[Kui93a] Benjamin J. Kuipers. Qualitative simulation: then and now. *Artificial Intelligence*, 59:133–140, 1993.

[Kui93b] Benjamin J. Kuipers. Reasoning with qualitative models. *Artificial Intelligence*, 59:125–132, 1993.

[Kui94] Benjamin Kuipers. *Qualitative Reasoning: Modeling and Simulation with Incomplete Knowledge*. Artificial Intelligence. MIT Press, 1994.

[KV94] Hermann Kopetz and Paulo Verissimo. Real Time and Dependability Concepts. In Sape Mullender, editor, *Distributed Systems*, chapter 16. Addison-Wesley, 1994.

[KvB95] Grzegorz Kondrak and Peter van Beck. A theoretical evaluation of selected backtracking algorithms. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence*, pages 541–547, Montreal, Quebec, August 1995.

[LHB94] 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.

[LN93] Franz Lackinger and Wolfgang Nejdl. Diamon: A Model-Based Troubleshooter Based on Qualitative Reasoning. *IEEE Expert*, 8(1):33–40, February 1993.

[LY95a] Wei-Ming Lin and Bo Yang. Fast Parallel Tree Search with Static Load-Balancing Forward Checking Technique. In *Proceedings of the ISCA International Conference on Parallel and Distributed Computing Systems*, pages 33–38, Orlando, Florida U.S.A., September 1995.

[LY95b] Wei-Ming Lin and Bo Yang. Probablistic Performance Analysis for Parallel Search Techniques. *International Journal of Parallel Programming*, 1995.

[Mac77] Alan K. Mackworth. Consistency in Networks of Relations. *Artificial Intelligence*, 8:99–118, 1977.

[Mac92] Alan K. Mackworth. Constraint Satisfaction. In Stuart C. Shapiro, editor, *Encyclopedia of Artificial Intelligence*, volume 1, pages 285–293. John Wiley & Sons, Inc., 1992.

[Mat95] Hideaki Matsumoto et al. Real-Time Simulator For Power System Using a Multi-DSP System (TMS320C40 x 32). In *Proceedings of the International Conference on Signal Processing Applications & Technology*, pages 1338–1342, Boston, USA, 1995.

[MF85] Alan K. Mackworth and Eugene C. Freuder. The Complexity of Some Polynomial Network Consistency Algorithms for Constraint Satisfaction Problems. *Artificial Intelligence*, 25:65–74, 1985.

[MH86] Roger Mohr and Thomas C. Henderson. Arc and Path Consistency Revisited. *Artificial Intelligence*, 28:225–233, 1986.

[MLL92] Dan Moldovan, Wing Lee, and Changhwa Lin. SNAP: A Marker-Propagation Architecture for Knowledge Processing. *IEEE Transactions on Parallel and Distributed Systems*, 3(4):397–410, July 1992.

[MLLC92] Dan Moldovan, Wing Lee, Changhwa Lin, and Minhwa Chung. SNAP Parallel Processing Applied to AI. *IEEE Computer*, 25(5):39–49, May 1992.

[Mol93] Dan I. Moldovan. *Parallel Processing: From Applications to Systems*. Morgan Kaufmann Publishers, 1993.

[Mus95] David J. Musliner et al. The Challanges of Real-Time AI. *IEEE Computer*, 28(1):58–66, January 1995.

[NA91] D. Nussbaum and A. Agarwal. Scalability of Parallel Machines. *Communications of the ACM*, 34(3):57–61, 1991.

[Ng91] Hwee Tou Ng. Model-Based, Multiple-Fault Diagnosis of Dynamic, Continuous Physical Devices. *IEEE Expert*, 6(6):38–43, December 1991.

[OMT94] Gérald Ouvradou, Aymeric Poulain Maubant, and André Thépaut. Hybrid Systems on a Multi-Grain Parallel Architecture. In H. Kitano, V. Kumar, and C.B. Suttner, editors, *Parallel Processing for Artificial Intelligence 2*, pages 3–10. Elsevier Science B.V., 1994.

[Pam95] Herbert Pamminger. Verteilte Simulation des Mehrwegeempfangs bei Infrarot–Übertragung auf Basis einer Multi–DSP Architektur. Master's thesis, Institute for Technical Informatics, Graz University of Technology, May 1995.

[Pla96] Marco Platzner. *Design, Implementation, and Experimental Evaluation of Coprocessor Architectures for Fast Qualitative Simulation*. PhD thesis, Graz University of Technology, 1996.

[PR95a] Marco Platzner and Bernhard Rinner. High–Performance Qualitative Simulation on a Multi–DSP Architecture. In *Proceedings of the International Conference on Signal Processing Applications & Technology*, pages 725–729, Boston, USA, October 1995.

[PR95b] Marco Platzner and Bernhard Rinner. Improving Performance of the Qualitative Simulator QSIM — Design and Implementation of a Specialized Computer Architecture. In *Proceedings of the ISCA International Conference on Parallel and Distributed Computing Systems*, pages 494–501, Orlando, USA, September 1995.

[Pro93] Patrick Prosser. Hybrid Algorithms for the Constraint Satisfaction Problem. *Computational Intelligence*, 9(3):268–299, 1993.

[PRW95a] Marco Platzner, Bernhard Rinner, and Reinhold Weiss. A Distributed Computer Architecture for Qualitative Simulation Based on a Multi–DSP and FPGAs. In *Proceedings of the 3rd Euromicro Workshop on Parallel and Distributed Processing*, pages 311–318, San Remo, Italy, January 1995. IEEE Computer Society Press.

[PRW95b] Marco Platzner, Bernhard Rinner, and Reinhold Weiss. Exploiting Parallelism in Constraint Satisfaction for Qualitative Simulation. *J.UCS The Journal of Universal Computer Science*, 1(12):811–820, December 1995.

[PRW95c] Marco Platzner, Bernhard Rinner, and Reinhold Weiss. Parallel Qualitative Simulation. In *Proceedings of the 1995 EUROSIM Conference*, pages 231–236, Vienna, Austria, September 1995.

[Pup87] Frank Puppe. *Diagnostisches Problemlösen mit Expertensystemen.* Informatik–Fachberichte 148. Springer Verlag, 1987.

[Rei87] Raymond Reiter. A Theory of Diagnosis from First Principles. *Artificial Intelligence*, 32:57–95, 1987.

[Rie95] Johannes Riedl. Parallele Algorithmen und Laufzeitmessungen für Constraint Satisfaction im qualitativen Simulator QSIM. Master's thesis, Institute for Technical Informatics, Graz University of Technology, March 1995.

[Rin93] Bernhard Rinner. Konzepte zur Parallelisierung des qualitativen Simulators QSIM. Master's thesis, Institute for Technical Informatics, Graz University of Technology, October 1993.

[Rin95] Bernhard Rinner. QSim Kernel Interface. Technical Report TR 95/2, Institute for Technical Informatics, Graz University of Technology, August 1995.

[RK93] V. Nageshwara Rao and Vipin Kumar. On the Efficiency of Parallel Backtracking. *IEEE Transactions on Parallel and Distributed Systems*, 4(4):427–437, April 1993.

[Rob92] Ian N. Robinson. Hardware to Support Runtime Intelligence. *IEEE Computer*, 25(5):63–66, May 1992.

[Sei92] Peter Ewald Seifter. *Eine verteilte Simulationsarchitektur für die Fehlersimulation in technischen Systemen.* PhD thesis, Graz University of Technology, 1992.

[SM95] Siddarth Subramanian and Raymond J. Mooney. Multiple-Fault Diagnosis Using General Qualitative Models with Behavioral Modes. In *International Joint Conference on Artificial Intelligence*, 1995.

[SSNB95] John A. Stankovic, Marco Spuri, Marco Di Natale, and Giorgio C. Buttazzo. Implications of Classical Scheduling Results for Real-Time Systems. *IEEE Computer*, 28(6):16–25, June 1995.

[SSW95] Johann Schinnerl, Christian Steger, and Reinhold Weiss. Design and Implementation of an Interactive Simulation Environment based on C40 Signal Processors. In *Proceedings of the International Conference on Signal Processing Applications & Technology*, pages 1799–1803, Boston, USA, October 1995.

[Sta88] John A. Stankovic. Misconceptions about Real-Time Computing: A Serious Problem for Next-Generation Systems. *IEEE Computer*, 21(10):10–19, October 1988.

[Ste95] Christian Steger. *Entwurf und Implementierung eines verteilten modellbasierten Fehlerdiagnose–Systems für Produktionsanlagen.* PhD thesis, Graz University of Technology, 1995.

[SW94a] Christian Steger and Reinhold Weiss. Distributed Blockoriented Realtime–Simulator with automatic Code–Generation for a Multi-DSP Architecture Based on TMS320C40. In *Proceedings of the European Simulation Multiconference*, pages 165–169, Barcelona, Spain, June 1994.

[SW94b] Christian Steger and Reinhold Weiss. Modelbased Real–Time Fault Diagnosis in Technical Processes. In *Proceedings of the Workshop on Distributed Control '94*, pages 15–20, Prag, May 1994.

[SW95] Christian Steger and Reinhold Weiss. A Model–Based Real–Time Fault Diagnosis System in Technical Processes. In *Proceedings of the 10th International Conference on Applications of Artificial Intelligence in Engineering*, pages 145–152, Udine, Italy, June 1995.

[TM92] Louise Travé-Massuyès. Qualitative reasoning over time: history and current prospects. *The Knowledge Engineering Review*, 7(1):1–18, March 1992.

[TMM95] Louise Travé-Massuyès and Robert Milne. Application oriented qualitative reasoning. *The Knowledge Engineering Review*, 10(2):181–204, 1995.

[VDC95] F. Valentinotti, G. DiCaro, and B. Crespi. A Parallel DSP System for Real-Time Disparity and Optical Flow Using Phase Difference. In *Proceedings of the International Conference on Signal Processing Applications & Technology*, pages 1492–1496, Boston, USA, 1995.

[Ver94] Eric Verhulst. Virtuoso: A virtual single processor programming system for distributed real-time applications. *Microprocessing and Microprogramming*, 40:103–115, 1994.

[Wal75] David Waltz. Understanding line drawings of scenes with shadows. In P.H. Winston, editor, *The Psychology of Computer Vision*, pages 19–91. MacGraw-Hill, 1975.

[WdK90] Daniel S. Weld and Johan de Kleer, editors. *Readings in qualitative reasoning about physical systems.* Morgan Kaufmann, 1990.

[Wer94] Hannes Werthner. *Qualitative Reasoning: Modeling and the Generation of Behavior.* Springer-Verlag, 1994.

[ZM93] Ying Zhang and Alan K. Mackworth. Parallel and Distributed Finate Constraint Satisfaction: Complexity, Algorithms and Experiments. In L. Kanal, V. Kumar, H. Kitano, and C. Suttner, editors, *Parallel Processing for Artificial Intelligence*, chapter 1. Elsevier Sience Publishers B.V., 1993.