Available online at www.sciencedirect.com
SCIENCE DIRECT®

ELSEVIER Electronic Notes in Theoretical Computer Science 124 (2005) 63–80

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

A Compiler for Mapping a Rule-Based Event-Triggered Program to a Hardware Engine

Carsten Albrecht[a] and Andreas C. Döring[b]

[a] *Institute of Computer Engineering, University of Lübeck, D-23538 Lübeck, Germany*

[b] *IBM Zurich Research Laboratory, CH-8804 Rüschlikon, Switzerland*

Abstract

In this paper we describe the RERAL compiler. RERAL is a Rule-based Event-driven Routing Algorithm Language. It is intended for the configuration of a router for regular networks, as found in parallel computers or computer clusters. The language combines predicate-logic-derived functional expressions with Petri-net-based asynchronity. The high performance requirements (a routing decision should take no more than few nanoseconds) imply sophisticated optimization methods in the compiler, in particular, flattening the program hierarchy, unrolling loops and mapping high-level program fragments to available application-specific hardware units.
We also point out a new application area of the concept, namely the management of a memory interface in a system-on-chip for increased bandwidth utilization.

Keywords: High-level Hardware Modelling, Application-Specific Programming Language, Rule-base Compiler

1 Introduction

Many components of computer or embedded systems use a combination of fixed hardware, processing units, and configurable hardware. For the latter a wide variety of configuration methods are known, but most of them require a detailed understanding of the architecture, tool issues and hardware-related aspects such as timing and resource constraints. When the system has to be programmed by experts, the complexity of this task is acceptable. However, a higher abstraction layer is desirable to allow the efficient use of programmable hardware structures by a non-expert and to make the expert more productive.

doi:10.1016/j.entcs.2004.07.015

In this paper we introduce a rule-based language and its tool environment that were created to describe routing algorithms for the configuration of interconnection networks in parallel computers or computer clusters. A second application, which turned out to be of interest more recently, is the medium-term control of memory-interface usage in Systems-On-Chip (SoC). Both applications have in common that

- several aspects of the application domain are combined,
- the inputs are formed by an infinite series of unrelated external events (such as the arrival of a message or a cache miss in an on-chip CPU), and
- there is some freedom in the reaction of the programmable component.

All these properties are reflected in the language RERAL (Rule-based Event-driven Routing-Algorithm Language). As the name suggests, it combines the aspects of event-triggered parallel evaluation with the descriptive power of rule-based expressions. Through the use of a compilation tool, a program can be transformed in such a way that it can be applied to a VLSI-implementable rule-evaluation engine which performs a complex algorithmic operation in few clock cycles. In particular, a hierarchically described routing decision is flattened such that it can be stored in a small on-chip memory, and individual evaluation requires only one access to this memory thanks to sophisticated address generation.

In this paper we present first the background for the two application areas (Section 3), with emphasis on routing algorithms. In particular we motivate the modular characteristic of the language and the necessary aspects of parallelism and functional complexity. The target architecture is only sketched so as to leave more room for the introduction of the language (Section 4) and the discussion of the compilation process (Section 5). Specifically, we demonstrate how a unification-based method can be used to extract specific functionalities for hardware building blocks in the rule-evaluation engine. If this functionality is spread over several rules in the user program, it has to be detected by the compiler to maintain hardware independence.

2 Related Work

The related disciplines are wide-spread; rule-based systems are typically used in the software world. However, to describe a specialized hardware system, only low-level descriptions with similarity to rule bases are in use. We have been told that the tool 'specializer' as described in [15] is able to perform many of the tasks the compiler presented here does. We did not verify this, like to point out that our compiler performs numerous hardware-specific operations that probably are not covered by a general-purpose tool. An example is the generation of the addressing logic for higher-dimensional arrays without multiplication [9].

Generating the scanner and parser from a given grammar automatically is a well-established technique, and we used Eli [13] for this purpose. For the implementation of the compiler we used the interpreter Hugs for the functional programming language Haskell [16]. The pattern matching features of the language, the rich set of functions for dealing with complex data structures including the automated memory handling provide a framework in which the required transformations in the compiler can be described on a high abstraction level.

The proprietary description language of state machines used in the software suite Log/iC has some similarity with very basic rules. Neonetworks announced a chip called StreamProcessor for networking applications that was claimed to exploit parallelism on a "supercomputer scale" and had a building block called "rule processor". However, information on this technology has been confidential, and the company has since gone out of business.

With respect to routing in parallel computer networks, Summerville et al. present an architecture for a bit-pattern-associative router in [18]. They describe their routing algorithms in a pseudo-language that is very similar to the basic pattern used in RERAL. The target hardware uses a pattern-matching circuit array which is similar to a ternary CAM (Content addressable Memory). As there are neither dedicated arithmetic circuits nor other specialized components in the proposed routing engine, only simple routing algorithms can be carried out without a huge increase in the association circuitry.

In the domain of Internet Protocol (IP)-based networks, rules for routing are popular because of the hierarchical organization of IP addressing. Consequently, there are many architectures that process rules, that combine range checks and prefix matching in IP addresses, together with range checks on the port number. An example is [19]. Because a parallel check is performed, the hardware effort scales with the number of permitted rules. Memory-oriented variants are also in use, e.g. [17]. IP-based routing requires the option to change the rule set dynamically, thus the mapping of the rule set defined on a high abstraction level to the representation in the hardware has to be computable very efficiently.

All these methods impose strong limitations on the structure of the rules, in particular they restrict the type of operations that can be applied to the variant inputs. Furthermore, only one rule set is considered for the reaction to one type of event. The reactions of the rules are very simple, such as a drop/non-drop decision in a fire wall. These limitations inhibit the implementation of advanced algorithms like the ones that will be introduced in the next section.

3 Area of Application

As pointed out in Section 1, we are considering two areas of applications: the routing in regular networks and the management of memory-access bandwidth in a SoC, such as a network processor.

Regular networks use a topology with a high regularity, e.g. a mesh or a hypercube. They are an important part of parallel computers, in particular PC clusters. The regularity allows the use of advanced routing algorithms that

- allow scaling the network size, with nearly constant hardware cost in the routers,
- adapt the path for a data item through a network dynamically, depending on link or router load (adaptivity), and
- route around failed routers or links dynamically (fault tolerance).

A good coverage of these routing algorithms and the basic architecture for routers for this class of networks is given in [10]. For our purpose it is sufficient to know that the network consists of links and routers (Figure 1) and that the data injected at the routers is transported in messages to other routers via the links until these messages reach a destination where they leave the network. The routers are identified by an address, and the messages have a header containing the destination router's address. There is a protocol, called link-level flow control, that ensures lossless transmission of data from one router to the next.

Routing algorithms react to several distinct types of external events: the arrival of a new message, the notification by the link-level flow control of a changed load situation on an attached link or neighboring router, the notification of the failure of a network component, or the completion of the transmission of a message, which releases a resource for the next message.

Routing algorithms typically cover several more or less independent aspects. Two aspects, deadlock and livelock avoidance, ensure that a message is transported to its destination in finite time. While a deadlock results in an infinitely long waiting time of messages within the routers because their further routing depends cyclically on each other, a livelock situation occurs when one or several messages move continuously through the network without ever reaching their target. For this reason, livelock avoidance specifically includes the knowledge of the network topology, in particular the capability

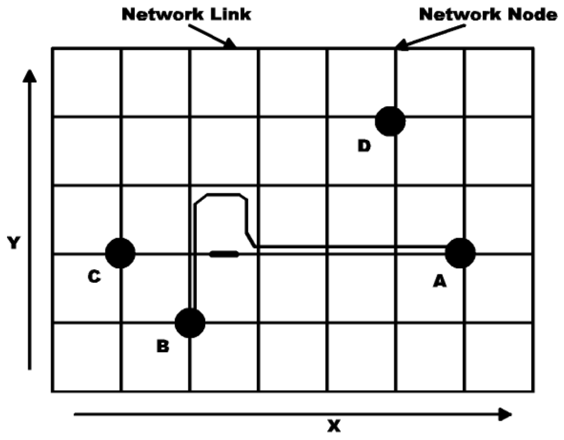

Fig. 1. A mesh network, the topology for NARA.

of the routers to determine a path to the destination for a given router address. Moreover, routing algorithms contain a local scheduling that decides which message is preferred in a resource conflict such as link usage. Two other aspects of importance are related to the avoidance of overloaded and broken routers or links, if this is possible. They are a combination of the collection and distribution process of relevant information and the application when routing an individual message.

The routing algorithm NARA (New Adaptive Routing Algorithm, [5]) is used as an example. It is intended for two-dimensional meshes as the one shown in Figure 1 where the address of a router is a coordinate pair (x, y) in an integer-addressed rectangle. To avoid livelocks, paths of minimal length are preferred, i.e. a message is routed such that it gets closer to the destination if this is possible.

Deadlock avoidance is based on the so-called *turn model* [14]. NARA distinguishes messages according to the difference in the Y-direction such that all messages whose destination has a smaller Y-coordinate than the source (for example a message sent from A to B) are handled separately from those with increasing Y-coordinates, e.g. C to D. For those messages with increasing Y-coordinates, the restriction is imposed that the message may not change its direction after it has used a link with increasing Y-coordinate. For the other messages, the same restriction applies symmetrically. A path of a message for this case is shown in the figure, the only downward part leads directly to the destination node. Furthermore, NARA applies an age-based local scheduling strategy that counts the events if a message loses arbitration to another message when competing for a free link. Finally, NARA contains a method for adaptivity that consists of summing up the buffer fill levels in a router and

```
ON in_message(indir,vc,dx,dy) -- a message has arrived
-- if all channels are in use
 IF FORALL j IN deadlock_free(indir,vc,dx,dy):  out_chan(j,vc)<>free
 THEN out_set(indir,vc)<-deadlock_free(indir,vc,dx,dy);
-- if any of the minimal paths is free
 IF EXISTS i IN minimal(dx,dy):  out_chan(i,vc)=free AND
    (FORALL j IN minimal(dx,dy):  out_chan(j,vc)=free AND
    mean_queue(vc,i)<=mean_queue(vc,j))
 THEN !send(vc,vc,indir,i),
    out_queue(vc,i)<-message_length(vc,indir);
-- if some channels are free but not the minimal ones
 IF EXISTS i IN deadlock_free(indir,vc,dx,dy)\minimal(dx,dy):
    out_chan(i,vc)=free AND
    (FORALL j IN deadlock_free(indir,vc,dx,dy)\minimal(dx,dy):
       (mean_queue(vc,i)>=mean_queue(vc,j)))
 THEN !send(vc,vc,indir,i),
    out_queue(vc,i)<-message_length(vc,indir);
END in_message
```

Fig. 2. RERAL implementation of the main rule base of NARA, `deadlock_free` and `minimal` are sub-bases.

transporting this information to neighboring routers. If a message arrives, it is checked whether any of the allowed links is available with respect to deadlock avoidance.

Clearly, this algorithm is complex and its description is preferably done in a high-level language. This language should allow the use of basic data types (integers, Booleans, and arrays of them), with appropriate operations (addition, comparisons, logical operations, etc.), and, most importantly, also allows individual aspects such as deadlock-avoidance algorithm to be described separately from adaptivity or the topological properties of the network. Aspects such as variables, and the notion of events ("whenever a message arrives, do the following") are also crucial. Finally, the high performance requirements must be reflected in the language allowing a high parallelism. Most of the events can be handled concurrently, but at certain points an atomic behavior needs to be guaranteed, for instance when a shared resource is assigned, e.g. a link to a message.

Furthermore, the hardware for executing the routing algorithm has to provide the resources for storing a structured state (variables, arrays, etc.) and for performing the algorithm on arriving events accordingly. The algorithmic step involves the selection of the appropriate rules, including evaluating arithmetic expressions on the parameters (for instance the destination node address). The result is the modification of state variables, the generation of commands to the data-transport part of the router, and in some cases the generation of new events for further actions. To achieve the performance goals, we have developed an architecture for a routing engine that combines configurable, problem-specific components for arithmetic expressions in premises and conclusions of the rules. An example is the circuit for fault tolerance based on finite ordered sets [8]. The logical skeleton of the rules is mapped to a look-up table. To keep the table compact, we have developed a set of compression methods that exploit regularity and symmetry in the algorithmic structure. Some of these methods can profit from ambiguities in the algorithm; in NARA for example such an undefined situation exists for messages that stay on the same Y-coordinate level (e.g. A to C): It does not matter into which class of messages they are assigned by an implementation.

A second area for applying the hardware-based evaluation of rule bases is the management of a memory interface in a SoC [12], such as a network processor. In a structure such as that of [11], several components (network interfaces, processor cores, coprocessors, extension interfaces) share a common memory interface. Because of the high pin number required for interfacing memories, bandwidth on this interface is typically a valuable resource. Therefore, optimizing its use can help to build a cheaper system or to achieve better performance at the same cost. However, the components have very different access characteristics; compare, for instance, the sporadic memory-access pattern with a fixed line size from a data cache and the fixed pattern of a tree search engine for IP header classification. Some of the accesses have a temporal elasticity, for instance the flushing of dirty data cache lines can sometimes be done in advance, i.e. before the cache line is reused and flushing is enforced. Another example is a network interface that typically contains a buffer. Depending on the current and future pressure on the memory interface, the point in time for data transport between this on-chip memory and the off-chip memory through the interface in question can be varied. The management algorithm for this task has to take the performance goals of the application and the utilization of the various components (processors, coprocessors) into account. The reaction in a given over- or underload situation has to be translated into the reactive capabilities of the individual requester components without degrading the situation for upcoming cycles. Because of the similarity of this management problem with the routing algorithms, we believe that the concept of a rule-based language, combined with the optimizing compiler and an application-tailored rule-evaluation engine, can also be applied. Of course, intensive studies including system simulations will have to be carried out to prove this.

4 RERAL

The language itself and its usage to define routing algorithms are given in [6]. The central building block of the language is a rule that consists of a condition (premise) and a list of commands (conclusion). A set of rules forms a rule base or a subbase. In general, subbases are functions returning a value to the caller.

Exploiting side effects, they become a powerful way of describing subroutines and shaping the code clearly. In contrast to traditional programming languages such as C/C++ and MODULA-2, all rules of a rule base are executed in parallel. The conditions are evaluated with respect to the global state at a rule-base call, and the commands belonging to the conclusions executed alter the global state also in parallel.

Here, we briefly state the main syntactical components of RERAL, using a routing-algorithm implementation as example.

Constant Definition

CONSTANT *LinkIndex :=* 0..5*;*

Constants are declared by finite sets of numbers, symbols or constant sets. Here, the constant *LinkIndex* consists of six numbers $\{0, 1, 2, 3, 4, 5\}$. These sets are used like types and constitute a high abstraction of hardware details.

Variable Definition

VARIABLE *Linkload(LinkIndex) IN* 0..63*;*

The variable *LinkLoad* is an array of numbers where each number is in the range of 0..63. Its size and indices are given by the constant set in parentheses. All variables have a finite (usually small) domain that is given by a set literal or a constant.

Rule-Base Declaration

DEFER SUBBASE *TorusMinimalXDim(Xdest, Ydest)*

This kind of declaration predefines a subbase that has two parameters. Subbases are one form of rule bases. They are the main structuring element and either define a function or work as subactions having side effects. The declaration is not necessary, but improves readability.

Subbase Definition

SUBBASE *opp(vc) ...* **END** *opp*

This declaration embeds the subbase. The dots replace a set of independent rules whose notion contains a high degree of parallelism. Every relevant case has to be covered by at least one rule (completeness).

Triggered Rule-Base Definition

ON *In_message(Dir , Chan) ...* **END** *In_message*

This rule base has to be executed exactly once for each occurring event bound to it. As rule bases describe only functions, time and sequence are expressed separately by the notion of events.

Rule

IF *vc=south* **THEN** *opp* $\leftarrow$ *north;*

This construct is the core idea of the rulebased language. It can be viewed as a guarded command. One rule represents one case of the algorithm. A rule base is some kind of case distinction where every rule covers at least one case. It is expressed by a predicate logic expression. In this case the action is the presentation of the function result. In addition to Boolean operators and arithmetic expressions subbases can be used.

Quantifier Expressions in Premise

EXISTS p **IN** $\{0, 1, 2, 3\}$*: OutChannelUsage(dir , p) = FREE*
FORALL p **IN** $\{0, 1, 2, 3\}$*: OutChannelUsage(dir , p) = FREE*

Both expressions are a sort of loop used in conventional languages but they avoid the sequentiality of conventional loops. The *EXISTS* expression is a short form for as many rules as the number of elements of the finite set. Here, the variable p may be reused in the conclusion. In contrast to the *EXISTS* expression, the *FORALL* expression is not a short form for multiple rules but for a sequence of *AND* expressions in the same premise. This variable cannot be reused in the conclusion but it is possible to establish the same loop in the conclusion as well, see below. Both expression introduce some kind of local name space.

Event Generation (Conclusion)

!*InternalSwitch(fromDir, fromChan, SOUTH, DetNetChan);*

Asynchronous control of the hardware is accomplished by generating an event. Events can also be used to cascade several rule interpretations to generate a final result.

Variable Assignment (Conclusion)

OutLinkUsage(ToDir) $\longleftarrow$ *OutLinkUsage(ToDir) -1*

Rule execution is atomic, i.e. if a variable is checked in the premise and changed in the conclusion, parallel execution of two rule bases has to be performed on the same system state.

FORALL-Quantifier in Conclusion

FORALL j **IN** *all_directions: !send_info(info,j,total_load)*

The conclusion can contain several commands that are executed concurrently. The same applies for “loops” which can be nested.

Overall the language eliminates as many sequential dependencies as possible. Note that the application of a subbase in a premise does not imply that the hierarchically lower subbase has to be executed before the main one. In contrast, the hierarchical structure is only a form of expression for one larger subbase that is evaluated in one piece. The subbase hierarchy allows abstraction of hardware details. The reading access of a variable cannot be distinguished syntactically from the application of a subbase. This allows the introduction of caching techniques (eliminating subbase calls) and the replacement of status arrays by methods calculating the original expected value from other sources. Hence, a higher interface can be defined whose implementation on the actual hardware can again be done using rule bases.

5 Compilation

The compilation process of RERAL programs simplifies all rule bases. There are several transformations, which can be done in an arbitrary sequence. The goal is a rule base for each event, which contains a minimal set of rules. The rules' premises and conclusions should be flat expressions (conjunctions of simple comparisons) and lists of simple commands. Only for those functions where a direct hardware implementation is available (e.g. supremum in a finite ordered set), should the corresponding identifier be found. One transformation aims at retrieving a minimal number of rule bases by replacing function calls by their subbase. Another transformation unrolls loops. They exist in premises using forall or exist quantifiers and in conclusions using a standard forall statement to loop over all elements of a finite set. Thirdly, all premises are searched for run-time-independent elements that are evaluated and the premises are shaped based on the results. Frequently the premise is reduced to a Boolean constant so that rules with false premises are dropped. All premise terms are collected in a table and replaced by a label to avoid multiple run-time computations.

The detailed processing can be approximated by the following description:

- Replacement of Constants
  The constant definitions are checked for interdependencies to detect illegal ring definitions, and ordered by them to minimize the number of replacements. At the end of this transformation, all constant values substitute their symbols. This process is especially important for subsequent premise evaluations.

- Solving Quantifiers
  All FORALL (Symbol: $\forall$) quantifiers used in premises are solved by the following equivalence:

  $\mathcal{M}$ is a finite set.
  $p : \mathcal{M} \to \mathcal{B}$ is a predicate.

  $$\boxed{\forall i \in \mathcal{M} : p(i) \qquad \Longleftrightarrow \qquad \bigwedge_{i \in \mathcal{M}} p(i)}$$

  The replacement of the EXISTS (Symbol: $\exists$) quantifier is more difficult. The conclusion can use the variable used by $\exists$ so that each element of $\mathcal{M}$ requires its own rule.

$\mathcal{M}$ is a finite set.
$p : \mathcal{M} \to \mathcal{B}$ is a predicate.
c is a parameterized conclusion.

$$\boxed{\begin{array}{lcl} \text{IF } \exists i \in \mathcal{M} : p(i) & \Longleftrightarrow & \forall i \in \mathcal{M} : \\ \text{THEN } c(i) & & \text{IF } p(i) \text{ THEN } c(i); \end{array}}$$

These replacements often allow the reduction of rule premises at compile time. If $p(i)$ includes further quantifiers, they are solved beginning with the innermost one.

- Flattening Hierarchies
Here, the modular structure of RERAL programs is decomposed by inserting subbases inline. The result is a distinctly grown rule base. Assuming rule base A has l rules and includes k calls of subbase B, A can grow to a size of $l^k - 1$ rules; if the subbase calls are spread over k rules, the rule base only grows to a size of $l * k - k$ rules.

  The order of inserting subbases has a high impact on the efficiency. Suppose that rule base A calls the subbases B and C and B also calls C, this can be processed in the following ways:
  · one inserts B into A, gets $\hat{A}$ as an intermediate result and inserts C into $\hat{A}$, or
  · one inserts C into B and A, gets $\hat{B}$ and $\hat{A}$, and inserts $\hat{B}$ instead of B into $\hat{A}$.
  Unfortunately, it is not possible to find an efficient way by syntactical analysis. Semantical aspects and dependencies decide this issue.

- Reductions
Because of the increasing number of rules per rule base due to preceding steps, reduction functions are welcome. Later on, each distinct premise term requires a hardware resource, such as a comparator, and each rule requires a table entry. Both are limited in the routing engine. Because flattening hierarchies and solving quantifiers can produce multiple copies of a single rule in the same rule base, multiple occurrences are removed. In particular, unfolded quantifiers allow reductions by evaluating comparisons of constants. Consequently, and considering that the Boolean operator $\wedge$ (*and*) appears more frequently than $\vee$ (*or*), many rules can be skipped because of fully evaluated false premises. Furthermore, arithmetic expressions are normalized by sorting variables and arranging them on one side in an inequality. These reduction steps are repeated whenever new rules are produced to avoid an explosion of the rule base.

6 Optimization

The expansion of rule bases and subbases produces an exponentially growing number of rules. Reducing and controlling all these rules is rather difficult for the compiler and in certain cases impossible because of missing run-time information considering e.g. dynamic sets. Taking into account that each rule requires chip space, the absolute number of rules is limited. Algebraic optimizations such as the evaluation of algebraic terms, e.g. comparisons, or the removal of multiple copies have yet been mentioned. Another idea is to find structures such as inline-inserted subbases or very small functions that are replaced by function calls whose hardware implementation is more space-efficient than that of the rule base. Candidates are functions working on huge sets because the required space depends on the size of these sets. When traditional methods are used to implement them, some only consume a fixed amount of space. In this case, the fixed-sized function typically outperforms the rule-based one. To define these structures, search the rule bases, and replace the occurrences by function calls, a substitution pattern is specified and each rule is transformed into a first-order logic representation and searched by a unification algorithm.

6.1 Unification

In general, unification tries to identify two symbolic expressions by replacing sub-expressions by other expressions. Assuming, for example, that f is a function symbol, a, and b are constants, and x and y are variables, the unification problem of the terms $f(a, x)$ and $f(y, b)$ is solved by the substitutions x/b and y/a, where e.g. x/b means that the right element b substitutes the left one x. The result of this example is $\{x/b, y/a\} \circ f(a, x) = \{x/b, y/a\} \circ f(y, b) = f(a, b)$. Here, applied to the language of first-order logic, the unification is a syntactic one. Baader and Snyder [3] give introduction to syntactic unification and also present Robinson's unification algorithm. It decides whether a set of terms is unifiable and, in the case of a positive decision, returns the most general unifier, i.e. a set of substitutions that constrains the functions less than all other possible unifiers do. This is often applied in automatic theorem provers.

6.2 Pattern Matching

To shape the performance of rule bases, a library could provide the programmer with performance-optimized subbases. For each library function, a substitution pattern that describes the high-level structure and function and its high-level substitute must be defined. To track these occurrences in the high-level program representation, a pattern and each rule of the rule base searched

```
IF EXISTS A IN Set:
   [(p(A))
   AND (FORALL B IN Set:                                 IF EXISTS A IN Set:  [p(A)]
     [(p(B))                               ⟹            THEN c(selectminimal(Set,p(),v()));
     AND ((v(A)) <= (v(B)))])]
THEN c(A);
```

Fig. 3. Exemplary pattern specification.

are checked by Robinson's unification algorithm. Figure 3 demonstrates an exemplary pattern. Its specification language is a mixture of first-order terms and RERAL syntax. The left-hand side defines the rule pattern searched in the rule and the right-hand side is the rule pattern that substitutes the matched rules. Identifiers beginning with a small letter represent functions with at least one argument, an initial capital letter denotes variables, and keywords are written in capital letters. The premise of the exemplary rule pattern in Figure 3 contains a minimum function whose result is used in the conclusion by all commands: "If there exists an element in the set that is smaller than or equal to all others then process it." The pattern premises resemble first-order terms; only comparison operators are predefined functions because of their high frequency. All other functions can be determined by general function symbols without any semantics known by the system. The representation of a conclusion allows two symbols: a variable and a function with arguments defined in the premise. Also a combination of the two is possible. The variable can match any sequence of commands, even an empty set. Functions must match at least one command that depends on the argument specified. In Figure 3, the function `p()` is an additional constraint for all elements that must be satisfied; `v()` is a kind of weight function. Each occurrence of a rule containing this pattern is replaced by the rule to the left by removing the innermost loop of the premise and introducing the library-function call `selectminimal`.

An assumption to apply first-order unification is that the specifications of both inputs are first-order terms. Each rule and, by analogy, the pattern are transformed into a kind of prefix notation; even the rule itself is a binary function with two arguments: premise and conclusion. The transformation result of the example of Figure 3 is shown in Figure 4. As the algorithm does not match higher-order terms, functions without a determined semantics are replaced by a variable so that the transformed pattern is more general. The original meaning of the pattern is restored by a list of constraints that contains a constraint for each generalization. Another example of generalization and constraints is the multiple frequencies of the same function, in which each occurrence of a function is replaced by its own variable. All substitutes of the variables must be the same function (constraint). The constraints needed to restore the semantic of a pattern are itemized below.

```
(F "RULE" [
   F "PREMISE" [
     F "EXISTSQ" [ V "A", V "Set",
       F "AND" [ V "p", F "FORALLQ" [ V "B", V "Set",
                          F "AND" [ V "p0",
                            F "LESSEQ" [V "v",V "v0"]]]]]],
F "CONCLUSION" [V "c"]],


F "RULE" [
  F "PREMISE" [
    F "EXISTSQ" [V "A",V "Set",V "p"]],
  F "CONCLUSION" [
    F "FUNCTION" [V "c",
                  F "FUNCTION" [V "selectminimal", V "Set",
                    F "FUNCTION" [ V "p0" ], F "FUNCTION" [V "v"]]]]]
)
```

Fig. 4. Transformed exemplary pattern.

```
SET [ F_PARAM_EQUAL 1 [V "c"],                       [TP (V "A",
      F_ALL_ELEM [V "A"] (V "p"),                         F "FUNCTION"
      F_ALL_ELEM [V "B"] (V "p0"),                          [ V "selectminimal",
      F_EQUAL (V "p") (V "p0"),                               V "Set",
      F_ALL_ELEM [V "A"] (V "v"),                             F "FUNCTION" [V "p0"],
      F_ALL_ELEM [V "B"] (V "v0"),                            F "FUNCTION" [V "v"]])]
      F_EQUAL (V "v") (V "v0"),
      C_ALL_ELEM [V "A"] [V "c"]],
```

Fig. 5. Constraints (left) and transpositions (right) of the exemplary pattern.

- `F_PARAM_EQUAL Int [Function]`
  All functions of the list must have the same number of arguments. This constraint is used to ensure equal length of sequences assigned to variables.
- `F_ALL_ELEM [Function] Function`
  All elements of the list must be an argument of the function.
- `F_EQUAL Function Function`
  In addition to their arguments both functions must be equal.
- `C_ALL_ELEM [Function] [Function]` / `C_ANY_ELEM [Function] [Function]`
  All variables of the first list must be bound by all elements/at least one element of the second list.

Figure 5 shows the list of constraints for the pattern of Figure 3 and its transformation, shown in Figure 4. The transpositions are derived from the target rule and are necessary to build the target rule. They must be applied to the pattern before the most general unifier is employed.

By processing the second rule of NARA, see Figure 2, using the pattern of Figure 3, the algorithm succeeds in computing the most general unifier, see Figure 6. This is attached to the transpositions gained by transformation and then all substitutions are executed. The final result, a rule including a library-function call because of a matched pattern, is shown in Figure 7.

```
[
TP (V "A",V "i"),
TP (V "Set", F "FUNCTION" [ V "minimal",
                           F "SYMB" [ V "dx" ],
                           F "SYMB" [ V "dy" ] ]),
TP (V "p", F "EQUAL" [ F "FUNCTION" [ V "out_chan",
                                     F "SYMB" [ V "i" ],
                                     F "SYMB" [ V "vc" ] ],
                                     F "SYMB" [ V "free" ] ]),
TP (V "B", V "j"),
TP (V "p0", F "EQUAL" [ F "FUNCTION" [ V "out_chan",
                                      F "SYMB" [ V "j" ],
                                      F "SYMB" [ V "vc" ] ],
                        F "SYMB" [ V "free" ] ]),
TP (V "v", F "FUNCTION" [ V "mean_queue",
                         F "SYMB" [ V "vc" ],
                         F "SYMB" [ V "i" ] ]),
TP (V "v0", F "FUNCTION" [ V "mean_queue",
                          F "SYMB" [ V "vc" ],
                          F "SYMB" [ V "j" ] ])
]
```

Fig. 6. Most general unifier of the exemplary pattern applied to the second rule of NARA.

```
IF EXISTS i IN minimal(dx, dy):  out_chan(i, vc) == free
THEN !send(vc, vc, indir,
           selectminimal(minimal(dx, dy), equal(), mean_queue())),
           out_queue(vc, selectminimal( minimal(dx, dy),
                                        equal(),
                                        mean_queue()))
                    ← message_length(vc, indir);
```

Fig. 7. Result of substitution for the exemplary pattern applied to the second rule of NARA.

7 Results

The compilation process and the generated implementation of the rule-based hardware specification are usually greedy for resources. Hence, especially memory size on processing-system side, and available logic gates and timing constraints on target-system side limit number and complexity of processable rules. The usage of functions that conserve utilization of resources provides room to implement more rules. Therefore, the number of rule bases that can be performed by the routing engine increases.

The application of the pattern of Figure 3, which selects the minimum of a set deploying a constraint and weight function, to NARA, see Section 3, delivers some numbers that support the benefit of our approach. Our goal was the decrease in the number of rules and the shortening of the premise to shrink the tables of the routing engine. Each comparison of a premise requires an arithmetic circuit and contributes to the address length for the table access. The expansion of the second rule of NARA delivers 117 rules with 7 comparisons per premise. Making use of the unification-based optimization, only 13 rules with 3 comparisons per premise remain. Here, a reduction to a tenth of the normal number of rules and halve the number of comparisons per premise is achieved. Unfortunately, the unification-based optimization has a high mismatch ratio because of the commutativity of several functions. Because exchanged operands of a binary, commutative operation for example can lead to mismatches, the operands were sorted by length to decrease the number of mismatches.

The compiler presented breaks a hierarchically described algorithm down into few tables, one table per event-triggered rule base. The table of the example rule base of NARA requires about 1 kByte.

Moreover, the prototype of the rule-evaluation engine, implemented on XILINX 4000 FPGAs, achieves a clock frequency of 50 MHz. Assuming that a standard-cell ASIC implementation in current technology would run at about 500 MHz a custom implementation could achieve clock frequencies of state-of-the-art microprocessors. The system reaction on a single event has a very low latency (60ns) because the prototype only consumes three clock cycles per decision. Applied to routing, this value would even satisfy the requirements of a network for state-of-the-art blade servers, or high-end clusters.

8 Conclusion

Heterogeneous configurable hardware units are comparatively new, and therefore compiler construction in this area poses new challenges. The experimental compiler presented here combines many different techniques and proves that a high-level abstract language can be used to achieve a very high performance. Combined with the fast reaction to external events and the compiler-enabled high abstraction level, an execution model for problems with extremely high real-time requirements and limited hardware resources is available.

It is remarkable that this transformation system breaks a hierarchically described routing algorithm like NARA down into few tables of a small size. In addition, the reduction in size and number of rules relieves the processing system and provides new headroom for larger and more complex rule bases. Besides, if the same methods are applied to a new area, the crucial functions for the application domain have to be identified first by analyzing the target set of algorithms. This allows the direct mapping of complex subproblems to configurable hardware units. The expected types of redundancy can also be identified, which allows the selection of appropriate address-generation methods.

Since the unification-based optimization technique used is not as successful as it should be, further improvements are desirable. As semantical unification has a high complexity and computational effort other sorting functions such as any sort of weight function evaluating and combining different pattern qualities should be tested.

References

[1] Albrecht, Carsten. *Benutzungsmustererkennung zur Einbindung von Hardwareeinheiten in den RERAL-Compiler.* Student's Thesis, Technical Report IAB-75, Institute of Computer Engineering, University of Lübeck, Germany, 2001.

[2] Aho, Alfred V., Ravi Sethi, and Jeffrey D. Ullman. *Compilers - Principles, Techniques and Tools.* Addison-Wesley, 1988.

[3] Baader, Franz and Wayne Snyder. *Unification Theory.* Chapter 8 of Handbook of Automated Reasoning, edited by Alan Robinson and Andrei Voronkov, Elsevier Science Publishers B.V., 1999.

[4] Bird, Richard. *Introduction to Functional Programming using Haskell.* 2nd Edition, Prentice Hall, 1998.

[5] Cunningham, Chris M. and Dimiter Avresky. *Fault-Tolerant Adaptive Routing for Two-Dimensional Meshes.* In Proceedings of the First International Symposium on High-Performance Computer Architecture, IEEE Computer Society, 1995.

[6] Döring, Andreas C., Gunther Lustig, Carsten Albrecht, and Wolfgang Obelöer. *Building a Compiler for an Application-Specific Language.* 1st Scottish Functional Programming Workshop, August 29th - September 1st, 1999, Stirling, UK.

[7] Döring, Andreas C., Wolfgang Obelöer, Gunther Lustig, and Erik Maehle. *A Flexible Approach for a Fault-Tolerant Router.* In Proceedings of the Workshop on Fault-Tolerant Parallel and Distributed Systems, Lecture Notes on Computer Science **1388** (1998), 693 – 713, Springer, Berlin/Heidelberg.

[8] Döring, Andreas C. and Gunther Lustig. *Implementation of Finite Lattices in VLSI for Fault-State Encoding in High-Speed Networks.* In Parallel and Distributed Processing, Lecture Notes in Computer Science **1800** (2000), Springer.

[9] Döring, Andreas C., Gunther Lustig. *Generating Addresses for Multi-dimensional Array Access in FPGA On-chip Memory.* Field-Programmable Logic and Applications FPL 2000, Lecture Notes in Computer Science **1896** (2000), 626 – 635, Springer-Verlag.

[10] Duato, Jose, Sudhakar Yalamanchili, and Lionel Ni. *Interconnection Networks – an Engineering Approach.* Morgan-Kaufmann Publishers, 2002.

[11] Gabrani, Maria, Gero Dittmann, Andreas Döring, Andreas Herkersdorf, Patricia Sagmeister, and Jan van Lunteren. *Design Methodology for a Modular Service-Driven Network Processor Architecture.* Computer Networks **41(5)** (2003), 623 – 640.

[12] Gabrani, Maria, Andreas Döring, Patricia Sagmeister, Peter Buchmann and Andreas Herkersdorf *Optimizing Bandwidth Usage in a Multi-Core Chip.* Patent Application No. CH9-2003-0024. European Patent Office.

[13] Gray, Robert W., Vincent P. Heuring, Steven P. Levi, Anthony M. Sloane, and William M. Waite. *Eli: A Complete Flexible Compiler Construction System.* Communications of the ACM **35(2)** (1992), 121 – 131.

[14] Glass, Christopher J. and Lionel M. Ni. *The Turn Model for Adaptive Routing.* Journal of the ACM **41(5)** (1994), 874 – 902.

[15] Helsen, Simon. *Region-based Program Specialization.* PhD Thesis, Albert-Ludwigs-Universität Freiburg, Germany, 2002.

[16] Peyton Jones, Simon, John Hughes et al. *Report on the Programming Language Haskell 98.* Technical Report, 1999, URL: http://www.haskell.org.

[17] van Lunteren, Jan and Ton Engbersen. *Multi-Field Packet Classification Using Ternary CAM.* IEE Electronic Letters **38(1)** (2002), 21 – 23.

[18] Summerville, Douglas H., José G. Delgado-Frias and Stamatis Vassiliadis. *A Flexible Bit-Pattern Associative Router for Interconnection Networks.* IEEE Transactions on Parallel and Distributed Systems, **7(5)** (1996), 477 – 485.

[19] Chopra, Vikram, Ajay Desai, Ranghunath Iyer, et al. *Method and Apparatus for High-speed Network Rule Processing.* US Patent, US6510509, 2003.