# Compiling CHR to Parallel Hardware

Andrea Triossi
DAIS, Università Ca' Foscari Venezia, Italy
triossi@unive.it

Salvatore Orlando
DAIS, Università Ca' Foscari Venezia, Italy
orlando@unive.it

Alessandra Raffaetà
DAIS, Università Ca' Foscari Venezia, Italy
raffaeta@unive.it

Thom Frühwirth
Inst. for Software Engineering and Compiler Construction, Ulm University, Germany
fruehwirth@uni-ulm.de

## Abstract

This paper investigates the compilation of a committed-choice rule-based language, Constraint Handling Rules (CHR), to specialized hardware circuits. The developed hardware is able to turn the intrinsic concurrency of the language into parallelism. Rules are applied by a custom executor that handles constraints according to the best degree of parallelism the implemented CHR specification can offer. Our framework deploys the target digital circuits through the Field Programmable Gate Array (FPGA) technology, by first compiling the CHR code fragment into a low level hardware description language. We also discuss the realization of a hybrid CHR interpreter, consisting of a software component running on a general purpose processor, coupled with a hardware accelerator. The latter unburdens the processor by executing in parallel the most computational intensive CHR rules directly compiled in hardware. Finally the performance of a prototype system is evaluated by time efficiency measures.

***Categories and Subject Descriptors*** F.4.1 [*Mathematical Logic*]: Logic and constraint programming

***Keywords*** CHR, Parallelism, Hardware acceleration

## 1. Introduction

In this paper we focus on the hardware compilation of Constraint Handling Rules (CHR) [11] programs. CHR is a committed-choice rule-based language, first developed for writing constraint solvers [12, 14], and nowadays well-known as a general-purpose language [7, 16]. The plain and clear semantics of CHR makes it suitable for concurrent computation, thus allowing programs to be interpreted in a parallel computation model [10].

The hardware compilation technique presented in this paper takes advantage of these features of CHR. Given a program in a suitable subset of CHR, it generates a parallel hardware whose components are: (i) a set of parallel hardware blocks, realizing the rewriting procedures expressed by the CHR rules in the program,

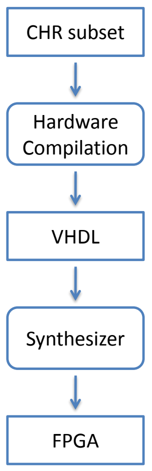

**Figure 1.** Hardware compilation diagram flow

and (ii) a custom unit that interconnects the blocks, and concurrently enacts the rules embedded in the hardware blocks with the constraints in the store.

More precisely, our technique to generate the final hardware digital circuit is based on an intermediate compilation phase, during which a CHR code fragment is first translated into a low level Hardware Description Language (HDL), namely VHDL [33]. From VHDL we can then easily generate synchronous digital circuits, by using automatic tools and the well known Field Programmable Gate Array (FPGA) [15] as deployment technology. Our methodology exploits the programmability features of FPGAs to generate specialized digital circuits for each specific code fragments occurring in a CHR program, in turn compiled in VHDL. The overall hardware compilation flow is depicted in Figure 1. As mentioned above the source language is a subset of CHR: this is due to the intrinsic limitations of hardware circuits, whose resources must be statically allocated. Still, the considered subset of CHR is Turing-complete [27] and it allows to provide natural solutions for several interesting problems.

Concerning the chosen hardware deployment technology, it is worth recalling that nowadays FPGAs take advantage of the growth in the number of transistors that can be integrated within a chip, and can also include more complex components, i.e., processors, memory blocks or special-purpose units. VHDL, the HDL target language of our compilation methodology, works at a very low level, i.e. very close to the Register Transfer Level (RTL). It models a synchronous digital circuit in terms of the flow of digital signals (data) between hardware registers, and the logical operations performed on those signals. The VHDL code can directly feed a vendor-specific design automation tool (the synthesizer) that through different steps produces a gate-level netlist to configure the FPGA. We could also adopt different behavioral HDLs (other than VHDL) as target languages of our hardware compilation methodology, provided that synthesizer tools to program the FPGA circuits are available for those HDLs. Finally, it is worth noting that a FPGA can be programmed multiple times, thus producing specialized circuits for different CHR code fragments at hand.

In order to overcome the limitations of a pure hardware compilation, which forces us to restrict to a subset of the CHR language, we also investigate a hybrid execution architecture for general CHR programs. It combines the custom hardware device described above with a CPU-based software interpreter. The idea is to move the heavier computational burden of a CHR program to this specialized parallel hardware (co-processor), by keeping the remaining part of the program on the main processor. Roughly, the CPU-based interpreter takes care of producing the constraints, while the custom hardware can efficiently consume or rewrite them.

In the current approach, the hardware-software partition for hybrid execution of a CHR program is established a priori by the programmer, who specifies the rules to deploy to the hardware accelerator. A wrapper function virtually encapsulates those rules. It is used as a call, which takes some constraints as input arguments. These are converted as a query for the hardware in a suitable format. The constraints resulting from the hardware execution are returned to the wrapper and made available to the software level.

We also evaluate our hardware compiling methodology in terms of performance. We prototyped the hardware specialized circuits for several significant CHR programs, and compared the execution times obtained. In most cases, we get an improvement of one (or many) order(s) of magnitude in the completion time over standard and optimized software-based CHR interpreters.

In summary, in this paper we provide the following original contributions:

- A novel technique for synthesizing behavioral hardware components starting from a subset of CHR;
- An implementation of an efficient and optimized parallel execution model of CHR by means of hardware blocks, implemented on FPGA (through an intermediate compilation into a low level VHDL language);
- The development of a custom reconfigurable hardware coprocessor that significantly speeds up the execution of a CHR program.

The work is based on some preliminary results that appeared in informal workshop proceedings [30, 31].

The paper is organized as follows. An overview of the CHR language and of the FPGA architecture is presented respectively in Sections 2 and 3. In Section 4 we focus on the technique adopted for generating hardware blocks from CHR rules and in Section 5 we show how to efficiently accommodate in hardware parallel rules execution. In Section 6 we illustrate how the FPGA can fit into a complete computing architecture. Beyond a running example that drives the reader through the description of the parallelism between CHR and hardware, several complete practical examples of implementations are provided. Classical algorithms usually adopted for showing the expressiveness of CHR in multiset transformation or constraint solving, are chosen as case studies. Section 7 discusses related works and finally Section 8 draws the concluding remarks.

## 2. CHR overview

Constraint Handling Rules is a declarative multi-headed guarded rule-based programming language. It employs two kinds of constraints: built-in constraints, which are predefined by the host language, and CHR constraints, which are user-defined by program rules. Each constraint can have multiple arguments and its number is called arity. Null-arity built-in constraints are *true* (empty constraint) and *false* (inconsistent constraint). A CHR program is composed of a finite set of rules acting on constraints. We can distinguish two kinds of rules:

$$\text{Simplification:}\quad Name \mathbin{@} H \Leftrightarrow G \mid B$$
$$\text{Propagation:}\quad Name \mathbin{@} H \Rightarrow G \mid B$$

Where $Name$ is an optional unique identifier of the rule, $H$ (*head*) is a non-empty conjunction of CHR constraints, $G$ (*guard*) is an optional conjunction of built-in constraints, and $B$ (*body*) is the goal, a conjunction of built-in and CHR constraints. These rules logically relate head and body provided that the guard is true. Simplification rules mean that the head is true if and only if the body is true and propagation rules mean that the body is true if the head is true. Rules are applied to an initial conjunction of constraints (*query*) until no more changes are possible. The intermediate goals of a computation are stored in the so called *constraint store*. During the computation if a simplification rule fires the head constraints are removed from the store and they are replaced by the body constraints. If the firing rule is a propagation rule the body constraints are added to the store keeping the head constraints. A third rule called simpagation permits to perform both a simplification and propagation rule:

$$\text{Simpagation:}\quad Name \mathbin{@} H_1 \backslash H_2 \Leftrightarrow G \mid B$$

This rule means that the first part of the head ($H_1$) is kept while the second is removed from the constraint store. Simplification and propagation rules are special cases of simpagation when either $H_1$ or $H_2$, respectively, are empty.

EXAMPLE 1. *We use, as a running example, the program below which computes the greatest common divisor (gcd) of a set of integers using Euclid's algorithm.*

```
R0 @  gcd(N) <=> N = 0 | true.
R1 @  gcd(N) \ gcd(M) <=> M>=N | Z is M-N, gcd(Z).
```

*Starting from a ground query* `gcd(n1),...,gcd(nk)` *the program computes the gcd of $n_1, \ldots, n_k$. Rule* `R0` *states that the constraint* `gcd` *with the argument equal to zero can be removed from the store, while* `R1` *states that if two constraints* `gcd(N)` *and* `gcd(M)` *are present, the latter can be replaced with* `gcd(M-N)` *if* `M>=N`.

A central property of CHR is *monotonicity*: if a rule can fire in a given state then the same firing is possible in a state including some additional constraints. In symbols, for conjunctions of constraints $A$, $B$ and $E$:

$$\frac{A \longmapsto B}{A \wedge E \longmapsto B \wedge E} \tag{1}$$

A direct consequence is the *online* property, i.e., the fact that constraints can be added incrementally during the execution of a program. In fact, monotonicity implies that a final state reached after an execution with an incremental addition of constraints could have been equivalently obtained by having all constraints since the beginning.

Monotonicity is also at the basis of the parallelism of CHR [10]. In fact, it implies that rules operating on disjoint parts of the constraint store can be safely fired in parallel. This property is referred as *weak parallelism*. Formally, if $A$, $B$, $C$ and $D$ are conjunctions of constraints:

$$\frac{A \longmapsto B \quad C \longmapsto D}{A \wedge C \longmapsto B \wedge D} \tag{2}$$

Weak parallelism cannot be applied to rules that operate on non disjoint sets of constraints. *Strong parallelism*, instead, allows for the parallel execution of rules operating on some common constraints provided that they do not modify them. In symbols:

$$\frac{A \wedge E \longmapsto B \wedge E \qquad C \wedge E \longmapsto D \wedge E}{A \wedge E \wedge C \longmapsto B \wedge E \wedge D} \tag{3}$$

## 3. FPGA overview

FPGAs are instances of reconfigurable computing, i.e. computer architectures able to merge the flexibility of software with the performance of hardware, using as processing element high speed configurable hardware fabric [15].

FPGAs are devices containing programmable interconnections between logic components, called *logic blocks*, that can be programmed to perform complex combinational functions. In most FPGAs the logic blocks contain memory elements like simple flip-flops or complete blocks of memory. FPGAs can also host hardware components like embedded hard microprocessors or IP (Intellectual Property) cores that use the logic blocks themselves to realize predefined structures like soft microprocessor cores, i.e., real CPUs entirely implemented using logic synthesis.

The architecture of an FPGA is model and vendor dependent, but in most cases it consists of a bi-dimensional array of configuration logic blocks (CLBs), I/O pads, and routing channels. CLBs are made of few logic cells commonly called *slices*. Each of them consists of some $n$-bit lookup tables (LUT), full adders (FAs) and D-type flip-flops. The $n$-bit LUT realizes the combinatorial part of the circuit: it can encode any $n$-input Boolean function by a truth table model. The FA is used when there is the need to perform arithmetic functions, otherwise it can be bypassed by a multiplexer. Likewise the final D flip-flop can be skipped if we wish an asynchronous output.

The FPGA programmer usually begins the design flow by describing the behavior of the desired hardware in a HDL. The HDLs commonly adopted by the hardware engineers are VHDL [33] (used in all the implementations proposed in this paper) and Verilog [32]. HDL code can directly feed a vendor-specific design automation tool (called *synthesizer*) that through different steps generates a technology-mapped netlist used to configure each CLB. Since each FPGA differs from design architecture, a dedicated process, named place-and-route, takes care of choosing which CLBs need to be employed and how to physically connect them. Before the actual implementation in hardware, the programmer can validate the map via timing analysis, simulation, and other verification methodologies. The final result of the design flow is then a bitstream file that can be transferred via a serial interface to the FPGA or to an external memory device charged to deploy it at every boot of the FPGA.

The key factor that brought FPGAs to success is their programmability. Such a feature guarantees a very short time to production which can easily explain why they quickly emerged as a way for generating effective and low cost hardware prototypes. However, nowadays FPGAs are not only used for prototyping, but, due to the decreasing cost per gate, they are employed as a principal component in many digital hardware designs.

## 4. Compilation to hardware

Here we discuss the main ideas behind our CHR-based hardware specification approach. First, we investigate the features of CHR that could hamper the hardware synthesis, then we address the correspondence between CHR rules and hardware and finally we describe how to reproduce in hardware the execution from the query to the result. As depicted in Figure 1, the complete compilation flow starts from a subset of CHR and goes to an implementation on FPGA passing through the low level VHDL language. Part of the produced VHDL code from the running example is reported in Appendix A.

### 4.1 The CHR subset

Since the hardware resources can be allocated only at compile time (dynamic allocation is not allowed in hardware due to physical bounds), we need to know the largest number of constraints that must be kept in the constraint store during the computation. In order to establish an upper bound to the growth of constraints, we consider a subset of CHR, which does not include propagation rules. Programs are composed of simpagation rules of the form:

$$\begin{array}{ll} \mathit{rule@} & c_1(\overline{X_1}),\ldots,c_p(\overline{X_p})\backslash c_{p+1}(\overline{X_{p+1}}),\ldots,c_n(\overline{X_n}) \Leftrightarrow \\ & g(\overline{X_1},\ldots,\overline{X_n})\,| \\ & Z_1\;\mathit{is}\;f_1(\overline{X_1},\ldots,\overline{X_n}),\ldots,Z_m\;\mathit{is}\;f_m(\overline{X_1},\ldots,\overline{X_n}), \\ & c_{i_1}(\overline{Z}),\ldots,c_{i_m}(\overline{Z}). \end{array} \tag{4}$$

where $\overline{X_i}$ ($i \in \{1,\ldots,n\}$) can be a set of variables, $\overline{Z} = Z_1,\ldots,Z_m$ and the number of body constraints is less than or equal to the number of constraints removed from the head ($m \leq n-p$) and no new type of constraints is introduced: $\{i_1,\ldots,i_m\} \subseteq \{p+1,\ldots,n\}$. In this way, the number of constraints cannot increase and the constraint store size is bounded by the width of the initial query.

Additionally, we will consider only computations starting from a ground goal. Note that, since the variables in the body of a rule are included in those occurring in the head, this implies that all constraints generated during the computations will be ground, a fact which will make possible the translation into hardware.

It is worth recalling that the CHR subset identified by the above conditions is still Turing-complete. This follows from [27], where several subclasses of CHR are shown to be Turing-complete. In particular it is shown that RAM machines can be simulated by CHR programs including only simpagation rules, without free variables and which do not increase the number of constraints (provided that the host language arithmetic is available).

### 4.2 Design of the hardware blocks

The framework we propose logically consists of two parts: $(i)$ Several hardware blocks representing the rewriting procedure expressed by the program rules; $(ii)$ an interconnection scheme among the blocks specific for a particular query. The first one is the hardware needed to implement the concurrent processes expressed by the CHR rules of the program, while the second one is intended for reproducing the query/solution mechanism typical of constraint programming.

The proposed hardware design scheme is outlined in Figure 2. A Program Hardware Block (PHB) is a collection of Rule Hardware Blocks (RHBs), each corresponding to a rule of the CHR program. Constraints are encoded as hardware signals and their arguments as the values that signals can assume. The initial query is directly placed in the constraint store from which several instances of the PHB concurrently retrieve the constraints, working on separate parts of the store. The newly computed constraints replace the input ones. A Combinatorial Switch (CS) sorts, partitions and assigns the constraints to the PHBs taking care of mixing the constraints in order to let the rules be executed on the entire store. The following paragraphs explain in detail the construction of the blocks.

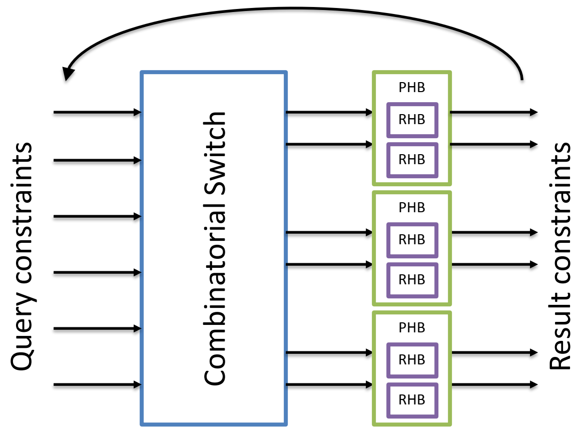

**Figure 2.** Hardware design scheme.

### 4.2.1 Rule Hardware Blocks

The RHB corresponding to the CHR rule in Eq. (4) inputs $n$ signals that have the value of the variables $\overline{X_1} \ldots \overline{X_n}$ (the arguments of the head constraints). If $\overline{X_1} \ldots \overline{X_n}$ are sets of variables, we use vectors of signals (*records* in VHDL). The computational part of the RHB is given by the functions $f_1 \ldots f_m$ that operate on the inputs. The resulting output signals have the value of the variables $\overline{X_1} \ldots \overline{X_p}$ and $Z_1 \ldots Z_m$.

We exploit *processes*, the basic VHDL concurrent statement, to translate the computational part of a rule to a sequential execution. Each rule is mapped into a single clocked *process* containing an *if* statement over the guard variables.

Since, due to constraint removals, the number of constraints can become smaller during the computation, each output signal for a given constraint is coupled with a *valid* signal. This tells to the other components whether the signal should be ignored.

EXAMPLE 2. *Figure 3 sketches the RHBs resulting from the two rules of the* gcd *program in Example 1. Notice that each constraint is associated with two signals: one contains the value of the variable of the constraint (solid line), and the other one models its validity (dashed line).*

*The block in Figure 3(a) corresponds to Rule* R0*. It has as input the value for variable* N *together with its* valid *signal. It performs a check over the guard and if the guard holds the* valid *signal is set to false whereas the value of the gcd signal is left unchanged. This simulates at the hardware level the removal of a constraint from the constraint store.*

*The block in Figure 3(b) is the translation of Rule* R1*. It has four input signals, corresponding to the values of the variables* N *and* M*, with their* valid *signals. According to the guard* M>=N *of* R1*, the inputs* N *and* M *feed a comparator that checks if the value of the second signal is greater than or equal to the first. If the condition is satisfied, the value of the second signal is replaced by* Z = M-N*, as computed by a subtractor, while the value of the first signal remains unchanged. If the guard does not hold, the outputs of the block coincide with the inputs. In both cases the* valid *signals remain unchanged. The computational part is carried out by the subtraction operator.*

### 4.2.2 Program Hardware Block

The PHB is the gluing hardware for the RHBs: it executes all the rules of the CHR program and hence it contains all the associated RHBs, with the corresponding input signals. Intuitively, for any

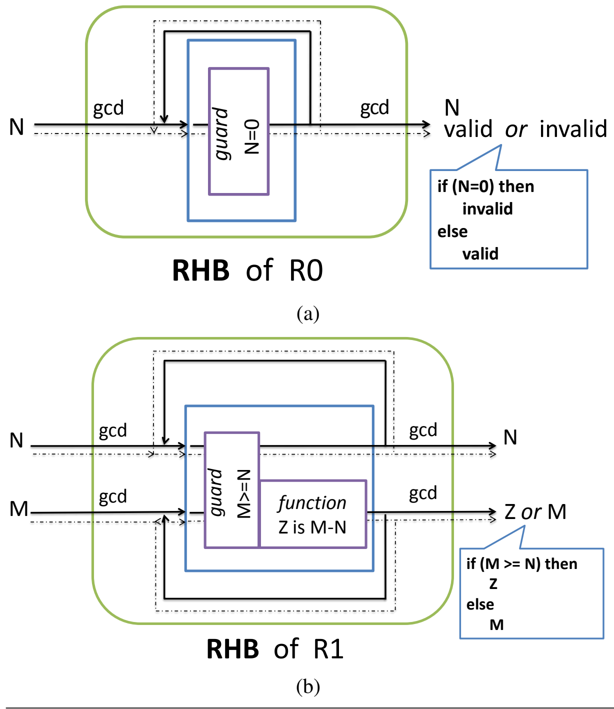

**Figure 3.** The Rule Hardware Blocks for the gcd rules.

constraint type, the PHB inputs the largest number of constraints of that type in input to the underlying RHBs. Additionally, the PHB takes as input the two global input signals *clk* and *reset* used for synchronization and initialization purposes. It provides for the *finish* control signal used to denote when the outputs are ready to be read by the following hardware blocks. The RHBs keep on applying the rules they implement until the output remains unchanged for two consecutive clock cycles.

Note that in the hardware each constraint is represented as a different signal. If the head of a rule contains more than one constraint of the same type, the corresponding signals must be considered as input in any possible order by a RHB encoding the rule. This is obtained by replicating the RHB a number of times equal to the possible permutations of the constraints of the same type.

More precisely, for the sake of simplicity, assume that there is a unique type of constraint and let $k$ be the total number of input constraints to the PHB. For a rule whose head contains $n$ constraints (as in the generic rule in Eq. (4)), the number of copies of RHBs needed is $k!/(k-n)!$, i.e., the number of sequences of length $n$ over the set of $k$ inputs. Finally there is a mechanism for ensuring that only one copy of the RHB can execute per clock cycle.

EXAMPLE 3. *Let us consider the PHB corresponding to the* gcd *program in Example 1. As depicted in Figure 4, it takes as input two signals corresponding to* gcd *constraints (the maximum between the number of inputs of* R0 *and* R1*). According to the general argument above, the number of RHB instances for Rule* R1 *is* $2 = 2!/(2-2)!$*. To see why these are required, note that when* N *is greater than* M*, the rule can fire only if fed with the constraint in reverse order, as it happens for the second copy RHB. Similarly the number of RHB instances needed for* R0 *is* $2 = 2!/(2-1)!$.

A certain degree of parallelism for rules is set at the level of the PHB. Here, according to the notion of strong parallelism for

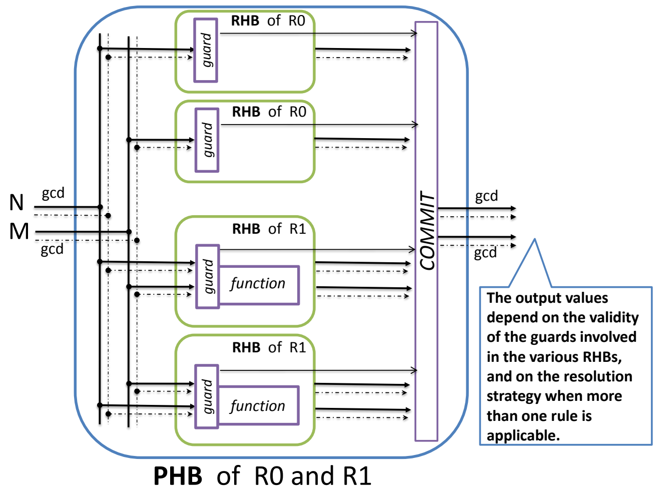
RHB of R0
guard
RHB of R0
guard
RHB of R1
guard
function
RHB of R1
guard
function
N
M
gcd
gcd
COMMIT
gcd
gcd
The output values depend on the validity of the guards involved in the various RHBs, and on the resolution strategy when more than one rule is applicable.
PHB of R0 and R1

**Figure 4.** The Program Hardware Block for the `gcd` program

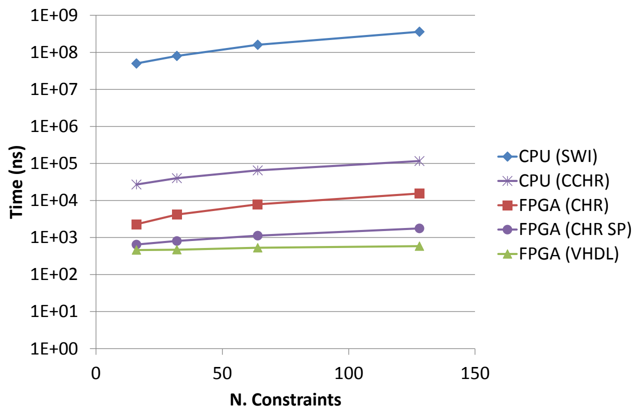

**Figure 5.** Gcd execution time (log scale)

CHR, introduced in Section 2, we allow for the parallel firing of rules sharing constraints which are not rewritten. Actually, all rule instances in the PHB are executed by one or more concurrent *processes* that fire synchronously at every clock cycle. Then, the commit block at the output stage selects only the outputs of a subset of rules which can fire in parallel, chosen according to some priority criteria. For instance in the PHB of the `gcd` example, Rule `R0` cannot be executed in parallel with `R1` because they could rewrite the same constraint.

### 4.2.3 Combinatorial Switch (CS)

A further level of parallelization is achieved by replicating the PHBs into several copies that operate on different parts of the global constraint store, according to weak parallelism as described in Section 2. PHBs can compute independently and concurrently because they operate on different constraints. Although they process data synchronously, since they share a common clock, it is not required that they terminate their computation at the same time. Indeed the CS acts as synchronization barrier letting the faster PHBs wait for the slower ones. It is also in charge to manage the communication among hardware blocks exchanging data: once all the PHBs have provided their results, it reassigns the output signals as input for other PHBs, applying first some permutation to guarantee that all the combinations will be considered.

In practice, the implementation of this interconnection element relies on a signal switch that sorts the $n$ constraints in the query according to all the possible $k$-combinations on $n$ (where $k$ is the number of inputs to the single PHB) and connects them to the inputs of the PHBs. The maximum number of PHBs that can work in parallel on the constraint store is $\lfloor n/k \rfloor$, since, according to weak parallelism, the same signal (and hence the same constraint) cannot be fed to different PHBs at the same time.

Implementing CS as a finite state machine leads to a total number of states $S$ equal to the number of possible combinations divided by the number of concurrent PHBs: $S = \frac{\binom{n}{k}}{\lfloor n/k \rfloor} \approx \frac{\prod_{i=1}^{k-1} n-i}{(k-1)!}$. Despite the good degree of parallelization achieved by the CS (it allows $\lfloor n/k \rfloor$ PHBs to execute in parallel), it needs a number of states $O(n^{k-1})$ in order to try all the possible combinations on the input signals. Since the time necessary for evaluating the query is proportional to the number of states, it is important to limit the number $k$ of inputs for each PHB. This leak in performance is related to the complexity of the *search for matching problem*: a very well known issue in CHR [26] and in general in multi-headed constraint languages. An additional problem is that CS simply combines all the constraints (including the invalid ones) in all possible ways. In presence of algorithms that considerably reduce the number of constraints during computation this can be highly inefficient. In fact, constraints that have been removed by the PHBs still continue to be shuffled by the CS uselessly. In Section 5 we will discuss how to improve time complexity to space complexity's cost giving optimized structures for the CS.

### 4.2.4 Some experiments with `gcd`

Here we describe the hardware implementation of the algorithm presented in Example 1 tailored for finding the greatest common divisor of at most 128 integers. The resulting hardware design relies on 64 PHBs deriving in parallel the gcd. The CS pairs the constraints in a round robin tournament scheme where each constraint is coupled once with each other. For comparison purposes we implement the same algorithm directly in behavioral VHDL using a parallel reduction that exploits the associative property of the gcd operation. Both hardware specifications are then synthesized in a Xilinx Virtex4 FPGA (xc4vfx60) running at 100MHz. Figure 5 reports the execution times for 16, 32, 64 and 128 2-byte integers. The two FPGA implementations are labeled respectively as FPGA (CHR) and FPGA (VHDL). The curves labeled CPU (SWI) and CPU (CCHR) refer to the computational time of the CHR `gcd` program, compiled respectively with the K.U.Leuven CHR system of SWI Prolog [23] and the fast C-based system CCHR [35], and running on Intel Xeon 3.60GHz processor with 3GB of memory. Observe that the FPGA implementations are at least one order of magnitude faster than the software implementations (including CCHR which is claimed to be the fastest CHR implementation currently available). This is somehow expected, due to the completely different hardware nature, but still it provides an indication of the appropriateness of our approach.

Compared to the VHDL solution, the execution time can be more than an order of magnitude larger. This is primarily due to the fact that, as mentioned above, the CS does not take into account that the number of constraints can drastically decrease. An optimization addressing this issue will be discussed in Section 5.1 (the outcome of such an optimization is reported in Figure 5, labeled FPGA (CHR SP)).

The area needed for the largest `gcd` implementation we tried is about $2 \cdot 10^5$ LUTs corresponding to about 8% of occupation of a medium size FPGA. Finally we should notice that the resulting highest frequencies of operations are all above 250 MHz and up to 350 MHz, which is quite good for a non pipelined architecture.

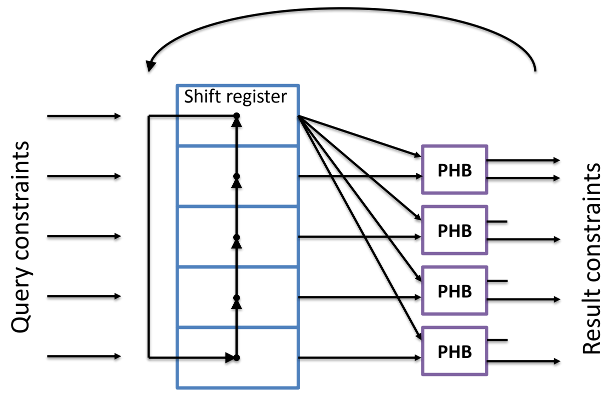

**Figure 6.** Optimization model for strong parallelism.

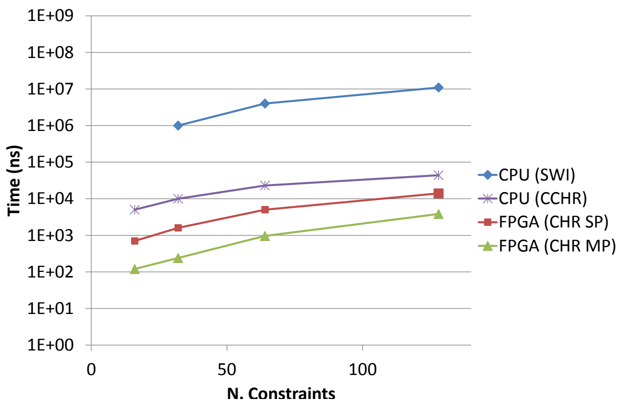

**Figure 7.** Prime execution time (log scale)

## 5. Optimizing hardware compilation

With the aim of facing the problem of time efficiency, three optimizations are proposed relying on different degrees of parallelization. In particular, in Section 5.1 we discuss how the property of strong parallelism can be further exploited with simple changes in the hardware framework. In Section 5.2 we show how the adoption of a set based semantics for CHR can enable a higher level of parallelism. Finally, in Section 5.3, the online property of CHR (see Section 2) is used for boosting the computation of a merge sort algorithm.

### 5.1 Strong parallelism

Here we propose an optimization which exploits the strong parallelism property of CHR also at the level of the CS. The idea consists of distinguishing between constraints that are read from those which are rewritten by the PHBs. Then, the same read constraints can be processed in parallel by all the PHBs. A hardware block, obtained as a modification of the CS used before, takes care of feeding the various PHBs instances with the same read constraints combined with different sets of rewritten constraints.

For instance, consider the `gcd` program. The PHB now contains a single instance of RHB for each Rule `R0` and `R1`. The PHB inputs a read constraint (the first input of `R1`) and a removed constraint (shared between the two rules). Note that in this case, since there is no rule duplication inside the PHB, the order of signals matters.

Figure 6 shows a possible refined CS for a five constraints query but the design can be easily adapted (with a linear growth) to a larger number of constraints. It relies on a circular shift register[1] preloaded with the query constraints and with one cell connected to all the first input (read constraint) and all the others connected to the second input (removed constraint) of each PHB. Each time the PHBs terminate their computation the new output constraints replace the old ones in the shift register and they shift until a *valid* constraint fills the first position of the register. Note that since the outputs carrying the read constraint refer to the same constraint for all PHBs, they are all left disconnected apart from the first one (see Figure 6).

Experimental results for the proposed strong parallel architecture are reported in Figure 5, labeled by FPGA (CHR SP). The reduction in execution time is relevant for all the experiments with different number of constraints, reaching up to one order of magnitude of speed up.

[1] A shift register is a cascade of registers in chain, with the data input of the first element connected to the output of the last one. An enable signal determines a circular shift of one position of the stored data.

### 5.2 Massive parallelism

The set-based semantics CHR$^{mp}$ [21] relies on the idea that constraints can be considered as multiplicity independent objects so that additional copies of them can be freely used. In such a context, a duplicate removal of one constraint can be replaced by the removal of two copies of the same constraint. The degree of parallelism introduced by this change of perspective is extremely high since multiple rule instances removing or reading the same constraint can be applied concurrently (intuitively each one operates on a different copy of the constraint). The main drawback of CHR$^{mp}$ is that it is not sound with respect to the sequential semantics when the program is not deletion-acyclic (i.e., when two distinct constraints are responsible for their mutual removal).

CHR$^{mp}$ is particularly suited for algorithms that considerably reduce the number of constraints like the filtering ones. Consider as an example the program below that, starting from the query `prime(2), ..., prime(n)` extracts the prime numbers in the interval $[2, n]$:

```
Prime @ prime(X) \ prime(Y) <=> Y mod X = 0 | true.
```

Note that Rule `Prime` is in the CHR fragment defined by Eq. (4) as the number of `prime` constraints decreases every time the rule fires. The program is also deletion-acyclic since two integers cannot be one a proper multiple of the other. Moreover, the execution of the program can take advantage also from strong parallelism since multiple instances of the rule can use the same `prime` constraint for reading.

The idea for exploiting massive parallelism consists of providing all possible combinations of constraints (order matters) in input to distinct parallel PHB instances in a single step. This time the same constraint will be fed to several PHBs. Valid outputs are collected: a constraint is valid if no PHB has removed it. This is realized in hardware by suitable AND gates. Finally, valid outputs are used as input in the next round. The architecture thus ideally uses $\binom{n}{k}$ PHBs where $n$ is the number of query constraints and $k$ is the number of inputs of each PHB. In practice, physical bounds can impose to use a smaller number of PHBs, in a way that processing all possible combinations of constraints will require more than one step.

Figure 7 reports the execution time for the `Prime` program. The strong parallelism and massive parallelism optimizations are tagged as FPGA (CHR SP) and FPGA (CHR MP), respectively. The improvement determined by the massive parallelism is about an order of magnitude for queries with a low number of constraints, and it decreases with higher numbers of constraints. This is due to the fact that the physical bounds of the hardware are quickly

```
M0    @  arc(X,A) \ arc(X,B) <=> A<B | arc(A,B).
M1    @  seq(N,A), seq(N,B) <=>
                A<B | seq(N+N,A), arc(A,B).
```

$$\Downarrow$$

```
M0    @  c(0,X,A) \ c(0,X,B) <=> A<B | c(0,A,B).
M1    @  c(1,N,A), c(1,N,B) <=>
                A<B | c(1,N+N,A), c(0,A,B).
```

**Figure 8.** Merge sort algorithm

reached. The occupied area of the FPGA determined by a complete parallelization would increase as $\binom{n}{k}$. Even though in this case $k$ is small ($k = 2$), we obtain a quadratic growth that is not sustainable and a partial serialization is needed.

### 5.3 Online optimization

In this section we illustrate the optimizations which can be allowed by the online property, working on a typical CHR program implementing the merge sort algorithm with optimal complexity [11].

The CHR program consists of the Rules M0 and M1 in the upper part of Figure 8. Given a ground query of the form seq(1,n1), . . . , seq(1,nk) the program returns the ordered sequence of the input numbers n1, . . . , nk, represented as a chain of arcs using the constraint arc/2. For instance, starting from seq(1,9), seq(1,3), seq(1,7), seq(1,4) the program returns the following arc constraints: arc(3,4), arc(4,7), arc(7,9).

Rule M0, where two arcs arc(X,A) and arc(X,B) start from the same number X, performs their ordered merge into a chain. The arc with the smaller target is kept by the rule, while the other is replaced by an arc between A and B. The insertion in the store of such a constraint may cause a new branch in the sequence, and hence the rule keeps firing until all the branches have been removed. In order to reach the optimal complexity $O(n\ log(n))$, the chains which are merged should have the same length. Rule M1 is responsible for the initialization of the arc/2 constraints in order to meet such a requirement. In a constraint seq(l,n) the second argument n is one of the numbers to be ordered, while the first argument l represents the number of elements reachable from n via arc connections. In the initial query, since no arc connections exist, all constraints are of the kind seq(1,ni).

The program, strictly speaking, does not belong to the CHR subset implementable in hardware. In fact, while Rule M0 leaves unchanged the number of arc/2 constraints, Rule M1 reduces by one the number of seq/2 constraints but introduces a *new* arc/2 constraint. However, the program can be easily transformed into an equivalent one in the CHR subset of interest. Since in Rule M1 the total number of constraints is left unchanged, it is sufficient to flatten the two types of constraints involved, i.e., arc/2 and seq/2, into a single one c/3, with an additional argument used to encode the constraint type. The transformed program can be found in the bottom part of Figure 8.

Note that such constraint flattening is always possible using a new constraint with an arity equal to the greatest arity of the original constraints plus one.

Following the methodology described in Section 4.2, the transformed program for merge sort can be compiled into hardware. Each PHB has two inputs and two outputs and it includes four RHBs. In fact, since both Rules M0 and M1 have two constraints of the same type in the head, as in the running example (see in particular Example 3), two RHBs are needed for each rule. Note that the commit stage of a PHB selects the result of a single RHB since after the constraint flattening no parallelization is possible between rules. The actual parallelization is performed by the CS that pairs the query constraints and assigns each couple to a different instance of the PHB. The number of instances of PHB which works in parallel on the constraint store is thus $\lfloor n/2 \rfloor$, where $n$ is the total number of constraints in the store.

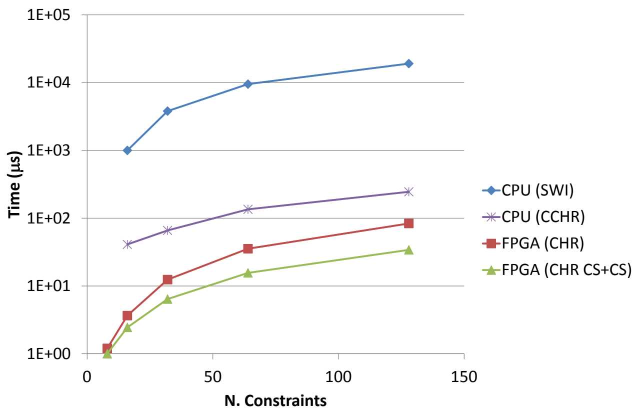

**Figure 9.** Merge sort execution time (log scale)

We next propose an alternative architecture, which further parallelizes rule executions by exploiting the online property of CHR. The key observation is that, relying on this property, the merging operation performed by Rule M0 can be executed in parallel to the generation of the arc constraints as carried out by Rule M1. Hence the program can be naturally split into two parts corresponding to the two rules and some of the resulting constraints of one part can be used to populate runtime the constraint store of the second one. The arc/2 constraints produced by Rule M1 are consumed only by Rule M0 while seq/2 constraints are produced and consumed only by Rule M1. If we consider the two rules as separate programs joined by the arc production and consumption, we can design a hardware consisting of two executors linked by a one-way FIFO buffer.

The executor for each rule consists of a CS and $\lfloor n/2 \rfloor$ instances of the PHB described above, where $n$ is the number of query constraints.

The seq/2 query constraints are loaded in the CS of the first executor and, as new arc/2 constraints are created (actually c/3 constraint with the first argument equal to 0), they are inserted in the FIFO that feeds the CS of the second executor. Such a CS at the beginning of the computation is empty, a fact which is concretely implemented by preloading the CS with constraints which are all non valid. Then, whenever a new constraint is received from the buffer it will replace one of the non valid constraints. The two CS and the FIFO should have the same dimension since, in a normal execution, all the seq constraints (except the last that always remains unpaired) are converted in arc constraints. The FIFO depth has to take into account the possibility that the second CS is not able to receive immediately the sent constraint because the receiving cell is occupied by a valid constraint.

Figure 9 shows a comparison between the execution time of the non optimized and optimized hardware implementations, labeled as FPGA(CHR) and FPGA(CHR CS + CS), respectively. The optimized implementation, when the number of elements to be ordered increases, outperforms the non-optimized one. This shows the advantage of exploiting the online property that gives the possibility of dividing the problem into two parts running in parallel. Also these experiments confirm that the FPGA implementations are much more efficient than the software ones, in SWI Prolog and CCHR.

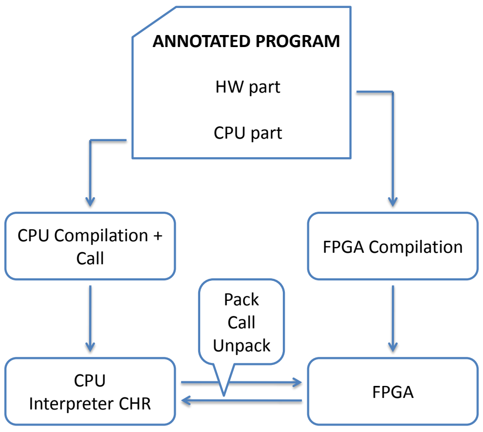

**Figure 10.** Hardware/software compilation diagram flow

Finally, we note that in the optimized FPGA architecture the CS of the second executor could be replaced by a shift register, like the one used in presence of strong parallelism (see Section 5.1). In fact, multiple instances of Rule `M0` can be strongly parallelized, because one constraint of the head is kept and hence can be shared among multiple rules. However experimental results shows that such a architecture does not speed up the execution at all. This is due to the fact that when the last `arc/2` constraint is generated by the first executor, the partial result of the second one approximates very closely the final result (which could be already the correct one). Thus when the last constraint is retrieved by the second executor, it has to apply just a single rule to reach the end of the execution. Hence the chosen parallelism for sorting the constraints does not matter.

## 6. An accelerator for CHR

In Section 4 it was shown how to synthesize hardware starting from a subset of CHR that does not allow constraint propagation. Since dynamic allocation is not permitted in hardware due to physical bounds, such a restriction might be expected for a hardware designer, but it turns out to be very restrictive for software programmer. In order to overcome this limitation, we propose a mixed (hardware/software) system where some rules, whose execution represents the heavier computational task, are executed by specialized hardware (synthesized through the aforementioned technique), while the remaining ones are executed by the main processor that can overcome the hardware limitations. The processor can easily take care of producing constraints, while the custom hardware can efficiently consume or rewrite them.

The issue of partitioning between hardware and software implementations is left to the programmer, who specifies which rules should be deployed to the hardware accelerator. A wrapper function virtually encapsulates those rules. It is used as a call, which takes some constraints as input arguments. These are converted to a query for the hardware in a suitable format. The constraints resulting from the hardware execution are given back to the wrapper and made available to the software level. The wrapper allows the programmer to access to lower level instructions (in this case a call to a hardware driver), which speed up the execution. This kind of modularity is known in the literature as hierarchical composition [8, 24] and an implementation similar to ours can be found for instance in [26].

The entire system compilation is split into two branches (see Figure 10) related to software and hardware parts. The source program is annotated by the programmer who specifies the rules that have to be executed by the hardware accelerator. The hardware compilation is performed according to the method in Section 4.2 which results in a bit stream directly deployable on an FPGA. On the other hand the standard software compilation will be necessarily altered to deal with the hardware realization of some rules of the program. Since our implementation relies on a CHR system that adopts Prolog as *host language*, the execution of the hardware implemented rules will be embedded in a custom made built-in foreign Prolog predicate (the wrapper). When it is called, all the constraints needed by the hardware are sent to the accelerator, which will return the resulting constraints back to the constraint store.

*GCD matrix calculation.* As a first case study, let us consider a program that, given a set of integers, computes the gcd of all the pairs of inputs. To this end it builds a bi-dimensional triangular upper matrix whose elements will contain the gcd of all pairs of integers belonging to the set. The query is formulated using the constraint `set/2`, where the second argument represents a number in the input set, while the first expresses its (arbitrary) order in the set, e.g., with the query `set(1,n1),...,set(k,nk)` the program will compute the gcd for the set $\{n_1, \ldots, n_k\}$. The matrix is built by using the constraint `gcd/3`. In the constraint `gcd(X,Y,M)` the first two arguments, `X,Y` denote the position of the element in the matrix whereas the third one will contain the gcd of `nX` and `nY`. The first and the last two rules reported in Figure 11 are the CHR implementation of the matrix computation. The computation of the gcd according to Euclid's algorithm is then expressed by Rules `GCD0` and `GCD1`. The propagation Rules `Matrix0` and `Matrix1` are employed to build the matrix from the initial set of inputs. Rule `Matrix0` produces the upper half of the matrix (due to the guard `X<Y`), creating the initial query `gcd(X,Y,N),gcd(X,Y,M)` for the computation of the `gcd` between the numbers `N` and `M`. Rule `Matrix1`, in contrast, generates the diagonal elements. In this case the `gcd` is trivially equal to the set element `N`. These two rules cannot be implemented in hardware because they are propagation rules that generate new `gcd` constraints.

In the hardware accelerator we deploy the functionality of rules `GCD0` and `GCD1` with the hardware blocks technique reported in Section 4.2. The remaining program running on the main processor consists of the two Rules `Matrix0` and `Matrix1`, with the addition of the rules reported in the central part of Figure 11. Rule `Pack` is intended to append all the constraints of type `gcd/3` to a *list* that has to be delivered to the hardware accelerator. `Call` is used to trigger the invocation of the custom Prolog predicate `hw_gcd/2` that is the actual responsible of the data transfer to and from the hardware accelerator. The constraint `call/0` is available to the programmer to make the rule fire at the preferred time. For instance, in our example we could use the query `set(1,n1),...,set(k,nk),call` if we wish that the `gcd` constraints were processed after the complete production of all of them. Finally the Rule `Unpack` transforms the output list returned by the hardware accelerator into constraints. In this particular example the application of such a rule would not be necessary because the output of the gcd computation is just one constraint, which could be returned by using the Rule `Call`. This rule is inserted for generality purpose.

The hardware setup of the test bench relies on a Xilinx Virtex4 FPGA (xc4vfx60) running at 100MHz and connected to a PCI-E root complex of an ASUS P7P550 motherboard hosting an Intel Core i7 CPU running at 2.8GHz. On the software side we use the CHR system [23] for SWI-Prolog which allows for an easy integration of memory mapping instructions thanks to the embedded interface to C [34]. We employed this feature for the wrapper implementation. In order to determine the total system execution time we used a single thread implementation in which the CPU is kept idle until the FPGA has performed its computation. Figure 12 com-

```
GCD0    @   gcd(_,_,0) <=> true.
GCD1    @   gcd(X,Y,N) \ gcd(X,Y,M) <=> M>=N | gcd(X,Y,M-N).
-----------------------------------------------------------------
Pack    @   gcd(X,Y,N), list_in(L)#passive <=> list_in([(X,Y,N)|L]).
Call    @   call, list_in(L1) <=> hw_gcd(L1,L2), list_out(L2).
Unpack  @   list_out([(X,Y,N)|L]) <=> list_out(L), gcd(X,Y,N).
-----------------------------------------------------------------
Matrix0 @   set(X,N), set(Y,M) ==> X<Y | gcd(X,Y,N), gcd(X,Y,M).
Matrix1 @   set(X,N) ==> gcd(X,X,N).
```

**Figure 11.** Gcd matrix program

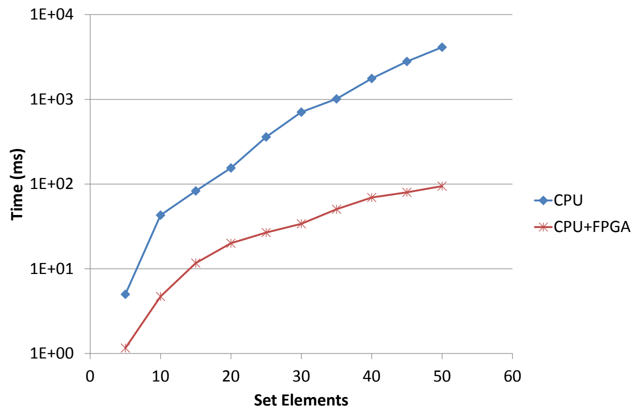

**Figure 12.** GCD matrix execution time (log scale)

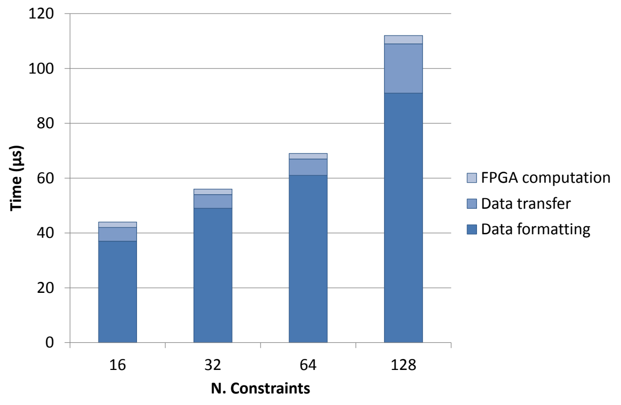

**Figure 13.** Additional time introduced by the hardware accelerator measured at different sizes of the query

pares the execution times of the gcd matrix running on the plain CPU and with the help of the FPGA (labeled CPU+FPGA in the plot). Even if the speed achieved is not comparable with the one obtained by the execution of the gcd algorithm entirely in FPGA (see Section 4.2.4), the execution time improvement with respect to the plain CPU is still in the range of one order of magnitude. This poorer speed up is rewarded by a higher flexibility.

A comparison of the extra time introduced with the addition of the hardware accelerator is presented in Figure 13, for 16, 32, 64, and 128 1-byte constraints. We measured the elapsed time as the sum of three different components: data formatting, data transfer and FPGA computation. The first one is the required time by a CHR rule for calling the foreign Prolog predicate that converts terms in basic C-type values, arranges the constraint values in data packets, decodes the incoming packets and unifies the C-type values with terms. The remaining two components are, respectively, the routing time to send data through the PCI-E bus and the time needed by the FPGA for packing/unpacking and processing data. The measures show that the most expensive phase is the data handling at the level of the CHR rule, responsible of the built-in execution that sets up the FPGA computation. Clearly such burden of few microseconds per constraint is fully rewarded by the speed up gained in the further concurrent execution of CHR rules in FPGA.

*Interval domain solver.* As a further case study we consider a classical problem for constraint programming, i.e., a finite domains system [1]. In the literature about CHR we can find several programs working on interval or enumeration constraints [13]. Here we implement a simple interval domains solver for bound consistency, whose CHR code can be found in Figure 15. The solver uses the CHR constraint `::/2` for stating that a given variable ranges on a finite set denoted by the custom operator `:`. For example, the constraint `X::a:b` means that the variable `X` can assume any integer value between `a` and `b`. Also `le/2`, `eq/2` and `ne/2` are CHR constraints, representing the less or equal, the equal and the not equal operators, while `min` and `max` are Prolog built-in operators. Rule `Redundant` eliminates an interval constraint for a variable when there exists another interval constraint for the same variable which is included in the first one. Rule `Intersect` replaces two intervals with their intersection which is obtained by calculating the maximum of the lower bounds and the minimum of the upper bounds. Rule `Inconsistent` identifies empty intervals (where lower bound is greater than the upper bound). Rules `LessEqual`, `Equal` and `NotEqual` represent the corresponding arithmetic relations. A sample query for the program can be: `X le Y, X::3:5, Y::2:4`. The program first produces the new constraints `X::3:4, Y::3:4`, and then it eliminates the redundant intervals giving as result `X le Y, X::3:4, Y::3:4`.

The first two Rules `Redundant` and `Intersect` are deployed on FPGA. Note that they are in the CHR fragment defined by Eq. (4): `Redundant` removes a constraint without introducing new ones while `Intersect` introduces a new constraint, but it removes two of them.

Observe that the typical queries include free logical variables as arguments for the constraints (e.g., `X` and `Y` occur free in the sample query above). This is not a problem for the hardware computation since during the whole program execution such variables are never bound. Hence they can be replaced by suitable indexes in the step of packing and formatting the constraints to be sent to the accelerator. When the output constraints from the accelerator are received the indexes can be replaced back with the corresponding logical variables.

The execution time of the interval domains solver is reported in Figure 14. The times correspond to queries of different length depending on the number of variables taken into account. More precisely, the queries contain 20 interval constraints (e.g. `X::3:15`) and one arithmetic relation (e.g. `X le Y`) for each variable. As

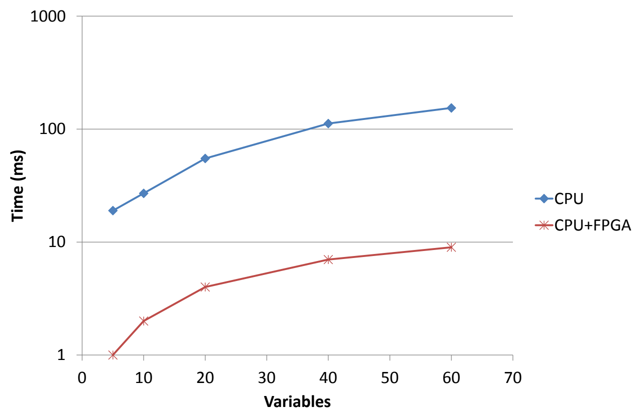

**Figure 14.** Interval domains solver execution time (log scale)

in the case of the computation of the gcd matrix, the speed up obtained with the support of the hardware accelerator is above one order of magnitude on all the queries we considered.

## 7. Related work

Our hardware compilation method, which starts from a subset of CHR to arrive at generating a synchronous digital circuit, can also be seen as an attempt to overcome the too low level feature of traditional HDLs, such as VHDL [33] and Verilog [32]. These are well proven and established standard languages for hardware design, but they force the hardware designer to think at the Register Transfer Level (RTL) level, thus modeling a synchronous digital circuit in terms of the flow of digital signals (data) between hardware registers, and the logical operations performed on those signals. From these low level language, is pretty easy to end in a gate-level netlist that can be directly mapped into hardware.

Many alternative environments have been proposed to unify the hardware engineers and the software developers through the use of a common high-level language, mainly based on imperative languages, but none of them has become a standard due to the inherent lack of concurrency and timing (essential elements for the hardware synthesis) of such languages [9]. On the other hand, extensions of commonly adopted HDLs like SystemVerilog [29] still require to the programmers to own a strong hardware background and are too specific to be used as general purpose languages.

We can count a large number of successful approaches to hardware description among the functional languages. Since the 80s one of the most popular domains in which functional languages have been extensively used is hardware design [25]. General purpose functional languages, like Haskell, have been widely used as host languages for embedding HDL (e.g. Hydra [20] or Lava [5]). More recent approaches like SAFL [19] move from a *structural* to a *behavioural* description of the hardware. They allow the programmer to directly describe the algorithm to be implemented rather than the interconnections among low-level hardware components.

Logic programming and especially Prolog have been used for many years as formalisms for hardware design specification and verification as well. We can mention some recent approaches [2, 3] that present a Prolog-based hardware design environment hinged on a high-level structural language called HIDE+. Such a language was developed with the purpose of filling the gap of the structural HDL languages that can only deal with small circuits. Indeed the HDL description tends to be very complex due to the need of making all the connections explicit. Another work on the track of the behavioural style was presented in [18]. It adopts the Byrd boxes model [6] of program execution originally developed as debugging tool for Prolog. The Byrd boxes are used to identify a statically allocable subset which can be executed by associating a single Byrd box with each predicate symbol.

CHR is deemed as a highly concurrent language and, indeed, in our hardware compilation framework we have largely exploited various forms of parallelism in CHR. However, it is broadly accepted that a parallel computation model for CHR is still *in fieri*. The first example of parallel implementation can be found in [10] where it is shown how to evaluate the degree of concurrency starting from the confluence analysis of a sequential program execution. Further works [17, 28] focus on the formal specification and the development of a parallel implementation of the CHR goal-based execution schema: multiple processor cores run multiple threads solving a single CHR goal. Other attempts to exploit concurrency in CHR were pursued in the last years, mainly driven by the CHR set-based operational semantics [22]. Although CHR programs usually adopt a multiset based semantics, it was shown how a large class of programs can benefit from a tabled rule execution schema that eliminates the need of a propagation history, and acquires a natural parallelism by the notion of set. The persistent constraint semantics presented in [4], which exploits the idea of a mixed store where the constraints can behave like a set or a multiset, achieves a higher degree of declarativity, keeping the potentiality of concurrency of the standard semantics. Finally, massive parallelism [21], used in Section 5.2, gives the possibility of applying multiple removals to the same constraint. Such semantics eliminates the conflicts in the constraint removals by allowing different rule instances to work concurrently on distinct copies of the constraints.

## 8. Conclusion

We described the general outline of an efficient hardware implementation of a CHR subset able to comply with the limitations imposed by hardware. The level of parallelization achieved provides a time efficiency comparable with that obtained with a design directly implemented in HDL. At the same time, the proposed solution offers a more general framework reusable for a wide range of tasks and easily integrable with existing low level HDLs. Different degrees of parallelization naturally embedded in CHR were pointed out and fully exploited thanks to the development of custom hardware structures. The proposed hardware compilation was validated on several case studies related to classical algorithms like Euclid's algorithm, a sieve for prime numbers and merge sort.

In order to cope with the static nature of hardware, which prevents a dynamic allocation, our translation has been restricted to a proper subset of CHR, not including propagation rules. For overcoming this limitation we proposed a classical CHR executor coupled with a hardware accelerator dedicated to simple tasks like the fast rewriting of some constraints. Such hybrid system can increase the performance of CHR, achieving a stronger coupling between algorithms and platforms. In case of data intensive algorithm, the burden of setting up the accelerator computation was fully paid off by the speed up gained in the concurrent execution of CHR rules in hardware.

Further improvements to the general framework, especially in terms of applicability to problems where the number of constraints does not necessarily decrease during the computation, will be subject to future research. A general treatment of rule dependencies at the PHB level is still missing and only appropriate considerations on rules interaction can lead to a hardware performing parallel execution, pipelining and balancing out circular dependencies. Regarding the hardware accelerator, we should mention the possibility of automating the process of rule selection for the hardware deployment. Results coming from a profiler could help a static analysis on the CHR program in order to identify the rules that are the most expensive to be executed. Moreover it would be interesting to test our

```
Redundant     @   X::A:B \ X::C:D <=> C=<A, B=<D | true.
Intersect     @   X::A:B, X::C:D <=> X::max(A,C):min(B,D).
─────────────────────────────────────────────────────────

Inconsistent  @   _::A:B <=> A>B | fail.
LessEqual     @   X le Y, X::A:_, Y::_:D ==> Y::A:D, X::A:D.
Equal         @   X eq Y, X::A:B, Y::C:D ==> Y::A:B, X::C:D.
NotEqual      @   X ne Y, X::A:A, Y::A:A <=> fail.
```

**Figure 15.** Interval domains solver algorithm

framework, which provided quite satisfactory preliminary result, to more complex applications.

## References

[1] F. Benhamou. Interval Constraint Logic Programming. In *Constraint Programming: Basics and Trends*, volume 910 of *LNCS*, pages 1–21. Springer, 1995.

[2] A. Benkrid and K. Benkrid. HIDE+: A Logic Based Hardware Development Environment. *Engineering Letters*, 16(3):460–468, 2008.

[3] K. Benkrid and D. Crookes. From Application Description to Hardware in Seconds: A Logic-Based Approach to Bridging the Gap. *IEEE Transaction on VLSI System*, 12(4):420–436, 2004.

[4] H. Betz, F. Raiser, and T. W. Frühwirth. A complete and terminating execution model for constraint handling rules. *Theory and Practice of Logic Programming*, 10(4-6):597–610, 2010.

[5] P. Bjesse, K. L. Claessen, M. Sheeran, and S. Singh. Lava: Hardware design in Haskell. In *Proc. of the International Conference on Functional Programming*, pages 174–184. ACM Press, 1999.

[6] L. Byrd. Understanding the Control Flow of Prolog Programs. In *Workshop on Logic Programming*, pages 127–138. S.A. Tärnlund, 1980.

[7] H. Christiansen. CHR grammars. *Theory and Practice of Logic Programming*, 5(4-5):467–501, 2005.

[8] G. J. Duck, P. J. Stuckey, M. Garcia de la Banda, and C. Holzbaur. Extending arbitrary solvers with constraint handling rules. In *Proc. of PPDP'03*, pages 79–90. ACM, 2003.

[9] S. A. Edwards. The Challenges of Hardware Synthesis from C-Like Languages. In *Proc. of DATE'05*, pages 66–67. IEEE Computer Society, 2005.

[10] T. Frühwirth. Parallelizing Union-Find in Constraint Handling Rules Using Confluence Analysis. In *Proc. of ICLP'05*, volume 3668 of *LNCS*, pages 113–127. Springer, 2005.

[11] T. Frühwirth. *Constraint Handling Rules*. Cambridge University Press, 2009.

[12] T. Frühwirth and S. Abdennadher. The Munich rent advisor: A success for logic programming on the internet. *Theory and Practice of Logic Programming*, 1(3):303–319, 2001.

[13] T. Frühwirth and S. Abdennadher. *Essentials of constraint programming*. Springer, 2003.

[14] T. Frühwirth, P. Brisset, and J.-R. Molwitz. Planning cordless business communication systems. *Expert, IEEE [see also IEEE Intelligent Systems and Their Applications]*, 11(1):50–55, 1996.

[15] S. Hauck and A. DeHon. *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation*. Morgan Kaufmann Publishers Inc., 2007.

[16] Z. Hongwei, S. E. Madnick, and M. Siegel. Reasoning About Temporal Context Using Ontology and Abductive Constraint Logic Programming. In *Proc. of PPSWR'04*, volume 3208 of *LNCS*, pages 90–101. Springer, 2004.

[17] E. S. L. Lam and M. Sulzmann. Concurrent Goal-Based Execution of Constraint Handling Rules. *CoRR*, abs/1006.3039, 2010.

[18] A. Mycroft. Allocated Prolog—Hardware Byrd Boxes. 2002. Presented as part of 25th anniversary celebrations of the Computer Science Department of the Universidad Politécnica de Madrid.

[19] A. Mycroft and R. Sharp. Higher-level techniques for hardware description and synthesis. *International Journal on Software Tools for Technology Transfer (STTT)*, 4(3):271–297, 2003.

[20] J. O'Donnell. Hydra: hardware description in a functional language using recursion equations and high order combining forms. In *The Fusion of Hardware Design and Verifications*, pages 309–328. North Holland, 1988.

[21] F. Raiser and T. Frühwirth. Exhaustive parallel rewriting with multiple removals. In *Proc. of 24th Workshop on (Constraint) Logic Programming*, 2010.

[22] B. Sarna-Starosta and C. R. Ramakrishnan. Compiling Constraint Handling Rules for Efficient Tabled Evaluation. In *Proc. of PADL '07*, volume 4354 of *LNCS*, pages 170–184. Springer, 2007.

[23] T. Schrijvers and B. Demoen. The K.U.Leuven CHR System: Implementation and Application. In *First Workshop on Constraint Handling Rules: Selected Contributions*, pages 1–5, 2004.

[24] T. Schrijvers, B. Demoen, G. Duck, P. Stuckey, and T. Frühwirth. Automatic Implication Checking for CHR Constraints. In *Proc. of RULE'05*, volume 147 of *ENTCS*, pages 93–111. Elsevier, 2006.

[25] M. Sheeran. Hardware design and functional programming: a perfect match. *Journal of Universal Computer Science*, 11(7):1135–1158, 2005.

[26] J. Sneyers, T. Schrijvers, and B. Demoen. Dijkstra's Algorithm with Fibonacci Heaps: An Executable Description in CHR. In *Proc. of WLP'06*, pages 182–191, 2006.

[27] J. Sneyers, T. Schrijvers, and B. Demoen. The computational power and complexity of constraint handling rules. *ACM Trans. Program. Lang. Syst.*, 31(2):1–42, 2009.

[28] M. Sulzmann and E. S. L. Lam. Parallel Execution of Multi Set Constraint Rewrite Rules. In *Proc. of PPDP'08*, pages 20–31. ACM Press, 2008.

[29] Systemverilog. *SytemVerilog 3.1 - Accelleras Extensions to Verilog(R)*, 2003. Accellera Organization Inc.

[30] A. Triossi. Boosting CHR through Hardware Acceleration. In *Proc. of CHR'11*, pages 1–3, 2011. Invited talk.

[31] A. Triossi, S. Orlando, A. Raffaetà, F. Raiser, and T. Frühwirth. Constraint-based hardware synthesis. In *Proc. of Workshop on (Constraint) Logic Programming*, pages 119–130, 2010.

[32] Verilog. *IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language*, 1996. http://www.ieee.org/.

[33] Vhdl. *IEEE Standard VHDL Language Reference Manual*, 1994. http://www.ieee.org/.

[34] J. Wielemaker. An overview of the SWI-Prolog Programming Environment. In *Proceedings of the 13th International Workshop on Logic Programming Environments*, pages 1–16, 2003.

[35] P. Wuille, T. Schrijvers, and D. B. CCHR: the fastest CHR Implementation, in C. In *Proc. of CHR'07*, pages 123–137, 2007.

## A. Generated VHDL code

This Appendix contains some program listings generated from the implementation of the running example presented in Section 4.

In the following we show the VHDL code that implements the hardware blocks needed to form the PHB of the gcd program reported in Example 3. The first part of the code shows the *entity* declaration of gcd, which contains the input and output signals of the PHB. Besides the signals related to the RHBs `gcdx`, `validx`, `gcd_outx` and `valid_outx`, the port listing contains the synchronization signals provided by the PHB: `clk`, `reset` and `finish`. The *architecture* of `gcd` has four *processes* executed in parallel, called `r0_1`, `r0_2`, `r1_1` and `r1_2`, which correspond to the four RHBs in Figure 4. In particular, they correspond to the two instances of Rule `R0` and `R1` presented in Figure 3. The committing part of the PHB is carried out by the *variable* `flag`, which gives a priority to the PHB outputs assignment. Finally, the *process* `finish_p` is charged to rise the finish signal, when the output signals cannot be further modified by the other *processes*.

```
entity gcd is
    Port ( clk : in  STD_LOGIC;
           reset : in STD_LOGIC;
           gcd1 : in  STD_LOGIC_VECTOR (7 downto 0);
           gcd2 : in  STD_LOGIC_VECTOR (7 downto 0);
           gcd_out1 : out  STD_LOGIC_VECTOR (7 downto 0)
                                             := X"00";
           gcd_out2 : out  STD_LOGIC_VECTOR (7 downto 0)
                                             := X"00";
           valid1 : out STD_LOGIC;
           valid2 : out STD_LOGIC;
           valid_out1 : out STD_LOGIC := '1';
           valid_out2 : out STD_LOGIC := '1';
           finish : out STD_LOGIC := '0');
end gcd;

architecture Behavioral of gcd is
  signal gcd1_sig : std_logic_vector (7 downto 0)
                                             := X"00";
  signal gcd2_sig : std_logic_vector (7 downto 0)
                                             := X"00";
  signal valid1_sig : std_logic := '1';
  signal valid2_sig : std_logic := '1';
  signal finish_sig : std_logic := '0';
  signal finish_sig_reg : std_logic := '0';
  signal gcd1_sig_reg : std_logic_vector (7 downto 0)
                                             := X"00";
  signal gcd2_sig_reg : std_logic_vector (7 downto 0)
                                             := X"00";
  shared variable flag : std_logic := '0';
  shared variable finish_flag : boolean := false;
begin
  r1_1: process (clk, reset, gcd1, gcd2, gcd1_sig,
                gcd2_sig, valid1_sig, valid2_sig)
  begin  -- process r1_1
    if reset = '1' then
      gcd2_sig <= gcd2;
    elsif (clk'event and clk='1') then
      if (valid1_sig='1' and valid2_sig='1') then
        if gcd2_sig>=gcd1_sig then
          gcd2_sig <= gcd2_sig - gcd1_sig;
          flag := '1';
        else
          flag := '0';
        end if;
      end if;
    end if;
  end process r1_1;

  r1_2: process (clk, reset, gcd1, gcd2, gcd1_sig,
                gcd2_sig, valid1_sig, valid2_sig)
  begin  -- process r1_2
    if reset='1' then
      gcd1_sig <= gcd1;
    elsif (clk'event and clk='1') then
      if (valid1_sig='1' and valid2_sig='1') then
        if flag='0' then
          if gcd1_sig>=gcd2_sig then
            gcd1_sig <= gcd1_sig - gcd2_sig;
          end if;
        end if;
      end if;
    end if;
  end process r1_2;

  r0_1: process (clk, reset, gcd1, gcd2, gcd1_sig,
                gcd2_sig, valid1_sig, valid2_sig)
  begin  -- process r0_1
    if reset = '1' then
      valid1_sig <= valid1;
    elsif (clk'event and clk='1') then
      if gcd1_sig=X"00" then
        valid1_sig <= '0';
      else
        valid1_sig <= '1';
      end if;
    end if;
  end process r0_1;

  r0_2: process (clk, reset, gcd1, gcd2, gcd1_sig,
                gcd2_sig, valid1_sig, valid2_sig)
  begin  -- process r0_2
    if reset='1' then
      valid2_sig <= valid2;
    elsif (clk'event and clk='1') then
      if gcd2_sig=X"00" then
        valid2_sig <= '0';
      else
        valid2_sig <= '1';
      end if;
    end if;
  end process r0_2;

  gcd_out1 <= gcd1_sig;
  gcd_out2 <= gcd2_sig;
  valid_out1 <= valid1_sig;
  valid_out2 <= valid2_sig;
  finish <= '1' when finish_sig='1' and finish_sig_reg='0'
                  and finish_flag else
           '0';

  finish_p: process (clk, reset, gcd1_sig, gcd2_sig,
                    finish_sig)
  begin  -- process finish_p
    if reset='1' then
      gcd1_sig_reg <= X"00";
      gcd2_sig_reg <= X"00";
      finish_sig_reg <= '0';
      finish_flag := true;
      finish_sig <= '0';
    elsif (clk'event and clk='1') then
      gcd1_sig_reg <= gcd1_sig;
      gcd2_sig_reg <= gcd2_sig;
      finish_sig_reg <= finish_sig;
      if gcd1_sig=gcd1_sig_reg and
           gcd2_sig=gcd2_sig_reg then
        finish_sig <= '1';
      else
        finish_sig <= '0';
      end if;
    end if;
    if finish_sig='1' then
      finish_flag := false;
    end if;
  end process finish_p;

end Behavioral;
```