Constraint-Based Hardware Synthesis

Andrea Triossi[1], Salvatore Orlando[1], Alessandra Raffaetà[1], Frank Raiser[2], Thom Frühwirth[2]

[1] Department of Computer Science, University Ca' Foscari, Venezia, Italy
{triossi,orlando,raffaeta}@unive.it
[2] Institute for Software Engineering and Compiler Construction, Ulm University, Germany
{frank.raiser,thom.fruehwirth}@uni-ulm.de

Abstract. We propose a high-level hardware description environment which aims at reducing the gap between application design and the well-established hardware description frameworks. Our motivations rise from an explicit demand for design representation languages at a higher abstraction level with respect to the ones currently adopted by hardware system engineering. A candidate solution can be identified in the constraint programming paradigm. In particular our work investigates the possibility of synthesising special-purpose hardware devices starting from the Constraint Handling Rule formalism. Our method can be used to guide the development of a prototype source-to-source compiler capable of producing, from a constraint based expression, compliant Hardware Description Language code. This paper includes a prototype implementation that allows for efficient parallel execution of multi-set constraint rewrite rules.

1 Introduction

The traditional hardware design flow usually begins with a high level application description, goes through a Register Transfer Level (RTL) model and ends in a gate-level netlist that can be directly mapped into hardware. While the second translation (from the RTL model to the gate-level specification) is commonly taken by a synthesiser there is still no standard practice for the first translation. Hardware Description Languages (HDLs), such as VHDL [19] and Verilog [18], are a well proven and established standard for hardware design, but force the designer of Application Specific Integrated Circuits (ASICs) to think at the RTL level for which HDLs are the perfect match. In other words, HDLs are characterised by a low level of abstraction: HDL is for hardware what assembly is for software. Although silicon process technology continues to evolve at an accelerated pace, design automation technology is now seen as the major technical barrier to progress. Furthermore integrated devices may well contain several processors, memory blocks or accelerating hardware units for dedicated functions that are more related with software architecture than low level HDL representations.

The motivation of introducing a new hardware modelling language primarily relies on the changing nature of the systems under design. The standard design approach of dividing the functionalities into hardware and software has led to a firm distinction between the programming languages adopted to describe the systems. Besides the clear advantage of moving from HDL to a high level language, we should take into account the benefit introduced by a reconfigurable programming environment. Reconfigurable computing is intended to fill the gap between hardware and software, achieving potentially much higher latency related performance than software, while maintaining a higher level of flexibility than hardware [4]. In order to obtain these performance benefits, reconfigurable systems are usually formed by a combination of reconfigurable logic and a general-purpose microprocessor. The processor performs the operations that cannot be done efficiently in the reconfigurable logic, such as data-dependent control, network tasks and possibly memory accesses, while the computational cores are mapped to the reconfigurable hardware. Field Programmable Gate Arrays (FPGAs) are an instance of dynamically programmable hardware: they are devices containing programmable interconnections between logic components, called logic blocks, that can be programmed to perform complex combinational functions. Software engineers can program FPGA by using a reconfigurable programming language and the reconfigurable compiler can provide an architecture-independent developing platform. We can mention, for example, Impulse-C [20] and Mitrion-C [21] as commercial reconfigurable programming languages that increase software productivity [6].

Our goal is to synthesise hardware starting from a language at a level higher than that of the commonly used behavioural HDLs in order to let the programmer to easily focus on system behaviour rather than on low level implementation details. The design procedure identified in [10] can be applied to a declarative paradigm rather than traditional imperative languages, inheriting all the well-known benefits for the programmer. In [12], the sentence "algorithm = logic + control", gave rise to a number of logic programming languages where only the meaning of the program needs to be expressed, while the control is generated by the compiler resulting in efficient executions. We will apply these principles to hardware choosing as input language the rule-based formalism Constraint Handling Rules (CHR) [8]. Its plane and clear semantics make it suitable to be directly implemented in hardware. One of the most important advantages of CHR towards this purpose relies on the fact that it is already structured for concurrent computations, thus matching the parallel characteristics of the target gate-level hardware, because it does not provide for backtracking search but it rather employs guards that are used to commit to a single possibility without trying the other ones. Guards can be also used for synchronisation among processes solving different goals. When a goal cannot be rewritten, the process solving this goal does not fail but it is blocked until, possibly, other processes will add the constraints that are necessary to entail the guard of an applicable clause.

Clearly our work can be also used as a complement of existing well-established frameworks. Our CHR-based system for hardware specification can be exploited, for example, as a way to rapidly verify if a program is correct and then to write an effective and efficient procedural description. On the other hand we aim at finding a hardware model that can execute a software specification rather than to specify hardware design from a software description (like other declarative frameworks cited in Section 2). While an algorithm-level hardware description can often be useful, we are aware of the importance for the user to retain full control of the fine grained specifications whenever a particular need arises. Hence our framework is developed taking into account the possibility of merging standard HDL and high-level programming language (CHR) compiled into synthesizable HDL.

The remainder of this paper is organised as follows. In Section 2 several approaches to high level synthesis are illustrated. The proposed hardware implementation is described in Section 3 while the experimental results are discussed in Section 4. Section 5 draws some concluding remarks.

2 Related work

With the aim of achieving a higher level of abstraction in hardware description and bringing closer to the hardware level programming languages commonly used for software design, two major approaches were pursued in the last decade: extending hardware description languages including VHDL and Verilog, and extending programming languages including C and C++.

The first approach ultimately resulted in SystemVerilog [23] and extensions to VHDL that improve simulation performance and hardware verification, and they help the synthesis process only by adding modeling interfaces. Indeed they come with the infrastructure needed for designing advanced testbenches, such as constrained-random stimuli generation, functional coverage, and assertions, but programmers still have to own a strong hardware background if they want to use them as hardware description environment.

The second line of research led to SystemC [22], a set of C++ classes and macros, which enable a designer to simulate concurrent processes using plain C++ syntax. SystemC has semantic similarities to VHDL and Verilog, but it has a syntactical overhead compared with these languages when used as a hardware description language because it is intended mainly for specification, architectural analysis, testbenches, and behavioural design. Furthermore skepticism about the usefulness of C++ design flow is expressed in [9] where the concern about the rising gap between the models and the synthesis is pointed out. Especially block-level designers explain that C++ is not the right direction for HDL development because synthesis and verification impose much stronger requirements on the language.

On the side of functional languages we can count a large number of successful approaches to hardware description. Since the 80s one of the most popular domains in which functional languages have been extensively used is hardware design [16]. General purpose functional languages, like Haskell, have been widely used as host languages for embedding HDL [3, 13]. Other examples of declarative hardware oriented languages are Pebble [14] or Ruby [11] that support structural descriptions based on abstractions such as blocks and their interconnections. They allow the user to focus on the essential structure of the system describing parameterised design concisely thanks to features such as iterative descriptions and static recursion in the circuit design. These extensions provide simple meta-languages that help programmers to deal with complex circuits rather than using a poor structural HDL.

Logic programming and especially Prolog are used as formalisms for hardware design specification and verification as well. The work [5] illustrates how the essential requirements of a HDL are satisfied using fundamental features of Flat Concurrent Prolog and how it can overcome known disadvantages of common HDL like overloading, verbosity or the lack of composite *if* statement. More recent approaches [2, 1] present a Prolog-based hardware design environment based on a high-level structural language called HIDE+. Such language was developed with the precise purpose of filling the gap of the structural HDL languages that can deal only with small circuits. Indeed the HDL description tends to be very complex due to the need of making all the connections explicit.

3 Hardware blocks construction

In this section we discuss the main ideas behind our CHR-based hardware specification approach. We aim at translating CHR rules into VHDL behavioural model of hardware modules, which directly manipulate constraints, and which can be synthesised in a specific technology using existing logic-level synthesis tools.

3.1 The CHR subset covered

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 should be kept in the constraint store. It is not trivial to foresee the maximum number of constraints to be stored during computation. Thus 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:

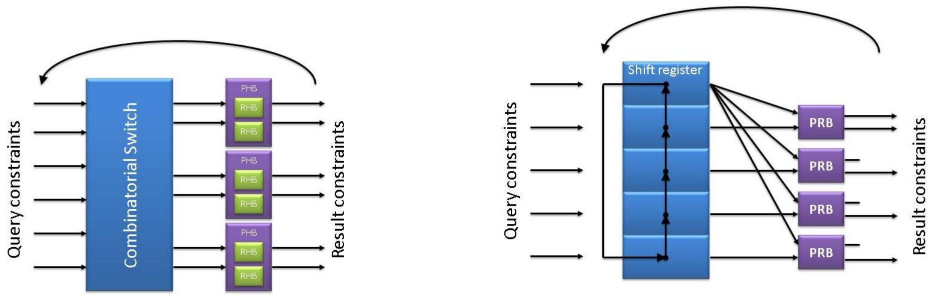
$$\begin{aligned} &rule\ @\ c_1(X_1), ..., c_p(X_p) \backslash c_{p+1}(X_{p+1}), ..., c_n(X_n) \Leftrightarrow \\ &g(X_1, ..., X_n) \mid Z_1 is f_1(X_1, ..., X_n), ..., Z_m is f_m(X_1, ..., X_n), c_{i_1}(Z_1), ..., c_{i_m}(Z_m). \end{aligned} \quad (1)$$

where X_i ($i \in \{1, \ldots, n\}$) can be a set of variables and the number of body constraints is less or equal than the number of constraints removed from the head ($m \leq n - p$) and no new type of constraints is introduced: $\{i_1, \ldots, i_m\} \subseteq \{p+1, ..., n\}$. In this way the number of constraints cannot increase and the constraint store can be bounded by the width of the initial query. Moreover, the rule is in head normal form: all the arguments in the head of the rule are variables and variables never occur more than once (all equality guards implicitly present in the head are written explicitly in the guard).

We recall that the semantics of a simpagation rule is the following: if the guard g is true, the first part of the head, $c_1(X_1), \ldots, c_p(X_p)$, is kept while the second one, $c_{p+1}(X_{p+1}), \ldots, c_n(X_n)$, is removed, and the constraints in the body $c_{i_1}(Z_1), \ldots, c_{i_m}(Z_m)$ are added to the constraint store.

Example 1. We consider, as running example, the following program which computes the greatest common divisor (gcd) between two integers using the Euclid's algorithm.

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



(a) (b)

Fig. 1. (a) Hardware design scheme (b) Optimisation model for gcd

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`.

It is clear that the number of constraints remains bounded during the computation. Indeed the first rule, if applied, removes a constraint from the store, instead the second removes a constraint and adds a new one, thus leaving the total number of constraints unchanged.

3.2 Principles of the hardware blocks

The framework we propose is logically divided into two parts:

1. Several hardware blocks representing the rewriting procedure expressed by the program rules.
2. An interconnection scheme among the blocks specific for a particular query.

The first one realizes the hardware needed to compute 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.

As depicted in Fig. 1(a) we call Program Hardware Block (PHB) a collection of Rule Hardware Blocks (RHBs) generated by each rule of the CHR program. The proposed approach considers the constraints as hardware signals and the arguments as the values that signals can assume. The initial query can be directly placed in the constraint store from which several instances of the PHB concurrently retrieve the constraints working on separate parts of the store and after computation they replace the input constraints with the new ones. A Combinatorial Switch (CS) sorts 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 details the construction of the blocks.

Rule Hardware Blocks The hardware corresponding to the CHR rule (1) has as inputs n signals that have the value of the variables $X_1 ... X_n$ (all the arguments of the head constraints). If $X_1 ... X_n$ are sets of variables we use vectors of signals (*records* in VHDL). The computational part of the RHB is given by the functions $f_1 ... f_m$ that operate on the inputs and the resulting outputs signals have the value of the variables $X_1 ... X_p$ and $Z_1 ... Z_m$.

We exploit the basic VHDL concurrent statement called *process* to translate the computational part of any rule to a sequential execution. Indeed each rule can be mapped in a single clocked *process* containing an *if* statement over the guard variables.

In order to take into account the possibility of a reduction of the number of constraints during the computation, each output signal for a given constraint is coupled with a *valid* signal that states to the following components whether to ignore the signal related to such constraint or not.

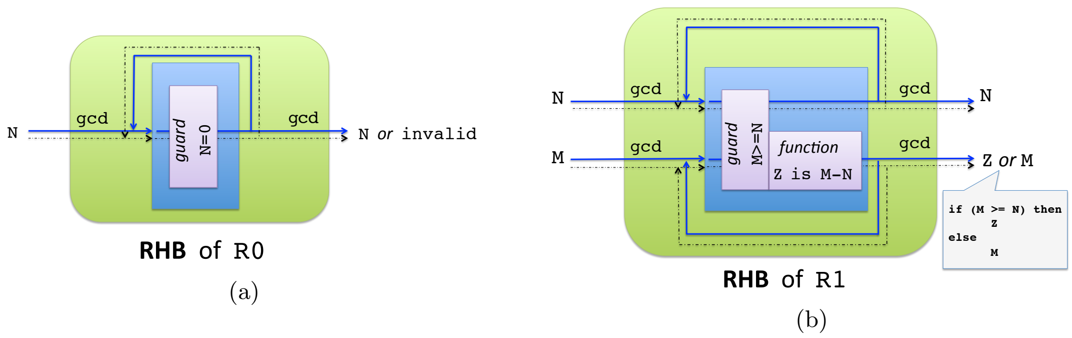


Fig. 2. The Rule Hardware Blocks for the gcd rules.

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

The block in Fig 2(a), that corresponds to R0, 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 Fig 2(b) is the translation of the second rule of gcd. It consists of four input signals, i.e. the values for the variables N and M with their *valid* signals. In this case the *valid* signals remain unchanged. If the guard holds the value of the signal for the second input constraint is replaced with Z = M-N while the value of the first one is not modified. If the guard does not hold the outputs of the block coincide with the inputs. The computational part is carried out by the subtraction operator.

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. PHB takes as input the two global input signals *clk* and *reset* used for synchronising and initialising 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 rule they stand for till the output remains unchanged for two consecutive clock cycles.

It is worth stressing 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 RHB a number of times equal to the possible permutations of the constraints of the same type. Finally we have to guarantee that only one copy of the RHB can execute per clock cycle.

Example 3. Let us consider rule R1 described in Example 1. Two instances of gcd are present in the head of the rule and hence two signals are created respectively with value N and M and they are the inputs of the RHB. Due to the guard of R1 these inputs feed a comparator that checks if the value of the second signal is greater or equal than the first. If the condition is satisfied the value of the second signal is replaced by the result of the subtractor that has as inputs the two signals. Now consider the case in which N is greater than M, the rule can fire as well because the head constraints are of the same type and so they can be swapped. For this reason PHB has to contain another copy of the RHB that executes such rule but with inputs in reverse order (see Fig. 3).

The PHB level is also used to set the rules parallelisation at the basis of the computation. As we said each rule is executed by one or more concurrent *processes* that fire synchronously every clock cycle. Therefore we exploit the notion of strong parallelism of CHR, introduced in [7], assuming

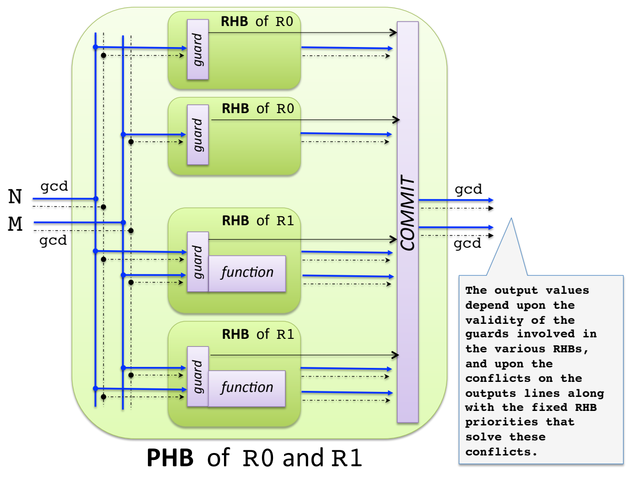

RHB of R0
guard
RHB of R0
guard
RHB of R1
guard
function
RHB of R1
guard
function
COMMIT
N
M
gcd
gcd
gcd
gcd
PHB of R0 and R1
The output values depend upon the validity of the guards involved in the various RHBs, and upon the conflicts on the outputs lines along with the fixed RHB priorities that solve these conflicts.

Fig. 3. The Program Hardware Block for the gcd program

that rules can work on common constraints at the same time if they do not rewrite them. If two rules try to change the same constraint then we cannot parallelise and we need to execute them one after the other if the latter is still applicable. According to the theoretical operational semantics [8], we can state that the provided rule application is fair since every rule that could fire does it every clock cycle or in the worst case in the subsequent cycle.

Combinatorial Switch A further level of parallelisation is achieved replicating the PHBs into several copies that operate on different parts of the global constraint store. PHBs can compute independently and concurrently because they are attempting to rewrite different constraints. Although they process data synchronously, since they share a common clock, it is not required that they terminate computation at the same time. Indeed the CS acts as synchronisation barrier letting the faster PHBs wait for the slower ones. It is also charged to manage communication among hardware blocks exchanging data between themselves: once all the PHBs have provided their results, it reassigns the output signals as input for other PHBs guaranteeing that all the permutation between them are covered. Exploiting the fact that the number of constraints cannot increase, the CS works directly on the signals coming from the PHB, but there are no impediments to retrieve the constraints from an external memory if the space capacity of the FPGA is not sufficient. In practice the implementation of this interconnection element relies on a signal switch that sorts the n query constraints according to all the possible k-combination on n (where k is the number of inputs to the single PHB) and connects them to all the inputs of the PHBs. 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:

$$S = \frac{\binom{n}{k}}{n/k} = \frac{\prod_{i=1}^{k-1} n - i}{(k-1)!}$$

Example 4. The gcd program presented in Example 1 shows that we can implement in hardware the Euclid's algorithm at the behavioural level, i.e., it describes a system in terms of what it does rather than in terms of its components and interconnections. Instead, here we illustrate an example of structural hardware design which shows the hardware granularity achievable generating the VHDL code for the basic building block of sequential circuits directly implementable in FPGA. According to the general scheme for rules representation described above, the following CHR rule implements the D flip-flop, the elementary memory block capable of storing the value of a signal:

```
d(X) \ q(_) <=> q(X).
```

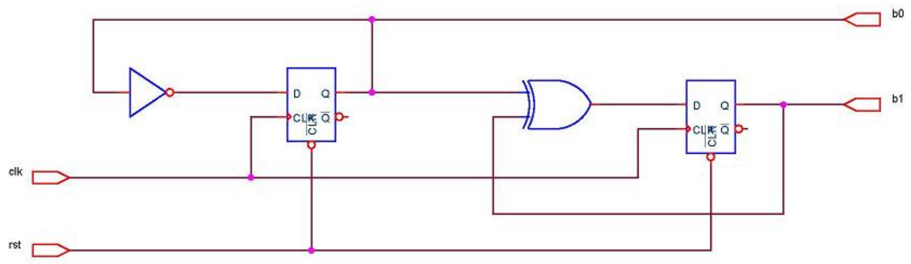


Fig. 4. 2-bit counter circuit

where `d/1` and `q/1` stand for the input and the output signals of the D flip-flop. The `q` constraint is rewritten every time a `d` constraint is present, as in the D flip-flop, every clock cycle, the value of the output is replaced by the value of the input.

By using again the idea that the removal and the introduction of the same constraint corresponds to the memory refresh we can implement more complex circuits. The following two lines code describes the hardware circuit of a two-bit counter represented in Fig. 4:

```
b0(X) <=> Z is (not X), b0(Z).
b0(X) \ b1(Y) <=> Z is (X xor Y), b1(Z).
```

where `not` and `xor` are operators predefined in HDL (built-in) that are used to implement the combinatorial logic part of the circuit.

4 Experimental results

We use the automatically generated hardware blocks described in Section 3 to implement in FPGA the running example presented in Example 1 to find the greatest common divisor at most of 128 integers. The resulting hardware design relies on 64 PHBs deriving in parallel the gcd while 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 behavioural VHDL using a knockout system where we compute the gcd in parallel of 64 pairs then of 32 and so on. Both hardware specifications are then synthesised and simulated with ISim the Xilinx ISE simulator at 100MHz reference clock frequency. Fig. 5 reports the execution times for 16, 32, 64 and 128 1-byte integers. The two FPGA implementations are labelled respectively as FPGA (CHR) and FPGA (VHDL). The curve labelled CPU refers to the computational time of the CHR gcd program running on Intel Xeon 3.60GHz processor with 3GB of memory. It is displayed just for an order of magnitude reference since we cannot compare them due to the completely different hardware nature.

The plot clearly shows how the execution time can increase to more than an order of magnitude with respect to the VHDL solution. This effect is primarily due to the fixed nature of the CS that can not reduce the number of possible combinations when the number of constraints decreases. In Section 4.1 we address this issue suggesting an optimisation for rules that have the property of strong parallelism like in the gcd case. The outcome of such optimisation is also reported in Fig. 5 (labelled as FPGA (CHR) Opt.) and it exhibits a relevant reduction of the execution time. Notice that the VHDL implementation leads to an execution time almost constant due to the complete parallelism achievable by hardware. We do not observe a super-linear trend in our implementation like the one noticed in [17] because the derivations of each pair of gcd constraints do not interfere each other. Finally we should notice that the resulting highest frequencies of operation are all above 250 MHz and up to 350 MHz, which is quite good for a non pipelined architecture.

Analogous results are obtained on a different example, namely the implementation of Floyd-Warshall algorithm aimed at finding the length of the shortest paths between all pairs of vertices in a weighted graph. A procedural version of the algorithm is the following:

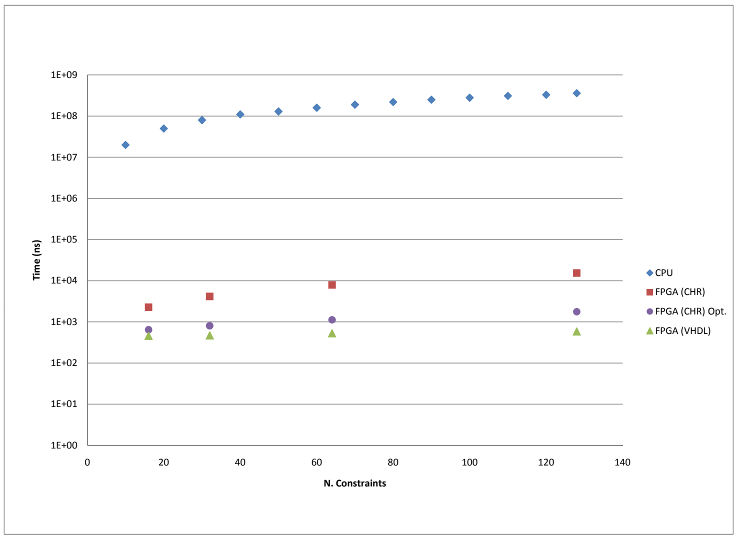


Fig. 5. Gcd execution time (log scale)

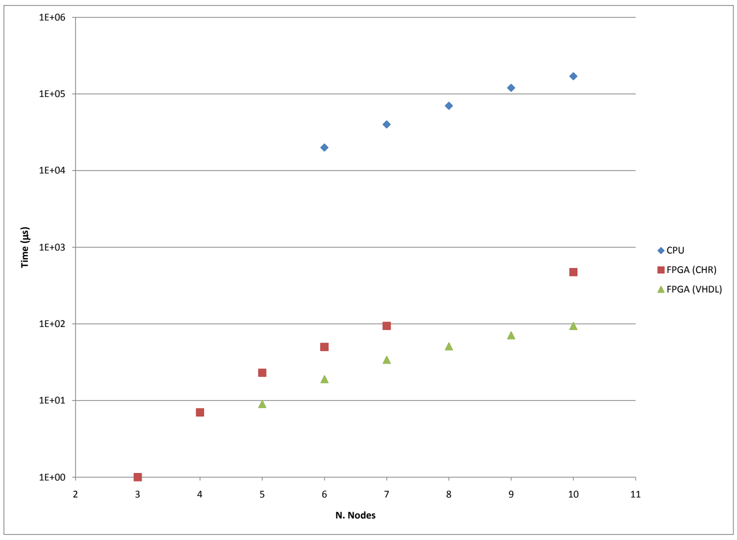


Fig. 6. Floyd-Warshall execution time (log scale)

```
1  for k=1 to N
2    for i=1 to N
3      for j=1 to N
4        $d_{i,j}$=min($d_{i,j}$,$d_{i,k}$+$d_{k,j}$)
```

where $d_{i,j}$ are the elements of the matrix representing the graph. In CHR the algorithm can be expressed as a simple rule with three constraints in the head standing for three edges that should be taken into account for the minimum computation:

```
edge(I,K,D1), edge(K,J,D2) \ edge(I,J,D3) <=>
            D3>D1+D2 | D4 is D1+D2, edge(I,J,D4).
```

From such rule our method generates a simple PHB that has as inputs and outputs three terns of signals that are respectively: the source, the destination and the weight of the edges. Depending on the query dimension n a CS with $\frac{1}{2}(n-1)(n-2)$ states assigns the constraints to $\lfloor \frac{n}{3} \rfloor$ PHBs. In Fig. 6 we compare our implementation with the VHDL based one described in [15] derived by a logic-level specification. As we can see from the plot, our implementation exceeds the best handcrafted design only by less than one order of magnitude and at the same time it delivers a high degree of flexibility: you can manipulate a one-line code rather than rearrange a fixed architecture of hundreds of lines.

4.1 Optimisation

Besides the general framework described in Section 3 we want to propose an optimisation in order to speed up the computation in presence of algorithms that considerably reduce the number of constraints during computation. In such cases it is worth noting that a CS, that simply combines all the constraints in all the possible combinations, is highly inefficient. In fact many constraints that are marked as not *valid* by the PHBs still continue to be shuffled by the CS uselessly. To face this issue we rely on the possibility of exploiting the strong parallelism property directly on the whole constraint store and not only on portion of it like the PHBs do. We need a new hardware block charged to combine, in parallel on several PHBs, the kept constraints with different sets of removed constraints. An example of such device can be provided optimising the CS obtained by the implementation of the gcd rules. Fig 1(b) shows a possible implementation for a five constraints query but the design can be easily increased linearly with the number of constraints. It relies on a circular shift register preloaded with the query constraints and with one cell connected to all the first input (kept constraint) of the PHBs and all the others connected to the second input (removed constraint) of each PHBs. 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 (we skip the steps with a not *valid* constraint in the first position). Using this topology there is no need to implement multiple instances of the same rule at the PHB level (see Section 3.2): indeed now the order of the input constraints matters because one is the kept `gcd` and the other is the removed one. As consequence, apart from the first PHB, the output carrying the kept constraint can be left disconnected because it refers always to the same constraint.

5 Conclusion

We described the general outline of an efficient hardware implementation of a CHR subset able to comply with the restricted bounds that hardware imposes. The level of parallelisation achieved provides a time efficiency comparable with that obtained with a HDL design. At the same time, the proposed solution offers a more general framework reusable for a wide range of tasks. Additionally, it was shown that, applying the same hardware generation technique to CHR with HDL built-in operators, we obtain elementary hardware blocks that can be easily integrated with existing HDL code.

Further optimisations applicable also to problems where the number of constraints does not necessarily decrease during the computation will be object of future works; however an important challenge would be the hardware implementation of complex programs as well. A general treatment of rules dependency 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. These studies can eventually open the doors to the production of a source-to-source compiler that takes as input CHR and carries out a structural hardware description in HDL ready for a synthesis tool.

References

1. A. Benkrid, K, Benkrid, HIDE+: A Logic Based Hardware Development Environment, Vol. 16, Issue 3, 2008.
2. 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, Vol. 12, No. 4, pp. 420-436, 2004.
3. P. Bjesse, K. L. Claessen, M. Sheeran and S. Singh, Lava: Hardware design in Haskell. In Proc. of International Conference on Functional Programming (ICFP), ACM SIGPLAN, 1998.
4. K. Compton, S. Hauck, Reconfigurable Computing: A Survey of Systems and Software, ACM Computing Surveys, Vol. 34, No. 2, June 2002, pp. 171-210.
5. Y. Dotan and B. Arazi, Concurrent Logic Programming as a Hardware Description Tool, IEEE Transaction on Computers, Vol. 39, No. 1, pp. 72-88, 1990.
6. E. El-Araby, M.vTaher, M. Abouellail, T. El-Ghazawi, G.B. Newby, Comparative Analysis of High Level Programming for Reconfigurable Computers: Methodology and Empirical Study, 3rd Southern Conference on Programmable Logic, pp.99-106, 2007.
7. T. Frühwirth, Parallelizing Union-Find in Constraint Handling Rules Using Confluence, 21st Conference on Logic Programming ICLP, October 2005.
8. T. Frühwirth, Constraint Handling Rules, Cambridge University Press, 2009.
9. R. Gupta, G. Berry, R. Chandra, D. Gajski, K. Konigsfeld, P. Schaumont, and I. Verbauhede, The next HDL (panel session): if C++ is the answer, what was the question?, in Proceedings of the 38th conference on Design automation, pp. 7172, ACM Press, 2001.
10. G. Hu, S. Ren, X. Wang, A comparison of C/C++-based Software/Hardware Co-design Description Languages, IEEE proceeding of The 9th International Conference for Young Computer Scientist, pp 1030-1034, 2008.
11. G. Jones and M. sheeran,, "Circuit design in Ruby", Formal Methods for VLSI Design, J. Staunstrup ed., North-Holland, 1990, pp. 13-70.
12. R. A. Kowalski, Algorithm = Logic + Control, Comm. ACM, August 1979.
13. J. Launchbury, J. R. Lewis and B. Cook, On embedding a microarchitectural design language within Haskell. SIGPLAN Not., 34(9):60-69, 1999.
14. W. Luk and S.W. McKeever, Pebble: a language for parametrised and reconfigurable hardware design, in Field-Programmable Logic and Applications, LNCS 1482, Springer, 1998.
15. Sui-Tung Mak and Kair-Pui Lam, Serial-Parallel Tradeoff Analysis Of All-Pairs Shortest Path Algorithms In Reconfigurable Computing, IEEE proceeding of FTP 2002, pp. 302-305, 2002.
16. M, Sheeran, Hardware design and functional programming: a perfect match, Journal of Universal Computer Science, 11(7):11351158, 2005.
17. M. Sulzmann and E. S. L. Lam, Parallel Execution of Multi Set Constraint Rewrite Rules, Proc. of 10th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP08) Valencia Spain, July 2008.
18. IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language, 1996. http://www.ieee.org/.
19. IEEE Standard VHDL Language Reference Manual, 1994, http://www.ieee.org/.
20. Impulse C, Impulse Accelerated Technologies, Inc., http://www.impulsec.com/.
21. Mitrion-C, Mitrionics, Inc., http://www.mitrionics.com/.
22. Open SystemC Initiative, SystemC version 2.0 Users's Guide, Technical report, 2001.
23. SytemVerilog 3.1 - Accelleras Extensions to Verilog(R), Accellera Organization Inc., 2003.