CCHR: the fastest CHR Implementation, in C

Pieter Wuille, Tom Schrijvers*, and Bart Demoen

Department of Computer Science, K.U.Leuven, Belgium
pieter@wuille.biz,{tom.schrijvers,bart.demoen}@cs.kuleuven.be

Abstract. CHR is usually compiled to high-level languages (like Prolog) that make it hard or impossible to express low-level optimizations. This is a pity, because it confines CHR to be a prototyping language only, with an unacceptable performance for production quality software.
This paper presents CCHR, a CHR system embedded in the C programming language, that compiles to low-level C code which is highly suitable for fine-grained performance improvements.
In this way CCHR program performance comes close to matching that of native C, and easily outperforms other CHR implementations.

1 Introduction

Constraint Handling Rules (CHR) [4] is a high-level, declarative language, originally designed for the implementation of user-defined, application-tailored, constraint solvers in a given host language. Nowadays CHR is increasingly being used as a general programming language for a wide range of applications — multi-agent systems, type systems and natural language processing, to name a few — and for the study of algorithms.

Since its conception in 1991, CHR systems have been developed for several host languages, most notably for Prolog and other declarative languages. The first full CHR system was developed by Christian Holzbaur and Frühwirth [6]. Currently, the main advanced CHR system for Prolog is the K.U.Leuven CHR system [12], available for a number of Prolog implementations. Other systems have been developed for HAL [2] and Haskell, e.g. [9].

As far as we know, the only imperative programming language for which CHR systems have been made is Java [1,17,16]. The compilation of CHR to Java allows for more efficiency through in place updates. However, Java lacks fine-grained control over low-level data structures, which prevents further optimization.

The C programming language was designed in 1972 as an imperative procedural language that could easily be translated into machine code. After many standardizations (K&R C, ANSI C, ISO C, C99), it is still heavily used. Through the use of a standardized preprocessor and usage of (platform specific) system headers, C source code can be portable, while having very system-specific features like pointers (providing direct memory access). The combination of these two properties makes C the target language of choice for compiling many higher-level languages.

* Post-doctoral researcher of the Fund for Scientific Research - Flanders.

In the footsteps of many other languages, we consider the suitability of C as a target language for CHR. Our contributions are:

1. CCHR: the first integration of CHR with the C language (Section 3),
2. a C library for logical variables useful for porting CHR programs from Prolog (Section 4),
3. a compilation scheme from CCHR to C code (Section 5), and
4. an implementation of CCHR that outperforms currently available CHR systems and comes very close to native C code, as our benchmarks show (Section 6).

For the rest of this text, we will assume familiarity with both CHR and C.

2 Overview

Before we dive into the technical details of the CCHR language and its compiler, we take a moment to reflect on the design principles and setting for CCHR.

Efficiency It should be clear by now that the primary design goal for CCHR is:

> 1. CCHR is an efficient system.

For this purpose we must borrow efficiency ideas from both the CHR and the C world. The former gives us a good high-level basis to start from: the refined operational semantics [3] and related optimizations [7,11]. The latter provides us many low-level tricks and techniques: macros for heavy inlining, lean data structures using pointers/arrays/bitwise operations, ...

Familiarity As its secondary goals we require that CCHR is close to other CHR systems:

> 2a. CCHR closely resembles other CHR systems.

This should allow CHR programmers to quickly switch from their current CHR system to CCHR, and existing programs to be easily ported. This is in the first place achieved by a familiar syntax and operational semantics. For the latter, the previously mentioned refined operational semantics is again a good choice: it is implemented by many other systems, and serves as a standard.

Example 1. The following CCHR program for computing the greatest common divisor illustrates point 2a. If you are familiar with CHR in Prolog, the code below should look familiar.

```
1 cchr {
2   constraint gcd(int);
3
4   triv @ gcd(0) <=> true;
5   dec  @ gcd(N) \ gcd(M) <=> M>=N | gcd(M-N);
6 }
```

Of course we should also be considerate of the host language:

2b. CCHR is tightly integrated with C.

This design principle means that C programmers should be able to incorporate CHR into their programs with a minimum of fuss. This means that CHR and C code can be mixed freely, constraints range over native C data types and, where possible, the underlying principles of the C language are respected. We should be particularly apprehensive of *programmer-managed* memory. Previous CHR implementations have largely neglected the issue of memory management (except for [15]), relying on automatic garbage collection. CCHR however cannot side-step this issue; it must duly free up any memory it allocates. But there's more: CCHR must actively support programmers in freeing any memory they allocate themselves and then hand over to CCHR for use.

3 The CCHR Language

The CCHR language consists of a *syntax* for writing embedded CHR programs in C, and a *runtime system* for invoking the CHR program from within C.

3.1 Syntax

The syntax of CCHR was heavily influenced by that of K.U.Leuven JCHR: a good compromise between the well-known Prolog CHR syntax and that of the host-language. We introduce the CCHR language with a small example program, the bottom-up computation of Fibonacci numbers listed in Figure 1.

Like JCHR, the CHR code is contained in a block, the `cchr` block (lines 6-13). However, unlike JCHR, this block does not sit in a file of its own, but can be embedded in a C program with the usual C definitions, e.g. a `main` function (lines 15–25).

Within the `cchr` block we have two kinds of elements: constraint declarations (line 7) and CHR rules (lines 9–12).

A constraint declaration is the keyword `constraint` followed by one or more constraint specifiers (line 7). A *constraint specifier* consists of the constraint name followed by its argument types [1] and, optionally, one or more options for customizing the constraint behavior. Options include:

- `option(fmt,`*`Fmt`*`,`*`FmtArgs`*`)`: a C `printf` format string and its arguments for customizing the pretty-printing of the constraint,
- `option(init,`*`Code`*`)`: code to run when a new constraint is initialized,
- `option(destr,`*`Code`*`)`: code to run when a constraint is destroyed,
- `option(add,`*`Code`*`)`: code to run when a constraint is stored, and
- `option(kill,`*`Code`*`)`: code to run when a constraint is removed from the store.

[1] The types are obligatory, as C is a statically typed language.

CHR rules follow closely the Prolog CHR syntax, notably exceptions being:

- A rule ends in a semi-colon as is usual for C (rather than a period).
- The keyword `alt` declares alternative formulations of a guard, and allows the CHR compiler to choose the form that is most efficient for indexing. Consider `alt(N2==N1+1,N2-1==N1)` in line 11. Given `N1` we can compute the `N2` to look up with the first form, and vice versa with the second form.
- Local variable definitions and arbitrary C statement blocks are allowed in guards and bodies.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  #include "fib_cchr.h" /* generated header file */
5
6  cchr {
7    constraint upto(int), fib(int,int);
8
9    begin @ upto(_) ==> fib(0,1), fib(1,1);
10   calc  @ upto(Max), fib(N2,M2) \ fib(N1,M1) ==>
11             alt(N2==N1+1,N2-1==N1), N2<Max |
12             fib(N2+1, M1+M2);
13 }
14
15 int main(int argc, char **argv) {
16   cchr_runtime_init();
17   cchr_add_upto_1(90);
18   cchr_consloop(j,fib_2,{
19     printf("fib(%i,%i)\n",
20       cchr_consarg(j,fib_2,1),
21       cchr_consarg(j,fib_2,2));
22   });
23   cchr_runtime_free();
24   return 0;
25 }
```

Fig. 1. Bottom-up Fibonacci in CCHR

3.2 The CCHR Runtime

The CHR constraints defined in a `cchr` block are meant to be called from within C code. For this purpose a number of runtime functions are provided in a header file (line 4), which is generated by the CCHR compiler.

Firstly, before any of the other runtime functions can be used, the CCHR runtime has to be initialized with the `cchr_runtime_init()`. This function allocates and initializes the constraint store and related datastructures. Similarly, the last runtime function to be called is `cchr_runtime_free()` which frees again all allocated memory.

New constraints can be added with:

`void cchr_add_`⟨*constraint*⟩`_`⟨*arity*⟩`(`⟨*arg1*⟩`,`⟨*arg2*⟩`,...);`

At any time, the CHR constraint store can be inspected. For this purpose we have the function:

`cchr_consloop(`⟨*var*⟩`,`⟨*constraint*⟩`_`⟨*arity*⟩`,`⟨*code*⟩`)`

where ⟨*code*⟩ is the C code to execute for instances of ⟨*constraint*⟩/⟨*arity*⟩. Within ⟨*code*⟩ the following macro may be used to refer to the ⟨*i*⟩th argument of the constraint instance:

`cchr_consarg(`⟨*var*⟩`,`⟨*constraint*⟩`_`⟨*arity*⟩`,`⟨*i*⟩`)`

For example, lines 18–22 print all the `fib/2` constraints in the constraint store.

Finally, when any of a constraint instance's arguments are changed, the programmer may manually reactivate that instance with one of:

```
cchr_reactivate_all();
cchr_reactivate_all_⟨constraint⟩_⟨arity⟩();
cchr_reactivate_⟨constraint⟩_⟨arity⟩(cchr_id_t PID);
```

3.3 An Advanced Example

Fibonacci numbers grow large very quickly, overflowing the standard `int` integer representation of C, or even larger `long` or `long long` datatypes that may be present. Figure 2 shows how we can adapt the previous program to use the unbounded integer representation of the GNU Multi-Precision library (GMP). We include the GMP header file (line 1) and define a type `big_int_t` that contains a pointer to a *big integer* (line 3). The second argument f `fib` is now declared to be of this type (line 7). When a `fib` constraint is destroyed (e.g. when the constraint store is freed), the memory allocated for the big integer should be freed as well (line 8).

The arithmetic on Fibonacci numbers in the rules now has to be performed with GMP procedures. For this purpose we first declared local variables (line 11, 16), and then call the appropriate procedures in C blocks (line 12, 17).

4 Logical Variables

Many CHR programs written for Prolog make heavy use of logical variables. When porting these programs to CCHR we are confronted with the lack of logical variables in C. In order to overcome this issue, we provide our own library of logical variables for C. This library is independent of CCHR; but it provides all the necessary ingredients to couple the logical variables with CCHR—or another constraint solver.

```
#include <gmp.h>

typedef struct { mpz_t v; } big_int_t;

cchr {
  constraint upto(int),
             fib(int,bigint_t) i
                option(destr,{mpz_clear($2.v);});

  begin @ upto(_) ==>
        bigint_t X=, bigint_t Y=,
        { mpz_init_set_si(X.v,1); mpz_init_set_si(Y.v,1) ;},
        fib(0,X), fib(1,Y);
  calc  @ upto(Max), fib(N2,M2) \ fib(N1,M1) <=>
              alt(N2==N1+1,N2-1==N1), N2<Max |
              big_int_t Sum=,
              { mpz_init(Sum.v); mpz_add(Sum.v,M1.v,M2.v); },
              fib(N2+1, Sum);
}
```

Fig. 2. Fibonacci revisited with the GNU Multiple-Precision library

4.1 Definition

Our generic library is defined in terms of macros, a primitive substitute for parametric polymorphism.

A new type of logical variable type ⟨*logical*⟩ is defined with:

`logical_header(`⟨*value*⟩`,`⟨*attribute*⟩`,`⟨*logical*⟩`)`

where ⟨*value*⟩ is the type of the values that can be assigned to the logical variable. The type ⟨*attribute*⟩ is the type of the attribute data that can be attached to a logical variable, similar to attributed variables in Prolog [5].

The operations ⟨*logical*⟩`_testeq` and ⟨*logical*⟩`_seteq` are generated for comparing and equating two logical variables, and ⟨*logical*⟩`_setval` assigns a value to a logical variable.

Example 2. The program is an excerpt of the well-known `leq` CHR program ported to CCHR:

```
logical_header(int,...,lint)

cchr {
  constraint leq(lint,lint);

  leq(X,Y) <=> lint_testeq(X,Y) | true;
```

```
7   leq(X1,Y1), leq(Y2,X2) <=>
8         lint_testeq(X1,X2),lint_testeq(Y1,Y2) |
9         {lint_seteq(X1,Y1);};
10  }
```

4.2 CCHR Integration

In the K.U.Leuven and other Prolog-based CHR systems, logical (attributed) variables are tightly integrated with the CHR program in two ways:

1. Whenever a variable is unified with either another variable or with a value, all constraints on that variable are reactivated.
2. An index is stored in a variable's attribute to quickly lookup all constraints on that variable for a multi-headed rule match.

For CCHR we exploit our logical variables in a similar way.
For the ⟨*attribute*⟩ we define a list of constraints (in pseudo-code):

```
struct {
  constraint element;
  leq_list*  tail;
} leq_list;
```

From the CCHR side, this list is kept up to date with the `add` and `kill` options:

```
constraint leq(int,int)
            option(add,{ leq_list* list = new leq_list;
                         list->element  = $0;
                         list->tail     = get_attr($1);
                         set_attr($1,list);
                         ... /* same for 2nd arg */
                       })
            option(kill,...);
```

The logical variables side provides a call-back mechanism to react on changes of the variables. Unlike Prolog's single call-back function, we make a distinction between two cases:

- ⟨*logical*⟩`_setval_callback(`⟨*var*⟩`,`⟨*val*⟩`)` is called when a logical variable is assigned a value.
  For CCHR we define this callback to reactivate all the constraints stored in the attribute of ⟨*var*⟩.
- ⟨*logical*⟩`_seteq_callback(`⟨*var1*⟩`,`⟨*var2*⟩`)` is called when two logical variables are equated.
  For CCHR we define this to first merge the attributes of ⟨*var1*⟩ and ⟨*var2*⟩, before reactivating the constraints.

The list in the attribute is also used by CCHR for lookup of constraints on a logical variable. This is usually more efficient than iterating through the whole constraint store.

Currently, the integration is done by hand, but it can easily be automated.

5 The CCHR Compiler

In this section we provide an overview of the CCHR compiler. After a general discussion of the general compiler architecture, we discuss two interesting aspects of the C back-end in more detail: the CHR assembler language and the constraint store data structures.

5.1 The Compiler Architecture

The compilation of CCHR is a staged process. Here we provide a brief overview of the subsequent stages. An overview can be found in Figure 3.

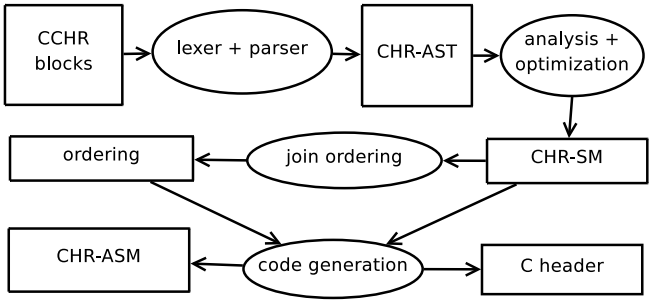

Fig. 3. The compiler architecture

We start from a C program interleaved with `cchr` blocks.

1. The `cchr` blocks are extracted from the program for further processing, while the rest of the program is left untouched.
2. The `cchr` blocks are transformed into a *CHR abstract syntax tree* (CHR-AST), using a Bison parser on top of a Flex lexer.
3. The CHR-AST is transformed into a *CHR semantic model* (CHR-SM), which is much more suitable for performing program analysis, transformation and code generation. The transformation involves a.o.
   - Identifiers are classified into constraints, variables, options, ...
   - The occurrences of constraints are determined.
   - CHR rules are transformed into *head normal form*.
   - Variable dependencies between head constraints are determined (for join ordering).

4. The CHR-SM is optimized using many of the well-known optimizations [7,11], e.g.: late storage optimization, join ordering and indexing.
5. Code is generated from the CHR-SM. Rather than generating low-level C code directly, we generate *CHR assembler* instructions (CHR-ASM), which is much closer to our problem domain.
6. The CHR-ASM code, together with C marco definitions for the assembler instructions, is merged again with the original C program.
7. The C preprocessor is invoked for expanding the CHR-ASM instructions into C code.
8. The C compiler is invoked to generate a binary executable.

5.2 The CHR assembler language

It is customary in compiler design to decouple the back-end code generation from the core of the compiler. This is were the CHR-ASM language fits in; it's as fine-grained as necessary, but not more. Every key concept in the low-level execution of CHR maps more or less directly on a single instruction. This makes code generation a fairly straightforward matter.

In the setting of CCHR we have conveniently avoided writing a back-end compiler from CHR-ASM to C code. By defining the assembler instructions as C macros, the C macro preprocessor takes care of this work for us. By replacing one set of macro definitions with another, we obtain a different behavior, without affecting the rest of the compiler. For example, the CCHR debug mode has been implemented in this way: C code for debug messages[2] is included in the macro definitions.

Although we lack the space for a full overview of the CHR-ASM language here, the following example should give a good impression.

Example 3. Figure 4 lists the CHR-ASM code for the first occurrence of the `fib` constraint of Figure 1.[3]

First, the arguments of the active constraints are read into local variables `N1` and `N2` (lines 1–2).

For the first lookup, a variable `K2` is declared that ranges over the `fib` constraints in the store, via the index on the first argument (line 3). The first argument of the constraints `K2` , should be `N+1` (line 4). Now we start the iteration (line 5), but ignore the active constraint (line 6). In the loop, we read the arguments of `K2` into local variables `N2,M2)` (lines 7–8).

For the second lookup, a nested loop is created over all instances `K1` of the `upto` constraint, not using an index (line 9). As with the other constraints, the argument of `K1` is read into local variable `Max` (line 10). If the guard succeeds (line 11), the active constraint is killed (line 12) and the body executed (line 13). After the execution of the body, we call the (optional) destructor of the killed constraint (line 14) and return from the occurrence (line 15).

---

[2] This is were the `fmt` option is used.
[3] For readability parentheses and commas have been omitted.

```
1  IMMLOCAL int N1 ARG(fib_2,arg1)
2  IMMLOCAL int M1 ARG(fib_2,arg2)
3  DEFIDXVAR fib_2 idx1 K2
4  SETIDXVAR fib_2 idx1 K2 arg1 LOCAL(N1)+1
5  IDXLOOP fib_2 idx1 K2
6    IF DIFFSELF(K2)
7      IMMLOCAL int N2 LARG(fib_2,K2,arg1)
8      IMMLOCAL int M2 LARG(fib_2,K2,arg2)
9      LOOP upto_1 K1
10       IMMLOCAL int Max LARG(upto_1,K1,arg1)
11       IF LOCAL(N2)<LOCAL(Max)
12         KILLSELF fib_2
13         ADD fib_2 LOCAL(N2)+1 LOCAL(M1)+LOCAL(M2)
14         DESTRUCT fib_2 LOCAL(N1) LOCAL(M1)
15         END
```

Fig. 4. CHR-ASM code for the fist occurrence of `fib`

5.3 Runtime Data Structures

The two main data structures we use in CCHR are:

- **doubly linked lists:** for the constraint store, and
- **hashtables:** for constraint store indexes and propagation histories.

Doubly Linked Lists The global CHR constraint store consists of one doubly linked list for every constraint symbol in the program. Every unindexed lookup of a constraint iterates through the appropriate list, which takes $\mathcal{O}(n)$ time where n is the number of instances. Because the list is doubly linked, we can both insert and remove instances cheaply and in $\mathcal{O}(1)$ time.

Our doubly linked lists are implemented on top of arrays, for reasons of memory locality. Each element in the array is a C struct containing the indexes of the previous and next element, as well as the actual payload: the constraint instance. As we have to do our own memory management, the array actually hosts two linked lists: one for the constraint instances, and one so-called *freelist* for the empty array entries. Array entries move between the two lists as instances are added and removed. When the array fills up (and the freelist empties) completely, a bigger array is allocated and all elements are copied. For that reason, we avoid direct C pointers into the array, but index into the array with position numbers.

Hashtables Hashtables are the key to the efficient execution of multi-headed rules [13]. Not only do they provide us with the same (though now amortized) $\mathcal{O}(1)$ addition and removal as linked lists, but also and more importantly lookup takes only constant time[4].

[4] Provided that we have the appropriate key.

The two main challenges for a decent hashtable implementation are *collisions* and the *hash function*. For the latter we adopted the public domain *lookup3* algorithm [8].

Collisions occur when distinct keys hash to the same entry in the table. The three main options for resolving a collision are:

1. We increase the table size to avoid the collision.
2. We attach an overflow bucket to the entry.
3. We put the new element in the next free entry in the table.

However, neither of these options is very satisfactory for high performance CHR. The first two options require frequent allocation of new tables (1) or overflow buckets (2). The third option leads to complicated lookup and removal procedures. Hence, CCHR turns to a forth option: *cuckoo hashing*. Cuckoo hashing employs two tables (A and B) with independent hash functions. A new element x is inserted at its appropriate entry in table A. If that entry was previously occupied by y, then y has to move to table B. This may cause z to move from B to A, and so on. For a full description of cuckoo hashing we refer to [10].

6 Evaluation

In this section we evaluate the performance of CCHR on a number of existing CHR benchmarks. We compare with two other CHR implementations, K.U.Leuven CHR in SWI-Prolog (v5.6.17) and K.U.Leuven JCHR (v1.5.1) in Java (1.5.0.11), as well as with a direct translation of the benchmarks in C (GCC 4.1.2).

Our benchmark suite consists of

- `gcd`: greatest common divisor algorithm by Euclid (see Section 2),
- `fib`: bottom-up Fibonacci with GMP (SWI,CCHR,C) or `java.math.BigInteger` (Java) (see Figure 2),
- `primes`: the prime number sieve of Erathostenes,
- `tak`: the Takeuchi function with tabulation,
- `leq`: a cycle of partial order constraints, and
- `ram`: a RAM simulator running a *count down from N* program.

All benchmarks were run on an AMD Athlon64 3500+ CPU with 1GB of RAM, running Linux 2.6.19 with a low load.

Table 1 lists the geometric averages of all six benchmarks for different problem sizes. The averages for C have been set to 1 and those of the other systems have been scaled relatively.

These results show that CCHR is easily two to three orders of magnitude faster than existing CHR systems for Prolog and Java. The gap with hand-written C code is for five out of six of the benchmarks quite small, less than one order of magniute. However, the `ram` benchmark shows that further improvement of CCHR is still possible and necessary.

benchmarks	**SWI-Prolog**	**JCHR**	**CCHR**	**C**
`gcd`	22,000	-	3.4	1
`fib`	21,000	940	8.5	1
`primes`	310	490	6.9	1
`tak`	210	110	4.3	1
`leq`	1,100	440	9.8	1
`ram`	4,700	11,000	120.0	1

Table 1. Relative Geometric Averages

Let us consider the two extremes, `gcd` and `ram`, in more detail. Firstly, the results for `gcd` are depicted in Figure 5. The benchmark computes `fib(5,1000`$\times n$`)` where n is the problem size. In other words, it consists of a loop that counts down from $1000 \times n$ in steps of 5. No timings have been recorded for JCHR, because the benchmark causes the Java stack to overflow, even for small values of n. CCHR does not suffer from these stack overflow problems, but manages to compile the program down to a tight loop. Hence, CCHR's performance is consistenly close to that of C. SWI-Prolog's stack behavior is similarly good, but it suffers from a "fat" loop which is four orders of magnitude slower than either C or CCHR.

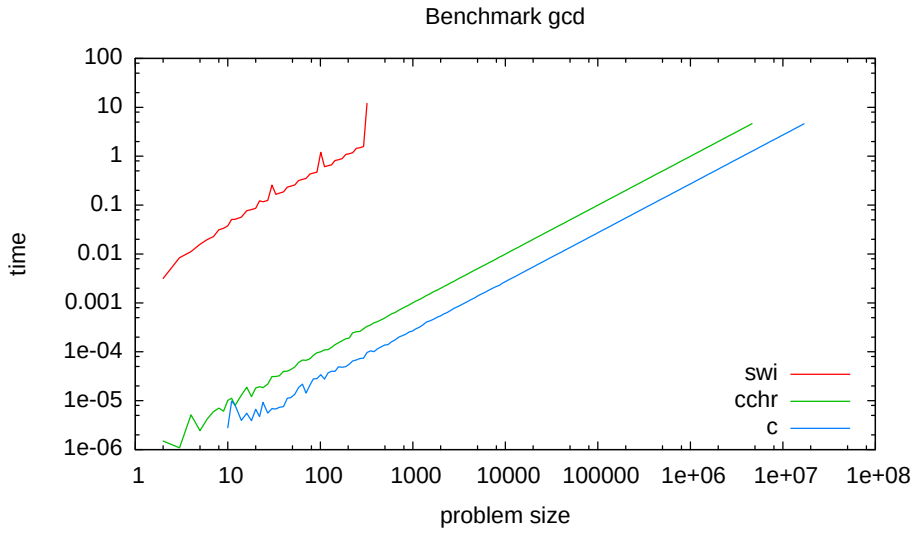

Fig. 5. The `gcd` benchmark

Secondly, the `ram` benchmarks are shown in Figure 6. This benchmark runs a RAM program that counts down from n, the problem size, on the RAM sim-

ulator [14]. The C implementation consists of a loop that 1) fetches the next instruction and 2) executes it in a `switch` statement. Both 1) and any operand fetches in 2) consist of direct array loopups. The lookups in the CHR implementations, in contrast, involve more costly constraint store lookups and constraint argument retrievals. This explains the performance gap between C and the CHR systems. A secondary effect of the costly loops, is the early stack overflow, first in JCHR and then CCHR.

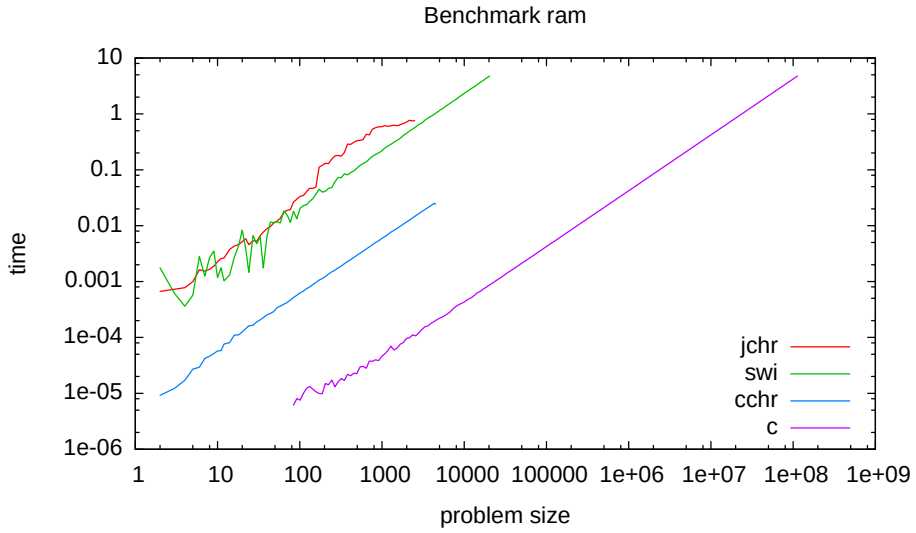

Fig. 6. The `ram` benchmark

7 Related Work

The first signficant achievement in the field of CHR implementation was the stack-based compilation model [6], which was later formalised as the refined operational semantics [3]. This stack-based model allows for efficient execution because it closely resembles the execution model of typical target languages of CHR. C shares this same stack-based model, which is why the refined operational semantics turns out to be a good basis for CCHR as well.

With a good compilation model in place, various optimizations were proposed [7]. This led to questions on the inherent efficiency of CHR: could particular algorithms like union-find be implemented with optimal complexity (yes: [13]), and in fact any algorithm at all (yes: [14])? However, good complexity in theory does not equal good constant factors in practice. In [14] it is reported that CHR

is at its best one order of magnitute slower than C. Our benchmarks show that the recent K.U.Leuven JCHR implementation [16] considerably improves upon constant factors, but still does not bridge the wide gap with C.

8 Conclusion

In this paper we have presented the CCHR system, a CHR system embedded in C. Thanks to its compilation to efficient low-level code, it easily outperforms existing CHR implementations and comes close to native C code.

CCHR opens up a whole new field of applications for CHR, including operating systems and embedded devices. On the level of language integration, CCHR offers the opportunity of providing CHR to many other programming languages that have a C interface. Both of these deserve further investigation.

As to the CCHR implementation itself, we would like to implement the full range of program analyses and optimizations described in the literature. In particular, we will pay attention to the choice of constraint store data structures, such as efficient indexes, and try to further narrow the gap with native C code.

References

1. S. Abdennadher, E. Krämer, M. Saft, and M. Schmauss. JACK: A Java Constraint Kit. In *Proceedings of the International Workshop on Functional and (Constraint) Logic Programming, Kiel*, Kiel, Germany, September 2001.
2. G. J. Duck. *Compilation of Constraint Handling Rules*. PhD thesis, University of Melbourne, Victoria, Australia, December 2005.
3. G. J. Duck, P. J. Stuckey, M. García de la Banda, and C. Holzbaur. The refined operational semantics of Constraint Handling Rules. In B. Demoen and V. Lifschitz, editors, *20th International Conference on Logic Programming (ICLP'04)*, volume 3132 of *Lecture Notes in Computer Science*, pages 90–104, Saint-Malo, France, September 2004. Springer-Verlag.
4. T. Frühwirth. Theory and practice of Constraint Handling Rules. *Journal of Logic Programming*, 37(1–3):95–138, October 1998.
5. C. Holzbaur. Metastructures vs. Attributed Variables in the Context of Extensible Unification. Technical Report TR-92-23, Austrian Research Institute for Artificial Intelligence, Vienna, Austria, 1992.
6. C. Holzbaur and T. Frühwirth. A Prolog Constraint Handling Rules Compiler and Runtime System. *Special Issue Journal of Applied Artificial Intelligence on Constraint Handling Rules*, 14(4), April 2000.
7. C. Holzbaur, M. García de la Banda, P. J. Stuckey, and G. J. Duck. Optimizing Compilation of Constraint Handling Rules in HAL. *Theory and Practice of Logic Programming: Special Issue on Constraint Handling Rules*, 5(Issue 4 & 5):503–531, 2005.
8. R. R. Jenkins. Hash functions for hash table lookup, 1997. http://burtleburtle.net/bob/hash/evahash.html.
9. E. S. Lam and M. Sulzmann. A Concurrent Constraint Handling Rules Implementation in Haskell with Software Transactional Memory. In *Workshop on Declarative Aspects of Multicore Programming (DAMP'07)*, Nice, France, 2007.

10. R. Pagh and F. F. Rodler. Cuckoo hashing. In *ESA '01: Proceedings of the 9th Annual European Symposium on Algorithms*, pages 121–133, London, UK, 2001. Springer-Verlag.
11. T. Schrijvers. *Analyses, Optimizations and Extensions of Constraint Handling Rules*. PhD thesis, K.U.Leuven, Leuven, Belgium, June 2005.
12. T. Schrijvers and B. Demoen. The K.U.Leuven CHR system: Implementation and application. In T. Frühwirth and M. Meister, editors, *First Workshop on Constraint Handling Rules: Selected Contributions*, pages 1–5, Ulm, Germany, May 2004.
13. T. Schrijvers and T. Frühwirth. Optimal Union-Find in Constraint Handling Rules. *Theory and Practice of Logic Programming*, 6(1&2), 2006.
14. J. Sneyers, T. Schrijvers, and B. Demoen. The computational power and complexity of Constraint Handling Rules. In T. Schrijvers and T. Frühwirth, editors, *2nd Workshop on Constraint Handling Rules (CHR'05)*, pages 3–17, Sitges, Spain, October 2005.
15. J. Sneyers, T. Schrijvers, and B. Demoen. Memory reuse for CHR. In S. Etalle and M. Truszczynski, editors, *22nd International Conference on Logic Programming (ICLP'06)*, volume 4079 of *Lecture Notes in Computer Science*, pages 72–86, Seattle, WA, USA, August 2006. Springer-Verlag.
16. P. Van Weert, T. Schrijvers, and B. Demoen. K.U.Leuven JCHR: a user-friendly, flexible and efficient CHR system for Java. In T. Schrijvers and T. Frühwirth, editors, *2nd Workshop on Constraint Handling Rules (CHR'05)*, pages 47–62, Sitges, Spain, October 2005.
17. A. Wolf. Adaptive Constraint Handling with CHR in Java. In *CP'01: Proceedings of the 7th International Conference on Principles and Practice of Constraint Programming*, volume 2239 of *Lecture Notes in Computer Science*, page 256. Springer-Verlag, January 2001.