OKA (pipeline hazards description translator)

Vladimir Makarov, vmakarov@gcc.gnu.org Apr 5, 2001

This document describes OKA (translator of a processor pipeline hazards description into code for fast recognition of pipeline hazards).

Contents

1 Introduction

OKA is a translator of a processor pipeline hazards description (PHD) into code for fast recognition of pipeline hazards. Instruction execution can be started only if its issue conditions are satisfied. If not, instruction is interlocked until its conditions are satisfied. Such an "interlock (pipeline) delay" causes interruption of the fetching of successor instructions (or demands NOP instructions, e.g. for MIPS).

There are two major kind of interlock delays in modern superscalar RISC processors. The first one is data dependence delay. The instruction execution is not started until all source data has been evaluated by

previous instructions (there are more complex cases when the instruction execution starts even when the data are not evaluated but will be ready till given time after the instruction execution start). Taking into account of such kind delay is simple. Data dependence (true, output, and anti-dependence) delay between two instructions is given by constant. In the most cases this approach is adequate. The second kind of interlock delay is reservation delay. Two such way dependent instructions under execution will be in need of shared processors resources, i.e. buses, internal registers, and/or functional units, which are reserved for some time. Taking into account of this kind of delay is complex especially for modern RISC processors. The goal of OKA is to generate code for fast recognition of such kind delay (pipeline hazards).

2 Pipeline hazards description language

A pipeline hazards description describes mainly reservations of processor functional units by an instruction during its execution. The instruction reservations are given by regular expression describing nondeterministic finite state automaton (NDFA).

2.1 Layout of pipeline hazards description

Pipeline hazards description structure has the following layout which is similar to one of YACC file.

```
DECLARATIONS
%%
EXPRESSIONS
%%
ADDITIONAL C/C++ CODE
```

The '%%' serves to separate the sections of description. All sections are optional. The first '%%' starts section of rules and is obligatory even if the section is empty, the second '%%' may be absent if section of additional C/C++ code is absent too.

The section of declarations contains declarations of functional units and instructions of a processor. The section also may contain declarations of automata on which result automaton are split in order to decrease size of tables needed for fast recognition of pipeline hazards. And finally the section may contain subsections of code on C/C++.

The next section contains expressions list which describes functional units reservations by instructions. Regular expressions in general case correspond to nondeterministic final state automaton. The expression list can be empty. In this case the result automaton can contains only arcs marked by special token corresponding advancing cycle.

The additional C/C++ code can contain any C/C++ code you want to use. Often functions which are not generated by the translator but are needed to work of the instruction scheduler go here. This code without changes is placed at the end of file generated by the translator.

2.2 Declarations

The section of declarations may contain the following construction:

```
%instruction IDENTIFIER ...
```

Such constructions declare instructions (or instruction class) of processor. All instructions must be defined in constructions of such kind. The same instruction identifier can be defined repeatedly. There is analogous construction which can serves to describe frequently repeated functional units reservations by real instructions.

```
%reservation IDENTIFIER ...
```

The functional unit declarations has the following form:

```
%unit <automaton identifier> IDENTIFIER ...
```

The construction is used to describe functional units of given processor. Optional identifier in angle brackets describes how to split result automaton onto smaller automata. Each such automaton will contain states corresponding to only reservations of functional units described with given automaton identifier. The same unit identifier can be defined repeatedly with the same automaton identifier.

Sometimes it is necessary to describe that some units can not be reserved simultaneously, e.g. floating point unit is pipelined but can execute only single or double floating point operation. The following construction is useful in such situations

```
%exclude IDENTIFIER ... : IDENTIFIER ...
```

The functional units left to the semicolon can not be reserved with the units right to the semicolon (and vise versa) on the same cycle. All units in the construction should belong the same automata.

All automaton identifiers present in unit declarations must be declared in the following construction

```
%automaton IDENTIFIER ...
```

If there is an automaton declaration, all unit declarations must be with a declared automaton. There may be also the following constructions in the declaration section

```
%local {
   C/C++ DECLARATIONS
}

%import {
   C/C++ DECLARATION
}

and

%export {
   C/C++ DECLARATION
}
```

which contain any C/C++ declarations (types, variables, macros, and so on) used in sections. The local C/C++ declarations are inserted at the begin of generated implementation file (see pipeline hazards description interface) but after include-directive of interface file. C/C++ declarations which start with '%import' are inserted at the begin of generated interface file. C/C++ declarations which start with '%export' are inserted at the end of generated interface file. For example, such C/C++ code may contain definitions of of external variables and functions which refer to definitions generated by OKA. All C/C++ declarations are placed in the same order as in the section of declarations.

2.3 Expressions

The section of declarations is followed by section of expressions. A expression is described the following construction

```
instruction or reservation identifiers : expression;
```

This construction means that instructions given in the left part reserves units according to the expression. In the case of reservation identifier, it is usually used for describing sub-expression frequently used in other expressions. Of course, each declared instruction and reservation must be present in only one such construction. The expression describes non-deterministic finite state automaton (NDFA) in general case and can contain the following forms:

```
EXPRESSION   EXPRESSION
EXPRESSION + EXPRESSION
EXPRESSION * NUMBER
EXPRESSION | EXPRESSION
%nothing
UNIT OR RESERVATION IDENTIFIER
[ EXPRESSION ]
( EXPRESSION )
```

Binary operator ‘ ’ (blank) and ‘+’ describes sequential reservations given by the expressions. In the first case the first unit in the right expression is reserved on the next cycle after the reservation of the last unit in left expression (so called concatenation with cycle advancing). The reservation of an unit is described simply by the unit identifier. In the second case the first unit in the right expression is reserved on the same cycle after the reservation of the last unit in left expression (so called concatenation without cycle advancing). Sometimes it is necessary to describe absence of unit reservations during several cycles. The construction ‘%nothing’ serves for this purpose. ‘%nothing’ can be in the same place as an unit identifier.

Reservation identifier is simply changed by the construction ‘(the corresponding reservation expression)’.

The construction ‘EXPRESSION * NUMBER’ is simply abbreviation of ‘EXPRESSION EXPRESSION ...’ where the expression is repeated by given positive number times.

The construction ‘EXPRESSION | EXPRESSION’ means that the instructions reserve units according to the left or to the right expression (so called alternative). If an unit is present only on one alternative it should belong to the same automaton as units on other alternative. OKA checks this and reports if it is not true.

All binary operators have the left associativity and the following priority:

```
‘*’          - the highest priority
‘+’ and ‘ ’  - the middle priority
‘|’          - the lowest priority
```

The construction ‘[EXPRESSION]’ serves for describing optional construction and is simply abbreviation of the following construction ‘ | EXPRESSION’.

The parentheses are used to grouping sub-expressions in another order then the one given by priorities and associativities of operators.

Full YACC syntax (with some hints in order to transform into LALR (1)-grammar) of pipeline hazards description language is placed in Appendix 1.

3 Generated code

A specification as described in the previous section is translated by OKA (pipeline hazards description translator) into interface and implementation files having the same names as one of specification file and correspondingly suffixes '.h' and '.c' (C code) or '.cpp' (C++ code).

3.1 C++ code

Interface file of PHD consists of the following definitions:

Class 'OKA_chip'.

Object of the class describes current state of processor. The class has the following public members:

1. Function

   `'int OKA_transition (int OKA_instruction)'`

   The function has one parameter: instruction code. If corresponding instruction can be started by the processor in its current state, the function returns 1 and the object (processor) changes own state which reflects starting the execution of given instruction. In the opposite case, when the instruction can not be started, the function returns 0 and the object (processor) does not change own state.

2. Function

   `'int OKA_is_dead_lock (void)'`

   The function returns 1 (TRUE) when transition from the corresponding processor state is not possible on any instruction. The single way to change object (processor) state is to advance time (on one cycle) with the aid of special pseudo-instruction code 'OKA__ADVANCE_CYCLE'. For example, dead lock state for dual-issue processor can be state reflecting starting two instructions on a cycle.

3. Function

   `'void OKA_reset (void)'`

   The function sets up the object (processor) in initial state. No any processor unit is busy in the state.

4. Constructor

   `'OKA_chip (void)'`

   The constructor simply calls function 'OKA_reset'.

Macros or enumeration (see option '-enum')

which declare instruction codes. Macros or enumeration constants have the same name as one in PHD and prefix 'OKA_' (see also option '-p'). OKA always generates additional code 'OKA__ADVANCE_CYCLE'. If such pseudo-instruction starts, the processor make transition into the state reflecting advancing time on one cycle. It is guaranteed that there is always transition from any processor state on given pseudo-instruction. Macros or enumeration are generated in interface file only if option '-export' is present on OKA command line. By default, the macros or the enumeration are generated in the implementation file. Usually, the last case means that the scheduler code is placed PHD in additional C/C++ code.

3.2 C code

Interface file of PHD consists of the following definitions of generated type and functions:

1. Structure 'OKA_chip' which describes state of the processor.
2. Type definition 'OKA_chip' which is simply structure 'OKA_chip'.
3. Function

   ```
   'int OKA_transition (OKA_chip *OKA_chip,
                        int OKA_instruction)'
   ```

   The function has two parameter: pointer to structure describing the processor state and instruction code. If corresponding instruction can be started by the processor in its current state, the function returns 1 and the structure is changed in order to reflects starting the execution of given instruction. In the opposite case, when the instruction can not be started, the function returns 0 and the structure is not changed.
4. Function

   ```
   'int OKA_is_dead_lock (OKA_chip *OKA_chip)'
   ```

   The function returns 1 (TRUE) when transition from the processor state given by the structure is not possible on any instruction. The single way to change processor state is to advance time (on one cycle) with the aid of special pseudo-instruction code 'OKA__ADVANCE_CYCLE'. For example, dead lock state for dual-issue processor can be state reflecting starting two instructions on a cycle.
5. Function

   ```
   'void OKA_reset (OKA_chip *OKA_chip)'
   ```

   The function sets up the structure (processor) in initial state. No any processor unit is busy in the state.
6. Macros or enumeration (see option '-enum') which declare instruction codes. Macros or enumeration constants have the same name as one in PHD and prefix 'OKA_' (see also option '-p'). OKA always generates additional code 'OKA__ADVANCE_CYCLE'. If such pseudo-instruction starts, the processor make transition into the state reflecting advancing time on one cycle. It is guaranteed that there is always transition from any processor state on given pseudo-instruction. Macros or enumeration are generated in interface file only if option '-export' is present on OKA command line. By default, the macros or the enumeration are generated in the implementation file. Usually, the last case means that the scheduler code is placed PHD in additional C/C++ code.

4 OKA Usage

```
OKA(1)                          User Manuals                          OKA(1)



NAME
       oka - pipeline Hazards Description Translator
```

SYNOPSIS

oka [-c++ -debug -enum -export -no-minimization -split number -pprefix -v] specification-file

DESCRIPTION

OKA generates code for fast recognition of pipeline hazards of procesâĂŘsor which is described in specification file. The specification file must have suffix ‘.oka’

The generated code consists of interface and implementation files havâĂŘing the same names as one of specification file and correspondingly suffixes ‘.h’ and ‘.c’ (C code) or ‘.cpp’ (C++ code).

Full documentation of OKA is in OKA User’s manual.

OPTIONS

The options which are known for OKA are:

-c++ OKA generates C++ code instead of C code (default).

-debug OKA creates code for output of debugging information during exeâĂŘcution of the generated code.

-enum OKA generates instruction codes as enumeration constant. By default OKA generates instructions code as macro definition.

-export
OKA generates macros defining identifiers of instructions in the interface file (instead of in the implementation file).

-no-minimization
OKA does not minimization of generated deterministic finite state automaton (DFA).

-split OKA makes automatic splitting automaton on given number automata in order to decrease sizes of generated tables. The option is taken into account only if constructions ‘%unit’ are absent in the specification file. This option has not been implemented yet.

-pprefix
Generated code uses ‘prefix’ instead of ‘OKA’ for names of the generated objects.

-time OKA outputs detail time statistics of its work into stderr.

-v OKA creates description file which contains description of result automaton and statistics information. The file will have

```
              the  same  name  as  one  of given specification file and suffix
              '.output' into standard stream.

FILES
       file.oka
              OKA specification file
       file.c
              generated C implementation file
       file.cpp
              generated C++ implementation file
       file.h
              generated interface file

       There are no any temporary files used by OKA.

ENVIRONMENT
       There are no environment variables which affect OKA behavior.

DIAGNOSTICS
       OKA diagnostics is self-explanatory.

AUTHOR
       Vladimir N. Makarov, vmakarov@gcc.gnu.org

SEE ALSO
       msta(1), shilka(1), sprut(1), nona(1).  OKA manual.

BUGS
       Please, report bugs to https://github.com/dino-lang/dino/issues.



COCOM                              5 APR 2001                              OKA(1)
```

5 Implementation

The OKA is implemented with other COCOM tools. NDFA(s) is created at the begin. After that NDFA(s) is transformed to DFA(s). DFA(s) is than minimized. Tables representing DFA(s) are compacted with the aid of comb-vector method. To decrease size of the generated tables also instructions are divided on equivalence classes. It is especially important when automaton is split on several automata.

6 Future of OKA development

1. Automatic splitting automaton on given number automata.
2. Code for determining what units are reserved in given state. It may be necessary in some very complex cases when given model is not sufficient for accurate recognition of pipeline hazards.

3. Possibility for generation of reverse automaton. Which can be used to insert instruction in already scheduled basic block, e.g. for trace scheduling or scheduling super-blocks.

4. Expansion of model of description of pipeline hazards in order to enable descriptions of processors with dynamic execution and register renaming.

7 Appendix 1 - Syntax of pipeline Hazards Description (YACC grammar)

```
 %token PERCENTS COMMA COLON SEMICOLON LEFT_PARENTHESIS RIGHT_PARENTHESIS
       LEFT_BRACKET RIGHT_BRACKET LEFT_ANGLE_BRACKET RIGHT_ANGLE_BRACKET
       PLUS BAR STAR
       LOCAL IMPORT EXPORT EXCLUSION AUTOMATON
       UNIT NOTHING INSTRUCTION RESERVATION

%token IDENTIFIER NUMBER CODE_INSERTION ADDITIONAL_C_CODE

%start description

%%

description : declaration_part  PERCENTS
              expression_definition_list  ADDITIONAL_C_CODE
            ;

declaration_part :
                 | declaration_part  declaration
                 ;

declaration : identifier_declaration
            | LOCAL  CODE_INSERTION
            | IMPORT  CODE_INSERTION
            | EXPORT  CODE_INSERTION
            ;

identifier_declaration : instruction_declaration
                       | reservation_declaration
                       | unit_declaration
                       | automaton_declaration
                       | exclusion_clause
                       ;

instruction_declaration : INSTRUCTION
                        | instruction_declaration  IDENTIFIER
                        ;

reservation_declaration : RESERVATION
                        | reservation_declaration  IDENTIFIER
                        ;
```

```
unit_declaration : UNIT  optional_automaton_identifier
                 | unit_declaration  IDENTIFIER
                 ;

exclusion_clause : EXCLUSION identifier_list COLON identifier_list
                 ;

identifier_list : IDENTIFIER
                | identifier_list IDENTIFIER
                ;

optional_automaton_identifier :
                              | LEFT_ANGLE_BRACKET
                                  IDENTIFIER  RIGHT_ANGLE_BRACKET
                              ;

automaton_declaration : AUTOMATON
                      | automaton_declaration  IDENTIFIER
;

expression_definition_list
   :
   | expression_definition_list  expression_definition
   ;

expression_definition : instruction_or_reservation_identifier_list  COLON
                        expression  SEMICOLON
                      ;

instruction_or_reservation_identifier_list
  : instruction_or_reservation_identifier
  | instruction_or_reservation_identifier_list
       COMMA  instruction_or_reservation_identifier
  ;

instruction_or_reservation_identifier : IDENTIFIER
                                      ;

expression : expression  expression
           | expression  PLUS  expression
           | expression STAR NUMBER
           | LEFT_PARENTHESIS  expression  RIGHT_PARENTHESIS
           | LEFT_BRACKET  expression  RIGHT_BRACKET
           | expression  BAR  expression
           | unit_or_reservation_identifier
           | NOTHING
           ;

unit_or_reservation_identifier : IDENTIFIER
                               ;
```

8 Appendix 2 - Description of Alpha architecture (EV5 version)

```
 /* Problems of the description:
   o Is it necessary divider_write_back if floating divide has not
     fixed latency? */

%automaton integer multiply float

%unit <integer> e0 e1 load_store_1 load_store_2 store_reservation

%unit <multiply> multiplier multiplier_write_back

%unit <float> fa fm float_divider divider_write_back

%instruction LDL LDQ LDQ_U LDS LDT STL STQ STQ_U STS STT
             LDL_L LDQ_L MB WMB STL_C STQ_C FETCH
             RS RC HW_MFPR HW_MTPR BLBC BLBS BEQ BNE BLT BLE BGT BGE
             FBEQ FBNE FBLT FBLE FBGT FBGE
             JMP JSR RET JSR_COROUTINE BSR BR HW_REI CALLPAL
             LDAH LDA ADDL ADDLV ADDQ ADDQV S4ADDL S4ADDQ S8ADDL S8ADDQ
             S4SUBL S4SUBQ S8SUBL S8SUBQ SUBL SUBLV SUBQ SUBQV
             AND BIC BIS ORNOT XOR EQV
             SLL SRA SRL EXTBL EXTWL EXTLL EXTQL
             EXTWH EXTLH EXTQH INSBL INSWL INSLL INSQL INSWH INSLH INSQH
             MSKBL MSKWL MSKLL MSKQL MSKWH MSKLH MSKQH ZAP ZAPNOT
             CMOVEQ CMOVNE CMOVLBS CMOVLT CMOVGE CMOVLBC CMOVLE CMOVGT
             CMPEQ CMPLT CMPLE CMPULT CMPULE CMPBGE
             MULL MULLV MULL1 MULLV1 MULL2 MULLV2
             MULQ MULQV MULQ1 MULQV1 MULQ2 MULQV2 UMULH UMULH1 UMULH2
             ADDS ADDT SUBS SUBT CPYSN CPYSE CVTLQ CVTQL CVTTQ
             FCMOVEQ FCMOVNE FCMOVLE FCMOVLT FCMOVGE FCMOVGT
             DIVS DIVT MULS MULT CPYS RPCC TRAPB UNOP

%%

/* Class LD:
   o An instruction of class LD can not be simulteniously issued with
     an instruction of class ST;
   o An instruction of class LD can not be issued in the second cycle
     after an instruction of class ST is issued. */
LDL, LDQ, LDQ_U, LDS, LDT:
       (e0 + multiplier_write_back | e1) + (load_store_1 | load_store_2)
       + store_reservation
;

/* Class ST:
   o An instruction of class LD can not be simulteniously issued with
     an instruction of class ST;
   o An instruction of class LD can not be issued in the second cycle
     after an instruction of class ST is issued. */
STL, STQ, STQ_U, STS, STT:
```

```
       e0 + multiplier_write_back + load_store_1 + load_store_2  %nothing
       store_reservation
;

/* Class MBX */
LDL_L, LDQ_L, MB, WMB, STL_C, STQ_C, FETCH:
       e0 + multiplier_write_back
;

/* Class RX */
RS, RC: e0 + multiplier_write_back
;

/* Class MXPR */
HW_MFPR, HW_MTPR: %nothing
;

/* Class IBR */
BLBC, BLBS, BEQ, BNE, BLT, BLE, BGT, BGE: e1
;

/* Class FBR */
FBEQ, FBNE, FBLT, FBLE, FBGT, FBGE: fa
;

/* Class JSR */
JMP, JSR, RET, JSR_COROUTINE, BSR, BR, HW_REI, CALLPAL: e1
;

/* Class IADD */
LDAH, LDA, ADDL, ADDLV, ADDQ, ADDQV, S4ADDL, S4ADDQ, S8ADDL, S8ADDQ,
  S4SUBL, S4SUBQ, S8SUBL, S8SUBQ, SUBL, SUBLV, SUBQ, SUBQV:
             e0 + multiplier_write_back
;

/* Class ILOG */
AND, BIC, BIS, ORNOT, XOR, EQV :  (e0 + multiplier_write_back | e1)
;

/* Class SHIFT */
SLL, SRA, SRL, EXTBL, EXTWL, EXTLL, EXTQL,
  EXTWH, EXTLH, EXTQH, INSBL, INSWL, INSLL, INSQL, INSWH, INSLH, INSQH,
  MSKBL, MSKWL, MSKLL, MSKQL, MSKWH, MSKLH, MSKQH, ZAP, ZAPNOT:
             e0 + multiplier_write_back
;

/* Class CMOV */
CMOVEQ, CMOVNE, CMOVLBS, CMOVLT, CMOVGE, CMOVLBC, CMOVLE, CMOVGT:
             (e0 + multiplier_write_back | e1)
;
```

```
/* Class ICMP */
CMPEQ, CMPLT, CMPLE, CMPULT, CMPULE, CMPBGE:
             (e0 + multiplier_write_back | e1)
;

/* Class IMULL:
   o Thirty-two-bit multiplies have an 8-cycle latency, and the
     multiplier can start a second multiply after 4 cycles, provided
     that the second multiply has no data dependency on the first;
   o No instruction can be issued to pipe e0 exactly two cycles before
     an integer multiplication complete.  */
MULL, MULLV:  e0 + multiplier_write_back + multiplier*4  %nothing*2
                 multiplier_write_back
;

/* Class IMULL with 1 cycle delay */
MULL1, MULLV1:  e0 + multiplier_write_back  %nothing + multiplier*4
                      %nothing*2 multiplier_write_back
;

/* Class IMULL with 2 cycles delay */
MULL2, MULLV2:  e0 + multiplier_write_back  %nothing*2 + multiplier*4
                      %nothing*2 multiplier_write_back
;

/* Class IMULQ:
   o Sixty-for-bit signed multiplies have an 12-cycle latency, and the
     multiplier can start a second multiply after 8 cycles, provided
     that the second multiply has no data dependency on the first;
   o No instruction can be issued to pipe e0 exactly two cycles before
     an integer multiplication complete. */
MULQ, MULQV: e0 + multiplier_write_back + multiplier*8  %nothing*2
                    multiplier_write_back
;

/* Class IMULQ with 1 cycle delay */
MULQ1, MULQV1: e0 + multiplier_write_back  %nothing + multiplier*8
                     %nothing*2  multiplier_write_back
;

/* Class IMULQ with 2 cycles delay */
MULQ2, MULQV2: e0 + multiplier_write_back  %nothing*2 + multiplier*8
                     %nothing*2  multiplier_write_back
;

/* Class IMULH
   o Sixty-for-bit unsigend multiplies have an 14-cycle latency, and
     the multiplier can start a second multiply after 8 cycles, provided
     that the second multiply has no data dependency on the first;
   o No instruction can be issued to pipe e0 exactly two cycles before
     an integer multiplication complete. */
```

```
UMULH: e0 + multiplier_write_back + multiplier*8  %nothing*4
       multiplier_write_back
;

/* Class IMULH with 1 cycle delay */
UMULH1: e0 + multiplier_write_back  %nothing + multiplier*8  %nothing*4
       multiplier_write_back
;

/* Class IMULH with 2 cycles delay */
UMULH2: e0 + multiplier_write_back  %nothing*2 + multiplier*8  %nothing*4
       multiplier_write_back
;

/* Class FADD */
ADDS, ADDT, SUBS, SUBT, CPYSN, CPYSE, CVTLQ, CVTQL, CVTTQ,
   FCMOVEQ, FCMOVNE, FCMOVLE, FCMOVLT, FCMOVGE, FCMOVGT:
             fa + divider_write_back
;

/* Class FDIV:
   o 2.4 bits per cycle average rate.  The next floating divide can be
     issued in the same cycle the result of the previous divide's result
     is avialable.
   o Instruction issue to teh add pipeline continues whaile a divide
     is in progress until the result is ready.  At that point the issue
     stage in the instruction umit stalls one cycle to allow the
     quotient to be sent the round adder and then be written into the
     register file. */
DIVS: fa + float_divider*18 + divider_write_back
;

/* Class FDIV:
   o 2.4 bits per cycle average rate.  The next floating divide can be
     issued in the same cycle the result of the previous divide's result
     is avialable.
   o Instruction issue to teh add pipeline continues whaile a divide
     is in progress until the result is ready.  At that point the issue
     stage in the instruction umit stalls one cycle to allow the
     quotient to be sent the round adder and then be written into the
     register file. */
DIVT: fa + float_divider*30 + divider_write_back
;

/* Class FMUL */
MULS, MULT: fm
;

/* Class FCPYS */
CPYS: (fa + divider_write_back | fm)
;
```

```
/* Class MISC */
RPCC, TRAPB: e0 + multiplier_write_back
;

/* Class UNOP */
UNOP: %nothing
;
```

9 Appendix 3 - Output of pipeline Hazards Description Translator

The following output was generated under Linux 1.2.8 on Compaq Aero (Intel SX-25, 8MB memory).

```
 bash$ time oka -v alpha-ev5.oka

Automaton ‘integer’
        36 NDFA states,             152 NDFA arcs
        32 DFA states,              138 DFA arcs
        24 minimal DFA states,      118 minimal DFA arcs
       146 all instructions           7 instruction equivalence classes

Automaton ‘multiply’
       186 NDFA states,            2283 NDFA arcs
       261 DFA states,             2958 DFA arcs
       236 minimal DFA states,     2748 minimal DFA arcs
       146 all instructions          13 instruction equivalence classes

Automaton ‘float’
       180 NDFA states,             720 NDFA arcs
       209 DFA states,              867 DFA arcs
       149 minimal DFA states,      687 minimal DFA arcs
       146 all instructions           8 instruction equivalence classes

   606 all allocated states,       3802 all allocated arcs
  1281 all allocated alternative states
  2177 all comb vector elements,   4428 all transition table elements
7.90user 1.09system 0:10.39elapsed 86%CPU (0avgtext+0avgdata 0maxresident)k
0inputs+0outputs (69major+247minor)pagefaults 0swaps
```