NAC: A lightweight IR for ASIP compilers

Nikolaos Kavvadias and K. Masselos
{nkavv,kmas}@uop.gr

Department of Computer Science and Technology,
University of Peloponnese,
Tripoli, Greece

18–21 July 2011

ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΛΟΠΟΝΝΗΣΟΥ
UNIVERSITY OF PELOPONNESE

Introduction and motivation

- Heterogeneous platform design using ASIPs (Application-Specific Instruction-set Processors) involves compiler-based design space exploration
- Compiler IR (intermediate representation) serves dual purpose: program representation and abstract machine
- Free/open-source/academic compilers:
  - GCC [1]: GIMPLE IR, interesting, yet unstable
  - LLVM [2]: register-based IR, targeted by clang frontend
  - COINS: S-expression IR
  - Machine-SUIF: influenced Microsoft Phoenix
  - ☞ All these approaches involve excessive and complex infrastructures
- Commercial ASIP compilers:
  - Synopsys Processor Designer: uses ACE CoSy CCMIR
  - IP (ASIP) Designer from Target Compiler Technologies
  - Tensilica XCC compiler
  - ☞ Complex and inaccessible IRs

Introduction of a new low-level intermediate language: NAC

- We propose NAC (N-Address Code), an open, extensible, typed-assembly language
  - Extensible typed assembly language similar in concept to GCC's GIMPLE and LLVM but with unique features
  - Arbitrary *m*-to-*n* mappings
  - Very light/unbiased semantics: a single operation construct
  - Bit-accurate data types (integer, fixed-point arithmetic)
  - Scalar, single-dimensional array and streamed I/O procedure arguments
  - Uses: RISC-like VM for static/dynamic analyses, CDFG extraction, reachability-based data flow analyses, input to HLS kernels, software compilation
  - Minimal SSA (scan-based) algorithms [3, 4]
    - No sophisticated concepts/data structures, no computation of the iterated dominance frontier
    - Suitable for rapid prototyping compilers

NAC EBNF grammar

```
nac_top = {gvar_def} {proc_def}.
gvar_def = "globalvar" anum decl_item_list ";".
proc_def = "procedure" [anum] "(" [arg_list] ")"
           "{" [{lvar_decl}] [{stmt}] "}".
stmt = nac | pcall | id ":".
nac = [id_list "<="] anum [id_list] ";".
pcall = ["(" id_list ")" "<="] anum ["(" id_list ")"] ";".
id_list = id {"," id}.
decl_item_list = decl_item {"," decl_item}.
decl_item = (anum | uninitarr | initarr).
arg_list = arg_decl {"," arg_decl}.
arg_decl = ("in" | "out") anum (anum | uninitarr).
lvar_decl = "localvar" anum decl_item_list ";".
initarr = anum "[" id "]" "=" "{" numer {"," numer} "}".
uninitarr = anum "[" [id] "]".
anum = (letter | "_") {letter | digit}.
id = anum | (["-"] (integer | fxpnum)).
```

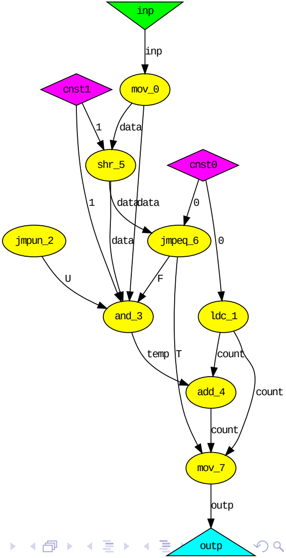
Example translation flow: ANSI C ⇒ NAC ⇒ Graphviz CDFG

```c
void popcount(uint inp,
  uint *outp)
{
  uint data, count, temp;

  data = inp;
  count = 0;

  while (data != 0)
  {
    count = count +
            (data & 0x1);
    data = data >> 0x1;
  }

  *outp = count;
}
```

ANSI C

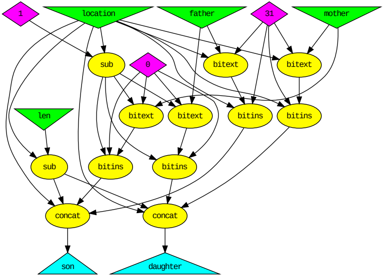
```
procedure popcount (in u32 inp,
  out u32 outp)
{
  localvar u32 data, count,
    temp;
BB1:
  data <= mov inp;
  count <= ldc 0;
  BB2 <= jmpun;
BB2:
  temp <= and data, 1;
  count <= add count, temp;
  data <= shr data, 1;
  BB3, BB2 <= jmpeq data, 0;
BB3:
  outp <= mov count;
}
```

NAC IR



IR transformations

- Motivation: single- (*crcsp*) and double-point (*crcdp*) genetic crossover operators
- Conventional mapping to IR produces slow code/implementation
- New IR nodes: **`bitins`** (bitfield insertion), **`bitext`** (extraction), **`concat`** (subword concatenation)
- IR extension through graph transformation, implementation on ByoRISC ASIP [5]



Post-transformation IR

GA oper.	Bit-level oper.	N_i/N_o	Cycles (seq.)	CI cyc.	CI area (MAU)
crcsp	No/Yes	4/1	76-13	–	–
crcsp	No/Yes	8/1	41-6	3-1	0.977-0.142
crcsp	No/Yes	8/2	5-1	3-1	1.867-0.153
crcdp	No/Yes	4/1	111-18	–	–
crcdp	No/Yes	8/1	58-8	3-1	1.466-0.147
crcdp	No/Yes	8/2	5-1	3-1	2.800-0.164

Results on Virtex-4

NAC virtual machine profiling

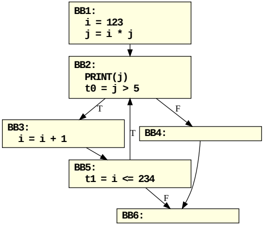
- NACVM: the abstract machine inferred by a basic NAC instruction set
- Examples of static/dynamic profiling using C backend
  - *atsort* (all topological sorts), *coins* (compute change), multimode *cordic* (DSP algorithm), *easter* (Easter date calculations), *fixsqrt* (fixed-point sqrt), *perfect* (perfect number detection), *sieve* (prime sieve), *xorshift* (100 PRNG calls)
- Shown: lines (NAC, CDFG), num. of CDFGs (P), nodes (N), and edges (E), ϕ statements, dynamic non-SSA instructions

Application	LOC (NAC)	LOC (dot)	P	V	E	#ϕs	#Instructions.
atsort	155	484	2	136	336	10	6907
coins	105	509	2	121	376	10	405726
cordic	56	178	1	57	115	7	256335
easter	47	111	1	46	59	2	3082
fixsqrt	32	87	1	29	52	6	833900
perfect	31	65	1	23	36	4	6590739
sieve	82	199	2	64	123	12	515687
xorshift	26	80	1	29	45	0	2000

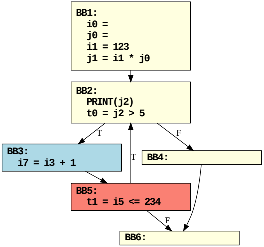
Variable numbering with algorithm P

```
VariableNumbering(List NACs, List vars):
  ssa_vars = empty; var_reads = zeroes;
  var_writes = ones; set_writes = 0;
  curr_bb = 0; prev_bb = -1;
  bbnum = get number of basic blocks from NACs;
  for stmt in NACs do
    // Update curr_bb, prev_bb, var_reads, var_writes.
    if stmt.bb != curr_bb then
      prev_bb = curr_bb; curr_bb = stmt.bbix;
      if curr_bb > 1 and set_writes == 0 then
        var_writes = bbnum; set_writes = 1;
      var_reads = curr_bb;
    for input operand (opnd) in stmt do
      if opnd is a localvar and is scalar then
        // Create a numbered version of opnd.
        ssaopnd = opnd ## var_reads['opnd'];
      update input operands of stmt;
    for output operand (opnd) in stmt do
      get opnd_ix = index of opnd in vars;
      if opnd is a localvar and is scalar then
        // Create numbered ver. of opnd, then copy it.
        if stmt.bb > 1 then
          var_writes['opnd'] += 1;
        var_reads['opnd'] = var_writes['opnd'];
        ssaopnd = opnd ## var_writes['opnd'];
        insert ssaopnd to ssa_vars list;
      update output operands of stmt;
    update stmt in NACs;
  delete localvar scalars from vars;
  merge ssa_vars with vars;
```

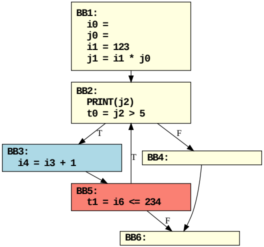
Example: Variable numbering with P and H



Prior SSA



Variable numbering with P



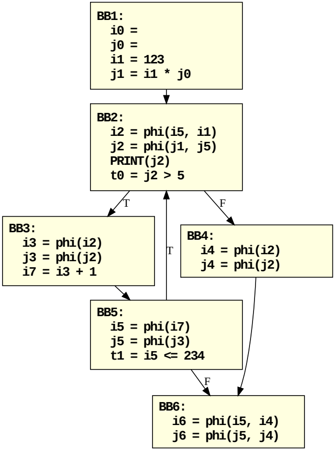
Variable numbering with H

- Algorithm P [3]
  - every variable is split at BB boundaries
  - ϕ-functions are placed for each variable in each BB
  - preassigns variable versions
- Algorithm H [4]
  - doesn't predetermine variable versions at control-flow joins
  - different ϕ-insertion as well, same ϕ-minimization/DCE to P

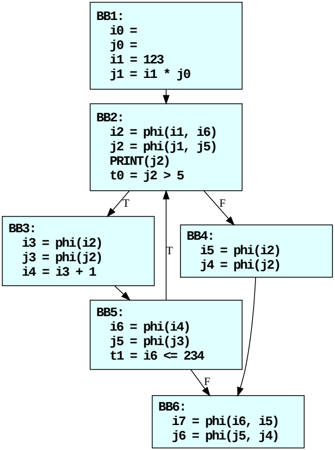
ϕ-insertion with algorithm P

```
PhiInsertion(List NACs, List vars, List labels,
  List nonssa_vars):
  phi_stmts = empty; bb_preds = zeroes; bb_preds_num = 0;
  (ST, G) = create CFG from (NACs, labels);
  for k in BBs(ST) do
    insert predecessor BBs of k in bb_preds;
    bb_preds_num = get number of predecessor BBs of k;
    for sopnd in nonssa_vars do
      if sopnd is localvar scalar, has def/use in k then
        phi_opnds_in = empty; phi_opnds_out = empty;
        // Create the phi stmt dest operand.
        if bb_preds_num > 1 then
          ssaopnd_out = sopnd ## k+1;
          insert ssaopnd_out to phi_opnds_out, vars;
        ix = 0;
        for n in bb_preds_num do
          if bb_preds[n] != -1 then
            ix = SSA ver of sopnd at last def in BB #n;
            if ix == 0 then ix = bb_preds[n] + 1;
            // Create the phi stmt source operand.
            ssaopnd_in = sopnd ## ix;
            insert ssaopnd_in to phi_opnds_in;
        // Create the phi statement.
        if k == 0 and BB #k does not define sopnd then
          phi_stmt = LOADCONST(phi_opnds_out);
        elsif BB #k has predecessors then
          phi_stmt = PHI(phi_opnds_out, phi_opnds_in);
        insert phi_stmt to phi_stmts;
  merge NACs with phi_stmts;
  update absolute addresses (addr) in NACs, labels;
```

Example: ϕ-insertion with P and H



ϕ-insertion with P



ϕ-insertion with H

Conclusions and references

- An extensible, typed-assembly intermediate language (NAC) was presented
- Its applicability was illustrated through cases of IR extension, profiling and compilation pass development (SSA construction)
- Descriptions of minimal SSA constuction algorithms with elementary data structures shown for the first time

- GCC. [Online]. Available: `http://gcc.gnu.org`
- LLVM. [Online]. Available: `http://llvm.org`
- A. W. Appel, "SSA is functional programming," *ACM SIGPLAN Notices*, vol. 33, no. 4, pp. 17–20, Apr 1998.
- J. Aycock and N. Horspool, "Simple generation of static single assignment form," in *Proc. 9th Int. Conf. in Compiler Construction*, 2000, pp. 110–125.
- N. Kavvadias and S. Nikolaidis, "The ByoRISC configurable processor family," in *Proc. IFIP/IEEE VLSI-SoC*, Oct. 2008, pp. 439–444, Rhodes Island, Greece.