NAC: A lightweight intermediate representation for ASIP compilers

Nikolaos Kavvadias and Kostas Masselos
Department of Computer Science and Technology, University of Peloponnese, 22100 Tripoli, Greece

Abstract—*ASIP processors are tuned for optimized mapping of narrow application sets in heterogeneous platforms. Their successful development relies on compiler-based design space exploration. The careful design of the compiler intermediate language is a necessity, due to its dual purpose as both the program representation and an abstract target machine. Its design affects the complexity, efficiency and ease of maintenance of all compilation phases.*

In this work, an extensible typed assembly intermediate language, NAC, is presented. It can be used for processor exploration, optimizing intermediate representation (IR) transformations and SSA compilation. Minimal SSA construction algorithms are thoroughly presented for the first time. [1]

Keywords: compilers, intermediate representation, SSA, ASIP

1. Introduction and related work

Recent compilation frameworks provide linear IRs for applying analyses, optimizations and as input for backend code generation. GCC [1] supports the GIMPLE IR. Many GCC optimizations have been rewritten for GIMPLE, but it is still undergoing grammar and interface changes. GCC supports backends for ASIP processors such as baseline Xtensa [2] but it is not suitable for rapid retargeting to non-trivial architectures. LLVM [3] uses a register-based IR named LLVM bitcode, targeted by a C/C++ companion frontend (clang). It is written in better coding style than GCC, but similarly the IR infrastructure and semantics are excessive.

In this paper, the NAC (N-Address Code) IR is introduced. NAC supports semantic-free n-input/m-output mappings, user-defined data types, and specifies a virtual machine architecture. NAC's strength is its simplicity: it is inherently easy to develop a CDFG (Control/Data Flow Graph) extraction API, apply graph-based IR transformations for domain specialization, investigate SSA (Static Single Assignment) construction algorithms and perform other compilation tasks for ASIPs.

Specifically, this paper investigates minimal SSA construction schemes [4], [5] that don't require the computation of the iterated dominance frontier [6]. For the first time, detailed implementations are illustrated to ease their adoption in new projects.

[1]The presented research work was co-funded by the European Union in the frame of the ENOSYS project (FP7-ICT-248821) (`www.enosys-project.eu`).

2. Representing programs in NAC

In this section, the NAC typed-assembly language is described. NAC provides arbitrary n-to-m mappings allowing the elimination of implicit side-effects, a single construct for all operations, and bit-accurate data types. It supports scalar, single-dimensional array and streamed I/O procedure arguments. NAC statements are labels, n-address instructions or procedure calls.

An n-address instruction is actually the specification of a mapping from a set of n ordered inputs to a set of m ordered outputs. Also termed as (n, m)-operation, it is formatted as follows:
`outp1, ..., outpm <= op inp1, ..., inpn;`
where: `op` is a mnemonic referring to an IR instruction, `inp1`, `...`, `inpn` are its n inputs, and `outp1`, `...`, `outpm` its m outputs.

All declared objects have an explicit static type specification: "globalvar" (a global scalar or array variable), "localvar" (a scalar or array local), "in" (an input argument to the given procedure), or "out" (an output argument).

NAC supports bit-accurate data types for (signed/unsigned) integer/fixed-point and floating-point arithmetic. Data type specifications are essentially strings that can be easily decoded by a regular expression scanner; typical examples are `u32, s11, q4.4u, q2.14s, f1.8.23`, respectively.

The EBNF grammar for NAC is shown in Fig. 1 where it can be seen that rules "nac" and "pcall" provide the means for the n-to-m generic mapping for operations and procedure calls, respectively. It is important to note that NAC has no predefined operator set; operators are defined through a textual mnemonic.

For instance, an addition of two scalar operands is written as: `a <= add b, c;`. Control-transfer operations include conditional and unconditional jumps explicitly visible in the IR. An example of an unconditional jump would be: `BB5 <= jmpun;` while conditional jumps always declare both targets: `BB1, BB2 <= jmpeq i, 10;`. This statement enables a control transfer to the entry of basic block BB1 when i equals to 10, otherwise to BB2.

Procedures are supported as non-atomic operations by using a similar form to operations. In `(y) <= sqrt(x);` the square root of an operand `x` is computed; procedure argument lists are indicated as enclosed in parentheses.

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

Fig. 1: EBNF grammar for NAC (N-Address Code).

Table 1: A set of basic operations for a NAC-based IR.

Mnemonic	Description	(N_i, N_o)
`ldc`	Load constant	(1,1)
`neg, mov`	Unary arithmetic op.	(1,1)
`add, sub, abs, min, max, mul, div, mod, shl, shr`	Binary arithmetic op.	(2,1)
`not, and, ior, xor`	Logical	(2,1)
`setzz`	Comparison for `zz`: (`eq,ne,lt,le,gt,ge`)	(2,1)
`muxzz`	Conditional selection	(3,1)
`load, store`	Load/Store from/to mem.	(2,1)
`sxt, zxt, trunc`	Type conversion	(1,1)
`jmpun`	Unconditional jump	(0,1)
`jmpzz`	Conditional jump	(2,2)

2.1 Encoding NAC information

A NAC program incorporates the complete information of a translation unit of the original program comprising of a "globalvar" definition list and a procedure list. A single NAC procedure is defined by the following set of lists: ordered input (output) arguments, "localvar" definitions, NAC statements and basic block (BB) labels.

Statements are organized in the form of a record. Lists *opnds_in* and *opnds_out* collect operand items, following the definition of an "OperandItem" record. This record is comprised of an identifier name, a data type specification, an operand type (*otype*) representation and an absolute operand item index. *otype* can take one of the following values: {INARG, OUTARG, LOCALVAR, GLOBALVAR, CONSTANT} and {INVAR, OUTVAR} as an additional def/use specifier for NAC statements.

3. Uses and extensions of NAC

3.1 A basic NAC implementation

A basic operation set for RISC compilation is summarized in Table 1. N_i (N_o) denotes the number of input (output) operands for each operation.

The memory access model defines dedicated address spaces per array, so that both loads and stores require the array identifier as an explicit operand. For an indexed load in C (`b = a[i];`), a frontend would generate the following NAC: `b <= load a, i;`, while for an indexed store (`a[i] = b;`) it is `a <= store b, i;`.

Table 2: CI characteristics for hand-optimized ANSI C implementations of *crcsp* and *crcdp*.

GA operator	Bit-level oper.	N_i/N_o	Cycles (seq.)	CI cycles	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

3.2 IR extensions

We have defined three custom IR operators, `bitins`, `bitext` and `concat`, for bitfield insertion/extraction from a word and concatenation of two or more subwords. As motivational examples, the single- (*crcsp*) and double-point (*crcdp*) crossover operators are examined. C code for the genetic algorithm operators was passed to Machine-SUIF [7] for IR generation using a peephole matching-based code selection pass for the ByoRISC ASIP [8]. ByoRISC supports CIs with up to 8 inputs and 8 outputs.

crcsp reads four inputs: two parent chromosomes (father, mother), crossover point (location), and chromosome length (len) and produces two independent outputs; the (son, daughter) chromosomes for the next generation. *crcdp* defines two crossover points for bitfield exchange.

In Table 2, with bit-level operators unused, the minimum number of cycles required for *crcsp* are 76 for a sequential schedule and 12 for an ASAP, while for the *crcdp* these limits are 111 and 14, respectively. When the bit-level operators are used, the sequential schedules without CIs require 13 and 18 cycles for *crcsp* and *crcdp* respectively with ASAP schedules of 5 cycles for both. When the $N_i/N_o = \{8/2\}$ constraint is used, a single-cycle multi-input, multi-output (MIMO) CI is identified for each crossover operator. The area requirement is estimated relatively to the area (multiplier area unit or MAU) of a 32-bit single-cycle multiplier characterized for a Virtex-4 FPGA (XC4VLX25).

3.3 CDFG construction

A novel, fast CDFG construction algorithm has been devised for both SSA and non-SSA NAC forms producing flat CDFGs (Fig. 2). A CDFG symbol table item is a node (operation, procedure call, globalvar, or constant) or edge (localvar) with user-defined attributes: the unique name, label and data type specification; node and edge type enumeration; respective order of incoming or outgoing edges; input/output argument order of a node and BB index. Further attributes can be defined, e.g. for scheduling bookkeeping.

3.4 Application profiling with NACVM

NAC programs can be either interpreted or translated to low-level C for performance evaluation on the corresponding

```
NACtoCDFG()
  input List NACs, List variables, List labels, Graph cfg;
  output SymbolTable st, Graph cdfg;
begin
  Insert constant, input/output arguments and global
  variable operand nodes to st;
  Insert operation nodes;
  Insert incoming {global/constant/input, operation} and
  outgoing {operation, global/output} edges;
  Add control-dependence edges among operation nodes;
  Add data-dependence edges among operation nodes,
  extract loop-carried dependencies via cfg reachability;
  Generate cdfg from st;
end
```

Fig. 2: CDFG construction algorithm accepting NAC input.

Table 3: Application profiling with a NAC framework.

App.	LOC (NAC)	LOC (dot)	$P/V/E$	#ϕs	#Instr.
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

abstract machine, NACVM. A set of realistic kernels has been selected: *atsort* (an all topological sorts algorithm by Knuth), *coins* (compute change with minimum amount of coins), multimode *cordic* computation, *easter* (Easter date calculations), *fixsqrt* (fixed-point sqrt), *perfect* (perfect number detection), *sieve* (prime sieve of Eratosthenes) and *xorshift* (100 calls to G. Marsaglia's PRNG).

Static and dynamic metrics have been collected in Table 3. For each application (App.), the lines of NAC and resulting CDFGs are given in columns 2-3, number of CDFGs (P: procedures), vertices and edges (for each procedure) in column 4, amount of ϕ statements (column 5) and lastly the number of dynamic instructions for the non-SSA case.

4. SSA construction algorithms

This paper argues that rapid prototyping compilers, would benefit from straightforward SSA construction schemes which don't require the use of sophisticated concepts and data structures [4], [5].

Algorithm P presents a "really-crude" approach for variable renaming and ϕ-function insertion [4]. In the first phase, every variable is split at BB boundaries, while in the second phase ϕ-functions are placed for each variable in each BB. Variable versions are actually preassigned in constant time and reflect a specific BB ordering (e.g. DFS). Thus, variable versioning starts from a positive integer n, equal to the number of BBs in the given CFG.

Algorithm H does not predetermine variable versions at control-flow joins but accounts ϕs the same way as actual computations visible in the original CFG. Due to this fact, ϕ-insertion also presents dissimilarities. Both methods share common ϕ-minimization and dead code elimination phases.

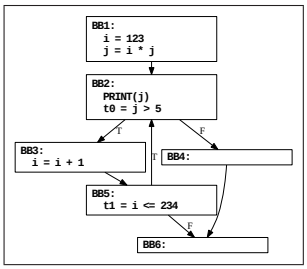

Fig. 3: CFG of the example subprogram from [5].

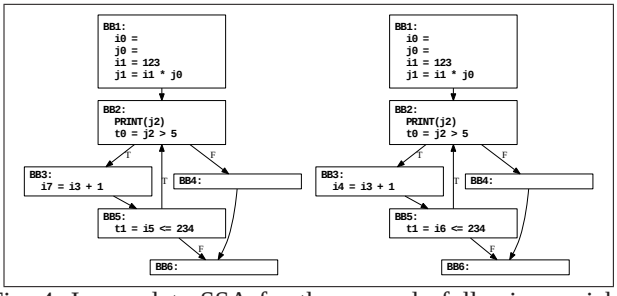

Fig. 4: Incomplete SSA for the example following variable numbering with algorithm P (left) and H (right).

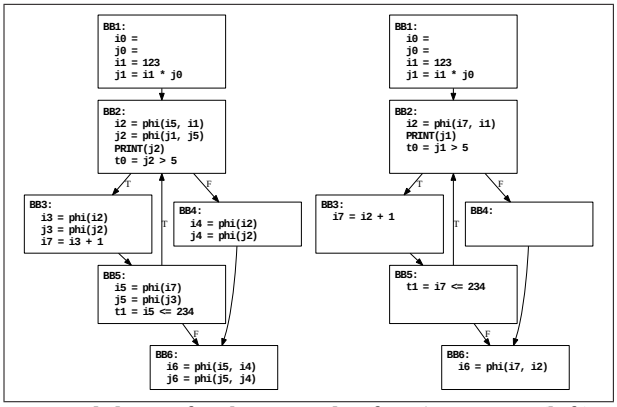

Fig. 5: Valid SSA for the example after ϕ-insertion (left) and ϕ-minimization and dead code elimination (right).

4.1 Motivating example

The motivating example from [5] is shown in Fig. 3, with incomplete SSA following variable numbering in Fig. 4.

Valid unoptimized and minimal SSA are shown in Fig. 5 involving the maximum and minimum possible number of ϕs, respectively, as generated by P. H presents only lexicographic and not semantic differences to this result. Both algorithms achieve the generation of minimal SSA involving the two ϕ statements in BB2 and BB6.

4.2 Analysis of algorithms P and H

Variable numbering in algorithm P is given in Fig. 6. Only arrays and maps (key-indexed) are used for sequences of same-type elements. Vectorized assignments to arrays/maps are allowed, copying a scalar to all elements. A single iterative form is used for iterating over a set or sequence. Lists can have subset updates, member insertions and deletions

```
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
    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
        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
        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;
```

Fig. 6: Variable numbering in 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;
        if bb_preds_num > 1 then
          ssaopnd_out = sopnd ## k+1;
          insert ssaopnd_out to phi_opnds_out;
          insert ssaopnd_out to 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;
            ssaopnd_in = sopnd ## ix;
            insert ssaopnd_in to phi_opnds_in;
        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;
```

Fig. 7: ϕ-insertion in algorithm P.

and can be merged. A key-based retrieval operation named *get* is also used. GNU C concatenation ## is used.

P alters in-place the NAC statement list and replaces the non-SSA variable list by a versioned one, *ssa_vars*. *var_reads* and *var_writes* define maps for keeping the version numbers of CFG variables. *set_writes* is a flag array for controlling proper initialization of *var_writes*. *curr_bb* and *prev_bb* are BB markers for the current and previous BB accessible in a single pass over NAC statements. *bbnum* is the number of BBs in the CFG. For each NAC, *var_writes* is set to *bbnum* for all except the entry BB. *var_reads* is set to *curr_bb*. Then, for each NAC input operand, which is a local scalar, a versioned variable is created using the entry in *var_reads*. For output operands, *var_writes* is incremented for a non-entry BB and the *var_reads* entry for the same operand is updated for future uses. A new SSA variable is defined according to the value of *var_writes* entry and is inserted in *ssa_vars*. After updating each NAC accordingly, local scalars are deleted from the initial list and *ssa_vars* is merged with *vars*.

Variable numbering in algorithm H uses a 'visited' map, *var_bb_id*. After some preprocessing, input operands are numbered in the exact same way as in P. Output operands are associated to defined SSA variables: for unvisited variables of entry blocks, *var_writes* is incremented by two, otherwise by one. Then unvisited variables are marked as visited. Afterwards, *var_reads* is updated as in P.

ϕ-insertion according to P reads both the non-SSA and SSA variable lists as shown in Fig. 7. ϕ statements are collected in *phi_stmts*. *bb_preds* is the list of all preceding BBs for a given block and *bb_preds_num* keeps their number. All BBs are scanned to update *bb_preds* and *bb_preds_num*, then for each non-SSA variable active in the given BB (k), the ϕ statement destination operand is created as the k+1 version. Determining input SSA operands for each NAC requires scanning all predecessors, and if any exist, to assign the version *bb_preds*[n]+1. Then, either a constant load or a ϕ statement is created, the latter for non-entry BBs.

ϕ-insertion in H examines all parsed BBs for determining subsequent variable versions for each ϕ output operand. If a def of this operand is found, its SSA version is incremented by two over the current index, otherwise by one. Source operand version is defined by a similar process, without the additional version increment.

5. Conclusions

In this paper, a semantic-free IR, named NAC, was presented, for use in rapid prototyping ASIP compilers. Its applicability is illustrated through cases of rule-based transformation for better CI generation, application profiling and self-contained description of minimal SSA construction algorithms.

References

[1] GCC. [Online]. Available: http://gcc.gnu.org

[2] Tensilica. [Online]. Available: http://www.tensilica.com

[3] LLVM. [Online]. Available: http://llvm.org

[4] A. W. Appel, "SSA is functional programming," *ACM SIGPLAN Notices*, vol. 33, no. 4, pp. 17–20, Apr 1998.

[5] J. Aycock and N. Horspool, "Simple generation of static single assignment form," in *Proc. 9th Int. Conf. in Compiler Construction*, 2000, pp. 110–125.

[6] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Trans. Prog. Lang. Syst.*, vol. 13, no. 4, pp. 451–490, Oct 1991.

[7] Machine-SUIF. [Online]. Available: http://www.eecs.harvard.edu/hube/software/

[8] N. Kavvadias and S. Nikolaidis, "The ByoRISC configurable processor family," in *Proc. IFIP/IEEE VLSI-SoC*, Oct. 2008, pp. 439–444.