aprof user manual

Title	aprof (ALMA profiler)
Author	Nikolaos Kavvadias
Contact	nkavv@uop.gr
Website	http://www.nkavvadias.com
Release Date	06 May 2013
Version	0.4.0
Rev. history	
v0.1.0	31-07-2012 Draft/preliminary binary release of nac2c, the compiled simulator of aprof.
v0.2.0	31-08-2012 Source release for the 1st increment of nac2c.
v0.3.0	30-11-2012 Binary release for the 1st draft release of aprof. nac2c is now considered a component of aprof.
v0.4.0	06-05-2013 Added tutorial section in README.

1. Introduction

"aprof" (ALMA profiler) is a performance and resource utilization estimation tool. For obtaining these measures, "aprof" implements an abstract machine with unlimited resources. It accepts input specification in either the NAC (N-Address Code) intermediate representation or ALMA IR (ANSI C) form. "aprof" produces two basic outcomes, a) the number of dynamic abstract machine cycles and b) basic block operation schedule that indicates resource utilization for a given application.

"aprof" consists of the following components:

- "libnac" is the implementation of an API as a static library that allows for storing, manipulating and examining NAC IR. For instance, the scheduler engines are considered as part of "libnac". As of the 0.3.0 release, two schedulers are available for a sequential and an intra-block parallel machine model.
- "nac2c" is an (application-specific) compiled simulator generator. The compiled simulators are then executed on a host platform (typically: x86-32).
- "instrument" is a collection of small C and TXL tools that allow for inserting basic block counters in order to obtain basic block execution frequencies.

- "prof" is a collection of small C and AWK tools for inserting the necessary code in compiled simulators for generating profiling reports.

The current NAC specification is detailed in the corresponding reference manual found in the /doc subdirectory in HTML and PDF form.

2. Obtaining and setting up aprof

aprof releases use the `aprof-[src|lin|win]-yymmdd.tar.bz2` naming convention.

- Select `src` for source, `lin` for Linux or `win` for Windows binaries release
- `yymmdd` is the release date

2.1 Obtaining aprof

Download aprof from the ALMA intranet (UOP directories).

Unarchive to a local directory:
e.g. `C:/cygwin/home/user` for Windows/Cygwin users
or `/home/user` for a Linux user

2.2 Setting up optional tools

For using aprof, a Linux or Windows installation is required. For Windows, Cygwin is suggested (optional) in order to significantly ease the use of aprof.

In any case, standard Unix/Linux tools are expected:

- bash
- make
- patch
- gawk

Boehm's garbage collector is also required, but is included both in source and compiled form (binary releases only) within the /thirdparty subdirectory.

For Windows:

- Go to http://sources.redhat.com/cygwin/
- Download the automated web installer (`setup.exe`)
- Copy it to an empty local directory (e.g. `C:\temp\cygwin`)
- Click `setup.exe`
- Select `Install from the Internet`. Make sure to select `make` since it might be disabled in the preselection.

Cygwin will then be setup in the `C:\cygwin` directory of your Windows OS.

For Linux:

- Any recent Linux distribution should do; try using Ubuntu 11.10.

2.3 aprof setup

There is no actual installation procedure; the user should just unzip the `aprof-[lin|win]-yymmdd.tar.bz2` binary release archive to a local directory. Usual choices include `C:/cygwin/home/user` for Windows (no Cygwin) users and `/home/user` for Windows Cygwin/Linux users where `user` is the name of the current user.

Then, change directory to `/home/user/aprof`. On Cygwin for instance, type:

```
$ cd /home/user/aprof
```

Set up the `APROFTOP` environmental variable:

```
$ source env.sh
```

The location of the garbage collector is adjusted accordingly in the corresponding makefiles.

You may add the `/aprof/bin` directory to your path:

```
$ export PATH=$APROFTOP/bin:$PATH
```

2.4. Building from sources

This subsection is relevant only to the source releases of aprof (`aprof-src-yymmdd.tar.bz2`). To build aprof from sources the following are required:

A) For Linux users:

- A typical Linux installation (bash, make, gawk)
- The TXL compiler from http://www.txl.ca (e.g. version 10.6)
- In case you want to use your system's `gc`, change `GCPATH` in `/src/Makefile.linux` accordingly. Then run the build script from the top-level subdirectory:

```
$ cd /home/user/aprof
$ ./build-lin.sh
```

- If you want to recompile `gc`, use the `build-a.sh` script. The script should be changed accordingly (comment and uncommented certain lines) for selecting either `gc6.8` or `gc-7.2alpha6` or for enabling a Windows Cygwin or a Linux build.

```
$ ./build-lin-a.sh
```

B) For Windows users:

- Windows XP SP2 or older (untested on newer systems).
- Cygwin environment (bash, make, gawk). Cygwin can be installed via an automated web installer (setup.exe) from http://sources.redhat.com/cygwin/
- TXL installation for Cygwin.

- Run the build script from the top-level subdirectory of aprof:

```
$ cd /home/user/aprof
$ ./build.sh
```

- Similarly to the Windows case, for rebuilding `gc`, use the following:

```
$ ./build-a.sh
```

3. File listing

The aprof distribution includes the following files. Files and/or directories denoted by a capital S are available in source releases of aprof. Similarly, a capital B denotes files/directories present solely in binary releases:

/aprof	Top-level directory
COPYRIGHT	aprof (binary or source code) license.
S build.sh **S** build-a.sh **S** build-lin.sh **S** build-lin-a.sh **S** clean.sh	Build script for aprof (Windows). Build script for aprof and gc (Windows). Build script for aprof (Linux). Build script for aprof and gc (Linux). Cleans up the /bin and /src subdirectories.
env.sh	Script to setup the environment.
B /aprof/bin	Binaries' directory
fixnac.exe meascycles.exe nac2c.exe nacbbinscount.exe nacparser.exe nactoglobal.exe cygwin1.dll	fixnac executable for either Windows or Linux. meascycles executable for either Windows or Linux. nac2c executable for either Windows or Linux. nacinsbbcount exec. for either Windows or Linux. nacparser executable for either Windows or Linux. nactoglobal executable for either Windows or Linux. Cygwin API DLL (not required with a Cygwin setup).
/aprof/doc	Documentation
README README.html README.pdf nac-refman.txt nac-refman.html nac-refman.pdf	This file. HTML version of README. PDF version of README. Reference manual for the NAC programming language. HTML version of the above. PDF version of the above.
S /aprof/src	Main source directory
/aprof/src/instrument	"instrument" directory
Makefile build.sh fixnac.c nac.Grm nacinsbbcount.txl nacparser.txl nactoglobal.txl	Makefile for Windows Cygwin and Linux. Bash script for building the TXL applications. Applies additional fixes to an instrumented NAC file. TXL grammar for NAC. Inserts basic block counters in NAC programs. NAC parser and pretty-printer. Moves all declarations to the earliest possible site.

/aprof/src/libnac	"libnac" directory
Makefile	Makefile for Windows Cygwin.
Makefile.linux	Makefile for Linux.
attrgraph.[c|h]	Attributed graphs API.
cdfa.[c|h]	Control and data flow analyses API (includes SSA).
cga.[c|h]	Call graph API (mainly SSA).
datastructs.h	Basic data structures and enums.
emit.[c|h]	Emitters for graph representations.
genansic.[c|h]	ANSI C code generation routines.
genmacros.h	General purpose C macros.
graph.[c|h]	Graph manipulation API
item.[c|h]	CDFG (Control-Data Flow Graph) items API.
list.[c|h]	Doubly-linked list and iterators API.
machine.[c|h]	Machine paramteters for the NAC abstract machine.
lexer.patch	Patch for the NAC lexer (`lex.nac.c`).
nac.[c|h]	NAC (N-Address Code) manipulation API.
nac.[l|y]	Lexer and parser for the NAC programming language.
sched.[c|h]	Scheduling (naive, ASAP) API.
symtab.[c|h]	Symbol table API.
utils.[c|h]	Various utility functions.
/aprof/src/nac2c	"nac2c" directory
Makefile	Makefile for Windows Cygwin.
Makefile.linux	Makefile for Linux.
nac2c.c	Driver code and option parsing for `nac2c`.
/aprof/src/prof	"prof" directory
Makefile	Makefile for Windows Cygwin and Linux.
build.sh	Bash script for building the TXL applications.
countbbs.awk	Counts the number of BBs in a NAC translation unit.
measycles.c	Counts the number of abstract machine cycles spent.
/aprof/tests	Test suite directory
*.0.nac	The aprof test suite. Includes 30 applications, each in the corresponding subdirectory: (binarysearch, bitrev, bubblesort, cordic, divider, editdist, fact, factr, fibo, fibor, fir, fixsqrt, frac, gcd, knapsack, loop1, mandel, matmult, minimal, mips, multiply, perfect, popcount, sieve, smithwaterman, sobel, tak, thornapprox, xorshift, yuv2rgba).
*.c	Reference C implementation for test suite, used for generating reference data.
clean-tests.sh	Clean the debris in all /tests subdirectories.
run-aprof.sh	Run the entire test suite.
run-aprof-app.sh	Run a single application from test suite.
thorn.pgm	PGM image required for running the `thornapprox` benchmark.
/aprof/thirdparty	Third-party source/binaries directory

B /gc **B** /gc-linux **B** /gc-mingw	Garbage collector binaries for Windows Cygwin. Garbage collector binaries for Linux. Garbage collector binaries for Windows MingW.
/src	Source code versions of the garbage collector.

4. aprof tools usage

4.1 nac2c usage

The basic usage of nac2c follows the syntax:

```
$ ./nac2c.exe [options] input.nac
```

The translated C representation of `input.nac` is produced in a series of output files called `input<i>_nac.c`, separately for each NAC-level procedure, where `input<i>` is the name of the corresponding procedure. Pre-existing files are overwritten.

options‘ is one or more of the following:

-d Enable debug output.

-force-data-types Force predefined data types as given in NAC code. Essentially disables the effect of both interval analysis and the alternative of using the unknown data type `na`.

-ssa Internal construction of SSA (Static Single Assignment) form.

-pseudo-ssa Internal construction of local SSA-like form.

-use-aycockhorspool Enables SSA construction using the Aycock-Horspool algorithm.

-keep-ssa Does not perform out-of-SSA conversion and thus keeps PHI statements in the generated CDFGs.

-phi-bbs Enable the generation of BB arguments in `phi` NAC statements.

-no-phi-bbs Disable the generation of BB arguments in `phi` statements (default).

-emit-ansic Emit the equivalent ANSI C program after processing (including SSA conversion, if enabled).

-emit-cdfg Generate the Graphviz representations for all procedure CDFGs.

-emit-cfg Generate the Graphviz representations for all procedure CFGs.

-emit-cg Generate the Graphviz representation of the application call graph.

-gcc Generate Makefile for GCC compilation (default).

-llvm Generate Makefile for LLVM compilation and/or interpretation.

4.2 fixnac usage

The basic usage of fixnac follows the syntax:

```
$ ./fixnac.exe [options] -i input.nac -o output.nac
```

Additional fixes are applied to the instrumented `input.nac` such as the additon of the declaration of the globalvar BB array for storing BB execution frequencies.

options' is one or more of the following:

-h Print this help.

-decl-bb-array Declare the _BB globalvar.

-init-bb-array Initialize the _BB globalvar to zeros. Only in the effect if **-decl-bb-array** has been defined.

-max-bbs <num> Specify the maximum number of basic blocks in a program. Default: 10000.

4.3 measycles usage

The basic usage of measycles follows the syntax:

```
$ ./measycles.exe input.nac
```

It reads the `input.nac` which is assumed to be uninstrumented, the `input_prof.txt` profiling report file and the corresponding `input_sched.txt` scheduling data file. Then it reports the total number of dynamic abstract machine cycles in the following form:

```
"Number of abstract machine cycles:  %lld
```

as a C-based long long int (64-bit signed integer).

4.4 TXL passes

Executables generated by TXL passes source files share a common invocation style:

```
$ ./<trans>.exe input.nac -q -raw > output.nac
```

This scheme applies for executables `nacbbinscounters`, `nacparser` and `nactoglobal`.

4.5 countbbs.awk usage

This AWK script generates a textual report named `bbs.txt` that stores the total number of basic blocks in the given NAC translation unit. `countbbs` is invoked as follows:

```
$ gawk -f ${APROFTOP}/countbbs.awk < ${app}.nac > bbs.txt
```

5. Running the test suite

The basic tests under the /tests subdirectory can be exercised by running corresponding test script:

```
$ cd $APROFTOP
$ cd tests
$ ./run-aprof.sh
```

Alternatively, each application can be tested separately using the `run-aprof-app.sh` script, e.g. as follows for the case of the `fibo` benchmark:

```
$ ./run-aprof-app.sh fibo
```

By running a benchmark, the following files can be generated, if using the appropriate options, assumably for a benchmark called `app` comprising of `proc` procedures:

ansic.mk	Makefile for GCC or LLVM compilation.
bbs.txt	Total number of BBs in the NAC translation unit.
builtin_names.txt	Name listing of builtin (black box) functions.
proc.dot proc.dot.png	CDFG representation in Graphviz for procedure proc. Visualization of the Graphviz CDFG for procedure proc.
proc_cfg.dot proc_cfg.dot.png	CFG representation in Graphviz for procedure proc. Visualization of the Graphviz CFG for procedure proc.
app_cg.dot app_cg.dot.png	Call graph representation in Graphviz for app. Visualization of the Graphviz call graph for app.
app.nac	Working NAC representation of the application.
app.exe	Executable generated by the C implementation of app.
app_test_data.txt	Reference test data generated by app.exe.
app_prof.txt	Basic block profiling report.
app_sched.txt	Scheduling report (number of static cycles per BB).
main.c	Generated C code containing the main() function.
main.h	Header/interface file for the generated files.
proc_nac.c	Backend C code generated from the corresponding NAC.
procedure_names.txt	Name listing of the procedures used in app.

6. Step-by-step guide to profiling

This section provides detailed information on the actual process of profiling. First, in order to profile an application which is assumed to be contained in a single NAC translation unit, two files are required:

- `app.0.nac`, which is the NAC representation of the application
- `app.c`, which is a C implementation that is used in the context of `aprof` for reference input/output data generation.

As a test vehicle, the iterative implementation of a factorial computation will be used, namely the `fact` application. Thus, the corresponding initial files are `fact.0.nac` and `fact.c`.

The contents of `fact.0.nac` are as follows:

```
procedure fact (in s32 n, out s32 y)
{
  localvar s32 res;
  localvar s32 x;
  localvar s32 i;
L0005:
  x <= mov n;
  res <= ldc 1;
  i <= ldc 1;
  D_1363 <= jmpun;
D_1362:
  res <= mul res, i;
  i <= add i, 1;
  D_1363 <= jmpun;
D_1363:
  D_1362, D_1364 <= jmple i, x;
D_1364:
  y <= mov res;
}
```

Since NAC is a relatively low-level language, a high-level language frontend would have to be used for profiling larger applications. In this sense, `fact.c` would serve as input to a C frontend producing NAC output.

The reference `fact.c` has the following contents:

```
#ifdef TEST
#include <stdio.h>
#endif

int fact(int n) {
  int res, x, i;
  x = n;
  res = 1;
  for (i = 1; i <= x; i++) {
    res = res * i;
  }
  return res;
}

#ifdef TEST
int main() {
  int i;
  int result;
  for (i = 0; i <= 13; i++) {
    result = fact(i);
    printf("%08x %08x\n", i, result);
```

```
  }
return 0;
}
#endif
```

To automate the profiling process, it is more suitable to use scripting. The `aprof` distribution contains reference scripts for profiling. Specifically, the `run-aprof-app.sh` can be used.

The rest of this guide will provide a detailed view of the approach taken by the aforementioned script in the form of a series of steps. The `$APROFTOP` environmental variable is the path to the top-level directory of `aprof`.

6.1 Generation of the reference test data

Assuming that `gcc` is used as the host machine compiler, the following prompt generates the corresponding executable:

```
gcc -DTEST -DDATAGEN -Wall -O2 -o fact.exe fact.c
```

Then, the reference data can be generated:

```
./$fact.exe >& fact_test_data.txt
```

The contents of `fact_test_data.txt` are input and output values for *n* and *y=fact(n)* in hexadecimal form:

```
00000000 00000001
00000001 00000001
00000002 00000002
00000003 00000006
00000004 00000018
00000005 00000078
00000006 000002d0
00000007 000013b0
00000008 00009d80
00000009 00058980
0000000a 00375f00
0000000b 02611500
0000000c 1c8cfc00
0000000d 7328cc00
```

6.2 Create a working copy of the NAC representation of fact

This can be accomplished by copying `fact.0.nac` to `fact.nac`:

```
cp -f fact.0.nac fact.nac
```

6.3 Tracking the number of basic blocks in the unit

The following bash script variable

```
num_bbs="0"
```

is used for maintaining the number of basic blocks in the NAC translation unit.

6.4 Generate the bbs.txt file

An AWK script, `countbbs.awk` is used for counting the basic blocks in the entire translation unit. This is performed by enumerating the labels in the NAC program, since all NAC basic blocks have explicit labels:

```
gawk -f ${APROFTOP}/src/prof/countbbs.awk < fact.nac >
bbs.txt
```

Then, the `bbs.txt` file is processed, to get the number of basic blocks:

```
# Process the bbs.txt file.
bbsfile="bbs.txt"
while read -r bbs;
do
  num_bbs="${bbs}"
done < ${bbsfile}
```

A `while` loop is used, in order to extract all the basic block counts in `bbs.txt` in case of a multi-translation unit application (currently unsupported by most features of `aprof`).

6.5 Instrumentation of the NAC file

The `nacinsbbcount` TXL pass inserts profiling code for dynamic basic block counting in NAC programs:

```
$APROFTOP/bin/nacinsbbcount.exe fact.nac ${txlopts} >
fact.1.nac
```

A usual setup for TXL options is:

```
txlcopts="-q -raw"
```

Then, `fixnac` is invoked for adding bookkeeping code as for the declaration of the `_BB` global array, its initialization and specifying the maximum number of basic blocks in the program.

```
APROFTOP/bin/fixnac.exe -decl-bb-array -init-bb-array \
-max-bbs ${num_bbs} -i fact.1.nac -o fact.2.nac
cp -f fact.2.nac fact.nac
```

The resulting `fact.nac` representation is as follows:

```
globalvar u64 _BB[4]={0,0,0,0};
procedure fact(in s32 n,out s32 y)
{
```

```
  localvar s32 res;
  localvar s32 x;
  localvar s32 i;
  localvar u32 _temp_addr;
  localvar u32 _temp_data;
L0005:
  _temp_addr <= ldc 0;
  _temp_data <= load _BB,_temp_addr;
  _temp_data <= add _temp_data,1;
  _BB <= store _temp_data,_temp_addr;
  x <= mov n;
  res <= ldc 1;
  i <= ldc 1;
  D_1363 <= jmpun;
D_1362:
  _temp_addr <= ldc 1;
  _temp_data <= load _BB,_temp_addr;
  _temp_data <= add _temp_data,1;
  _BB <= store _temp_data,_temp_addr;
  res <= mul res,i;
  i <= add i,1;
  D_1363 <= jmpun;
D_1363:
  _temp_addr <= ldc 2;
  _temp_data <= load _BB,_temp_addr;
  _temp_data <= add _temp_data,1;
  _BB <= store _temp_data,_temp_addr;
  D_1362,D_1364 <= jmple i,x;
D_1364:
  _temp_addr <= ldc 3;
  _temp_data <= load _BB,_temp_addr;
  _temp_data <= add _temp_data,1;
  _BB <= store _temp_data,_temp_addr;
  y <= mov res;
}
```

6.6 Generation of the backend C files for the given NAC t.u.

The profiling process is based on the generation of a compiled simulator for the NAC program. This is accomplished with the use of the `nac2c` decompiler which is applied on the original form of the application (`fact.0.nac`). This is needed in order to extract the static schedule of the initial form of the application.

Either the sequential or the ASAP scheduler can be used, which correspondingly reflect a sequential or intra-block parallel abstract machine.

First, a static scheduling extraction run of `nac2c` must be performed.

For enabling the sequential scheduler the following should be used:

```
$APROFTOP/bin/nac2c.exe -force-data-types -emit-ansic
-emit-cdfg -sched-naive fact.0.nac
```

The ASAP scheduler is enabled as follows, since it mandates at least pseudo-SSA (Static-Single Assignment):

```
$APROFTOP/bin/nac2c.exe -force-data-types -ssa
-pseudo-ssa \
-emit-ansic -emit-cdfg -sched-asap fact.0.nac
```

Then, `nac2c` generates a multitude of files, which have been detailed in Section 5.

A file named `fact_sched.txt` is expected to be passed to a second run of `aprof`, which is the profiling run:

```
cp -f fact.0_sched.txt fact_sched.txt
```

`fact.sched.txt` contains the estimated static cycles per basic block:

```
5
4
2
2
```

`aprof` proceeds with the second run of `nac2c`:

```
$APROFTOP/bin/nac2c.exe -force-data-types -emit-ansic
-emit-cdfg -prof fact.nac
```

6.7 Optional step for generating CDFG views

Optionally, the Graphviz (*.dot) representation of each NAC procedure can be visualized using the following snippet:

```
procfile="procedure_names.txt"
while read -r app2;
do
  echo "Creating CDFG view for ${app2}"
  dot -Tpng -O ${app2}.dot
done < ${procfile}
```

6.8 Building and running the compiled simulator

In this step, the `ansic.mk` generated Makefile must be run in order to build `main.exe`, which is the compiled simulator for the examined application, `fact`.

```
$ make -f ansic.mk clean
$ make -f ansic.mk
$ ./main
```

This run produces `fact_prof.txt` which contains dynamic basic block counts:

```
14
91
105
14
```

6.9 Calculation of dynamic abstract machine cycles

Finally, `measycles` is used for combining the dynamic basic block counts written in `fact_prof.txt` with the static cycle estimates which are found in `fact_sched.txt`:

$APROFTOP/bin/measycles.exe fact.nac“

As a result, the profiling estimate is produced in the standard output. For instance, the sequential scheduler produces:

```
Number of abstract machine cycles:  650
```

while the ASAP scheduler computes the following:

```
Number of abstract machine cycles:  551
```

7. Contact

You may contact me for further questions/suggestions/corrections at:

Nikolaos Kavvadias <nkavv@uop.gr>
<nikolaos.kavvadias@gmail.com>
http://www.nkavvadias.com
Department of Computer Science and Technology
University of Peloponnese
Tripoli, Greece