# Profiling Infrastructure for apeNEXT

Enrico Boldrini
Università degli Studi di Firenze
boldra@tiscali.it

*Aim of the project was to implement a profiling infrastructure for apeNEXT programs. apeNEXT programmers can use the profiler to have precise information about the efficiency of their programs and detect bottlenecks for apeNEXT resources. The profiler can do both static code analysis and dynamic program analysis of apeNEXT programs. Reports of each analysis are then stored in XML format. Special care was put on writing a graphical interface through which a user can interact with the results in a friendly way.*

## 1 Introduction

An overview of the apeNEXT system is given, as well as an introduction on performance analysis. The document will continue then with the description and usage of the newly written profiling infrastructure for apeNEXT.

### *1.1 Overview of the apeNEXT system*

apeNEXT is a custom designed, massively parallel computer, developed in a project started by DESY, INFN and the University of Paris Sud.

The Zeuthen installation is composed by currently 7 crates, each of them containing 256 processors. The peak performance per crate is 250 GFlops (300 after upgrade), the aggregated peak performance reaches 2 TFlops (2.5 after upgrade).

APE machines are designed for numerical simulations in understanding the interactions of elementary particles, in particular their strong interactions described by quantum chromodynamics (QCD). The formulation of QCD which can be simulated on computers is called Lattice QCD (LQCD). Features of apeNEXT machines are support for complex numbers, the implementation of the multiply-add operation $a \times b + c$ (a, b, c complex numbers), a large register file, node interconnections optimized for nearest neighbors communication, VLIW microcode.

apeNEXT users write normally their programs in C or Tao languages. Then the source file is processed through a set of tools (the compilation chain). At first the high level assembler (.sasm file) is generated by the c compiler and the Tao compiler. Then mpp generates the low level assembly program (.masm file), that can be optimized with sofan. At the last step the shaker takes care of making the microcode (.mem file) that is the code that the machine will execute.

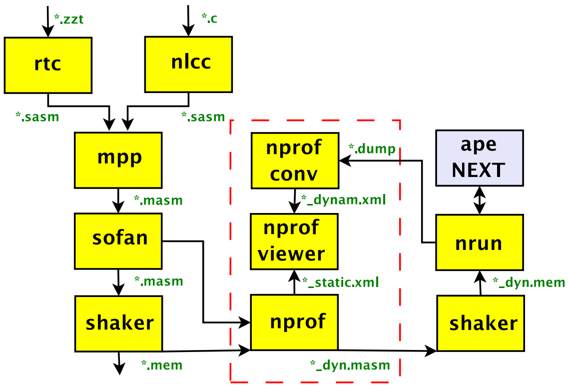



Fig. 1: *Compilation chain and the profiling infrastructure.*

### *1.2 Performance Analysis*

The objective of a profiler is to provide performance measurements of user programs. With these data the user can understand better the behavior of his program, find out bottlenecks for the resources and try to rearrange his code in a more efficient way.

If the analysis is done only by inspecting user source code it is called static code analysis. If information about performance is gathered as the program runs we are dealing with dynamic program analysis.

#### 1.2.1 Static analysis

This kind of analysis gives a characterization of a program without running it. It is performed directly on the source code or on some form of the object code. Uses of the information obtained from the analysis usually vary from highlighting possible coding errors to formal methods that mathematically prove properties about a given program. Only on microcode driven machines (that is the apeNEXT case) we can use results of static code analysis to accurately predict the performance of user programs.

The apeNEXT processor is a microcoded VLIW which means that the code is statically scheduled by the compiler: the code generated is exactly the code executed cycle by cycle by the processor. This characteristic enables us to predict exactly the execution time of the code, major part of the performance and other precise informations such as density of floating point operations and density of load/store.

Still, the performance also depends on dynamic behavior: memory behavior - its latency and the occurrence of a page fault; network behavior - delays introduced by the asynchronous execution of each nodes affect the global performance of the program.

#### 1.2.2 Dynamic analysis

Tools for dynamic analysis measure the behavior of a program as it runs. Measurements that can be performed include the frequency and duration of function calls, counting of page faults, saturation level of memory bandwidth. The output is a stream of recorded events (a trace) or a statistical summary of the events observed (a profile). Dynamic analyzers use a wide variety of techniques to collect data, including hardware interrupts, code instrumentation, operating system hooks.

The strategy used by nprof is to instrument the assembly program with additional instructions to collect the required information at run time. Instrumenting the program is a powerful technique that can measure very specific informations like memory access patterns, but it is also a delicate operation that can cause changes in the performance of the program, and alter the results of the analysis.

### *1.3 Entities hierarchy*

The smallest part of the user program on which we can define measurements is called a sblock. Generally speaking it comprises the instructions that are between two so called *slabels* of the assembly program. Higher scopes on which we can perform analysis are called block, procedure and program levels. They are all basically composed by sblocks and they have a hierarchical structure.

## 2 Profiling tools overview

The profiling infrastructure is composed of two programs, nprof and nprof-viewer. nprof can perform static analysis, that is analysis of the code of a given program without running it, and dynamic analysis, that is the investigation of a program's behavior using information gathered as the program runs. After both cases nprof generates reports files in XML format; nprof-viewer take these XML files and let the user interact with them through a graphical interface.

Details on how to use the programs and a technical description are provided in the following sections.

### *2.1 Use of nprof for the static analysis*

The first objective of nprof is to generate the static code characterization. To perform this step nprof takes on input the low level assembly program (.masm file) and the corresponding microcode program (.mem file). These two files are generated through the normal compilation chain starting from an high level program (C or tao programming languages). An user can run nprof with the following syntax:

```
nprof MASMFILE MEMFILE
```

Here the specific task of nprof is to analyze the static performance of all code sections. A set of measurements is calculated at sblocks level, it comprises measurements about code size, code compression, instructions type, register file port usage, number of words loaded from or stored to memory, number of complex/float operations.

An XML file is generated as a result of this first step. It contains the results of static measurements that have been performed on each

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<PROG filename="example.iasm" architecture="APENEXT">
<PROC start="1" name="__START_OF_MASM__">
  <BLOCK start="2" name="__START_OF_MASM__">
   <SBLOCK start="2" name="__START_OF_MASM__">
    <STATIC>
     <icache>fifo</icache>
     <p0>1</p0>
     <p1>0</p1>
     ...
    </STATIC>
   </SBLOCK>
  </BLOCK>
  <BLOCK start="12" name="__SC_INIT_DONE__">
   <SBLOCK start="12" name="__SC_INIT_DONE__">
    <STATIC>
     <icache>fifo</icache>
     <p0>2</p0>
     <p1>2</p1>
     ...
```

Fig. 2: *An example of a static analysis XML file*

sblock of the program as well as the structure of the program itself in terms of program, procedures, blocks, sblocks.

## 2.2 *Use of nprof for the dynamic analysis*

The second task of nprof is to perform dynamic analysis of a user program. The only measurement that can be performed until now is the number of times a procedure is executed. To perform this analysis the user can launch nprof with the syntax:

```
nprof MASMFILE MEMFILE -dynProcedures
```

The result of this process is a modification of the original MASMFILE, it contains the original assembly code and additional assembly instructions which will increment the counter corresponding to each procedure. These counters are dumped to a file at the end of program execution. This file has to be converted by nprof in an XML dynamic analysis report file. Syntax:

```
nprof MASMFILE MEMFILE -dumpfile
```

The result of the dynamic analysis is an XML dynamic analysis report file.

## 2.3 *nprof-viewer*

Aim of nprof-viewer is to show the results to the user in a friendly way. It is a graphical interface, through which the user can interact in a number of ways.

To start the main program it is enough to type `nprof-viewer` at the shell prompt. The user can interact with the main window with a menu. At first an XML static file has to be loaded. A file chooser dialog can be activated through the menu `File->Open`.

A feature of nprof-viewer is the possibility of changing the granularity level (from the menu `Granularity Level`). User can choose between program, procedures, blocks and sblocks levels. Especially procedure level can be interesting for the user that want to analyze his C functions. Other granularity levels are blocks and sblocks and all.

By default the granularity level is set on sblocks. After the XML file is loaded nprof-viewer shows in a table the list of sblocks that are included in the program. Columns contain attributes and measurements. In the column called *Code* the user can view the source code of his high level program and identify directly the lines of code to which the analysis refers. The interactive search of user code in big programs is available.

The column called *Basic Info* offers in a simplified representation the results of the analysis. There are four ways of presenting the results, they are called `Basic Info`, `Raw Info`, `Memory Subsystem`, `Arithmetic Unit`. The user can switch between them through the menu `Profile Information`. Each of the views offer relevant measurements in a compact way, that can help the user to identify characteristics of his code.

The user can also add columns with measurements of interest through the menu `Options->Add Measurements`. Many measurements are choosable from a dialog window, they are presented as check-buttons. If the user toggles a button the column concerning that measurement will show up or hide.

Measurements that show a percentage information are colored according to its value to help the user to identify critical sections of code. The user can click on the header of these columns to sort rows of the table by the corresponding measurement.

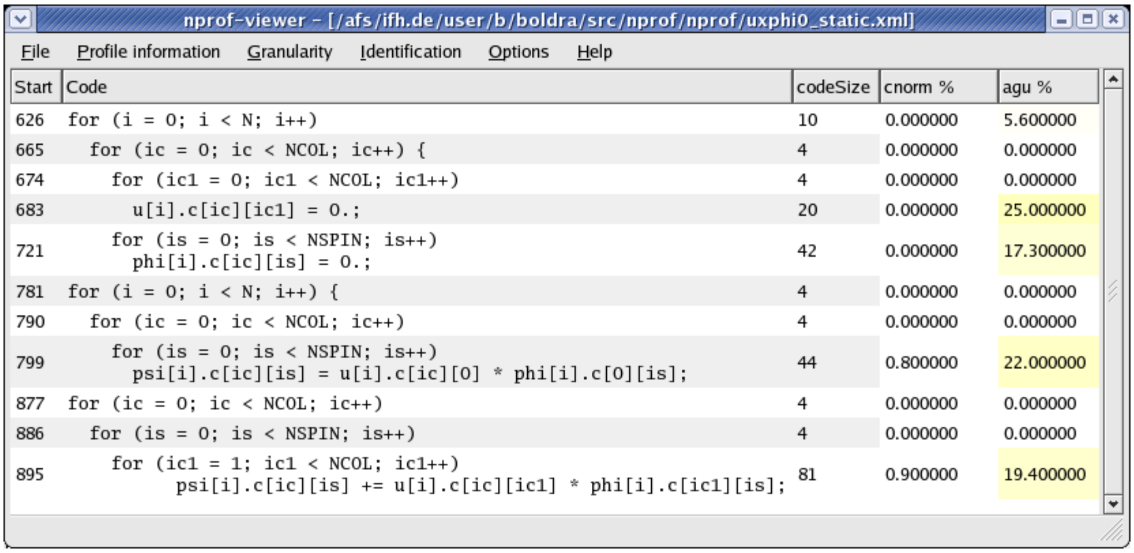
The menu `Identification` is useful to change the way in which rows are identified. The user can choose between `Labels`, `Code` or `Both`. Choosing Labels each row is identified only by the assembly label, Code shows the high level source code. When clicking on `Hide silent labels` rows that don't correspond to user code are hidden. This option can be useful to hide information about utility code that is linked automatically with the program.

```c
#include <stdio.h>

#define NCOL 3
#define NSPIN 4
#define N 100

typedef struct { complex c[NCOL][NCOL];
} su3;
typedef struct { complex
c[NCOL][NSPIN]; } color_spin;

int main()
{
  int i, ic, is, ic0, ic1, k;
  su3 u[N];
  color_spin phi[N], psi[N];

  /* Initialize memory */
  for (i = 0; i < N; i++)
    for (ic = 0; ic < NCOL; ic++) {
      for (ic1 = 0; ic1 < NCOL; ic1++)
        u[i].c[ic][ic1] = 0.;
      for (is = 0; is < NSPIN; is++)
        phi[i].c[ic][is] = 0.;
    }


  /* Calculate product psi[i] = u[i] phi[i] */
  for (i = 0; i < N; i++) {
    for (ic = 0; ic < NCOL; ic++)
      for (is = 0; is < NSPIN; is++)
psi[i].c[ic][is] = u[i].c[ic][0] *
phi[i].c[0][is];

    for (ic = 0; ic < NCOL; ic++)
      for (is = 0; is < NSPIN; is++)
        for (ic1 = 1; ic1 < NCOL;
ic1++)
   psi[i].c[ic][is] += u[i].c[ic][ic1]
* phi[i].c[ic1][is];
  }

  return 0;
}
```

Fig. 3: *Naïve implementation of product calculation, from file uxphi0.c*

```c
#include <stdio.h>

#define NCOL 3
#define NSPIN 4
#define N 100

typedef struct { complex c[NCOL][NCOL];
} su3;
typedef struct { complex
c[NCOL][NSPIN]; } color_spin;

int main()
{
  int i, ic, is, ic0, ic1, k;
  su3 u[N];
  color_spin phi[N], psi[N];

  /* Initialize memory */
  for (i = 0; i < N; i++)
    for (ic = 0; ic < NCOL; ic++) {
      for (ic1 = 0; ic1 < NCOL; ic1++)
        u[i].c[ic][ic1] = 0.;
      for (is = 0; is < NSPIN; is++)
        phi[i].c[ic][is] = 0.;
    }


  /* Calculate product psi[i] = u[i] phi[i] */
#pragma cache
  {
  for (i = 0; i < N; i++) {
    for (ic = 0; ic < NCOL; ic++)
      for (is = 0; is < NSPIN; is++)
psi[i].c[ic][is] = u[i].c[ic][0] *
phi[i].c[0][is];

#pragma for
    for (ic = 0; ic < NCOL; ic++)
#pragma for
      for (is = 0; is < NSPIN; is++)
#pragma for
        for (ic1 = 1; ic1 < NCOL;
ic1++)
   psi[i].c[ic][is] += u[i].c[ic][ic1]
* phi[i].c[ic1][is];
#pragma endfor
#pragma endfor
#pragma endfor
  }
  }

  return 0;
}
```

Fig. 4: *Optimized implementation of product calculation, from file uxphi2.c*

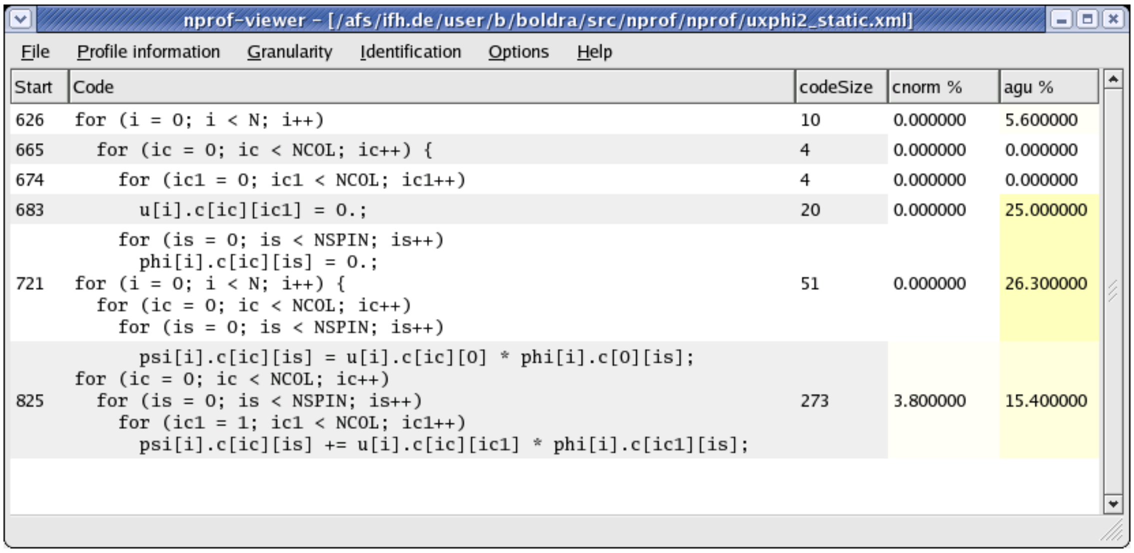


nprof-viewer - [/afs/ifh.de/user/b/boldra/src/nprof/nprof/uxphi0_static.xml]

File Profile information Granularity Identification Options Help

| Start | Code | codeSize | cnorm % | agu % |
|---|---|---|---|---|
| 626 | `for (i = 0; i < N; i++)` | 10 | 0.000000 | 5.600000 |
| 665 | `for (ic = 0; ic < NCOL; ic++) {` | 4 | 0.000000 | 0.000000 |
| 674 | `for (ic1 = 0; ic1 < NCOL; ic1++)` | 4 | 0.000000 | 0.000000 |
| 683 | `u[i].c[ic][ic1] = 0.;` | 20 | 0.000000 | 25.000000 |
| 721 | `for (is = 0; is < NSPIN; is++)`<br>`phi[i].c[ic][is] = 0.;` | 42 | 0.000000 | 17.300000 |
| 781 | `for (i = 0; i < N; i++) {` | 4 | 0.000000 | 0.000000 |
| 790 | `for (ic = 0; ic < NCOL; ic++)` | 4 | 0.000000 | 0.000000 |
| 799 | `for (is = 0; is < NSPIN; is++)`<br>`psi[i].c[ic][is] = u[i].c[ic][0] * phi[i].c[0][is];` | 44 | 0.800000 | 22.000000 |
| 877 | `for (ic = 0; ic < NCOL; ic++)` | 4 | 0.000000 | 0.000000 |
| 886 | `for (is = 0; is < NSPIN; is++)` | 4 | 0.000000 | 0.000000 |
| 895 | `for (ic1 = 1; ic1 < NCOL; ic1++)`<br>`psi[i].c[ic][is] += u[i].c[ic][ic1] * phi[i].c[ic1][is];` | 81 | 0.900000 | 19.400000 |

Fig. 5: *nprof-viewer main window: non optimized program. Relevant loops are in line 799, 895.*



nprof-viewer - [/afs/ifh.de/user/b/boldra/src/nprof/nprof/uxphi2_static.xml]

File Profile information Granularity Identification Options Help

| Start | Code | codeSize | cnorm % | agu % |
|---|---|---|---|---|
| 626 | `for (i = 0; i < N; i++)` | 10 | 0.000000 | 5.600000 |
| 665 | `for (ic = 0; ic < NCOL; ic++) {` | 4 | 0.000000 | 0.000000 |
| 674 | `for (ic1 = 0; ic1 < NCOL; ic1++)` | 4 | 0.000000 | 0.000000 |
| 683 | `u[i].c[ic][ic1] = 0.;` | 20 | 0.000000 | 25.000000 |
| 721 | `for (is = 0; is < NSPIN; is++)`<br>`phi[i].c[ic][is] = 0.;`<br>`for (i = 0; i < N; i++) {`<br>`for (ic = 0; ic < NCOL; ic++)`<br>`for (is = 0; is < NSPIN; is++)` | 51 | 0.000000 | 26.300000 |
| 825 | `psi[i].c[ic][is] = u[i].c[ic][0] * phi[i].c[0][is];`<br>`for (ic = 0; ic < NCOL; ic++)`<br>`for (is = 0; is < NSPIN; is++)`<br>`for (ic1 = 1; ic1 < NCOL; ic1++)`<br>`psi[i].c[ic][is] += u[i].c[ic][ic1] * phi[i].c[ic1][is];` | 273 | 3.800000 | 15.400000 |

Fig. 6: *nprof-viewer main window: optimized program. The unrolled loops are now together in one row with start index 825.*

## 3 Example

Let's have a look to an example use of the profiler. The program, showed in figure 3, performs the operation:

$$\psi^a_{i,\mu} = \sum_b U_i^{a,b} \phi^b_{i,\mu}$$

We take a look at it with nprof-viewer, and select the display of two important measurements: code size and cnorm percentage. The code size of a section of code is roughly proportional to the execution time of that section. Cnorm percentage tells us the ratio of multiply-add operations (complex normal operations) over the total number of instructions (vliw).

Of course we want the first number to be small and the second big. In this way we have a fast program that saturate the arithmetic unit pipeline with calculations.

If we look at the section of code that actually perform the major calculations in nprof-viewer (figure 5) we can see that cnorm percentage is 0.9%, while codeSize is 81 instructions.

Now we try to add the pragma `#pragma for` before each for loop to tell the compiler that he should unroll these loops. Changes to the source code are reported in figure 4. We can ask ourself if this changes results in actually an improvement to the program performance.

Looking at nprof-viewer we can have a positive answer to this question. We see that codeSize for this section has increased. This seems to be a negative factor, but we have to take into account that the section is executed less often, due to the unrolling of the for loops.

If we look at the density of cnorm operations (figure 6) we find now a value of 3.8%. The improvement in this case is a factor of 4.2!

## 4 Conclusion

nprof and nprof-viewer are tools that can be effectively used by apeNEXT programmers as aid in the optimization process of their programs.

These tools can also be profitably used by compilers developers to test the gain in performance of their automatically generated code.

The profile infrastructure can be extended with new features. To help this process the graphical interface is designed in a way that new dynamic measurements can be added without modifications. Other minor improvements include source code highlighting, qualitative/high level reports of measurements.

## 5 Acknowledgments

I would like to thank the whole APE Group for their availability and kindness in every day work: Dirk, Laurent, Hani, Hubert, Nils. I would like especially to thank Dirk and Laurent that helped substantially during the process of writing the code and the documentation with prolific ideas and concrete aid.

I would like to thank also Karlheinz Hiller, Gregorio Landi and Elisabetta Gallo for making possible this beautiful experience as summer student in Zeuthen.

## References

[1] D. Pleiter et al., apeNEXT: "Experiences from Initial Operation", CHEP'06, Tata Institute of Fundamental Research, India, February 14, 2006

[2] D. Pleiter, "Code Optimization on apeNEXT", Bielefeld, December 13, 2003

[3] L. Morin, D. Pleiter, "Profiling Infrastructure for apeNEXT", DESY, 2006

[4] Salto Project Web site, INRIA, http://www.irisa.fr/caps/PROJECTS/Salto/

[5] APE Group DESY Zeuthen Web site, http://www-zeuthen.desy.de/ape