PerfExpert Tutorial

version 1.0

Author:

Geert Borstlap

Written/contributions by:

James C. Browne, Leonardo Fialho

Acknowledgement: TACC-staff, Susan H. Mehringer

Audience:

This PerfExpert Tutorial is designed for **HPC-Users** at the **University of Texas**, the members of the **Flemish Supercomputing Centre (VSC)**, being the **Antwerp University Association**, the **Brussels University Association**, the **Ghent University Association**, the **Hasselt University Association**, and the **KU Leuven Association** who want to profile and

optimise their applications.

The audience may be completely unaware of the profiling and optimisation concepts but must have some basic understanding of HPC programming.

Contents:

This tutorial gives answers to the typical questions that a new PerfExpert user may have. The aim is to learn how to use PerfExpert.

Part	Questions	Chapter	Title
Tutorial	What's PerfExpert?	**1**	Introduction
	How to profile and interpret the report?	**2**	Profiling
	How shall I optimise my code?	**3**	Optimising
	But what about my complex parallel program?	**4**	Multi-core & Multi-node Profiling
	What can I do more?	**5**	More PerfExpert Functionality

The **Annexes** contains some useful reference guides.

Part	Title	Chapter
Annex	Annex 1: PerfExpert Options	**A**
	Annex 2: Optimisation Patterns	**B**

Notification:

In this tutorial specific actions dealing with the software are separated from the accompanying text:

```
$ Actions (i.e., to be entered at a command line in your Terminal) in an exercise
    are preceded by a $ and printed in \strong{bold}. You'll find those actions in a
     grey frame.
```

"Directory" is the notation for directories or specific files. (e.g., " /examples/") **"Text"** Is the notation for text to be entered.

Tip: A "Tip" paragraph is used for remarks or tips.

More support:

Before starting the course, the example programs and configuration files used in this PerfExpert Tutorial must be copied to your own work directory.

If you have received a new VSC account, all examples are present in the tutorials directory. In order to have your own local copy, copy the full directory to your home directory.

```
$ cp -r /apps/brussel/tutorials/perfexpert/examples/ ~/
```

Apart from this Tutorial, the PerfExpert documentation on Texas Advanced Computing Center (TACC) website (`https://www.tacc.utexas.edu/perfexpert`) will serve as a reference for all PerfExpert operations.

Tip: The users are advised to get self-organised. There are only limited resources available at the TACC, which are best effort based. The user applications and own developed software remain solely the responsibility of the end-user.

Contact Information:

For all technical questions, please join the PerfExpert mailing list or contact the TACC staff: perfexpert-subscribe@lists.tacc.utexas.edu

We also welcome your feedback, comments and suggestions for improving the PerfExpert Tutorial via this mailing list.

Glossary

Cluster A group of compute nodes.

Compute Node The computational units on which batch or interactive jobs are processed. A compute node is pretty much comparable to a single personal computer. It contains one or more sockets, each holding a single processor or CPU. The compute node is equipped with memory (RAM) that is accessible by all its CPUs.

Core An individual compute unit inside a CPU.

CPU A single processing unit. A CPU is a consumable resource. Compute nodes typically contain of one or more CPUs.

Distributed memory system Computing system consisting of many compute nodes connected by a network, each with their own memory. Accessing memory on a neighbouring node is possible but requires explicit communication.

Flops Floating-point Operations Per second.

FTP File Transfer Protocol, used to copy files between distinct machines (over a network.) FTP is unencrypted, and as such blocked on certain systems. SFTP or SCP are secure alternatives to FTP.

Grid A group of clusters.

HPC High Performance Computing, high performance computing and multiple-task computing on a supercomputer. The VUB-HPC is the HPC infrastructure at the Free University of Brussels.

Infiniband A high speed switched fabric computer network communications link used in VUB-HPC.

Job constraints A set of conditions that must be fulfilled for the job to start.

L1d Level 1 data cache, often called **primary cache**, is a static memory integrated with processor core that is used to store data recently accessed by a processor and also data which may be required by the next operations..

L2d Level 2 data cache, also called **secondary cache**, is a memory that is used to store recently accessed data and also data, which may be required for the next operations. The goal of having the level 2 cache is to reduce data access time in cases when the same data was already accessed before..

L3d Level 3 data cache. Extra cache level built into motherboards between the microprocessor and the main memory..

LAN Local Area Network.

LCC The Last Level Cache is the last level in the memory hierarchy before main memory. Any memory requests missing here must be serviced by local or remote DRAM, with significant increase in latency when compared with data serviced by the cache memory..

Linux An operating system, similar to UNIX.

Login Node On VUB-HPC clusters, login nodes serve multiple functions. From a login node you can submit and monitor batch jobs, analyse computational results, run editors, plots, debuggers, compilers, do housekeeping chores as adjust shell settings, copy files and in general manage your account. You connect to these servers when want to start working on the VUB-HPC.

Memory A quantity of physical memory (RAM). Memory is provided by compute nodes. It is required as a constraint or consumed as a consumable resource by jobs. Within Moab, memory is tracked and reported in megabytes (MB).

Metrics A measure of some property, activity or performance of a computer sub-system. These metrics are visualised by graphs in, e.g., Ganglia.

Moab Moab is a job scheduler, which allocates resources for jobs that are requesting resources.

Modules VUB-HPC uses an open source software package called "Environment Modules" (Modules for short) which allows you to add various path definitions to your shell environment.

MPI MPI stands for Message-Passing Interface. It supports a parallel programming method designed for distributed memory systems, but can also be used well on shared memory systems.

Node Typically, a machine, one computer. A node is the fundamental object associated with compute resources.

Node Attribute A node attribute is a non-quantitative aspect of a node. Attributes typically describe the node itself or possibly aspects of various node resources such as processors or memory. While it is probably not optimal to aggregate node and resource attributes together in this manner, it is common practice. Common node attributes include processor architecture, operating system, and processor speed. Jobs often specify that resources be allocated from nodes possessing certain node attributes.

PBS, TORQUE or **OpenPBS** are Open Source resource managers, which are responsible for collecting status and health information from compute nodes and keeping track of jobs running in the system. It is also responsible for spawning the actual executable that is associated with a job, e.g., running the executable on the corresponding compute node. Client commands for submitting and managing jobs can be installed on any host, but in general are installed and used from the Login nodes.

Processor A processing unit. A processor is a consumable resource. Nodes typically consist of one or more processors. (same as CPU).

Queues PBS/TORQUE queues, or "classes" as Moab refers to them, represent groups of computing resources with specific parameters. A queue with a 12 hour runtime or "walltime" would allow jobs requesting 12 hours or less to use this queue.

scp Secure Copy is a protocol to copy files between distinct machines. SCP or scp is used extensively on VUB-HPC clusters to stage in data from outside resources.

Scratch Supercomputers generally have what is called scratch space: storage available for temporary use. Use the scratch filesystem when, for example you are downloading and uncompressing applications, reading and writing input/output data during a batch job, or when you work with large datasets. Scratch is generally a lot faster then the Data or Home filesystem.

sftp Secure File Transfer Protocol, used to copy files between distinct machines.

Shared memory system Computing system in which all of the processors share one global memory space. However, access times from a processor to different regions of memory are not necessarily uniform. This is called NUMA: Non-uniform memory access. Memory closer to the CPU your process is running on will generally be faster to access than memory that is closer to a different CPU. You can pin processes to a certain CPU to ensure they always use the same memory.

SSH Secure Shell (SSH), sometimes known as Secure Socket Shell, is a Unix-based command interface and protocol for securely getting access to a remote computer. It is widely used by network administrators to control Web and other kinds of servers remotely. SSH is actually a suite of three utilities - slogin, ssh, and scp - that are secure versions of the earlier UNIX utilities, rlogin, rsh, and rcp. SSH commands are encrypted and secure in several ways. Both ends of the client/server connection are authenticated using a digital certificate, and passwords are protected by encryption. Popular implementations include OpenSSH on Linux/Mac and Putty on Windows.

ssh-keys OpenSSH is a network connectivity tool, which encrypts all traffic including passwords to effectively eliminate eavesdropping, connection hijacking, and other network-level attacks. SSH-keys are part of the OpenSSH bundle. On VUB-HPC clusters, ssh-keys allow password-less access between compute nodes while running batch or interactive parallel jobs.

Super-computer A computer with an extremely high processing capacity or processing power.

Swap space A quantity of virtual memory available for use by batch jobs. Swap is a consumable resource provided by nodes and consumed by jobs.

TACC Texas Advanced Computing Center (creators of the PerfExpert tool).

TLB Translation Look-aside Buffer, a table in the processor's memory that contains information about the virtual memory pages the processor has accessed recently. The table cross-references a program's virtual addresses with the corresponding absolute addresses in physical memory that the program has most recently used. The TLB enables faster computing because it allows the address processing to take place independent of the normal address-translation pipeline..

UA University of Antwerp (creators of this tutorial).

Walltime Walltime is the length of time specified in the job-script for which the job will run on a batch system, you can visuallyse walltime as the time measured by a wall mounted clock (or your digital wrist watch). This is a computational resource.

Contents

Objective: to learn how to use PerfExpert for Program Optimisation

Quote:

The First Rule of Program Optimisation: Don't do it.

The Second Rule of Program Optimisation (for experts only!): Don't do it yet.

— Michael A. Jackson —

The Third Rule of Program Optimisation: Let the computer do it for you.

— The PerfExpert Team —

Chapter 1

Introduction

1.1 Purpose

HPC systems are notorious for operating at a small fraction of their peak performance, and the ongoing migration to multi-core and multi-socket compute nodes further complicates performance optimisation.

The previously available **performance optimisation tools** require considerable effort to learn and use. To enable wide access to performance optimisation, TACC and its technology insertion partners have developed PerfExpert, a tool that combines a simple user interface with a sophisticated analysis engine to:

1. **Detect** and **diagnosis** the causes for typical core, socket, and node-level performance bottlenecks in procedures and loops of an application;
2. Apply **pattern-based software transformations** on the application source code to enhance performance on identified bottlenecks;
3. Provide **performance analysis report** and **suggestions for remediation** for application's performance bottlenecks, which compilers are unable to apply automatically.

PerfExpert was developed at the Texas Advanced Computing Center (**TACC**) as an easy to use tool for diagnosing the performance of scientific applications. While most profiler tools give a sense of **where** performance bottlenecks and hotspots occur, PerfExpert aims to explain **why**, even going so far as to provide relevant code modification suggestions that could improve performance.

1.2 Under the hood

The performance of scientific computing applications is heavily influenced by memory access and floating-point unit (FPU) utilisation. Many modern CPUs contain a variety of hardware performance counter events that are useful in diagnosing resource utilisation and access patterns. These counters capture events such as L1 and L2 cache misses, branch instruction mispredictions, and execution of floating point instructions. PerfExpert collects these statistics and attempts to

correlate them with the code being analysed. Thus, individual functions or even separate loop nests can be characterised in terms of the kinds of CPU instructions and memory accesses they produce.

PerfExpert is built on top of two profiling tools that provide access to the **timing** and **hardware performance counter events figures**: PAPI and HPCToolkit.

1. **PAPI** is a library that provides access to CPU hardware performance counter events. Because the implementation of performance counters differs across CPU manufacturers and models, PAPI provides an abstracted interface that allows the user to retrieve the desired information regardless of the underlying OS or CPU architecture.
2. **HPCToolkit** is a profiling tool that uses periodic *statistical sampling* to read the performance counter events (using PAPI) and call structure of any executable compiled with debugging symbols. This approach is important because it can be run against executables compiled with full optimisation.

By combining the statistical sampling of program execution from HPCToolkit with the hardware performance counter events values from PAPI, an accurate correlation between CPU instructions, cache accesses, program structure, and execution time can be assembled. PerfExpert orchestrates the profiling process to provide easy to use reports and suggestions for improvement.

Summarised: PerfExpert aims to be an Easy-to-Use Performance Diagnosis Tool for HPC Applications.

1.3 Environment Configuration

First, connect to hydra of the Free University of Brussels:

```
$ ssh vsc10002@hydra.vub.ac.be
```

You can now create your own copy of the examples by copying the examples for this tutorial to your home directory.

```
$ cp -r /apps/brussel/tutorials/perfexpert/examples ~/
```

The runs with PerfExpert should be made using a data set size for each compute node which is representative to a full production runs but for which execution time is not more than about ten or fifteen minutes since PerfExpert will run your application multiple times (actually, three times on Stampede) with different performance counters enabled.

For that reason, before you run PerfExpert for your own purpose later, you should either request interactive access to computational resources (compute node), or modify the job script that you use to run your application and specify a running time that is about 3 (for hydra) times the normal running time of the program.

For the exercises in this tutorial, we request interactive access on a compute node to work on for, e.g., 2 hours:

```
$ qsub -l walltime=2:00:00 -l nodes=1:ppn=4 -I
```

PerfExpert depends on HPCToolkit, and PAPI library, thus it requires the papi, hpctoolkit, and perfexpert modules to be loaded. The "module help" command provides additional information.

And load the appropriate modules:

```
$ module load PerfExpert
```

```
$ module load openmpi/1.6.5/gcc/4.8.2
```

For more info on iterative access to a compute node on hydra, please, have a look on the introduction to VUB-HPC tutorial.

Chapter 2

Profiling

Tip: Find the source code for the examples in this Chapter in the directory: "~/examples/ch02-profiling".

2.1 Profiling Strategy

2.1.1 Introduction

Profiling ("program profiling", "software profiling") is a form of dynamic program analysis that measures, for example, the space (memory) or time complexity of a program, the usage of particular instructions, or frequency and duration of function calls. The most common use of profiling information is to aid program optimisation. Non-optimised programs will needlessly consume more computer resources on Stampede (computational, storage, network), which could otherwise reduce the execution time of your application or be made available to other users. Profiling can also help you find many other statistics through which many potential bugs can be spotted and sorted out. All users of Stampede are highly encouraged to profile their applications, before using the computing facilities.

Profiling is achieved by instrumenting either the program source code or its binary executable form using a tool called a *profiler* (or *code profiler*). Profilers, which are also programs themselves, analyse target programs by collecting information on their execution. Since profilers interrupt program execution to collect information, they have a finite resolution in the time measurements, which should be taken with a grain of salt.

Different profiling tools exist. These tools do not compete, but complement each other. The user can use *any* (or multiple) of these profiling tools to check and improve his application.

First we, go to the directory of the examples of Chapter02:

```
$ cd ~/examples/ch02-profiling
```

2.1.2 Profiling with PerfExpert

Source-code performance optimisation has four stages:

1. *measurement*,
2. *analysis and diagnosis of bottlenecks*,
3. determination of *optimisations*, and
4. *rewriting source code.*

Executing these steps for today's complex many processor and heterogeneous computer architectures requires a wide spectrum of knowledge and tools that many application developers would rather not have to learn.

PerfExpert utilises knowledge of architectures and compilers to implement (partial) automation of performance optimisation for multicore chips and heterogeneous nodes of cluster computers. PerfExpert automates the first three performance optimisation stages then implements those optimisations as part of the fourth stage.

Remember to load the appropriate PerfExpert modules:

```
$ module load PerfExpert
```

For our first example, we'll use a simple serial 1000×1000 matrix multiplication program. Check out the program "mm.c", compile and run it.

```
$ cat mm.c
$ gcc -o mm mm.c
$ ./mm
```

In order to see the total execution time of the program, you could time it:

```
$ time ./mm
real 0m7.554s
user 0m7.537s
sys 0m0.010s
```

Just adding "perfexpert" in front of your normal command line will activate the PerfExpert profiler. It will start the PerfExpert profiling tool, which will run the executable multiple times and collect the performance metrics.

We also have to define a threshold, which defines the relevance (in % of runtime) of the code fragments that PerfExpert should investigate. We want to focus our performance checks to those code segments, which consume the majority of the computer resources. This threshold value should be between 0 and 1. (> 0 and ≤ 1). A threshold of 0.3 means that PerfExpert will only focus on those functions, which are responsible for more than 30% of the runtime.

```
$ perfexpert 0.3 ./mm
[perfexpert] Collecting measurements [hpctoolkit]
[perfexpert]    [1] 8.270767878 seconds (includes measurement overhead)
[perfexpert]    [2] 9.886446243 seconds (includes measurement overhead)
[perfexpert]    [3] 8.258720295 seconds (includes measurement overhead)
[perfexpert] Analysing measurements
-------------------------------------------------------------------------
Total running time for source is 8.66 seconds between all 16 cores
The wall-clock time for source is approximately 0.54 seconds
Module source takes 100.00 of the total runtime
-------------------------------------------------------------------------
Loop in function compute in ~unknown-file~:0 (99.96% of the total runtime)
=========================================================================
ratio to total instrns     % 0.........25.........50.........75.........100
- floating point       93.3 ******************************************
- data accesses        12.0 ******
* GFLOPS (% max)        12.5 ******
- packed                0.0
- scalar               12.5 ******
-------------------------------------------------------------------------
performance assessment LCPI good.....okay.....fair..... poor
    .....bad
* overall              0.93 >>>>>>>>>>>>>>>>>>
* data accesses        1.08 >>>>>>>>>>>>>>>>>>>>
- L1d hits             0.42 >>>>>>>>
- L2d hits             0.57 >>>>>>>>>>
- L3d hits             0.09 >>
- LLC misses           0.00
* instruction accesses 0.01
- L1i hits             0.00
- L2i hits             0.00
- L2i misses           0.01
* data TLB             0.57 >>>>>>>>>>
* instruction TLB      0.00
* branch instructions  0.04 >
- correctly predicted  0.04 >
- mispredicted         0.00
* floating-point instr 1.63 >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
- slow FP instr        0.00
- fast FP instr        1.63 >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
=========================================================================
[perfexpert] Selecting optimizations
...
```

You notice that:

1. PerfExpert used the HPCToolkit to collect the performance measurements. It is using statistical sampling of timers and hardware performance counters to collect accurate measurements of the program.

2. The matrix multiplication program ran 3 times.

3. Each run took 8 to 9 seconds to complete. The average execution time was 8.66 seconds, which is a bit more as the execution time without profiling. It shows that profiling causes some extra execution overheads.

4. One function was detected in the program that took more than 30% of the total execution time. The "compute" function in an " unknown file " used 99.96% of the runtime.
5. A detailed analysis of the "compute" function was presented. We'll discuss this in depth in the next chapter.
6. Three recommendations to improve the speed of the "compute" function were suggested.

In this run, PerfExpert was not able to relate each hotspot with its exact location (line number) in source file because this information was not available in the object file. Programs compiled with the "-g" option will have debug information in the generated binary files, and PerfExpert will be able to report very specific on the line-number and source-file where a certain hotspot occurred.

Tip: Always compile your programs with the debug option (-g) on.

Tip: If you are using gcc 4.8 or newer, you should use -gdwarf-3 option!

Apart from the total running time, PerfExpert performance analysis report includes, for each hotspot (i.e., for each subroutine above the threshold):

1. *Efficiency of the Subroutine*
   (a) Instruction execution ratios (with respect to total instructions);
   (b) Computational efficiency (GFLOPs measurements);
2. *LCPI Performance assessment*
   (a) Overall performance;
   (b) Values for 6 categories:
      i. Data access
      ii. Data TLB
      iii. Instruction Access
      iv. Instruction TLB
      v. Branches
      vi. FP instructions

2.2 Profiling the Efficiency of a subroutine

2.2.1 Instruction Execution Rations

The program composition part shows what percentages of the total instructions were *computational* (floating-point instructions) and what percentage were instructions that *accessed data.* Although there are other kinds of instructions, these two groups should be the most relevant on any given scientific application.

```
ratio to total instrns     % 0.........25.........50.........75.........100
- floating point       93.3 ****************************************
- data accesses        12.0 ******
```

A software program spend its time on 3 types of instructions:

1. **Computations**: Typically, we want our program to be busy with computations (floating-point or other) to solve our scientific problem, and as such a high percentage for FP computations is considered to be good and desired.
2. **Data Access:** The computations will need to read input data and will generate output data. Data access (preferably from the fastest lowest cache levels) is needed, but the overall times spend on reading and writing data shall be limited as it is not the core goal of our program. A high percentage of time spend on data access might indicate data access issues.
3. **Program logic:** The time spent on other instructions (control flow, testing, loops, branching) can be considered as a "waste" of time and should be low. In most applications, this amount is very low, and as such we do not generate metrics for them. The percentages for *Computations*, *Data Access* and *Program Logic* should total 100%[1], so we can easily estimate the % for this category.

These ratios gives a rough estimate in trying to understand whether optimising the program for either *data accesses* or *floating-point instructions* would have a significant impact on the total running time of the program.

2.2.2 Computational Efficiency (GFLOPS)

The PerfExpert performance analysis report also shows the GFLOPs rating, which is the number of floating-point operations executed per second in multiples of 10^9. The value for the GFLOPS metric is displayed as a percentage of the maximum possible achievable GFLOP rate for that particular machine.

In the previous Matrix-Multiplication example, our GFLOP values were:

```
* GFLOPS (% max)        12.5 ******
  - packed                0.0
  - scalar               12.5 ******
```

The higher the GFLOPS percentage, the more efficient are computations are done. Although it is rare for real-world programs to match even 50% of the maximum value, this metric can serve as an estimate of how efficiently the code performs computations.

This performance report gives us also an indication about the "vectorisation degree", i.e., the percentage of the computations that were "vectorised" (= packed) and run in parallel.

```
* GFLOPS (% max)        12.5 ******
  - packed                0.0
  - scalar               12.5 ******
```

1. *Packed*: This represents the % of FP instructions which were vectorised. A value of 0.0 means that no vector FP computations were done. We want this value to be high.

[1]Some architectures such as Intel Sandy Bridge, Intel Ivy Bridge, and Intel Haswell does not provide accurate values in their performance counter events, specially for the floating-point instructions. For that reason, it is possible to see application with more than 100% of floating-point instruction, with is obviously not possible.

2. *Scalar*: This represents the % of FP instructions which were not vectorised. These computations were executed one by one in serial mode and the value should be low.

You noticed that the Matrix Multiplication program was not able to "*vectorise*" (Packed was 0%) and that all our FP computations were executed sequentially (Scalar was 12.5%).

2.3 Profiling the LCPI Performance Categories

The next, and major, section of the PerfExpert performance analysis report shows the LCPI values, which is the ratio of cycles spent in the code segment for a specific category, divided by the total number of instructions in the code segment.

2.3.1 Overall Performance of the Function

The overall value is the ratio of the total cycles taken by the code segment to the total instructions executed in the code segment. It gives a sense of the average cost in cycles of all instructions in a code segment.

```
performance assessment  LCPI good........okay.........fair.........poor.............
    bad
* overall               0.93 >>>>>>>>>>>>>>>>>>>
```

The LCPI value is a good indicator of the cost arising from instructions of the specific category.

Hence, the higher the LCPI, the slower the program.

Generally, a value of 0.5 or lower for an LCPI is considered to be good. However, it is only necessary to look at the ratings (good, okay, . . . , bad).

The rest of the report maps this overall LCPI, into the six constituent categories: *data accesses*, *instruction accesses*, *data TLB accesses*, *instruction TLB accesses*, *branches* and *floating point computations*. Without getting into the details of instruction operation on Intel and AMD chips, one can say that these six categories record performance in non-overlapping ways. That is, they roughly represent six separate categories of performance for any application.

Individual machine instructions or events can vary between architectures, and are generally not common knowledge. In order to cope with this, PerfExpert groups instructions into categories and displays LCPI values in terms of these categories. There is an architecture-dependent mapping between specific instructions and PerfExpert categories behind the scenes, but a PerfExpert user does not need to be aware of these details.

Remarks

In each case, the classification (data access, instruction access, data TLB, etc.) is shown so that it is easy to understand which category is responsible for the performance slowdown. For instance if the overall LCPI is poor and the data access LCPI is high, then you should concentrate on improving the access to program variables and memory. Additional LCPI details help in relating performance numbers to the process architecture.

While high LCPI values may indicate costly and inefficient use of certain computing resources,

they do not specify what should be done to make the code segment more efficient. PerfExpert uses a set of rules, which map the LCPI metric values to one or more specific modifications (recommendations or suggestions for optimisation) to the code segment to correct the inefficiency in the code segment. We will discuss this recommendation system later in chapter 4.

Before we can enhance or speed-up your own program, you must fully be able to interpret the performance report. We will use some lab exercises to build up this competence.

We will change the matrix multiplication program in an "evil" manner, to deliberately generate bad performance reports. The goal is to let the user understand the potential causes of bad performance and to teach him to interpret the performance report.

2.3.2 Category #1: Data Access

The Data accesses counts the LCPI arising from accesses to memory for program variables. Measures the cost of memory and cache accesses for variables.

High LCPI values in this category generally indicate poor memory access patterns. L1 and L2 cache hits and LLC misses are included in this category.

```
* data accesses          1.08 >>>>>>>>>>>>>>>>>>>>>
  - L1d hits             0.42 >>>>>>>>
  - L2d hits             0.57 >>>>>>>>>>>
  - L3d hits             0.09 >>
  - LLC misses           0.00
```

We will start from the previous matrix multiplication program and generate a program with inefficient data-access. Instead of looping correctly through the matrices ("row from matrix a" $\times$ "column from matrix b"), we will now access both matrices "a" and "b" in a vertical manner (column by column). *We do of course not aim to perform a correct matrix-multiplication*, as we are not interested in the results. We just want to show the effect of accessing data in a scattered (non-linear) way and causing *non-optimal data access locality*.

Explore the test1.c program.

```
$ cat test1.c
```

— test1.c —

```c
//
// test1.c
// Test program to invoke data misses
//
#include <stdlib.h>
#include <stdio.h>

// You can play with this dimension, and see the effect on performance
#define n 1000

static double a[n][n], b[n][n], c[n][n];

void compute()
{
  register int i, j, k;
  for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
      for (k = 0; k < n; k++)
        c[i][j] += a[k][i] * b[k][j];
}

int main(int argc, char *argv[])
{
  register int i, j, k;

  // Initialize the matrices
  for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) {
      a[i][j] = i+j+0.1;
      b[i][j] = i-j+0.2;
      c[i][j] = 0;
    }
  }

  // And do a matrix multiplication
  compute();

  // Some compilers optimize the full "compute" functions away, when it detects that
     its computed elements are not being used.
  // As such, we print the first and last elements of the generated matrix
  printf ("c[0][0]=%f\n", c[0][0]);
  printf ("c[%d][%d]=%f\n", n-1, n-1, c[n-1][n-1]);

  return 0;
}
```

And compile and run PerfExpert.

We will now compile with the "-g" option, which will include debug information in the generated binary files. With this debug information, PerfExpert will be able to relate each function with a location (line number) in a specific file. A Makefile is provided in the examples directory, so as from now, one could also use "make" to compile all the programs at once.

The "-c" option in PerfExpert will print colours.

```
$ gcc -g -o test1 test1.c
$ perfexpert -c 0.9 ./test1
[perfexpert] Collecting measurements [hpctoolkit]
[perfexpert]    [1] 20.045568079 seconds (includes measurement overhead)
[perfexpert]    [2] 21.401645102 seconds (includes measurement overhead)
[perfexpert]    [3] 16.191897395 seconds (includes measurement overhead)
[perfexpert] Analysing measurements
-----------------------------------------------------------------------------
Total running time for test1 is 22.94 seconds between all 16 cores
The wall-clock time for test1 is approximately 1.43 seconds
Module test1 takes 100.00% of the total runtime
-----------------------------------------------------------------------------
Loop in function compute in test1.c:21 (99.98% of the total runtime)
=============================================================================
ratio to total instrns    %  0.........25.........50.........75.........100
 - floating point      100.0 ***********************************************
 - data accesses        12.0 ******
* GFLOPS (% max)        10.2 *****
 - packed                0.0
 - scalar               10.2 *****
-----------------------------------------------------------------------------
performance assessment  LCPI good..........okay........s.fair........poor
    .............bad
colorred* overall 2.47 »»»»»»»»»»»»»»»»»»»»»»»>
* data accesses         2.94 >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
 - L1d hits             0.42 >>>>>>>>
 - L2d hits             1.03 >>>>>>>>>>>>>>>>>>>>>
 - L3d hits             1.22 >>>>>>>>>>>>>>>>>>>>>>>>
 - LLC misses           0.26 >>>>>
* instruction accesses  0.02
 - L1i hits              0.00
 - L2i hits             0.00
 - L2i misses           0.02
* data TLB              1.43 >>>>>>>>>>>>>>>>>>>>>>>>>>>>>
* instruction TLB       0.00
* branch instructions   0.04 >
 - correctly predicted  0.04 >
 - mispredicted         0.00
* floating-point instr  3.50 >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>+
 - slow FP instr        0.00
 - fast FP instr        3.50 >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>+
=============================================================================
```

We notice that:

1. The average *execution time* more than doubles, from 8 seconds to 20 seconds.
2. The *overall* performance is bad. (The line in red)
3. And we see a lot of *data access* spread over the L1, L2 and L3 data spaces, and a lot of LCC (last Level Cache) misses. It is disastrous.
4. The *data TLB* is high again, which is related to non-optimal data access.
5. The *instruction TLB and instruction accesses* are good. This means that our (part of the) program is loaded in the instruction cache once, and remains there. This behaviour is expected as we have a very small program.

6. There are little *branch instructions*, all of which have been correctly predicted. There are no complex *branch* instructions (if, else, switch or goto statements) in our small program.

7. We have quite a lot of *floating point instructions*. This was expected, as it was the core functionality of our program. We only have *fast* Floating Point instructions, as we did not program many "expensive" (slow) divisions and/or square-roots calculations.

This *analysis* indicates a *performance problem related to data access* in the "compute" function. This function is responsible for 99.98% of the total runtime, so an improvement might heavily impact to total runtime.

Tip: It is interesting to run this example with full (-O3) and without (-O0) compiler optimisation and check the difference in runtime.

```
$ gcc -O3 -g -o test1 test1.c
$ perfexpert -c 0.8 ./test1
$ gcc -O0 -g -o test1 test1.c
$ perfexpert -c 0.7 ./test1
```

You will notice that our performance issue is design-related, and thus the compiler is not (or hardly) able to optimise our program.

2.3.3 Category #2: Instruction Access

The Instruction accesses count the LCPI arising from memory accesses for code (functions and loops) and it measures the cost of fetching instructions from memory. Tight, small, non-branchy loops tend to have the best LCPI values.

```
* instruction accesses  0.01
 - L1i hits             0.00
 - L2i hits             0.00
 - L2i misses           0.01
```

This metric will rarely cause problems in scientific programming. Programs with a high(er) values of this metric are typically huge (in bytes) and may have a lot of function calls (mostly) to statically linked libraries. It is however difficult to simulate this behaviour in a small comprehensive program.

2.3.4 Category #3: Data TLB

The Data Translation Lookaside Buffer (TLB) is used for mapping physical memory address to pages in a virtual memory subsystem. Typically, it can only contain a mapping for a small subset of pages. Access to pages outside of this buffer requires an update of the buffer, as well as other memory accounting, and it is expensive. Typically, this happens when memory accesses are not close to one another, such as random access or non-unit stride.

```
* data TLB              0.57 >>>>>>>>>>
```

High LCPI values indicate lots of TLB updates, and imply poor memory locality. The Data TLB provides an approximate measure of penalty arising from long strides in accesses or irregularity of accesses.

Probably the most common cause for TLB misses is due to memory being accesses with large, often constant, distances between them. The distance between memory accesses is usually referred to as a stride. Unitary stride is used to describe memory accesses that are sequential, i.e., the next element is exactly one element away. A stride of 5 refers to a memory access pattern where every 5^{th} element in a list is accessed, skipping over 4 elements at a time. Multi-dimensional arrays have inherently long strides, depending on how they are accessed.

Explore the test3.c program.

```
$ cat test3.c
...
for (j = 0; j $<$ MAX_J; j++)
 for (i = 0; i $<$ MAX_I; i++)
   a[i*STRIDE+j] += 2;
...
```

And compile and run PerfExpert.

We do not want to see the recommendations yet, so we add the "-r0" option. (r0 = give me zero recommendations)

```
$ gcc -g -o test3 test3.c
$ perfexpert -r0 -c 0.9 ./test3
[perfexpert] Collecting measurements [hpctoolkit]
[perfexpert]    [1] 6.756021233 seconds (includes measurement overhead)
[perfexpert]    [2] 8.094997327 seconds (includes measurement overhead)
[perfexpert]    [3] 6.352170922 seconds (includes measurement overhead)
[perfexpert] Analysing measurements
----------------------------------------------------------------------------
Total running time for test3 is 5.37 seconds between all 16 cores
The wall-clock time for test3 is approximately 0.34 seconds
Module test3 takes 100.00% of the total runtime
----------------------------------------------------------------------------
Loop in function compute in test3.c:0 (95.24% of the total runtime)
============================================================================
ratio to total instrns    %  0.........25.........50.........75.........100
 - floating point      100.0 **********************************************
 - data accesses         7.3 ****
* GFLOPS (% max)        12.5 ******
 - packed                0.0
 - scalar               12.5 ******
----------------------------------------------------------------------------
performance assessment  LCPI good.......okay.......fair.......poor.......bad
* overall 3.66 »»»»»»»»»»»»»»»»»»»»»»»+
* data accesses        19.48 >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>+
 - L1d hits             0.26 >>>>>
 - L2d hits             0.94 >>>>>>>>>>>>>>>>>>>
 - L3d hits             0.08 >>
 - LLC misses          18.21 >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>+
* instruction accesses  0.00
 - L1i hits             0.00
 - L2i hits             0.00
 - L2i misses           0.00
* data TLB              1.03 >>>>>>>>>>>>>>>>>>>>>
* instruction TLB       0.00
* branch instructions   0.07 >
 - correctly predicted  0.07 >
 - mispredicted         0.00
* floating-point instr  6.36 >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
 - slow FP instr        0.00
 - fast FP instr        6.36 >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
=========================================================================================
```

As aforementioned, mappings between virtual and physical memory are facilitated by a page table, which is kept in memory. To minimise references to this table, recently-used portions of the page table are cached in a hierarchy of “translation lookaside buffers”, or TLBs, which are consulted on every virtual address translation. As with data caches, the farther a request has to go to be satisfied, the worse the performance impact. These metric estimates the performance penalty paid for missing the first-level data TLB (DTLB) that includes hitting in the second-level data TLB (STLB) as well as performing a hardware page walks on an STLB miss.

We notice that:

1. The *overall* performance is bad. (The line in red)
2. And we see many *data access* spread over the L1, L2 and L3 data spaces, and a lot of LCC (last Level Cache) misses.

3. We see a lot of data TLB misses too, which are very related to the other cache misses.
4. We see high LCPI values for floating-point instructions, however, as floating-point instructions usually requires data access, the high LCPI values in this case are mostly a consequence of the high data access LPCI values.

2.3.5 Category #4: Instruction TLB

The Instruction TLB reflects cost of fetching instructions due to irregular accesses. High LCPI values here indicate poor memory locality when fetching instructions.

```
* instruction TLB       0.00
```

As for Category #2 (Instruction Access), high values for this metric do rarely occur. You can expect higher values in big programs (i.e., programs which the binary size is in the order of hundred of megabytes).

2.3.6 Category #5: Branch Instructions

The Branch Instructions counts cost of jumps (i.e., if statements, loop conditions, etc.). This measures the penalty from jumps as a result of conditionals or loops. High LCPI values may indicate branchy, unpredictable code. Branch LCPIs can be divided into LCPIs from correctly predicted and from mispredicted branch instructions.

```
* branch instructions   0.04 >
 - correctly predicted  0.04 >
 - mispredicted         0.00
```

The performance of a branch statement (e.g., if-elseif-else, switch) depends on whether its condition has a predictable pattern. If the condition is mostly true or mostly false, the branch predictor logic in the processor will pick up the pattern. On the other hand, if the pattern is unpredictable, the execution of such branch-statement will be much more expensive.

In modern CPUs, the processor tries and fetches instructions from memory before they are needed as otherwise the CPU has to wait for the instruction (stall). This is called pre-fetching and the instructions are held in an instruction pipeline. Of course, the flow of the application is not always known at that moment, e.g., in case of a branch. Therefore, a specific digital circuit was designed (called a *branch predictor*) that tries to guess which way a branch (e.g., an if-then-else structure) will go before this is known for sure. The purpose of the branch predictor is to improve the instruction pipeline throughput. Branch predictors play a critical role in achieving high effective performance in many modern pipelined microprocessor architectures.

Branches (i.e., conditional jumps) present a difficulty for the processor pipeline. After fetching a branch instruction, the processor needs to fetch the next instruction. But, there are *multiple possible* "next" instructions! The processor won't be sure which instruction is the next one until the branching instruction makes it to the end of the pipeline.

Instead of stalling the pipeline until the branching instruction is fully executed, modern processors attempt to *predict* whether the jump will or will not be taken. Then, the processor can fetch

the instruction that it thinks is the next one. If the prediction turns out wrong, the processor will simply discard the partially executed instructions that are in the pipeline and the pipeline starts over with the correct branch, incurring a delay.[2]

The *test5a* program will create a large array of randomly obtained values of 0's, 1's, 2's and 3's. We loop through the array and branch for each value, doing time-wasting work. As all the values were randomly obtained, we expect that the computer is not able to predict (find a pattern) in the branches.

Explore the test5a.c program:

```
$ cat test5a.c
```

— test5a.c —

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <time.h>
4
5  // You can play with this dimension, and see the effect on performance
6  #define ARRAY_LEN 400000000
7  int a[ARRAY_LEN];
8
9  void compute()
10 {
11   int i;
12   int zero = 0, one = 0, two = 0, three = 0;
13
14   for(i = 0; i < ARRAY_LEN; ++i)
15   {
16     if (a[i] == 0)
17       ++zero;
18     else if (a[i] == 1)
19       ++one;
20     else if (a[i] == 2)
21       ++two;
22     else
23       ++three;
24   }
25   printf ("zero (%d), one (%d), two (%d), three (%d)\n", zero, one, two, three);
26 }
27
28 int main()
29 {
30   int i;
31
32   // Generate a new array...
33   srand ((unsigned int)time(NULL));
34   for(i = 0; i < ARRAY_LEN; ++i)
35     a[i] = rand() % 4;
36
37   compute();
38 }
```

[2]There is another technique known as "speculative execution" which actually executes more than one branch possibility, however, this tutorial is not intended to cover this technique.

In our example, all the values in the branch are completely randomly obtained, so the computer has a chance of 1 out of 4 to make a correct prediction.

Lets compile and run the example:

```
$ gcc -g -o test5a test5a.c
$ perfexpert -c 0.1 ./test5a
[perfexpert] Collecting measurements [hpctoolkit]
[perfexpert]    [1] 8.639570165 seconds (includes measurement overhead)
[perfexpert]    [2] 8.567133423 seconds (includes measurement overhead)
[perfexpert]    [3] 8.294661367 seconds (includes measurement overhead)
[perfexpert] Analysing measurements
-------------------------------------------------------------------------------
Total running time for test5a is 9.06 seconds between all 16 cores
The wall-clock time for test5a is approximately 0.57 seconds
Module test5a takes 64.77% of the total runtime
Module libc-2.12.so takes 35.23% of the total runtime
-------------------------------------------------------------------------------
Loop in function compute in test5a.c:12 (49.20% of the total runtime)
===============================================================================
ratio to total instrns    %  0..........25.........50.........75.........100
 - floating point      100.0 ********************************************
 - data accesses        46.8 **********************
* GFLOPS (% max)       12.5 ******
 - packed               0.0
 - scalar              12.5 ******
-------------------------------------------------------------------------------
performance assessment  LCPI good.......okay.......fair.......poor.......bad
* overall 1.88 »»»»»»»»»»»»»»»»»»»»»»»»
* data accesses         1.67 >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
 - L1d hits             1.64 >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
 - L2d hits             0.01
 - L3d hits             0.00
 - LLC misses           0.02
* instruction accesses  0.00
 - L1i hits             0.00
 - L2i hits             0.00
 - L2i misses           0.00
* data TLB              0.00
* instruction TLB       0.00
* branch instructions   1.14 >>>>>>>>>>>>>>>>>>>>>>>>
 - correctly predicted  0.25 >>>>>
 - mispredicted         0.89 >>>>>>>>>>>>>>>>>>
* floating-point instr  3.27 >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
 - slow FP instr        0.00
 - fast FP instr        3.27 >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
```

We notice that:

1. The *overall* performance is poor. (The line in red)
2. Data access, instruction access, data TLB and Instruction TLB are all fine.
3. As expected, we see a lot of mispredicted branches. Around 25% is correctly predicted while 75% is mispredicted.

The time that is wasted in the case of a branch misprediction is equal to the number of stages

in the pipeline from the fetch stage to the execute stage. Modern microprocessors tend to have quite long pipelines so that the misprediction delay is between 10 and 20 clock cycles. The longer the pipeline the greater the need for a good branch prediction.

In order to compare the effect of the branch predictor when it is able to provide good predictions, we also prepared a similar program (i.e., test5b.c), but now with completely predictable branches. We alternatively put the values 0, 1, 2 and 3 (= i%4) in the array.

Explore, compile, run the example and check the results, paying particular attention to the overall execution time and the mispredicted branches.

```
$ cat test5b.c
...
 for(i = 0; i $<$ ARRAY\_LEN; ++i)
   a[i] = i PERCENTAGE-SIGN 4;
...
$ gcc -g -o test5b test5b.c
$ perfexpert -c 0.3 ./test5b
...
[perfexpert] Collecting measurements [hpctoolkit]
[perfexpert]    [1] 6.687911876 seconds (includes measurement overhead)
[perfexpert]    [2] 6.677188251 seconds (includes measurement overhead)
[perfexpert]    [3] 6.660273106 seconds (includes measurement overhead)
[perfexpert] Analysing measurements
* branch instructions   0.17 >>>
 - correctly predicted  0.17 >>>
 - mispredicted         0.00
...
```

The branch predictor in the processor has detected the pattern and is now able to completely predict the branches, which has its effect on the execution time of the program.

How come? The first time a conditional jump instruction is encountered, there is not much information to base a prediction on. But the branch predictor keeps records of whether branches are taken or not taken. When it encounters a conditional jump that has been seen several times before then it can base the prediction on the history. The branch predictor may, for example, recognise that the conditional jump is taken more often than not, or that it is taken every second time. In our case, the branch predictor was able to follow the sequential 1-2-3-4 logic.

2.3.7 Category #6: Floating Point instructions

The floating-point instructions count LCPI from executing computational (floating-point) instructions. High LCPI values can indicate non-vectorised floating-point operations, expensive operations like calculating square roots, and most commonly stalls caused by poor data access.

```
* floating-point instr  1.63 >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
 - slow FP instr        0.00
 - fast FP instr        1.63 >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
```

For floating-point instructions, the division is based on floating-point instructions that take few cycles to execute (e.g., add, subtract and multiply instructions) and on floating-point instructions that take longer to execute (e.g., divide and square-root instructions).

We will start again from the previous matrix multiplication program and generate a program with "expensive" slow floating-point instructions. We have added a number of square-root ("sqrt") and division ("/") operations in the computation. We just want to show the effect of these "slow" instructions on the overall performance.

Explore the test6.c program.

```
$ cat test6.c
```

— test6.c —

```c
//
// Test program to show the effect of adding sqrt computations to the performance report
//
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

// You can play with this dimension, and see the effect on performance
#define n 1000

static double a[n][n], b[n][n], c[n][n];

void compute()
{
  register int i, j, k;
  for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
      for (k = 0; k < n; k++)
        c[i][j] += sqrt(c[i][j] + sqrt(a[i][k]) / sqrt(b[k][j]));
}

int main(int argc, char *argv[])
{
  register int i, j, k;

  // Initialize the matrices
  for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) {
      a[i][j] = i+j;
      b[i][j] = i-j;
      c[i][j] = 0;
    }
  }

  // And do a matrix multiplication
  compute();

  // Some compilers optimize the full "compute" functions away, when it detects that its computed elements are not being used.
  // As such, we print the first and last elements of the generated matrix
  printf ("c[0][0]=%f\n", c[0][0]);
  printf ("c[%d][%d]=%f\n", n-1, n-1, c[n-1][n-1]);

  return 0;
}
```

We have to compile with the "–lm" option, to make sure that the mathematical library is linked into our executable.

```
$ gcc -lm -g -o test6 test6.c
$ perfexpert -c 0.5 ./test6
[perfexpert] Collecting measurements [hpctoolkit]
[perfexpert]    [1] 38.660883281 seconds (includes measurement overhead)
[perfexpert]    [2] 36.109102082 seconds (includes measurement overhead)
[perfexpert]    [3] 31.387831572 seconds (includes measurement overhead)
[perfexpert] Analysing measurements
-------------------------------------------------------------------------
Total running time for test6 is 28.57 seconds between all 16 cores
The wall-clock time for test6 is approximately 1.79 seconds
Module test6 takes 71.60% of the total runtime
Module libm-2.12.so takes 27.86% of the total runtime
Module libc-2.12.so takes 0.54% of the total runtime
-------------------------------------------------------------------------
Loop in function compute in test6.c:17 (71.47% of the total runtime)
=========================================================================
ratio to total instrns    %  0.........25.........50.........75.........100
 - floating point       63.7 ********************************
 - data accesses        31.2 ****************
* GFLOPS (% max)         9.5 *****
 - packed                0.0
 - scalar                9.5 *****
-------------------------------------------------------------------------
performance assessment  LCPI good.......okay.......fair.......poor.......bad
* overall 0.84 »»»»»»»»»»>
* data accesses         1.38 >>>>>>>>>>>>>>>>>>>>>>>>>>>>
 - L1d hits             1.09 >>>>>>>>>>>>>>>>>>>>>>>
 - L2d hits             0.03 >
 - L3d hits             0.00
 - LLC misses           0.26 >>>>>
* instruction accesses  0.01
 - L1i hits             0.00
 - L2i hits             0.00
 - L2i misses           0.01
* data TLB              0.14 >>>
* instruction TLB       0.00
* branch instructions   0.35 >>>>>>>
 - correctly predicted  0.23 >>>>>
 - mispredicted         0.12 >>
* floating-point instr  1.73 >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
 - slow FP instr        0.62 >>>>>>>>>>>>
 - fast FP instr        1.11 >>>>>>>>>>>>>>>>>>>>>>>>
=========================================================================
```

We notice in this example that:

1. The average *execution time* more than quadruples, from 8 seconds to 33 seconds.
2. The square-root function ("sqrt") in the mathematical library (libm) consumes about 27% of the total runtime.
3. The *overall* performance is okay to fair. (The line in yellow).
4. Most *data access* occurs in the L1 cache (which is perfect) and we only have limited LCC (last Level Cache) misses.

5. The *data TLB* is low.
6. The *instruction TLB and instruction accesses* are perfect.
7. Also the *branch instructions* look okay. Although we do not written branches in our code, loop control flow generates a number of intrinsic branches.
8. But we have a lot of *floating point instructions*. This was expected, as it was the core functionality of our program. We have a considerable amount of *slow* Floating Point instructions (division and square-root), which will be causing the slow performance.

This *analysis* indicates a *performance* issue related to the *slow floating-point calculations*. Just the square root operations (calls to libm) are responsible for 28% of the runtime, so an improvement might heavily impact to total runtime.

As a general rule of thumb: Both floating-point division and square root are considered as slow operations. Addition, subtraction and multiplication are fast FP operations. Square root can be expect to be approximately the same speed or somewhat slower (i.e., approx. $1\times$ - $2\times$ lower performance) compared to a division.

Chapter 3

Optimising

Tip: Find the source code for the examples in this chapter in the directory: "~/examples/ch03-optimizing".

First we, go to the directory of the examples of Chapter03:

```
$ cd ~/examples/ch03-optimizing
```

The blocks of code where time is being wasted during execution are called *hotspots*.

Modern compilers are good at low-level code optimisation but they are limited by their ability to analyse the code and identify opportunities for optimisation. The programmer can help the compiler by using a clear and simple programming style and by avoiding code constructs that are unnecessary complicated. All more advanced optimisations have to be done by the programmer on the level of the source code, when he designs and develops the data structures and algorithms. Therefore, the developer needs an understanding of the program, how the data is stored in the computer memory, how it is accessed, and most important the impact of the data access in the memory subsystem (cache and TLB). Code optimisation is therefore best done as part of the program development process.

In this chapter, PerfExpert will

1. find the hotspots in several code samples, and will also
2. guide us in the optimisation of the code by suggestion optimisations.

3.1 Optimisation Strategy

3.1.1 Find the Hotspots

In order to optimise a program, we need to know where the program spends most of its execution time. These are called *hotspots* in the execution. Optimising code that is not a hotspot doesn't speed up the execution and may instead make the program more difficult to understand and maintain.

When profiling and measuring the execution of a program, we should execute the program to solve a *typical case*, i.e., a problem instance that is highly representative for most use of the program.

PerfExpert has one single mandatory option: the *threshold* (relevance in % of the total runtime) to take hotspots into consideration. Its value range varies between 0 and 1. A value of 0.25 means that PerfExpert will only investigate code blocks that are responsible for more than 25% of the execution time.

PerfExpert can sort the hotspots according to:

1. **Relevance**: Sorted according to their relevance in % of the total runtime. The code blocks were the execution time is higher will be presented on top. But those code blocks might already be programmed in a full professional and well-performing manner, and maybe cannot be optimised anymore.
2. **Performance**: Sorted according to their performance. The code blocks with the worst performance will be presented on top. These code blocks bears the potential to generate the best performance improvement.
3. **Mixed or Impact:** Sorted according to the formula *Relevance* × *Performance*. The code blocks with a combination of the worst performance and longest execution times (thus having the highest *impact*) are presented on top. Those are the best candidates for optimisation.

By default, PerfExpert will list its hotspots unsorted.

The test.c program has several slow (bad programmed) functions, i.e., compute1, compute2 and compute3. Explore and compile the test.c program, and explore the 3 results:

```
$ more test.c
$ gcc -g -o test test.c
$ perfexpert -order=relevance -c 0.2 ./test
$ perfexpert -order=performance -c 0.2 ./test
$ perfexpert -order=mixed -c 0.1 ./test
```

3.1.2 Optimisation Priority

A good optimisation strategy is to run PerfExpert with a high threshold (e.g., 0.3) and first focus on those functions, which consumes most of the computation time. You can then systematically decrease it (0.3 » 0.2 » 0.1) to also cover other functions.

Some LCPI categories have a higher potential to speed up your code as compared to other. When optimising, experience has learnt that it is wise to focus on the following order of importance:

1. **Data Access** & **Data TLB** issues
2. **Vectorisation**
3. **Others** such as:

(a) Branches
(b) FP operations
(c) Instructions (which are unlikely to have a huge performance impact)

The Working Method to optimise is an iterative process:

1. **Step #1 – Measure & Analyse:** The first step is to run PerfExpert and to understand and interpret the report. This analysis should give you an idea where the bottlenecks are.
2. **Step #2 - Adapt:** With this information, you can (try to) improve the performance by adapting the program. In order to help you in the right directions, some recommendations are given for each hotspot.
3. **Step #3 - Re-test:** Subsequent invocations of PerfExpert shall be run in order to quantify the effect of implementing a change to the code.

You can repeat these steps in an iterative way until you're comfortable with the performance of your program.

3.1.3 PerfExpert Recommendations

Based on the characteristics revealed during profiling, PerfExpert is able to provide specific recommendations for changes that could lead to improved performance. These recommendations come from a database of scenarios that correspond to certain patterns of CPU resources utilisation.

These recommendations are chosen from an extensive database of scenarios based upon the profiling results, not necessarily code analysis. Certain recommendations may not be applicable to the code base being analysed, nor may they necessarily improve performance. It is up to the user to understand the code being analysed, and then determine if a given recommendation makes sense.

All the possible recommendations are listed and described in Annex 2: Optimisation. Future version of PerfExpert may extend this list of recommendations.

3.2 Optimising the efficiency of a subroutine

3.2.1 Optimising Instruction Execution Ratios

Optimising the "Instruction Ratios" means most of the time increase vectorisation and lower the instruction percentages for "Data access". You will use the specific metrics of the Data Access categories (L1, L2, L3, LLC misses and Data TLB), which will be discussed later on.

But we want to show you the effect that "Accessing Data" has on the overall performance. We have slightly adapted the matrix multiplication program, so that it uses the values of a[i][k] (which was i+k) and b[k][j] (which was k-j) directly:

Explore the mm1.c program.

```
$ cat mm1.c
...
void compute()
{
 register int i, j, k;
 for (i = 0; i $<$ n; i++)
   for (j = 0; j $<$ n; j++)
     for (k = 0; k $<$ n; k++)
       c[i][j] = (i+k) * (k-j);
}
...
```

And compile and run PerfExpert.

The "-c" option in PerfExpert will print colours.

```
$ gcc -g -o mm1 mm1.c
$ perfexpert -c 0.9 ./mm1
...
Loop in function compute in mm1.c:12 (99.76% of the total runtime)
================================================================================
The performance of this code section is good!
--------------------------------------------------------------------------------
ratio to total instrns    %  0.........25.........50.........75.........100
 - floating point       32.1 ***************
 - data accesses         0.0
...
```

We have now avoided the reference to our matrices "a" and "b" inside the loop. You see that the percentage of time needed for data accesses was reduced to zero.

Note: Of course, your program needs to perform the work. It is not always acceptable to reduce the amount of time spent on data access by just not doing any useful computation!

3.2.2 Optimising the Computational Efficiency (GFLOPS)

In our Matrix-Multiplication example of chapter 2, we noticed that the Matrix Multiplication program was not able to vectorise (Packed was 0%), all computation were sequential (Scalar was 12.5%).

But remember that we have compiled the program without any specific *Optimisation* option. The GCC compiler uses level 1 (-O1) as its default "Optimisation level"!

Now, lets see what happens if we would compile the same program with FULL optimisation (the maximum optimisation level is "-O3") and run our profiler again:

```
$ gcc -g -O3 -o mm mm.c
$ perfexpert -c 0.9 mm
...
* GFLOPS (% max)        31.9 ****************
.- packed               19.4 **********
 - scalar               12.5 ******
...
```

The program is now able to "*vectorise*" much better.

Of course, in the first place we need to design our program in such a way that the compiler is able to vectorise. A situation where we create a read/write data dependency within the loop (e.g., for ... { a[i+1] = a[i] * c[i]; }) shall be avoided, as such program logic may not allow the compiler to "vectorise" this loop.

3.3 Optimising the LCPI Performance Categories

3.3.1 Category #1: Data Access

These kinds of issues occur regularly. The *typical issues* are that a high number of CPU cycles are being spent, whilst waiting for LLC load misses to be serviced.

In general, *some possible optimisations* to improve *data access locality* are:

1. to redesign the data structure;
2. to reduce data working set size;
3. to reorder or interchange the loops, (this works when the loop execution is not important, but one must be aware that Fortran accesses storage by column, C by row);
4. to perform loop fusion (Takes two adjacent loops that have same iteration and combines them into a single loop);
5. to block and consuming data in chunks that fit in the LLC;
6. to explore hardware pre-fetchers. (Frequently, the CPU is not fed fast enough with data from memory, which causes a bottleneck resulting in a CPU waiting for data to process).

We start again from the "test1.c" program, which gave bad results on Data Access. We compile and run it again and we will now focus on the recommendations PerfExpert suggests. We use the –r6 option of PerfExpert, so that we get 6 recommendations.

```
$ more test1.c
...
void compute()
{
register int i, j, k;
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    for (k = 0; k < n; k++)
      d[i][j] += a[i][j] + (b[i][k] * c[k][j]);
}
...
$ gcc -g -o test1 test1.c
$ perfexpert -c -r6 0.9 ./test1
...
Performance assessment  LCPI good.......okay.......fair.......poor.......bad
* overall              2.47 >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
* data accesses        2.94 >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>+
- L1d hits             0.42 >>>>>>>>
- L2d hits             1.03 >>>>>>>>>>>>>>>>>>>>>>
- L3d hits             1.22 >>>>>>>>>>>>>>>>>>>>>>>>
- LLC misses           0.26 >>>>>
...
\#
\# Here is a possible recommendation for this code segment
\#
Description: move loop invariant memory accesses out of loop
Reason: this optimisation reduces the number of executed (slow) memory accesses
Code example:
loop i {
  a[i] = b[i] * c[j]
}
=====>
temp = c[j];
loop i {
 a[i] = b[i] * temp;
}
...
$ gcc -g -o test1 test1.c
$ perfexpert -c -r6 0.9 ./test1
```

There is clearly a problem with our data access LCPI. Several recommendations might not look relevant or applicable, and can be ignored. But the recommendation to move loop invariant memory access out of the loop looks promising.

We have implemented this first improvement in the file "test1_i1.c".

Good Practice: We keep track of the iterations by adding "_i1", "_i2", etc. to the original filename.

```
$ more test1_i1.c
...
void compute()
{
register int i, j, k;
double tmp;
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
  {
    tmp = a[i][j];
    for (k = 0; k < n; k++)
      d[i][j] += tmp + (b[i][k] * c[k][j]);
  }
}
$ gcc -g -o test1_i1 test1_i1.c
$ perfexpert -c 0.5 ./test1_i1
```

Check the result, admire the performance improvement and be invited to optimise further.

3.3.2 Category #2: Instruction Access

The Instruction accesses measures the cost of fetching instructions from memory. This metric will rarely cause problems in scientific programming. Programs with a high(er) values of this metric are typically huge (in bytes) and may have a lot of function calls (mostly) to statically linked libraries.

Potential solutions are:

1. Link with dynamic libraries.
2. Check for dead code / limit your code.
3. Keep often used subroutines together.

3.3.3 Category #3: Data TLB

These kinds of issues also occur regularly and are often related with our problems in category #1. The likely *problem* is that a significant proportion of cycles are being spent handling first-level data TLB misses.

As with ordinary data caching, *potential solutions* are:

1. to focus on improving data locality;
2. to reducing working-set size to reduce DTLB overhead;
3. to collocate frequently-used data on the same page;
4. to try using larger page sizes for large amounts of frequently-used data.

Let's see what recommendations are suggested for our previous test3 example:

```
$ more test3.c
...
void compute()
{
register int i, j, k;
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    for (k = 0; k < n; k++)
      c[i][j] += a[i][k] * b[k][j];
}
...
\#
\# Here is a possible recommendation for this code segment
\#
Description: change the order of loops
Reason: this optimisation may improve the memory access pattern and make it more
    cache and TLB friendly
Code example:
loop i {
  loop j {...}
}
=====>
loop j {
  loop i {...}
}
...
$ gcc -g -o test3 test3.c
$ perfexpert -c -r6 0.5 ./test3
```

Explore the suggested recommendations.

The recommendation to change *the order of the loops* (as shown above) takes our interest. We have prepared a modified source file for you, called "test3_i1.c", where we implemented this suggested recommendation.

Explore the new file, compile and run:

```
$ cat test3_i1.c
...
void compute()
{
register int i, j, k;
for (i = 0; i < n; i++)
  for (k = 0; k < n; k++)
    for (j = 0; j < n; j++)
      c[i][j] = a[i][k] * b[k][j];
}
...
$ gcc -g -o test3_i1 test3_i1.c
$ perfexpert -c 0.6 ./test3_i1
```

And note the improvement, isn't it quite ... spectacular?

3.3.4 Category #4: Instruction TLB

As for Category #2 (Instruction Access), high values for this metric do rarely occur. You can expect higher values in big programs (i.e., programs which the binary size is in the order of hundred of megabytes).

Potential solutions are:

1. Link with dynamic libraries.
2. Check for dead code / limit your code.
3. Keep often used subroutines together.

3.3.5 Category #5: Branches

Branch prediction is a feature of the hardware, and not of the compiler.[1]

In general, some *possible optimisations* are:

1. Avoid or limit branches in your code (A common branch is a check to see whether you're at the end of a loop);
2. To put the most likely branch upfront;
3. Apply code patterns to allow the CPU to make better branch prediction;
4. Use special tags in your code to force the compiler to a certain prediction (e.g., GCC has macros for encoding branch prediction information, where you can define "likely" and "unlikely" paths).

3.3.6 Category #6: Floating-Point Operations

These kinds of issues also occur regularly. The likely *problem* is that a significant proportion of cycles are being spent on expensive FP operations (such as division and square root). Of course, as computing is often the core of the program, we cannot always avoid those issues.

In general, some *possible optimisations* are:

1. to redesign the code and check if the number of "sqrt" and/or "division" calculations can be reduced (by using a pre-calculated value for instance);
2. to explore specific arithmetic compiler options (e.g., -ffast-math). "-ffast-math" allows the compiler to use faster hardware floating point instructions, at the potential expense of IEEE floating point compliance. If your program doesn't need strict compliance, this setting should theoretically net a boost in floating point performance for "*free*".

[1] Although it is possible to insert compiler-specific annotations (aka pragmas) to tell the compiler which is the most occurring branch result.

3. To use *invsqrt*: many modern platforms also offer inverse square root, which has the speed approximately the same as *sqrt*, but is often more useful (e.g., by having *invsqrt* you can compute both *sqrt* and *div* with one multiplication for each).

Optimising floating point performance requires the right combination of good programming techniques and compiler optimisations.

But, as the computations are often unavoidable, you'll sometimes find yourself in a position that you are not able to optimise.

Lets see what recommendations are suggested for our test6 example:

```
$ more test6.c
...
void compute()
{
register int i, j, k;
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++) {
    for (k = 0; k < n; k++) {
      d[i][j] += (a[i][k] * b[k][j]) / c[i][j];
    }
  }
}
...
$ gcc -g -o test6 test6.c
$ perfexpert -c -r12 0.5 ./test6
...
\#
\# Here is a possible recommendation for this code segment
\#
Description: Accumulate and then normalize instead of normalizing each element
Code example:
loop i {
 x = x + a[i] / b;
}
=====>
loop i {
 x = x + a[i];
}
x = x / b;
...
```

Explore the suggested recommendations.

Use the -r option to select the number of recommendations for optimisation you want for each code section, which is a performance bottleneck. By default, PerfExpert provides you with 3 optimisations. The recommendation to change (as shown above) takes our interest. We have prepared a modified source file for you, called "test6_i1.c", where we implemented this suggested recommendation.

Explore the new file, compile and run:

```
$ cat test6_i1.c
...
void compute()
{
register int i, j, k;
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++) {
    for (k = 0; k < n; k++) {
      d[i][j] += a[i][k] * b[k][j];
    }
    d[i][j] /= c[i][j];
  }
}
...
$ gcc -g -o test6_i1 test6_i1.c
$ perfexpert -c 0.3 ./test6_i1
```

And note the improvement.

Chapter 4

Multi-core & Multi-node Profiling

Tip: Find the source code for the examples in this Chapter in the directory: "~/examples/ch04-parallel".

PerfExpert will read hardware performance counter events of the various cores and/or nodes that participated in the parallel computing. PerfExpert will generate a summary report, which is based on the performance of all components.

As such, optimising a parallel application with PerfExpert is done in exactly the same way as a serial application. PerfExpert handles all additional complexity under the hood; the user gets the same kind of report.

We have created 3 parallel equivalents of the Matrix Multiplication program, one in OpenMP, one in MPI and one with a combination of both. It is interesting to see how much we can reduce the runtime of our application by spreading all the work over the different cores. We have adapted the size of the matrices in the examples to [2000 × 2000] as to increase the workload a bit.

First we, go to the directory of the examples of Chapter04:

```
$ cd ~/examples/ch04-parallel
```

4.1 Multi-Core Profiling (with OpenMP)

First we try the serial equivalent program, to have a base for the benchmark.

```
$ gcc -g -o mm mm.c
$ time ./mm
real 1m17.981s
user 1m17.511s
sys 0m0.040s
```

Now, explore the OpenMP program, compile it and run.

Note: Compiling with OpenMP requires the additional OpenMP library to be linked in with

the "-fopenmp" option.

```
$ more mm_omp.c
$ gcc -fopenmp -g -o mm_omp mm_omp.c
$ OMP_NUM_THREADS=8 perfexpert -c 0.2 ./mm_omp
[perfexpert] Collecting measurements [hpctoolkit]
[perfexpert]    [1] 2.558715357 seconds (includes measurement overhead)
[perfexpert]    [2] 2.366958066 seconds (includes measurement overhead)
[perfexpert]    [3] 2.168382589 seconds (includes measurement overhead)
[perfexpert] Analysing measurements
-------------------------------------------------------------------------
Total running time for mm_omp is 9.77 seconds between all 16 cores
The wall-clock time for mm_omp is approximately 0.61 seconds
Module mm_omp takes 99.81% of the total runtime
Module libmonitor.so.0.0.0 takes 0.00% of the total runtime
Module libgomp.so.1.0.0 takes 0.11% of the total runtime
-------------------------------------------------------------------------
...
```

You notice that:

1. The runtime of the executable was sped up by a factor 3 to 4.
2. The "mm_omp" program ran on all 16 cores.
3. The libgomp.so.1.0.0 library used 0.11% of the execution time. This allows us to gather an idea of the "overhead" of OpenMP.

4.2 Multi-Node Profiling (with MPI)

The mm_mpi.c program contains the Matrix Multiplication program developed with MPI. First we compile and run the program. Do not forget to request a private node for this task.

For VSC, first load the MPI module:

```
$ module load impi
```

Institute	Command
TACC	$ **module load impi**
VSC	$ **module load impi**
VUBrussel	$ **module load openmpi/1.6.5/gcc/4.8.2** $ **module load impi**

```
$ mpicc -o mm_mpi mm_mpi.c
$ mpirun mm_mpi
```

The command line to run PerfExpert includes the MPI launcher script (i.e., mpirun). If we would start "PerfExpert 0.1 mpirun mm_mpi", we would analyse the performance of the MPI launcher instead of the performance of your application. This is of course not what we want. You can use the "*-p*" (prefix) command line argument of PerfExpert to define a prefix to the

normal command line. In our case, we want to set the MPI launcher and all its arguments (e.g., -p "mpirun").

The command becomes:

```
$ perfexpert -p "mpirun" -c 0.1 ./mm_mpi
[perfexpert] Collecting measurements [hpctoolkit]
[perfexpert]    [1] 11.375589158 seconds (includes measurement overhead)
[perfexpert]    [2] 11.637017437 seconds (includes measurement overhead)
[perfexpert]    [3] 10.165687700 seconds (includes measurement overhead)
[perfexpert] Analysing measurements
-------------------------------------------------------------------------------
Total running time for mm_mpi is 126.54 seconds between all 16 cores
The wall-clock time for mm_mpi is approximately 7.91 seconds
...
```

No difference, whatsoever, in the report that is displayed to the user.

4.3 Profiling with OpenMP and MPI

The "*mm_mpi_omp.c*" program contains the Matrix Multiplication program developed with OpenMP and MPI. The trick with OpenMP and MPI is not any different.

First we compile and run the program.

```
$ mpicc -fopenmp -o mm_mpi_omp mm_mpi_omp.c
The command becomes:
$ perfexpert -p "mpirun" 0.1 ./mm_mpi_omp
[perfexpert] Collecting measurements [hpctoolkit]
[perfexpert]    [1] 11.759415785 seconds (includes measurement overhead)
[perfexpert]    [2] 14.620955584 seconds (includes measurement overhead)
[perfexpert]    [3] 11.391528953 seconds (includes measurement overhead)
[perfexpert] Analysing measurements
...
```

Chapter 5

More PerfExpert Functionality

5.1 Special Options

5.1.1 Arguments

If your program takes any argument that starts with a "-" signal, PerfExpert will interpret this as a command line option of its own. To help PerfExpert handle arguments and options correctly, use quotes and add a space before the program's arguments (e.g., "-s 50").

If your program would normally executed by the command:

```
$ my_prog -n 1000 -s 50
```

the PerfExpert command line would become:

```
$ perfexpert 0.1 ./my_prog "-n 1000 -s 50"
```

5.1.2 Input/Output pipes

When your program reads from standard input via an input pipe (i.e., "<"), use the "-i" option. The Output pipe ('>') is not available because PerfExpert already sets a default output file.

If your program would normally executed by the command:

```
$ my_prog < input_file
```

the PerfExpert command line would become:

```
$ perfexpert -i input_file 0.1 ./my_prog
```

5.1.3 Prologue and Epilogue

PerfExpert will run your application multiple times to collect different performance metrics. If you want to execute a program or script **before** (or **after**) each run, you may use the "**-b**" (or "**-a**") options. This is sometimes needed to reset the proper start situation for your program, to temporarily change environment variables, or to clean up after the run.

The target program should be the filename of the application you want to analyse, not a shell script. Otherwise, PerfExpert will analyse the performance of the shell script instead of the performance of you application.

If your program requires some scripts to run and after the real program, such as:

```
$ before_script
$ my_prog
$ after_script
```

the PerfExpert command line would become:

```
$ perfexpert -b before_script -a after_script 0.1 ./my_prog
```

5.1.4 Debugging

In the event a bug (*or call it a special feature*) might occur, the user can do some first aid troubleshooting.

Verbose:

You could run perfexpert in verbose mode with the –v option. The verbose level runs from 0 to 10, and it will list the different programs that PerfExpert will start and also a lot of internal operations.

Garbage Directory:

PerfExpert creates a hidden directory for each run, wherein it stores all its internal information. The directory name has a unique format such as "*.perfexpert-temp.kPUtGV*". This directory structure contains a database, the results of the measurements of the various runs, code fragments identified as bottleneck, the log files, etc.

The full directory structure is removed after a proper run. In case you deliberate want to keep this directory structure, the user can add the "-g" (keep garbage) option.

5.2 PerfExpert Automatic-mode

This component of the software is still under construction!

PerfExpert can also automatically optimise your code. The user will still get the performance analysis report and the list of suggestion for bottleneck remediation when no automatic optimisation is possible.

In order to allow PerfExpert to automatically optimise, PerfExpert must be told where to find the single source file or where to find the Makefile to compile multiple files.

The automatic optimisation is achieved by using one the 2 extra options:

1. -s <source-file>

2. -m <Makefile>

If you select the -m option and the application is composed of multiple files, your source code tree should have a Makefile file to enable PerfExpert compile your code. If your application is composed of a single source code file, the option -s is sufficient for you. If you do not select -m or -s options, PerfExpert requires only the binary code and will show you only the performance analysis report and the list of suggestion for bottleneck remediation.

Caution: PerfExpert may, if you choose to use the full capabilities for automated optimisation, change your source code during the process of optimisation. PerfExpert always saves the original file with a different name (e.g., mm_omp.c.old_27301) as well as adds annotations to your source code for each optimisation it makes. We cannot, however, fully guarantee that code modifications for optimisations will not break your code. We recommend having a full backup of your original source code before using PerfExpert.

5.2.1 Automatic Optimisation of single source file

PerfExpert will automatically try a number of code transformations, and it will recompile it for every attempt according to a default compilation command:

```
gcc -O3 -g -o <binary-file> <source-file>
```

Now, try it on a sequential source:

```
$ perfexpert -s mm.c -r50 0.1 ./mm
```

The new code will have been compiled with the command:

```
gcc -O3 -g -o mm mm.c
```

You can use CC, CFLAGS and LDFLAGS to select different compiler and compilation/linkage flags. The command line would become:

```
$ CC="icc" CFLAGS="-g -O2" perfexpert -s mm.c -r50 0.05 ./mm
```

When using PerfExpert's automatic optimisations on the OpenMP version of the simple matrix multiply code, and if we want that the OpenMP-enabled binary will run with 16 threads, we will use the following command line options:

```
$ OMP_NUM_THREADS=16 CFLAGS="-fopenmp" perfexpert -s mm_omp.c -r50 0.05 ./mm_omp
```

In this case, PerfExpert will compile the mm_omp.c code using the system's default compiler, which is GCC in the case of Stampede. PerfExpert will take into consideration only code fragments (loops and functions) that take more 5% of the runtime.

To select a different compiler, you should specify the CC environment variable as below:

```
$ CC="icc" OMP_NUM_THREADS=16 CFLAGS="-fopenmp" perfexpert -s mm_omp.c -r50 0.05
    ./mm_omp
```

5.2.2 Automatic Optimisation of multiple source files

The same approach can be followed with multiple source files. In his case, the user need to create a Makefile, which can be used to recompile the executable.

Explore the contents of the Makefile and list the files in the directory:

```
$ more Makefile
$ ls
compute1.c  compute1.h  compute2.c  compute2.h  compute3.c  compute3.h  main.c  main
    .h  Makefile
```

And run in automatic mode:

```
$ perfexpert -m Makefile -r50 -c 0.1 ./main
```

5.3 Running PerfExpert in Batch mode (with Job Script)

One can also run perfexpert in batch mode. This is often desired when the runtime of the program is a bit higher, and you do not want to wait for the results. You can replace the normal command line to start up your executable (e.g., "./mm_omp") in the job script with the perfexpert command line. (e.g., "*perfexpert -c 0.3 mm_ omp*"). When running your application with a job script, be sure to specify a running time that is about 6x the normal running time of the program.

Compile the parallel Matrix Multiplication program, explore the job-script and submit the job:

```
$ gcc -fopenmp mm_omp.c -g -o mm_omp
$ more mm_omp.job
```

Institute	**Command**
TACC	$ **sbatch mm_omp.job**
VSC	$ **qsub mm_omp.job**

The job will now be submitted to a node on the supercomputer. PerfExpert will run on the allocated node, start the users application and generate the performance analysis report. The results are printed on standard output (stdout), which was re-directed to the file "MyJob.o%j" ("%j" is replaced by the jobID).

When the job is finished, the generated report can be viewed in the file in our current directory.

```
$ ls
...
MyJob.o2225027
MyJob.e2225027
...
$ more MyJob.o2225027
[perfexpert] Collecting measurements [hpctoolkit]
[perfexpert]    [1] 1.992842066 seconds (includes measurement overhead)
[perfexpert]    [2] 1.564767383 seconds (includes measurement overhead)
[perfexpert]    [3] 1.373591388 seconds (includes measurement overhead)
[perfexpert] Analysing measurements
-------------------------------------------------------------------------
Total running time for mm_omp is 8.86 seconds between all 16 cores
The wall-clock time for mm_omp is approximately 0.55 seconds
...
```

Appendix A

Annex 1: PerfExpert Options

There are several different options for applying PerfExpert. The following summary shows you how to choose the options to run PerfExpert to match your needs.

```
$ perfexpert -h
Usage: perfexpert [OPTION...] THRESHOLD PROGRAM [PROGRAM ARGUMENTS]
PerfExpert -- an easy-to-use automatic performance diagnosis and optimization tool
    for HPC applications
 THRESHOLD           Threshold (relevance % of runtime) to take hotspots into
                     consideration (range: between 0 and 1, accepts fraction)
 PROGRAM             Program (binary) to analyze (do not use shell scripts)
 PROGRAM ARGUMENTS   Program arguments, see documentation if any argument
                     starts with a dash sign ('-')
Automatic optimization options:
 -m, --target=TARGET        Use GNU standard 'make' command to compile the
                            code (it will run in the current directory)
 -s, --source=FILE          Set the source code file (does not work with
                            multiple files, choose -m option instead)
Use CC, CFLAGS, CPPFLAGS and LDFLAGS to set compiler/linker options
Target program execution options:
 -a, --after=COMMAND        Command to execute after each run of the target
                            program
 -b, --before=COMMAND       Command to execute before each run of the target
                            program
 -i, --inputfile=FILE       Input file to the target program. Use this option
                            instead of input pipe redirect ('<'). Output pipe
                            ('>') is not available because PerfExpert already
                            set a default output file
 -p, --prefix=PREFIX        Add a prefix to the command line, use double
                            quotes to set arguments with spaces within (e.g.
                            -p "mpirun -n 2")
Output formatting options:
 -c, --colorful             Enable ANSI colors
 -o, --order=relevance/performance/mixed
                            Order in which hotspots should be sorted (default:
                            unsorted)
 -r, --recommendations=COUNT   Number of recommendations PerfExpert should
                            provide (default: 3)
 -v, -l, --verbose=LEVEL, --verbose-level=LEVEL
                            Enable verbose mode (default: 5, range: 0-10)
Other options:
 -d, --database=FILE        Select a recommendation database file different
                            from the default
 -e, --only-experiments     Tell PerfExpert to not perform any analysis just
                            run the target program (for further manual
                            analysis)
 -g, --leave-garbage        Do not remove temporary directory when finalize
 -n, --do-not-run           Do not run PerfExpert, just parse the command line
                            (for debugging)
 -t, --measurement-tool=hpctoolkit/vtune
                            Set the tool that should be used to collect
                            performance counter values (default: hpctoolkit)
Informational options:
 -?, --help                 Give this help list
     --usage                Give a short usage message
 -V, --version              Print program version
Mandatory or optional arguments to long options are also mandatory or optional
for any corresponding short options.
Report bugs to PerfExpert mailing list: perfexpert@lists.tacc.utexas.edu.
```

Appendix B

Annex 2: Optimisation Patterns

B.1 Memory Access Optimisations

R001	**Direct array access**	
Category	Bad memory access Bad L2 data access	
Description	Access arrays directly instead of using local copies	
Code Sample	**Old**	**New**

	loop j {
	a[j] = b[i][j][k];
	}
	...
	loop j {
	... a[j] ...
	}
		loop j {
		... b[i][j][k] ...
		}

R002	**Group allocations**	
Category	Bad memory access	
Description	Allocate an array of elements instead of each element individually	
Code Sample	**Old**	**New**

	loop {
	... c = malloc(1); ...
	}
		top = n;
		loop {
		if (top == n) {
		tmp = malloc(n);
		top = 0;
		}
		...
		c = &tmp[top++]; ...
		}

R003	**Loop fission**	
Category	Bad memory access	
Description	Apply loop fission so every loop accesses just a couple of different arrays	
Code Sample	**Old**	**New**

	loop i {
	a[i] = a[i] * b[i] - c[i];
	}
		loop i {
		a[i] = a[i] * b[i];
		}
		loop i {
		a[i] = a[i] - c[i];
		}

R004	**Avoid unneeded array updates**	
Category	Bad memory access Bad L2 data access	
Description	Avoid unnecessary array updates	
Code Sample	**Old**	**New**

	loop i {
	a[i] = . . . ;
	. . . a[i] . . .
	}
	// array a[] not read
		loop i {
		temp = . . . ;
		. . . temp . . .
		}

R005	**Change the order of loops**	
Category	Bad memory access Bad data TLB access	
Description	Change the order of loops	
Code Sample	**Old**	**New**

	loop i {
	loop j {. . . }
	}
		loop j {
		loop i {. . . }
		}

R006	**Componentize loops**	
Category	Bad memory access Bad L1 data access Bad L2 data access	
Description	Componentize important loops by factoring them into their own subroutines	
Code Sample	**Old**	**New**

	loop i {. . . }
	. . .
	loop j {. . . }
		void li() {loop i {. . . }}
		void lj() {loop j {. . . }}
		. . .
		li();
		. . .
		lj();

R007	**Loop blocking and interchange**	
Category	Bad memory access	
Description	Employ loop blocking and interchange	
Code Sample	**Old**	**New**

	loop i {
	loop k {
	loop j {
	c[i][j] = c[i][j] + a[i][k] * b[k][j];
	}
	}
	}
		loop k step s {
		loop j step s {
		loop i {
		for (kk = k; kk < k + s; kk++)
		{
		for (jj = j; jj < j + s; jj++) {
		c[i][jj] = c[i][jj] + a[i][kk] * b[kk][jj];
		}
		}
		}
		}
		}

R0.9	**Fuse loops**	
Category	Bad memory access Bad L2 data access	
Description	Fuse multiple loops that access the same data	
Code Sample	**Old**	**New**

	loop i {
	a[i] = x[i];
	}
	loop i {
	b[i] = x[i] - 1;
	}
		loop i {
		a[i] = x[i];
		b[i] = x[i] - 1;
		}

R009		
Category	Bad memory access	
Description	Pad memory areas so that temporal elements do not map to same set in cache	
Code Sample	**Old**	**New**

	double a[const * cache_size/8], b[const * cache_size/8];
	loop i {
	... a[i] + b[i] ...
	}
		double a[const * cache_size/8 + 8], b[const * cache_size/8 + 8];
		loop i {
		... a[i] + b[i] ...
		}

R010		
Category	Bad memory access Bad L2 data access	
Description	Reuse temporary arrays for different operations	
Code Sample	**Old**	**New**

	loop i {
	t1[i] = . . . ;
	. . . t1[i] . . . ;
	}
	. . .
	loop j {
	t2[j] = . . . ;
	. . . t2[j] . . . ;
	}
		loop i {
		t[i] = . . . ;
		. . . t[i] . . . ;
		}
		. . .
		loop j {
		t[j] = . . . ;
		. . . t[j] . . . ;
		}

R011		
Category	Bad memory access	
Description	Split structures into hot and cold parts, where hot part has pointer to cold part	
Code Sample	**Old**	**New**

	struct s {
	hot_field;
	many_cold_fields;
	} a[n];
		struct s_hot {
		hot_field;
		struct s_cold *ptr;
		} a_hot[n];
		struct s_cold {
		many_cold_fields;
		} a_cold[n];

R012	**Smaller types**	
Category	Bad memory access Bad data TLB access	
Description	Use smaller types (e.g., float instead of double or short instead of int)	
Code Sample	**Old**	**New**
	double a[n];	float a[n];

R013	**Align data structures**	
Category	Bad L1 data access	
Description	Align data, especially arrays and structs	
Code Sample	**Old**	**New**
	int x[1024];	__declspec(align(16)) int x[1024];

R014	**Use local scalar data**	
Category	Bad L1 data access	
Description	Copy data into local scalar variables and operate on the local copies	
Code Sample	**Old**	**New**

	x = a[i] * a[i];
	...
	a[i] = x / b;
	...
	b = a[i] + 1.0;
		t = a[i];
		x = t * t;
		...
		a[i] = t = x / b;
		...
		b = t + 1.0;

R015	**Eliminate sub-expressions**	
Category	Bad L1 data access	
Description	Eliminate common subexpressions involving memory accesses	
Code Sample	**Old**	**New**

	d[i] = a * b[i] + c[i];
	y[i] = a * b[i] + x[i];
		temp = a * b[i];
		d[i] = temp + c[i];
		y[i] = temp + x[i];

R016	**Allow vectorisation**	
Category	Bad L1 data access	
Description	Enable the use of vector instructions to transfer more data per access	
Code Sample	**Old**	**New**

	align arrays, use only stride-one accesses, make loop count even (pad arrays)
	struct {
	double a, b;
	} s[127];
	for (i = 0; i < 127; i++) {
	s[i].a = 0;
	s[i].b = 0;
	}
		__declspec(align(16)) double a[128], b[128];
		for (i = 0; i < 128; i++) {
		a[i] = 0;
		b[i] = 0;
		}

R017	**Restrict**	
Category	Bad L1 data access	
Description	Help the compiler by marking pointers to non-overlapping data with “restrict”	
Code Sample	**Old**	**New**
	void *a, *b;	void * restrict a, * restrict b;

R018	**Invariant memory**	
Category	Bad L1 data access	
Description	Move loop invariant memory accesses out of loop	
Code Sample	**Old**	**New**

	loop i {
	a[i] = b[i] * c[j]
	}
		temp = c[j];
		loop i {
		a[i] = b[i] * temp;
		}

R019	**Unroll outer loops**	
Category	Bad L1 data access	
Description	Unroll outer loop	
Code Sample	**Old**	**New**

	loop i {
	loop j {
	a[i][j] = b[i][j] * c[j];
	}
	}
		loop i step 4 {
		loop j {
		a[i][j] = b[i][j] * c[j];
		a[i+1][j] = b[i+1][j] * c[j];
		a[i+2][j] = b[i+2][j] * c[j];
		a[i+3][j] = b[i+3][j] * c[j];
		}
		}

R020	**Double to float**	
Category	Bad L1 data access Bad float point instructions (slow FP)	
Description	Use float instead of double data type if loss of precision is acceptable.	
Code Sample	**Old**	**New**
	double a[n];	float a[n];

R021	**Compute iso loading**	
Category	Bad L2 data access Bad memory access	
Description	Compute values rather than loading them if doable with few operations	
Code Sample	**Old**	**New**

	loop i {
	t[i] = a[i] * 0.5;
	}
	loop i {
	a[i] = c[i] - t[i];
	}
		loop i {
		a[i] = c[i] - (a[i] * 0.5);
		}

B.2 Instruction Access optimisations

R100	**Avoid code duplication**	
Category	Bad instruction access	
Description	Factor out sequences of common code into subroutines	
Code Sample	**Old**	**New**
	same_code; same_code;	void f() {same_code;} f(); f();

R101	**Lower loop unroll factor**	
Category	Bad instruction access	
Description	Lower the loop unroll factor	
Code Sample	**Old**	**New**

	loop i step 4 {
	code_i;
	code_i+1;
	code_i+2;
	code_i+3;
	}
		loop i step 2 {
		code_i;
		code_i+1;
		}

R102	**Make subroutines more general**	
Category	Bad instruction access	
Description	Make subroutines more general and use them more	
Code Sample	**Old**	**New**

	void f() {
	statements1;
	statementsX;
	}
	void g() {
	statements2;
	statementsX;
	}
		void fg(int flag) {
		if (flag) {
		statements1;
		} else {
		statements2;
		}
		statementsX;
		}

R103	**Sort according call chains**	
Category	Bad instruction access Bad instruction TLB access	
Description	Sort subroutines by call chains (subroutine coloring)	
Code Sample	**Old**	**New**

	f() {...}
	g() {...}
	h() {...}
	loop {
	f();
	h();
	}
		g() {...}
		f() {...}
		h() {...}
		loop {
		f();
		h();
		}

R104	**Split off cold code**	
Category	Bad instruction access	
Description	Split off cold code into separate subroutines and place them at end of file	
Code Sample	**Old**	**New**

	if (unlikely_condition) {
	lots_of_code
	}
		void f() {lots_of_code}
		...
		if (unlikely_condition)
		f();

R105	**Trace scheduling**	
Category	Bad instruction access	
Description	Use trace scheduling to reduce the branch taken frequency	
Code Sample	**Old**	**New**

	if (likely_condition)
	f();
	else
	g();
	h();
		if (!likely_condition) {
		g(); h();
		} else {
		f(); h();
		}

R2.0	**Change order of subroutines**	
Category	Bad instruction TLB access	
Description	Change the order of subroutine calls	
Code Sample	**Old**	**New**
	f(); h();	h(); f();

R107	**Inlining**	
Category	Bad instruction TLB access	
Description	Use inlining	
Code Sample	**Old**	**New**

	float f(float x) {
	return x * x;
	}
	z = f(y);
		z = y * y;

B.3 Branch optimisations

R200	**Booleans to integer**	
Category	Bad branch instructions	
Description	Express Boolean logic in form of integer computation	
Code Sample	**Old**	**New**
	if ((a==0) && (b==0) && (c==0)) {...}	if ((a |b |c) == 0) {...}

R201	**Move loop into subroutine**	
Category	Bad branch instructions	
Description	Move a loop around a subroutine call into the subroutine	
Code Sample	**Old**	**New**

	f(x) {. . . x. . . };
	loop i {
	f(a[i]);
	}
		f(x[]) {
		loop j {
		. . . x[j]. . .
		}
		};
		f(a);

R202	**Move loop invariant tests out of loop**	
Category	Bad branch instructions	
Description	Move loop invariant tests out of loop	
Code Sample	**Old**	**New**

	loop i {
	if (x < y)
	a[i] = x * b[i];
	else
	a[i] = y * b[i];
	}
		if (x < y) {
		loop i {
		a[i] = x * b[i];
		}
		} else {
		loop i {
		a[i] = y * b[i];
		}
		}

R203	**Remove IF**	
Category	Bad branch instructions	
Description	Remove IF	
Code Sample	**Old**	**New**

	/* x is 0 or -1 */
	if (x == 0)
	a = b;
	else
	a = c;
		a = (b & x) |(c & x);

R204	**Most often used loops**	
Category	Bad branch instructions	
Description	Special-case the most often used loop count(s)	
Code Sample	**Old**	**New**

	for (i = 0; i < n; i++)
	{. . . }
		/* Suppose n=4 occurs most of the time*/
		if (n == 4) {
		for (i = 0; i < 4; i++)
		{. . . }
		} else {
		for (i = 0; i < n; i++)
		{. . . }
		}

R205	**Unroll loops**	
Category	Bad branch instructions	
Description	Unroll loops (more)	
Code Sample	**Old**	**New**

	loop i {
	a[i] = a[i] * b[i];
	}
		loop i step 2 {
		a[i] = a[i] * b[i];
		a[i+1] = a[i+1] * b[i+1];
		}

R206	**Conditional moves**	
Category	Bad branch instructions	
Description	Use trivial assignments inside THEN/ELSE to allow the use of conditional moves	
Code Sample	**Old**	**New**

	if (x < y)
	a = x + y;
		temp = x + y;
		if (x < y)
		a = temp;

B.4 Floating Point Optimisations

R300	**Associativity**	
Category	Bad float point instructions (fast FP) Bad float point instructions (slow FP)	
Description	Eliminate floating-point operations through associativity	
Code Sample	**Old**	**New**

	d[i] = a[i] * b[i] * c[i];
	y[i] = x[i] * a[i] * b[i];
		temp = a[i] * b[i];
		d[i] = temp * c[i];
		y[i] = x[i] * temp;

R301	**Distributivity**	
Category	Bad float point instructions (fast FP) Bad float point instructions (slow FP)	
Description	Eliminate floating-point operations through distributivity	
Code Sample	**Old**	**New**
	d[i] = a[i] * b[i] + a[i] * c[i];	d[i] = a[i] * (b[i] + c[i]);

R302	**Move loop invariant computations out**	
Category	Bad float point instructions (fast FP) Bad float point instructions (slow FP)	
Description	Move loop invariant computations out of loop	
Code Sample	**Old**	**New**

	loop i {
	x = x + a * b * c[i];
	}
		temp = a * b;
		loop i {
		x = x + temp * c[i];
		}

R303	**Normalize after accumulations**	
Category	Bad float point instructions (slow FP)	
Description	Accumulate and then normalize instead of normalizing each element	
Code Sample	**Old**	**New**

	loop i {
	x = x + a[i] / b;
	}
		loop i {
		x = x + a[i];
		}
		x = x / b;

R304	**Rework SQRT in conditions**	
Category	Bad float point instructions (slow FP)	
Description	Compare squared values instead of computing the square root	
Code Sample	**Old**	**New**
	if (x < sqrt(y)) {...}	if ((x < 0.0) ||(x*x < y)) {...}

R305	**Work with reciprocal**	
Category	Bad float point instructions (slow FP)	
Description	Compute the reciprocal outside of loop and use multiplication inside the loop	
Code Sample	**Old**	**New**

	loop i {
	a[i] = b[i] / c;
	}
		cinv = 1.0 / c;
		loop i {
		a[i] = b[i] * cinv;
		}