Analyzing data Users analyze the data recorded by the monitoring system to localize optimization potential. Performance data is either recorded during program execution and assessed after the application finished, this approach of *post-mortem* analysis is also referred to as *offline* analysis. An advantage of this methodology is that data can be analyzed multiple times and compared with older results. Another approach is to gather and assess data *online* – while the program runs. This way feedback is provided immediately to the user, who could adjust settings to the monitoring environment depending on the results.

Due to the vast amount of data, sophisticated tools are required to localize performance issues of the system and correlate them with application behavior and finally identify source code causing them. Tools operate either manually, i.e., the user must inspect the data himself; a *semi-automatic tool* could give hints to the user where abnormalities or inefficiencies are found, or try to assess data automatically. Tool environments, which localize and tune code automatically, without user interaction, are on the wishlist of all programmers. However, due to the system and application complexity those are only applicable for a very small set of problems. Usually, tools offer analysis capability in several *views* or *displays*, each relevant to a particular type of analysis.

2.4.3. Available Tools for Analysis of Sequential Programs

There exist plenty of tools that assist in performance analysis and optimizations of sequential code, a handful of tools of different categories are briefly introduced: *GNU gprof* generates a profile of the application in user-space. *OProfile* can record and investigate application and system behavior including activity of the Linux kernel. CPU counters can be related to the individual operations. *PAPI* is a library which accesses CPU counters and provides additional hardware statistics. *Likwid* is a lightweight tool suite that reads CPU counters for an application. *LTTng* traces and visualizes activity of processes and within the kernel. However, compared to OProfile symbolic information of the application program is not supported.

All tools mentioned are licensed under an open and free license. The state of the latest stable versions available is discussed as of February 2011.

GNU gprof

GNU compilers can be instructed to include code into a program that will periodically collect samples of the program counter. During run-time profiles of function call timings and the *call graph*[40] are stored in a file. This profiling data can then be analyzed by the command line tool `gprof`.

To demonstrate the application of the tool, an excerpt of the `gprof` output for a run of the `partdiff-seq` PDE solver is given in Listing 2.1. In the *flat profile* (up to Line 12), the time is shown per function. While 6 functions have been invoked once (Lines 6 to 11), all run-time is spent in the function `calculate()` (Line 6). The textual representation of the call graph is also provided starting with Line 15 of the output. In Line 20 and Line 21, it is shown that the function `calculate()` is called from main once, additional invocations from different functions would generate further sections. The main function calls all 6 subroutines (Lines 24-30).

Most platforms provide tools alike to `gprof` to analyze performance of sequential programs. To analyze a parallel application, a profiler must be aware of the parallelism and provide an approach to handle it, for example, by generating one output for each of the spawned processes.

Listing 2.1: Excerpt of a `gprof` output for partdiff-seq

```
1  Flat profile:
2
3   Each sample counts as 0.01 seconds.
4    %   cumulative   self              self     total
5   time   seconds   seconds    calls   s/call   s/call  name
6  100.05     30.95    30.95        1    30.95    30.95  calculate
7    0.00     30.95     0.00        1     0.00     0.00  AskParams
```

[40] The call graph is a directed graph providing information about function invocation, the nodes of the graph represent functions and the edges function calls.

```
  0.00     30.95     0.00        1     0.00     0.00  DisplayMatrix
  0.00     30.95     0.00        1     0.00     0.00  allocateMatrices
  0.00     30.95     0.00        1     0.00     0.00  displayStatistics
  0.00     30.95     0.00        1     0.00     0.00  initMatrices

[...]

                     Call graph (explanation follows)

granularity: each sample hit covers 2 byte(s) for 0.03% of 30.95 seconds

index % time    self  children    called     name
                30.95    0.00       1/1           main [2]
[1]    100.0   30.95    0.00       1         calculate [1]
-----------------------------------------------
                                                  <spontaneous>
[2]    100.0    0.00   30.95                 main [2]
                30.95    0.00       1/1           calculate [1]
                 0.00    0.00       1/1           AskParams [3]
                 0.00    0.00       1/1           initMatrices [7]
                 0.00    0.00       1/1           allocateMatrices [5]
                 0.00    0.00       1/1           displayStatistics [6]
                 0.00    0.00       1/1           DisplayMatrix [4]
```

OProfile

OProfile[41] provides a sophisticated system-level profiling for the Linux operating system. Compared to `gprof`, OProfile gathers information from all running processes at the same time. Also, kernel internals are captured, and a configurable set of hardware performance counters. Profiling must be enabled and started by the super user, then all activities are recorded. Several tools are provided that analyze data recorded from user-space.

An example for system-level profiling of a desktop system running our PDE is given in listing 2.2. In this profile, the concurrent activities can be identified, also the time spent in kernel space[42] and in certain libraries becomes apparent. For each application, the user can create an individual profile, covering activities of the particular application as well as activities triggered by library and system calls.

A very handy feature of the OProfile system is that source code (and additionally assembler) can be annotated with the performance observations. This makes it easier to localize the time-consuming lines. In Listing 2.3, the source code of `calculate()` is shown with the sample count. The time spent in individual code lines becomes apparent, for example, 12% of the run-time is spent in Line 20, showing the optimization potential within this line.

Accessible hardware counters on the desktop system are shown in Listing 2.4[43].

Listing 2.2: Excerpt of a system-wide OProfile output while running partdiff-seq

```

CPU: Intel Architectural Perfmon, speed 1199 MHz (estimated)
Counted CPU_CLK_UNHALTED events (Clock cycles when not halted) with a unit mask of 0x00 (No unit mask) count 100000
samples  %        image name               app name                 symbol name
1068951  71.0457  partdiff-seq             partdiff-seq             calculate
185685   12.3412  no-vmlinux               no-vmlinux               /no-vmlinux
26927     1.7896  libgstflump3dec.so       libgstflump3dec.so       /usr/lib/gstreamer-0.10/libgstflump3dec.so
16738     1.1125  libspeexdsp.so.1.5.0     libspeexdsp.so.1.5.0     /usr/lib/libspeexdsp.so.1.5.0
13980     0.9292  Xorg                     Xorg                     /usr/bin/Xorg
12617     0.8386  libQtGui.so.4.7.0        libQtGui.so.4.7.0        /usr/lib/libQtGui.so.4.7.0
12110     0.8049  libQtCore.so.4.7.0       libQtCore.so.4.7.0       /usr/lib/libQtCore.so.4.7.0
10177     0.6764  libxul.so                libxul.so                /usr/lib/firefox-3.6.13/libxul.so
8375      0.5566  libmozjs.so              libmozjs.so              /usr/lib/firefox-3.6.13/libmozjs.so
8311      0.5524  libflashplayer.so        libflashplayer.so        /usr/lib/flashplugin-installer/libflashplayer.so
6670      0.4433  libdrm_intel.so.1.0.0    libdrm_intel.so.1.0.0    /lib/libdrm_intel.so.1.0.0
6097      0.4052  libpulsecommon-0.9.21.so libpulsecommon-0.9.21.so /usr/lib/libpulsecommon-0.9.21.so
4185      0.2781  oprofiled                oprofiled                /usr/bin/oprofiled
4111      0.2732  libpthread-2.12.1.so     libpthread-2.12.1.so     pthread_mutex_lock
3819      0.2538  libgstreamer-0.10.so.0.26.0 libgstreamer-0.10.so.0.26.0 /usr/lib/libgstreamer-0.10.so.0.26.0
3255      0.2163  libpthread-2.12.1.so     libpthread-2.12.1.so     pthread_mutex_unlock
```

Listing 2.3: Excerpt of the annotated partdiff-seq source code

```
             :/* ********************************************************************** */
```

[41] Visit `http://oprofile.sourceforge.net/` for further information.

[42] By providing the kernel symbol table, all activity inside the kernel is accounted in a fine-grained manner to the kernel-internal symbols; in the listing the `no-vmlinux` symbol aggregates all kernel activities.

[43] The output was created by using `opcontrol -list-events`.

```
2                :/* calculate: solves the equation                                    */
3                :/* ******************************************************************* */
4                :void calculate(void)
5                :{ /* calculate total: 1068951 99.9979 */
6                :  int i,j;                                    /* local variables for loops  */
7                :
8                :  while(term_iteration>0)
9                :  {
10      2 1.9e-04 :    maxresiduum=0;
11   5088  0.4760 :    for(i=1;i<N;i++)                                  /* over all rows     */
12               :    {                                                 /*                   */
13  74459  6.9655 :      for(j=1;j<N;j++)                                /* over all columns  */
14               :      {
15 484776 45.3497 :        star=                    -Matrix[m2][i-1][j]
16               :           -Matrix[m2][i][j-1]   -Matrix[m2][i][j+1]
17               :                                 -Matrix[m2][i+1][j]  +4.0*Matrix[m2][i][j];
18               :        residuum=getResiduum(i,j);
19     315  0.0295 :        korrektur=residuum;
20  134782 12.6086 :        residuum = (residuum<0) ? -residuum:  residuum; // if (residuum<0) residuum=residuum*(-1);
21  135906 12.7137 :        maxresiduum = (residuum < maxresiduum) ? maxresiduum : residuum;
22               :
23   66943  6.2624 :        Matrix[m1][i][j]=Matrix[m2][i][j]+korrektur;
24               :      }
25               :    }
26               :    stat_iteration=stat_iteration+1;
27      56  0.0052 :    stat_precision=maxresiduum;
28               :    checkQuit();
29               :  }
30               :}
31               :
```

Listing 2.4: Available OProfile events (accessible hardware counters) on an Intel Nehalem system

```
1
2  OProfile: available events for CPU type "Intel␣Architectural␣Perfmon"
3
4  See Intel 64 and IA-32 Architectures Software Developer's Manual
5  Volume 3B (Document 253669) Chapter 18 for architectural perfmon events
6  This is a limited set of fallback events because oprofile doesn't know your CPU
7  CPU_CLK_UNHALTED: (counter: all)
8          Clock cycles when not halted (min count: 6000)
9  INST_RETIRED: (counter: all)
10         number of instructions retired (min count: 6000)
11 LLC_MISSES: (counter: all)
12         Last level cache demand requests from this core that missed the LLC (min count: 6000)
13         Unit masks (default 0x41)
14         ----------
15         0x41: No unit mask
16 LLC_REFS: (counter: all)
17         Last level cache demand requests from this core (min count: 6000)
18         Unit masks (default 0x4f)
19         ----------
20         0x4f: No unit mask
21 BR_INST_RETIRED: (counter: all)
22         number of branch instructions retired (min count: 500)
23 BR_MISS_PRED_RETIRED: (counter: all)
24         number of mispredicted branches retired (precise) (min count: 500)
```

PAPI and Likwid

There are several approaches that access CPU counters. *PAPI*, the *Performance API* [MCW+05, TJYD09], is a portable library which enables programs to gather performance events of all modern x86 and Power processors.

PAPI evolved from an interface for CPU counters to the component-oriented PAPI-C [TJYD10], which extends the original PAPI to capture counters from a multitude of sources – ACPI, `lm_sensors` for temperature, or specific network interfaces (Myrinet). PAPI-C leverages the existing inhomogeneous vendor (and software) interfaces.

An alternative library and user space program to profile hardware counters for an application is Likwid [THW10]. Likwid is a user space tool that profiles hardware counters for the whole application execution. When the tool is started, it initializes the counters, then starts the application, and once the application terminates, the counters are stopped and a brief report is created. A small API is offered that allows a developer to restrict the measured code regions and, furthermore, it supports to split execution into phases that can be assessed individually.

An exemplary profile of the floating point group of `partdiff-seq` is given in Listing 2.5. In this example the Intel processor operated on an average clock of 3.427 GHz (Line 28), the number of *cycles per*

instruction(CPI) is 0.52, which means every other cycle one instruction is *retired*[44] on the core. The number of double precision floating point operations per second is about 1.7 GFlop/s. One can easily conduct from the average clock speed and CPI that if we execute one instruction every other cycle, then effectively 1.7 Gigainstructions are executed per second, which means most operations were floating point instructions. This little example already demonstrates the power of hardware counters and simple theoretic considerations.

Available groups for Likwid are shown in Listing 2.6. The group to measure is selected upon startup of Likwid. Note that the memory group is aggregated for all cores on a given chip.

Listing 2.5: Excerpt of the likwid output for partdiff-seq

```
1
2  -------------------------------------------------------------
3  -------------------------------------------------------------
4  CPU type:       Intel Core Westmere processor
5  CPU clock:      2.79 GHz
6  Measuring group FLOPS_DP
7  -------------------------------------------------------------
8  /opt/likwid/bin/likwid-pin -c 1 ./partdiff-seq 0 2 100 1 2 10000
9  [likwid-pin] Main PID -> core 1 - OK
10
11 [...]
12 < program output >
13 [...]
14
15 +--------------------------------------+-------------+
16 |                Event                 |   core 1    |
17 +--------------------------------------+-------------+
18 |           INSTR_RETIRED_ANY          | 2.02167e+11 |
19 |         CPU_CLK_UNHALTED_CORE        | 1.04973e+11 |
20 |         CPU_CLK_UNHALTED_REF         | 8.55369e+10 |
21 |    FP_COMP_OPS_EXE_SSE_FP_PACKED     | 1.16896e+06 |
22 |    FP_COMP_OPS_EXE_SSE_FP_SCALAR     | 5.22953e+10 |
23 | FP_COMP_OPS_EXE_SSE_SINGLE_PRECISION | 1.0281e+07  |
24 | FP_COMP_OPS_EXE_SSE_DOUBLE_PRECISION | 5.22862e+10 |
25 +--------------------------------------+-------------+
26 +-------------------------+-----------+
27 |          Metric         |  core 1   |
28 +-------------------------+-----------+
29 |       Runtime [s]       |  37.5892  |
30 |       Clock [MHz]       |  3427.22  |
31 |           CPI           |  0.51924  |
32 | DP MFlops/s (DP assumed) |  1696.62  |
33 |      Packed MUOPS/s     |  0.037923 |
34 |      Scalar MUOPS/s     |  1696.55  |
35 |        SP MUOPS/s       |  0.333533 |
36 |        DP MUOPS/s       |  1696.25  |
37 +-------------------------+-----------+
```

Listing 2.6: Available Likwid groups on an Intel Nehalem system

```
1  BRANCH: Branch prediction miss rate/ratio
2  CACHE: Data cache miss rate/ratio
3  CLOCK: Clock of cores
4  DATA: Load to store ratio
5  FLOPS_DP: Double Precision MFlops/s
6  FLOPS_SP: Single Precision MFlops/s
7  FLOPS_X87: X87 MFlops/s
8  L2: L2 cache bandwidth in MBytes/s
9  L2CACHE: L2 cache miss rate/ratio
10 L3: L3 cache bandwidth in MBytes/s
11 L3CACHE: L3 cache miss rate/ratio
12 MEM: Main memory bandwidth in MBytes/s
13 TLB: TLB miss rate/ratio
14 VIEW: Double Precision MFlops/s
```

LTTng

The *Linux Trace Toolkit*[45] provides a user-space and a kernel-space tracer [FDD09]. The user space tracer (UST) provides an API by which a developer can instrument the source code. Also, the *GNU Project De-*

[44] Many modern CPUs execute operations that might depend on operations that are still in the pipeline, for example, speculative execution of branches before the branch condition is actually evaluated; if the prediction is correct, the results of the executed instructions are stored, otherwise these results are invalid and must be discarded. When there are no conflicts, a processor retires the instruction allowing write back of the results. For the speed of the execution the speculatively executed instructions are irrelevant, therefore, they are not covered by the CPI metric.

[45] Documentation of the *Linux Trace Toolkit* including all tools is elaborate and available on `http://lttng.org/`. A quick-start tutorial can be found here: `http://omappedia.com/wiki/Using_LTTng`.

bugger (`gdb`) can use UST to record GDB tracepoints. The kernel tracer captures activity for all processes and the system.

Traces are recorded in the *Common Trace Format* (CTF)[46]. The user-space tracer records events with zero-copy[47], resulting in a low overhead of 700 ns per event (according to the documentation). The overhead for the kernel-tracer is even lower.

LLTV is the *LTTng Viewer*, a tool providing statistical, graphical and text-based *views* of the recorded traces. Internally, the individual views are realized as modules. It has been shown capable of handling traces with a size of 10 GiB. Another viewer is incorporated into the *eclipse*[48] IDE as a plugin. With *LTTng*, several traces can be visualized concurrently; this technique has been used to view traces of a virtual machine together with a trace of the XEN host system [DD08].

Exemplary screenshots of LLTV for the desktop system running `partdiff-seq` are discussed next. In Figure 2.12, the *event view* and *control flow view* are presented. The *event view* – shown in the upper half of the screenshots – lists individual events in a textual representation while the *control flow view* – visible in the lower half of the screen – shows the activity in a timeline for each individual process. Graphical representation of the trace encodes the activity of each process in colors; green, for instance, encodes that the process on the left executes in user space (this is *partdiff-seq*).

The *statistical view* and *resource view* are shown in Figure 2.13. In the statistical view, aggregated information of all kinds of activity is provided. In the resource view, the activity on each CPU and interrupt can be observed, white color encodes user-space activity dispatched on a CPU. For easy analysis, the screenshot has been modified to show the likely[49] execution of *partdiff-seq* on the CPUs with green color. The vertical line marks the start of the program, then the single threaded program is migrated between all 4 cores on the quad-core system.

Consequently, with the tool, interactions between processes and system activity at a given time can be analyzed extensively. Note that while the overhead of the tracer is very low compared to other tools tracing 50 seconds of system activity created a set of files with the aggregated size of 57 MiB.

The tool provides capabilities to filter events, which is unfortunately not available for the graphical trace representation.

2.4.4. Available Tools for Analysis of Parallel Programs

Compared to the tools for sequential programs, tools for parallel applications are more involved. In the following, a few tools are presented which are aware of the programming and execution models of parallel applications, hence, they explicitly support the analysis of parallel applications. *VampirTrace* records execution behavior of HPC applications in the *Open Trace Format*[50]. Popular post-mortem performance analysis tools that analyze these trace files are *TAU*, *Vampir* and *Scalasca*. Those tools provide several ways to assist a developer in assessing application behavior.

An experimental tool environment which should be named is PIOviz [LKK+06a]. This environment is capable of tracing not only parallel applications, but also triggered activity on the *Parallel Virtual File System* version 2 (PVFS). In the following, brief introductions and screenshots are provided for all the mentioned tools.

[46] `http://www.efficios.com/ctf`

[47] Zero-copy is a technique in which data is communicated without copying it between memory buffers – in this case no copy between kernel-space and user-space is necessary.

[48] `http://www.eclipse.org/`

[49] It is not possible with the tool to actually relate applications with the resource view, therefore, the execution is guessed by the author based on the information provided by the control flow view.

[50] Further information is provided on Section 2.4.5.