Journal of Physics: Conference Series **396** (2012) 052072 doi:10.1088/1742-6596/396/5/052072

GOoDA: The Generic Optimization Data Analyzer

P Calafiura[1], S Eranian[2], D Levinthal[2], S Kama[3] and R A Vitillo[1]

[1] Lawrence Berkeley National Laboratory , 1 Cyclotron Road, Berkeley, CA 94703, US
[2] Google, 1600 Amphitheatre Parkway, Mountain View, CA 94043, US
[3] Southern Methodist University, Department of Physics, Dallas, TX 75275, US

E-mail: lblcalaf@gmail.com, eranian@google.com, levinth@google.com, sami.kama@cern.ch, ravitillo@lbl.gov

Abstract. Modern superscalar, out-of-order microprocessors dominate large scale server computing. Monitoring their activity, during program execution, has become complicated due to the complexity of the microarchitectures and their IO interactions. Recent processors have thousands of performance monitoring events. These are required to actually provide coverage for all of the complex interactions and performance issues that can occur. Knowing which data to collect and how to interpret the results has become an unreasonable burden for code developers whose tasks are already hard enough. It becomes the task of the analysis tool developer to bridge this gap. To address this issue, a generic decomposition of how a microprocessor is using the consumed cycles allows code developers to quickly understand which of the myriad of microarchitectural complexities they are battling, without requiring a detailed knowledge of the microarchitecture. When this approach is intrinsically integrated into a performance data analysis tool, it enables software developers to take advantage of the microarchitectural methodology that has only been available to experts. The Generic Optimization Data Analyzer (GOoDA) project integrates this expertise into a profiling tool in order to lower the required expertise of the user and, being designed from the ground up with large-scale object-oriented applications in mind, it will be particularly useful for large HENP codebases

1. Introduction

The ATLAS experiment at the CERN LHC has produced tens of Petabytes of data over the last few years which are actively analyzed by a community of more than 2000 physicists using software developed by about 200 developers. Given the healthy competition among analysis groups both within and without ATLAS, efficient use of available computing resources has a direct impact on ATLAS physics reach. ATLAS main performance optimization challenges are to keep under control both the memory footprint of ATLAS applications, and the CPU usage: current ATLAS computing model requires ATLAS applications to use less than 2GB of memory/core, and for every percentage increase in required CPU resources, ATLAS computing procurement and operation costs increase by approximately \$100K/year.

ATLAS has a large C++ codebase: more than 4M lines of code, organized in 4000+ libraries. The OO design and implementation of ATLAS code has traditionally focused more on flexibility than optimal performance. Preliminary performance studies [1] have shown that more than 30% of CPU utilization comes from function call overhead and from L1 Instruction cache latency. Unfortunately none of the tools ATLAS tried, commercial or otherwise, provided the combination of scalability, usability, and detailed analysis needed by ATLAS developers to analyze and monitor performance issues of their large-scale OO C++ applications.

2. Cycle Accounting Methodology

Performance analysis and software optimization is about minimizing the cycle count, so in order to present the assorted measurements of incompatibilities of code and data with the microprocessor architecture in a sensible manner all such performance issues must be presented in the same units, cycles. If one considers the spectrum of such issues, even though there might easily be hundreds of different possible problems on a given processor they can be grouped into roughly a dozen classes that are independent of the microarchitecture. These classes can be organized as a cycle accounting tree. Not all classes will be applicable on all processors, for instance Simultaneous Multithreading (SMT) collisions cannot occur on processors without SMT. Furthermore, on the vast majority of processors, most of the classes cannot be measured in any sort of reliable and accurate manner and evaluated in terms of cycles. If a critical issue cannot be expressed as a cost in cycles then in reality it has not been measured in a useful manner and the user should find a platform where it can be measured and not pretend that a misleading measurement is actually useful.

The usual manner used for evaluating the cost of an issue is by running a customized benchmark that serializes the cost of a single issue and also validates that the processor has an event that counts the occurrences of the issue. The penalty is then evaluated as the difference of the cycles/iteration with the issue minus the cycles/iteration for the kernel when the issue does not occur. The event must also be validated to show it does not count anything else. Under such circumstances the cost of the issue during the execution of a program can then be evaluated as the product of the penalty times the occurrence count. This is effectively serializing execution and thus overcounts the cost as it does not consider temporally overlapping penalties. For some performance bottlenecks it is possible to define an upper limit for the cost of a sum of related issues and thereby allow a correction for overlapping penalties.

This generic performance metric based approach can be applied for workload characterizations with counting mode data collection or using profile data collected with periodic interrupt based sampling. In counting mode it can be done more accurately since the fluctuations of profiling and the precision of the location of the interrupts do not apply. This also means that differences and ratios of performance events can be used, which are usually inaccurate when done at the level of instruction pointer values associated with profiled data.

The generic decomposition is best explained by figure 1, as it is self explanatory for the most part. The branches of the graph are computed with whatever techniques are supported by a given microarchitecture. The techniques frequently change dramatically from one generation to the next as the coverage of the branches improves and new techniques are developed.

The Intel Westmere-EP processors are used as an example as the branch coverage is fairly good. As an illustration of how a branch is evaluated we consider the load latency one as evaluated on a WSM-EP processor. The penalties considered are the extra cycles that a code must wait for load instructions to deliver data to the functional units. On a dual socket system with three cache levels and a Non Uniform Memory Architecture (NUMA) there are a significant number of individual issues with different costs. There are further the costs associated with Data Translation Lookaside Buffer (DTLB) translations and those associated with loads blocked from being able to forward previously stored data residing in store buffers. The penalties were evaluated with specific kernels as discussed above and are processor frequency dependent (both core and uncore frequency).

$$6 * \texttt{mem\_load\_retired:l2\_hit}$$
$$52 * \texttt{mem\_load\_retired:l3\_unshared\_hit}$$
$$85 * (\texttt{mem\_load\_retired:other\_core\_l2\_hit\_hitm} - \texttt{mem\_uncore\_retired:local\_hitm})$$
$$95 * \texttt{mem\_uncore\_retired:local\_hitm}$$
$$250 * \texttt{mem\_uncore\_retired:local\_dram\_and\_remote\_cache\_hit}$$

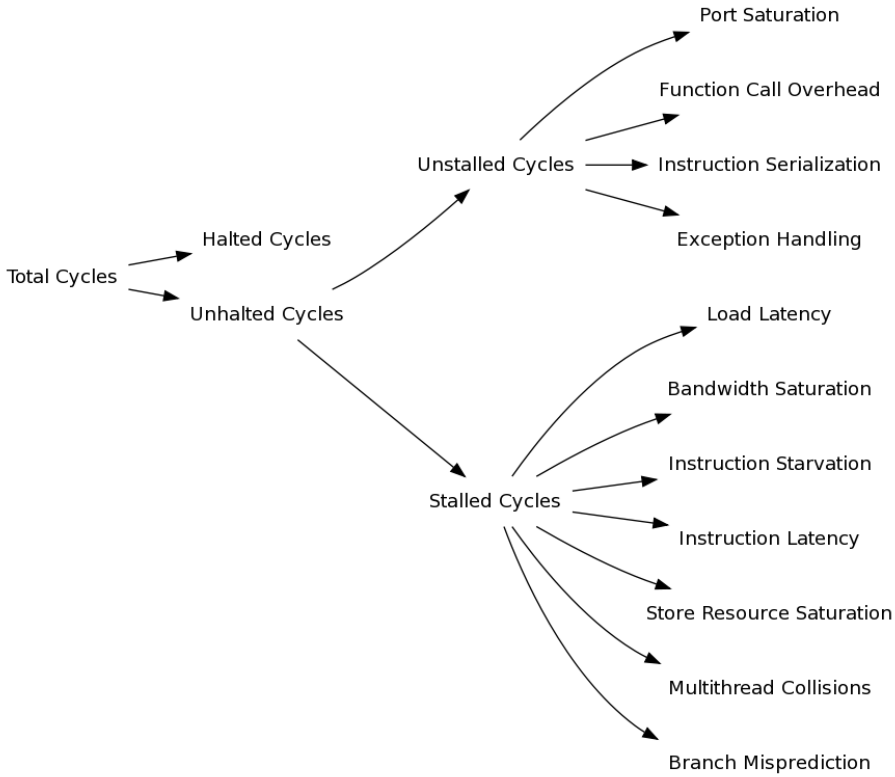

Figure 1. Cycle Accounting Tree

$450 * \texttt{mem\_uncore\_retired:remote\_dram}$
$450 * \texttt{mem\_uncore\_retired:remote\_hitm}$
$250 * \texttt{mem\_uncore\_retired:other\_llc\_miss}$
$7 * (\texttt{dtlb\_load\_misses:stlb\_hit} + \texttt{dtlb\_load\_misses:walk\_completed}) +$
$\texttt{dtlb\_load\_misses:walk\_cycles}$
$8 * \texttt{load\_block\_overlap\_store}$

Small load latency penalties can be hidden by the combination of out-of-order execution and good compilation. So the terms associated with

$6 * \texttt{mem\_load\_retired:l2\_hit}$
$7 * \texttt{dtlb\_load\_misses:stlb\_hit}$

can usually be ignored.

3. Architecture

GOoDA is conceptually composed of 4 main components: the perf_events Linux subsystem, the perf tool, a performance data analyzer and a web based visualizer. The former is a performance monitoring interface that was introduced into the Linux kernel in 2009. The goal was to provide a unified interface to access not only the hardware performance counters, but also kernel software counters and tracepoints.

To demonstrate the capabilities of the perf_events interface, a monitoring tool, called perf has been developed. Unlike other tools, it has been included in the kernel source tree very early on as a way to avoid breakages while the interface kept evolving. The authors have contributed to both of the former projects by fixing bugs and adding several features needed by GOoDA.

The analyzer reads a collected data file from perf, accumulates the statistics per processes, functions, assembly and source lines and generates the various callgraphs and control flow graphs. The web based visualizer is able to read the generated spreadsheets and graph files in order to represent them in a browser.

The following subsections go into the details of the roles of the various subsystem.

3.1. Perf_events

Since 2009, perf_events has been is constant development. First available on X86 hardware, it was rapidly ported to PowerPC, ARM, MIPS, SPARC, and Alpha.

The design of perf_event [2] is radically different from any other existing interfaces. First, this is system call interface, i.e., not a device driver. Second, it exposes a high level abstraction known as an event. Tools manipulate events not counters or registers. Events can count processor cycles, the number of context switches, the number of times a function is called and so on and are encoded as 64-bit integers. The kernel manages the events uniformly regardless of their source, whether processor hardware counters or kernel software counters. Events are manipulated via file descriptors which are obtained via the new perf_event_open() system call. Operations on events then use the regular file interface. To read an event, tools use the read() system call, to start/stop, they use ioctl(), to destroy an event, they call close().

In the past, monitoring interfaces have always exposed the raw hardware. For instance, in perfmon2 [3], tools had to program hardware registers. In OProfile [4], tools had to program counters exposed via a dedicated filesystem. Perf_events hides all the hardware details, including the number of physical counters, the counter constraints on events, the layout of the counter config registers. Users can monitor more events than there are counters. The kernel can time-multiplex events automatically. Similarly, it is possible to share the hardware counters between multiple tools at the same time.

Perf_events exports generic hardware events to count common metrics such as elapsed cycles, the number of instruction retired. Internally, those events are then mapped onto the actual hardware events for the host processor. Although this approach makes developing generic tools easier, it has problems when it comes to interpreting and comparing results. By nature, hardware events are closely tied to the micro-architecture of each processor. Comparing counts between processors may not always be easy because of subtle differences. For instance, on Intel SandyBridge processor, the generic cycles event is mapped to a hardware event which is influenced by the Turbo mode. As such the conversion from cycles to time is not possible unlike on some older processors. Fortunately, to allow power-users to measure all the hardware events, it is possible to pass their raw encodings to the kernel.

The kernel offers two modes of measurement: per-thread and per-cpu. In per-thread mode, events are attached to a thread. They are saved and restored when the thread context switches. In per-cpu mode, any thread running on the monitored logical CPU is included in the measurement. To measure one event across a multi-processor system, it is necessary to program the event on each of the processors. Creating per-cpu events require root privileges for obvious security reasons.

Perf_event also supports a profiling mode where you collect samples during the execution of a program. The sampling period is always expressed as a number of occurrences of an event and not time. All events are counted as 64-bit integers by the interface. However, hardware counters have a fixed width, usually around 40 bits. It is possible to cause an interrupt when the counter wraps around. The mechanism is used for sampling and also for emulating a 64-bit counter

when not available. If the sampling period is P cycles, then the counter is programmed to -P. The interface supports two modes for the period: fixed and rate. In fixed mode, the period is set by tools and does not change. In rate mode, tools specify a target sampling rate in Hertz and the kernel dynamically adjusts the period at each timer tick to try and achieve the rate. The actual period can be saved in each sample to enable normalization of the samples later on. It should be noted that on Intel X86 processors, perf_events uses the Non-Maskable Interrupt (NMI) vector for counter overflows. Thus it is possible to capture samples inside Linux kernel critical sections where interrupts are disabled, thereby minimizing the number of potential blind spots. To prevent against malicious use of the sampling mechanism to flood the kernel with interrupts, perf_events enforces a maximum interrupt rate per CPU per second. Sampling may therefore be throttled and unthrottled dynamically by the kernel.

Samples are saved into a kernel buffer which is made visible to tools via the mmap() system call. By default, there is one sampling buffer per event instance. However, it is possible to force all samples into the buffer of a single event. Along with samples the kernel captures enough meta-data to allow offline analysis and in particular correlation of sampled addresses to actual modules and functions. As such, fork(), exit(), mmap() are recorded in the buffer. Event throttling events are also recorded in the buffer.

Model specific hardware features are also supported by perf_events. For instance, starting with Intel Core, the Precise Event-Based Sampling (PEBS) facility is supported and abstracted as precise sampling mode. It provides a way to collect instruction addresses with a controlled skid unlike regular interrupt-based sampling. Similarly, Last Branch Record (LBR) is supported since kernel v3.3 for Intel Core processors and later. It is abstracted as the branch stack sampling feature. It allows sampling the last N consecutive taken branches that led to an instruction. For each branch, the source and target addresses are captured. This is a very useful feature for statistical basic block profiling and function call counts.

Perf_events can also be used by developers interested in collecting kernel traces. As alluded to earlier, the interface provides access to the same set of tracepoints as the ftrace interface. It is therefore possible to count the number of times the kernel executes through a tracepoint but it is also possible to collect a trace via the sampling mechanism by simply using a period of one event.

The development of perf_events is far from over. Many hardware features are still unsupported, such as AMD Instruction-Based Sampling (IBS), Intel uncore PMU on Nehalem and later processors. But support for those features is being worked on actively by hardware vendors and developers around the world.

3.2. Perf tool

Perf [5] is a command line tool which allows counting and sampling. It follow the git design model with a major command, perf, and sub-commands: list, stat, record, report, annotate. The tool is used to collect data and also analyze them. It has an extensible modular design which makes adding new sub-command relatively easy.

For counting mode, the perf stat command is used. It is possible to measure any of the generic events or raw hardware events. Perf stat also works in system-wide mode and can show a per-cpu break or aggregated view of the counts. To collect samples, the perf record command is used. This generated a binary output file called perf.data which can then be analyze by the perf report command for a function-level view or perf annotate for an instruction-level view. Both commands can be run with a text-based UI.

It is possible to sample on many events at the same time. However, the current perf report command only displays profiles per event and not the samples side-by-side to help locate correlations between events.

Sampling in system-wide mode is also possible. The data from all CPUs are aggregated into

a single perf.data file. Yet each sample includes the CPU number, thus it is still posssible to generate per-cpu profiles.

The perf record tool is used by the GOoDA collection scripts to collects samples into a perf.data file. That file is then parsed directly by GOoDA to generate its output spreadsheets.

3.3. Analyzer

The analysis of the interrupt records in the perf.data file is done in a two phase process. The raw records are processed creating a hierarchical tree of of structures that represents the module memory maps per Process Identifier (PID). PIDs with the same binary path are merged to refer to a single principal process, which is defined as the first PID of the set that gets a sample record. Only the principal process has a tree of module structures where the sample data is accumulated by Relative Virtual Address (RVA). Module structures are created when the first sample is bound to a particular memory map (mmap) record. All mmap structures with the same path, for a family of PIDs defined by having the same principal process, will then have pointers to that module. This completes the merging process.

All structures have a sample data array that accumulates sample counts per core and the totals. The module structures have a linked list of RVA structures (also accessible through dynamic hash lookup) and these thus represent the sample distributions per instruction pointer. Once the raw data has been stored into the memory model, the linked lists of RVA structures are sorted into increasing order based on RVA. The modules structure address ranges are then divided into function ranges using a popen call to readelf [6]. The functions are sorted into increasing address order and the RVA sample data is then accumulated into sample structures for each function.

The process and module structures/process are sorted into descending order by total sample count. A global linked list of all functions with samples is created and it is also sorted into descending order based on total samples. These two operations define the process/module tables and the hotspot function tables.

The hottest N functions (N = 20 by default, set by input argument) are then processed. The address ranges are disassembled. The branch instructions are identified and the list of all branch instructions and targets are used to break the disassembly into basic blocks. Each basic block has a sample structure which accumulates the data from all the instructions in the basic block. The debug information for each assembly address is walked to find the two ends of the inlining chain. For each function a principal source file is identified (allowing for the debug information to be imperfect) and every assembly line has a principal source file line associated with it. Each source file sample line has a sample structure which accumulates the data from all the assembly lines that point to it. This enables the creation of the lined source and assembly tables. The basic block information is used to make a Control Flow Graph using graphviz [7].

Intel Sandybridge processors have a LBR mechanism that is sufficiently sophisticated to create a Call Graph with hardware based data that can be collected with perf. The LBR is filtered to capture only near return branches, thus 16 samples are collected per interrupt vastly reducing the impact of data collection shadowing. The LBR data is processed using each sample in two directions to identify the sources and targets of function calls. The data is stored in linked list structures inside the RVA structures. The data is aggregated per function and sorted by decreasing branch sample count. The top 10 sources and targets for each function are added to the function spreadsheet as expandable rows. A Call Graph is created with graphviz, based on the hottest 50 functions and their sources and targets except for the sources of hot leaf functions that have more than 50 sources. There is no point in drowning the graph in calls to memset and memcpy.

The tables that are created have control structures to enable the expandable cycle accounting display. Each table has a row containing an encoding defining every columns position in the

multi-level cycle accounting tree. There are also rows containing the event names, sampling periods, penalties, multiplexing corrections and whether a column can be displayed in cycles. Thus once the table is created there is no need to go back to the raw data. The architecture dependent cycle accounting is encoded in a template table that is used to drive the evaluation of derived data columns like the branch and sub-branch cycle count values for each table row. Thus a single set of functions appears to be capable of handling almost any architecture that supports real measurements of the generic cycle accounting metrics.

3.4. Visualizer

To display the spreadsheets generated by the analyzer, a web based GUI was developed. The choice of using a web based GUI came from to the need to share a report with more people in an easy way without installing additional applications. We liked the approach taken by callgrind and kcachegrind [8] where each tool has a specific purpose and the responsibility of generating and presenting the data are separated.

With the advent of efficient Just in Time (JIT) engines for Javascript, entire client-side applications based only on the latest HyperText Markup Language 5 (HTML5) web standards became a reality. By HTML5 we refer not only to the markup language and the basic Document Object Model (DOM) scripting API that comes with it, but also the related technologies like Cascading Style Sheets (CSS) 3 and Scalable Vector Graphics (SVG). The main drawback of using these technologies is that the browser vendors not always adhere completely to the standards, so implementing the same functionality on different platforms might become tricky. The situation has clearly improved in recent years and HTML5 fixed some of the incompatibilities between different browsers. Different browsers running on different operating systems can cause further incompatibilities: even something as simple as picking a consistent set of fonts available on each possible configuration of the main operating systems can become not trivial.

The logic behind the visualizer is very simple, it starts by loading the desired spreadsheets using Asynchronous Javascript and XML (Ajax) from the local disk or a web server[1]. Once a report is selected, the spreadsheets corresponding to the hot functions and hot processes are downloaded in parallel. As soon as a downlod is completed the visualizer proceeds to fill the corresponding grid with the downloaded data. To have a completely stand-alone javascript application without any kind of server side interaction, the visualizer requires that the spreadsheets belonging to the reports are pregenerated. In future releases we might explore the possibility to dynamically interact through a webserver with the GOoDA analyzer to generate spreadsheets on the fly. For now, since the analyzer and the visualizer are completely decoupled, a user interested only in viewing some reports can do it on any platform he desires, provided a recent HTML5 compliant browser is installed. To allow the browser to handle tables with hundred of thousands of rows, the internal nodes of the DOM that belong to the tables are dynamically generated only for the currently visible part, and a fraction of the neighboring view of the whole grid. This simple mechanism allows to achieve smooth scrolling without having to create an HTML table with the full size of the grid. This technique also brings a significant reduction of memory usage. The visualizer can be deployed either on a web server or directly be opened through a browser from the local disk.

4. An Use Case

GOoDA has been used to assess performance issues of Athena, the ATLAS analysis software. Perf data collection ran on a dual socket server containing Intel(R) Xeon(R) X5650 CPUs which

[1] at the time of this writing some browsers like Chrome apply some restrictions on file system access via Ajax, but these can be disabled through a browser option.

are based on the Westmere architecture. The machine, running with Hyper-Threading enabled, features 24 virtual cores and 48 GBs of memory.

An optimized x86_64 build of the Athena (17.2.2) software package was selected for the analysis. During the recording, all CPUs were utilized by 32 instances of Athena processes, each doing full offline reconstruction of the collisions recorded by the ATLAS detector. The recording was started at the beginning of the collision events processing and stopped after the last event was processed in order to remove the effects of software initialization and finalization. The total sampling time is of the order of 30 minutes. The recorded perf.data file was then processed by the GOoDA-analyzer and displayed by the GOoDA-visualizer.

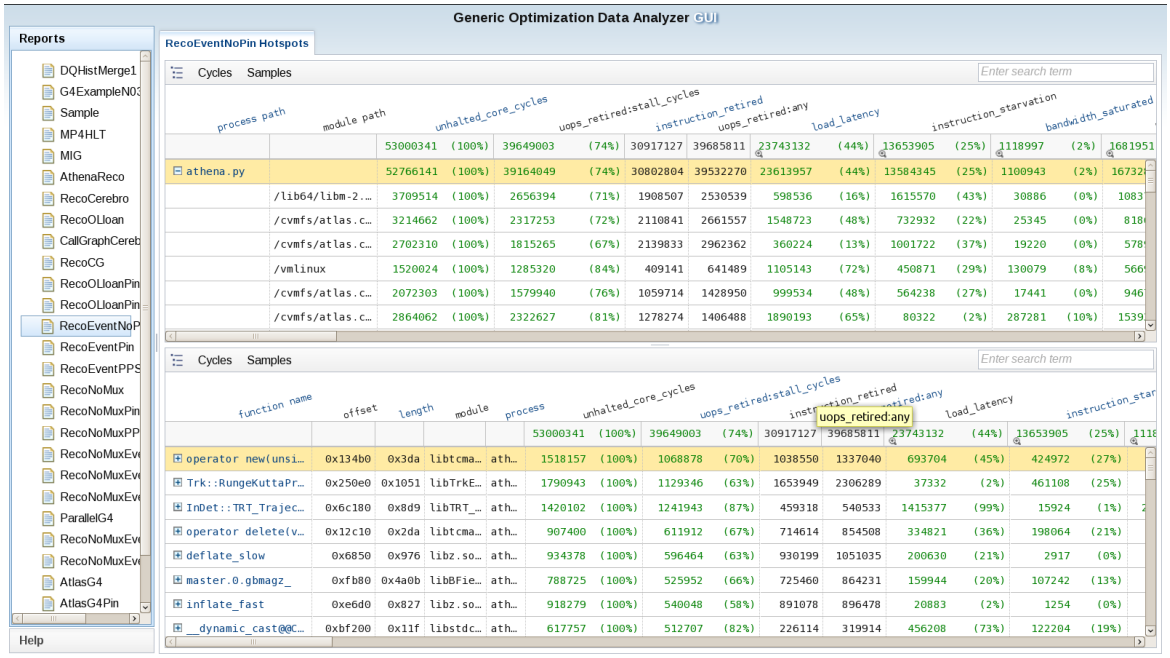

Figure 2. Hotspots View

Figure 2 shows the main display of the report generated by the analyzer. The top part of the window displays the event distributions per process, and when expanded, the statistics for each of the modules. The bottom part shows the hottest functions in the selected process. As it can be seen from the figure, memory allocation and de-allocation, Track Extrapolation in the Transition Radiation Tracker (TRT) and Runge-Kutta propagation are listed as the hottest operations in Athena.

Figure 3 shows the details of the TRT Track Extrapolation code. The bottom part of the window shows the Control Flow Graph of the function. The colors of the boxes represent the hotness of the block: the darker the box, the more samples were collected in it. The top part of the window displays the disassembly of the object file and the relative source code that generated it.

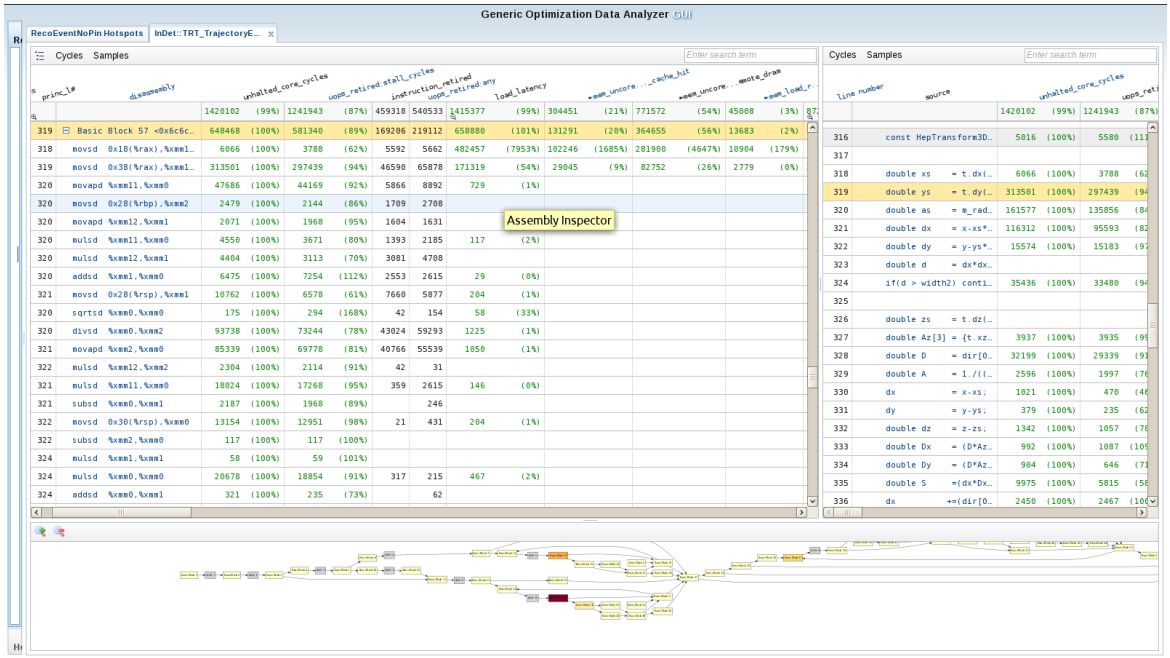

Figure 3. TRT Track Extrapolation

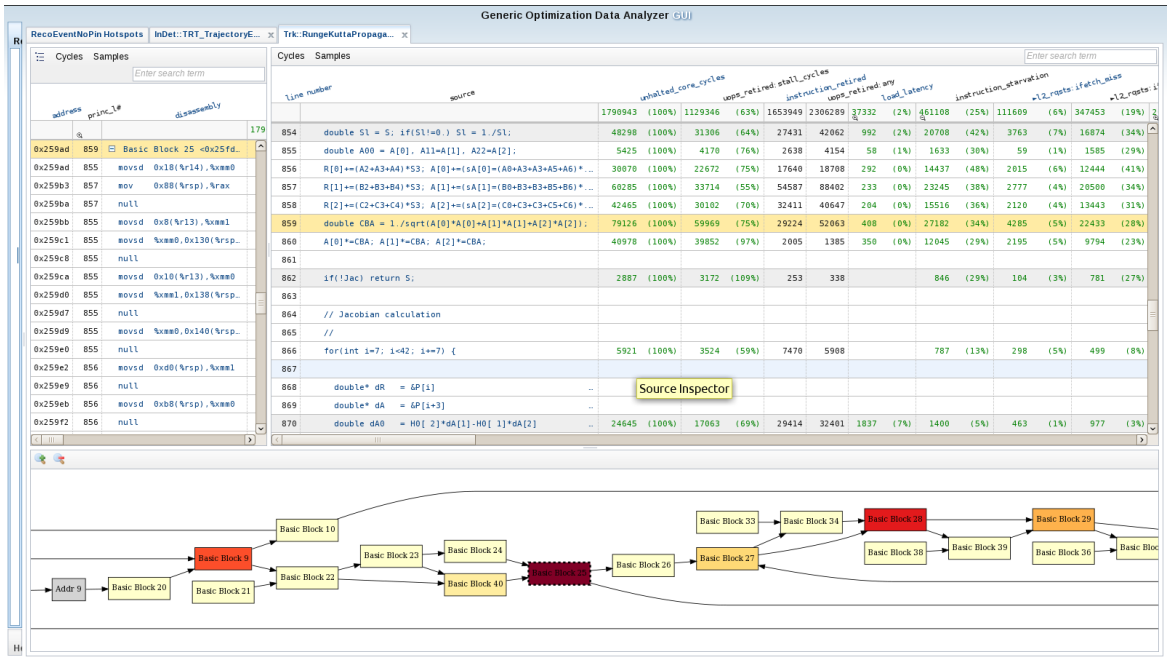

Figure 4. Runge-Kutta Propagation

A glance to the disassembly reveals that the function is dominated by the Load Latency and Bandwidth Saturation branches of the cycle tree. Expanding the Load Latency branch shows

that there are many remote dram accesses which are expensive on NUMA architectures. Remote dram access issues can be mitigated by pinning the application to a socket. Further investigation reveals that the cache misses are caused by memory allocated dynamically in a scattered manner. Allocating the objects in a consecutive manner is expected to alleviate the problem. Also, modifying the data structure in order to take the access pattern into consideration, might further improve the situation.

Figure 4 shows that the Runge-Kutta Propagation code used in the TRT is affected by instructions with high latency. The code block causing the issue is composed of several expensive arithmetic operations such as divisions and square roots. A redesign of the algorithm as well as the usage of vectorization could significantly speedup the code.

5. Conclusions

Thanks to GOoDA's novel approach of cycle accounting, the cycles spent by a process can be divided in categories which highlight the main bottlenecks in HPC and Enterprise applications.

The development of the tool, which was released under an open source license [9], is far from over at the time of this writing, yet it is already fully usable to improve the performance of HENP applications. For instance:

- callgraphs can be used to locate functions with high callcounts and consequently force inlining
- functions dominated by high load latency signal inefficiencies in the usage of the memory hierarchy and suggest the use of alternative data structures
- vectorization opportunities are highlighted by locating functions dominated by high latency instructions

References

[1] C. Leggett, S. Binet, K. Jackson, D. Levinthal, M. Tatarkhanov and Y. Yao, J. Phys. Conf. Ser. **331**, 042015 (2011).
[2] S. Eranian, Overview of the perf_event API, CSCAds workshop, summer 2009, http://cscads.rice.edu/workshops/summer09/slides/performance-tools/cscads09-eranian.pdf
[3] The hardware based performance monitoring interface for Linux, http://perfmon2.sourceforge.net
[4] OProfile - A system Profiler for Linux, http://oprofile.sourceforge.net
[5] S. Eranian et al., Perf tool tutorial, https://perf.wiki.kernel.org/index.php/Tutorial
[6] GNU Binutils, http://www.gnu.org/software/binutils
[7] Graphviz - Graph Visualization Software, http://www.graphviz.org
[8] kcachegrind - Call Graph Viewer, http://kcachegrind.sourceforge.net
[9] GOoDA - PMU Event Analysis Pacakge, http://code.google.com/p/gooda/