Mantis: A Unified Performance and Power Profiling Interface on Heterogeneous Systems

Melanie Cornelius
Department of Computer Science
Illinois Institute of Technology
Chicago, USA
mdooley1@hawk.iit.edu

H.E. Greenblatt
Department of Computer Science
Illinois Institute of Technology
Chicago, USA
hgreenbl@hawk.iit.edu

Zhiling Lan
Department of Computer Science
Illinois Institute of Technology
Chicago, USA
lan@iit.edu

***Abstract*—In high performance computing (HPC), profiling an application on heterogeneous systems is complex. There are many profiling tools, each with their own characteristics: supported architectures, setup and runtime constraints, output data formats, user code requirements, etc. Managing complicated profiling typically falls to the user as an expensive, rapidly-compounding manual process. In this work, we present Mantis, a unified interface to managing complex profiling on heterogeneous systems. It not only provides a simple interface for automating complex profiling via many tools on different devices, but also offers a unified output data format for accelerating post-profiling data analysis. Mantis is modular, transparent to user-code, and easy to use. This paper identifies the use-cases and design challenges for Mantis. An end-to-end example of Mantis in use is also presented.**

***Index Terms*—Measurement tools and techniques; Profiling, trace collection, synthetic traces**

I. Introduction

High-Performance Computing (HPC) is continuously expanding in scope, and the technology supporting HPC has grown to match [1]. To understand the field's expanding capabilities, profiling tools have grown alongside HPC, supporting new hardware architectures, programming models, compilers, and more [2], [3]. Such growth presents a familiar trade-off: more capabilities in HPC means more capabilities in profiling at the cost of increased workload and complexity for the user.

A. Complexity in Profiling

Profiling tools often support a limited number of architectures and platforms. Intel VTune, for example, is a much-used profiling tool which works only on Intel architectures [4]. Similarly, AMD's μProf and NVIDIA's Nsight Systems work only on their so-named hardware/software architectures and platforms [5], [6]. This creates challenges for users evaluating application performance across different architectures since each profiling tool has diverse and specific requirements.

In addition, many profiling tools impose requirements on user code. For example, PAPI is a much-beloved tool in HPC, but it requires direct integration into user code [7]. This means the user must understand both what they wish to profile and where in their code it would be useful to perform the profiling, and neither is guaranteed to be trivial or directly relevant to the user's study.

Finally, profiling tools offer an enormous variety of data output formats. Digesting and combining results, especially across tools and architectures, falls to post-processing. Considered in a vacuum, post-processing data is neither difficult nor complex—but when compounded across every tool, architecture, and measurement under study, post-processing can become tedious to the user. Even when two profiling tools, say Linux perf and Intel VTune, offer a CSV output format, these CSVs do not share a schema and cannot necessarily be merged into one file [8], [4]. This holds for more complex formats, such as databases, and even more so for proprietary or tool-specific formats like some used with NVIDIA profilers [6], [9], [10].

Consequently, the user wishing to profile an HPC application has to contend not just with their own code and field of study, but also the requirements of their profiling tools and data post-processing. In this work, we present Mantis to address the above problem. More specifically, *Mantis is built to automate and simplify application profiling on the user's behalf.*

Figure 1 highlights the differences between profiling with Mantis versus profiling with existing profiling tools. The left column lists some of the manual work involved in existing profiling. For every component in their system (CPUs, GPUs, host-side system elements like memory, etc), users must select appropriate profiling tools, configure the tool to collect the desired measurements, run the tool with their code, and manually reformat the data for later analysis—possibly many times, depending on the quantity of variables under study. The right column in this figure shows the simplification of profiling via Mantis. Once a user installs Mantis and modifies a Mantis-supplied configuration file, Mantis manages and runs the profiling tools needed to gather the requested measurements. In the end, the user is handed a single, unified data format containing all requested measurements over all code variations, experiment repetitions (iterations), profiling tools, and so on. Mantis takes a tedious, manual process and turns it into a simple, automatic one—and the user's only obligation is to modify a configuration file provided by Mantis.

B. Mantis Highlights

Mantis is developed to tackle the complexity of profiling across many diverse systems, offering modularity and user-

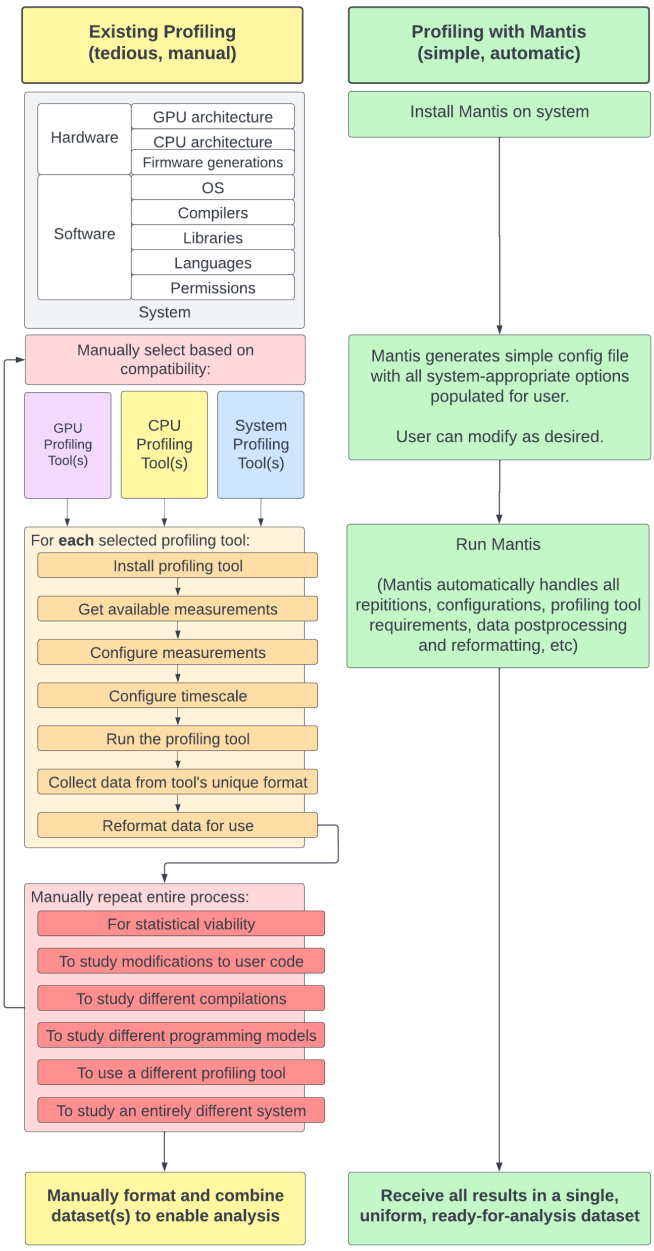

Fig. 1. Existing profiling versus profiling with Mantis

simplicity unmatched by the authors' experiences with existing tools. It has two design goals: (1) Mantis offers a simple user interface for managing complex, multi-tool profiling across diverse heterogeneous systems, and (2) Mantis handles post-processing requirements to output in a a unified data format (UDF).

These goals are achieved through a standard, robustly modular architecture which can be easily extended across many hardware and software architectures. Furthermore, Mantis is designed around an intentionally simple user interface—a configuration file. If the user's chosen system can run their code and the necessary profiling tools, Mantis handles much of the manual work on the behalf of the user.

A key design challenge is to balance supporting as many profiling tools as possible (such as Nsight Systems, µProf, perf, and many more) and post-processing each tool's output into a single, uniform format—all while keeping the user's manual processing and wall-clock runtime to a minimum.

C. Paper Outline

This paper outlines Mantis's capabilities and its design techniques. We discuss the architecture of Mantis in terms of design techniques in Section II. We then present several use cases in Section III. We discuss future work in Section IV, followed by the conclusions.

II. Mantis Architecture

The design of Mantis contains three key parts: a *modular design* to support different tools, and architectures, a *unified output data format*, and *a configuration file* as the user interface.

A. Mantis Requirements

Mantis requires Linux. It is implemented in Python and requires Python 3.4+. It currently contains 1,600 lines of code and is still growing. Mantis is open-source under the GNU Lesser General Public License v2.1 [11]. The source is available via GitHub [12]. Instructions for installation are located in the GitHub repo.

Each profiling tool supported by Mantis requires support on the host platform. If a user wishes to make use of measurements collected via perf, for example, perf must be both installed on that system and available to the user [13]. The command

```
mantis-monitor --detect_tools
```

outputs a list of profiling tools available on the system and supported by Mantis.

B. Profiling Tools Supported

Mantis currently supports profiling via Linux perf [8], the /proc filesystem [14], and several NVIDIA profiling interfaces [6], [9], [10]. Work to support similar profiling interfaces for NVIDIA and Intel GPUs via AMD's µProf suite and Intel's VTune is ongoing.

1) Linux Perf: Linux perf is a tool to measure performance counter events and metrics according to the host CPU's architecture [8]. When using Linux perf, a user may want to collect many performance counters over the course of their code's runtime. Linux perf is intentionally low-overhead and very suitable for precise profiling tasks—unless incorrectly used. It is possible to request performance counters very frequently, or to request very many of them; this can interfere with performance, as only so many counters can be monitored at a time without thrashing the registers holding those counters [13].

Mantis handles these concerns for the user. First, Mantis enforces a lower limit (50 milliseconds) on the requested timescale (see the listing at 1). Next, Mantis ensures all

requested counters are measured, but if many counters are requested, the perf collector may divide the requested data across multiple runs, calling into the user's code—and perf—multiple times to reduce any risk of performance interference.

2) The /proc Filesystem: /proc is a virtual filesystem sometimes referred to as a 'process information pseudo-filesystem' [14]. When run on Linux, a process populates a "file" in the OS's /proc filesystem. This data contains runtime system information and is widely used by system utilities, such as sysctl, lspci, and lsmod. While it is possible to make use of this system manually, collecting meaningful data usually entails writing scripts or modifying user code.

Mantis handles reading the /proc filesystem on behalf of the user, offering information on the user's code from the OS's perspective. Mantis can, for instance, collect the current memory in use for a family of user processes, or track the high-water mark of memory use over time. It can also collect detailed information on how long each user process ran to get detailed wall-clock times.

3) NVIDIA Profiling Tools: When profiling on an NVIDIA GPU, there are many (many) profiling data options available. Mantis tries to support as many of these as possible, working around the many complexities involved when profiling on a GPU.

First, Mantis handles using nvprof [9], the old profiling system, as well as the Nsight Systems suite [6], the new profiling system. Generally, the same data can be gathered from either tool, but the specifics of usage vary wildly [15], [16]. Then, much data can be gathered from NVIDIA SMI, the more general system-side tool for monitoring the physical stats of an NVIDIA device. Which tool a user should use (and which profiling options are thus available) depends on the NVIDIA GPU generation—and Mantis handles this complexity.

Mantis offers the user its best guess on the generation of NVIDIA GPU in the system when generating a sample configuration file. It will then populate the available profiling "modes" for that generation. Currently supported collection modes include power_time, utilization_time, memory_basic_time, temperature_time, clocks_time, and gpu_trace. The time-series measurements are collected once per timescale (as set in the config file), while the GPU trace option offers summary information on GPU API activities, including CUDA calls, OpenCL calls, memory movement commands, and many more. Each NVIDIA profiling mode is documented in the wiki.

More collection modes for NVIDIA GPUs are under development.

C. Profiling at Runtime and the Unified Output Format

Profiling tools, ranging from Linux perf to Intel's VTune to the NVIDIA Nsight Systems suite, tend to be highly configurable. From selecting the desired measurements, timescales, and units to the desired output format, profiling tools tend to look very different at runtime, and most have the ability to output to different formats.

This flexibility is convenient when using an individual profiling tool or two, but it rapidly grows into a daunting and expensive time-sink when dealing with many measurements, output schema, architectures, hardware devices, operating systems, and so on.

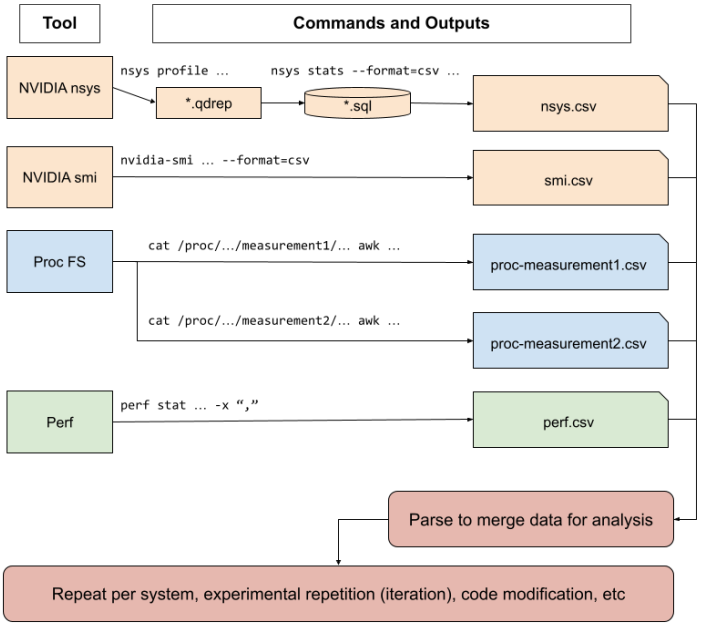

Fig. 2. Existing profiling and post-processing pipeline, where the bottom boxes (in red) highlight the tedious manual processing involved in existing profiling tools.

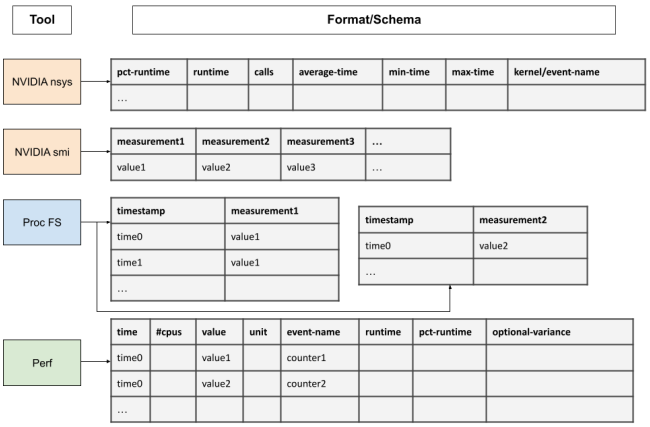

Fig. 3. Different data formats/schema generated from existing profiling tools.

Figure 2 presents the manual processing and post-processing pipeline involved in existing profiling tools. Note this pipeline must be repeated for every experiment repetition, code change, system change, and so on. Additionally, even if a user requests one generic output format from all tools, like CSV files or SQL databases, the schema are seldom able to be joined. Figure 3

presents example data formats/schema from different profiling tools listed in Figure 2 [17] [18] [19] [20].

Mantis is designed to provide an automatic and unified interface across different profiling tools. It handles the complexity of running different profiling tools on the user's behalf. As long as the system *can* run the profiling tool, Mantis will handle the details. Additionally, every profiling tool integrated with Mantis has its output data post-processed into a unified data format (UDF). The UDF can be output in a number of filetypes, such as to a CSV file, JSON file, SQL database, or a Python Pandas DataFrame [21]. The UDF holds regardless of the system on which Mantis is used, so all data collected by Mantis can be combined (or converted) using Mantis.

Any UDF file can be converted to another filetype using the command

```
mantis-monitor --convert_data=types filename
```

where *types* is a comma-separated list of supported types to convert to (JSON, CSV, pkl, SQL) and *filename* is the UDF file to convert from. So, for example,

```
mantis-monitor --convert_data=json,csv filename.sql
```

will output the data from filename.sql as a JSON and CSV file. Mantis can also merge any UDF files using the command

```
mantis-monitor --merge_data=filenames types output-filename
```

where *filenames* is a comma-separated list of all files to combine, *types* is a comma-separated list of supported types to save the file as, and *output-filename* is the name to use for the saved file(s).

A sample UDF CSV file is shown in Figure 6, and full details of the UDF schema can be found in the Mantis Wiki.

D. User Interface

1) Running Mantis: Installing the mantis_monitor Python package (instructions on the GitHub repository README) provides an executable named mantis-monitor. The command

```
mantis-monitor config.yaml
```

runs Mantis using the configuration specified in config.yaml.

Advanced users may also choose to use Mantis as a package.

2) The Config File: The user's primary interaction with Mantis occurs in the configuration file. See code listing 1 for an example.

Listing 1. Example Config File

```
1  benchmarks:
2    CustomThing1:
3      cmd: ./run/user/code_A
4      name: CustomUserCode1
5    CustomThing2:
6      cmd: ./run/user/code_B --options
7      name: CustomUserCode2
8
9  collection_modes:
10   perf:
11     perf_counters:
12       - instructions
13       - major-faults
14       - branch-instructions
15       - branch-misses
16       - bus-cycles
17       - cache-misses
18       - cache-references
19       - cpu-cycles
20       - context-switches
21       - cpu-migrations
22       - major-faults
23       - minor-faults
24   nvidia:
25     modes:
26       - power_time
27       - utilization_time
28       - memory_basic_time
29       - temperature_time
30       - clocks_time
31       - gpu_trace
32     gen:
33       - sm_80
34   memory:
35     modes:
36       - high_watermark
37       - memory_over_time
38       - api_trace
39       - gpu_trace
40       - power_over_time
41
42 formatter_modes:
43 - CSV
44 - SQL
45
46 time_count: 1000
47 iterations: 2
48 test_name: sample_config
```

This configuration file contains the following:

- Commands to run two different invocations of user code (lines 1-7)
- Options for running perf, NVIDIA, and host-side memory profilers (lines 9-40)
  - A list of perf counters to measure (lines 10-23, verified against the system's available counters)
  - A list of collection modes to run on the NVIDIA GPU (lines 24-33)
  - A list of collection modes for host-side memory measurements (lines 34-40)
- Options for formatting the output (lines 42-44)
- Options for measurement frequency (in ms, line 46) and number of overall experiment repetitions (iterations, line 47) to run
- The test name to attach to the output data (line 48)

This is the extent of the user's interaction with Mantis. As long as the commands entered for the benchmarks effectively run the user's code, Mantis will run the requested profiling tools and collection modes around the user's code, gather all data, repeat over the requested number of iterations, and output to the requested data format.

To simplify determining what tools and measurements are available, Mantis can provide a user configuration file populated with all measurements possible on that system. This is done via the command

```
mantis-monitor --generate_config
```

On a system with many available profiling tools, the resulting configuration file will have many measurements the user should prune to their requirements.

Documentation of all Mantis configuration options is available on the Mantis GitHub wiki.

E. Modular Design

In order to provide the flexibility required for profiling on many architectures and platforms, and in order to ensure convenience for the user and correctness of the unified data format, the architecture of Mantis is robustly modular.

Figure 4 shows the four main components of Mantis, while Figure 5 provides a key for the symbols used in the former. The design leverages standard object-oriented interfaces to a simple central controller. This allows for separation of concerns and keeps necessary component entanglements tidy. Each component will be discussed briefly with an emphasis on the challenge it overcomes.

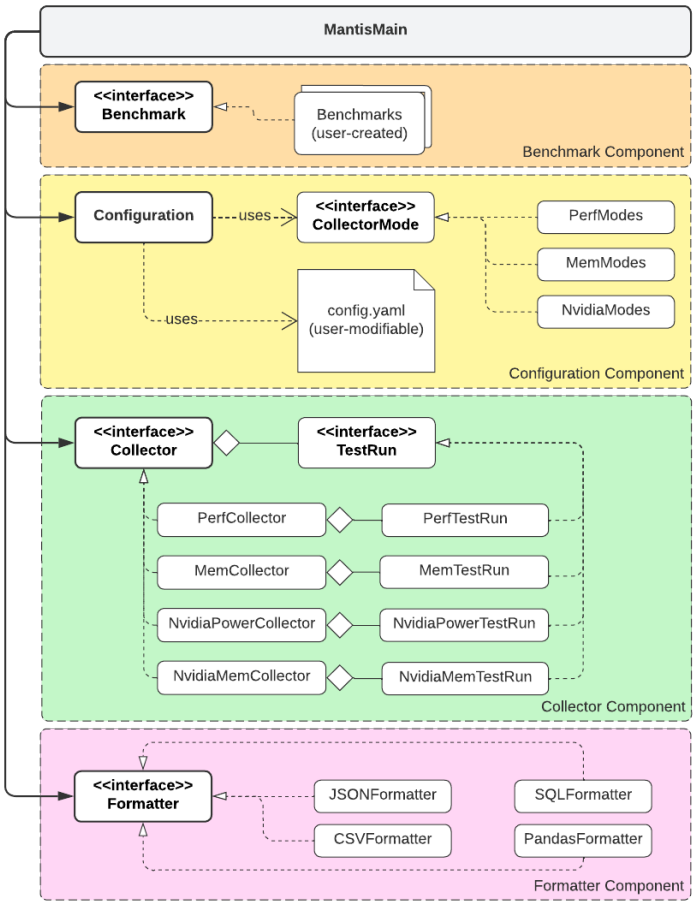

Fig. 4. Mantis classes and relationships

This architecture is extensible; a user interested in implementing a custom Benchmark, Collector, or Formatter need

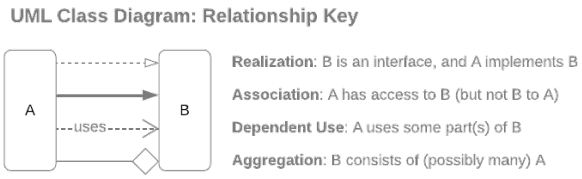

Fig. 5. Relationship types

only import the mantis_monitor package, define a class inheriting from the relevant interface, and invoke mantis-monitor. This is completely optional—Mantis tries to anticipate user needs with its built-in classes, and ideally configuration files will almost always be sufficient—but it lends Mantis a fully general power. Several examples are given in the GitHub repo, and step-by-step instructions are available on the wiki.

1) Configuration Module: The Configuration module is one of the least complex but most important components in Mantis. The user's config file is ingested and parsed into options specifying how to run the user's code, what profiling data they would like, and their preferred output format(s). While this role is vital, the functionality is intentionally simple.

First, the Configuration class can generate a sample config file for the user containing all appropriate profiling tool options for their system. This is useful because tools offer different options on different systems. For example, Linux perf may not offer the same event counters on different CPUs [8]. Alternatively, the profiling modes available on an NVIDIA A100 differ from those offered for a V100, since the supported profiling tools are different [15].

Second, when Mantis is invoked, the Configuration class parses the user's config file into a map of selected options, which are then used to invoke all further module components.

2) Benchmark Module: Another intentionally simple module in Mantis is the Benchmark module. Here, using the options provided in the config file, instructions for running a user's code are stored for use by Mantis during profiling. This can happen in one of two ways.

The simplest and by far most common Benchmark class implementation is a generic form. In the config file, the user simply provides a command line to run and some name to give the experiment. As many experiments and different command lines as needed can be passed into the config file, and Mantis will run each to profile for the requested data. An example of this can be seen under the "benchmarks:" heading in the code listing at listing 1.

In some cases, however, this may not cover all user needs. For example, it may be necessary to perform major setup/teardown actions between each test run, like recompiling under a different compiler or cleaning up intermediate files. To accommodate this, special attention has been given to the API of the Benchmark interface class. Users who choose to implement a custom Benchmark have access to fine-grained, complex control, including hooks before and after each it-

eration and each benchmark, utility functions dealing with common patterns, and the ability to define and handle custom config file options. This API is documented on the wiki.

Implementing a custom Benchmark class is recommended only for very complex user code invocation. In general, users need only provide to their config file the command line they use to run their code, and Mantis will do the rest.

3) Formatter Module: The final simple module in Mantis is the Formatter interface. Since all data is stored in the UDF within Mantis, Formatter classes are simply instructions for converting the internal form of UDF (currently a Python Pandas DataFrame) into the desired file format and vice-versa. Currently, supported Formatters include CSV files, JSON files, SQL databases, and Python pickles [22] of the Pandas DataFrame.

While Formatter implementations are not equivalent (the SQL database looks very different from the CSV file), all use the UDF. Mantis is perfectly capable of ingesting multiple output files from multiple runs of Mantis, regardless of the Formatter type(s) used to generate the files, and combining them into a single internal UDF to output to the user's desired new format. This is especially useful when profiling across many heterogeneous systems or when combining results from many invocations of Mantis. The wiki contains much information on how to call Mantis to combine datasets in this way.

In general, the functionality of the Formatter implementations simply allow the user to select their output filetype and to combine datasets when desired.

4) Collector Module: The Collector implementations are the most functionally complex Mantis components by far.

In Mantis, any Collector class implementation functions as a profiling tool. This generally takes one of two forms: (1) Calling an external profiling tool with appropriate arguments and passing this tool the user's code, or (2) starting and tracking a profiling tool in the background, then initiating the user's code, then stopping the profiling tool's process. In either case, the details of invoking user code are delegated to the Benchmark module(s) in use, and all invocations of profiling tools happen from within Python.

Collectors currently exist to monitor the /proc filesystem [14], Linux perf [8], and several NVIDIA tools [6], [9], [10]. Depending on the user's config file options, these tools are invoked in different ways and with different options to gather the requested data. This data is then converted to the internal Mantis UDF. The details of this vary enormously between profiling tools and are often subtly (or blatantly) complex.

Other complexities, outside the details of the profiling tools supported by each Collector class, are handled implicitly by the Collector interface. For example, all data is ingested at the Collector level and transformed to the internal UDF for later output. The Collector classes also attempt to overlap their measurements as much as possible to avoid re-running user code and wasting resource time. Collector implementations are the granularity at which Mantis experiments can be paused and re-engaged, so if a user's invocation of Mantis, say, overruns a job's scheduled time, the user can re-invoke Mantis on the system in a new job, and Mantis will pick back up where the previous Collectors left off to finish the user's requested measurements.

Implementing a Collector is still kept as simple as possible, should a user want to support a new profiling tool not yet offered by the Mantis suite. This is done through the modular design of Mantis and through the use of many helper utilities. Very detailed information on this can be found on the wiki. In general, users need not be concerned with new profiling tools and can just focus on using Mantis's rich set of current options to understand their code.

The Collector modules are a powerful, complex, and ever-growing vitality to Mantis. These Collectors are the backbone of Mantis's purpose—to support, simply, profiling user code on heterogeneous systems without backbreaking user effort in combining, tracking, and processing their profiling data.

III. Mantis Use-Case

To prove capability, Mantis was studied on a system across three microbenchmarks and multiple compilers as well as through one application study.

A. Experiment Platform

The system used was ThetaGPU at Argonne National Laboratory [23]. One node was used at a time. ThetaGPU contains 24 NVIDIA DGX A100 nodes. Each DGX A100 node comprises eight NVIDIA A100 Tensor Core GPUs and two AMD Rome 64 core CPUs. The nodes used for this study included 320 GB of DDR4 host-side memory as well as 320 GB GPU memory, four 3.84 TB Gen4 NVME drives, and the network supports up to per-node 25 gigabits per second in bandwidth. The dedicated compute fabric in ThetaGPU comprises 20 Mellanox QM9700 HDR200 40-port switches wired in a fat-tree topology.

B. Microbenchmarks

Two microbenchmarks, XSBench [24] and RSBench [25], were studied to show Mantis's capabilities on a heterogeneous system employing multiple programming models (CPU-only, offloading to a GPU, different compilers, etc). Both run a core kernel to a Monte-Carlo neutron transport algorithm. XSBench uses the standard kernel, while RSBench employs a technique to use an order of magnitude less memory but requires expansion for computation.

XSBench and RSBench were profiled using several collection modes including a selection of perf counters and the NVIDIA power over time mode. Additionally, several programming models for XSBench and RSBench were examined, including the CUDA, OpenMP threading (CPU-only), and OpenMP offload (GPU-offload) versions.

Figure 6 shows a sample of the resulting UDF output. Each row contains data from a single experimental run. The "collector_name" column gives the unique name of the Collector used during that benchmark run. "iteration", "timescale (ms)", "units", and "measurements" are metadata columns

benchmark_name	collector_name	iteration	timescale	units	measurements	instructions	L1-dcache-load-misses	...	gpu_0_power.draw
XSBench_cuda	PerfCollector_0	0	1000	count per timescale milliseconds	['instructions', 'L1-dcache-load-misses',...]	[(1.00819863, 1640576043.0), (2.013520331, 5635473743.0), ...]	[(1.00819863, 36339614.0), ...]		null
XSBench_cuda	NvidiaPowerTime	0	1000	time, W	['power.draw']	null	null		[('2022/06/24 09:15:45.202', 74.57), ('2022/06/24 09:15:46.236', 70.68),...]
...									
XSBench_openmp-threading	PerfCollector_0	0	1000	count per timescale milliseconds	['instructions', 'L1-dcache-load-misses',...]	[(1.008215711, 2964467925.0), (2.013634666, 99262274025.0),...]	[(1.008215711, 73657798.0), (2.013634666, 119936937.0),...]		null
...									
XSBench_openmp-offload	PerfCollector_0	0	1000	count per timescale milliseconds	['instructions', 'L1-dcache-load-misses',...]	[(1.008092552, 741121793.0), (2.013521884, 3572111864.0), ...]	[(1.008092552, 16514271.0), (2.013521884, 81200111.0),...]		
XSBench_openmp-offload	NvidiaPowerTime	0	1000	time, W	['power.draw']	null	null		[('2022/06/24 09:25:22.239', 71.54), ('2022/06/24 09:25:23.265', 70.95),...]
...									
RSBench_cuda	PerfCollector_0	0	1000	count per timescale milliseconds	['instructions', 'L1-dcache-load-misses',...]	[(1.008158606, 1495457249.0), (2.013062106, 6548343980.0),...]	[(1.008158606, 28291916.0), (2.013062106, 7649611.0),...]		null
RSBench_cuda	NvidiaPowerTime	0	1000	time, W	['power.draw']	null	null		[('2022/06/24 14:19:21.109', 73.62), ('2022/06/24 14:19:22.134', 239.39)]

Fig. 6. The unified data format from Mantis, which shows a subset of microbenchmark results.

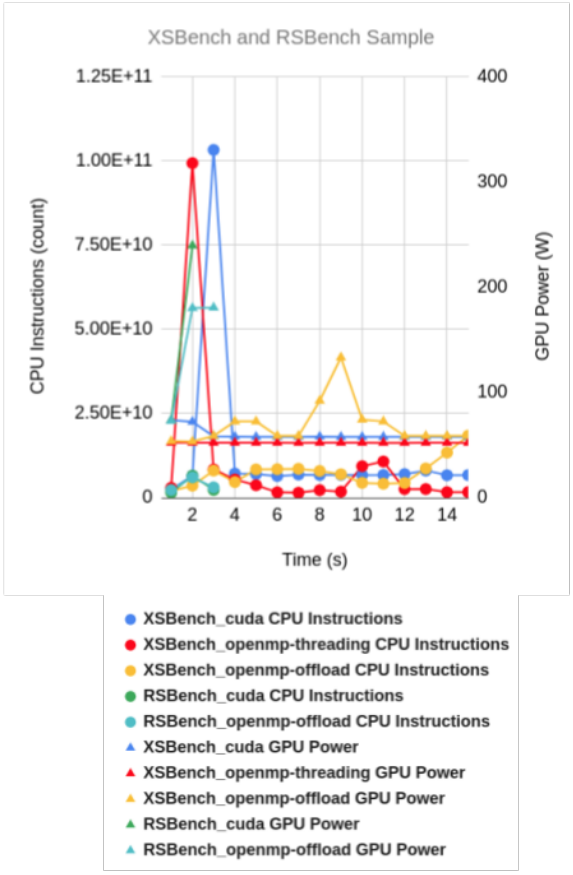

Fig. 7. Subset of microbenchmark results showing ease of analysis when profiling on heterogeneous systems

which describe the data in the following cells. The remaining columns each correspond to one of the entries in the "measurements" cell of the same row, and contain the value of that measurement as collected during that benchmark run. Many values collected by Mantis take the form of a (time, value) array of tuples. Other values take the form (descriptor, value) for non-timeseries measurements, such as the memory high watermark value for a benchmark run. See the wiki for details on Formatter output options.

Figure 7 then shows this data plotted against time. Note that this is only a small subset of the data collected overall by Mantis, even for this simple test. (See the wiki for the full dataset collected during this example.) As we can see from this sample of measurements, Mantis easily enables direct comparison of measurements across a heterogeneous architecture.

C. Application Case-Study

An application, PythonFOAM [26], was also studied. PythonFOAM is a computational fluid dynamics toolbox based on the popular OpenFOAM [27]; it enhances OpenFOAM (which is written in C++) with Python in order to add powerful, flexible *in situ* data analysis capabilities, including machine learning [28]. Its combination of distinct computational workloads—and its capacity for offloading its data analysis to dedicated hardware—make it, likewise, a useful showcase for heterogeneous profiling under Mantis.

In particular, the AEFoam example solver, which trains a TensorFlow [29] autoencoder to generate low-dimensional representations of the simulation state, was profiled. Profiling included selected CPU perf counters, GPU trace data, and GPU utilization and power consumption over time. AEFoam was run under GPU-enabled TensorFlow both with the GPU visible and without (in pure-CPU mode). This configuration, in addition to the setup and teardown of the OpenFOAM

case directory, was handled cleanly and replicably from within Mantis.

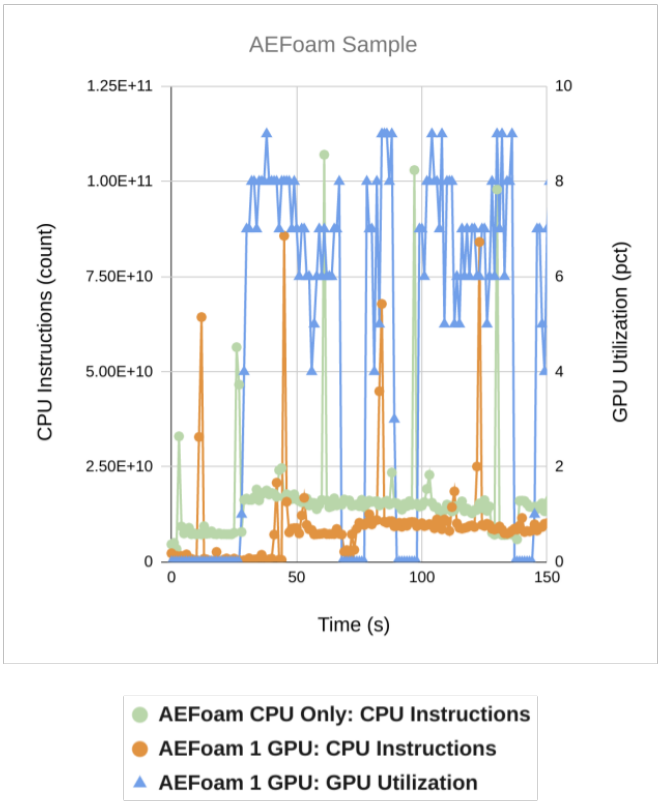

Fig. 8. Subset of AEFoam results using Mantis

Figure 8 shows a very simple example of an identifiable program inefficiency. Note the phases between CPU instructions per second and GPU cycles per second. Both measurements show periods of low activity, and these periods often overlap, suggesting a more optimal order or arrangement of tasks could be achieved. (In fact, this version of AEFoam alternates between CFD simulation on the CPU and autoencoder training on the GPU, when it could parallelize the tasks by offloading and training the autoencoder on the previous batch of snapshots while collecting the current one.) This is simply an example; more sophisticated analysis of the data could yield more profitable insights, including into PythonFOAM's design objective—tuning the tradeoff between computation and I/O.

IV. Future Work

Mantis is in active development and use. Current tasks include (but are not limited to) implementing AMD GPU profiling modes, integrating with Intel's VTune for Intel GPU profiling, expanding to operating systems other than Linux, and deploying and testing Mantis on additional public systems. In addition, investigation into collector parallelizability is ongoing. Finally, utility functions and user-friendly features (such as progress display and suspend/resume) are being added.

V. Conclusions

Mantis is an interface developed to simplify profiling user code on heterogeneous platforms. It provides an unified interface between users and various profiling tools deployed on heterogeneous CPU-GPU systems. The goals of Mantis are simple: (1) to automate performance and power profiling on heterogeneous systems, and (2) to simplify post-profiling analysis via a unified data output. Mantis achieves this, overcoming many challenges in design and implementation, using a highly-modular and easily-extended architecture. The only user interactions required involve modification of a configuration file to inform Mantis on what code to run and which metrics to profile for. Mantis does the rest—including repeating measurements for statistical viability, handling the complexities of running different profiling tools (such as NVIDIA's Nsight Systems or Linux perf), and postprocessing the resulting data into a unified data format. Mantis is under constant development to better support its goals and is licensed open-source at the given GitHub repo.

Acknowledgment

This work is supported in part by US National Science Foundation grants CCF-2109316, CNS-1717763, and CCF-2119294 and U.S. Department of Energy, Office of Science, under contract DE-AC02-06CH11357.

References

[1] Giovanni Agosta, William Fornaciari, Giuseppe Massari, Anna Pupykina, Federico Reghenzani, and Michele Zanella. 2018. Managing Heterogeneous Resources in HPC Systems. In Proceedings of the 9th Workshop and 7th Workshop on Parallel Programming and RunTime Management Techniques for Manycore Architectures and Design Tools and Architectures for Multicore Embedded Computing Platforms (PARMA-DITAM '18). Association for Computing Machinery, New York, NY, USA, 7–12. https://doi.org/10.1145/3183767.3183769.

[2] Mantovani, F.; Calore, E. Performance and Power Analysis of HPC Workloads on Heterogeneous Multi-Node Clusters. J. Low Power Electron. Appl. 2018, 8, 13. https://doi.org/10.3390/jlpea8020013.

[3] A. D. Malony et al., "Parallel Performance Measurement of Heterogeneous Parallel Systems with GPUs," 2011 International Conference on Parallel Processing, 2011, pp. 176-185, doi: 10.1109/ICPP.2011.71.

[4] "Intel® Vtune™ Profiler." Intel, https://www.intel.com/content/www/us/en/develop/documentation/vtune-help/top/installation.html.

[5] "AMD UPROF." AMD, 18 Jan. 2022, https://developer.amd.com/amd-uprof/.

[6] "NVIDIA Nsight Systems User Guide." NVIDIA Documentation Center, https://docs.nvidia.com/nsight-systems/UserGuide/index.html.

[7] Terpstra, D., Jagode, H., You, H., Dongarra, J. "Collecting Performance Data with PAPI-C", Tools for High Performance Computing 2009, Springer Berlin / Heidelberg, 3rd Parallel Tools Workshop, Dresden, Germany, pp. 157-173, 2010.

[8] "Perf: Linux Profiling with Performance Counters." Perf Wiki, https://perf.wiki.kernel.org/index.php/Main_Page.

[9] "Nvprof User's Guide." NVIDIA Documentation Center, https://docs.nvidia.com/cuda/profiler-users-guide/index.html#nvprof.

[10] "Nvidia System Management Interface." NVIDIA Developer, 31 May 2022, https://developer.nvidia.com/nvidia-system-management-interface.

[11] "GNU Lesser General Public License V2.1 - GNU Project - Free Software Foundation." Www.gnu.org, www.gnu.org/licenses/old-licenses/lgpl-2.1.en.html. Accessed 16 July 2022.

[12] Cornelius, Greenblatt (2022) Mantis-Monitor (Version 1.0). https://github.com/SPEAR-IIT/mantis-monitor

[13] "Tutorial - Perf Wiki." Perf.wiki.kernel.org, perf.wiki.kernel.org/index.php/Tutorial#Troubleshooting_and_Tips. Accessed 16 July 2022.

[14] “The /Proc Filesystem.” The /Proc Filesystem - The Linux Kernel Documentation, https://www.kernel.org/doc/html/latest/filesystems/proc.html.
[15] “Profiling - NERSC Development System Documentation.” Docs-Dev.nersc.gov, docs-dev.nersc.gov/cgpu/software/profiling/. Accessed 15 July 2022.
[16] Yang, Charlene. ”Hierarchical roofline analysis: How to collect data using performance tools on intel cpus and nvidia gpus.” arXiv preprint arXiv:2009.02449 (2020).
[17] “Perf-Stat(1) - Linux Manual Page.” Perf-Stat(1) - Linux Manual Page, CSV_FORM, man7.org, https://man7.org/linux/man-pages/man1/perf-stat.1.htmlCSV_FORMAT. Accessed 24 Aug. 2022.
[18] “Useful Nvidia-Smi Queries — NVIDIA.” Useful Nvidia-Smi Queries — NVIDIA, nvidia.custhelp.com, 29 Sept. 2021, https://nvidia.custhelp.com/app/answers/detail/a_id/3751/ /useful-nvidia-smi-queries.
[19] “1.How to Access the /Proc-Filesystem.” 1.How to Access the /Proc-Filesystem, tldp.org, https://tldp.org/HOWTO/Linux+IPv6-HOWTO/ch11s01.html. Accessed 24 Aug. 2022.
[20] “User Guide: Nsight Systems Documentation.” User Guide: Nsight Systems Documentation, docs.nvidia.com, https://docs.nvidia.com/nsight-systems/UserGuide/index.html. Accessed 24 Aug. 2022.
[21] “Pandas.” Pandas, https://pandas.pydata.org/.
[22] “Pickle - Python Object Serialization¶.” Pickle - Python Object Serialization - Python 3.10.5 Documentation, https://docs.python.org/3/library/pickle.html.
[23] “Argonne Leadership Computing Facility.” Theta/ThetaGPU Machine Overview | Argonne Leadership Computing Facility, https://www.alcf.anl.gov/support-center/theta/theta-thetagpu-overview.
[24] “ANL-CESAR/XSBench.” GitHub, 4 June 2022, github.com/ANL-CESAR/XSBench. Accessed 15 July 2022.
[25] “ANL-CESAR/RSBench.” GitHub, 17 Feb. 2022, github.com/ANL-CESAR/RSBench. Accessed 15 July 2022.
[26] Argonne-Lcf. “Argonne-LCF/Pythonfoam: In-Situ Data Analyses and Machine Learning with Openfoam and Python.” GitHub, Argonne LCF, https://github.com/argonne-lcf/PythonFOAM.
[27] “OpenFOAM | Free CFD Software | The OpenFOAM Foundation”. OpenFOAM.org, https://openfoam.org/
[28] Maulik, Romit, et al. ”PythonFOAM: In-situ data analyses with OpenFOAM and Python.” Journal of Computational Science (2022): 101750.
[29] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Rafal Jozefowicz, Yangqing Jia, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Mike Schuster, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.