

Parallel Performance Wizard: A Generalized Performance Analysis Tool
Hung-Hsun Su, Max Billingsley III, Seth Koehler, John Curreri, Alan D. George
University of Florida

Scientific programmers face many challenges in optimizing the total time-to-solution of their computing problems, as increasing architectural complexity and growing degrees of parallelism lead to programs that do not yield an expected performance level. To combat this problem, programmers use an iterative performance optimization process facilitated by performance analysis tools, which alleviate the work required to determine causes of performance degradation. Many such tools currently exist, but they generally support a limited selection of parallel programming models by way of tight coupling with the individual models. A drawback of this tight coupling is that it makes it difficult to extend these tools to support additional models. In this poster, we present the extension of Parallel Performance Wizard (PPW), a tool originally designed for partitioned global-address-space (PGAS) models such as Unified Parallel C (UPC), Co-Array Fortran, Titanium, and SHMEM, to employ a framework based on generic operation types that makes it possible to add support for additional models with minimal effort. To show the effectiveness of this approach, we illustrate how support for the Message Passing Interface (MPI) was quickly added to PPW. Finally, we discuss how PPW's framework can be employed to support non-traditional paradigms such as Reconfigurable Computing (RC), where the software application makes use of hardware resources such as Field Programmable Gate Arrays (FPGAs) to achieve additional speedup in solving challenging computational problems.

PPW was designed to maximize user productivity as a full-featured performance analysis tool supporting PGAS programming models. PGAS model support was facilitated by the Global Address Space Performance (GASP) interface, which specifies the interaction between the user program, compiler and the performance analysis tool. Using data gathered from GASP, PPW provides visualizations and semi-automatic analyses that facilitate the optimization process. Visualizations include a profile table and call-tree diagram, which show statistical data giving a high-level overview of program performance, a timeline view of trace data via export to the Jumpshot viewer, and viewers for shared-memory distribution and communication volume (Figure 1) that aid in understanding the one-sided communication patterns that typify PGAS applications. PPW also includes mechanisms for automatic detection of possible performance bottlenecks and in some cases can provide practical hints on how to remove these bottlenecks.

The framework of PPW has been modified to work with generic operation types, such as all-to-all, one-sided data get, two-sided send, etc., to facilitate adding support for new programming models quickly. This new framework is depicted in Figure 2. For each supported model, a mapping between model-specific constructs and their corresponding operation types (e.g., from `upc_memget` to one-sided get, from `mpi_send` to two-sided send, etc.) is first specified. Given this classification, PPW is structured to work primarily with the defined generic operation types and only consults the mapping when model-specific information is needed. PPW also employs a new bottleneck detection and resolution model to support semi-automatic analysis based on the generic operation types. With these improvements made to the framework, we have significantly reduced the effort needed to support additional programming models, as many tool components do not require modification in the process. In most cases, one can follow the following steps to add support for a new model:

1. Provide a mapping from the model's constructs to the generic operation types.
2. Enable automatic instrumentation of applications written using the model, via an implementation of GASP.
3. Make any small changes to the bottleneck resolution components needed for analysis.

By following these steps, we were able to implement MPI support for PPW in a few months, while developing a new tool would have been a multi-year effort. Figure 3 shows a screenshot of the profile table visualization for an MPI ray-tracing application.

PPW's framework also forms an effective basis for supporting performance analysis of RC applications employing both traditional CPUs and FPGAs. By extending PPW's generic types to include operations such as FPGA initialization and communication, the activity and status of the application executing on the FPGA can be captured. The process for adding support to PPW for RC includes the same three steps listed above, but with some additional complications, such as non-standard APIs for accessing FPGAs, difficulties in expressing FPGA computation by using a standard function call-trace model, and the addition of a Hardware Measurement Module (HMM) during instrumentation. However, with the introduction of hardware capabilities, PPW's framework can be employed to monitor performance not only across programming models, but across the hardware/software boundary itself. Figure 4 shows a high-level view of hardware-related components integrated into PPW.

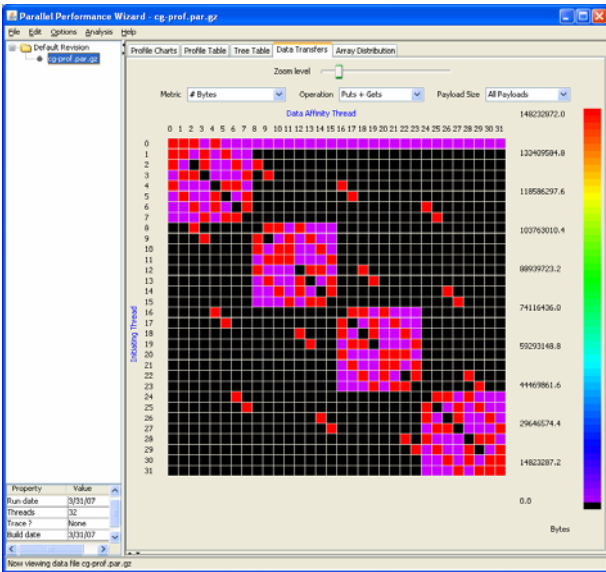


Figure 1: PPW communication volume viewer. This visualization graphically illustrates the communication pattern from the UPC implementation of the NAS CG benchmark (v2.4).

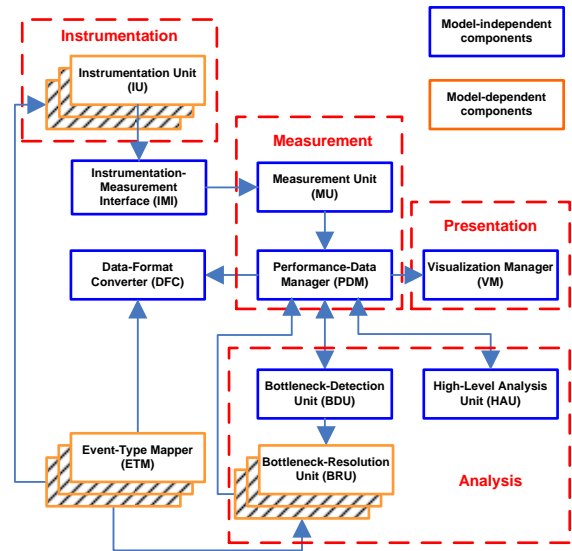


Figure 2: PPW infrastructure. This diagram depicts the updated infrastructure of PPW that operates on generic operation types

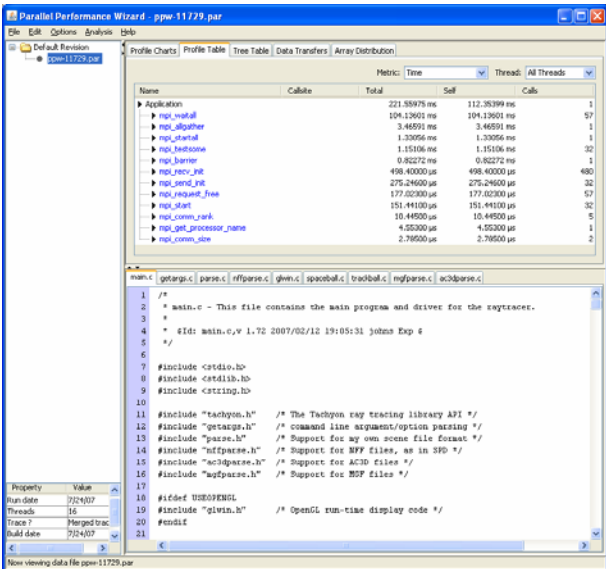


Figure 3: PPW call-tree diagram. This display shows profile data alongside the original source code for an MPI ray-tracing application.

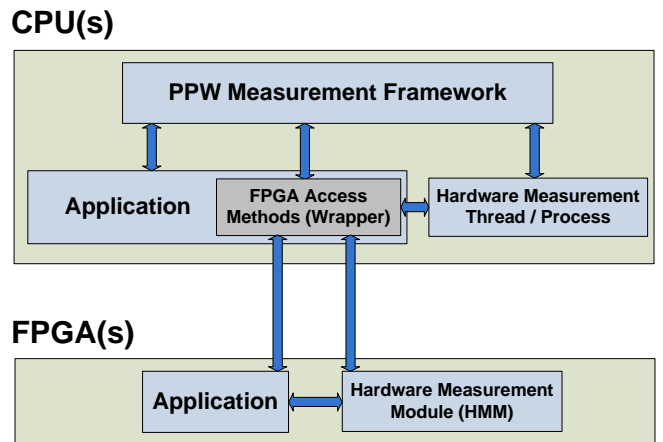


Figure 4: PPW/RC framework. This diagram shows the connections between FPGA-related components and the PPW measurement framework.