Timemory: Modular Performance Analysis for HPC

Jonathan R. Madsen[1(✉)], Muaaz G. Awan[1], Hugo Brunie[1], Jack Deslippe[1], Rahul Gayatri[1], Leonid Oliker[2], Yunsong Wang[1], Charlene Yang[1], and Samuel Williams[2]

[1] NERSC, Lawrence Berkeley National Laboratory, Berkeley, CA, USA
jrmadsen@lbl.gov
[2] CRD, Lawrence Berkeley National Laboratory, Berkeley, CA, USA

Abstract. HPC has undergone a significant transition toward heterogeneous architectures. This transition has introduced several issues in code migration to support multiple frameworks for targeting the various architectures. In order to cope with these challenges, projects such as Kokkos and LLVM create abstractions which map a generic front-end API to the backend that supports the targeted architecture. This paper presents a complementary framework for performance measurement and analysis. Several performance measurement and analysis tools in existence provide their capabilities through various methods but the common theme among these tools are prohibitive limitations in terms of user-level extensions. For this reason, software developers commonly have to learn multiple tools and valuable analysis methods, such as the roofline model, are frequently required to be generated manually. The timemory framework provides complete modularity for performance measurement and analysis and eliminates all restrictions on user-level extensions. The timemory framework also provides a highly-efficient and intuitive method for handling multiple tools/measurements (*i.e.,* "components") concurrently. The intersection of these characteristics provide ample evidence that timemory can serve as the common interface for existing performance measurement and analysis tools. Timemory components are developed in C++ but includes multi-language support for C, Fortran, and Python codes. Numerous components are provided by the library itself – including, but not limited to, timers, memory usage, hardware counters, and FLOP and instruction roofline models. Additionally, analysis of the intrinsic overhead demonstrates superior performance in comparison with popular tools.

Keywords: C · C++ · Python · CUDA · Fortran · Instrumentation · Timing · Memory · Performance · Cross-platform · Measurement · Cross-language · Analysis · Hardware-counters · Roofline · MPI · CUPTI · PAPI · Gotcha · UPC++ · gperftools · TAU · Caliper · LIKWID · Score-P · VTune · NVTX · ERT · Timemory

© Springer Nature Switzerland AG 2020
P. Sadayappan et al. (Eds.): ISC High Performance 2020, LNCS 12151, pp. 434–452, 2020.
https://doi.org/10.1007/978-3-030-50743-5_22

1 Introduction

A straightforward modular system for user-defined performance measurements and analysis is notably absent from the vast ecosystem of specialized and generic tools for sophisticated performance measurements and reflective analysis. The modular compiler infrastructure provided by LLVM [18] is an excellent example of the benefits of modularity and has resulted in the development of a number of tools filling various generic and specialized needs [19]. The programming model abstractions provided by Kokkos [7] is an excellent example of using C++ templates to provide a generic and flexible front-end that adapts to the targeted architecture at compile-time. Timemory attempts to provide the analogue to the LLVM infrastructure and Kokkos model in the realm of performance measurement and analysis. The framework provides a viable solution to a common instrumentation interface [5] for multiplexing performance measurement and analysis tools. As a common instrumentation interface, timemory would provide a straightforward method for projects with existing instrumentation APIs to locally[1] wrap their existing API and introduce a significant number of new capabilities to the existing tool[2] while requiring no significant changes to the tool itself. Projects that adopt the timemory framework gain the capability to arbitrarily define multiple bundles of performance measurement and analysis tools to the need of the project and can customize the activation or deactivation of these tools in any manner desired. This paper will outline the current state of performance tools, highlight several key innovations developed in timemory, and then provide examples which demonstrate how these innovations have enabled an extensive suite of tools and capabilities.

The timemory library is written in C++14 using template metaprogramming, is presently available for codes written in C, C++, Python, and Fortran, and supports interoperability with CUDA, MPI, UPC++, and various forms of multi-threading. Overall, the contributions through timemory include:

- Common performance measurement and analysis framework with full support for user-level extensions
- Common framework for: generation of custom event-based, statistical, and/or instrumentation profilers, custom preload libraries, and tool multiplexing
- Type-safe method for arbitrarily wrapping existing tools which can store data in any valid C++ data type
- Highly-efficient instrumentation API with almost negligible overhead when disabled at runtime
- Static and dynamic generation of arbitrary component bundles
- Intermixed call-stack tracing, timeline tracing, and flat-profiling
- Intermixed usage of different tool bundles[3].

[1] *i.e.,* within the existing project's code and without any required changes upstream to timemory.

[2] Cross-language support, JSON/XML/text output, call-graphs, statistical analysis, plotting, sampling, MPI support, UPC++ support, multi-threading support.

[3] *e.g.,* Bundle of A, B, and C can be used alongside bundle of A, C, and D and/or bundle of E, F, and G.

2 Motivation

2.1 Need for Composite Components

A variety of performance measurement and analysis tools co-exist in the HPC ecosystem. Well known examples include TAU [26], Caliper [6], HPCToolkit [1], and LIKWID [28]. Each one of these tools provide their capabilities via design abstractions around the lower-level interfaces for the hardware and generally build upon the work of more specialized libraries such as PAPI [27], CUPTI [9], and Linux perf [11]. However, each tool tends to have a special set of features in order to provide a unique draw and use case scenario. The special set of feature(s) provided by the tools form a complementary set of capabilities with other tools which make them worthwhile to use in combination, however in order to provide these features, there is commonly a redundancy in basic functionality[4] [17]. The most disparate properties among these tools is the data storage model, control methods, and input/output schema. The data storage model is influenced heavily by the design of the library and very few libraries directly expose methods for accessing the raw data handled by the library. The plausible culprit for the commonality of obscuring the data storage model is the type-obfuscation that arises from either the common C-style generic design patterns, which commonly restrict supported data types to those listed in an enumeration, and the C++-style generic design pattern of dynamic polymorphism, which requires non-templated types for virtual functions. Thus, providing access to the data model is not only prone to complexity and lack of type-safety but it may also have to be completely re-factored to support new features which necessitate adding explicit support for new data types.

The timemory library presents an unique solution to these challenges. Through the use of C++ template meta-programming, a package can expose any number of unique C++ classes that encapsulate a performance measurement or analysis pattern. The C++ classes have only one core requirement: a public type declaration of the `value_type` used by the component, which can be any valid C++ data type, including `void`. In timemory, only the names for the functions are required to be consistent and there are no restrictions on the data types that non-void functions return. Thus, one component can implement the data access member function `get()` to return a floating-point value and another component can implement this member function to return an array of integers. Once this minimal set of requirements is provided, the component can be bundled alongside any number of other components into a single handle. Various other capabilities/features can be activated within a component simply through implementing the corresponding member function inside the component. These member functions are optional due to extensive use of SFINAE and empty base-class implementations of these functions. Additionally, components

[4] Support for various parallelization models, data acquisition techniques (instrumentation, sampling, etc.), and injection techniques (symbol overloading, binary modification, etc.).

can be designed as composites of other components. This building-block characteristic is unique to the framework and strengthens the argument for timemory as the universal interface for performance measurement and analysis.

2.2 Need for Common Instrumentation Interface

Numerous tools provide instrumentation APIs that are directly inserted into the application source code. The instrumentation APIs for many tools provide the capability to enable/disable a tool when connected, provide context labels for code regions, and track simple event metrics. Common examples include the ittnotify [8] API for Intel's VTune Amplifier and Advisor, NVTX for NVIDIA's Nsight and NVprof, gperftools, LIKWID, TAU, and Score-P [17]. Some of these tools center their usage around a command-line tool while other tools, such as the Caliper package, focus their usage around instrumentation markers.

The potential for performance degradation via instrumentation APIs, even when dormant at runtime, is supported by the results of applying an edge-case scenario of injecting 500,000 runtime-disabled instrumentation points within a matrix multiplication benchmark (Fig. 1) to Caliper, TAU, and timemory [5]. Unlike statistical profilers which take measurements at a given rate, the overhead of deterministic instrumentation cannot be fully negated and the overhead associated with the instrumentation is subject to high variability: Caliper markers increased run time by ~397% while TAU markers increased run time by ~262%. These overheads stand in stark contrast to the methods provided by timemory, which increased the runtime by a minimum of ~42% and a maximum of ~82%. The primary objective of timemory is not to serve as a replacement for Caliper, TAU, etc. but, instead, provide a common, easily extendable interface

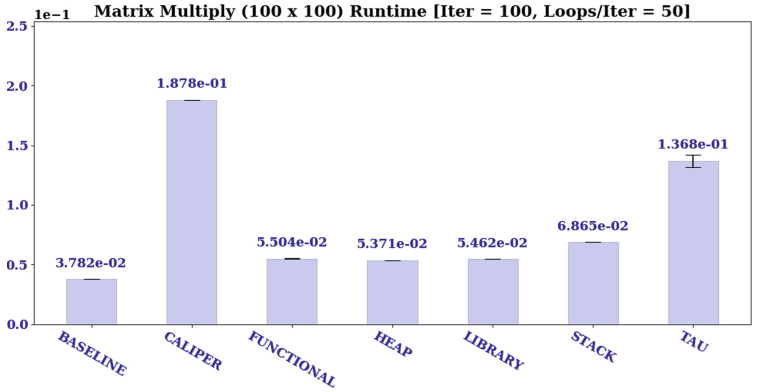

Fig. 1. Average (samples = 100) runtime of 500,000 dormant instrumentations for 100×100 Matrix-Multiply Calculation. `BASELINE` is without instrumentation, `CALIPER` is with Caliper instrumentation, `TAU` is with TAU instrumentation, and remaining data points (`FUNCTIONAL`, `HEAP`, `LIBRARY`, `STACK`) use different models of timemory instrumentation, where each model has different compile-time and runtime capabilities.

for the deployment of performance analysis tools which is optimized for minimal overhead when not being utilized. With concerns about unintentional overhead minimized, HPC developers can safely provide built-in performance monitoring which can deploy whichever performance tool(s) are available for a given architecture. Significant progress towards this objective has been achieved: at present, timemory provides one or more components for ARM-MAP, Caliper, TAU, LIKWID, CrayPAT, Intel VTune, Intel Advisor, gperftools, CUDA, NVTX, CUPTI, and PAPI.

2.3 Need for Object-Level Analysis Granularity

Profiling tools generally support one or more granularities for reporting performance measurements: functions, addresses, lines, and files, in descending order of commonality. However, object-oriented programming is a widely utilized paradigm in HPC and is supported by C++, Fortran, and Python. The tasking and object-oriented model presents a challenge for performance measurements frameworks using a procedural design. Since the lifetime of objects typically overlap, these designs struggle to distinguish measurements from different objects when only the function, file, and line metadata is available. In other words, object-oriented codes violate the LIFO model of function call-stacks that these frameworks might rely upon. Furthermore, providing measurements and analysis for an object introduces a configuration issue for the tool when objects derive from abstract objects because the tool (ideally) should support the user coalescing the data at an arbitrary abstraction granularity of their choosing.

The Geant4 toolkit [2] – a Monte Carlo particle transport toolkit for the simulation of the passage of particles through matter[5] – provides an excellent example of the need for a new performance analysis model that tracks measurements at object-level granularity. The Geant4 toolkit is written in C++ and makes extensive use of dynamic polymorphism in ∼1 million lines of code. This code supports 125+ derived particle types, 550+ derived physics processes, and 1000+ derived process models. Each particle type is subject to a unique set of stochastic physics process model pairs whose probabilities for interaction and secondary particle generation vary tremendously across the spectrum of particle energy and target material. From a performance analysis standpoint, this creates a challenging task for determining improvable "hotspots" and traditional performance analysis fails because the Geant4 execution model does not have any core "hotspot" routines at function-level granularity.

Timemory proposes that in order to provide object tracing measurements and customization of the abstraction-level[6], the analysis tool itself should provide instrumentation *objects* which locally store intermediate data instead of instrumentation points which invoke global functions or pseudo-instrumentation objects which couple the global function invocations to RAII. With this intermediate storage design, these instrumentation objects can be inserted into the

[5] *i.e.,* Radiation shielding, particle accelerator simulations, nuclear reactor design.
[6] *i.e.,* Ability to associate measurements with either the derived or abstract object.

target object itself at the desired abstraction-level, be treated by the application as just another member variable, increase data locality for measurements, and support asynchronous paradigms. Timemory also proposes that a well-designed framework adhering to these principles should provide multiple variants of these instrumentation objects which (A) utilize RAII to easily couple of the measurement scope to the scope of the target object and (B) permit the insertion or activation of different analysis types during compilation and/or runtime.

3 Library Design

The timemory library is implemented in C++14 with the curiously recurring template pattern (CRTP) style and was designed from the outset to:

- Allow for user-level implementations of tools (also called "components")
- Allow for components to store measurements in an arbitrary data type
- Allow for arbitrary bundling of tools into a single handle
- Fully support modularity
- Utilize thread-local memory to minimize synchronization bottlenecks
- Strictly avoid spawning background work in library core
- Minimize any runtime logic which can be evaluated at compile-time
- Minimize overhead when enabled at compile-time but disabled at runtime
- Provide an easy-to-use interface.

A sample of the basic design of timemory in C++ is demonstrated in Listing 1.1.

Listing 1.1. Sample Usage in C++ of bundle of tools combining: wall-clock timer, peak memory measurement, and various markers for external tools which are removed at compile-time when not available.

```
#include <timemory/timemory.hpp>
using namespace tim::component;
using markers_t = type_list<nvtx_marker, likwid_marker, tau_marker>;
using tools_t = tim::component_tuple<wall_clock, peak_rss, markers_t>;

void foo() {
    tools_t obj("foo");     // create marker
    obj.start();            // start all components
    sleep(1);               // sleep for 1 second
    obj.stop();             // stop all components
    // access specific component
    wall_clock* wc      = obj.get<wall_clock>();
    double      elapsed = wc->get();    // computed value
    std::string unit    = wall_clock::display_unit();
    // Output: "Wall time: 1.000 sec"
    printf("Wall time: %f %s\n", elapsed, unit.c_str());
}
```

3.1 Components

Timemory uses the term "component" to refer to a single structure that provides a certain functionality in the form of a "caliper" (*i.e.,* a region enclosed by a `start` and `stop`). The definition of a component is straightforward and a sample is provided in Listing 1.2. In general, a component inherits from a templated base class and specifies itself as the first template parameter and the data type that the component will be using to store the metric (if any). The data type can be any valid C++ data type, *e.g.,* `int`, `double`, `vector<MyClass>`, etc. Components that accumulate no internal data, such as a component that just forwards the marker labels to another tool, can designate the data type as `void`.

Listing 1.2. Sample component in timemory. The macros `TIMEMORY_<XYZ>` are used for type declarations and setting type-traits which activate various features for the type, *e.g.,* statistics, unit conversion support, category-specific formatting, etc.

```
1  TIMEMORY_DECLARE_COMPONENT(wall_clock)
2  TIMEMORY_CONCRETE_TRAIT(uses_timing_units, wall_clock, true_type)
3  TIMEMORY_CONCRETE_TRAIT(is_timing_category, wall_clock, true_type)
4  TIMEMORY_STATISTICS_TYPE(wall_clock, double)
5
6  struct wall_clock : public base<wall_clock, int64_t>
7  {
8      static string     label()       { return "wall"; }
9      static string     description() { return "wall-clock timer"; }
10     static value_type record()      { return get_time_now(); }
11     // 'value' and 'accum' are inherited int64_t
12     void   start()  { value = record(); }
13     void   stop()   { value = (record() - value); accum += value; }
14     // 'get_units()' is base-class func controlled via type-traits
15     double get() const         { return accum * get_units(); }
16 };
```

Several type-traits are provided to customize functionality, provide default output formatting, unit support and conversions, etc. The details of the various type-traits are beyond the scope of this paper with the exception of the most important type-trait with respect to portability: `is_available`. This type-trait creates a template meta-programming system through which a type can be forward declared, and thus be portably declared as a component in a bundle, but filtered out entirely from the template specification before the type is instantiated when `is_available` evaluates to false. Thus, `tuple<A, B, C>` will be implicitly implemented as `tuple<A, B>` if component `C` is not available. The timemory-provided components which rely on external packages use the absence of the package-specific pre-processor definition (*e.g.,* `TIMEMORY_USE_PAPI`) to set the `is_available` type-trait to false. In addition to the portability benefits, this feature also allows timemory to minimally function as a header-only library for C++ codes.

3.2 Data Storage

The data storage for each component type is handled dynamically via storage class singletons that are templated on the component type. Each storage class singleton maintains a unique call-graph per-thread (see Fig. 2) for components which store data. This call-graph handles the accumulation of data throughout the application and supports arbitrarily mixing hierarchical, timeline, and flat node insertion modes. This approach also enables an arbitrary number of components to operate independently by eliminating the need for fixed array limits on the number of tools that can concurrently allocate storage space. Furthermore, since the component-specific storage is templated on the component type, the data storage model ensures complete type-safety.

3.3 Parallelism Support

The timemory framework supports both MPI and UPC++ for distributed memory parallelism and neither backend imposes any communication overhead during the application execution outside of the one-time communication to the zeroth rank during finalization and output. Within a process, timemory makes careful use of static and thread-local static storage singletons to provide an efficient model for multi-threading which is highly scalable for HPC. The data storage model is entirely free from the use of synchronization primitives (*i.e.,* locks) outside of the construction and destruction of the storage singleton on a worker

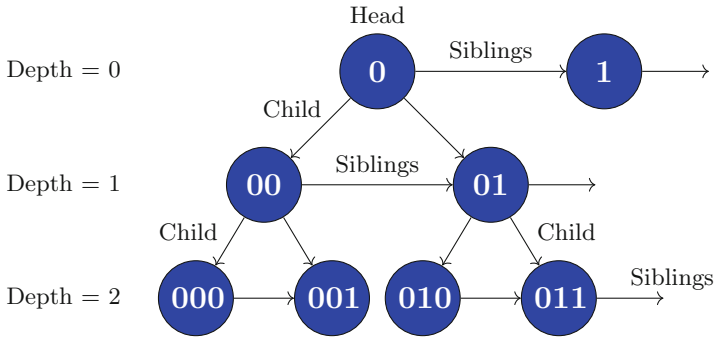

Fig. 2. Call-graph per component. Each node is keyed to a label (*e.g.,* function name, file, and line number) and contains an instance of the component. The component instance within the call-graph provides data-storage only. When a new component instance is created and assigned a label, the component searches the children of the current node for a matching key. If no matching key is found, the component creates a new node. The address of the node is stored internally in the component and when the component instance is stopped, the instances adds its data to the component at that node address and resets it's internal data to zero. Thus, temporary component instances are fully responsible for finding and creating new nodes for persistent storage and updating those nodes.

thread. During the construction of the storage singleton on a worker thread, the master instance is locked to ensure the worker-thread can safely bookmark (perform a copy) the current instrumentation stack location. Beyond this point, no synchronization is performed until the worker thread terminates and cleans up the thread-local memory. At this point, the manager thread is locked and the instrumentation stack from the worker thread is inserted as a child of the bookmarked location on the master thread.

3.4 Bundling Components

Timemory provides variadic template wrappers that allow multiple components to be bundled together into a single handle whose member functions correspond to the invocation of the similarly named member function for each component. The variadic template wrappers rely on the temporary construction of operation classes which are templated per-component (see Listing 1.3). These operation classes are the key to the flexibility of timemory. These classes provide both the ability to specialize the behavior of a component in a multiplexing scenario (see Listing 1.4) and provide a generic interface for calling similarly named member functions with different signatures per component through the use of SFINAE (see Listing 1.3). The instantiation and translation of these concepts for a generic variadic wrapper (Listing 1.5) is demonstrated in Listing 1.6.

Listing 1.3. Sample `foo` operation struct that is templated on a component. SFINAE is used to determine desired call signature at compile-time and `int` and `long` are used to control overload resolution. The first `bar` function checks if T has `foo()` member function that accepts the given arguments. If this check fails, then the `foo()` member function is called without arguments. This methodology can be easily extended to a third option that does not call the member function at all.

```
template <typename T>
struct foo {
    foo(T obj, args...) { bar(obj, 0, args...);  }

    auto bar(T obj, int, args...) -> decltype(obj.foo(args...), void())
    { obj.foo(args...);  }

    void bar(T obj, long, ...) { obj.foo();  }
};
```

Listing 1.4. Sample specialization of `foo` operation struct from Listing 1.3 for component **A** where it is known that **A** does not accept arguments and has `foo()` member function

```
template <> struct foo<A> { foo(T obj, ...) { obj.foo(); } };
```

Listing 1.5. Sample of internals from generic variadic component wrapper (`template <typename... T> struct component_tuple`) combining concepts from Listing 1.2 and Listing 1.3. Template parameters are omitted for readability.

```
// generic data type held by component_tuple<T...>
tuple<T...> m_data;
// generic foo member function for component_tuple<T...>
void foo(args...) { apply<operation::foo<T>...>(m_data, args...); }
    };
```

Listing 1.6. Sample of internals from generic variadic component wrapper `component_tuple<A, B>` when Listing 1.5 is instantiated with types A and B. Template parameters are omitted for readability.

```
// data type held by component_tuple<A, B>
tuple<A, B> m_data;
// foo member function for component_tuple<A, B> after instantiation
void foo(args...) {
    operation::foo<A>(get<0>(m_data), args...);
    operation::foo<B>(get<1>(m_data), args...); }
};
```

The variadic wrappers are provided in numerous flavors for compile-time and runtime configuration via various type-traits, configuration bundles, callbacks, and custom environment variables. These various methods are provided to empower projects to build in custom schemes for utilizing their bundles which conform to the standard configuration methods of the project itself. Thus, the timemory framework facilitates the generation of easy-to-use built-in performance diagnostic tools that can be quickly switched on by developers and users when performance analysis is either desired or required.

4 Profiling Capabilities

Profilers generally fall into two broad categories: statistical profilers which operate via sampling and instrumentation profilers. Instrumentation profilers effectively inject additional instructions into the binary and are implemented through several methods: manually, automatic source-level (tool that modifies source-code), compiler-assisted, binary translation (tool that modifies compiled binary), runtime instrumentation (tool that supervises and controls execution after temporarily injecting instrumentation), and runtime injection (a lightweight form of runtime instrumentation that instruments jumps to helpers functions). At present, timemory supports manual instrumentation, runtime instrumentation for dynamically-linked binaries via Gotcha [23], binary translation, runtime instrumentation, and a simple command-line execution wrapper similar to the UNIX command-line tool `time` except with extensions to include memory and I/O values and rates and hardware counters. Additionally, timemory distributes

a number of "instrumentation libraries" which provide simple function interfaces for activating instrumentation around performance monitoring APIs exposed by several commonly-used APIs, *e.g.,* Kokkos, MPI, and OpenMP. These instrumentation libraries can be directly inserted into the application codes or injected externally via binary translation or runtime instrumentation. For Python codes, the timemory package supports context-managers and decorators for instrumenting specific functions and regions of code and can also leverage the built-in debugging and profiling capabilities of the interpreter.

Dynamic Instrumentation. Timemory provides a command-line tool, `timemory-run`, for runtime instrumentation and binary translation of dynamically- and statically-linked binaries via the Dyninst [4] toolkit. The command-line tool combines a number of features derived from various positive experiences with existing profiling tools. These features include: using regular expressions (regex) and/or text files for precise selection of which modules and functions to instrument (inclusive, exclusive, and inclusive/exclusive unions), lightweight stub instrumentation during binary translation for `LD_PRELOAD`, loop instrumentation, defining the default set of components during binary translation, insertion of user-defined functions from custom instrumentation libraries, and two different modes which offer a choice between whether the dynamic instrumentation is affected by manual timemory instrumentation with the C/C++/Fortran library interface. With respect to these two different "modes" of instrumentation, an application using manual timemory instrumentation may be precisely configured at a high-level to collect different components in different regions of the code and dynamic instrumentation may be deployed for fine-grained analysis. In one scenario, a user may want to keep these precise configurations intact as a reference point for the overhead of the fine-grained analysis. In another scenario, the user may want to propagate these precise configurations to the dynamic instrumentation. The aforementioned "modes" of instrumentation address these two scenarios. In one mode, the set of components collected by the dynamic instrumentation points are configurable via its own distinct environment variable and unaffected by changes to the component collection set via the library interface. In the second mode, instrumentation is synchronized with the manual instrumentation: both the manual and dynamic instrumentation are configurable with the same environment variable and modifications to the instrumentation component set via the library interface are applied to the dynamic instrumentation.

Statistical Profiling. At present, timemory does not provide an API for the generation of performance measurements via sampling on par with the facilities for instrumentation. However, the need for this capability was factored into the design of the library and is currently being deployed in the `timem` execution wrapper. This command-line tool is similar to the UNIX `time` command except it extends the measurement set beyond timers to include resource usage and hardware counter measurements. This command-line tool uses a fork + execv model

and thus, in order to post-process and produce output, only the parent process can invoke start and stop on the component bundle since the child process never returns. Although this model does not present issues for numerous components that either inherently include or are configurable to include activity within child processes, certain components such as those which read from Linux process ID files (*e.g.,* `/proc/<PID>/statm`) or hardware counters must record measurements at or very near the end of the child process but before the child process exits and execution on the parent process resumes (where stop on the component bundle in the parent process is called). Thus, when building this command-line tool, the components with this criteria must be customized to sample their value(s) during an interrupt and measurements during the stop operation must be either be discarded or the operation itself should not be invoked. Through the use of a local specialization on the corresponding operation classes introduced in Sect. 3.4, this is easily accomplished: `operation::start<T>` and `operation::stop<T>` for any component `T` requiring sampling is locally specialized so that the start and stop member functions of an instance of `T` are never invoked when start or stop is invoked on the component bundle and `operation::sample<T>` is specialized for these components to update their values accordingly. The success of this model for the `timem` executable will likely serve as a template for the creation of independent sampling libraries which can be inserted into applications directly and/or through the dynamic instrumentation command-line tool.

Gotcha Support. The timemory library simplifies using Gotcha for re-writing the Global Offset Table on the Linux operating system that links inter-library call-sites and variable references to their targets. In general, a set of components for performance measurement or analysis can be injected around any externally linked function in as little as 2–3 lines of code plus one line for each function to be wrapped.

Listing 1.7 demonstrates a hypothetical timemory Gotcha implementation which wraps a wall-clock timer around the C `exp(double)` function and a C++ function, `sum_exp`, which takes an array of floating-point values and accumulates the result of calling `exp` in each value. Thus, invocation of the `sum_exp` function with two floating-point values results in a nested hierarchy of one wall-clock measurement around `sum_exp` at depth 0 and two wall-clock measurements around `exp` as children of `sum_exp` in the call-graph (see Listing 1.8).

Listing 1.7. Sample Gotcha specification around two external dynamically-linked functions: `exp` and `sum_exp`

```
1  using wc_t  = component_tuple<wall_clock>;
2  using got_t = gotcha<2, wc_t>;
3
4  extern "C" double exp(double);
5  double sum_exp(vector<double>);
6
7  int main() {
```

```
  got_t::get_initializer() = []()
  { TIMEMORY_C_GOTCHA  (got_t, 0, exp);
    TIMEMORY_CXX_GOTCHA(got_t, 1, sum_exp); };

  auto_tuple<got_t> obj("example");
  auto ret = sum_exp({ 1.0, 2.0});
}
```

Listing 1.8. Abbreviated output for Listing 1.7

```
|   LABEL     |  COUNT   |  DEPTH   | METRIC   |  UNITS   |   SUM    |
|-------------|----------|----------|----------|----------|----------|
| >>> sum_exp |        1 |        0 | wall     | msec     |    0.072 |
| >>> |_exp   |        2 |        1 | wall     | msec     |    0.043 |
```

In addition to instrumenting functions, the timemory Gotcha component can be used to provide wholesale function replacement of the Gotcha wrappee when (1) a third template parameter is provided, (2) the third template parameter is a timemory component, and (3) the timemory component provided as the third template parameter has an overloaded function operator (`operator()`) whose return type and arguments match the function being wrapped, *e.g.,* to replace `double exp(double)`, the timemory component provided as the third template parameter must provide `double operator()(double)`. Thus, not only can the timemory Gotcha component be utilized to instrument external function calls but it can also be utilized to provide wholesale replacement of external function calls for optimization, as illustrated in Sect. 5.2. Finally, similar to the `operator()` overloading scheme, components which are instrumenting functions instead of replacing them can provide `void audit(Args...)` member functions where `Args...` matches the function parameter types of the original function and/or the return type of the original function to gain access to the values of the input parameters before the original function is invoked and the return value of the original function before it returns.[7]

Instrumentation Libraries. Timemory distributes several stand-alone libraries which can be utilized to activate instrumentation around APIs which provide their own performance monitoring framework, *e.g.,* Kokkos, MPI, and OpenMP. With respect to Kokkos, timemory generates one traditional profiling library whose selection of components is configurable via environment variables at runtime and then over a dozen of pre-configured profiling libraries with dedicated functionality, *e.g.,* `kp_timemory_trip_count.so` is explicitly configured to collect trip-counts, `kp_timemory_cpu_flops.so` is explicitly configured to count floating-point operations, etc. Concerning OpenMP, timemory distributes a library that provides instrumentation via the OMPT [13] call-back system.

[7] Users can also alternatively provide `void audit(string, Args...)` if the (demangled) name of the function is required.

For MPI, timemory distributes a library which leverages it's Gotcha capabilities to wrap the equivalent of the PMPI [22] interface without breaking any existing user-defined MPI functions using the PMPI interface. Additionally, both the OpenMP and MPI instrumentation libraries provide reference counting modes to enable scoped instrumentation. Although these libraries will satisfy the needs of the vast majority of use cases, we would like to note that the implementation of these instrumentation libraries is straight-forward and the MPI and OpenMP instrumentation libraries require less than 100 lines of code – with minimal effort these libraries can be customized to include user-defined components which, when paired with the Gotcha method to wrap the targeted function call, can produce instrumentation libraries which are capable of replacing the original function call or analyzing the input parameters and return values of the function call and then inserted into the binary via the dynamic instrumentation tool.

5 Novel Use Cases

5.1 Performance Measurements and Analysis in Geant4

Section 3.4 introduced the concept of using timemory to build an extensible, built-in performance measurement and analysis framework that conforms to the design of the project. This concept was put into practice within the Geant4 toolkit, whose description was provided in Sect. 2.3.

The Geant4 source code implements a G4TiMemory header file which provides empty macro replacements when Geant4 is configured without timemory support. When Geant4 is configured with timemory support, Geant4 takes advantage of the pre-defined `tim::auto_timer` bundle to instrument always-on high-level measurements around approximately two dozen core routines. To provide user-customizable performance analysis in low-level functions invoked at a high frequencies, Geant4 defines a `G4Profiler` class templated on the value of the profiler type enumeration and a variadic list of types that form an instrumentation context (see Listing 1.9). Using this scheme, each instrumentation instance can arbitrarily adapt to the runtime data analyzed in the callbacks and selectively: enable/disable the instrumentation, customize the label, and add/remove components.

Listing 1.9. Geant4 Profiler Definitions for timemory. The `query`, `label`, and `tweak` functions apply their arguments to call-backs provided by the user-application. `G4ProfilerBundle` is an alias to the timemory `user_bundle` component which provides an interface for manipulating an array of components during runtime.

```
template <size_t Category, typename... Types>
class G4Profiler {
    using type    = tim::auto_tuple<G4ProfilerBundle<Category>>;
    static bool   query(Types...);
    static string label(Types...);
    static type&  tweak(type&, Types...);
};
```

Listing 1.10. Hypothetical User Configuration of G4TrackProfiler which only instruments Electrons, customizes the label to reflect the physical volume of track, and defaults to wall-clock and thread-specific cpu-clock timers for instrumentation unless the electron energy is below 100 keV, at which point the API supplements the instrumentation to include data collection for the classical roofline plot on the CPU. Data types abbreviated for readability, assume all code is specifically applied to the G4TrackProfiler.

```
get_query()   = [](G4Track* t) { return t->GetType() == Electron; };
get_labeler() = [](G4Track* t) { return t->GetVolumeName(); };
get_tweak()   = [](auto& p, G4Track* t) {
    if(t->GetEnergy() < 100.*keV) { p.insert<cpu_roofline_flops>(); }
    return p;
configure<wall_clock, thread_cpu_clock>();
};
```

5.2 Mixed-Precision Analysis

Floating-point arithmetic [15,16] is ubiquitous in High Performance Computing applications and it is the source of numerical bugs [10]. Due to the complexity of understanding the impact of floating-point arithmetic on result accuracy, many applications are written entirely with double precision despite the growing gap between half, single, and double precision performance [21].

The precision required in different phases of an application to achieve the desired precision in the result remains an open question. One of the projects at NERSC consists of developing a systematic approach to optimize scientific applications using multiple precisions for calls to mathematical library functions (exp, log, sin, cos, etc.). The basic idea is to intercept these function calls and to execute some of them in lower precision, searching the space by using existing heuristics [25].

Section 4 introduced the timemory Gotcha capability for providing wholesale function replacement for optimization purposes and Listing 1.11 demonstrates the simplicity of this feature: the struct `mixed_prec_exp_t` shown there is a fully-defined timemory component. Additionally, as a by-product of the object-based design of and reference counting within the Gotcha component, timemory introduces the concept of a "scoped Gotcha", which deactivates the Gotcha wrapper when no object of that Gotcha component is within a start/stop region. Thus, in the mixed-precision analysis scenario, the developer can perform piece-wise analysis by simply changing the scope(s) of one or more instances of this component within a variadic wrapper, executing the application, and validating the result(s) until all regions which permit mixed-precision have been identified.

Listing 1.11. Using the Gotcha framework through timemory component

```
struct mixed_prec_exp : tim::component::base<mixed_prec_exp, void>
{
  double operator()(double v) { return PrecisionTuner(expf, exp, v); }
};
// pair the operator of mixed_prec_exp with a Gotcha
using mixed_prec_exp_t = gotcha<1, tuple<>, mixed_prec_exp>;
```

5.3 Roofline

The roofline model [29] is a visually intuitive performance model used to bound the performance of various numerical methods and operations running on multi-core, many-core, or accelerator processor architectures. It is a valuable tool in HPC to determine inherent performance limitations related to locality, bandwidth, and different parallelization paradigms.

The roofline model is an excellent example of the benefits of the timemory design. The generation of a roofline plot requires 3 capabilities: (1) a method for measuring the wall-clock run-time for all the desired regions, (2) a method for collecting the desired hardware-counter values for the all the desired regions, and (3) an empirical method for approximating the peak performance characteristics (*i.e.,* the "roof" part of the roofline). Although numerous existing tools undoubtedly included the capabilities #1 and #2 and capability #3 could be provided by the user's runtime, these tools do not expose enough modularity for this calculation to be fully integrated into the tool itself with respect to input and output. In other words, the lack of modularity in these tools necessitates the user engage in post-processing of the data outside of the application execution in order to generate the final result. Within the timemory framework, combining these three capabilities into a stand-alone output is arbitrary to provide since (1) there are no restrictions with respect to components using other components, (2) components are designed to be fully-functional when used explicitly instead of through a variadic wrapper, and (3) explicitly used component instances without variadic wrappers do not interact with the global call-graph storage unless the `insert_node()` and `pop_node()` member functions are invoked[8].

At present, timemory is the only existing tool, to the knowledge of the authors, that is capable of generating the roofline for both the CPU and GPU. Furthermore, timemory contains a built-in extension of the Roofline Model Toolkit [20] that is capable of stand-alone execution and provides a level of customization unavailable in any existing Roofline tools. The design of the roofline toolkit is such that the traditional algorithms for calculating the various peak-performance metrics of the roofline, *e.g.,* fused-multiply-add operations, can be customized within user applications in order to better emulate the operations of the target application.

[8] Thus, this eliminates the potential for data-corruption in the call-graph storage.

5.4 Instruction Roofline

Timemory provides support for instructions roofline plot generation on the GPU for applications which are integer heavy and do not make use of floating-point instructions. In [12] authors have used a GPU Kernel of Smith-Waterman algorithm [3] (GPU-BSW) as a case-study. Here, we use the Diagonal-Major memory indexing version of the same kernel to validate the timemory generated instruction roofline against the manually generated one in [12].

We used timemory's built-in features to auto-generate the instruction roofline shown in Fig. 3. It can be observed that the timemory generated roofline is similar to the manually created roofline in [12] for the same kernel on the same GPU (NVIDIA V100).

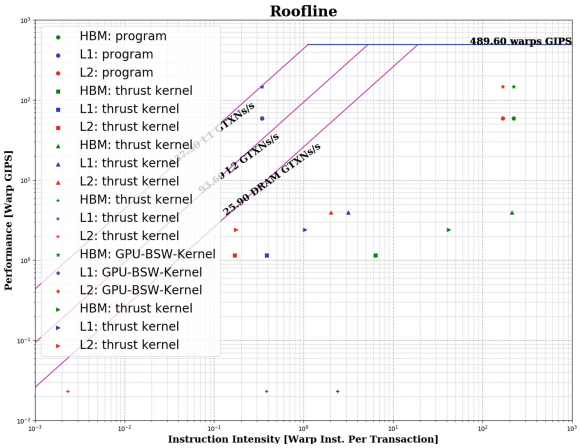

Fig. 3. Timemory generated instruction roofline for the diagonal major indexing GPU-BSW kernel

6 Future Work

In the near future, planned support includes MPI performance variables (MPI-T) [24] and extensions to the Python interface for post-processing context-trees and Jupyter notebooks. In the long-term, there are two goals for timemory which have not been prioritized. The first goal is to add support for compiler-assisted instrumentation in the form of compiler-flags and pragmas. The second goal is support for ClangJIT [14] which could theoretically add limited support for the injection of new components from C, Python, and Fortran.

7 Conclusion

This paper presents a unified framework for performance measurement and analysis, timemory. It provides an easy-to-use interface, supports multiple programming languages, object-level measurement granularity, and superior performance

in runtime overhead. The most significant contributions of timemory lie in its modular design, straightforward implementation of complex analysis methods such as the Roofline analysis, flexibility and extensibility for user-defined analysis, simplifications to the Gotcha model, and wide applicability to modern architectures such as CPUs and GPUs. With these favorable features, HPC users and performance engineers are expected to be able to perform profiling and analysis of large scale HPC applications in an easier, faster, and more flexible way.

Acknowledgment. Authors from Lawrence Berkeley National Laboratory were supported by the U.S. Department of Energy's Advanced Scientific Computing Research Program under contract DE-AC02-05CH11231.

This research used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-05CH11231.

References

1. Adhianto, L., et al.: HPCToolkit: tools for performance analysis of optimized parallel programs. Concurr. Comput.: Pract. Exp. **22**(6), 685–701 (2010). https://doi.org/10.1002/cpe.v22:6. http://hpctoolkit.org
2. Agostinelli, S., et al.: Geant4 simulation toolkit, **506**(3), 250–303 (2003). https://doi.org/10.1016/S0168-9002(03)01368-8. http://www.sciencedirect.com/science/article/pii/S0168900203013688
3. Awan, M.G.: GPU accelerated smith-waterman for performing batch alignments (GPU-BSW) (2019). https://github.com/m-gul/GPU-BSW
4. Bernat, A.R., et al.: Anywhere, any-time binary instrumentation. In: PASTE 2011, pp. 9–16. ACM, New York (2011). https://doi.org/10.1145/2024569.2024572
5. Boehme, D., et al.: The case for a common instrumentation interface for HPC codes. In: Workshop on Programming and Performance Visualization Tools (ProTools 19) (October 2019)
6. Boehme, D., et al.: Caliper: performance introspection for HPC software stacks. In: SC 2016, pp. 47:1–47:11. IEEE Computer Society (November 2016). http://dl.acm.org/citation.cfm?id=3014904.3014967. lLNL-CONF-699263
7. Carter Edwards, H., et al.: Kokkos. J. Parallel Distrib. Comput. **74**(12), 3202–3216 (2014). https://doi.org/10.1016/j.jpdc.2014.07.003
8. Corp., I.: Intel VTune profiler user guide - instrumenting your application (2019)
9. Corp., N.: CUPTI documentation (2019). https://docs.nvidia.com/cupti/Cupti/index.html
10. Di Franco, A., et al.: A comprehensive study of real-world numerical bug characteristics, pp. 509–519 (October 2017). https://doi.org/10.1109/ASE.2017.8115662
11. Dimakopoulou, M., et al.: Reliable and efficient performance monitoring in Linux. In: SC 2016, pp. 34:1–34:13. IEEE Press, Piscataway (2016). http://dl.acm.org/citation.cfm?id=3014904.3014950
12. Ding, N., et al.: An instruction roofline model for GPUs. In: Performance Modeling, Benchmarking and Simulation (PMBS19) (2019)
13. Eichenberger, A.E., et al.: OMPT: an OpenMP tools application programming interface for performance analysis. In: Rendell, A.P., Chapman, B.M., Müller, M.S. (eds.) IWOMP 2013. LNCS, vol. 8122, pp. 171–185. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40698-0_13

14. Finkel, H., et al.: ClangJIT: enhancing C++ with just-in-time compilation. CoRR (2019). http://arxiv.org/abs/1904.08555
15. Goldberg, D.: What every computer scientist should know about floating-point arithmetic. ACM Comput. Surv. **23**(1), 5–48 (1991). https://doi.org/10.1145/103162.103163. http://portal.acm.org/citation.cfm?doid=103162.103163
16. Kahan, W.: Personal website (2008). http://people.eecs.berkeley.edu/~wkahan/. Accessed 16 Dec 2019
17. Knüpfer, A., et al.: Score-p: a joint performance measurement run-time infrastructure for periscope, scalasca, tau, and vampir. In: Tools for High Performance Computing 2011, pp. 79–91. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31476-6_7
18. Lattner, C., Adve, V.: LLVM: a compilation framework for lifelong program analysis & transformation. In: Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization, CGO 2004, pp. 75–86. IEEE Computer Society, Washington (2004). http://dl.acm.org/citation.cfm?id=977395.977673
19. LLVM: LLVM (2019). https://llvm.org/
20. Lo, Y.J., et al.: Roofline model toolkit: a practical tool for architectural and program analysis. In: Jarvis, S.A., Wright, S.A., Hammond, S.D. (eds.) PMBS 2014. LNCS, vol. 8966, pp. 129–148. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-17248-4_7
21. Markidis, S., et al.: NVIDIA tensor core programmability, performance & precision, pp. 522–531 (May 2018). https://doi.org/10.1109/IPDPSW.2018.00091. arXiv: 1803.04014
22. Mintchev, S., Getov, V.: PMPI: high-level message passing in Fortran77 and C. In: Hertzberger, B., Sloot, P. (eds.) HPCN-Europe 1997. LNCS, vol. 1225, pp. 601–614. Springer, Heidelberg (1997). https://doi.org/10.1007/BFb0031632
23. Poliakoff, D., LeGendre, M.: Gotcha: an function-wrapping interface for HPC tools. In: Bhatele, A., Boehme, D., Levine, J.A., Malony, A.D., Schulz, M. (eds.) ESPT/VPA 2017-2018. LNCS, vol. 11027, pp. 185–197. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17872-7_11
24. Ramesh, S., et al.: MPI performance engineering with the MPI tool interface: The integration of mvapich and tau, EuroMPI 2017, pp. 16:1–16:11. ACM, New York (2017). https://doi.org/10.1145/3127024.3127036
25. Rubio-González, C., et al.: Precimonious: tuning assistant for floating-point precision, pp. 1–12 (November 2013). https://doi.org/10.1145/2503210.2503296
26. Shende, S.S., Malony, A.D.: The tau parallel performance system. Int. J. High Perform. Comput. Appl. **20**(2), 287–311 (2006). https://doi.org/10.1177/1094342006064482
27. Terpstra, D., et al.: Collecting performance data with PAPI-C. In: Müller, M., Resch, M., Schulz, A., Nagel, W. (eds.)Tools for High Performance Computing 2009, pp. 157–173. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-11261-4_11
28. Treibig, J., et al.: LIKWID: a lightweight performance-oriented tool suite for x86 multicore environments. In: ICPPW 2010, pp. 207–216. IEEE Computer Society, Washington (2010). https://doi.org/10.1109/ICPPW.2010.38
29. Williams, S., et al.: Roofline: an insightful visual performance model for multicore architectures. Commun. ACM **52**(4), 65–76 (2009). https://doi.org/10.1145/1498765.1498785