Autonomic Performance Environment for eXascale (APEX)

2.3.1

Generated by Doxygen 1.8.17

Chapter 1

APEX: Autonomic Performance Environment for eXascale

1.1 Copyright

1.2 Overview

One of the key components of the XPRESS project is a new approach to performance observation, measurement, analysis and runtime decision making in order to optimize performance. The particular challenges of accurately measuring the performance characteristics of ParalleX applications requires a new approach to parallel performance observation. The standard model of multiple operating system processes and threads observing themselves in a first-person manner while writing out performance profiles or traces for offline analysis will not adequately capture the full execution context, nor provide opportunities for runtime adaptation within OpenX. The approach taken in the XPRESS project is a new performance measurement system, called (Autonomic Performance Environment for eXascale). APEX will include methods for information sharing between the layers of the software stack, from the hardware through operating and runtime systems, all the way to domain specific or legacy applications. The performance measurement components will incorporate relevant information across stack layers, with merging of third-person performance observation of node-level and global resources, remote processes, and both operating and runtime system threads.

1.3 Introduction

1.3.1 Interfaces

Essentially, APEX is both a measurement system for introspection, as well as a Policy Engine for modifying runtime behavior based on the observations. While APEX has capabilities for generating profile data for post-mortem analysis, the key purpose of the measurement is to provide support for policy enforcement. To that end, APEX is designed to have very low overhead and minimize perturbation of runtime worker thread productivity. APEX supports both start/stop timers and either event-based or periodic counter samples. Measurements are taken synchronously, but profiling statistics and internal state management is performed by (preferably lower-priority) threads distinct from the running application. The heart of APEX is an event handler that dispatches events to registered listeners within APEX. Policy enforcement can trigger synchronously when events are triggered by the OS/RS or application, or can occur asynchronously on a periodic basis.

APEX is a library written in C++, and has both C and C++ external interfaces. While the C interface can be used for either language, some C++ applications prefer to work with namespaces (i.e. apex::$*$) rather than prefixes (i.e. apex_$*$). All functionality is supported through both interfaces, and the C interface contains inlined implementations of the C++ code.

While the designed purpose for APEX is supporting the current and future needs of ParalleX runtimes within the XPRESS project (such as HPX3, HPX5), experimental support is also available for introspection of current runtimes such as OpenMP. APEX could potentially be integrated into other runtime systems, such as any of a number of lightweight task based systems. The introspection provided by APEX is intended to be in the third-person model, rather than traditional first-person, per-thread/per-process application profile or tracing measurement. APEX is designed to combine information from the OS, Runtime, hardware and application in order to guide policy decisions.

For distributed communication, APEX provides an API to be implemented for the required communication for a given application. An MPI implementation is provided as a reference, and both HPX3 and HPX5 implementations have been implemented. In this way, APEX is integrated into the observed runtime, and asyncrhonous communication is provided at a lower priority, in order to minimize perturbation of the application.

The direct links to each API are here:

- C API : apex.h
- C++ API : apex

1.4 User Manual

For a complete user manual, please see the APEX documentation.

1.5 Acknowledgements

Support for this work was provided through Scientific Discovery through Advanced Computing (SciDAC) program funded by U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research (and Basic Energy Sciences/Biological and Environmental Research/High Energy Physics/Fusion Energy Sciences/Nuclear Physics) under award numbers DE-SC0008638, DE-SC0008704, DE-FG02-11ER26050 and DE-SC0006925.

Chapter 2

Namespace Index

2.1 Namespace List

Here is a list of all documented namespaces with brief descriptions:

Chapter 3

Class Index

3.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 4

File Index

4.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 5

Namespace Documentation

5.1 apex Namespace Reference

The main APEX namespace.

Classes

- class scoped_thread

  A convenience class for creating a scoped thread.
- class scoped_timer

  A convenience class for using APEX in C++ applications.
- struct task_wrapper

  A wrapper around APEX tasks.

Functions

- static std::shared_ptr< task_wrapper > null_task_wrapper (nullptr)

  declare a default "null" pointer for std::shared_ptr<task_wrapper>& references
- uint64_t init (const char *thread_name, const uint64_t comm_rank, const uint64_t comm_size)

  Intialize APEX.
- std::string dump (bool reset)

  Dump output from APEX.
- void finalize (void)

  Finalize APEX.
- void cleanup (void)

  Cleanup APEX.
- profiler * start (const std::string &timer_name)

  Start a timer.
- profiler * start (const apex_function_address function_address)

  Start a timer.
- void start (std::shared_ptr< task_wrapper > task_wrapper_ptr)

  Start a timer.
- void stop (profiler *the_profiler, bool cleanup=true)

  Stop a timer.

- void stop (std::shared_ptr< task_wrapper > task_wrapper_ptr)

  Stop a timer.
- void yield (profiler *the_profiler)

  Stop a timer, but don't increment the number of calls.
- void yield (std::shared_ptr< task_wrapper > task_wrapper_ptr)

  Stop a timer, but don't increment the number of calls.
- profiler * resume (const std::string &timer_name)

  Resume a timer.
- profiler * resume (const apex_function_address function_address)

  Resume a timer.
- void resume (std::shared_ptr< task_wrapper > task_wrapper_ptr)

  Resume a timer.
- void reset (const std::string &timer_name)

  Reset a timer or counter.
- void reset (apex_function_address function_address)

  Reset a timer.
- void set_state (apex_thread_state state)

  Set the thread state.
- void sample_value (const std::string &name, double value)

  Sample a state value.
- std::shared_ptr< task_wrapper > new_task (const std::string &name, const uint64_t task_id=UINTMAX_MAX, const std::shared_ptr< apex::task_wrapper > parent_task=null_task_wrapper)

  Create a new task (dependency).
- std::shared_ptr< task_wrapper > new_task (const apex_function_address function_address, const uint64_t task_id=UINTMAX_MAX, const std::shared_ptr< apex::task_wrapper > parent_task=null_task_wrapper)

  Create a new task (dependency).
- std::shared_ptr< task_wrapper > update_task (std::shared_ptr< task_wrapper > wrapper, const std::string &name)

  Update a task (dependency).
- std::shared_ptr< task_wrapper > update_task (std::shared_ptr< task_wrapper > wrapper, const apex_function_address function_address)

  Update a task wrapper (dependency).
- apex_event_type register_custom_event (const std::string &name)

  Register an event type with APEX.
- void custom_event (apex_event_type event_type, void *custom_data)

  Trigger a custom event.
- std::string & version (void)

  Return the APEX version.
- void register_thread (const std::string &name)

  Register a new thread.
- void exit_thread (void)

  Exit a thread.
- apex_policy_handle * register_policy (const apex_event_type when, std::function< int(apex_context const &)> f)

  Register a policy with APEX.
- std::set< apex_policy_handle * > register_policy (std::set< apex_event_type > when, std::function< int(apex_context const &)> f)

  Register a policy with APEX.
- apex_policy_handle * register_periodic_policy (unsigned long period, std::function< int(apex_context const &)> f)

  Register a policy with APEX.
- apex_policy_handle * sample_runtime_counter (unsigned long period, const std::string &counter_name)

Periodically sample a runtime counter.

- void deregister_policy (apex_policy_handle *handle)

  Deregister a policy with APEX.
- void stop_all_async_threads (void)

  Stop all asynchronous APEX background threads.
- apex_profile * get_profile (apex_function_address function_address)

  Get the current profile for the specified function address.
- apex_profile * get_profile (const std::string &timer_name)

  Get the current profile for the specified function address.
- int setup_power_cap_throttling (void)

  Initialize the power cap throttling policy.
- int setup_timer_throttling (apex_function_address the_address, apex_optimization_criteria_t criteria, apex_optimization_method_t method, unsigned long update_interval)

  Setup throttling to optimize for the specified function.
- int setup_throughput_tuning (apex_function_address the_address, apex_optimization_criteria_t criteria, apex_event_type event_type, int num_inputs, long **inputs, long *mins, long *maxs, long *steps)

  Setup throttling to optimize for the specified function, using multiple input criteria.
- apex_tuning_session_handle setup_custom_tuning (std::function< double(void)> metric, apex_event_type event_type, int num_inputs, long **inputs, long *mins, long *maxs, long *steps)

  Setup tuning of specified parameters to optimize for a custom metric, using multiple input criteria.
- apex_tuning_session_handle setup_custom_tuning (apex_tuning_request &request)

  Setup tuning of specified parameters to optimize for a custom metric, using multiple input criteria of potentially multiple types.
- int setup_timer_throttling (const std::string &the_name, apex_optimization_criteria_t criteria, apex_optimization_method_t method, unsigned long update_interval)

  Setup throttling to optimize for the specified function or counter.
- int shutdown_throttling (void)

  Terminate the throttling policy.
- int get_thread_cap (void)

  Get the current thread cap set by the throttling.
- void set_thread_cap (int new_cap)

  Set the current thread cap for throttling.
- std::vector< std::pair< std::string, long * > > & get_tunable_params (apex_tuning_session_handle h)

  Return a vector of the current tunable parameters.
- bool has_session_converged (apex_tuning_session_handle handle)

  Check whether a tuning session has converged.
- void get_best_values (apex_tuning_session_handle h)

  Set a tuning session's values to the best known values.
- void print_options (void)

  Print out all configuration settings for APEX.
- void send (uint64_t tag, uint64_t size, uint64_t target)

  Notify APEX that the current thread is sending a parcel/message to another rank/locality/process.
- void recv (uint64_t tag, uint64_t size, uint64_t source_rank, uint64_t source_thread)

  Notify APEX that the current thread is receiving a parcel/message from another rank/locality/process.

5.1.1 Detailed Description

The main APEX namespace.

The C++ interface for APEX uses the apex namespace. In comparison, The C interface has functions that start with "apex_".

5.1.2 Function Documentation

5.1.2.1 cleanup()

```
void apex::cleanup (
            void  )
```

Cleanup APEX.

Warning

For best results, this function should be explicitly called to free all memory allocated by APEX. If not explicitly called from the application or runtime, it will be automatically called when the APEX main singleton object is destructed. apex::finalize will be automatically called from apex::cleanup if it has not yet been called.

The cleanup method will free all allocated memory for APEX.

Returns

No return value.

See also

apex::init apex::finalize

5.1.2.2 custom_event()

```
void apex::custom_event (
            apex_event_type event_type,
            void * custom_data )
```

Trigger a custom event.

This function will pass a custom event to the APEX event listeners. Each listeners' custom event handler will handle the custom event. Policy functions will be passed the custom event name in the event context.

Parameters

event_type	The type of the custom event
custom_data	Data specific to the custom event

Returns

No return value.

See also

apex::register_custom_event

5.1.2.3 deregister_policy()

```
void apex::deregister_policy (
            apex_policy_handle * handle )
```

Deregister a policy with APEX.

This function will deregister the specified policy. In order to enable the policy again, it should be registered using apex::register_policy or apex::register_periodic_policy.

Parameters

handle	The handle of the policy to be deregistered.

See also

apex::register_policy, apex::register_periodic_policy

5.1.2.4 dump()

```
std::string apex::dump (
            bool reset )
```

Dump output from APEX.

The stop measurement method will terminate all measurement and optionally:

- print a report to the screen
- write a profile to disk (if requested)
- output all other visualization data

Parameters

reset	Whether to reset all statistics

Returns

a string containing the output

See also

apex::finalize

5.1.2.5 exit_thread()

```
void apex::exit_thread (
            void  )
```

Exit a thread.

For multithreaded applications, exit this thread and clean up.

Warning

Failure to exit a thread with APEX may invalidate statistics.

Returns

No return value.

5.1.2.6 finalize()

```
void apex::finalize (
            void  )
```

Finalize APEX.

The stop measurement method will terminate all measurement and optionally:

- print a report to the screen
- write a profile to disk (if requested)

Returns

No return value.

See also

apex::init

5.1.2.7 get_best_values()

```
void apex::get_best_values (
            apex_tuning_session_handle h )
```

Set a tuning session's values to the best known values.

Parameters

h	The handle for the tuning session of interest.

5.1.2.8 get_profile() [1/2]

```
apex_profile* apex::get_profile (
            apex_function_address function_address )
```

Get the current profile for the specified function address.

This function will return the current profile for the specified address. Because profiles are updated out-of-band, it is possible that this profile value is out of date.

Parameters

function_address	The address of the function.

Returns

The current profile for that timed function.

5.1.2.9 get_profile() [2/2]

```
apex_profile* apex::get_profile (
            const std::string & timer_name )
```

Get the current profile for the specified function address.

This function will return the current profile for the specified address. Because profiles are updated out-of-band, it is possible that this profile value is out of date. This profile can be either a timer or a sampled value.

Parameters

timer_name	The name of the function

Returns

The current profile for that timed function or sampled value.

5.1.2.10 get_thread_cap()

```
int apex::get_thread_cap (
            void  )
```

Get the current thread cap set by the throttling.

This function will return the current thread cap based on the throttling policy.

Returns

The current thread cap value.

5.1.2.11 get_tunable_params()

```
std::vector<std::pair<std::string,long*> >& apex::get_tunable_params (
            apex_tuning_session_handle h )
```

Return a vector of the current tunable parameters.

Returns

A vector of pairs; the first element is the name of the tunable parameter, while the second is a pointer to its value.

5.1.2.12 has_session_converged()

```
bool apex::has_session_converged (
            apex_tuning_session_handle handle )
```

Check whether a tuning session has converged.

Parameters

handle	The handle for the tuning session of interest.

Returns

true if the tuning session has converged, otherwise false

5.1.2.13 init()

```
uint64_t apex::init (
            const char * thread_name,
```

```
            const uint64_t comm_rank,
            const uint64_t comm_size )
```

Intialize APEX.

Warning

For best results, this function should be called before any other APEX functions.

Use this version of apex::init when you do not have access to the input arguments.

Parameters

thread_name	The name of the thread, or NULL. The lifetime of the thread will be timed with a timer using this same name.
comm_rank	The rank of this process within the full distributed application, i.e. MPI rank or HPX locality.
comm_size	The total number of processes within the full distributed application, i.e. MPI comm_size or total number of HPX localities.

Returns

APEX_NOERROR on success, or APEX_ERROR on failure.

See also

apex::init apex::finalize

5.1.2.14 new_task() [1/2]

```
std::shared_ptr<task_wrapper> apex::new_task (
            const apex_function_address function_address,
            const uint64_t task_id = UINTMAX_MAX,
            const std::shared_ptr< apex::task_wrapper > parent_task = null_task_wrapper )
```

Create a new task (dependency).

This function will note a task dependency between the current timer (task) and the new task.

Parameters

function_address	The function address of the timer.
task_id	The ID of the task (default of -1 implies none provided by runtime)
parent_task	The apex::task_wrapper (if available) that is the parent task of this task

Returns

pointer to an apex::task_wrapper object

5.1.2.15 new_task() [2/2]

```
std::shared_ptr<task_wrapper> apex::new_task (
            const std::string & name,
            const uint64_t task_id = UINTMAX_MAX,
            const std::shared_ptr< apex::task_wrapper > parent_task = null_task_wrapper )
```

Create a new task (dependency).

This function will note a task dependency between the current timer (task) and the new task.

Parameters

name	The name of the timer.
task_id	The ID of the task (default of UINTMAX_MAX implies none provided by runtime)
parent_task	The apex::task_wrapper (if available) that is the parent task of this task

Returns

pointer to an apex::task_wrapper object

5.1.2.16 null_task_wrapper()

```
static std::shared_ptr<task_wrapper> apex::null_task_wrapper (
            nullptr ) [static]
```

declare a default "null" pointer for std::shared_ptr<task_wrapper>& references

See also

apex::task_wrapper

5.1.2.17 print_options()

```
void apex::print_options (
            void )
```

Print out all configuration settings for APEX.

5.1.2.18 recv()

```
void apex::recv (
            uint64_t tag,
            uint64_t size,
            uint64_t source_rank,
            uint64_t source_thread )
```

Notify APEX that the current thread is receiving a parcel/message from another rank/locality/process.

This method notifies APEX that the current thread is receiving a parcel/message from another rank/locality/process. The tag is meant to be an identifier for the message, not required to be unique. The source value is the APEX rank of the source of the message.

Parameters

tag	The message identifier
size	The message size (in bytes)
source_rank	The message source (as a rank/locality index)
source_thread	The message source (as a worker index - 0 if unknown)

5.1.2.19 register_custom_event()

```
apex_event_type apex::register_custom_event (
            const std::string & name )
```

Register an event type with APEX.

Create a user-defined event type for APEX.

Parameters

name	The name of the custom event

Returns

The index of the custom event.

See also

apex::custom_event

5.1.2.20 register_periodic_policy()

```
apex_policy_handle* apex::register_periodic_policy (
            unsigned long period,
            std::function< int(apex_context const &)> f )
```

Register a policy with APEX.

Apex provides the ability to call an application-specified function periodically. This assigns the passed in function to be called on a periodic basis. The context for the event will be passed to the registered function.

Parameters

period	How frequently the function should be called (in microseconds)
f	The function to be called when that event is handled by APEX.

Returns

A handle to the policy, to be stored if the policy is to be un-registered later.

5.1.2.21 register_policy() [1/2]

```
apex_policy_handle* apex::register_policy (
            const apex_event_type when,
            std::function< int(apex_context const &)> f )
```

Register a policy with APEX.

Apex provides the ability to call an application-specified function when certain events occur in the APEX library, or periodically. This assigns the passed in function to the event, so that when that event occurs in APEX, the function is called. The context for the event will be passed to the registered function.

Parameters

when	The APEX event when this function should be called
f	The function to be called when that event is handled by APEX.

Returns

A handle to the policy, to be stored if the policy is to be un-registered later.

See also

apex::deregister_policy, apex::register_periodic_policy

5.1.2.22 register_policy() [2/2]

```
std::set<apex_policy_handle*> apex::register_policy (
            std::set< apex_event_type > when,
            std::function< int(apex_context const &)> f )
```

Register a policy with APEX.

Apex provides the ability to call an application-specified function when certain events occur in the APEX library, or periodically. This assigns the passed in function to the event, so that when that event occurs in APEX, the function is called. The context for the event will be passed to the registered function.

Parameters

when	The set of APEX events when this function should be called
f	The function to be called when that event is handled by APEX.

Returns

A handle to the policy, to be stored if the policy is to be un-registered later.

See also

apex::deregister_policy, apex::register_periodic_policy

5.1.2.23 register_thread()

```
void apex::register_thread (
            const std::string & name )
```

Register a new thread.

For multithreaded applications, register a new thread with APEX.

Warning

Failure to register a thread with APEX may invalidate statistics, and may prevent the ability to use timers or sampled values for this thread.

Parameters

name	The name that will be assigned to the new thread.

Returns

No return value.

5.1.2.24 reset() [1/2]

```
void apex::reset (
            apex_function_address function_address )
```

Reset a timer.

This function will reset the profile associated with the specified timer to zero.

Parameters

function_address	The function address of the timer.

Returns

No return value.

5.1.2.25 reset() [2/2]

```
void apex::reset (
            const std::string & timer_name )
```

Reset a timer or counter.

This function will reset the profile associated with the specified timer or counter name to zero.

Parameters

timer_name	The name of the timer.

Returns

No return value.

See also

apex::get_profile

5.1.2.26 resume() [1/3]

```
profiler* apex::resume (
            const apex_function_address function_address )
```

Resume a timer.

This function will create a profiler object in APEX, and return a handle to the object. The object will be associated with the address passed in to this function. The difference between this function and the apex::start function is that the number of calls to that timer will not be incremented.

Parameters

function_address	The address of the function to be timed

Returns

The handle for the timer object in APEX. Not intended to be queried by the application. Should be retained locally, if possible, and passed in to the matching apex::stop call when the timer should be stopped.

See also

apex::stop, apex::yield, apex::start

5.1.2.27 resume() [2/3]

```
profiler* apex::resume (
            const std::string & timer_name )
```

Resume a timer.

This function will create a profiler object in APEX, and return a handle to the object. The object will be associated with the name passed in to this function. The difference between this function and the apex::start function is that the number of calls to that timer will not be incremented.

Parameters

timer_name	The name of the timer.

Returns

The handle for the timer object in APEX. Not intended to be queried by the application. Should be retained locally, if possible, and passed in to the matching apex::stop() call when the timer should be stopped.

See also

apex::stop, apex::yield, apex::start

5.1.2.28 resume() [3/3]

```
void apex::resume (
            std::shared_ptr< task_wrapper > task_wrapper_ptr )
```

Resume a timer.

This function will create a profiler object in APEX, and return a handle to the object. The object will be associated with the address passed in to this function. The difference between this function and the apex::start function is that the number of calls to that timer will not be incremented.

Parameters

task_wrapper_ptr	A pointer to an apex::task_wrapper created by apex::new_task. APEX will use this to store the profiler data.

Returns

No return value.

See also

apex::stop, apex::yield, apex::start

5.1.2.29 sample_runtime_counter()

```
apex_policy_handle* apex::sample_runtime_counter (
            unsigned long period,
            const std::string & counter_name )
```

Periodically sample a runtime counter.

Apex provides the ability to call an application-specified function periodically. This assigns the passed in function to be called on a periodic basis. The context for the event will be passed to the registered function.

Parameters

period	How frequently the counter should be called (in microseconds)
counter_name	The name of the counter to sample

Returns

A handle to the policy, to be stored if the policy is to be un-registered later.

5.1.2.30 sample_value()

```
void apex::sample_value (
            const std::string & name,
            double value )
```

Sample a state value.

This function will retain a sample of some value. The profile for this sampled value will store the min, mean, max, total and standard deviation for this value for all times it is sampled.

Parameters

name	The name of the sampled value
value	The sampled value

Returns

No return value.

5.1.2.31 send()

```
void apex::send (
             uint64_t tag,
             uint64_t size,
             uint64_t target )
```

Notify APEX that the current thread is sending a parcel/message to another rank/locality/process.

This method notifies APEX that the current thread is sending a parcel/message to another rank/locality/process. The tag is meant to be an identifier for the message, not required to be unique. The target value is the APEX rank of the target of the message.

Parameters

tag	The message identifier
size	The message size (in bytes)
target	The message target (as an APEX rank)

5.1.2.32 set_state()

```
void apex::set_state (
             apex_thread_state state )
```

Set the thread state.

This function will set the thread state in APEX for 3rd party observation

Parameters

state	The state of the thread.

Returns

No return value.

5.1.2.33 set_thread_cap()

```
void apex::set_thread_cap (
             int new_cap )
```

Set the current thread cap for throttling.

This function will set the current thread cap based on an external throttling policy.

Parameters

new_cap	The current thread cap value.

5.1.2.34 setup_custom_tuning() [1/2]

```
apex_tuning_session_handle apex::setup_custom_tuning (
            apex_tuning_request & request )
```

Setup tuning of specified parameters to optimize for a custom metric, using multiple input criteria of potentially multiple types.

This function will initialize a policy to optimize a custom metric, using metric and parameters specified in the tuning request. The system tries to minimize the custom metric.

Parameters

request	An apex_tuning_request object that specifies the tuning parameters.

Returns

A handle to the tuning session.

5.1.2.35 setup_custom_tuning() [2/2]

```
apex_tuning_session_handle apex::setup_custom_tuning (
            std::function< double(void)> metric,
            apex_event_type event_type,
            int num_inputs,
            long ** inputs,
            long * mins,
            long * maxs,
            long * steps )
```

Setup tuning of specified parameters to optimize for a custom metric, using multiple input criteria.

This function will initialize a policy to optimize a custom metric, using the list of tunable parameters. The system tries to minimize the custom metric. After evaluating the state of the system, the policy will assign new values to the inputs.

Parameters

metric	A function returning the value to be minimized.

Parameters

event_type	The apex_event_type that should trigger this policy
num_inputs	The number of tunable inputs for optimization
inputs	An array of addresses to inputs for optimization
mins	An array of minimum values for each input
maxs	An array of maximum values for each input
steps	An array of step values for each input

Returns

A handle to the tuning session

5.1.2.36 setup_power_cap_throttling()

```
int apex::setup_power_cap_throttling (
            void  )
```

Initialize the power cap throttling policy.

This function will initialize APEX for power cap throttling. There are several environment variables that control power cap throttling:

HPX_THROTTLING If set, throttling will be enabled and initialized at startup.

APEX_THROTTLING_MAX_THREADS The maximum number of threads the throttling system will allow. The default value is 48.

APEX_THROTTLING_MIN_THREADS The minimum number of threads the throttling system will allow. The default value is 12.

APEX_THROTTLING_MAX_WATTS The maximum number of Watts the system can consume as an average rate. The default value is 220.

APEX_THROTTLING_MIN_WATTS The minimum number of Watts the system can consume as an average rate. The default value is 180.

HPX_ENERGY_THROTTLING If set, power/energy throttling will be performed.

HPX_ENERGY TBD

After evaluating the state of the system, the policy will set the thread cap, which can be queried using apex::get_thread_cap().

Returns

APEX_NOERROR on success, otherwise an error code.

5.1.2.37 setup_throughput_tuning()

```
int apex::setup_throughput_tuning (
              apex_function_address the_address,
              apex_optimization_criteria_t criteria,
              apex_event_type event_type,
              int num_inputs,
              long ** inputs,
              long * mins,
              long * maxs,
              long * steps )
```

Setup throttling to optimize for the specified function, using multiple input criteria.

This function will initialize a policy to optimize the specified function, using the list of tunable inputs for the specified function. The optimization criteria include maximizing throughput, minimizing or maximizing time spent in the specified function. After evaluating the state of the system, the policy will assign new values to the inputs.

Parameters

the_address	The address of the function to be optimized.
criteria	The optimization criteria.
event_type	The apex_event_type that should trigger this policy
num_inputs	The number of tunable inputs for optimization
inputs	An array of addresses to inputs for optimization
mins	An array of minimum values for each input
maxs	An array of maximum values for each input
steps	An array of step values for each input

Returns

APEX_NOERROR on success, otherwise an error code.

5.1.2.38 setup_timer_throttling() [1/2]

```
int apex::setup_timer_throttling (
              apex_function_address the_address,
              apex_optimization_criteria_t criteria,
              apex_optimization_method_t method,
              unsigned long update_interval )
```

Setup throttling to optimize for the specified function.

This function will initialize the throttling policy to optimize for the specified function. The optimization criteria include maximizing throughput, minimizing or maximizing time spent in the specified function. After evaluating the state of the system, the policy will set the thread cap, which can be queried using apex::get_thread_cap().

Parameters

the_address	The address of the function to be optimized.
criteria	The optimization criteria.
method	The optimization method.
update_interval	The time between observations, in microseconds.

Returns

APEX_NOERROR on success, otherwise an error code.

5.1.2.39 setup_timer_throttling() [2/2]

```
int apex::setup_timer_throttling (
            const std::string & the_name,
            apex_optimization_criteria_t criteria,
            apex_optimization_method_t method,
            unsigned long update_interval )
```

Setup throttling to optimize for the specified function or counter.

This function will initialize the throttling policy to optimize for the specified function or counter. The optimization criteria include maximizing throughput, minimizing or maximizing time spent in the specified function or value sampled in the counter. After evaluating the state of the system, the policy will set the thread cap, which can be queried using apex::get_thread_cap().

Parameters

the_name	The name of the function or counter to be optimized.
criteria	The optimization criteria.
method	The optimization method.
update_interval	The time between observations, in microseconds.

Returns

APEX_NOERROR on success, otherwise an error code.

5.1.2.40 shutdown_throttling()

```
int apex::shutdown_throttling (
            void  )
```

Terminate the throttling policy.

This function will terminate the throttling policy.

Returns

APEX_NOERROR on success, otherwise an error code.

5.1.2.41 start() [1/3]

```
profiler* apex::start (
            const apex_function_address function_address )
```

Start a timer.

This function will create a profiler object in APEX, and return a handle to the object. The object will be associated with the address passed in to this function.

Parameters

function_address	The address of the function to be timed

Returns

The handle for the timer object in APEX. Not intended to be queried by the application. Should be retained locally, if possible, and passed in to the matching apex::stop call when the timer should be stopped.

See also

apex::stop, apex::yield, apex::resume

5.1.2.42 start() [2/3]

```
profiler* apex::start (
            const std::string & timer_name )
```

Start a timer.

This function will create a profiler object in APEX, and return a handle to the object. The object will be associated with the name passed in to this function.

Parameters

timer_name	The name of the timer.

Returns

The handle for the timer object in APEX. Not intended to be queried by the application. Should be retained locally, if possible, and passed in to the matching apex::stop() call when the timer should be stopped.

See also

apex::stop, apex::yield, apex::resume

5.1.2.43 start() [3/3]

```
void apex::start (
            std::shared_ptr< task_wrapper > task_wrapper_ptr )
```

Start a timer.

This function will create a profiler object in APEX, and return a handle to the object. The object will be associated with the task_wrapper passed in to this function.

Parameters

task_wrapper_ptr	A pointer to an apex::task_wrapper created by apex::new_task. APEX will use this to store the profiler data.

Returns

No return value.

See also

apex::stop, apex::yield, apex::resume apex::new_task

5.1.2.44 stop() [1/2]

```
void apex::stop (
            profiler * the_profiler,
            bool cleanup = true )
```

Stop a timer.

This function will stop the specified profiler object, and queue the profiler to be processed out-of-band. The timer value will eventually added to the profile for the process.

Parameters

the_profiler	The handle of the profiler object.
cleanup	Internal use only.

Returns

No return value.

See also

apex::start, apex::yield, apex::resume

5.1.2.45 stop() [2/2]

```
void apex::stop (
            std::shared_ptr< task_wrapper > task_wrapper_ptr )
```

Stop a timer.

This function will stop the specified profiler object, and queue the profiler to be processed out-of-band. The timer value will eventually added to the profile for the process.

Parameters

task_wrapper_ptr	an apex::task_wrapper pointer that was started

Returns

No return value.

See also

apex::start, apex::yield, apex::resume, apex::new_task

5.1.2.46 stop_all_async_threads()

```
void apex::stop_all_async_threads (
            void  )
```

Stop all asynchronous APEX background threads.

This function will stop all background threads from all listeners, and force them to exit. This is necessary for stopping perodic policies that would otherwise prevent runtimes from finishing work and calling apex::finalize().

See also

apex::deregister_policy, apex::register_policy apex::finalize

5.1.2.47 update_task() [1/2]

```
std::shared_ptr<task_wrapper> apex::update_task (
            std::shared_ptr< task_wrapper > wrapper,
            const apex_function_address function_address )
```

Update a task wrapper (dependency).

This function will update the function address that this task wrapper refers to.

Parameters

wrapper	The existing apex::task_wrapper object
function_address	The new function address of the timer.

5.1.2.48 update_task() [2/2]

```
std::shared_ptr<task_wrapper> apex::update_task (
            std::shared_ptr< task_wrapper > wrapper,
            const std::string & name )
```

Update a task (dependency).

This function will update the name that this task wrapper refers to.

Parameters

wrapper	The existing apex::task_wrapper object
name	The new name of the timer.

5.1.2.49 version()

```
std::string& apex::version (
            void  )
```

Return the APEX version.

Returns

A string with the APEX version.

5.1.2.50 yield() [1/2]

```
void apex::yield (
            profiler * the_profiler )
```

Stop a timer, but don't increment the number of calls.

This function will stop the specified profiler object, and queue the profiler to be processed out-of-band. The timer value will eventually added to the profile for the process. The number of calls will NOT be incremented - this "task" was yielded, not completed. It will be resumed by another thread at a later time.

Parameters

the_profiler	The handle of the profiler object.

Returns

No return value.

See also

apex::start, apex::stop, apex::resume

5.1.2.51 yield() [2/2]

```
void apex::yield (
            std::shared_ptr< task_wrapper > task_wrapper_ptr )
```

Stop a timer, but don't increment the number of calls.

This function will stop the specified profiler object, and queue the profiler to be processed out-of-band. The timer value will eventually added to the profile for the process. The number of calls will NOT be incremented - this "task" was yielded, not completed. It will be resumed by another thread at a later time.

Parameters

task_wrapper_ptr	an apex::task_wrapper pointer that was started

Returns

No return value.

See also

apex::start, apex::stop, apex::resume

Chapter 6

Class Documentation

6.1 apex_event_type Struct Reference

Typedef for enumerating the different event types.

`#include <apex_types.h>`

6.1.1 Detailed Description

Typedef for enumerating the different event types.

The documentation for this struct was generated from the following file:

- /Users/khuck/src/xpress-apex/src/apex/apex_types.h

6.2 apex_profiler_type Struct Reference

Typedef for enumerating the different timer types.

`#include <apex_types.h>`

6.2.1 Detailed Description

Typedef for enumerating the different timer types.

The documentation for this struct was generated from the following file:

- /Users/khuck/src/xpress-apex/src/apex/apex_types.h

6.3 apex::scoped_thread Class Reference

A convenience class for creating a scoped thread.

`#include <apex_api.hpp>`

Public Member Functions

- scoped_thread (const std::string &thread_name)
  Constructor.
- ~scoped_thread ()
  Destructor.

6.3.1 Detailed Description

A convenience class for creating a scoped thread.

This object will register a thread when it starts, and when the object goes out of scope, it will tell APEX that the thread has exited.

6.3.2 Constructor & Destructor Documentation

6.3.2.1 scoped_thread()

```
apex::scoped_thread::scoped_thread (
            const std::string & thread_name )  [inline]
```

Constructor.

Parameters

thread_name	The name of this thread (nullptr allowed).

The documentation for this class was generated from the following file:

- /Users/khuck/src/xpress-apex/src/apex/apex_api.hpp

6.4 apex::scoped_timer Class Reference

A convenience class for using APEX in C++ applications.

`#include <apex_api.hpp>`

Public Member Functions

- scoped_timer (uint64_t func)
  Construct and start an APEX timer.
- scoped_timer (std::string func)
  Construct and start an APEX timer.
- scoped_timer (uint64_t func, std::shared_ptr< apex::task_wrapper > parent)
  Register a new thread with APEX, then construct and start an APEX timer.
- scoped_timer (std::string func, std::shared_ptr< apex::task_wrapper > parent)
  Register a new thread with APEX, then construct and start an APEX timer.
- void start (void)
  Start the APEX timer.
- void stop (void)
  Stop the APEX timer.
- void yield (void)
  Yield the APEX timer.
- void resume (void)
  Resume the APEX timer.
- ~scoped_timer ()
  Destructor.
- std::shared_ptr< apex::task_wrapper > get_task_wrapper (void)
  Get the internal task wrapper object.

6.4.1 Detailed Description

A convenience class for using APEX in C++ applications.

This class will automatically start an APEX timer when the object is constructed, and automatically stop the timer when the object goes out of scope. There are options for registering a thread when creating a timer for the first time on a thread.

6.4.2 Constructor & Destructor Documentation

6.4.2.1 scoped_timer() [1/4]

```
apex::scoped_timer::scoped_timer (
            uint64_t func ) [inline]
```

Construct and start an APEX timer.

Parameters

func	The address of a function used to identify the timer type

6.4.2.2 scoped_timer() [2/4]

```
apex::scoped_timer::scoped_timer (
            std::string func )  [inline]
```

Construct and start an APEX timer.

Parameters

func	The name of a function used to identify the timer type

6.4.2.3 scoped_timer() [3/4]

```
apex::scoped_timer::scoped_timer (
            uint64_t func,
            std::shared_ptr< apex::task_wrapper > parent )  [inline]
```

Register a new thread with APEX, then construct and start an APEX timer.

Parameters

func	The address of a function used to identify the timer type
parent	The parent task wrapper (if available) of this new scoped_timer

6.4.2.4 scoped_timer() [4/4]

```
apex::scoped_timer::scoped_timer (
            std::string func,
            std::shared_ptr< apex::task_wrapper > parent )  [inline]
```

Register a new thread with APEX, then construct and start an APEX timer.

Parameters

func	The name of a function used to identify the timer type
parent	The parent task wrapper (if available) of this new scoped_timer

6.4.3 Member Function Documentation

6.4.3.1 resume()

```
void apex::scoped_timer::resume (
            void  )  [inline]
```

Resume the APEX timer.

6.4.3.2 start()

```
void apex::scoped_timer::start (
            void  )  [inline]
```

Start the APEX timer.

6.4.3.3 stop()

```
void apex::scoped_timer::stop (
            void  )  [inline]
```

Stop the APEX timer.

6.4.3.4 yield()

```
void apex::scoped_timer::yield (
            void  )  [inline]
```

Yield the APEX timer.

The documentation for this class was generated from the following file:

- /Users/khuck/src/xpress-apex/src/apex/apex_api.hpp

6.5 apex::task_wrapper Struct Reference

A wrapper around APEX tasks.

`#include <task_wrapper.hpp>`

Public Member Functions

- task_wrapper (void)
  Constructor.
- task_identifier * get_task_id (void)
  Get the task_identifier for this task_wrapper.

Static Public Member Functions

- static std::shared_ptr< task_wrapper > & get_apex_main_wrapper (void)

  Static method to get a pre-defined task_wrapper around "main".

Public Attributes

- task_identifier * task_id

  A pointer to the task_identifier for this task_wrapper.
- profiler * prof

  A pointer to the active profiler object timing this task.
- uint64_t guid

  An internally generated GUID for this task.
- uint64_t parent_guid

  An internally generated GUID for the parent task of this task.
- std::shared_ptr< task_wrapper > parent

  A managed pointer to the parent task_wrapper for this task.
- std::vector< profiler * > data_ptr

  Internal usage, used to manage HPX direct actions when their parent task is yielded by the runtime.
- task_identifier * alias

  If the task changes names after creation (due to the application of an annotation) then the alias becomes the new task_identifier for the task.

6.5.1 Detailed Description

A wrapper around APEX tasks.

6.5.2 Member Function Documentation

6.5.2.1 get_apex_main_wrapper()

```
static std::shared_ptr<task_wrapper>& apex::task_wrapper::get_apex_main_wrapper (
            void  )  [inline], [static]
```

Static method to get a pre-defined task_wrapper around "main".

Returns

A shared pointer to the task_wrapper

6.5.2.2 get_task_id()

```
task_identifier* apex::task_wrapper::get_task_id (
            void  )  [inline]
```

Get the task_identifier for this task_wrapper.

Returns

A pointer to the task_identifier

The documentation for this struct was generated from the following file:

- /Users/khuck/src/xpress-apex/src/apex/task_wrapper.hpp

Chapter 7

File Documentation

7.1 /Users/khuck/src/xpress-apex/doc/apex.dox File Reference

7.2 /Users/khuck/src/xpress-apex/src/apex/apex.h File Reference

```
#include "apex_types.h"
#include "apex_export.h"
#include "stdbool.h"
#include "stdint.h"
```

Functions

- int apex_init (const char *thread_name, const uint64_t comm_rank, const uint64_t comm_size)

  Intialize APEX.
- const char * apex_dump (bool reset)

  Dump output from APEX.
- void apex_finalize ()

  Finalize APEX.
- void apex_cleanup ()

  Cleanup APEX.
- apex_profiler_handle apex_start (apex_profiler_type type, void *identifier)

  Start a timer.
- apex_profiler_handle apex_start_guid (apex_profiler_type type, void *identifier, uint64_t guid)

  Start a timer.
- void apex_stop (apex_profiler_handle profiler)

  Stop a timer.
- void apex_yield (apex_profiler_handle profiler)

  Stop a timer, but don't increment the number of calls.
- apex_profiler_handle apex_resume (apex_profiler_type type, void *identifier)

  Resume a timer.
- apex_profiler_handle apex_resume_guid (apex_profiler_type type, void *identifier, uint64_t guid)

  Resume a timer.
- void apex_reset (apex_profiler_type type, void *identifier)

Reset a timer or counter.

- void apex_set_state (apex_thread_state state)

  Set the thread state.
- void apex_sample_value (const char *name, double value)

  Sample a state value.
- void apex_new_task (apex_profiler_type type, void *identifier, uint64_t task_id)

  Create a new task (dependency).
- apex_event_type apex_register_custom_event (const char *name)

  Register an event type with APEX.
- void apex_custom_event (apex_event_type event_type, void *custom_data)

  Trigger a custom event.
- const char * apex_version (void)

  Return the APEX version.
- void apex_register_thread (const char *name)

  Register a new thread.
- void apex_exit_thread (void)

  Exit a thread.
- apex_policy_handle * apex_register_policy (const apex_event_type when, apex_policy_function f)

  Register a policy with APEX.
- apex_policy_handle * apex_register_periodic_policy (unsigned long period, apex_policy_function f)

  Register a policy with APEX.
- void apex_deregister_policy (apex_policy_handle *handle)

  Deregister a policy with APEX.
- apex_profile * apex_get_profile (apex_profiler_type type, void *identifier)

  Get the current profile for the specified id.
- double apex_current_power_high (void)

  Get the current power reading.
- int apex_setup_power_cap_throttling (void)

  Initialize the power cap throttling policy.
- int apex_setup_timer_throttling (apex_profiler_type type, void *identifier, apex_optimization_criteria_t criteria, apex_optimization_method_t method, unsigned long update_interval)

  Setup throttling to optimize for the specified function.
- int apex_setup_throughput_tuning (apex_profiler_type type, void *identifier, apex_optimization_criteria_t criteria, apex_event_type event_type, int num_inputs, long **inputs, long *mins, long *maxs, long *steps)

  Setup throttling to optimize for the specified function, using multiple input criteria.
- int apex_shutdown_throttling (void)

  Terminate the throttling policy.
- int apex_get_thread_cap (void)

  Get the current thread cap set by the throttling.
- void apex_set_thread_cap (int new_cap)

  Set the current thread cap for throttling.
- void apex_print_options (void)

  Print the current APEX settings.
- void apex_send (uint64_t tag, uint64_t size, uint64_t target)

  Notify APEX that the current thread is sending a parcel/message to another rank/locality/process.
- void apex_recv (uint64_t tag, uint64_t size, uint64_t source_rank, uint64_t source_thread)

  Notify APEX that the current thread is receiving a parcel/message from another rank/locality/process.
- uint64_t apex_hardware_concurrency (void)

  Get the number of possible threads supported on this system.

7.2.1 Function Documentation

7.2.1.1 apex_cleanup()

```
void apex_cleanup ( )
```

Cleanup APEX.

Warning

For best results, this function should be explicitly called to free all memory allocated by APEX. If not explicitly called from the application or runtime, it will be automatically called when the APEX main singleton object is destructed. apex_finalize will be automatically called from apex_cleanup if it has not yet been called.

The cleanup method will free all allocated memory for APEX.

Returns

No return value.

See also

apex_finalize

7.2.1.2 apex_current_power_high()

```
double apex_current_power_high (
            void  )
```

Get the current power reading.

This function will return the current power level for the node, measured in Watts.

Returns

The current power level in Watts.

7.2.1.3 apex_custom_event()

```
void apex_custom_event (
            apex_event_type event_type,
            void * custom_data )
```

Trigger a custom event.

This function will pass a custom event to the APEX event listeners. Each listeners' custom event handler will handle the custom event. Policy functions will be passed the custom event name in the event context.

Parameters

event_type	The type of the custom event
custom_data	Data specific to the custom event

Returns

No return value.

See also

apex_register_custom_event

7.2.1.4 apex_deregister_policy()

```
void apex_deregister_policy (
            apex_policy_handle * handle )
```

Deregister a policy with APEX.

This function will deregister the specified policy. In order to enable the policy again, it should be registered using apex_register_policy or apex_register_periodic_policy.

Parameters

handle	The handle of the policy to be deregistered.

See also

apex_register_policy, apex_register_periodic_policy

7.2.1.5 apex_dump()

```
const char* apex_dump (
            bool reset )
```

Dump output from APEX.

The stop measurement method will terminate all measurement and optionally:

- print a report to the screen
- write a profile to disk (if requested)
- output all other visualization data

Parameters

reset	Flag indicating whether to reset all statistics (true) or not (false).

Returns

a string containing the output.

See also

apex::finalize

7.2.1.6 apex_exit_thread()

```
void apex_exit_thread (
            void  )
```

Exit a thread.

For multithreaded applications, exit this thread and clean up.

Warning

Failure to exit a thread with APEX may invalidate statistics.

Returns

No return value.

7.2.1.7 apex_finalize()

```
void apex_finalize ( )
```

Finalize APEX.

The stop measurement method will terminate all measurement and optionally:

- print a report to the screen (if requested)
- write a profile to disk (if requested)

Returns

No return value.

See also

apex_init

7.2.1.8 apex_get_profile()

```
apex_profile* apex_get_profile (
            apex_profiler_type type,
            void * identifier )
```

Get the current profile for the specified id.

This function will return the current profile for the specified profiler id. Because profiles are updated out-of-band, it is possible that this profile value is out of date. This profile can be either a timer or a sampled value.

Parameters

type	The type of the address to be returned. This can be one of the apex_profiler_type values.
identifier	The function address of the function to be returned, or a "const char *" pointer to the name of the timer / counter.

Returns

The current profile for that timed function or sampled value.

7.2.1.9 apex_get_thread_cap()

```
int apex_get_thread_cap (
            void  )
```

Get the current thread cap set by the throttling.

This function will return the current thread cap based on the throttling policy.

Returns

The current thread cap value.

7.2.1.10 apex_hardware_concurrency()

```
uint64_t apex_hardware_concurrency (
            void  )
```

Get the number of possible threads supported on this system.

This method queries the system to determine how many threads can be executed concurrently without oversubscription.

Returns

number of hardware threads available

7.2.1.11 apex_init()

```
int apex_init (
            const char * thread_name,
            const uint64_t comm_rank,
            const uint64_t comm_size )
```

Intialize APEX.

Warning

For best results, this function should be called before any other APEX functions.

Use this version of apex_init when you do not have access to the input arguments.

Parameters

thread_name	The name of the thread, or nullptr. The lifetime of the thread will be timed with a timer using this same name.
comm_rank	The rank of this process within the full distributed application, i.e. MPI rank or HPX locality.
comm_size	The total number of processes within the full distributed application, i.e. MPI comm_size or total number of HPX localities.

Returns

APEX_NOERROR on success, or APEX_ERROR on failure.

See also

apex_finalize

7.2.1.12 apex_new_task()

```
void apex_new_task (
            apex_profiler_type type,
            void * identifier,
            uint64_t task_id )
```

Create a new task (dependency).

This function will note a task dependency between the current timer (task) and the new task.

Parameters

type	The type of the address to be reset. This can be one of the apex_profiler_type values.
identifier	The function address of the function of the task, or a "const char *" pointer to the name of the task.
task_id	The ID of the task

Returns

No return value.

7.2.1.13 apex_print_options()

```
void apex_print_options (
            void  )
```

Print the current APEX settings.

This function will print all the current APEX settings.

7.2.1.14 apex_recv()

```
void apex_recv (
            uint64_t tag,
            uint64_t size,
            uint64_t source_rank,
            uint64_t source_thread )
```

Notify APEX that the current thread is receiving a parcel/message from another rank/locality/process.

This method notifies APEX that the current thread is receiving a parcel/message from another rank/locality/process. The tag is meant to be an identifier for the message, not required to be unique. The source value is the APEX rank of the source of the message.

Parameters

tag	The message identifier
size	The message size (in bytes)
source_rank	The message source (as a rank/locality index)
source_thread	The message source (as a worker thread index)

7.2.1.15 apex_register_custom_event()

```
apex_event_type apex_register_custom_event (
            const char * name )
```

Register an event type with APEX.

Create a user-defined event type for APEX.

Parameters

name	The name of the custom event

Returns

The index of the custom event.

See also

apex_custom_event

7.2.1.16 apex_register_periodic_policy()

```
apex_policy_handle* apex_register_periodic_policy (
            unsigned long period,
            apex_policy_function f )
```

Register a policy with APEX.

Apex provides the ability to call an application-specified function periodically. This assigns the passed in function to be called on a periodic basis. The context for the event will be passed to the registered function.

Parameters

period	How frequently the function should be called
f	The function to be called when that event is handled by APEX.

Returns

A handle to the policy, to be stored if the policy is to be un-registered later.

See also

apex_deregister_policy, apex_register_policy

7.2.1.17 apex_register_policy()

```
apex_policy_handle* apex_register_policy (
            const apex_event_type when,
            apex_policy_function f )
```

Register a policy with APEX.

Apex provides the ability to call an application-specified function when certain events occur in the APEX library, or periodically. This assigns the passed in function to the event, so that when that event occurs in APEX, the function is called. The context for the event will be passed to the registered function.

Parameters

when	The APEX event when this function should be called
f	The function to be called when that event is handled by APEX.

Returns

A handle to the policy, to be stored if the policy is to be un-registered later.

See also

apex_deregister_policy, apex_register_periodic_policy

7.2.1.18 apex_register_thread()

```
void apex_register_thread (
            const char * name )
```

Register a new thread.

For multithreaded applications, register a new thread with APEX.

Warning

Failure to register a thread with APEX may invalidate statistics, and may prevent the ability to use timers or sampled values for this thread.

Parameters

name	The name that will be assigned to the new thread.

Returns

No return value.

7.2.1.19 apex_reset()

```
void apex_reset (
            apex_profiler_type type,
            void * identifier )
```

Reset a timer or counter.

This function will reset the profile associated with the specified timer or counter id to zero.

Parameters

type	The type of the address to be reset. This can be one of the apex_profiler_type values.
identifier	The function address of the function to be reset, or a "const char *" pointer to the name of the timer / counter.

Returns

No return value.

See also

apex_get_profile

7.2.1.20 apex_resume()

```
apex_profiler_handle apex_resume (
            apex_profiler_type type,
            void * identifier )
```

Resume a timer.

This function will create a profiler object in APEX, and return a handle to the object. The object will be associated with the name and/or function address passed in to this function. The difference between this function and the apex_start function is that the number of calls to that timer will not be incremented.

Parameters

type	The type of the address to be stored. This can be one of the apex_profiler_type values.
identifier	The function address of the function to be timed, or a "const char *" pointer to the name of the timer.

Returns

The handle for the timer object in APEX. Not intended to be queried by the application. Should be retained locally, if possible, and passed in to the matching apex_stop call when the timer should be stopped.

See also

apex_start, apex_stop, apex_yield

7.2.1.21 apex_resume_guid()

```
apex_profiler_handle apex_resume_guid (
            apex_profiler_type type,
            void * identifier,
            uint64_t guid )
```

Resume a timer.

This function will create a profiler object in APEX, and return a handle to the object. The object will be associated with the name and/or function address passed in to this function. The difference between this function and the apex_start function is that the number of calls to that timer will not be incremented.

Parameters

type	The type of the address to be stored. This can be one of the apex_profiler_type values.
identifier	The function address of the function to be timed, or a "const char *" pointer to the name of the timer.
guid	A globally unique identifier for this task.

Returns

The handle for the timer object in APEX. Not intended to be queried by the application. Should be retained locally, if possible, and passed in to the matching apex_stop call when the timer should be stopped.

See also

apex_start, apex_stop, apex_yield

7.2.1.22 apex_sample_value()

```
void apex_sample_value (
            const char * name,
            double value )
```

Sample a state value.

This function will retain a sample of some value. The profile for this sampled value will store the min, mean, max, total and standard deviation for this value for all times it is sampled.

Parameters

name	The name of the sampled value
value	The sampled value

Returns

No return value.

7.2.1.23 apex_send()

```
void apex_send (
            uint64_t tag,
            uint64_t size,
            uint64_t target )
```

Notify APEX that the current thread is sending a parcel/message to another rank/locality/process.

This method notifies APEX that the current thread is sending a parcel/message to another rank/locality/process. The tag is meant to be an identifier for the message, not required to be unique. The target value is the APEX rank of the target of the message.

Parameters

tag	The message identifier
size	The message size (in bytes)
target	The message target (as an APEX rank)

7.2.1.24 apex_set_state()

```
void apex_set_state (
             apex_thread_state state )
```

Set the thread state.

This function will set the thread state in APEX for 3rd party observation

Parameters

state	The state of the thread.

Returns

No return value.

7.2.1.25 apex_set_thread_cap()

```
void apex_set_thread_cap (
             int new_cap )
```

Set the current thread cap for throttling.

This function will set the current thread cap based on an external throttling policy.

Parameters

new_cap	The current thread cap value.

7.2.1.26 apex_setup_power_cap_throttling()

```
int apex_setup_power_cap_throttling (
             void  )
```

Initialize the power cap throttling policy.

This function will initialize APEX for power cap throttling. There are several environment variables that control power cap throttling:

HPX_THROTTLING If set, throttling will be enabled and initialized at startup.

APEX_THROTTLING_MAX_THREADS The maximum number of threads the throttling system will allow. The default value is 48.

APEX_THROTTLING_MIN_THREADS The minimum number of threads the throttling system will allow. The default value is 12.

APEX_THROTTLING_MAX_WATTS The maximum number of Watts the system can consume as an average rate. The default value is 220.

APEX_THROTTLING_MIN_WATTS The minimum number of Watts the system can consume as an average rate. The default value is 180.

HPX_ENERGY_THROTTLING If set, power/energy throttling will be performed.

HPX_ENERGY TBD

After evaluating the state of the system, the policy will set the thread cap, which can be queried using apex_get_thread_cap().

Returns

APEX_NOERROR on success, otherwise an error code.

7.2.1.27 apex_setup_throughput_tuning()

```
int apex_setup_throughput_tuning (
            apex_profiler_type type,
            void * identifier,
            apex_optimization_criteria_t criteria,
            apex_event_type event_type,
            int num_inputs,
            long ** inputs,
            long * mins,
            long * maxs,
            long * steps )
```

Setup throttling to optimize for the specified function, using multiple input criteria.

This function will initialize a policy to optimize the specified function, using the list of tunable inputs for the specified function. The optimization criteria include maximizing throughput, minimizing or maximizing time spent in the specified function. After evaluating the state of the system, the policy will assign new values to the inputs.

Parameters

type	The type of the address to be optimized. This can be one of the apex_profiler_type values.
identifier	The function address of the function to be optimized, or a "const char $*$" pointer to the name of the counter/timer.
criteria	The optimization criteria.
event_type	The apex_event_type that should trigger this policy
num_inputs	The number of tunable inputs for optimization
inputs	An array of addresses to inputs for optimization
mins	An array of minimum values for each input
maxs	An array of maximum values for each input
steps	An array of step values for each input

Returns

APEX_NOERROR on success, otherwise an error code.

7.2.1.28 apex_setup_timer_throttling()

```
int apex_setup_timer_throttling (
            apex_profiler_type type,
            void * identifier,
            apex_optimization_criteria_t criteria,
            apex_optimization_method_t method,
            unsigned long update_interval )
```

Setup throttling to optimize for the specified function.

This function will initialize the throttling policy to optimize for the specified function. The optimization criteria include maximizing throughput, minimizing or maximizing time spent in the specified function. After evaluating the state of the system, the policy will set the thread cap, which can be queried using apex_get_thread_cap().

Parameters

type	The type of the address to be optimized. This can be one of the apex_profiler_type values.
identifier	The function address of the function to be optimized, or a "const char *" pointer to the name of the counter/timer.
criteria	The optimization criteria.
method	The optimization method.
update_interval	The time between observations, in microseconds.

Returns

APEX_NOERROR on success, otherwise an error code.

7.2.1.29 apex_shutdown_throttling()

```
int apex_shutdown_throttling (
            void  )
```

Terminate the throttling policy.

This function will terminate the throttling policy.

Returns

APEX_NOERROR on success, otherwise an error code.

7.2.1.30 apex_start()

```
apex_profiler_handle apex_start (
            apex_profiler_type type,
            void * identifier )
```

Start a timer.

This function will create a profiler object in APEX, and return a handle to the object. The object will be associated with the address or name passed in to this function. If both are zero (null) then the call will fail and the return value will be null.

Parameters

type	The type of the address to be stored. This can be one of the apex_profiler_type values.
identifier	The function address of the function to be timed, or a "const char *" pointer to the name of the timer.

Returns

The handle for the timer object in APEX. Not intended to be queried by the application. Should be retained locally, if possible, and passed in to the matching apex_stop call when the timer should be stopped.

See also

apex_stop, apex_resume, apex_yield

7.2.1.31 apex_start_guid()

```
apex_profiler_handle apex_start_guid (
            apex_profiler_type type,
            void * identifier,
            uint64_t guid )
```

Start a timer.

This function will create a profiler object in APEX, and return a handle to the object. The object will be associated with the address or name passed in to this function. If both are zero (null) then the call will fail and the return value will be null.

Parameters

type	The type of the address to be stored. This can be one of the apex_profiler_type values.
identifier	The function address of the function to be timed, or a "const char *" pointer to the name of the timer.
guid	A globally unique identifier for this task.

Returns

The handle for the timer object in APEX. Not intended to be queried by the application. Should be retained locally, if possible, and passed in to the matching apex_stop call when the timer should be stopped.

See also

apex_stop, apex_resume, apex_yield

7.2.1.32 apex_stop()

```
void apex_stop (
            apex_profiler_handle profiler )
```

Stop a timer.

This function will stop the specified profiler object, and queue the profiler to be processed out-of-band. The timer value will eventually added to the profile for the process.

Parameters

profiler	The handle of the profiler object.

Returns

No return value.

See also

apex_start, apex_yield, apex_resume

7.2.1.33 apex_version()

```
const char* apex_version (
            void  )
```

Return the APEX version.

Returns

A character string with the APEX version. This string should not be freed after the calling function is done with it.

7.2.1.34 apex_yield()

```
void apex_yield (
            apex_profiler_handle profiler )
```

Stop a timer, but don't increment the number of calls.

This function will stop the specified profiler object, and queue the profiler to be processed out-of-band. The timer value will eventually added to the profile for the process. The number of calls will NOT be incremented - this "task" was yielded, not completed. It will be resumed by another thread at a later time.

Parameters

profiler	The handle of the profiler object.

Returns

No return value.

See also

apex_start, apex_stop, apex_resume

7.3 /Users/khuck/src/xpress-apex/src/apex/apex_api.hpp File Reference

Classes

- class apex::scoped_timer
  A convenience class for using APEX in C++ applications.
- class apex::scoped_thread
  A convenience class for creating a scoped thread.

Namespaces

- apex
  The main APEX namespace.

Macros

- #define APEX_SCOPED_TIMER
  A convenient macro for inserting an APEX self-stopping timer.

Functions

- static std::shared_ptr< task_wrapper > apex::null_task_wrapper (nullptr)
  declare a default "null" pointer for std::shared_ptr<task_wrapper>& references
- uint64_t apex::init (const char *thread_name, const uint64_t comm_rank, const uint64_t comm_size)
  Intialize APEX.
- std::string apex::dump (bool reset)
  Dump output from APEX.
- void apex::finalize (void)
  Finalize APEX.
- void apex::cleanup (void)
  Cleanup APEX.
- profiler * apex::start (const std::string &timer_name)
  Start a timer.
- profiler * apex::start (const apex_function_address function_address)
  Start a timer.

- void apex::start (std::shared_ptr< task_wrapper > task_wrapper_ptr)

  Start a timer.
- void apex::stop (profiler *the_profiler, bool cleanup=true)

  Stop a timer.
- void apex::stop (std::shared_ptr< task_wrapper > task_wrapper_ptr)

  Stop a timer.
- void apex::yield (profiler *the_profiler)

  Stop a timer, but don't increment the number of calls.
- void apex::yield (std::shared_ptr< task_wrapper > task_wrapper_ptr)

  Stop a timer, but don't increment the number of calls.
- profiler * apex::resume (const std::string &timer_name)

  Resume a timer.
- profiler * apex::resume (const apex_function_address function_address)

  Resume a timer.
- void apex::resume (std::shared_ptr< task_wrapper > task_wrapper_ptr)

  Resume a timer.
- void apex::reset (const std::string &timer_name)

  Reset a timer or counter.
- void apex::reset (apex_function_address function_address)

  Reset a timer.
- void apex::set_state (apex_thread_state state)

  Set the thread state.
- void apex::sample_value (const std::string &name, double value)

  Sample a state value.
- std::shared_ptr< task_wrapper > apex::new_task (const std::string &name, const uint64_t task_id=UINT↩ MAX_MAX, const std::shared_ptr< apex::task_wrapper > parent_task=null_task_wrapper)

  Create a new task (dependency).
- std::shared_ptr< task_wrapper > apex::new_task (const apex_function_address function_address, const uint64_t task_id=UINTMAX_MAX, const std::shared_ptr< apex::task_wrapper > parent_task=null_task_↩ wrapper)

  Create a new task (dependency).
- std::shared_ptr< task_wrapper > apex::update_task (std::shared_ptr< task_wrapper > wrapper, const std↩ ::string &name)

  Update a task (dependency).
- std::shared_ptr< task_wrapper > apex::update_task (std::shared_ptr< task_wrapper > wrapper, const apex_function_address function_address)

  Update a task wrapper (dependency).
- apex_event_type apex::register_custom_event (const std::string &name)

  Register an event type with APEX.
- void apex::custom_event (apex_event_type event_type, void *custom_data)

  Trigger a custom event.
- std::string & apex::version (void)

  Return the APEX version.
- void apex::register_thread (const std::string &name)

  Register a new thread.
- void apex::exit_thread (void)

  Exit a thread.
- apex_policy_handle * apex::register_policy (const apex_event_type when, std::function< int(apex_context const &)> f)

  Register a policy with APEX.
- std::set< apex_policy_handle * > apex::register_policy (std::set< apex_event_type > when, std::function< int(apex_context const &)> f)

Register a policy with APEX.

- apex_policy_handle * apex::register_periodic_policy (unsigned long period, std::function< int(apex_context const &)> f)

  Register a policy with APEX.

- apex_policy_handle * apex::sample_runtime_counter (unsigned long period, const std::string &counter_↩ name)

  Periodically sample a runtime counter.

- void apex::deregister_policy (apex_policy_handle *handle)

  Deregister a policy with APEX.

- void apex::stop_all_async_threads (void)

  Stop all asynchronous APEX background threads.

- apex_profile * apex::get_profile (apex_function_address function_address)

  Get the current profile for the specified function address.

- apex_profile * apex::get_profile (const std::string &timer_name)

  Get the current profile for the specified function address.

- int apex::setup_power_cap_throttling (void)

  Initialize the power cap throttling policy.

- int apex::setup_timer_throttling (apex_function_address the_address, apex_optimization_criteria_t criteria, apex_optimization_method_t method, unsigned long update_interval)

  Setup throttling to optimize for the specified function.

- int apex::setup_throughput_tuning (apex_function_address the_address, apex_optimization_criteria_t criteria, apex_event_type event_type, int num_inputs, long **inputs, long *mins, long *maxs, long *steps)

  Setup throttling to optimize for the specified function, using multiple input criteria.

- apex_tuning_session_handle apex::setup_custom_tuning (std::function< double(void)> metric, apex_event_type event_type, int num_inputs, long **inputs, long *mins, long *maxs, long *steps)

  Setup tuning of specified parameters to optimize for a custom metric, using multiple input criteria.

- apex_tuning_session_handle apex::setup_custom_tuning (apex_tuning_request &request)

  Setup tuning of specified parameters to optimize for a custom metric, using multiple input criteria of potentially multiple types.

- int apex::setup_timer_throttling (const std::string &the_name, apex_optimization_criteria_t criteria, apex_optimization_method_t method, unsigned long update_interval)

  Setup throttling to optimize for the specified function or counter.

- int apex::shutdown_throttling (void)

  Terminate the throttling policy.

- int apex::get_thread_cap (void)

  Get the current thread cap set by the throttling.

- void apex::set_thread_cap (int new_cap)

  Set the current thread cap for throttling.

- std::vector< std::pair< std::string, long * > > & apex::get_tunable_params (apex_tuning_session_handle h)

  Return a vector of the current tunable parameters.

- bool apex::has_session_converged (apex_tuning_session_handle handle)

  Check whether a tuning session has converged.

- void apex::get_best_values (apex_tuning_session_handle h)

  Set a tuning session's values to the best known values.

- void apex::print_options (void)

  Print out all configuration settings for APEX.

- void apex::send (uint64_t tag, uint64_t size, uint64_t target)

  Notify APEX that the current thread is sending a parcel/message to another rank/locality/process.

- void apex::recv (uint64_t tag, uint64_t size, uint64_t source_rank, uint64_t source_thread)

  Notify APEX that the current thread is receiving a parcel/message from another rank/locality/process.

7.3.1 Macro Definition Documentation

7.3.1.1 APEX_SCOPED_TIMER

```
#define APEX_SCOPED_TIMER
```

Value:

```
std::ostringstream _s_foo; \
_s_foo « __func__ « " [" « __FILE__ « ":" « __LINE__ « "]"; \
apex::scoped_timer __foo(_s_foo.str());
```

A convenient macro for inserting an APEX self-stopping timer.

This macro will create a timer using the values of **func**, **LINE** and **FILE** from the preprocessor.

7.4 /Users/khuck/src/xpress-apex/src/apex/apex_types.h File Reference

```
#include <stdint.h>
#include <stdbool.h>
#include <unistd.h>
```

Classes

- struct _policy_handle
- struct _context
- struct _profile

Macros

- #define APEX_NULL_PROFILER_HANDLE (apex_profiler_handle)(nullptr)
- #define APEX_MAX_EVENTS 128
- #define APEX_NULL_FUNCTION_ADDRESS 0L
- #define APEX_IDLE_TIME "APEX Idle"
- #define APEX_NON_IDLE_TIME "APEX Non-Idle"
- #define APEX_IDLE_RATE "APEX Idle Rate"
- #define APEX_DEFAULT_OTF2_ARCHIVE_PATH "OTF2_archive"
- #define APEX_DEFAULT_OTF2_ARCHIVE_NAME "APEX"

Typedefs

- typedef void * apex_profiler_handle
- typedef uintptr_t apex_function_address
- typedef int(* apex_policy_function) (apex_context const context)
- typedef uint32_t apex_tuning_session_handle

Enumerations

- enum _apex_profiler_type { APEX_FUNCTION_ADDRESS = 0, APEX_NAME_STRING }
- enum _error_codes { APEX_NOERROR = 0, APEX_ERROR }
- enum _event_type {
  APEX_INVALID_EVENT = -1, APEX_STARTUP = 0, APEX_SHUTDOWN, APEX_DUMP,
  APEX_RESET, APEX_NEW_NODE, APEX_NEW_THREAD, APEX_EXIT_THREAD,
  APEX_START_EVENT, APEX_RESUME_EVENT, APEX_STOP_EVENT, APEX_YIELD_EVENT,
  APEX_SAMPLE_VALUE, APEX_SEND, APEX_RECV, APEX_PERIODIC,
  APEX_CUSTOM_EVENT_1, **APEX_CUSTOM_EVENT_2**, **APEX_CUSTOM_EVENT_3**, **APEX_CUSTOM_EVENT_4**,
  APEX_CUSTOM_EVENT_5, **APEX_CUSTOM_EVENT_6**, **APEX_CUSTOM_EVENT_7**, **APEX_CUSTOM_EVENT_8**,
  APEX_UNUSED_EVENT = APEX_MAX_EVENTS }
- enum _thread_state {
  APEX_IDLE, APEX_BUSY, APEX_THROTTLED, APEX_WAITING,
  APEX_BLOCKED }
- enum apex_optimization_criteria_t { APEX_MAXIMIZE_THROUGHPUT, APEX_MAXIMIZE_ACCUMULATED, APEX_MINIMIZE_ACCUMULATED }
- enum apex_optimization_method_t { APEX_SIMPLE_HYSTERESIS, APEX_DISCRETE_HILL_CLIMBING, APEX_ACTIVE_HARMONY }
- enum _profile_type { APEX_TIMER, APEX_COUNTER }

7.4.1 Class Documentation

7.4.1.1 struct _policy_handle

A reference to the policy object, so that policies can be "unregistered", or paused later

Class Members

apex_event_type	event_type	The type of policy
int	id	The ID of the policy, used internally to APEX
unsigned long	period	If periodic, the length of the period

7.4.1.2 struct _context

The APEX context when an event occurs.

Class Members

void *	data	Data associated with the event, such as the custom_data for a custom_event
apex_event_type	event_type	The type of the event currently processing
apex_policy_handle *	policy_handle	The policy handle for the current policy function

7.4.1.3 struct _profile

The profile object for a timer in APEX.

Class Members

double	accumulated	Accumulated values for all calls/samples
double	calls	Number of times a timer was called, or the number of samples collected for a counter
double	maximum	Maximum value seen by the timer or counter
double	minimum	Minimum value seen by the timer or counter
double	papi_metrics[8]	Array of accumulated PAPI hardware metrics
double	sum_squares	Running sum of squares calculation for all calls/samples
int	times_reset	How many times was this timer reset
apex_profile_type	type	Whether this is a timer or a counter

7.4.2 Macro Definition Documentation

7.4.2.1 APEX_DEFAULT_OTF2_ARCHIVE_NAME

```
#define APEX_DEFAULT_OTF2_ARCHIVE_NAME "APEX"
```

Default OTF2 trace name

7.4.2.2 APEX_DEFAULT_OTF2_ARCHIVE_PATH

```
#define APEX_DEFAULT_OTF2_ARCHIVE_PATH "OTF2_archive"
```

Default OTF2 trace path

7.4.2.3 APEX_IDLE_RATE

```
#define APEX_IDLE_RATE "APEX Idle Rate"
```

Special profile counter for derived idle rate

7.4.2.4 APEX_IDLE_TIME

```
#define APEX_IDLE_TIME "APEX Idle"
```

Special profile counter for derived idle time

7.4.2.5 APEX_MAX_EVENTS

```
#define APEX_MAX_EVENTS 128
```

The maximum number of event types. Allows for many custom events.

7.4.2.6 APEX_NON_IDLE_TIME

```
#define APEX_NON_IDLE_TIME "APEX Non-Idle"
```

Special profile counter for derived non-idle time

7.4.2.7 APEX_NULL_FUNCTION_ADDRESS

```
#define APEX_NULL_FUNCTION_ADDRESS 0L
```

A null pointer representing an APEX function address. Used when a null APEX function address is to be passed in to any apex functions to represent "all functions".

7.4.2.8 APEX_NULL_PROFILER_HANDLE

```
#define APEX_NULL_PROFILER_HANDLE (apex_profiler_handle)(nullptr)
```

A null pointer representing an APEX profiler handle. Used when a null APEX profile handle is to be passed in to apex::stop when the profiler object wasn't retained locally.

7.4.3 Typedef Documentation

7.4.3.1 apex_function_address

```
typedef uintptr_t apex_function_address
```

Rather than use void pointers everywhere, be explicit about what the functions are expecting.

7.4.3.2 apex_policy_function

```
typedef int(* apex_policy_function) (apex_context const context)
```

Rather than use void pointers everywhere, be explicit about what the functions are expecting.

7.4.3.3 apex_profiler_handle

```
typedef void* apex_profiler_handle
```

The address of a C++ object in APEX. Not useful for the caller that gets it back, but required for stopping the timer later.

7.4.3.4 apex_tuning_session_handle

```
typedef uint32_t apex_tuning_session_handle
```

A handle to a tuning session.

7.4.4 Enumeration Type Documentation

7.4.4.1 _apex_profiler_type

```
enum _apex_profiler_type
```

Enumerator

APEX_FUNCTION_ADDRESS	The ID is a function (or instruction) address
APEX_NAME_STRING	The ID is a character string

7.4.4.2 _error_codes

`enum _error_codes`

Typedef for enumerating the different event types

Enumerator

APEX_NOERROR	No error occurred
APEX_ERROR	Some error occurred - check stderr output for details

7.4.4.3 _event_type

`enum _event_type`

Enumerator

APEX_STARTUP	APEX is initialized
APEX_SHUTDOWN	APEX is terminated
APEX_DUMP	APEX is dumping output
APEX_RESET	APEX is resetting data structures
APEX_NEW_NODE	APEX has registered a new process ID
APEX_NEW_THREAD	APEX has registered a new OS thread
APEX_EXIT_THREAD	APEX has exited an OS thread
APEX_START_EVENT	APEX has processed a timer start event
APEX_RESUME_EVENT	APEX has processed a timer resume event (the number of calls is not incremented)
APEX_STOP_EVENT	APEX has processed a timer stop event
APEX_YIELD_EVENT	APEX has processed a timer yield event
APEX_SAMPLE_VALUE	APEX has processed a sampled value
APEX_SEND	APEX has processed a send event
APEX_RECV	APEX has processed a recv event
APEX_PERIODIC	APEX has processed a periodic timer
APEX_CUSTOM_EVENT_1	APEX has processed a custom event - useful for large granularity application control events

7.4.4.4 _profile_type

enum _profile_type

The type of a profiler object

Enumerator

APEX_TIMER	This profile is a instrumented timer
APEX_COUNTER	This profile is a sampled counter

7.4.4.5 _thread_state

enum _thread_state

Typedef for enumerating the thread states.

Enumerator

APEX_IDLE	Thread is idle
APEX_BUSY	Thread is working
APEX_THROTTLED	Thread is throttled (sleeping)
APEX_WAITING	Thread is waiting for a resource
APEX_BLOCKED	Thread is blocked

7.4.4.6 apex_optimization_criteria_t

enum apex_optimization_criteria_t

Typedef for enumerating the different optimization strategies for throttling.

Enumerator

APEX_MAXIMIZE_THROUGHPUT	maximize the number of calls to a timer/counter
APEX_MAXIMIZE_ACCUMULATED	maximize the accumulated value of a timer/counter
APEX_MINIMIZE_ACCUMULATED	minimize the accumulated value of a timer/counter

7.4.4.7 apex_optimization_method_t

enum apex_optimization_method_t

Typedef for enumerating the different optimization methods for throttling.

Enumerator

APEX_SIMPLE_HYSTERESIS	optimize using sliding window of historical observations. A running average of the most recent N observations are used as the measurement.
APEX_DISCRETE_HILL_CLIMBING	Use a discrete hill climbing algorithm for optimization
APEX_ACTIVE_HARMONY	Use Active Harmony for optimization.

7.5 /Users/khuck/src/xpress-apex/src/comm/apex_global.h File Reference

```
#include "apex.h"
```

Functions

- int action_apex_reduce (void *unused)

  the function declaration, this is the funcion that does the reduction
- int action_apex_get_value (void *args)

  Each node has to populate their local value.
- int apex_periodic_policy_func (apex_context const context)

  A policy function to do periodic output.
- void apex_global_setup (apex_profiler_type type, void *in_action)

  The function to set up global reductions.
- void apex_global_teardown (void)

  The function to tear down global reductions, if necessary.

7.5.1 Function Documentation

7.5.1.1 action_apex_get_value()

```
int action_apex_get_value (
            void * args )
```

Each node has to populate their local value.

Parameters

args	Local data to be exchanged globally.

Returns

0 on no error.

7.5.1.2 action_apex_reduce()

```
int action_apex_reduce (
            void * unused )
```

the function declaration, this is the funcion that does the reduction

Parameters

unused	Unused value.

Returns

0 on no error.

7.5.1.3 apex_global_setup()

```
void apex_global_setup (
            apex_profiler_type type,
            void * in_action )
```

The function to set up global reductions.

Parameters

type	The type of the profiler
in_action	The name of a timer or address of a function. This is the function timer that should be reduced globally. This value is used for the example.

Returns

0 on no error.

7.5.1.4 apex_periodic_policy_func()

```
int apex_periodic_policy_func (
            apex_context const context )
```

A policy function to do periodic output.

Parameters

context	The context for the periodic policy.

Returns

0 on no error.

Index