HPX Performance Counters

HARTMUT KAIser (LSU), MACIEJ BRODOWICZ (IU)

HKAIsER@CCT.LSU.EDU, MBRODOWI@INDIANA.EDU
What are HPX Performance Counters

- A performance counter is the representation of an arbitrary data source (values are currently int64_t)
- Each performance counter has some descriptive metadata associated
- Accessible through a uniform interface
- Embedded in a framework allowing to discover, create, and query available performance counter types and instances
- Relies on HPX component framework
  - Easy to extend with application specific counters
  - Counters can be discovered, instantiated and queried locally and remotely, no code difference
- Default command line options for simple analysis tasks
- Full blown API to embed counters into application
struct counter_value
{
    uint64_t time_; // The local time when data was collected
    uint64_t count_; // The invocation counter for the data
    int64_t value_; // The current counter value
    int64_t scaling_; // The scaling of the current counter value
    counter_status status_; // The status of the counter value
    bool scale_inverse_; // If true, value_ needs to be divided by
                          // scaling_, otherwise it has to be multiplied.
};
struct counter_info
{
    counter_type type_;  // The type of the described counter
    uint32_t version_;  // The version of the described counter
    // using the 0xMMmmSSSS scheme
    counter_status status_; // The status of the counter object
    std::string fullname_; // The full name of this counter
    std::string helptext_; // The full descriptive text for this counter
    std::string uom_; // The unit of measure for this counter
};
struct performance_counter
{
    // Retrieve the descriptive information about the Performance Counter.
    counter_info get_counter_info() const;
    // Retrieve the current Performance Counter value.
    counter_value get_counter_value();
    // Reset the Performance Counter (value).
    void reset_counter_value();
    // Set the (initial) value of the Performance Counter.
    void set_counter_value(counter_value const& value);
    bool start(); // Start the Performance Counter.
    bool stop();  // Stop the Performance Counter.
};
Performance Counter Names

- Performance counter names are strings and have 2 parts
  - Counter type name
    - `/threads/count/cumulative`, `/threads/count/instantaneous/pending`
  - Counter instance name
    - `locality#0/total`, `locality#0/worker-thread#1`
- Both parts are combined in order to reference a particular counter instance:
  - `/threads{locality#0/total}/count/cumulative`
- General format is:
  - `/objectname{fullinstancename}/counternam@parameters`
  - Where `fullinstancename` is
    - `parentinstancename#parentindex/instancename#instanceindex`
    - Another full counter name
Discover Counters

// Call the supplied function for each registered counter type.
void discover_counter_types(
    std::function<bool(counter_info const&, error_code&)> discover_func,
    discover_counters_mode mode = discover_counters_minimal,
    error_code& ec = throws);

// Call the supplied function for the given registered counter type.
void discover_counter_type(std::string name,
    std::function<bool(counter_info const&, error_code&)> discover_func,
    discover_counters_mode mode = discover_counters_minimal,
    error_code& ec = throws);
// Get the global id of an existing performance counter, if the counter does not exist yet, the function attempts to create the counter based on the given counter info.
future<id_type> get_counter_async(counter_info info, error_code& ec = throws);

// same, but ‘synchronous’
id_type get_counter(counter_info info, error_code& ec = throws);
// Maintenance
counter_info get_info(id_type id, error_code& ec = throws);
counter_value get_value(id_type id, error_code& ec = throws);
bool start(id_type id, error_code& ec = throws);
bool stop(id_type id, error_code& ec = throws);

// Get current counter value
counter_value get_value(id_type id, error_code& ec = throws);
Creating New Counters

// Counter ‘data’
std::atomic<int64_t> counter(0);

// Performance counter function
int64_t some_performance_data()
{
    return ++counter;
}

void register_counter_type()
{
    // Call the HPX API function to register the counter type.
    install_counter_type("/test/data", 
        &some_performance_data, 
        "returns a linearly increasing counter value" 
        );
}

Any HPX application provides a set of default command line options to query performance counters:

- List all available counters
  - `--hpx:list-counters`
- List extended info for all available counters
  - `--hpx:list-counter-infos`
- Print given counter(s) at program exit
  - `--hpx:print-counter=<counter name>`
- Print given counter(s) at given time intervals
  - `--hpx:print-counter-interval=500` (milliseconds)
- Redirect counter output to given file (instead of console)
  - `--hpx:print-counter-destination=<filename>`
Command Line Example

hello_world -t4 \
   --hpx:print-counter=/threads{locality#0/worker-thread#*}/count/cumulative

hello world from OS-thread 0 on locality 0
hello world from OS-thread 3 on locality 0
hello world from OS-thread 1 on locality 0
hello world from OS-thread 2 on locality 0
/threads{locality#0/worker-thread#0}/count/cumulative,1,0.041645[s],37
/threads{locality#0/worker-thread#1}/count/cumulative,1,0.041763[s],26
/threads{locality#0/worker-thread#2}/count/cumulative,1,0.042001[s],45
/threads{locality#0/worker-thread#3}/count/cumulative,1,0.042284[s],40
Example:
PAPI Performance Counters
HPX PAPI Counters: Features

- Instrumentation per OS-thread
  - Target thread selection through counter name
  - Dynamic counter addition and removal at runtime
  - Different counter sets for each thread possible
- Supports all PAPI event types (presets and native)
- Transparent access to hardware counters
- Domain selection (user, kernel, …)
- Start/stop functionality with low precious resource footprint
  - No additional hardware registers consumed by stopped counters
- Available event listing as command line options
  - Compatible with `--hpx:list-counters full` and `minimal` options
  - Additional variants controlled via `--papi-event-info`
HPX PAPI Counters: Installation

- Install PAPI library (or use system version if available)
  - Download the recent version of the sources from http://icl.cs.utk.edu/PAPI/
  - Configure, compile, and install the library
- Configure HPX sources with cmake
  - PAPI installation in typical system paths is autodetected
  - If that fails, pass the root of PAPI installation as command line option to cmake: "]-DPAPI_ROOT=<dir>"
- Compile and install HPX
  - "make papi_counters_component" must be invoked explicitly
  - Install the libraries via "make install"
  - Verify that libpapi_counters.so* files are present in lib/hpx subdirectory under installation root
- Counters are now ready to use with any HPX application
Using --hpx:list-counter-info

```
[-]
/papi{locality#*/io-thread#*}/PAPI_TOT_CYC
/papi{locality#*/main-thread#*}/PAPI_TOT_CYC
/papi{locality#*/parcel-thread-tcp*}/PAPI_TOT_CYC
/papi{locality#*/timer-thread#*}/PAPI_TOT_CYC
/papi{locality#*/worker-thread#*}/PAPI_TOT_CYC
/papi{locality#*/io-thread#*}/PAPI_LST_INS
/papi{locality#*/main-thread#*}/PAPI_LST_INS
```

Using --papi-event-info=all (both preset and native event information)

```
[-]
---------------------------------------------------------------
Event         : ix86arch::LLC_MISSES
Type          : native
Code          : 0x4000003d
Registers     : -
Description   : count each cache miss condition for references to the last level cache. The event count may include speculation, but excludes cache line fills due to hardware prefetch
---------------------------------------------------------------
```
PAPI Counters Usage Examples

- Count all cache misses and data TLB misses in the first worker thread encountered in both user and kernel space (domain=all)
  ```
  fibonacci --papi-domain=all --hpx:print-counter="/papi{locality#0/worker-thread#0}/PAPI_L1_TCM" \
  --hpx:print-counter="/papi{locality#0/worker-thread#0}/PAPI_TLB_DM"
  fibonacci(10) == 55
  elapsed time: 0.00154779 [s]
  /papi{locality#0/worker-thread#0}/PAPI_L1_TCM,1,0.001780[s],232371
  /papi{locality#0/worker-thread#0}/PAPI_TLB_DM,1,0.001797[s],11230
  ```

- Count completed load and store instructions in worker thread 0, and number of conditional branches taken and not taken in worker thread 1 in user space (default domain)
  ```
  fibonacci --hpx:threads=2 --hpx:print-counter="/papi{locality#0/worker-thread#0}/PAPI_LST_INS" \
  --hpx:print-counter="/papi{locality#0/worker-thread#1}/PAPI_BR_TKN" \
  --hpx:print-counter="/papi{locality#0/worker-thread#1}/PAPI_BR_NTK"
  fibonacci(10) == 55
  elapsed time: 0.00320422 [s]
  /papi{locality#0/worker-thread#0}/PAPI_LST_INS,1,0.002972[s],860016
  /papi{locality#0/worker-thread#1}/PAPI_BR_TKN,1,0.002983[s],191069
  /papi{locality#0/worker-thread#1}/PAPI_BR_NTK,1,0.002997[s],85167
  ```
Open Issues

- No user-level thread monitoring
  - No good intuitive interface
  - Accumulation of values may substantially increase the overhead of context switch
- Not all events apply to threads
  - E.g., certain types of native “uncore” events
  - May require modification of naming scheme to better reflect event properties
- Limited number of hardware counters
  - Code supporting fixed interval multiplexing through PAPI library already implemented
  - Again, naming scheme change needed to avoid over-constraining the functionality
Concluding Remarks

- HPX documentation
  - http://stellar.cct.lsu.edu/files/hpx_0.9.5/html/index.html
- HPX source code
  - http://stellar.cct.lsu.edu/downloads/
  - https://github.com/STEllAR-GROUP/hpx