Using the PAPI Cray NPU Component

S–0046–10

RECORD OF REVISION

S–0046–10 Published March 2013 Supports PAPI release 5.1.02 running on Cray XE, Cray XK, and Cray XC30 systems.

Contents

Using the PAPI Cray NPU Component

With the release of PAPI 5.1.0.2, Cray now provides direct access to the Gemini™ and Aries™ network performance counters via the Performance API (PAPI). This document is intended for third-party software developers and users planning to write their own programs that will use PAPI functions to access the Gemini and Aries network counters.

This document assumes that you already have a good understanding of PAPI. For reference documentation and more information, see the PAPI website at http://icl.cs.utk.edu/papi/.

For more information about PAPI as implemented on Cray systems, see the `intro_papi`(3) and `papi_counters`(5) man pages.

1.1 Prerequisites

Because of the structure of the Cray system, any program running on a Cray must be launched using the `aprun` utility. This is required to ensure that the program runs on a compute node, and not on a service node or the native processor of an esLogin system. This is even more important when writing programs that use PAPI functions, as you want to be certain you are addressing the hardware counters in the compute node processors and Gemini or Aries network router chips, and not other hardware that may be unrelated to the nodes on which you run applications.

Therefore, before using PAPI, verify that the following modules are loaded:

- `craype-network-gemini` (Cray XE and Cray XKsystems)
- `craype-network-aries` (Cray XC30 systems)
- `craype-`*processor* (depending on the type of CPU used on your system's compute nodes)
- `PrgEnv-`*compiler* (depending on your choice of compiler)
- `papi/5.1.0.2` (or later)

Note: The `papi`, `perftools`, and `perftools-lite` modules are mutually exclusive. Therefore, in addition to verifying that the above modules **are** loaded, verify also that the `perftools` and `perftools-lite` modules are **not** loaded.

1.2 Enabling Access

Before using your own programs or tools to access network performance counters, you must set the `CRAY_NPU_ACCESS` environment variable. This environment variable is a bitmask that reflects your expected use of the network counters. If the network counters access bits are not set correctly, `PAPI_read` and `PAPI_write` will issue an error message, and these functions will return nonzero, indicating a failure.

The valid values are as follows.

`0x1` Requests access only to local processor tile counters on routers directly attached to the nodes of a job; places the job only on compute nodes where there is no reserved network performance counter access to the routers servicing those nodes. This guarantees that no router is attached directly to nodes from two such jobs that are running concurrently.

`0x2` Requests access to both local processor tile counters and network tile counters on routers directly attached to the nodes of a job; places the job only on compute nodes where there is no reserved network performance counter access to the routers servicing those nodes. This guarantees that no router is attached directly to nodes from two such jobs that are running concurrently.

`0x4` Requests access to all tile counters on all routers in the system; job is only placed if no other application has reserved access to any network tiles on the whole system (i.e. there can be other applications with reserved access to only local processor tile counters with the earlier constraint about local processor tiles). A job with this setting will not be run concurrently with any other job that sets `CRAY_NPU_ACCESS` to a value of `0x2` or `0x4`.

This environment variable must be set in order to ensure that the PAPI Cray NPU Component interacts as expected with ALPS, `aprun`, and `apsched`, and that the job requesting access to the network performance counters is placed on the system in a physical location that is minimally affected by other jobs' network traffic.

1.3 Programming Tips

The example programs in Examples on page 8 use simple C programs to illustrate how to use PAPI to address network counters directly. The best way to determine how to use the PAPI Cray NPU Component in your programs is by studying the example programs.

While developing code to access PAPI counters, bear the following tips in mind.

- The Gemini network router chips used in Cray XE and Cray XK systems have substantial differences from the Aries network router chips used in Cray XC30 systems. For a full list of the counters available for use by your program, load the `papi` module and use the `PAPI_native_avail` command.
- If the `perftools` module is loaded, you can find the list of Gemini counters in the `$CRAYPAT_ROOT/share/Counters.papi_gemini` and `$CRAYPAT_ROOT/share/Counters.papi_gemini.xml` files, or in the **counters→gemini** topics in the `pat_help` system.

  Likewise, the Aries counters are found in `$CRAYPAT_ROOT/share/Counters.papi_aries` or `$CRAYPAT_ROOT/share/Counters.papi_aries.xml`, or in the **counters→aries** topics in `pat_help`.

  Further information about PAPI utilities can be found in the `intro_papi`(3) man page. However, the `perftools`, `perftools-lite`, and `papi` modules are mutually exclusive; you may load one, but not two or three at the same time. Therefore, if you unload the `perftools` or `perftools-lite` module and load the `papi` module, the Cray-originated man pages, the `pat_help` command, and the `$CRAYPAT_ROOT` path will be unavailable. If you plan to use utilities or develop programs that use the PAPI Cray NPU Component, plan your work and save your reference information accordingly.
- In PAPI, most event names are quite long. When used with CrayPat, these event names are typically placed in a text file and passed in to CrayPat using the `PAT_RT_HWPC_FILE`, `PAT_RT_NWPC_FILE`, or `PAT_RT_ACCPC_FILE` environment variables. When writing code that uses PAPI calls directly, you will need to determine the best way to represent these lengthy event names in your code.
- When developing your PAPI program, remember that only one event set for a given component can be running at one time. While multiple event sets can be defined, only one can be active; that is, associated with a `PAPI_start..PAPI_stop` pair. If multiple event sets are activated at the same time, the resulting `PAPI_write` and `PAPI_read` calls are not guaranteed to produce valid results. The PAPI library does not prevent the user from implementing this behavior. `PAPI_OK` is still returned even if these conditions are violated.

- One outstanding difference between Gemini and Aries counters is that all Gemini counters are read-only, while some Aries counters are also writable. Accordingly, Accessing Aries Network Performance Counters, Configuring Three Events, and Reading Eight Events on page 10 shows how to both write and read Aries counters.
- For more information about the Aries counters, see *Using the Aries Hardware Counters* (S–0045).

1.4 Examples

The following examples demonstrate how to use PAPI in simple C programs to address network counters directly.

1.4.1 Accessing Gemini Network Performance Counters and Reading Two Events

Example 1. Gemini Counters: Reading Two Events

```
/*        Simple example of accessing Gemini network performance counters
 *        via PAPI. Collecting and checking all return values has been
 *        left out for clarity.
 */

#include <papi.h>
#include <stdio.h>
#include <string.h>

int
main (int argc, char *argv[ ])
{
        int i;
        int event_set = PAPI_NULL;
        int code = 0;
        long long values[2];
        char *events[ ] = {
                "GM_RMT_PERF_PUT_BYTES_RX",
                "GM_RMT_PERF_SEND_BYTES_RX",
                NULL
 };

        PAPI_library_init (PAPI_VER_CURRENT);
        PAPI_create_eventset (&event_set);

        for (i=0; events[i] != NULL; i++) {
                PAPI_event_name_to_code (events[i], &code);
                PAPI_add_event (event_set, code);
        }
```

```
        PAPI_start (event_set);

        /* perform work */

        memset (values, 0, sizeof(values));
        PAPI_stop (event_set, values);

        /* process values */
        for (i=0; events[i] != NULL; i++) {
                printf (" %s = %lld\n", events[i], values[i]);
        }

        PAPI_cleanup_eventset (event_set);
        PAPI_destroy_eventset (&event_set);
        PAPI_shutdown ( );

        return 0;
}
```

1.4.2 Accessing Aries Network Performance Counters and Reading Two Events

Example 2. Aries Counters: Reading Two Events

```
/*        Simple example of accessing Aries network performance counters
 *        via PAPI. Collecting and checking all return values has been
 *        left out for clarity.
 */

#include <papi.h>
#include <stdio.h>
#include <string.h>

int
main (int argc, char *argv[ ])
{

        int i;
        int code = 0;
        int event_set = PAPI_NULL;
        long long values[2];
        char *events[ ] = {
                "AR_NIC_SSID_PRF_CNTR_SSIDS_IN_USE:MAX_SSIDS_IN_USE",
                "AR_NIC_SSID_PRF_CNTR_SSIDS_IN_USE:CUR_SSIDS_IN_USE",
                NULL
        };

        PAPI_library_init (PAPI_VER_CURRENT);
        PAPI_create_eventset (&event_set);

        for (i=0; events[i] != NULL; i++) {
                PAPI_event_name_to_code (events[i], &code);
                PAPI_add_event (event_set, code);
        }
```

```
        PAPI_start (event_set);

        /* perform work */

        memset (values, 0, sizeof(values));
        PAPI_stop (event_set, values);

        /* process values */
        for (i=0; events[i] != NULL; i++) {
                printf (" %s = %lld\n", events[i], values[i]);
        }

        PAPI_cleanup_eventset (event_set);
        PAPI_destroy_eventset (&event_set);
        PAPI_shutdown ( );

        return 0;
}
```

1.4.3 Accessing Aries Network Performance Counters, Configuring Three Events, and Reading Eight Events

This example demonstrates how to use `PAPI_write` with Aries configuration events. Only one event set can be active at a time. This example uses the bin histogram events.

Example 3. Aries Counters: Configuring Three Events and Reading Eight Events

```
/*
 *      Copyright 2013 Cray Inc. All Rights Reserved.
 */

/*      Simple example of accessing Aries network performance counters
 *      via PAPI. This example configures the histogram bin counter.
 *      The CRAY_NPU_ACCESS permissions must be set accordingly.
 *      Only one event set can be running (PAPI_start..PAPI_stop) at a time.
 *      Collecting and checking all return values has been left out for
 *      clarity.
 */

#include <papi.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

int
main (int argc, char *argv[ ])
{
        int i;
        int code = 0;
        int cidx;
        int nctrs;
        int wr_event_set = PAPI_NULL;
        int rd_event_set = PAPI_NULL;
```

```
long long rd_values[8];
long long *wr_values;
struct {
        char *ename;            /* configuration event:qualifier */
        long long value;        /* value to set */
} wr_events[ ] = {
        { "AR_NIC_ORB_CFG_NET_RSP_HIST_1:BIN1_MIN", 1024 },
        { "AR_NIC_ORB_CFG_NET_RSP_HIST_1:BIN2_MIN", 8192 },
        { "AR_NIC_ORB_CFG_NET_RSP_HIST_1:BIN3_MIN", 16384 },
        { NULL, 0x0 }
};
char *rd_events[ ] = {
        "AR_NIC_ORB_PRF_NET_RSP_HIST_BIN01:BIN0_COUNT",
        "AR_NIC_ORB_PRF_NET_RSP_HIST_BIN01:BIN1_COUNT",
        "AR_NIC_ORB_PRF_NET_RSP_HIST_BIN23:BIN2_COUNT",
        "AR_NIC_ORB_PRF_NET_RSP_HIST_BIN23:BIN3_COUNT",
        "AR_NIC_ORB_PRF_NET_RSP_HIST_BIN45:BIN4_COUNT",
        "AR_NIC_ORB_PRF_NET_RSP_HIST_BIN45:BIN5_COUNT",
        "AR_NIC_ORB_PRF_NET_RSP_HIST_BIN67:BIN6_COUNT",
        "AR_NIC_ORB_PRF_NET_RSP_HIST_BIN67:BIN7_COUNT",
        NULL
};

PAPI_library_init (PAPI_VER_CURRENT);

/* set up the configuration event set */

PAPI_create_eventset (&wr_event_set);

cidx = PAPI_get_component_index ("cray_npu");
nctrs = PAPI_num_cmp_hwctrs (cidx);
wr_values = calloc (nctrs, sizeof(*wr_values));

for (i=0; wr_events[i].ename != NULL; i++) {
        PAPI_event_name_to_code (wr_events[i].ename, &code);
        PAPI_add_event (wr_event_set, code);
        wr_values[i] = wr_events[i].value;
}

PAPI_start (wr_event_set);
PAPI_write (wr_event_set, wr_values);
PAPI_stop (wr_event_set, NULL);

PAPI_cleanup_eventset (wr_event_set);
PAPI_destroy_eventset (&wr_event_set);

/* set up the counting event set configured by the above */

PAPI_create_eventset (&rd_event_set);

for (i=0; rd_events[i] != NULL; i++) {
        PAPI_event_name_to_code (rd_events[i], &code);
        PAPI_add_event (rd_event_set, code);
}

PAPI_start (rd_event_set);

/* perform work */
```

```
        memset (rd_values, 0, sizeof(rd_values));
        PAPI_stop (rd_event_set, rd_values);

        /* process values */
        for (i=0; events[i] != NULL; i++) {
            printf (" %s = %lld\n", events[i], values[i]);
        }

        PAPI_cleanup_eventset (rd_event_set);
        PAPI_destroy_eventset (&rd_event_set);
        PAPI_shutdown ( );

        return 0;
}
```