Performance Counter Monitoring for the Blue Gene/Q Architecture

Heike McCraw
Innovative Computing Laboratory
University of Tennessee, Knoxville

1 Introduction and Motivation

With the increasing scale and complexity of large computing systems the effort of performance optimization grows more and more and so does the responsibility of performance analysis tool developers. To be of value to the High Performance Computing (HPC) community, performance analysis tools have to be customized as quick as possible in order to support new processor generations as well as changes in system designs.
The Blue Gene/Q (BG/Q) system is the third generation in the IBM Blue Gene line of massively parallel, energy efficient supercomputers, and it has been scheduled to be released in 2012. BG/Q will be capable of scaling to over a million processor cores while making the trade-off of lower power consumption over raw processor speed [5]. BG/Q increases not only in size but also in complexity compared to its Blue Gene predecessors. Consequently, gaining insight into the intricate ways in which software and hardware are interacting requires richer and more capable performance analysis methods in order to be able to improve efficiency and scalability of applications that utilize this advanced system.

Unfortunately, very little effort has been put into hardware performance monitoring tools for the BG/Q predecessor Blue Gene/P (BG/P). The HPC community was left behind with rather poor and incomplete methods which made it more or less impractical to collect hardware performance counter data on this system. To eliminate this limitation, an extensive effort has been made which includes careful planning long before the BG/Q release as well as close collaboration with IBM. This paper provides detailed information about the expansion of PAPI to support hardware performance monitoring for the BG/Q platform. It offers insight into relevant implementation designs as well as supported monitoring features. As of today, this project is still ongoing and under non-disclosure agreement (NDA) with IBM.
This customization of PAPI to support BG/Q also includes a growing number of PAPI components to provide valuable performance data that not only originates from the processing cores but also from compute nodes and the system as a whole. These additional components allow hardware performance counter monitoring of the network, the I/O system and the Compute Node Kernel in addition to the CPU component.

This paper is organized as follows. The coming section provides a brief overview of the BG/Q hardware and software architecture with focus on the features that are particularly relevant for this project. Section 3 goes into detail on how PAPI has been expanded with 5 components to support hardware performance counter monitoring on the BG/Q platform. A summary and sketch of future work is provided in Section 4.

2 Overview of the Blue Gene/Q Architecture

2.1 Hardware Architecture

The BG/Q processor is an 18-core CPU and only 16 cores are used to perform mathematical calculations. The 17th core is used for node control tasks such as offloading I/O operations which ”talk” to Linux running on the I/O node. (Note, the I/O nodes are separate from the compute nodes; so, Linux is not actually running on the 17th core.) The 18th core is a spare core which is used when there are corrupt cores on the chip. The corrupt core is swapped and software transparent.
The processor uses PowerPC A2 cores, operating at a moderate clock frequency of 1.6 GHz and consuming a modest 55 watts at peak [2]. The Blue Gene line has always been known for throughput and energy efficiency, and so emphasizes the A2 architecture. Despite the low power consumption, the chip delivers a very respectable 204 Gflops [2]. This is due to a combination of features like the tight core count, support for up to four threads per core, and a quad floating-point unit. I will elaborate on those features later in the chapter. Just for comparison reason, the Power7 at 3.5 GHz and 8 cores delivers about 256 Gflops, but consumes 200 watts, which makes the BG/Q chip approximately three times more energy efficient per peak Flop (3.72 Gflops/watt for A2 versus 1.28 Gflops/watt for Power7) [2].

Even compared to its Blue Gene predecessors, BG/Q represents a big change in performance, thanks to a large raise in both, core count and clock frequency. The BG/Q chip delivers 15 times as many peak FLOPS as its BG/P counterpart and 36 times as many as the original BG/L design (see Table 1 for comparison).

Version	Core Architecture	Instruction Set	FPU	Clock Speed	Core Count	Interconnect	Peak Performance
Blue Gene/L	PowerPC 440	32-bit	dual 64-bit	700 MHz	2	3D torus	5.6 Gigaflops
Blue Gene/P	PowerPC 450	32-bit	dual 64-bit	850 MHz	4	3D torus	13.6 Gigaflops
Blue Gene/Q	PowerPC A2	64-bit	quad 64-bit	1600 MHz	18	5D torus	204.8 Gigaflops

Table 1: Brief summary of the three Blue Gene versions

This PowerPC A2 core has a 64-bit instruction set compared to the 32-bit chips used in the prior BG/L and BG/P supercomputers. As mentioned earlier, each A2 core has support for up to four threads but what’s interesting, it has in-order dispatch, execution and completion instead of out-of-order execution which is common in many RISC processor designs [3]. The A2 core has a 16 KB private L1 data cache and another 16 KB private L1 instruction cache, as well as 32 MB of embedded dynamic random access memory (eDRAM) acting as a L2 cache, and 8 GB (or 16 GB) of main memory [4]. The L2 cache as well as the main memory are shared between the cores on the chip.

The quad double-precision Floating Point Unit (FPU) (available on each core) has four pipelines which can be used to execute scalar floating point instructions, four SIMD instructions, or two complex arithmetic SIMD instructions [3]. These instructions are extensions of the Power instruction set. The FPU has a six-stage pipeline and has permutation instructions to reorganize vector data on the fly; it can perform a maximum of eight concurrent floating point operations per clock cycle plus a load and a store [3].

Every BG/Q processor has two DDR3 memory controllers, each interfacing with eight slices of the L2 cache to handle their cache misses (one controllers for each half of the 16 cores on the chip) [1, 3]. This is an important feature to know and I will come back to it in more detail when I talk about the PAPI `L2Unit` component in section 3.2.

BG/Q peer-to-peer communication between compute nodes is performed over a 5-dimensional (5D) Torus network (note that BG/L and P feature a 3D Torus). Each node has 11 links and each link can transmit data at 2 GB/s and simultaneously receive at 2 GB/s for a total bandwidth of 44 GB/s. While 10 links connect the compute nodes, the 11th link provides connection to the I/O nodes.

By default a custom lightweight operating system called Compute Node Kernel (CNK) is loaded on the compute nodes while I/O nodes run Linux OS [5]. The I/O architecture is significantly different from previous BG generations since it is separated from the compute nodes and moved to independent I/O racks.

2.2 Performance Monitoring Architecture

This section focuses on two for this project relevant system information. It provides a brief summary of the features provided by the Blue Gene Performance Monitoring API (BGPM) and the Universal Performance Counting (UPC) hardware.

BGPM implements a programming interface for the BG/Q UPC hardware, while the PAPI implementation for BG/Q accesses the BGPM interface under the cover to allow users and third-party programs to monitor and sample hardware performance counters in a traditional way. The term "traditional" here refers to the advantage that no code modifications are necessary if a code that includes PAPI functions is ported to the BG/Q architecture. BGPM provides multiple hardware and software units that can be monitored, like the P Unit (CPU related events), L2 Unit (L2 cache related), I/O Unit, Network Unit, and Compute Kernel Node Unit. Each unit supplies separate control of events and counters.

The BG/Q UPC hardware programs and counts performance events from multiple hardware modules within a BG/Q node [1]. The following list provides more details on those hardware modules and also summarizes which BGPM unit interfaces with a module:

- Each of the 18 A2 CPU cores has a *local UPC module*. Each of these modules provides 24 counters (14-bit) to sample A2 events, L1 cache related events, floating point operations, etc. The BGPM `PUnit` interfaces with these modules.
  - Also, the *local UPC module* is broken down into 5 internal sub-modules: *FU*, *XU*, *IU*, *LSU* and *MMU*. The sub-modules are transparently identifiable from the `PUnit` event names. See Table 2 for an example selection of `PUnit` events.
- Furthermore, each of the 16 L2 memory slices (per chip) has a *L2 UPC module* that provides 6 counters. The BGPM `L2Unit` interfaces with these modules.
- The *Message*, *PCIe*, and *DevBus module* - which are collectively referred to as *I/O modules* - provide together 43 counters. The BGPM `IOUnit` interfaces with these *I/O modules*. The three I/O sub-modules are transparently identifiable from the `IOUnit` event names. See Table 6 for an example selection of `IOUnit` events.

The counters from each of the UPC modules mentioned above, are periodically accumulated into a corresponding 64-bit counter, which is located in the *central UPC module*. The *central UPC module* is responsible for overflow detection and counter aggregation [1].

- Furthermore, the network provides a local *UPC network module* with 66 counters (each of the 11 links has 6 counters) (64-bits). The BGPM `NWUnit` interfaces with the network modules.
- And the memory controller provides a module with 32-bit counters

The counters of those 2 modules are kept separate from the *central UPC module*.

3 PAPI BGPM Components

In general, hardware performance event monitoring requires user code instrumentation with either the native BGPM API or a tool like PAPI which uses BGPM under the cover. The following five sections talk about the 5 different components that have been implemented in PAPI to allow users to monitor hardware performance counters on the BG/Q architecture through the standard Performance API interface. Prior to this is a short description on how the configuration has been designed to make the installation of PAPI with its 5 different BG/Q components as easy and transparent as possible to the user community.

Configuration: PAPI can be configured with one, multiple or all BGPM components. However, if no additional BGPM component has been added to the configure line (means, the `--with-components` configure option is omitted) then by default, PAPI will be configured with the `PUnit` component. The following shows how to configure PAPI with all 5 BGPM components. Note that the order of the components does not matter. The configure option `--with-OS=bgq` takes care of all the BG/Q-specific settings that are necessary behind the scene - like compiler flags, correctness of the BGPM header and library paths, bit mode, cross-compiling etc. - to ensure a successful compilation of PAPI on BG/Q.

```
./configure  --prefix=<PAPI INSTALL PATH> \
             --with-OS=bgq \
             --with-bgpm_installdir=/bgsys/drivers/ppcfloor \
             CC=/bgsys/drivers/ppcfloor/gnu-linux/bin/powerpc64-bgq-linux-gcc \
             F77=/bgsys/drivers/ppcfloor/gnu-linux/bin/powerpc64-bgq-linux-gfortran \
             --with-components="bgpm/L2unit bgpm/CNKunit bgpm/IOunit bgpm/NWunit"
```

Compilation and Linking: An application that makes calls to PAPI functions can be compiled and linked with PAPI either statically or dynamically on the BG/Q system. It is important to note that the use of the static PAPI library on BG/Q requires explicit linking with the Realtime Extensions library (`-lrt`) as well as the Standard C++ library (`-lstdc++`) (in addition to the PAPI library (`-lpapi`)). The reason for the additional linking steps is, we are not able to include the Realtime Extensions and Standard C++ libraries - which are both required by BGPM - without generating legal issues. We are not allowed to extract any contents from libraries we do not own. Below are the required steps in order to compile and link an application with **static PAPI on BG/Q**:

```
CC = /bgsys/drivers/ppcfloor/gnu-linux/bin/powerpc64-bgq-linux-gcc
PAPIINCLUDE = <install_path>/papi4bgq/include
PAPILIB = <install_path>/papi4bgq/lib
INCLUDE = -I. -I$(PAPIINCLUDE)
LIBS := -L$(PAPILIB) -lpapi -lrt -lstdc++
```

A user will not have to add these additional libraries if the dynamic PAPI library is used on BG/Q. Below are the required steps in order to compile and link an application with **dynamic PAPI on BG/Q**. Note the additionally required compiler flag `dynamic`:

```
CC = /bgsys/drivers/ppcfloor/gnu-linux/bin/powerpc64-bgq-linux-gcc
CCFLAGS = -dynamic
PAPIINCLUDE = <install_path>/papi4bgq/include
PAPILIB = <install_path>/papi4bgq/lib
INCLUDE = -I. -I$(PAPIINCLUDE)
LIBS := -L$(PAPILIB) -lpapi
```

3.1 P Unit Component

The PAPI `PUnit` component is handled as component 0 in PAPI - which is the default CPU component. This implies that no additional configure options are necessary in order for a user to be able to access the *local UPC module* counters. Table 2 shows an example selection of native `PUnit` events provided by the PAPI utility `papi_native_avail`. The five internal sub-modules of the *local UPC module* (described in Section 2.2) are easily identifiable through the event names.

In addition to native events, a user can select predefined events for the `PUnit` component on BG/Q. The implementation part is almost entirely finished and working. However, some of the currently available predefined events for the BG/Q architecture will need to be redefined. This is work in progress and the correctness of some of the event definitions are currently in discussion with IBM. Table 3 shows a list of predefined events that a user can choose from on BG/Q. Out of 107 possible predefined events, there are currently 41 events available of which 12 are derived events.

Overflow: Only the *local UPC module*, *L2* and *IO* UPC hardware support performance monitor interrupts when a programmed counter overflows [1]. For that reason, only the `PUnit`, `L2Unit`, and `IOUnit` provide overflow support in BGPM. PAPI offers overflow support for the CPU component (which in case of BG/Q is the `PUnit` component). Also the extension of overflow support to the PAPI `L2Unit` as well as `IOUnit` is completed and tested.

In general, a PAPI user assigns a decimal threshold value to the `PAPI_overflow()` function. Within PAPI this value is then subtracted from the maximum counter value (in addition to adding 1) and loaded into the counter. The counter then counts up to 0 and generates an overflow signal if the assigned threshold value is reached for the given event(s).

PUnit Event	Description
PEVT_AXU_INSTR_COMMIT	A valid AXU (non-load/store) instruction is in EX6, past the last flush point. - AXU uCode sub-operations are also counted by PEVT_XU_ COMMIT instead.
PEVT_AXU_CR_COMMIT	A valid AXU CR updater instruction is in EX6, past the last flush point.
PEVT_AXU_IDLE	No valid AXU instruction is in the EX6 stage.
...	...
PEVT_IU_IL1_MISS	A thread is waiting for a reload from the L2. - Not when CI=1. - Not when thread held off for a reload that another thread is waiting for. - Still counts even if flush has occurred.
PEVT_IU_IL1_MISS_CYC	Number of cycles a thread is waiting for a reload from the L2. - Not when CI=1. - Not when thread held off for a reload that another thread is waiting for. - Still counts even if flush has occurred.
PEVT_IU_IL1_RELOADS_DROPPED	Number of times a reload from the L2 is dropped, per thread - Not when CI=1 - Does not count when not loading cache due to a back invalidate to that address
...	...
PEVT_XU_BR_COMMIT_CORE	Number of Branches committed
PEVT_XU_BR_MISPRED_COMMIT_CORE	Number of mispredicted Branches committed (does not include target address mispredicted)
PEVT_XU_PPC_COMMIT	Number of instructions committed. uCode sequences count as one instruction.
...	...
PEVT_LSU_COMMIT_STS	Number of completed store commands. - Microcoded instructions will count more than once. - Does not count syncs,tlb ops,dcbz,icswx, or data cache management instructions. - Includes stcx, but does not wait for stcx complete response from the L2. - Includes cache-inhibited stores
PEVT_LSU_COMMIT_ST_MISSES	Number of completed store commands that missed the L1 Data Cache. Note that store misses are pipelined and write through to the L2, so the store time typically has less impact on performance than load misses. - Microcoded instructions may be counted more than once. - Does not count syncs,tlb ops, dcbz, icswx, or data cache management instructions. - Includes stcx, but does not wait for stcx complete response from the L2. - Does not includes cache-inhibited stores
PEVT_LSU_COMMIT_LD_MISSES	Number of completed load commands that missed the L1 Data Cache. - Microcoded instructions may be counted more than once. - Does not count dcbt[st][ls][ep]. - Include larx. - Does not includes cache-inhibited loads
...	...
PEVT_MMU_TLB_HIT_DIRECT_IERAT	TLB hit direct entry (instruction, ind=0 entry hit for fetch)
PEVT_MMU_TLB_MISS_DIRECT_IERAT	TLB miss direct entry (instruction, ind=0 entry missed for fetch)
PEVT_MMU_TLB_MISS_INDIR_IERAT	TLB miss indirect entry (instruction, ind=1 entry missed for fetch, results in i-tlb exception)
...	...

Table 2: Small selection of PUnit events, available on the BG/Q architecture (it also shows different events from PUnit's 5 internal sub-units mentioned in section 2.2). Currently, there are 269 PUnit events available.

On the other hand, the overflow implementation within BGPM is handled slightly differently. Here the threshold is the value that is actually programmed into the counter. However, we do not want PAPI to expose this value (that is programmed into the counter) to user space for the reason that different platforms have different widths of the counters, hence this value would differ from platform to platform. In order to keep

Name	Code	Avail	Deriv	Description (Note)
PAPI_L1_ICM	0x80000001	Yes	No	Level 1 instruction cache misses
PAPI_FXU_IDL	0x80000011	Yes	No	Cycles integer units are idle
PAPI_TLB_DM	0x80000014	Yes	Yes	Data translation lookaside buffer misses
PAPI_TLB_IM	0x80000015	Yes	No	Instruction translation lookaside buffer misses
PAPI_TLB_TL	0x80000016	Yes	Yes	Total translation lookaside buffer misses
PAPI_L1_LDM	0x80000017	Yes	No	Level 1 load misses
PAPI_L1_STM	0x80000018	Yes	No	Level 1 store misses
PAPI_BTAC_M	0x8000001b	Yes	No	Branch target address cache misses
PAPI_PRF_DM	0x8000001c	Yes	No	Data prefetch cache misses
PAPI_TLB_SD	0x8000001e	Yes	No	Translation lookaside buffer shootdowns
PAPI_CSR_FAL	0x8000001f	Yes	No	Failed store conditional instructions
PAPI_CSR_SU	0x80000020	Yes	Yes	Successful store conditional instructions
PAPI_CSR_TOT	0x80000021	Yes	No	Total store conditional instructions
PAPI_MEM_RCY	0x80000023	Yes	No	Cycles Stalled Waiting for memory Reads
PAPI_STL_CCY	0x80000027	Yes	Yes	Cycles with no instructions completed
PAPI_HW_INT	0x80000029	Yes	No	Hardware interrupts
PAPI_BR_UCN	0x8000002a	Yes	No	Unconditional branch instructions
PAPI_BR_CN	0x8000002b	Yes	No	Conditional branch instructions
PAPI_BR_TKN	0x8000002c	Yes	Yes	Conditional branch instructions taken
PAPI_BR_NTK	0x8000002d	Yes	Yes	Conditional branch instructions not taken
PAPI_BR_MSP	0x8000002e	Yes	No	Conditional branch instructions mispredicted
PAPI_BR_PRC	0x8000002f	Yes	Yes	Conditional branch instructions correctly predicted
PAPI_FMA_INS	0x80000030	Yes	Yes	FMA instructions completed
PAPI_TOT_INS	0x80000032	Yes	No	Instructions completed
PAPI_INT_INS	0x80000033	Yes	No	Integer instructions
PAPI_FP_INS	0x80000034	Yes	No	Floating point instructions
PAPI_LD_INS	0x80000035	Yes	Yes	Load instructions
PAPI_SR_INS	0x80000036	Yes	No	Store instructions
PAPI_BR_INS	0x80000037	Yes	No	Branch instructions
PAPI_RES_STL	0x80000039	Yes	No	Cycles stalled on any resource
PAPI_FP_STAL	0x8000003a	Yes	No	Cycles the FP unit(s) are stalled
PAPI_TOT_CYC	0x8000003b	Yes	No	Total cycles
PAPI_LST_INS	0x8000003c	Yes	Yes	Load/store instructions completed
PAPI_SYC_INS	0x8000003d	Yes	No	Synchronization instructions completed
PAPI_L1_DCR	0x80000043	Yes	No	Level 1 data cache reads
PAPI_L1_ICR	0x8000004f	Yes	No	Level 1 instruction cache reads
PAPI_FML_INS	0x80000061	Yes	Yes	Floating point multiply instructions
PAPI_FAD_INS	0x80000062	Yes	Yes	Floating point add instructions
PAPI_FDV_INS	0x80000063	Yes	No	Floating point divide instructions
PAPI_FSQ_INS	0x80000064	Yes	No	Floating point square root instructions
PAPI_FP_OPS	0x80000066	Yes	No	Floating point operations

Table 3: Selection of predefined events, available on the BG/Q architecture. Out of 107 possible events, currently 41 are available, of which 12 are derived. Note, events that are not defined yet for BG/Q are omitted from this table.

the input parameters for the `PAPI_overflow()` function unchanged, and still support overflow through BGPM, we convert the threshold value that is assigned by the PAPI user to the value that is programmed into the counter. This new value is then used as input parameter by `BGPM_overflow()` within PAPI. This workaround has been tested and works reliably with the standard PAPI interface.
Furthermore, when an event is set for overflow the `Bpm_SetOverflowHandler()` function is used to register a function to be called within a private signal hander. There is a handler for each event set. On overflow, the user's handler will be called with the event set handler, instruction counter, and context structure. This implementation has been validated with multiple PAPI tests that emphasize on counter

overflow.

In addition, the HPCToolkit [6] from Rice University uses PAPI overflow to get interrupts from the hardware performance counters. To expand the test suite, we ran tests that target just those features in PAPI that the HPCToolkit depends on. We were able to get sustained interrupts for non-threaded as well as threaded programs over a wide range of interrupt rates. The interrupts were stable even at 50,000/seconds. In addition, this has been tested and validated for PAPI predefined events for cycles, idle cycles, integer instructions, branch instructions, flops and L1 cache misses. Tests that use L2 and L3 cache events were omitted since those events are currently not available for overflow on BG/Q. Also testing overflow with multiple events succeeded. Table 4 shows the result of an HPCToolkit test that generates a list of PAPI predefined events that are available for overflow on BG/Q. Note that "Failed" only means that the test program did not trigger that event, not that `PAPI_overflow()` is broken for that event. Most of the "Failed" events are instruction cache misses. To trigger an instruction cache miss, one would need a large program with lots of lines of code, which is not given by the small, targeted HPCToolkit tests. Also, reason for the square root event failure is, the test program does not compute square roots.

Name	Result	Description (Note)
`PAPI_L1_ICM`	Failed	Level 1 instruction cache misses
`PAPI_TLB_IM`	Failed	Instruction translation lookaside buffer misses
`PAPI_BTAC_M`	Failed	Branch target address cache misses
`PAPI_PRF_DM`	Failed	Data prefetch cache misses
`PAPI_TLB_SD`	Failed	Translation lookaside buffer shootdowns
`PAPI_CSR_FAL`	Failed	Failed store conditional instructions
`PAPI_CSR_TOT`	Failed	Total store conditional instructions
`PAPI_HW_INT`	Failed	Hardware interrupts
`PAPI_FSQ_INS`	Failed	Floating point square root instructions
`PAPI_FXU_IDL`	Passed	Cycles integer units are idle
`PAPI_L1_LDM`	Passed	Level 1 load misses
`PAPI_L1_STM`	Passed	Level 1 store misses
`PAPI_MEM_RCY`	Passed	Cycles Stalled Waiting for memory Reads
`PAPI_BR_UCN`	Passed	Unconditional branch instructions
`PAPI_BR_CN`	Passed	Conditional branch instructions
`PAPI_BR_MSP`	Passed	Conditional branch instructions mispredicted
`PAPI_TOT_INS`	Passed	Instructions completed
`PAPI_INT_INS`	Passed	Integer instructions
`PAPI_FP_INS`	Passed	Floating point instructions
`PAPI_SR_INS`	Passed	Store instructions
`PAPI_BR_INS`	Passed	Branch instructions
`PAPI_RES_STL`	Passed	Cycles stalled on any resource
`PAPI_FP_STAL`	Passed	Cycles the FP unit(s) are stalled
`PAPI_TOT_CYC`	Passed	Total cycles
`PAPI_SYC_INS`	Passed	Synchronization instructions completed
`PAPI_L1_DCR`	Passed	Level 1 data cache reads
`PAPI_L1_ICR`	Passed	Level 1 instruction cache reads
`PAPI_FDV_INS`	Passed	Floating point divide instructions
`PAPI_FP_OPS`	Passed	Floating point operations

Table 4: A summary of the predefined events that are available for overflow on BG/Q. Total PAPI Presets: 107, Available: 41, Overflow: 29, Passed: 20

It is important to note that BGPM comes with another restriction that freezes an event set after it has been applied. That means that no more changes - including setting the event set for overflow - can be made. Since

PAPI does not carry this kind of restriction and in order to maintain PAPI's flexibility, we implemented a workaround to eliminate this constraint. `PAPI_overflow()` now checks if an event set has been applied before enabling it for overflow. If that's the case, the BGPM event set is deleted, a new one is created and rebuilt as it was prior to deletion, it then is set for overflow and finally applied. This implementation has been validated and stress-tested with various overflow tests.

Fast versus Slow Overflow: Punit counters freeze on overflow until the overflow handling is complete. However L2 and I/O units do not freeze on overflow. The L2 and I/O counts will be stopped when the interrupt is handled. The signal handler restarts L2 and I/O counting when done [1].
`PUnit` counters can detect a counter overflow and raise an interrupt within a few cycles (O(4)) of the overflowing event [1]. However, according to the BGPM documentation it takes up to O(800) cycles before the readable counter value is updated. This latency does not effect the overflow detection, and so we refer to a `PUnit` overflow as a "Fast Overflow".
The `IOUnit` and `L2Unit` take up to 800 processor cycles to accumulate an event and detect an overflow [1]. Hence, we refer to this as a "Slow Overflow", and the program counters may alter up to 800 cycles or more after the event.

Multiplexing: PAPI supports multiplexing for the BG/Q platform. The BGPM `Punit` does not directly implement multiplexing of event sets [1]. However, it does indirectly support multiplexing by supporting a multiplexed event set type [1]. A multiplexed event set type will maintain sets of events which can be counted simultaneously, while pushing conflicting events to other internal sets [1]. BGPM comes with some restrictions that an event set has to be empty before the activation of multiplexing is possible. In order to maintain PAPI's flexibility, we implemented a workaround to eliminate this constraint. If an event set is not empty before converting the event set into a multiplexed set, PAPI now deletes the BGPM event set, creates a new empty BGPM event set, enables multiplexing by calling `Bgpm_SetMultiplex()`, and rebuilds the BGPM event set as it was prior to deletion. This implementation has been validated with multiple PAPI tests that emphasize on multiplexing.

Profiling: The PAPI tests that stress profiling do not work yet on the grounds that the executable regions require reading the "/proc/PID/maps" file. However, the "/proc" file system does not de facto exist on the Blue Gene systems - at least not in a useful fashion since what is seen is "/proc" on the I/O node that is associated with the compute node, but this contains information only for processes running on the I/O node. There is a "/jobs" that contains a subset of the "/proc" information for processes on the compute node, but it does not contain the "maps" file. Clearly there is more work required in finding out if there is another way to get access to the required information other than using the BG/Q personality function `Kernel_GetPersonality()`. Since we are looking for the memory map information for a process we might be able to try the Linux call `dl_iterate_phdr()` which walks through a list of shared objects [7]. However, this is currently work in progress.

3.2 L2 Unit Component

The shared L2 cache on the BG/Q system is split into 16 separate slices. Each of the 16 slices has a *L2 UPC module* that provides 6 counters. Those 6 counters are node-wide, and cannot be isolated to a single core or thread [1]. As mentioned earlier, every BG/Q processor has two DDR3 memory controllers, each interfacing with eight slices of the L2 cache to handle their cache misses (one controllers for each half of the 16 cores on the chip) [1, 3]. The counting hardware can either keep the counts from each slice separate, or combine the counts from each slice into single values (which is the default). The combined counts are significantly important if a user wants to sample on overflows. Actually, the separate slice counts are not particularly interesting except for perhaps investigating cache imbalances because consecutive memory lines are mapped to a separate slices. The node-wide ”combined” or ”sliced” operation is selected by creating an event set from the ”combined” (default), or ”sliced” group of events. Hence a user cannot assign events from both groups. See Table 5 for a small selection of `L2Unit` events. Currently, there are 32 `L2Unit` events available on the BG/Q architecture.

Overflow: If `L2Unit` event overflow is desired, the overflow signal is ”slow” (see the end of Section 3.1 for details that describe the difference between fast and slow overflow). As mentioned before, PAPI does support overflow for `PUnit` events as well as `L2Unit` and `IOUnit` events.

`L2Unit` Event	Description
`PEVT_L2_HITS`	hits in L2, both load and store. Network Polling store operations from core 17 on BG/Q pollute in this count during normal use
`PEVT_L2_MISSES`	cacheline miss in L2 (both loads and stores
`PEVT_L2_PREFETCH`	fetching cacheline ahead of L1P prefetch
...	...

Table 5: Small selection of L2Unit events, available on the BG/Q architecture. Currently, there are 32 L2Unit events available.

3.3 I/O Unit Component

The *I/O module* provides a total of 43 counters for the *Message*, *PCIe*, and *DevBus module*. These counters are node-wide and cannot be isolated to any particular core or thread [1]. See Table 6 for a small selection of `IOUnit` events. Currently, there are 44 `IOUnit` events available on the BG/Q architecture. The three I/O sub-modules are transparently identifiable from the `IOUnit` event names.

Overflow: If `IOUnit` event overflow is desired, the overflow signal is ”slow” (see the end of Section 3.1 for details that describe the difference between fast and slow overflow). As mentioned before, PAPI does support overflow for `PUnit` events as well as `L2Unit` and `IOUnit` events.

`IOUnit` Event	Description
`PEVT_MU_PKT_INJ`	A new packet has been injected (Packet has been stored to ND FIFO
`PEVT_MU_MSG_INJ`	A new message has been injected (All packets of the message have been stored to ND FIFO
`PEVT_MU_FIFO_PKT_RCV`	A new FIFO packet has been received (The packet has been stored to L2. There is no pending switch request)
...	...
`PEVT_PCIE_INB_RD_BYTES`	Inbound Read Bytes Request
`PEVT_PCIE_INB_RDS`	Inbound Read Request
`PEVT_PCIE_INB_RD_CMPLT`	Inbound Read Completion
...	...
`PEVT_DB_PCIE_INB_WRT_BYTES`	PCIe inbound write bytes written
`PEVT_DB_PCIE_OUTB_RD_BYTES`	PCIe outbound read bytes requested
`PEVT_DB_PCIE_OUTB_RDS`	PCIe outbound read request
...	...

Table 6: Small selection of I/OUnit events, available on the BG/Q architecture. Currently, there are 44 I/OUnit events available.

3.4 NW Unit Component

The 5D Torus network provides a local *UPC network module* with 66 counters - each of the 11 links has 6 64-bit-counters. As of right now, a PAPI user cannot select which network link to attach to. Currently, all network links are attached and this is hard-coded in the PAPI `NWUnit` component. We are discussing options for supporting the other enumerations for network links as well. One possible option to try may be using event attribute strings for that purpose.
See Table 7 for a small selection of `NWUnit` events. Currently, there are 31 `NWUnit` events available on the BG/Q architecture.

`NWUnit` Event	Description
`PEVT_NW_USER_PP_SENT`	Number of 32 byte user point to point packet chunks sent. Includes packets originating or passing through the current node
`PEVT_NW_USER_DYN_PP_SENT`	Number of 32 byte user dynamic point to point packet chunks sent. Includes packets originating or passing through the current node
`PEVT_NW_USER_ESC_PP_SENT`	Number of 32 byte user escape point to point packet chunks sent. Includes packets originating or passing through the current node
...	...

Table 7: Small selection of NWUnit events, available on the BG/Q architecture. Currently, there are 31 NWUnit events available.

3.5 CNK Unit Component

CNK is the lightweight Compute Node Kernel and it is the only kernel that runs on all the 16 compute cores. In general, on Linux kernels the “/proc” file system is the usual access method for kernel counts. Since BG/Q does not have a “/proc” filesystem (as mentioned earlier), BGPM offers a “virtual” `CNKUnit`

that has software counters collected by the kernel. The kernel counter values are read via a system call that requests the data from the lightweight compute node kernel that's running on all the compute cores.
Also, there is a read operation to get the raw value since the system has been booted. See Table 8 for a small selection of `CNKUnit` events. Currently, there are 29 `CNKUnit` events available on the BG/Q architecture.

`CNKUnit` Event	Description
`PEVT_CNKHWT_SYSCALL`	External Input Interrupt
`PEVT_CNKHWT_CRITICAL`	Critical Input Interrupt
`PEVT_CNKHWT_FIT`	Fixed Interval Timer Interrupt
...	...

Table 8: Small selection of CNKUnit events, available on the BG/Q architecture. Currently, there are 29 CNKUnit events available.

4 Summary and Future Work

Performance analysis tools for parallel applications running on large scale computing systems typically rely on hardware performance counters to gather performance relevant data from the system. In order to allow the HPC community to collect hardware performance counter data on IBM's latest Blue Gene system BG/Q, PAPI has been extended with 5 additional components.
The PAPI customization for BG/Q accesses the BGPM interface under the cover, allowing users and third-party programs to monitor and sample hardware performance counters in a traditional way using the default PAPI interface. The added PAPI components allow hardware performance counter monitoring not only for the processing units but also for the 5D Torus network, the I/O system, and the Compute Node Kernel.
Future efforts will focus on a further improved `NWUnit` component that is supposed to allow a user to select which of the 11 network links to attach an event set to.

References

[1] BGPM Documentation

[2] `http://www.hpcwire.com/hpcwire/2011-08-22/ibm_specs_out_blue_gene_q_chip.html`

[3] `http://www.theregister.co.uk/2011/08/22/ibm_bluegene_q_chip`

[4] `http://www.multicoreinfo.com/2011/02/bluegeneq`

[5] T. Budnik, et al., "Blue Gene/Q Resource Management Architecture", IEEE MTAGS 2010, New Orleans, USA, 2010

[6] `http://hpctoolkit.org`

[7] `http://linux.die.net/man/3/dl_iterate_phdr`