Next-Generation Performance Counters: Towards Monitoring Over Thousand Concurrent Events

Valentina Salapura [†], Karthik Ganesan[⋆], Alan Gara [†], Michael Gschwind [†], James C. Sexton [†] and Robert E. Walkup [†]

[†]IBM Thomas J. Watson Research Center
Yorktown Heights, NY
[⋆] University of Texas at Austin
Austin, TX

Abstract

We present a novel performance monitor architecture, implemented in the Blue Gene/P^{TM} supercomputer. This performance monitor supports the tracking of a large number of concurrent events by using a hybrid counter architecture. The counters have their low order data implemented in registers which are concurrently updated, while the high order counter data is maintained in a dense SRAM array that is updated from the registers on a regular basis. The performance monitoring architecture includes support for per-event thresholding and fast event notification, using a two-phase interrupt-arming and triggering protocol. A first implementation provides 256 concurrent 64b counters which offers an up to 64x increase in counter number compared to performance monitors typically found in microprocessors today, and thereby dramatically expands the capabilities of counter-based performance tuning.

1 Introduction

Traditionally, the architecture and design of performance monitors have not been prime considerations in overall processor design, since they do not directly impact processor performance. Most current processors support only a very limited number of performance counters, because of the cost in area, wiring resources and power dissipation.

The implementation expense has made it unattractive to provide a robust performance monitor functionality on commodity processors. Most parts will never be used for performance tuning, yet this capability adds cost to all shipping parts. The broad cost will benefit only a few development systems.

However, for large high-performance computing systems, performance statistics and feedback on program hot spots and bottlenecks are of paramount importance. As performance monitor design can have a direct impact on the overall effectiveness of such a computer system, the investment in performance counter design is justified by the performance gain enabled by them.

As all installed high-performance supercomputer systems are used both as development and production systems, feedback derived from performance monitoring will typically increase the overall system efficiency.

In the future, we can see that the importance of performance monitors will increase, in response to the emergence of control techniques that dynamically re-tune system behavior, to both increase performance and decrease power [6, 8].

This paper gives the first disclosure of a novel hybrid counter array architecture based on SRAM arrays and discrete counters, as implemented in the Blue Gene/P^{TM} Compute chip. The ability to track a large number of events concurrently is a new capability for computer systems that will give programmers and performance analysts better insights into the performance of applications on the system. It will offer actionable information to assist with application tuning. The quantitative information that can be obtained on the effectiveness of various computer sub-systems will guide design decisions for future systems.

This paper is organized as follows: we give an overview of the state of the art of performance monitoring in microprocessors in Section 2, and give an overview of the Blue Gene/P architecture in Section 3. We introduce our hybrid architecture is Section 4, and discuss interrupt generation in Section 5. Section 6 describes the configurations of the counters, and Section 7 its software front-end. We give an example for counter usage in Section 8, and draw our conclusions in Section 9.

2 State of the Art in Performance Counter Design

Many processor architectures include a set of performance counters that monitor system components such as processors, memory, and network I/O by counting specific events, such as cache misses, pipeline stalls and floating point operations. Statistics of such events can be collected in hardware with little or no overhead from the operating system or the application running on it, making performance counters a powerful means to monitor an application and analyze its performance. An overview of existing performance monitoring systems is presented by Sprunt [14].

However, most traditional processors support a very limited number of counters. For example, Intel's X86 and IBM PowerPC implementations typically support 4 to 8 event counters. While typically each counter can be programmed to count a specific event from the set of possible counter events, it is not possible to count more than N events simultaneously, where N is the number of counters physically implemented on the chip. Furthermore, the event routing and multiplexing resources on the chip typically impose additional limitations on which combinations of events can be monitored concurrently.

If an application tuning specialist needs to collect information on more than N processor, memory or I/O events, execution of the application will have to be repeated several times, each time with a different setting of the performance counters. In addition to being time consuming, the collected statistics can also be inaccurate and hard to correlate, as separate application runs can exhibit differences in behavior and triggered events. This can be due to differences in machine resources, such as cache contents, predictor values, I/O characteristics due to network load, etc. This is especially true for multiprocessor applications.

An alternative approach to this problem is time multiplexing the performance counters [1, 10]. In this approach, performance counters are reconfigured for different sets of counter events at regular time intervals. However, time-multiplexing of performance counters introduces reconfiguration overhead, and time alignment of samples. Additionally, the results of time-multiplexing hardware events are statistically similar (within 15%) to non-multiplexed data [1].

Martonosi *et al.* [7] explain the importance of employing hardware performance counters for tuning multiprocessor systems. In shared address space multiprocessor systems, much of communication and synchronization occurs via the cache coherence mechanism, and is therefore virtually impossible to measure in software. In message passing multiprocessor systems, hardware performance counters are invaluable for monitoring fine grained communication.

A robust performance monitoring subsystem requires not only providing a large number of counters, but also comparatively wide counters (such as 64b per counter) to capture a representative workload execution period, and avoid counter overflows and wrap-around. Small counter widths also lead to spurious interference by software handlers to unload and reset counters during application runs. However, area and power consumption are limiting factors on the number of counters that can be implemented.

Conventionally, performance monitors have been targeted at internal microprocessor core events, while less consideration was given to system-level events. As uniprocessor applications give way to multi-core or massively parallel many-core solutions, understanding the program behavior at the system level, and specifically the interaction between cores, becomes of paramount importance for application performance tuning.

In this article, we explore a performance monitor design for tracking events in a massively parallel multiprocessor system such as Blue Gene. The goal of this work was to provide a scalable solution, supporting over a thousand events, including core events, memory hierarchy and coherence events, as well as I/O and network traffic events.

This goal leads to a number of competing requirements:

- A big state space requires dense storage technologies to store a large number of bits efficiently. Memories, such as SRAM, achieve high density by providing only a limited number of access ports to a large number of bits.

- Supporting simultaneous events requires separate parallel access to counters to gather and accumulate statistics. Each simultaneous event requires the ability to read and modify the hardware performance counter at any point in time concurrently with any other combination of events.

- Low-latency threshold notification requires concurrent comparison of all updated events with a threshold, to raise an interrupt when a threshold is reached.

3 Blue Gene/P System Overview

The Blue Gene® system family is the first high performance computing (HPC) system that implements a large number of performance counters for performance optimization and tuning. The first generation chip, the Blue Gene/L$^{\text{TM}}$ compute chip [5, 12, 13] has 48 32-bit performance counters, and mapping of events onto physical counters is handled through the user-level API BGLperfctr [9]. The second generation Blue Gene/P compute chip [4] implements the larger and more versatile set of performance counters described in this paper.

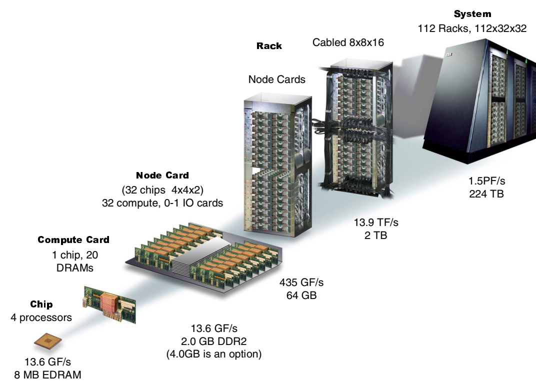

Figure 1. BlueGene/P System Architecture

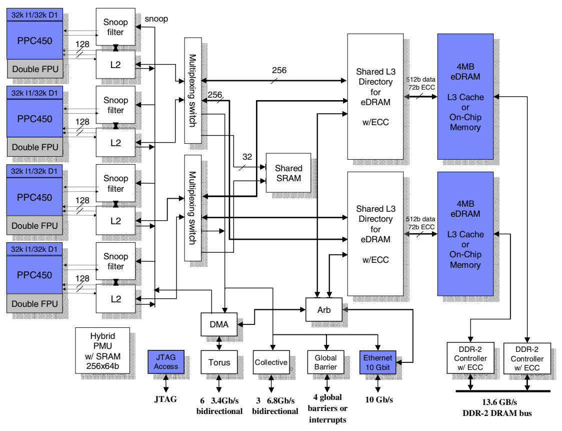

Figure 2. Blue Gene/P Compute chip architecture.

The Blue Gene/P supercomputer is a scalable, distributed-memory system consisting of up to 262,144 nodes (illustrated in Figure 1). Each node comprises a single ASIC, the Blue Gene/P Compute (BPC) chip, and its associated DRAM chips. The BPC chip is a highly integrated System-on-a-Chip (SoC) chip multiprocessor (CMP), based on four PowerPC 450 embedded processor cores.

The PowerPC 450 core is a high-performance, out-of-order industry-standard PowerPC microprocessor core originally targeted at high-end embedded systems. The processor supports 2-way superscalar instruction execution with a seven stage pipelined microarchitecture. The processor cores include highly associative first level instruction and data caches with a capacity of 32KB each. As illustrated in Figure 2, on the BPC chip, each PowerPC 450 core is coupled to a small, private, second-level cache whose principal responsibility is to prefetch streams of data from the 8 MB shared third-level cache. The L3 cache interfaces to two on-chip memory controllers, which directly control 2 GB or 4 GB of external DDR2 DRAM.

A dual-pipeline SIMD floating point unit is attached to each processor core. The floating point unit pairs two floating-point register files and two execution pipes. Both primary and secondary register files are independently addressable, but they can be jointly accessed by SIMD instructions. SIMD execution exploits the data-level parallelism often present in high-performance computing workloads, and reduces the number of instructions necessary to fetch, issue and complete, while increasing the number of operations completed.

The BPC chip also integrates the interfaces to five dedicated communication networks: the torus network, the collective network, the barrier network, 10Gb/s Ethernet, and IEEE1149.1 (JTAG). The main network is the torus, which provides high performance data communication to nearest neighbor nodes in a 3D mesh configuration (with ends wrapped around) with low latency and high throughput. The collective network supports efficient collective operations, such as broadcast and reduction.

4 A Scalable Performance Monitor Architecture

The hybrid performance monitor architecture implemented in the BPC chip provides concurrent access to a large number of counters as well as a high area density. This is achieved by splitting the counters into a high rate of change portion implemented using register logic, and a densely implemented portion using SRAM arrays.

Thus, we build each counter from a 52 bit SRAM word to provide the high-order bits, and a 12 bit counter to provide the low-order bits. At the maximum event rate of 1 event

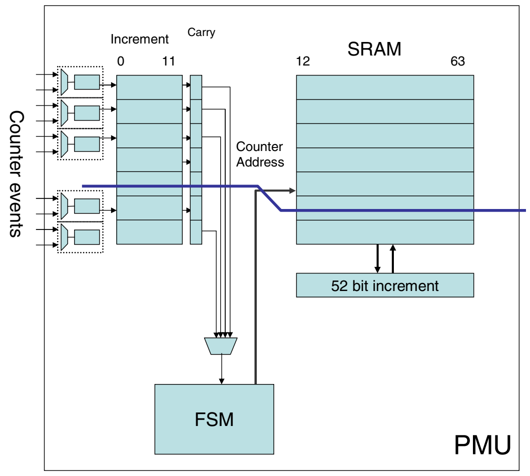

Figure 3. Hybrid performance counter architecture.

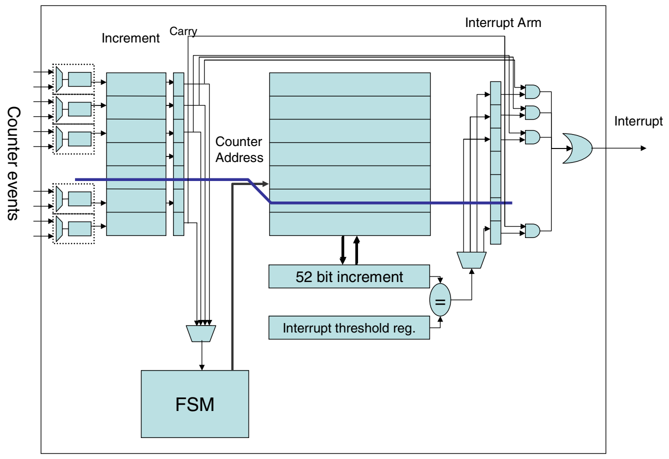

Figure 4. Hybrid counter array with interrupt pre-indication.

per cycle, the low-order counter will overflow at the earliest after $2^{12} = 4096$ cycles. This overflow condition is captured in a carry latch.

The rate-reduced overflow event is accumulated in the dense SRAM array portion of the counters. Because an overflow event can occur only every 2^{12} cycles, a maintenance state machine can keep up with this event rate if it is guaranteed to service every counter in less than 4096 cycles. Thus, with 256 low-order counters coupled to a memory array with 256 entries we will have 16 cycles to read each memory word, increment based on the associated carry bit, and write back.

The basic architecture of this hybrid counter design is illustrated in Figure 3. The 256 counter events to monitor are selected from a set of 1024 events selected by input multiplexers. The selected performance events are counted in the counter block with 256 12-bit wide counters.

When a counter from the counter block reaches its maximum, its value rolls over to zero, the corresponding "carry bit" is set, and the counter continues to track performance events. The 52 most significant bits of counters are stored in a 256×52 bit SRAM memory array block. The width of the memory array is increased with additional bits to implement a parity protection scheme.

A maintenance state machine cycles through all memory entries (SRAM word addresses) in a round-robin fashion, and checks the status of the carry latch associated with that counter. If the carry bit is set, this memory entry is read out, incremented in the increment block, and the incremented counter value is stored back to the memory array. Note that we implement only one incrementer shared by all memory entries rather than 256, one for each counter. Reading out a memory entry from the SRAM memory array, incrementing the counter value, and storing the result back into the memory is easily accomplished within the 16 cycles available.

5 Low-Latency Threshold Interrupts

Thresholding enables interrupting the CPU on interesting events, while continuously monitoring a process. This allows the exploitation of workload-specific events to guide the optimization of data placement, thread assignment to processors [3], and communication patterns.

Without a hardware-supported thresholding capability, monitoring software would have to periodically poll the performance monitor counters to determine their status. In some cases, event polling can involve scanning a large memory array (for example to determine if a packet was received), which is both time consuming and disruptive [7].

Thresholding monitors a counter value. If the counter value reaches or exceeds the user specified threshold value, an interrupt is generated to notify the CPU. This mechanism can be used by a higher level application, monitoring software or the operating system to take software action in response to monitored behavior.

To ensure low-latency interrupt handling, the threshold value must be compared against the counter value and a notification must be signaled. However, values stored in multiple array entries cannot be readily compared simultaneously. In addition, allocating a full comparator for each entry would lead to prohibitive hardware cost.

Again, we exploit the hybrid architecture. A value meets or exceeds a threshold exactly if the high order part matches and the low order part meets or exceeds the threshold value.

To implement threshold comparison efficiently, it follows the approach of hybrid event counting, as shown in figure 4. The state machine implementing the array update also implements a threshold comparison for the high-order 52 bits of the event counters that have interrupt enabled. When the high order 52 bits match, a single bit match indicator is set, "arming" the interrupt. The next time the low-order bits of the same event counter result in a carry-out, an interrupt will be raised.

Based on this implementation, a threshold notification event can be obtained for exact counter multiples of 2^{12} = 4096. Thresholds which are not multiples of 4096 of the form $4096 \times n + m$ can be achieved by configuring a threshold value of $4096 \times n$ and preloading the low-order bit counter with the value $(4096 - m)$.

6 Flexible Event Monitoring Framework

In a system, there are a variety of event types and there are various uses for any single event type. For example, a cache miss event may be used either as an edge-triggered or as a level sensitive event. As an edge-triggered event, the performance monitor captures the total number of cache misses which occurred. This information can be used for a variety of purposes ranging from identifying hot spots, to understanding cache behavior and to optimizing data layout. On the other hand, level-sensitive counting of the cache miss event captures the total number of cycles spent in cache miss handling. This is useful for deriving CPI stacks to understand overall program behavior, and to understand memory subsystem issues such as queuing and port contention.

Edge sensitive counting is most useful for counting distinct events, such as the occurrence of interrupts, or memory read and write requests. These can be used to compile such information as instruction frequency statistics, hit and miss statistics at different cache levels or to track communication parameters in system networks. Level sensitive counting can capture aggregate delay information, such as aggregate latency cycles associated with different events.

Thus, in addition to providing for a large number of concurrent events, it is also important to provide flexibility in capturing events based on specific event properties.

This flexibility is achieved by associating configuration information with each counter. The configuration information is accessible as memory-mapped configuration registers from user-space, as shown in Figure 5. User-space configurability reduces the overhead of configuring and accessing performance monitors, and avoids disruption and contamination of performance monitor data by operating system intervention.

Each counter has a set of configuration bits to define its characteristics. To reduce the cost of configuration information, our implementation groups counters into small counter groups which share a configuration register, with separate fields for each counter.

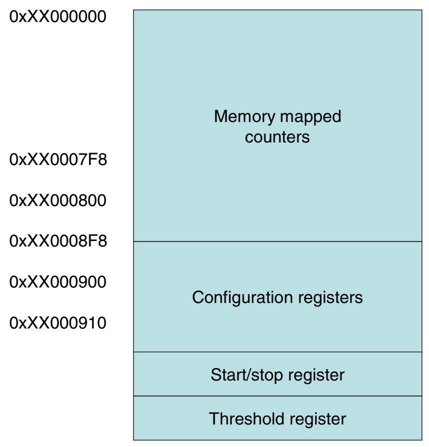

Figure 5. Memory mapping of performance counters.

In this framework, each counter can be individually and separately configured to count in one of four different signal level modes. The user has the choice between level-sensitive events (low- or high-active) and edge-sensitive signaling (low-high- or high-low transition). Each counter can be configured to select one of four associated counter inputs. An additional bit per counter enables an interrupt if a specified threshold value is reached.

To help in application tuning, we support a mechanism to isolate counting events only on code segments of an application which are analyzed. Typically, such relevant code segments are functions or loops, and it is desirable to count events generated only by these code portions, and not from the whole application.

To achieve this, we implement a single start/stop facility simultaneously operating on all counters, to stop and restart event counting. This is achieved by writing a single start/stop register (separate from the configuration registers) in the memory-mapped I/O region of the counter unit. See Figure 5.

7 Performance Monitor Software

The first level of software support for performance monitors is PAPI (Performance Application Programming Interface), a standardized interface [11] to provide applications with access to hardware performance monitors (HPM). PAPI includes functions that allow user applications to initialize the HPM, initiate and reset HPM counters, read the HPM counters and generate interrupts on HPM counter overflow and register interrupt handlers.

In addition to PAPI, other performance monitor interfaces can be supported by our design.

However, the BlueGene/P performance monitoring unit provides more advanced features that are best taken advantage of with hardware-specific functions. For example, the global accessibility of configuration and count values allows simultaneous program execution and monitoring.

Thus, a single monitoring thread executing as part of a system service, or as part of an application, can read the performance monitor values and either provide them to online (on-the-fly) system optimization tasks, or send them to another computer for collection or analysis.

Dynamic system optimization with on-the-fly performance analysis to optimize data layout, thread placement and optimization of communication patterns is best performed as part of the operating system and system management stack.

Finally, to isolate the analysis of critical code regions, the start/stop function can be used for a variety of purposes. System services can use the function to disable monitoring of system operations which might perturb application profiles, or the start/stop function may be used to count data only corresponding to a defined program region. The latter case is especially useful when optimizing aspects of an application which significantly affect execution time, such as data layout or inter-processor communication.

8 System Integration

A single performance monitor unit is implemented on each Blue Gene/P Compute chip. Due to its hybrid architecture, the majority of counter state takes only a small fraction of the overall counter unit area, as seen in figure 6.

While the usage of the present performance monitor unit for application and system tuning will be described in future work, we give an example of data collection with the unit.

For any given time interval, the performance monitor unit can track 256 events simultaneously, giving wide information about the application that is executing. The collected data includes information on processor cores, floating point units, all levels of caches, coherence traffic information, and a number of events related to network traffic. An example of events monitored during a Block Tri-diagonal Solver (BT) application run from NAS benchmark [2] on a Blue Gene/P system is shown in Table 1. We list here only a subset of events for processor cores 0 and 1 and their respective floating point units and L2 prefetch caches, to give an illustration of the versatility of the performance monitoring unit.

The large number of available counters enable the gathering of a lot of information about the processors, memory, and network subsystems. This information can be used for a variety of system measurements as well as for system optimizations. Examples are the counters for floating point operations, that can be used to determine the performance of the system expressed in FLOPS. Other uses are related to the power management of the chip, static and dynamic compiler optimizations, and workload distribution. For application tuning, performance counters can provide feedback to application developers on how efficiently they are using the resources of the machine, for example the number of cycles the system was stalled waiting on memory access, or information on floating point pipeline issues.

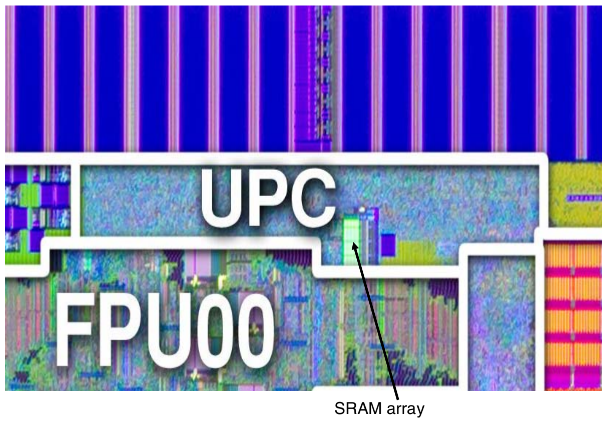

Figure 6. The Blue Gene/P Universal Performance Counter (UPC) unit captures node-wide counts of microprocessor, memory hierarchy, network and I/O events. While the SRAM stores most of the counter state, it occupies only a small fraction of the unit area.

In addition, the rich set of performance counters allows us to collect information that may influence the architecture of future systems. Examples are:

- We can precisely measure the data volume to the off-chip DDR memory, and thus optimally size the memory controllers
- We can determine the number of clock cycles that processors are waiting for data reads from the off-chip memory to find out if the system is stalled on memory accesses. System stalls would allow the start of additional threads.
- Performance counters include performance information for all levels of the caches, including coherence information for snoop traffic. This information could be used for sizing memory and symmetric multiprocessors (SMP).
- Performance counters are available for tracking net-

Event name	Counter ID	Average value	Event name	Counter ID	Average value
PU0_JPIPE_INSTRUCTIONS	0	1.96E+09	PU1_JPIPE_INSTRUCTIONSD	35	1.55E+09
PU0_JPIPE_ADD_SUB	1	2.87E+10	PU1_JPIPE_ADD_SUB	36	2.93E+10
PU0_JPIPE_LOGICAL_OPS	2	3.26E+09	PU1_JPIPE_LOGICAL_OPS	37	1.97E+09
PU0_IPIPE_INSTRUCTIONS	4	7.7E+09	PU1_IPIPE_INSTRUCTIONS	39	7.07E+09
PU0_IPIPE_MULT_DIV	5	38224109	PU1_IPIPE_MULT_DIV	40	4772199
PU0_IPIPE_ADD_SUB	6	1.05E+10	PU1_IPIPE_ADD_SUB	41	1.17E+10
PU0_IPIPE_LOGICAL_OPS	7	3.12E+09	PU1_IPIPE_LOGICAL_OPS	42	2.9E+09
PU0_IPIPE_BRANCHES	9	1.32E+10	PU1_IPIPE_BRANCHES	44	1.23E+10
PU0_DCACHE_MISS	15	2.46E+08	PU1_DCACHE_MISS	50	2.44E+08
PU0_DCACHE_HIT	16	1.06E+10	PU1_DCACHE_HIT	51	1.15E+10
PU0_DATA_LOADS	17	5.5E+09	PU1_DATA_LOADS	52	5.84E+09
PU0_DATA_STORES	18	4.03E+09	PU1_DATA_STORES	53	4.58E+09
PU0_ICACHE_MISS	20	20859131	PU1_ICACHE_MISS	55	4628756
PU0_FPU_ADD_SUB_1	22	39898517	PU1_FPU_ADD_SUB_1	57	39834798
PU0_FPU_MULT_1	23	2.24E+08	PU1_FPU_MULT_1	58	2.23E+08
PU0_FPU_FMA_2	24	5.55E+08	PU1_FPU_FMA_2	59	5.55E+08
PU0_FPU_DIV_1	25	8252723	PU1_FPU_DIV_1	60	8254148
PU0_FPU_ADD_SUB_2	27	55481	PU1_FPU_ADD_SUB_2	62	55861
PU0_FPU_MULT_2	28	691425	PU1_FPU_MULT_2	63	691476
PU0_FPU_FMA_4	29	1053207	PU1_FPU_FMA_4	64	1052646
PU0_FPU_QUADWORD_LOADS	31	11050217	PU1_FPU_QUADWORD_LOADS	66	11041834
PU0_FPU_QUADWORD_STORES	33	36801959	PU1_FPU_QUADWORD_STORES	68	36815807
PU0_L2_VALID_PREFETCH_REQUESTS	72	91491349	PU1_L2_VALID_PREFETCH_REQUESTS	104	89531718
PU0_L2_PREFETCH_HITS_IN_STREAM	74	64467321	PU1_L2_PREFETCH_HITS_IN_STREAM	106	64296927
PU0_L2_CYCLES_PREFETCH_PENDING	75	61389539	PU1_L2_CYCLES_PREFETCH_PENDING	107	60868353
PU0_L2_PAGE_ALREADY_IN_L2	76	56301845	PU1_L2_PAGE_ALREADY_IN_L2	108	51160022
PU0_L2_READ_REQUESTS	81	4.14E+08	PU1_L2_READ_REQUESTS	113	4.87E+08
PU0_L2_L3_READ_REQUESTS	83	56667890	PU1_L2_L3_READ_REQUESTS	115	54606960
PU0_L2_NETBUS_READ_REQUESTS	84	3.34E+08	PU1_L2_NETBUS_READ_REQUESTS	116	3.43E+08
PU0_L2_PREFETCHABLE_REQUESTS	86	91515948	PU1_L2_PREFETCHABLE_REQUESTS	118	89553124
PU0_L2_HIT	87	90846332	PU1_L2_HIT	119	88933771

Table 1. Example subset of counter events collected during a run of the BT application from NAS benchmark on a Blue Gene/P chip.

work events. We can gather insight into the communication requirements of parallel applications.

As an illustration of the use of the performance monitor unit for compiler development and tuning, we counted how many floating point operations are generated for the single versus the dual (SIMD) floating point unit execution pipes. Figure 7 illustrates this phenomenon for the FFT (FT) NAS parallel benchmark. When the qarch440d compiler flag is used to extract data parallelism (in addition to the normal optimizations), we observe that this FFT application can significantly benefit from deploying the SIMD FPU. In addition to generating SIMD FPU operations, the compiler option also introduced quadloads and quadstores into the instruction mix, reducing the number of required double and single store operations. Information such as this is especially useful for large applications with thousands of routines, where this information would be extremely hard to extract by using assembly listings.

In a forthcoming paper, we will more extensively describe the use of the performance monitor unit for compiler tuning, memory subsystem and network analysis, and application tuning.

9 Conclusion

We have discussed the design of an advanced performance monitor unit for a multiprocessor system which supports a large number of simultaneous events.

The unit is configurable to provide flexibility in adapting to different usage scenarios. Threshold-based interrupts and global start/stop functionality offer a simple and efficient way for the performance monitor unit to interact with operating systems, application software and performance tuning tools.

The use of an area-efficient yet flexible hybrid implementation of dense memory arrays for most significant bits coupled with classic counters for least significant bits allows for the simultaneous monitoring of a large number of events with wide counters, while preserving performance monitor counter configurability.

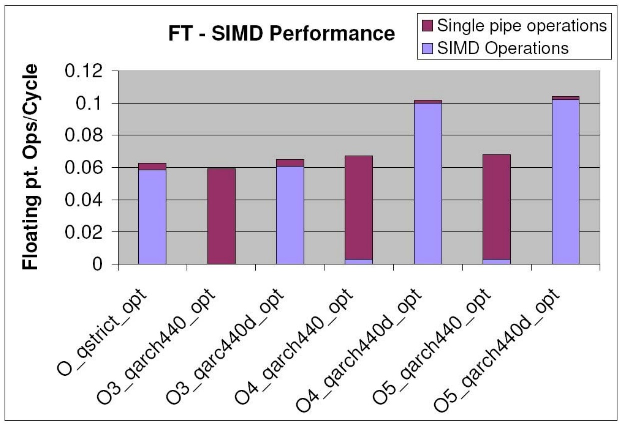

Figure 7. FT - SIMD instructions for different compiler optimizations

The capability to monitor a large number of concurrent events mitigates the complexity of time-multiplexing of performance counters, time alignment of samples, and other difficulties usually connected with performance monitoring efforts. Application tuning and operating system specialists can concentrate on data mining and performance optimization.

While the unit has been designed in the context of a massively parallel supercomputer, performance monitoring in multiprocessor systems will become increasingly important for a broader range of microprocessors as chip multiprocessors with many cores enter the mainstream computing market.

10 Acknowledgments

The Blue Gene/P system resulted from the dedicated work of a large team, and we thank all members of that team. We thank in particular Ruud Haring for his help and suggestions during the preparation of this paper.

The Blue Gene/P project has been supported and partially funded by Argonne National Laboratory and the Lawrence Livermore National Laboratory on behalf of the United States Department of Energy under Subcontract No. B554331.

References

[1] R. Azimi, M. Stumm, and R. Wisniewski. Online performance analysis by statistical sampling of microprocessor performance counters. In *ACM International Conference on Supercomputing*, June 2005.

[2] D. Bailey, T. Harris, W. Saphir, R. van der Wijngaart, A. Woo, and M. Yarrow. The NAS parallel benchmarks 2.0. Technical Report NAS-95-929, NASA Ames Research Center, December 1995.

[3] G. Bhanot, A. Gara, P. Heidelberger, E. Lawless, J. Sexton, and R. Walkup. Optimizing task layout on the Blue Gene/L supercomputer. *IBM Journal of Research and Development*, 49(2/3), 2005.

[4] BlueGene/P team. Overview of the Blue Gene/P project. *IBM Journal of Research and Development*, 52(1/2), January/March 2008.

[5] A. Bright, M. Ellavsky, A. Gara, R. Haring, G. Kopcsay, R. Lembach, J. Marcella, M. Ohmacht, and V. Salapura. Creating the BlueGene/L supercomputer from low power SoC ASICs. In *Digest of Technical Papers, 2005 IEEE International Solid-State Circuits Conference*, pages 188–189, 2005.

[6] E. Duesterwald, C. Cascaval, and S. Dwarkadas. Characterizing and predicting program behavior and its variability. In *PACT 2003*, September 2003.

[7] M. Martonosi, D. Clark, and M. Mesarina. The SHRIMP performance monitor: Design and applications. In *ACM SIGMETRICS Symposium on Parallel and Distributed Tools*, May 1996.

[8] B. Matthews, J.-D. Wellman, and M. Gschwind. Exploring real time multimedia content creation in video games. In *Media and Streaming Processors MSP*, December 2004.

[9] L. Mindlin, J. Brunheroto, and J. E. Moreira. Obtaining hardware performance metrics for the Blue Gene/L supercomputer. *Springer Lecture Notes in Computer Science*, 2790/2004, 2003.

[10] T. Mytkowicz, P. F. Sweeney, M. Hauswirth, and A. Diwan. Time interpolation: So many metrics, so few registers. In *MICRO 40*, December 2007.

[11] PAPI performance monitoring API. http://icl.cs.utk.edu/papi.

[12] V. Salapura, R. Bickford, M. Blumrich, A. A. Bright, D. Chen, P. Coteus, A. Gara, M. Giampapa, M. Gschwind, M. Gupta, S. Hall, R. A. Haring, P. Heidelberger, D. Hoenicke, G. V. Kopcsay, M. Ohmacht, R. A. Rand, T. Takken, and P. Vranas. Power and performance optimization at the system level. In *ACM Computing Frontiers 2005*, Ischia, Italy, May 2005. ACM Press.

[13] V. Salapura, R. Walkup, and A. Gara. Exploiting workload parallelism for performance and power optimization in Blue Gene. *IEEE Micro*, 26(5), September 2006.

[14] B. Sprunt. The basics of performance-monitoring hardware. *IEEE Micro*, pages 64–71, July-August 2002.