Obtaining Hardware Performance Metrics for the BlueGene/L Supercomputer

Pedro Mindlin, José R. Brunheroto, Luiz DeRose, and José E. Moreira

IBM Thomas J. Watson Research Center
Yorktown Heights, NY 10598-0218
{pamindli,brunhe,laderose,jmoreira}@us.ibm.com

Abstract. Hardware performance monitoring is the basis of modern performance analysis tools for application optimization. We are interested in providing such performance analysis tools for the new BlueGene/L supercomputer as early as possible, so that applications can be tuned for that machine. We are faced with two challenges in achieving that goal. First, the machine is still going through its final design and assembly stages and, therefore, it is not yet available to system and application programmers. Second, and most important, key hardware performance metrics, such as instruction counters and Level 1 cache behavior counters, are missing from the BlueGene/L architecture. Our solution to those problems has been to implement a set of nonarchitected performance counters in an instruction-set simulator of BlueGene/L, and to provide a mechanism for executing code to retrieve the value of those counters. Using that mechanism, we have ported a version of the libHPM performance analysis library. We validate our implementation by comparing our results for BlueGene/L to analytical models and to results from a real machine.

1 Introduction

The BlueGene/L supercomputer [1] is a highly scalable parallel machine being developed at the IBM T. J. Watson Research Center in collaboration with Lawrence Livermore National Laboratory. Each of the 65,536 compute nodes and 1,024 I/O nodes of BlueGene/L is implemented using system-on-a-chip technology. The node chip contains two independent and symmetric PowerPC processors, each with a 2-element vector floating-point unit. A multi-level cache hierarchy, including a shared 4 MB level-2 cache, is part of the chip. Figure 1 illustrates the internals for the BlueGene/L node chip.

Although it is based on the mature PowerPC architecture, there are enough innovations in the BlueGene/L node chip to warrant new work in optimizing code for this machine. Even if we restrict ourselves to single-node performance, understanding the behavior of the memory hierarchy and floating-point operations is necessary to optimize applications in general.

The use of analysis tools based on hardware performance counters [2,5,6,7,10,12,13, 14,15] is a well established methodology for understanding and optimizing the performance of applications. These tools typically rely on software instrumentation to obtain the value of various hardware performance counters through kernel services. They also rely on kernel services that *virtualize* those counters. That is, they provide a different set

H. Kosch, L. Böszörményi, H. Hellwagner (Eds.): Euro-Par 2003, LNCS 2790, pp. 109–118, 2003.

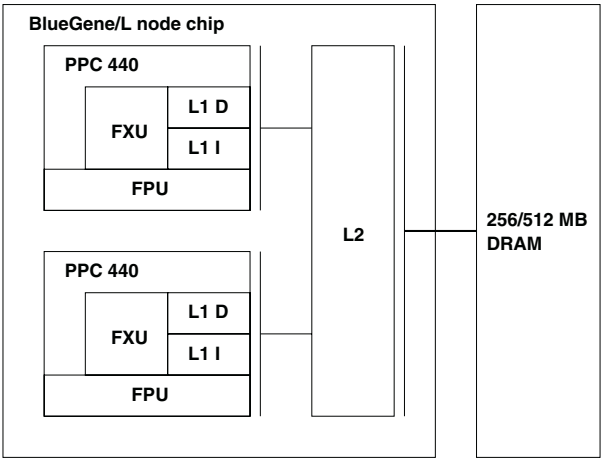

Fig. 1. Internal organization of the BlueGene/L node chip. Each node consists of two PowerPC model 440 CPU cores (PPC 440), which include a fixed-point unit (FXU) and 32KB level-1 data and instruction caches (L1 D and L1 I). Associated with each core is a 2-element vector floating-point unit (FPU). A 4MB shared level-2 unified cache (L2) is also included in the chip. 256 or 512 MB of DRAM (external to the chip) complete a BlueGene/L node.

of counters for each process, even when the hardware only implements a single set of physical counters. The virtualization also handles overflowing of counters by creating virtual 64-bit counters from the physical (which are normally 32-bit wide) counters.

We plan to provide a version of the libHPM performance analysis library, from the Hardware Performance Monitor (HPM) toolkit [10], on BlueGene/L. We believe this will be useful to applications developers in that machine, since many of them are already familiar with libHPM in other platforms. However, a deficiency of the performance data that can be obtained from the BlueGene/L machine is the limited scope of its available performance counters. The PowerPC 440 core used in BlueGene/L is primarily targeted at embedded applications. As is the case with most processors for that market, it does not support any performance counters. Although the designers of the BlueGene/L node chip have included a significant amount of performance counters outside the CPU cores (*e.g.*, L2 cache behavior, floating-point operations, communication with other nodes), there are critical data not available. In particular, counters for instruction execution and L1 cache behavior are not present in BlueGene/L.

Hence, our solution to providing complete hardware performance data for BlueGene/L applications relies on our architecturally accurate system simulator, BGLsim. This simulator works by executing each machine instruction of BlueGene/L code. We have extended the simulator with a set of non-architected performance counters, that are updated as each instruction executes. Since they are implemented in the internals of the simulator, they are not restricted by the architecture of the real machine. Therefore, we can count anything we want without disturbing the execution of any code. Per process and user/supervisor counters can be implemented directly in the simulator, without any

counter virtualization code in the kernel. Finally, we provide a lightweight architected interface for user- or supervisor-level code to retrieve the counter information. Using this architected interface, we have ported libHPM to BlueGene/L. As a result, we have a well-known tool for obtaining performance data implemented with a minimum of perturbation to the user code. We have validated our implementation through analytical models and by comparing results from our simulator with results from a real machine.

The rest of this paper is organized as follows. Section 2 presents the BGLsim architectural simulator for BlueGene/L. Section 3 discusses the specifics of implementing all layers needed to support a hardware performance monitoring tool on BGLsim, from the implementation of the performance counters to the modifications needed to port libHPM. Section 4 discusses the experimental results from using our performance tool chain. Finally, Section 5 presents our conclusions.

2 The BGLsim Simulator

BGLsim [8] is an architecturally accurate instruction-set simulator for the BlueGene/L machine. BGLsim exposes all architected features of the hardware, including processors, floating-point units, caches, memory, interconnection, and other supporting devices. This approach allows an user to run complete and unmodified code, from simple self-contained executables to full Linux images. The simulator supports interaction mechanisms for inspecting detailed machine state, providing more monitoring capabilities beyond what is possible with real hardware. BGLsim was developed primarily to support development of system software and application code in advance of hardware availability. It can simulate multi-node BlueGene/L machines, but in this paper we restrict our discussion to the simulation of a single BlueGene/L node system.

BGLsim can simulate up to 2 million BlueGene/L instructions per second when running on a 2 GHz Pentium 4 machine with 512 MB of RAM. We have found the performance of BGLsim adequate for the development of system software and for experimentation with applications. In particular, BGLsim was used for the porting of the Linux operating system to the BlueGene/L machine.

We routinely use Linux on BGLsim to run user applications and test programs. This is useful to evaluate the applications themselves, to test compilers, and to validate hardware and software designs choices. At this time, we have successfully run database applications (eDB2), codes from the Splash-2, Spec2000, NAS Parallel Benchmarks, and ASCI Purple Benchmarks suites, various linear algebra kernels, and the Linpack benchmark.

While running an application, BGLsim can collect various execution data for that application. Section 3 describes a particular form of that data, represented by performance counters. BGLsim can also generate instruction traces that can be fed to trace-driven analysis tools, such as SIGMA [11].

BGLsim supports a *call-through* mechanism that allows executing code to communicate directly with the simulator. This call-through mechanism is implemented through a reserved PowerPC 440 instruction, which we call the call-through instruction. When BGLsim encounters this instruction in the executing code, it invokes its internal call-through function. Parameters to that function are passed in pre-defined registers. De-

pending on those parameters, specific actions are performed. The call-through instruction can be issued from kernel or user mode. Examples of call-through services provided by BGLsim include:

- *File transfer*: transfers a file from host machine to simulated machine.
- *Task switch*: informs the simulator that the operating system (Linux) has switched to a different process.
- *Task exit*: informs the simulator that a given process has terminated.
- *Trace on/off*: controls the acquisition of trace information.
- *Counter services*: controls the acquisition of data from performance counters.

As we will see in Section 3, performance monitoring on BGLsim relies heavily on call-through services. In particular, we have modified Linux to use call-through services to inform BGLsim about which process is currently executing. BGLsim then uses this information to control monitoring (*e.g.*, performance counters) associated with a specific process.

The PowerPC 440 processors supports two execution modes: *supervisor* (or *kernel*) and *user*. A program that executes in supervisor mode is allowed to issue privileged instructions, as opposed to a program running in user mode. BGLsim is aware of the execution mode for each instruction, and this information is also used to control monitoring. By combining information about execution mode with the information about processes discussed above, BGLsim can segregate monitoring data by user process and, within a process, by execution mode.

3 Accessing Hardware Performance Counters on BGLsim

The BlueGene/L node chip provides the user with a limited set of hardware performance counters, implemented as configurable 32- or 64-bit registers. These counters can be configured to count a large number of hardware events but there are significant drawbacks. The number of distinct events that can be simultaneously counted is limited by chip design constraints, therefore restraining the user's ability to make thorough performance analysis with a single application run. Moreover, the set of events to be simultaneously counted cannot be freely chosen, and some important events, such as level-1 cache hits and instructions executed, cannot be counted at all.

As an architecturally accurate simulator, BGLsim should only implement the hardware performance counters actually available in the chip. However, we found that BGLsim allowed for much more: the creation of a set of non architected counters, which could be configured to count any set of events at the same time.

The BGLsim non architected performance counters were organized as two sets of 64 counters. The counters are 64-bit wide. Pairs of counters, one in each set, count the same event, but in separate modes: one set for user mode and the other set for supervisor mode. Furthermore, we have implemented an array of 128 of these double sets of counters, making it possible to assign each double set to a different Linux process id (PID), thus acquiring data from up to 128 distinct Linux concurrent processes.

To provide applications with performance data, we had first to enable access to the BGLsim non architected performance counter infra-structure to executing code. This

was accomplished through the call-through mechanism described in Section 2. Through the counter services of the call-through mechanism, executing code can reset and read the values of the non architected performance counters. We used the task switch and task exit services (used by Linux) to switch the (double) set of counters being used for counting events. We also used the knowledge of execution mode to separate counting between the user and supervisor sets of counters. That way, each event that happens (*e.g.*, instruction execution, floating-point operation, load, store, cache hit or miss), is associated with a specific process and a specific mode.

To make the performance counters easily accessible to an application, we defined the *BGLCounters* application programming interface. This API defines the names of the existing counters in BGLsim to the application, and creates wrapper functions that, in turn, call the corresponding call-through services with the appropriate parameters. The BGLCounters API provides six basic functions: initialize the counters, select which counters will be active (usually all), start the counters, stop the counters, reset the counters, and read the current values of the counters.

Finally, we ported libHPM to run on BGLsim. This port was done by extending the library to use the BGLCounters API, adding support for new hardware counters and derived metrics that are related to the BlueGene/L architecture, such as the 2-element vector floating-point unit, and by exploiting the possibility of counting both at user mode and at supervisor mode during the same execution of the program. Figure 2 displays a snapshot of the libHPM output from the execution of an instrumented CG code, from the NAS Parallel Benchmarks [4]. It exhibits the feature of providing the counts for both user and kernel activities, which, in contrast with other hardware counters based tools, is a unique feature of this version of libHPM.

```
Counter                                        User     Kernel

BGL_INST (Instructions completed)      :  308099072    9715300
BGL_LOAD (Loads completed)             :   31607898    1055210
BGL_STORE (Stores completed)           :    1207204     593399
BGL_D1_HIT (L1 data hits (l/s))        :   85123481    1633792
BGL_D1_L_MISS (L1 data load misses)    :   17041434     166502
BGL_D1_S_MISS (L1 data store misses)   :     301437      51876
BGL_I1_MISS (L1 instruction misses)    :      10406       6930
BGL_D2_MISS (L2 data load misses)      :        149        150
BGL_DTLB_MISS (Data TLB misses)        :     107001        201
BGL_ITLB_MISS (Instruction TLB misses) :       2283          0
BGL_FP_LOAD (Floating point loads)     :   66920747        198
BGL_FP_STORE (Floating point stores)   :    2730585        198
BGL_FP_DIV (Floating point divides)    :        780          0
BGL_FMA (Floating point multiply-add)  :   32662140          0
BGL_FLOPS (Floating point operations)  :   33755085          0
```

Fig. 2. libHPM output for the Instrumented CG code running on BGLsim.

4 Experimental Results

In order to validate BGLsim's hardware performance counters mechanism, we performed two sets of experiments. The first validation was analytical, using the ROSE [9] set of micro-benchmarks, which allows us to estimate the counts for specific metrics related with the memory hierarchy, such as loads, stores, cache hits or misses, and TLB misses. In the second set of experiments we used the serial version of the NAS Parallel Benchmarks and generated code for BlueGene/L and for an IBM Power3 machine, using the same compiler (GNU gcc and g77 version 3.2.2). We ran these programs on BGLsim and on a real Power3 system, collecting the hardware metrics with libHPM. With this approach, we were able to validate metrics that should match in both architectures, such as instructions, loads, stores, and floating point operations.

4.1 Analytical Comparison

The Blue Gene memory hierarchy parameters used in this analysis are: level-1 cache: 32 Kbytes with line size of 32 bytes; level-2 cache: 4 Mbytes with line size of 128 bytes; Linux page size: 4 Kbytes; TLB: 64 entries, fully associative.

We used seven functions from the ROSE micro-benchmark set: (1) *sequential stores*, (2) *sequential loads*, (3) *sequential loads and stores*, (4) *random loads*, (5) *multiple random loads*, (6) *matrix transposition*, and (7) *matrix multiplication*. The first five functions use a vector D with $n = 2^{20}$ double precision elements (8 Mbytes). The functions random loads and multiple random loads also use a vector, `I`, of 1024 integers (4 Kbytes), containing random indices in the range $(1..n)$. We also executed these two functions (and sequential loads) using a vector D_2 with twice the size of D. Finally, we used matrices of 64×64 double precision elements (32 Kbytes each) in the matrix functions. The transposition function only uses one matrix, while the multiplication uses three matrices. We called a function to evict all matrix elements between the two calls (transpose and multiply). Those functions perform the following operations (where c is a constant, and s is a double precision variable):

Sequential stores: $D[k] = c$, for $k = 1..n$.
Stores 1M double precision words (8 Mbytes) sequentially; we expect approximately 2K page faults, generating 2K TLB misses. Each page has 128 Level 1 and 32 Level 2 lines. Hence, this function should generate 256K store misses at Level 1 and 64K store misses at Level 2.

Sequential loads: $s = s + c * D[k]$, for $k = 1..n$.
Loads 1M double precision words. Again we expect approximately 2K page faults, 2K TLB misses, 256K Level 1 load misses, and 64K Level 2 misses.

Sequential loads and stores: $D[k] = D[k] + c * D[k-1]$, for $k = 2..n$.
Loads 2M double precision words and stores 1M double precision words. Since it is loading adjacent entries and always storing one of the loaded entries, we should expect only 1 miss for every 8 loads, with a total of 256K load misses in Level 1 and no store misses. In Level 2 we expect one fourth of the misses in Level 1 (64K).

Random loads: $s = s + D[I[k]], k = 1..1024$.
Access 1K random elements of the vector D. We expect approximately 1K TLB

misses and 1K load misses at level 1. D has twice the Level 2 cache size and was just traversed. Hence, accesses to its second half should hit in Level 2 cache. Since we are dealing with random elements, we would expect that approximately half of these accesses would cause misses. We also expect to have an additional 2 TLB misses, 128 Level 1 misses, and 32 Level 2 misses, due to the sequential access to I. When running this function using D_2, we expect the same number of TLB and Level 1 misses, but approximately $3/4$ of the accesses should miss in Level 2, since D_2 is four times the size of the Level 2 cache.

Multiple random loads: $s = s + D[I[k] + 16 * j], j = 0..31, k = 1..1024$
For each entry of I it accesses 32 consecutive entries from D, with a stride of 16, which corresponds to four Level 1 cache lines and one Level 2 cache line. we expect again one Level 1 miss and $1/2$ Level 2 miss for each load, with a total of 16K Level 1 misses and 8K Level 2 misses, but only 2K TLB misses overall[1]. A similar behavior from the one described for the function Random loads is expected for the Index vector I and the runs using D_2.

Matrix transposition: This function executes the operations shown in Figure 3a, where `N` is 64. It performs $2 * (64^2 - 64) = 8064$ loads and the same number of stores. Since the matrix fits exactly in the Level 1 cache, we expect approximately 1 load miss for every 8 loads, and no store misses. Also, since the matrix has 32 Kbytes, we expect 9 TLB misses.

Matrix multiplication: Executes the operations shown in Figure 3b, assuming the matrix `B` is transposed. It loads matrices `A` and `B` 64 times, and matrix `C` once. Hence, it performs $129 * 64^2$ loads and 64^2 stores. We would expect 8 TLB misses for each matrix, with an extra TLB miss to compensate for not starting the arrays at a page boundary. In each iteration the outer loop traverses `B` completely. Since `B` has exactly the size of the level-1 cache, and one row of `A` and `C` are also loaded per iteration, there would be no reuse of `B` in subsequent iterations of the outer loop. `B` should generate $(64^2)/4$ Level 1 misses in each iteration of the outer loop. The rows of of `A` and `C` are reused during the iterations of the two inner loops. This function should generate approximately $(64 + 1 + 1) * 64^2/4$ level-1 misses (66K). Since all three matrices fit in Level 2, we expect to have approximately $3 * 64^2/16$ Level 2 misses (768).

```
for (i = 0; i < N; i++)
 for (j = 0; j < N; j++)
  if (i != j) {
   aux = B[i][j];
   B[i][j] = B[j][i];
   B[j][i] = aux;
}
```

(a)

```
for (i = 0; i < N; i++)
 for (j = 0; j < N; j++)
  for (k = 0; k < N; k++)
   C[i][j] += A[i][k]*B[j][k];
```

(b)

Fig. 3. Pseudo-codes for matrix transposition (a) and matrix multiplication (b)

[1] Unless the first access to D hits the beginning of a page, a page boundary will be crossed during the span of 32 accesses, causing the second TLB miss for each entry of I.

Table 1 summarizes the expected values for each function and presents the error (difference) between observed and expected counts. In all cases the measured values are close to the expected numbers, with the largest differences being less than 2%. Also, the expected numbers for functions random loads and multiple random loads should be viewed as upper bounds, because they deal with random entries. The larger the vector D gets, the closer the measured value should be from these bounds.

Table 1. Expected counts and error (difference) between expected and observed counts for the micro-benchmark functions.

	Stores				Loads						Total		TLB	
	FP		L1 Misses		FP		FX		L1 Misses		L2 Misses		Misses	
functions	exp.	error	exp	error	exp.	error	exp	error	exp.	error	exp	error	exp.	error
seq stores	1M	0	256K	+1	0	+1	0	0	0	+1	64K	+8	2K	+1
seq loads	0	0	0	0	1M	+1	0	0	256K	+2	64K	+4	2K	+2
seq ld & st	1M	-1	0	0	2M	-1	0	0	256K	+2	64K	+8	2K	+1
random ld D	0	0	0	0	1K	0	1K	0	1152	-53	544	+8	1K	-37
random ld D_2	0	0	0	0	1K	0	1K	0	1152	-53	800	+3	1K	-37
mult rnd ld D	0	0	0	0	32K	0	1K	0	32896	-244	16416	+242	2K	-23
mult rnd ld D_2	0	0	0	0	32K	0	1K	0	32896	-367	24608	+119	2K	-31
transpose	8064	0	0	0	8064	0	0	0	1K	+298	252	+14	9	0
matrix mult	4K	0	0	0	516K	0	0	0	66K	+1153	768	+3	25	-1

4.2 Validation Using a Real Machine

Table 2 shows the counts for the 8 codes in the NAS Parallel Benchmarks. (We used the serial version of these benchmarks, which run on a single processor.) The programs were compiled with the GNU gcc (IS) and g77 (all the others) compilers, with `-O3` optimization flag. We ran the Class S problem sizes for the serial versions of the benchmarks on both an IBM Power3 system and on BGLsim. Due to the architectural differences in the memory hierarchy of the BlueGene/L and the Power3 systems, we restricted this comparison to metrics that should match in both architectures (floating-point operations, load, stores, and instructions).

We observe that for some benchmarks (*e.g.*, CG and IS) the agreement between Power3 and BGLsim is excellent, while for others (*e.g.*, EP and LU) the difference is substantial. We are investigating the source of those differences. It should be noted that, even though we are using the same version of the GNU compilers in both cases, those compilers are targeting different machines. It is possible the compilers are performing different optimizations for the different machines. As future work, we will pursue executing the exactly same object code on both targets, to factor out compiler effects.

We also compared the wall clock time for executing the 8 benchmarks on the Power3 machine and on BGLsim, running on a 600 MHz Pentium III processor. The wall clock slowdown varied from 500 (for IS) to 3000 (BT).

5 Conclusions

Performance analysis tools, such as the HPM toolkit, are becoming an important component in the methodology application programmers use to optimize their code. Central

Table 2. Counts from the NAS benchmark, when running on a Power3 and on BGLsim. Note that counts are in units of one thousand (K).

Code	System	Counters								
		Instructions	FX loads	FP loads	Total loads	FX stores	FP stores	Total stores	FP ops	FMAs
BT	Power3	603,174K			215,921K			92,015K	228,617K	13,671K
	BGLsim	530,599K	335K	157,081K	157,417K	9,993K	80,923K	90,916K	210,607K	11,475K
	difference	12.03%			27.10%			1.19%	7.88%	16.06%
CG	Power3	306,865K			98,616K			3,970K	33,762K	32,663K
	BGLsim	308,099K	31,607K	66,920K	98,528K	1,207K	2,730K	3,937K	33,755K	32,662K
	difference	0.4%			0.09%			0.83%	0.02%	0.00%
EP	Power3	5,893,189K			1,248,195K			363,302K	2,501,740K	391,379K
	BGLsim	6,267,337K	403,626K	743,087K	1,146,714K	617,816K	338,413K	956,230K	1,431,830K	350,281K
	difference	6.35%			8.13%			50.28%	42.77%	10.50%
FT	Power3	551,146K			109,204K			97,260K	160,806K	37,226K
	BGLsim	524,995K	7,585K	95,145K	102,730K	2,917K	93,361K	96,278K	147,363K	37,225K
	difference	4.74%			5.93%			1.01%	8.36%	0.00%
IS	Power3	8,767K			2,008K			1,352K		
	BGLsim	8,770K	2,007K		2,007K	1,351K		1,352K		
	difference	0.03%			0,01%			0,00%		
LU	Power3	291,671K			94,758K			27,351K	108,157K	18,560K
	BGLsim	233,270K	4,432K	66,677K	71,109K	8,860K	27,798K	30,659K	77,100K	15,470K
	difference	20.02%			24.96%			12.09%	28.79%	16.65%
MG	Power3	20,827K			5,072K			2,581K	5,517K	367K
	BGLsim	16,956K	271K	3,860K	4,131K	1,257K	1,131K	2,388K	3,516K	367K
	difference	18.59%			18.54%			7.47%	36.27%	0.00%
SP	Power3	276,419K			88,658K			28,278K	78,299K	13,884K
	BGLsim	263,325K	726K	71,592K	72,318K	2,274K	26,893K	29,168K	75,812K	13,884K
	difference	4.74%			18.43%			3.15%	3.18%	0.00%

to those performance tools is hardware performance monitoring. We are interested in providing such performance analysis tools for the new BlueGene/L supercomputer as early as possible, so that applications can be tuned for that machine.

In this paper, we have shown that an architecturally accurate instruction-set simulator, BGLsim, can be extended to provide a set of performance counters that are not present in the real machine. Those nonarchitected counters can be made available to running code, and integrated with a performance analysis library (libHPM), providing detailed performance information on a per process and per mode (user/supervisor) basis.

We validated our implementation through two sets of experiments. We first compared our results to expected values from analytical modes. We also compared results from our implementation of libHPM in BlueGene/L to results from libHPM in a real Power3 machine. Because of differences in architectural details between the BlueGene/L PowerPC 440 processor and the Power3 processor (caches size and associativity, TLB size and associativity), we restricted our comparison to values that should match in both architectures (numbers of floating-point operations, load, stores, instructions). Through both sets of experiments, we have shown coherent results from libHPM in BGLsim in some codes from the NAS Parallel Benchmarks.

Our implementation of libHPM on BGLsim has already been successfully used to investigate the performance of an MPI library for BlueGene/L [3]. We want to make it even more useful by extending it to measure and display multi-chip performance information. Given the limitations of performance counters in the real BlueGene/L hardware, we expect that libHPM in BGLsim will be useful even after hardware is available. The simultaneous access to a large number of performance counters, and consequently to

a large number of derived metrics, allows the user of BGLsim to obtain a wide range of performance data from a single application run. Most hardware implementations of performance counters limit the use to just a few simultaneous counters, forcing the user to repeat application runs with different configurations.

The BGLsim approach can be used as a tool for architectural exploration. We can verify the sensitivity of performance metrics of applications to various architectural parameters (*e.g.*, cache size), and guide architectural decisions for future machines.

References

1. N. Adiga et al. An Overview of the BlueGene/L Supercomputer. In *Proceedings of SC2002*, Baltimore, Maryland, November 2002.
2. D. H. Ahn and J. S. Vetter. Scalable Analysis Techniques for Microprocessor Performance Counter Metrics. In *Proceedings of SC2002*, Baltimore, Maryland, November 2002.
3. G. Almasi, C. Archer, J. G. Castaños, M. Gupta, X. Martorell, J. E. Moreira, W. Gropp, S. Rus, and B. Toonen. MPI on BlueGene/L: Designing an efficient general purpose messaging solution for a large cellular system. Technical report, IBM Research, 2003.
4. 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.
5. S. Browne, J. Dongarra, N. Garner, K. London, and P. Mucci. A Scalable Cross-Platform Infrastructure for Application Performance Tuning Using Hardware Counters. In *Proceedings of Supercomputing'00*, November 2000.
6. B. Buck and J. K. Hollingsworth. Using Hardware Performance Monitors to Isolate Memory Bottlenecks. In *Proceedings of Supercomputing'02*, November 2000.
7. J. Caubet, J. G. J. Labarta, L. DeRose, and J. Vetter. A Dynamic Tracing Mechanism for Performance Analysis of OpenMP Applications. In *Proceedings of the Workshop on OpenMP Applications and Tools – WOMPAT 2001*, pages 53–67, July 2001.
8. L. Ceze, K. Strauss, G. Almasi, P. J. Bohrer, J. R. Brunheroto, C. Caşcaval, J. G. C. nos, D. Lieber, X. Martorell, J. E. Moreira, A. Sanomiya, and E. Schenfeld. Full circle: Simulating linux clusters on linux clusters. In *Proceedings of the Fourth LCI International Conference on Linux Clusters: The HPC Revolution 2003*, San Jose, CA, June 2003.
9. L. DeRose. The Reveal Only Specific Events (ROSE) Micro-benchmark Set for Verification of Hardware Activity. Technical report, IBM Research, 2003.
10. L. DeRose. The Hardware Performance Monitor Toolkit. In *Proceedings of Euro-Par*, pages 122–131, August 2001.
11. L. DeRose, K. Ekanadham, and J. K. Hollingsworth. SIGMA: A Simulator Infrastructure to Guide Memory Analysis. In *Proceedings of SC2002*, Baltimore, Maryland, November 2002.
12. C. Janssen. *The visual Profiler. http://aros.ca.sandia.gov/~cljanss/perf/vprof/.* Sandia National Laboratories, 2002.
13. J. M. May. MPX: Software for Multiplexing Hardware Performance Counters in Multithreaded Programs. In *Proceedings of 2001 International Parallel and Distributed Processing Symposium*, April 2001.
14. M. Pettersson. *Linux X86 Performance-Monitoring Counters Driver.* `http://user.it.uu.se/~mikpe/linux/perfctr/`. Uppsala University – Sweden, 2002.
15. Research Centre Juelich GmbH. *PCL - The Performance Counter Library: A Common Interface to Access Hardware Performance Counters on Microprocessors.* `http://www.fz-juelich.de/zam/PCL/`.