Fast Full-System Execution-Driven Performance Simulator for Blue Gene/Q

Diego S. Gallo[1,⋆], Jose R. Brunheroto[2], and Kyung Dong Ryu[2]

[1] IBM Research Brazil, Sao Paulo, SP 04007-900, Brazil
[2] IBM T.J. Watson Research Center, Yorktown Heights, NY 10598, USA
dsgallo@br.ibm.com,{brunhe,kryu}@us.ibm.com

Abstract. Full-system execution-driven simulators are essential tools in the development of supercomputers, such as IBM Blue Gene/Q. They enable software teams to develop and run code before the real system becomes available, typically a few years after the beginning of the project. Functional simulators support early software development. The addition of timing information allows for early performance assessment of applications to provide feedback on the hardware and system design. The techniques employed to implement a timing model for Blue Gene/Q, built on top of the functional model, are presented. Our simulator runs several orders of magnitude faster than traditional cycle-accurate simulators. The experiments with micro-kernels from the Sequoia Benchmark Suite demonstrate that our simulator provides the timing accuracy between 3 to 17% of the actual measurement from the real Blue Gene/Q machine. We also present some architecture design exploration and its performance implications.

1 Introduction

A full-system simulator is an essential tool in the development process of a supercomputer. Cycle-accurate models are important for hardware validation and verification of the architecture being designed before spending all the cost and time building a chip. Those models also help in power consumption and reliability analysis. Since the level of detail needed to be simulated is very high, cycle-accurate models are both time-consuming to develop and very slow to run in software-based simulators (often a mere 10-100 processor cycles per second [1]).

On the other hand, functional models are faster to develop, run faster, and can be built before all the hardware details are fully specified, providing support for the system software, compiler, and applications teams to start developing and testing their softwares long before any hardware prototype becomes available. That alleviates the dependencies among teams, allowing them to work in parallel, and consequently reducing the overall development cycle, although not much can be done using the purely-functional simulator to assure that the performance of those "components" (system software, compiler, and applications) will be

⋆ Work done while at IBM T.J. Watson Research Center.

F. Wolf, B. Mohr, and D. an Mey (Eds.): Euro-Par 2013, LNCS 8097, pp. 16–27, 2013.

satisfactory. Due to that, a large optimization effort usually takes place after the hardware becomes available, leading to either unnecessary delays to deliver the system or a poor performance when the system is initially delivered.

It is common to have a cycle-accurate model developed for software-based simulators as soon as the hardware is fully specified, as such simulation is used for hardware verification. Nevertheless, it does not provide much value for the software teams, because the time it takes just to boot the firmware and kernel is already prohibitive, and often it is not feasible to run any applications with minimally useful problem sizes. Consequently, the use of cycle-accurate software-based simulations in any optimization effort is extremely hard.

Alternatively, FPGA-based VHDL simulators can be used to provide reasonably fast cycle-accurate simulation. Although these simulators are important for running the validation tests faster, and even to run, test and optimize software, they take longer to be available due to the need for complete VHDL code, and are not as versatile as software-based simulators, since to change its behavior, one would have to alter the VHDL, which is usually non-parametrical. Additionally, FPGA simulators are expensive and typically in short supply.

Contrarily, software-based simulators are usually extremely flexible through parameterization, allowing a large variety of experiments to be performed through parameter exploration. A list of contributions from this work is itemized below:

- Methodology to build a timing model on top of a functional model, extending previous work [2] to model multi-threaded processors;
- Implementation of a timing model for the Blue Gene/Q (BG/Q) hardware, reasonably accurate for application performance analysis;
- Support architectural design decisions through parameter exploration, comparing the performance in different scenarios before writing any VHDL; and
- Fine-grained performance information about application execution without the need for instrumenting the applications.

In Section 2 the BG/Q system is briefly described, followed by a description of the Mambo simulator and the BG/Q functional model. Our proposed timing model is presented in Section 3, with validation of the model in Section 4. The simulation execution speeds of the different models (with increasing level of detail) are shown in Section 5, and some use cases of our full-system event-driven performance simulator, both in terms of profiling and what-if experiments, are shown in Section 6. Finally, Section 7 concludes this work.

2 Blue Gene/Q model on Mambo Simulator

This section briefly describes the BG/Q system itself, followed by a description of the functional model implemented on Mambo to simulate the system.

2.1 BlueGene/Q System Overview

The BG/Q system [8, 10] comprises compute racks with two midplanes. Each midplane has 16 node cards, with 32 nodes assembled on each card. That adds

up to 1024 nodes per rack, providing a dense concentration of processing power. A five-dimensional (5D) torus network connects the compute nodes, providing node-to-node communication and integrating hardware-assisted barrier and collectives functionality onto the same network [4].

Each node contains 18 IBM PowerPC A2 cores, of which 16 are used to run application code, one is used to offload services provided by the lightweight kernel operating system CNK (Compute Node Kernel) [7], and one is a spare core used for increasing the yield during the chip manufacturing process. Figure 1 shows a block diagram of the A2 core components and the relationships among them.

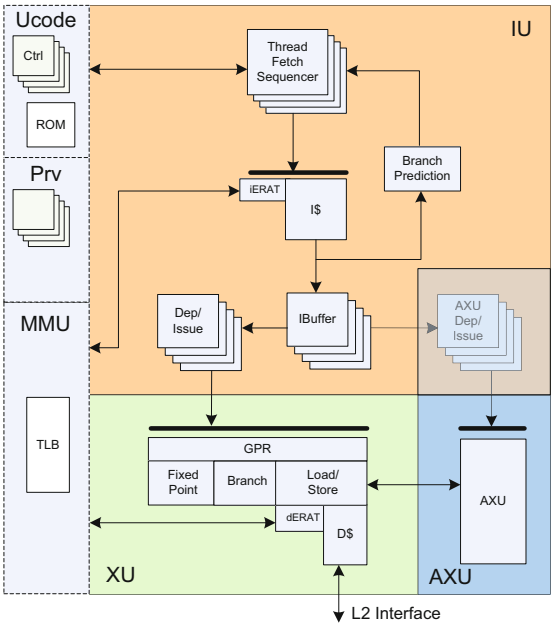


Fig. 1. BG/Q A2 core block diagram

These 1.6GHz cores have four in-order execution hardware threads each, that combined can issue up to two instructions per cycle: one integer instruction to the eXecution Unit (XU) and one floating-point instruction to the Auxiliary eXecution Unit (AXU). The Instruction Unit (IU) is responsible for fetching instructions into each thread instruction buffer, predicting branches direction, checking for register dependencies, and arbitrating between the threads that have an instruction ready to be issued.

The AXU is a 4-way Quad-vector Processing Unit (QPU), that implements the Quad Processing eXtension (QPX) to the Power ISA [6]. Due to the QPU, four multiply-and-add double-word floating-point operations can be executed in a SIMD (Single Instruction Multiple Data) way, providing 8 flops/cycle per core, summing up 204.8 Gflop/s of peak performance on each compute node.

As for the memory subsystem, each A2 core has its own private L1 instruction and data caches (16KB each). The node has a shared L2 cache of 32MB that is connected to 16GB of physical main memory. Additionally, each core has a prefetcher engine (L1P) with 32 entries of 128 bytes that sits between the L1 and L2 to prefetch data and to serve as a write-back buffer.

2.2 Functional Model of Blue Gene/Q on Mambo

The Mambo simulator [3] currently provides support for modeling and simulating the execution of a multitude of systems, including PowerPC, Cell, A2, and embedded processors, among others. Its modular and configurable design provides building blocks that can be re-used, as well as a scheduling mechanism that efficiently handles the context switch between any sort of tasks one can create (to handle processors, threads, and any other types of resources). This design allows focusing on the system characteristics instead of the simulation platform when modeling a system.

The purely functional model of BG/Q on Mambo comprises the A2 core model (its instruction set running on four hardware threads) combined with the instruction set provided by the QPX. All the instructions are implemented in an architecturally-accurate manner, providing bit-accurate results. Additionally, a memory subsystem with all the cache levels of the BG/Q system (private L1 cache and L1P prefetcher, shared L2 cache, and main memory) can be enabled, providing a first set of performance metrics in the form of hit/miss rates through the different cache levels. When the memory hierarchy model is enabled in the simulator, we say that it is running in *warmup* mode.

Even though the hardware threads are implemented in the functional model, each with its register file and instruction pointer, there is no notion of the execution pipelines or contention in instruction issue for resources that are shared among the threads. One instruction from each thread can be issued at every cycle, respecting only locks and barriers for the correctness of the execution.

Consequently, even though the memory subsystem with the cache hierarchy is implemented, providing statistics regarding hit/miss rates, these may be inaccurate since the execution time of the instructions is not respected. That can cause differences in cache behavior, since the four threads inside a core interfere with each other at the L1 and L1P level, and the seventeen cores inside a node share the L2 cache. The next section describes our model that not only solves this issue, but also provides a large variety of additional information about the simulated execution, allowing a better understanding of application performance.

3 Timing Model for Blue Gene/Q

On top of the purely functional model and the memory subsystem previously described, and based on previous work about tracking resource dependencies for a pseudo-cycle-accurate timing model for Blue Gene/L [2], a timing model for the BG/Q compute node was developed extending the previous work twofold:

First, extending the framework to support multi-threaded cores, since the A2 core has four hardware threads as opposed to the single-threaded PowerPC 440 on BG/L, and second, considering in our model the resources that mostly impact system performance, such as the thread instruction-fetch sequencer, the instruction buffer and the dynamic branch predictor from the Instruction Unit, and the Load Miss Queue (LMQ) and Store Queue (SQ) from the memory subsystem.

The proposed timing model relies on three main factors: time-stamping the most relevant resources (e.g. scalar, floating-point, and special-purpose register files; XU and QPU execution units); checking time-stamps of dependent resources before issuing instructions (waiting until all the resources are available); and updating time-stamps of all used resources after executing an instruction with the corresponding time when each resource becomes available.

This mechanism guarantees that an instruction will be issued if and only if the register dependencies are satisfied and the necessary execution unit is available, limiting the instruction execution rate by the two most restrictive requirements. The update of dependent resources after an instruction is executed is based on:

1. *Instruction latency*: Number of cycles needed between instructions that have register dependencies, i.e. that use a register that was a target in a previous instruction (for BG/Q, usually 1 or 2 cycles for XU instructions and 6 to 8 cycles for QPU instructions). Time-stamps of target registers are updated to reflect the instruction latencies.
2. *Instruction throughput*: Specified as the number of cycles in which a unit will be busy for the execution of a given instruction. Time-stamps of the XU or QPU execution unit will be updated accordingly (usually available in the next cycle, except for instructions that are not fully pipelined), e.g. some multiply instructions and micro-coded instructions).

The write-back latencies (i.e., the interval from when the instruction is issued until the resulting value is written back to the register) for arithmetic and logic instructions are usually directly dependent on the number of stages in the pipeline, except for instructions that re-enter the pipeline (e.g. multiply), and micro-coded instructions (e.g. divide), while the write-back latencies for load instructions depend on the memory subsystem. To avoid the need for a cycle-accurate memory subsystem, which would slow down the whole simulation and defeat the initial goal of having a really fast performance simulator, our model uses the average latencies to each cache level and to main memory. For every load, the average latency corresponding to the level of the hierarchy in which the data is located (i.e. L1, L1P, L2, or main memory) is used to estimate the load latency. The results in Section 4 show that using this approximation regarding the load latency still allows reasonable accuracy simply by modeling the cache hierarchy and the contention caused by the LMQ and SQ.

Additionally, since the execution units are shared among the four hardware threads, which may be competing for the same unit, their use have to be constrained respecting the round-robin policy for threads with the same priority. By incrementing the time-stamp of the XU or QPU according to the instruction

throughput and using the Mambo simulator infrastructure to schedule the next events in two steps, firstly the threads that are waiting for dependencies, and secondly the threads that just executed, the round-robin policy is respected.

Moreover, the thread instruction-fetch sequencer and instruction buffer were implemented in our model to correctly handle the timing implications of fetching instructions ahead of time. In the A2 core, each thread has an instruction buffer that can hold up to eight instructions (32 bytes). At each cycle, the Instruction Unit can begin fetching instructions for one thread. A group of four instructions (16 byte-aligned bytes) is fetched, and instructions will be discarded in case the address to fetch is not at the beginning of this group. Additionally, instructions after a branch will also be discarded if the group contains a predicted-taken branch. The thread that will fetch instructions in a given cycle is chosen in a round-robin manner, and a thread has high priority if its instruction buffer is completely empty and there is no fetch request in flight.

Furthermore, the branch prediction mechanism was also implemented in the model to correctly reflect misprediction penalties in the execution time. On BG/Q, conditional branches with a hint are statically predicted via their hints, and all the other conditional branches are predicted using a gshare-like dynamic branch prediction mechanism that remembers prior branch directions using a Branch History Table (BHT). The BHT contains 1024 entries, 2 bits each, that are incremented for taken branches (saturating at three) and decremented for not-taken branches (saturating at zero). A branch is predicted as taken if the counter is two or three, and not-taken otherwise. Finally, to index the BHT, the lowest address bits of the instruction are XORed with a per-thread Global Branch History Register (GBHR), which helps in correctly predicting interleaved branches. In the event of a mispredicted branch, a flush is generated in the pipeline and there will be a minimum of 13 cycles from when the branch instruction is fetched (or issued) until the correct target is fetched (or issued).

Lastly, the LMQ and SQ are also important points of contention in the BG/Q node that had to be modeled for timing correctness, since they can cause threads to stall. Each core has an eight-entry LMQ, shared among the four threads, that holds load misses and non-cacheable loads while they are outstanding to the L2, allowing the thread to continue issuing instructions (provided that the target register from the load is not used). Additionally, a thread will stall if issuing a load request that misses the L1 when the LMQ is full.

Regarding store instructions, there are no store buffers in the A2 core. Stores are sent directly to the L1P, where they are queued and wait for arbitration in the L2 crossbar switch (having priority over load requests during the arbitration). Additionally, while in the L1P, two stores (8 bytes each) to adjacent and aligned memory locations can be combined to improve memory bandwidth. If the SQ is full, a thread issuing a store request will stall until one entry is successfully consumed by the crossbar switch. The SQ on BG/Q has 20 entries, modeled in our timing model to reflect contention caused due to store instructions.

The parameters in the timing model (e.g. latency to caches and memory, depth of pipelines, number of execution pipelines, sizes of LMQ and SQ) can

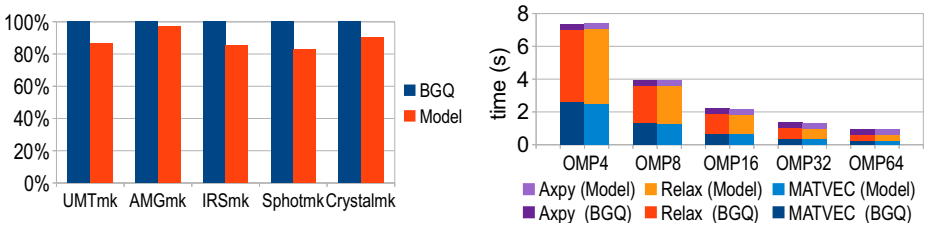
be configured, providing a way to quickly experiment with what-if scenarios. The next sections show the accuracy of the implemented model for BG/Q, its simulation speed compared to the purely functional model, and results from use cases, including advanced profiling and what-if experiments.

4 Timing Model Validation

In order to validate the model, we ran experiments with the microkernels (UMTmk, AMGmk, IRSmk, SPhotmk, and Crystalmk) from the Sequoia Benchmark suite [9] to cover multiple applications with different characteristics (e.g., different instruction mixes, memory access patterns, and SIMD utilization). These microkernels were developed at Lawrence Livermore National Lab to represent some of the main attributes of applications that make use of their supercomputers, providing meaningful performance numbers in more realistic scenarios than just the LINPACK (LINear algebra PACKage) benchmark [5] and others.

All the microkernels were used to validate the single-thread accuracy of the timing model. Additionally, since AMGmk has OpenMP support, we validated the accuracy of our model for the entire BG/Q node running it using up to 16 cores and 64 hardware threads. It is worth mentioning that for measuring the accuracy of our timing model, the absolute performance of the codes was not relevant. Thus, we simply used the reference benchmark codes without any optimization other than optimizations provided by the compiler.

Figure 2a shows the accuracy of our model for each of the five Sequoia microkernels. Time reported by our Mambo timing model for each of them is normalized to the time measured running the application using BG/Q hardware. Accuracy of the model for these benchmarks vary from -3% (AMGmk) up to -17% (SPhotmk). Additionally, Figure 2b illustrates the accuracy of our model when running the OpenMP version of the AMGmk while varying the number of threads. Across the different configurations, accuracy stays within 5% for each of the three phases of this microkernel (MATVEC, Relax, and Axpy).



(a) (b)

Fig. 2. Mambo timing model accuracy for the Sequoia microkernels (a) and time spent on each phase of AMGmk varying the number of OpenMP threads (b)

Note also that the AMGmk benchmark parallelizes loops both in the MATVEC and Relax phases through the use of OpenMP pragmas, decreasing the time spent in each phase when increasing the number of threads (i.e. 4, 8, 16, 32, and 64

threads). Contrarily, time spent in the Axpy phase is constant independently of the number of OpenMP threads, because the reference code does not specify any parallelism in that phase.

5 Timing Model Execution Performance

As mentioned and shown earlier in this paper, our timing model allows developers to obtain reasonably accurate performance information before all the hardware details are completely specified and any VHDL is written. Additionally, it is much faster than software-based cycle-accurate models, and more flexible than FPGA-based simulators.

Table 1 illustrates the average simulation execution speed for the Sphotmk benchmark running in a single thread (except for the MESA cycle-accurate value, which is a reference value). It compares the speed among the different models and different levels of detail, and also the real hardware.

It is worth mentioning that the row named "profiler enabled" shows the slowdown in the simulation when enabling our profiling functionality. Also, speed comparison should be done on an instruction-per-second basis, otherwise one would get the impression that our timing model is actually faster than the purely functional model, because it simulates more "cycles per second". That is only an artifact due to some instructions advancing the cycle counter by multiple cycles because of different dependencies and contention. Additionally, while the A2 core in the real BG/Q hardware runs at 1.6 GHz, all the other values are averages measured over the simulation executions, because different instructions take different amounts of time to simulate.

Table 1. Simulation execution speed for the different models

	Average simulation execution speed	
	cycles / sec	**instructions / sec**
Real BG/Q hardware	(1.6 GHz)	(375,434,609)
Purely functional	1,682,411	1,077,807
Warmup (cache hierarchy)	1,501,073	961,636
Proposed timing model	2,570,047	677,792
Profiler enabled	1,793,757	473,063
MESA cycle-accurate simulator [1]	(10-100)	(2-20)

Note that in the purely functional mode and the warmup mode, the difference between the number of cycles and the number of instructions is simply due to the fact that you can specify the maximum possible throughput for each instruction, with some instructions taking multiple cycles to execute. Nevertheless, it does not take into account any of the register and load dependencies, instruction buffering, branch misprediction penalties or anything else, basically allowing the issue of one instruction per thread per cycle at all times, thus leading to an extremely optimistic CPI (Cycles Per Instruction) value.

Additionally, when the application uses multiple cores in a single node, there is a near-linear slowdown in the simulation speed (cycles/sec), because there

is more work to be simulated, except for the MESA cycle-accurate simulator, since it is always simulating the full node (all the cores), even when there are no instructions being issued on any of the other cores, because it is an accurate representation of the hardware based on the VHDL, simulating the propagation of every clock cycle through the entire node. Looking into the number of instructions executed per second in each mode, the warmup mode is verified to have slowed the simulation by 11%, while the slowdown due to the timing model was 37%. Enabling the profiler slowed the simulation a total of 56%. Nevertheless, the slowdown is minimal when compared with the simulation speed of the software-based cycle-accurate simulator (MESA), which is four to five orders of magnitude slower in this case (single-threaded application). Even comparing the speed when using all 64 threads, the cycle-accurate simulation is still 3 to 4 orders of magnitude slower than our timing model.

6 Use Cases for the Timing Model

In this section, two use cases enabled by the proposed timing model are illustrated: *application profiling* and *what-if experiments*. The first allows developers to obtain performance information as fine-grained as they would like, since the application execution is not disturbed by the collection of information. The second allows hardware architects to assess the impact of different architectural decisions, providing performance information for hypothetical scenarios.

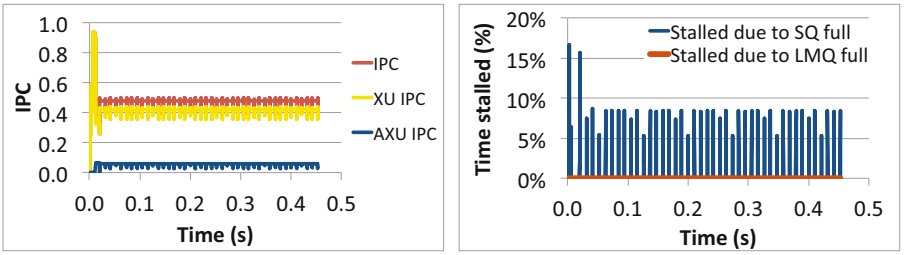
6.1 Application Profiling

The functional model of the simulator provides some basic profiling capabilities, allowing the collection of the instruction mix from applications. Enabling the cache hierarchy model allows additionally collecting hit/miss rates at the different cache levels, but since each thread may execute one instruction per cycle in that mode without any contention due to the execution units, memory access, register dependencies, or anything else, the hit/miss rates might be inaccurate for some applications (e.g. unbalanced multi-threaded applications).

The timing model proposed herein adds new profiling capabilities to the simulator, therefore allowing the collection of fine-grained performance information without the need for instrumenting the application in ways that could alter its behavior (since fine-grained instrumentation often generates a prohibitive level of overhead and disturbance in the application).

To illustrate the profiling capabilities our timing model adds to the simulator, Figure 3a shows the IPC (Instructions Per Cycle) values for a section of the AMGmk benchmark, in both the XU and QPU execution units, and Figure 3b shows the percentage of time the application stalled due to a load request that misses the L1 when the LMQ is full (almost zero for this benchmark) and due to a store request when the SQ is full (as high as 16.7%, i.e., 16,700 cycles stalled due to SQ full in a 100,000 cycles sampling).

Developers can easily track the performance of an application with the information provided by our timing model, gaining insight into the parts of the



(a) (b)

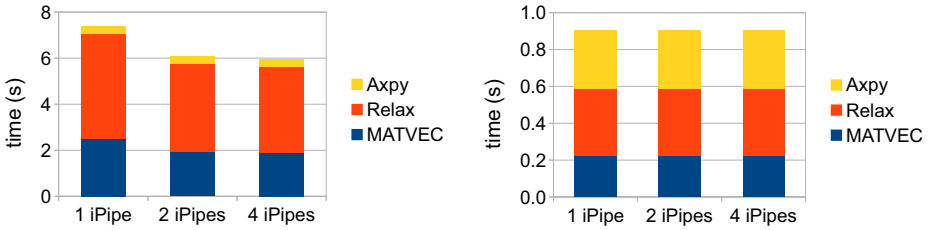
Fig. 3. Average IPC over each sample interval (a) and percentage of time stalled due to LMQ or SQ full (b) throughout the application execution

applications that should be optimized, and the causes that are leading to poor performance (e.g. what sections of the application saturate the memory, filling up the LMQ or SQ, and what sections saturate one of the execution pipelines).

6.2 What-if experiments

Another interesting use case for our timing model is the possibility to experiment with slightly different architectural designs, assessing their performance implications. This helps evaluating tradeoffs during the concept phase of the project. Since the compiler has not been changed to take advantage of the different architectural designs being explored, the performance results shown in this subsection might be underestimated. Nevertheless, they are still noteworthy.

Figure 4a evaluates the impact of having multiple integer execution pipelines (iPipes), simulating the execution of the AMGmk benchmark, using 4 OpenMP threads, configuring the A2 core model to have 1, 2 or 4 iPipes.



(a) AMGmk OMP4 timing (b) AMGmk OMP64 timing

Fig. 4. Impact on timing if A2 core had multiple integer pipelines

Note that adding a second iPipe in the A2 core would lead to a performance gain both in the 'Relax' and 'MATVEC' phases of AMGmk, achieving a 20% reduction in the total execution time. The reason is that a second iPipe would alleviate stalls allowing concurrent instruction issue by multiple threads. However, the experiment shows that more than 2 iPipes would not help. Nonetheless, it is also interesting to note that when using 16 cores for the application (having 64

OpenMP threads), the benefits of having multiple iPipes would disappear due to other bottlenecks, as shown in Figure 4b.

Another experiment evaluates the impact of different store queue sizes on AMGmk performance. Figure 5 shows that if a store queue with half the size of the current queue in the system (i.e. 10 entries instead of 20 entries) was used, the total execution time of the benchmark would increase by more than 50% and the time spent on the MATVEC phase alone would more than double. On the other hand, if we had twice as many entries in the store queue (i.e. 40 entries), the performance gain would be minimal (only 3%) for this benchmark.

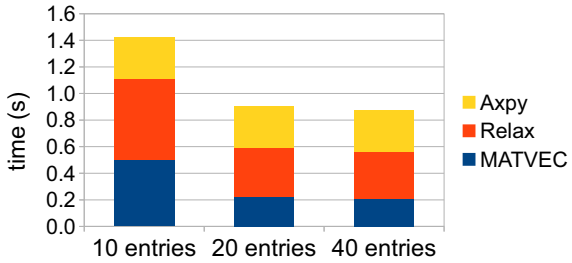


Fig. 5. Impact on AMGmk OMP4 timing if A2 had different store queue sizes

7 Conclusions

During the early phases of a supercomputer architecture definition, or any new processor architecture design, a full-system execution-driven performance simulator can provide fairly accurate information about the applications performance in such new hardware. Additionally, it allows design space exploration over different architectural parameters (e.g. number of execution units inside the core, size of queues and buffers, cache hierarchy and memory latencies), providing data to help evaluate tradeoffs (e.g. chip cost, area and power consumption vs. performance gain).

After the architecture is defined and the hardware starts to be produced, the timing model can provide valuable fine-grained information regarding the execution, allowing not only the analysis of subsections of the application without the need to instrument it, which would possibly alter its behavior, but also providing detailed information of where exactly the application is stalling due to busy execution units, register dependencies, load dependencies, or full load-miss or store queues. In that way, we augmented the capabilities provided by the BG/Q performance counters, allowing the simulator to collect performance information that cannot otherwise be collected by the hardware without interfering with the execution.

All the fine-grained information provided by our timing model, added to the reasonably fast simulation speed (tens of thousands times faster than the software-based cycle accurate simulator), makes it also a very helpful tool for optimizing application performance.

Acknowledgment. The authors would like to thank Martin Ohmacht for the detailed information about memory subsystem, Bob Walkup for pointing out the correct libraries and compilers, Fred Mintzer and Dave Singer for providing us access to the BG/Q system, and Bryan Rosenburg for all the help in improving this paper. The BG/Q project has been supported and partially funded by the Argonne National Laboratory and the Lawrence Livermore National Laboratory on behalf of the US Department of Energy, under Lawrence Livermore National Laboratory subcontract number B554331.

References

1. Asaad, S., Tierno, J., Bellofatto, R., Brezzo, B., Haymes, C., Kapur, M., Parker, B., Roewer, T., Saha, P., Takken, T.: A cycle-accurate, cycle-reproducible multi-FPGA system for accelerating multi-core processor simulation. In: Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays, pp. 153–161. ACM Press, New York (2012)
2. Bachega, L., Brunheroto, J., DeRose, L., Mindlin, P., Moreira, J.: The BlueGene/L pseudo cycle-accurate simulator. In: Proceedings of the 2004 IEEE International Symposium on Performance Analysis of Systems and Software, pp. 36–44. IEEE Computer Society, Washington, DC (2004)
3. Bohrer, P., Simpson, R., Speight, E., Sudeep, K., Van Hensbergen, E., Zhang, L., Peterson, J., Elnozahy, M., Rajamony, R., Gheith, A., Rockhold, R., Lefurgy, C., Shafi, H., Nakra, T.: Mambo - A Full System Simulator for the PowerPC Architecture. ACM SIGMETRICS Perform. Eval. Rev. 31(4), 8–12 (2004)
4. Chen, D., Eisley, N.A., Heidelberger, P., Senger, R.M., Sugawara, Y., Kumar, S., Salapura, V., Satterfield, D.L., Steinmacher-Burow, B., Parker, J.J.: The IBM Blue Gene/Q interconnection network and message unit. In: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 1–10. ACM Press, New York (2011)
5. Dongarra, J.J., Luszczek, P., Petitet, A.: The LINPACK Benchmark: past, present and future. Concurrency and Computation: Practice and Experience 15(9), 803–820 (2003)
6. Fox, T., Gschwind, M., Moreno, J.: QPX Architecture: Quad Processing eXtension to the Power ISA. Tech. rep., IBM Research, Yorktown Heights, NY (2012)
7. Giampapa, M., Gooding, T., Inglett, T., Wisniewski, R.W.: Experiences with a Lightweight Supercomputer Kernel: Lessons Learned from Blue Gene's CNK. In: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 1–10. IEEE Computer Society, Washington, DC (2010)
8. Haring, R.A., Ohmacht, M., Fox, T.W., Gschwind, M.K., Sugavanam, K., Coteus, P.W., Heidelberger, P., Blumrich, M.A., Wisniewski, R.W., Gara, A., Chiu, G.L.T., Boyle, P.A., Christ, N.H., Kim, C.: The IBM Blue Gene/Q Compute Chip. IEEE Micro 32(2), 48–60 (2012)
9. Lawrence Livermore National Laboratories: ASC Sequoia Benchmark Codes, `https://asc.llnl.gov/sequoia/benchmarks/`
10. The Blue Gene Team: Blue Gene/Q: by co-design. Computer Science - Research and Development (2012)