iQ: an efficient and flexible queue-based simulation framework

Damian Roca, Daniel Nemirovsky,
Marc Casas, Miquel Moreto, and Mateo Valero
Barcelona Supercomputing Center (BSC-CNS)
and Universitat Politècnica de Catalunya (UPC)
Barcelona, Spain
Email: {name}.{surname}@bsc.es

Mario Nemirovsky
ICREA Senior Research Professor at BSC-CNS
Barcelona, Spain
Email: mario.nemirovsky@bsc.es

***Abstract*—Conventional system simulators are readily used by computer architects to design and evaluate their processor designs. These simulators provide reasonable levels of accuracy and execution detail but suffer from long simulation latencies and increased implementation complexity. In this work we propose iQ, a queue-based modeling technique that targets design space exploration and optimization studies at the core component level. iQ emulates processor elements by abstracting the implementation details into modular components composed of queue structures, delay parameters, probabilistic driven message generation and event control. Its easy reconfigurability makes iQ a highly flexible and powerful processor simulator. We have used iQ to build an Ivy Bridge and a Core 2 Duo processor model and have validated them against real hardware running SPEC CPU2006 Int achieving average error rates of 9.55% and 8.93%.**

I. Introduction

Computer architects use several simulation tools in order to design, evaluate and optimize computing systems. However, the wide variety of simulation tools makes the selection process a non-trivial task. While some techniques are based on analytical models [1], [2], others rely on the use of system simulators [3] for bottleneck identification and design verification.

Full system simulators such as Gem5 [3] are currently commonplace. Other researchers have been using trace-driven simulators [4] in an effort to reduce the execution time while maintaining accuracy. Furthermore, researchers use representative reduced traces [5], [6] which capture the workload behavior. These tools are excellent for detailed simulations but they are cumbersome in dealing with the initial stages of design space exploration due to the lengthy simulation time and substantial development effort involved.

In contrast, there are other tools more suitable for design space exploration [7], [8]. They reduce the simulation time required while maintaining accuracy, but the underlying complexity remains the same. Thus, another abstraction level is required. Analytical models should cover this area but are oversimplified. To improve the outputs of these models, some researchers [9], [10], [11], [12] have used queuing theory [13] to construct multi-threaded processors models to analyze resource contention without focusing on the processor implementation. However, finer granularity simulation is often desired when pinpointing micro-architectural bottlenecks or exploring diverse design parameters and components.

To meet this challenge, architects use a two-stage process. In the first stage, a high-level simulator is used for the design space exploration analysis. The bottleneck identification and the performance improvement estimation obtained guide the second stage, a more detailed simulation to test and validate component designs. In this work we present a fine-grained queue-based simulator, iQ, to be used in the first stage. The main requirements for iQ are a large complexity abstraction and fast simulations, while maintaining the error within acceptable boundaries. To satisfy these needs, we based our framework on queue theory and statistical information. The combination of these techniques allows us to represent any processor component or functionality with queues, servers, delays, and communication lines. While the queues correspond to the need of handling an instruction flow, the delays are the representation of the required time to perform an action over an instruction. Although there are previous works on both areas, our intent is to provide a generic framework which it is detailed enough to be used as the default tool for fast design space exploration.

To represent applications, a dynamic instruction flow is generated based on a statistical profile formed by the instruction's distribution probability and register dependency information. Once the profile is available, it is time to build a processor model based on the queue elements. To construct processor models, architectural information is required, which can be determined easily for an existent processor (ALUs, ROB length, etc) or in a new design the researcher defines these parameters. With the profile and the model, architects can study the impact of new components and/or analyze the bottlenecks on these models with simulations that take a few seconds, and most important modifications are feasible in real-time due to iQ's abstraction level. Later we demonstrate that accuracy is not lost to gain simulation speed.

In this paper, we first provide a detailed description of iQ's characteristics. Although our technique can be used to simulate any computer architecture (including processors, GPUs, and FPGAs), we implement and validate an Ivy Bridge and a Core 2 Duo model against real hardware. A design space analysis is

 DOI: 10.1109/MASCOTS.2017.13

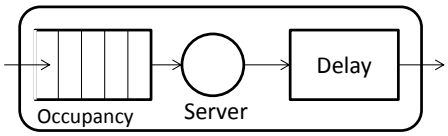


Fig. 1: Generic modular queue structure. It is sequential and formed by three elements: a queue, a server, and a delay.

TABLE I: Generic queue structure configuration for different processor components

Processor component	Configuration
Non-pipelined Integer ALU	Server=ALU latency Delay=0
Pipelined Memory ALU	Server= 1 cycle Delay=mALU latency - 1
Partially pipelined Cache level	Server=non pipelined latency Delay=Cache latency-server

then presented which showcases the usefulness iQ provides for architects, saving them time and effort by quickly identifying bottlenecks and revealing improvement options.

Contributions:

- New simulation paradigm of splitting the one simulator based technique into a queuing model plus full system simulator.
- Simulation methodology based on queue models and statistical information for design space exploration and bottleneck identification at the core component level.
- iQ's performance evaluation and validation against real hardware

II. iQ MODEL

A. Background

In the case of an arithmetic unit such as an Integer ALU in a processor, the queue can model the ALU input queue where the message represents an arithmetic instruction such as an add, and the service time added is the time the ALU unit takes to execute that arithmetic instruction. Dependencies between messages (i.e., instructions) or within computational resources (e.g., ALUs, branch predictors, Out-of-Order tracking) are also accounted to model the performance of the system characteristics being modeled.

A probabilistic model can also be included to emulate non-deterministic behavior such as branch miss-predictions and cache hits and misses. A collection of discrete events drives the execution simulation, in representation of computational cycles. Total performance is measured as a collection of processed messages per total events, in other words, instructions per cycle.

B. Queuing Model

Processors are formed by a wide variety of components such as functional units and memory levels. To represent them using queuing model we have implemented a modular queue structure that is capable of representing different behaviors through a set of variable configurations. This module, represented in Figure 1 is formed by a queue, a server, and a delay. The users can configure the queue length and the delays required to process instructions. The **Queue occupancy** models the resource contention and availability.

Server: This parameter is used to model the time to execute the proper function over the instructions. The service time is the latency required to process an instruction. While an instruction is being serviced, the subsequent instructions wait in the queue.

Delay: This parameter is used to complement the Server latency to ensure the appropriate total delay for the component the instruction will pass through. In this manner we ensure that the combination of Service and Delay time is used to represent any structure, pipelined or not.

Table I shows latency configurations to achieve the desired structure. For instance, a fully pipelined structure, such as Multiply ALU with a four cycles latency where a new instruction may begin execution each cycle. Then the total execution time is the sum of a Service time (once cycle) and a Delay time (three cycles). Each cycle the server processes a new instruction but their total execution time is 4 due to the delay.

Time-line: An event (i.e., cycle) is used to not only keep track of the number of cycles elapsed, but also tracks and schedules instruction events to maintain proper execution flow. For instance, assuming instruction A enters an ALU (which takes four cycles to compute) at cycle 42. Then, an action at cycle 46 is scheduled which will move instruction A to the following stage of execution. Performance is measured by dividing the total number of retired instructions by total elapsed cycles (i.e., IPC). The end of the simulation is reached when the variation between different IPC intervals is negligible and thus we consider the IPC is stabilized. The amount of time until the IPC reach that point is variable but usually is within tens of seconds.

C. Modeling hardware and software characteristics

Hardware: Conceptually, architects need to have a high level view of a processor (like the 5-stage pipeline) to determine which basic modules are required to emulate the behavior of a processor. Identifying the processor components to include in the model will determine the extra modules. For example, each ALU or memory level can be included with a generic module. To reduce the development effort and the simulation time, iQ models do not require knowing all the specific details but must only capture the main behavior of each component. For example, in constructing a cache module, details such as size and number of lines do not need to be included, the cache module can still provide accurate results only being configured to know the hit/miss ratios and corresponding latencies.

Software: To simulate instructions, iQ uses instruction types which are user defined classes resulting in a pseudo-ISA. This technique eliminates the complexity and necessity of using binaries and compilers specific to our simulator. Applications must be profiled using hardware counters and tools like Pin [14]. This process enables architects to gauge the makeup of the application's instruction mix (e.g., arithmetic,

memory, and branch) and register dependencies (distance between the creation and the use of value). iQ uses the profile to feed an instruction generator module that creates a representative code dynamically during the simulation. For instance, iQ uses a random number generator to produce different instructions types based on a probability distribution given by the application profile.

D. iQ Advantages

The easily configurable and modular nature of the modules in iQ allows a great flexibility in emulating the behavior of different processors' components. An important step iQ takes towards improving performance analysis is to quickly and accurately evaluate the usefulness of the proposed architectural solutions before expending significant amounts of time and effort in modifying conventional simulators. As a consequence, the relation between the different components and their impact on total performance can be evaluated in real-time.

Accessibility : We use Omnet++ [15] to construct and simulate different hardware models. It provides an intuitive simulation environment and support for the libraries containing the generic queue modules. A configuration file controls all the hardware and software parameters. A public release of the simulator is available at GitHub [16]. Furthermore, iQ is being used within the H2020 project dRedBox [17] in their design space exploration analysis for disaggregated data centers with remarkable results.

III. Building an illustrative iQ model

A. Simulation setup

Target Architecture: We have used the iQ simulator to construct and simulate a model of the Intel(R) Core(TM) i7-3740QM CPU (Ivy Bridge). We evaluate a single core running single threaded applications. The architectural specifications for the Intel Ivy Bridge are publicly available [18], [19].

Host machine: We run our simulator on a Dell Latitude E6430 laptop. The processor is an Ivy Bridge with four cores and 8 GB of DDR3 RAM. Simulation accuracy and execution time are the two main characteristics evaluated. To provide a detailed and fair simulation evaluation, we compare our Ivy Bridge iQ model against the real processor through the Instructions per Cycle (IPC).

Benchmarks: We evaluated our simulator running the SPEC CPU2006 Int benchmarks [20], except omnetpp since the dependency profiling tool was not capable of executing it and astar due to a segmentation fault in the Core 2 Duo. To obtain the application profile, we used the hardware counters via perf and Pin on a system OpenSuse 13.1 and gcc 4.8.11. We classify the different executable instructions to fall within one of four iQ's instruction class types which we have defined: Int, Load, Store, and Branch. We used the MICA tool [21] to obtain the register dependency distance between instructions.

B. iQ Ivy Bridge modules

Our processor's model structure is based on the 5-stage pipeline. Fetch, Control (joins Decode and Issue), Execution, Memory, and Retirement are modeled using iQ based modules detailed below.

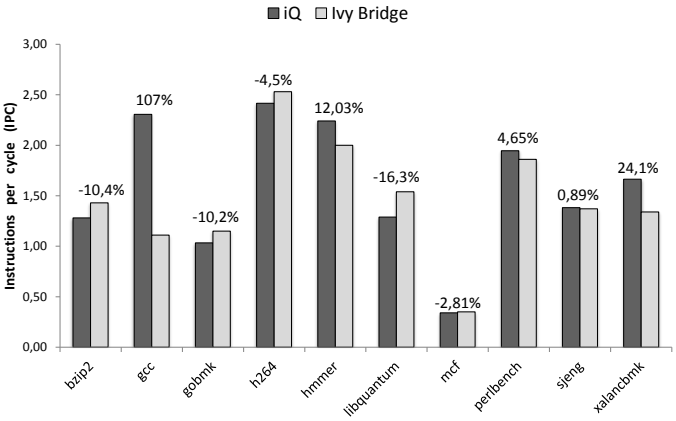
1) Fetch: The fetch module represents the fetch stage and the L1 instruction cache (i-cache), plus the dynamic generation of instructions. An application can be composed of several phases with different profiles, and the architect can specify the phase execution order. In the SPEC CPU2006 Int case we observed almost a flat profile during the execution, corroborating [22]. In consequence, we defined a single phase profile information. The parameters required to categorize each phase include: (i) the distribution of different instruction types and (ii) and the dependency information. The fetch module generates instructions based on this information.

An important parameter to represent the fetch stage accurately is the number of instructions per cycle that a real chip is capable of processing, which for the Ivy Bridge case is four [19]. Then, this module generates four instructions on each request.

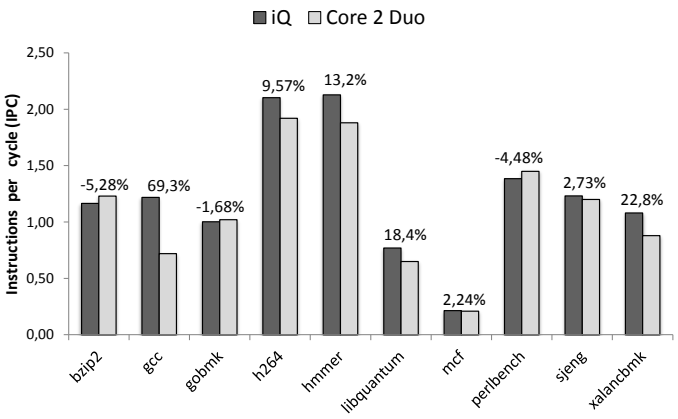
Icache. Since we model the L1 i-cache, the simulator needs to determine whether a memory operation results in a cache hit or miss. This hit/miss ratio is set in the iQ configuration file. A random value is used to determine a hit or a miss according to the range obtained applying the miss probability to the the desired distribution. In this case, we assume all the instruction misses go to the shared L3 cache, and thus we apply the LLC latency (28 cycles [19]). If it is a hit, in the next cycle four instructions will be sent to the next module.

2) Control: This module is responsible for emulating the decode/issue stages and out-of-order execution, including processing of dependency checks and branch predictions. Instructions received from the fetch module are stored in the ready queue waiting to be processed. Similar to the fetch stage, the important parameter for modeling the decode stage is the number of decoded instructions per cycle that the processor can deliver. Before issuing an instruction to the modules emulating the execution stage, the control module must check the dependency information to determine whether the instruction will be blocked due to interdependencies or due to lack of free computational resources such as ALUs, Ld/St queue and ROB entries.

Modeling Dependencies. A consequence of representing the instructions with messages which do not include register information is that the register renaming and the pool of available registers have to be emulated with statistical information. To achieve this objective and also to collect insightful information, we use two queues. The first queue tracks the instructions under execution inside the processor. The second queue tracks the instructions blocked in this stage due to dependency reasons. To determine and control for inter-instruction dependencies, a dependency distance probability at the register level is utilized. Before issuing an instruction, a random number is generated which will determine if the instruction depends upon a previous instruction and what distance. The id of the instruction at the corresponding distance will be chosen as the one the current instruction depends upon. The current instruction becomes blocked until the instruction



(a) iQ model and real Ivy Bridge. The absolute average error rate without gcc is 9.55% (19.42% with gcc)



(b) iQ model and real Core 2 Duo. The absolute average error rate without gcc is 8.93% (14.96% with gcc)

Fig. 2: iQ accuracy: IPC comparison between iQ models and real processors with errors rates shown on top.

it depends on finishes execution.

Branch predictor. To predict branches, we use a similar method as with the instruction cache by generating a random number and checking whether it falls within the probability ranges of a true or false branch prediction. If the branch is correctly predicted, it is sent to the scheduler function which emulates the next stage in execution. On the other hand, if there is a miss-prediction, the pipeline is flushed by emptying the ready instruction queue and a penalty is applied to the next clock event. This penalty sums the cost of the pipeline's flush and the average memory access latency to fetch new instructions. In the Ivy Bridge model the value is set up at 60 cycles.

Issue. The scheduler function checks if there is a free functional unit able to execute the instruction. In the Ivy Bridge case, up to five instructions can be executed simultaneously: three integer instructions (or 2 integer and 1 branch) and two memory instructions (2 loads, 1 load and 1 store, or one of either type). If there is an available functional unit (FU), the scheduler issues the instruction for execution by sending it to the corresponding FU module. In case the FU is occupied, it leaves the instruction in the ready queue until the module becomes available. Out of order execution is simulated using the re-order buffer (ROB) length to define the number of instructions that the processor can examine inside the ready queue to find a suitable instruction to send. That length is reduced by taking into account the number of instructions under execution and also the blocked instructions.

3) Integer/Branch functional units: These functional units are capable of executing three integer instructions or two integer instructions and one branch. To emulate their functionality the generic compound module from Table I is used. The required time to execute these instructions types in the real processor is one cycle so the service time is configured to be one and the delay value is set to zero. The queue length is unbounded because it is controlled through the maximum distance between the oldest instruction and the one to be sent.

4) Memory hierarchy: The Intel Ivy Bridge processor is capable of executing two loads simultaneously, represented with two memory FUs. The cache miss ratios are specified for each memory level. iQ uses a random number to determine the outcome of the memory accesses. To execute the operation, a cache control module is required after the generic functional unit. After the memory instruction goes through the L1 d-cache modules, it arrives to the L1 control module. At this point, whether the memory access is a cache hit or miss is determined. If it is a hit, then the instruction is sent to the retirement module. In case of a miss, it is sent to the L2 cache module. The same procedure is applied for L2 and L3. Main memory accesses are treated differently because they always hit.

5) Retirement: The retirement module emulates the retirement stage of a processor. The processor model retires instructions in an out-of-order fashion since instructions are retired when they arrive. This does not affect the accuracy since iQ uses statistical profiles of the application and not real code. If a traditional trace was used instead, then the in-order retirement would have to be used. There is almost no difference on accuracy or simulation time using in-order or out-of-order retirement. Once the instruction is retired, its id is sent to the control module. The control module can then check possible dependencies on that id and proceed to execute those instructions. This module also collects statistics about the entire simulation such as the latency required to execute each instruction, histograms about queue occupancies, number of retired instructions separated by type, etc.

IV. Evaluation

A. Accuracy

Figure 2a presents the comparison between the simulated and real Ivy Bridge IPC values running the SPEC CPU2006 Int benchmarks. It also shows the percentage relative error between these values. Apart from gcc that will be explained separately, the average error rate for the other benchmarks

is 9.55%. This error is comparable to those obtained with cycle-accurate simulators specifically modified to match a specific platform [23], proving that iQ can modelate the key components of a processor. There is only one outlier, gcc. The dependency model has been identified as the source of error due to its current inability of representing long and frequent dependency chains that limit the performance of the processor for gcc. As future work, a more detailed dependency representation has to be implemented to allow users to simulate correlated dependencies.

B. iQ for another processor

To extend iQ's validation, we constructed a processor model based on the Intel Core 2 Duo [24]. The methodology is the same as with the previous model but the parameters of the predefined modules have been reconfigured based on the Core 2 Duo specifications. The application profile (instruction mix and register dependencies) is the same since this information is micro-architecture independent and used the same SPEC CPU2006 Int benchmarks. Figure 2b presents the IPC comparison between the model and the real chip. As with the Ivy Bridge, our Core 2 Duo model exhibits competitive accuracy rates, except for gcc (for similar reasons as in the previous model).

C. Simulation Speed

iQ's execution time is reduced by the fact that the model only needs to execute the application profile until the variation in the output IPC value is negligible and does not need to execute the entire program. In less than ten thousand cycles all the benchmarks have a stable output IPC, and thus the simulations can finish. After measuring the real CPU execution time required for simulating those cycles, the final IPC is obtained in 2 seconds on average with a maximum of 4.2, demonstrating a remarkable speedup over the times required by other simulators.

D. Comparison with other simulators

Table II compares the absolute average error (second column) and average simulation time (third column) between our iQ simulation platform and other state of the art proposals. To populate it, we used the numbers from the original papers. The third column presents the execution time for each simulation technique: Gem5 runs the test input set of the SPEC2006, Sniper runs the large set of Splash-2 [25], ZSim runs 50 billion instructions, analytical model runs 1 billion and iQ runs until the IPC is stable. Although different benchmark sets are used, Table II positions iQ with current techniques. We see that iQ is the fastest and provides very low error percentage.

Gem5 is a full system simulator which detail results in very slow but highly accurate simulations [23]. However, the development cost is high, precluding its use for design space exploration analysis. The simulation speed of Sniper is improved compared to full-system simulators like Gem5 and MARSS, in the range of a few MIPS, while the accuracy is relatively high. While Sniper's scope is to perform rapid and accurate simulations by using interval simulation, its flexibility is still limited since modifications take time and each simulation can take hours.

TABLE II: Comparison between iQ and other simulators based on the numbers provided in the original articles

Simulator	**Avg. error (%)**	**Avg. Sim time** (hours)
iQ	9.55	0,0005
Analytical [26]	13	0.055
ZSim [8]	9.7	1,12
Sniper [7]	19.8	6,94
MARSS [27]	15.5	86,80
Gem 5 [23]	13	69,4

ZSim is a suitable tool to perform accurate simulations, but not to perform design space exploration analysis because the execution time is not small enough and modifications still require significant efforts for such high-level studies. In Table II we show their performance values for the SPEC benchmarks obtained from their website [28].

Analytical models present similar simulation speeds to iQ but the resulting accuracy depends on the model's complexity. In Table II we show one of the latest models [26]. To achieve that accuracy, that model is based on interval simulation, checkpoints, and cache and predictors. However, analytical models provide average performance values which obscure the dynamic behavior. Instead, queue models capture the latency distribution. This information is crucial to perform optimal design space exploration analysis and modifications to the processor architecture itself (more ALUs, memory ports, etc).

Conversely, an iQ based processor model emulates the real behavior through abstractions. Despite this fact, the accuracy level outperforms nearly all other simulators and simulation time is better than that of complex analytical models. Thanks to the component abstractions and the instruction generator, iQ performs simulations within a few seconds compared to several hours.

V. Design space exploration analysis

This section demonstrates the practical usefulness of the iQ model and how it can be used by architects to run design level analysis and optimize their systems. A modern processor is a complex machine with many more factors and parameter values that should also be evaluated. This fact indicates that the number of simulations growths exponentially with the number of factors. For instance, conducting an experiment consisting of 7 different parameter values for these 5 different factors concurrently means 16807 simulations for each benchmark (9 hours to run).

Eight configurations out of this multi-factor study are presented in Table III and compared to the default Ivy Bridge architecture. They form a representative subset of solutions based on the number of factors involved and the improvement achieved for comparison reasons. Although there are more configurations that accomplish the objective, choosing the appropriate solution will depend on the cost functions accounting for power, area, and implementation feasibility.

TABLE III: Design space exploration: factor and parameter configurations with the resultant performance improvement

Configuration	**Fetch** (inst/cycle)	**RoB length** (entries)	**Branch penalty** (cycles)	**DRAM latency** (cycles)	**LLC latency** (cycles)	**Improvement** (%)
Default(-)	4	168	60	180	28	-
A	5	128	40	-	-	15.33
B	5	256	40	-	-	18.19
C	5	-	50	-	Miss ratio -10%	15.46
D	5	128	50	130	-	16.55
E	5	256	-	125	-	15.32
F	-	256	40	150	-	15.28
G	6	-	-	140	Miss ratio -5%	17.03
H	6	256	55	170	25	15.00

Regarding the results from Table III the only difference between configuration A and configuration B is the increase of the size of the RoB length in B to 256. A larger number of fetched instructions per cycle combined with a reduced branch penalty makes that more instructions are available for execution each cycle. Then, this fact is exploited by a larger RoB length which translates into an improvement of 18.19%, while A obtains 15.33%. It becomes the architect's job to make a cost-benefit analysis and determine whether this extra 2.86% improvement is worth the implementation and energy costs. Configuration G provides a similar improvement, a 17.03%, with a different combination of parameter values. Instead of modifying the RoB length and the branch penalty it reduces the LLC miss ratio by 5% and also reduces the DRAM latency. Improving the memory hierarchy reduces the time instructions are blocked due to pending memory requests and allows to exploit a larger instruction level parallelism (ILP).

Following with the memory improvements, configuration C includes a reduction of the LLC miss ratio by 10%. In this case, a less aggressive fetch is compensated with a reduced branch penalty. However, this configuration is not so optimal as the previous one and presents a 15.46% improvement. Knowing that the LLC cache miss ratio needs to be decreased the architect can then use a cache simulator to decide which cache scheme fulfills the new miss ratios. Also remarkable is the configuration H, where all the factors are slightly modified. In comparison with the previous configurations where specific components were targeted, H proves that a minimum enhancement in all the processor stages achieves a 15% improvement.

Other solutions are more aggressive, such as E. To achieve a drastic reduction of the DRAM latency may not seem feasible. However, new memory technologies may enable such breakthroughs and then the architect can estimate its impact. Once configuration E revealed its potential with a 15.32% improvement, the architect can iterate on top of it. Decreasing the RoB length following the premises from previous analysis to 128 and reducing the branch penalty by 10 cycles results in 16.55% improvement (configuration D).

VI. Related Work

Different techniques have been used to provide feasible design space exploration tools. SMARTS [29] provides reduced representative subsets of benchmarks to reduce the simulation time although the underlying processor model can compromise its advantages. To avoid the use of third tools, synthetic traces [1] can be generated recreating the original behavior from a previous execution reducing the simulation time. TaskPoint [30] applies sampled simulation to task-based programs. Cook et al. [31] developed a design space exploration technique based on Monte Carlo methods. Lee et al. [32] used a regression model to analyze the trade-offs between performance and power consumption.

Prior work has also used queue models to simulate multi-processor systems [11], [10], [33]. They exhibit a higher level of abstraction in the processor architecture, reduced to a traffic generator, because they focus on the multi-threaded contention problems. The first work only simulates the different cache levels of the memory hierarchy and how the requests access them. They do not simulate the ISA, focusing on the memory accesses. The second work adds more detail to the memory hierarchy implementing the bank scheme and the main memory, but the rest of the processor still remains hidden. iQ extends both works by implementing the remaining processor components, a more complete ISA, and defining a generic framework easily extensible to other processor architectures. Then, iQ can perform a fine-grain analysis of the entire processor.

Data center scale simulations have been performed using the queue methodology, both in performance [34] and in power [12], sharing the granularity problem of previous works.

VII. Conclusion

In this work, we have proposed iQ, a queue-based model simulator at the core components level that targets design space exploration studies. We have shown how iQ emulates processor components by abstracting the implementation details. Its modular nature and easy reconfigurability of the component parameters make iQ a highly flexible and powerful processor simulator.

Acknowledgments

The authors would like to thank Mauricio Breternitz, Rodolfo Milito, and Vasilis Karakostas for their helpful reviews. Damian Roca work was supported by a Doctoral Scholarship provided by Fundación La Caixa. This work has been supported by the Spanish Government (Severo Ochoa grants SEV2015-0493) and by the Spanish Ministry of Science and Innovation (contracts TIN2015-65316-P).

REFERENCES

[1] S. Nussbaum and J. Smith, "Modeling superscalar processors via statistical simulation," *Proceedings of PACT*, 2001.

[2] L. Eeckhout and K. D. Bosschere, "Hybrid analytical-statistical modeling for efficiently exploring architecture and workload design spaces," *Proceedings of PACT*, 2001.

[3] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, *et al.*, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.

[4] A. Rico, A. Duran, F. Cabarcas, Y. Etsion, A. Ramirez, and M. Valero, "Trace-driven simulation of multithreaded applications," in *Proceedings of ISPASS*, pp. 87–96, IEEE, 2011.

[5] T. Lafage and A. Seznec, "Choosing representative slices of program execution for microarchitecture simulations: A preliminary application to the data stream," *Workload characterization of emerging computer applications*, 2001.

[6] R. Wunderlich, T. Wenisch, B. Falsafi, and J. Hoe, "Smarts: accelerating microarchitecture simulation via rigorous statistical sampling," *Proceedings of ISCA.*, 2003.

[7] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sniper: exploring the level of abstraction for scalable and accurate parallel multi-core simulation," in *Proceedings of 2011 SC*, p. 52, ACM, 2011.

[8] D. Sanchez and C. Kozyrakis, "ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-Core Systems," *Proceedings of ISCA*, 2013.

[9] M. Véran and D. Potier, *QNAP 2: A portable environment for queueing systems modelling*. PhD thesis, INRIA, 1984.

[10] R. Zilan, J. Verdú, and J. García, "An abstraction methodology for the evaluation of multi-core multi-threaded architectures," *IEEE 19th MASCOTS*, 2011.

[11] T. F. Tsuei and W. Yamamoto, "Queuing simulation model for multiprocessor systems," *Computer*, 2003.

[12] D. Meisner and T. Wenisch, "Stochastic queuing simulation for data center workloads," *Exascale Evaluation and Research Techniques Workshop*, 2010.

[13] E. Gelenbe and I. Mitrani, *Analysis and synthesis of computer systems*, vol. 4. World Scientific, 2010.

[14] S. Berkowits, "Pin-a dynamic binary instrumentation tool," 2012.

[15] A. Varga, "The OMNeT++ discrete event simulation system," *ESM)*, 2001.

[16] *iQ, code repository*. Available at https://github.com/damianroca/iQ---Queue-model-simulation,.

[17] *H2020 project, dRedBox*. Available at http://www.dredbox.eu/home.html,.

[18] I. Corporation, "Intel 64 and IA-32 Architectures Optimization Reference Manual," no. March, 2014.

[19] A. Fog, "The microarchitecture of intel, amd and via cpus/an optimization guide for assembly programmers and compiler makers," 2012.

[20] J. L. Henning, "Spec cpu2006 benchmark descriptions," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 4, pp. 1–17, 2006.

[21] K. Hoste and L. Eeckhout, "Microarchitecture independent workload characterization," in *IEEE Micro*, 2007.

[22] *Phase Behavior ZSim analysis*. Available at http://zsim.csail.mit.edu/validation/time_series/,.

[23] A. Gutierrez, R. G. Dreslinski, T. Mudge, C. Sudanthi, C. D. Emmons, M. Hayenga, and N. Paver, "Sources of error in full-system simulation," *Proceedings of ISPASS*, 2014.

[24] R. Intel, "Intel 64 and ia-32 architectures optimization reference manual," *Intel Corporation, May*, 2012.

[25] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The splash-2 programs: Characterization and methodological considerations," in *ACM SIGARCH computer architecture news*, 1995.

[26] S. Van den Steen, S. De Pestel, M. Mechri, S. Eyerman, T. Carlson, D. Black-Schaffer, E. Hagersten, and L. Eeckhout, "Micro-architecture independent analytical processor performance and power modeling," in *Proceedings of ISPASS*, IEEE, 2015.

[27] A. Patel, F. Afram, S. Chen, and K. Ghose, "Marss: Micro architectural systems simulator.," in *ISCA tutorial 6*, 2012.

[28] *SPEC benchmarks performance*. Available at http://zsim.csail.mit.edu/validation/ooo.txt,.

[29] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe, "Smarts: Accelerating microarchitecture simulation via rigorous statistical sampling," in *Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on*, IEEE, 2003.

[30] T. Grass, A. Rico, M. Casas, M. Moreto, and E. Ayguadé, "Taskpoint: Sampled simulation of task-based programs," in *Performance Analysis of Systems and Software (ISPASS), 2016 IEEE International Symposium on*, pp. 296–306, IEEE, 2016.

[31] J. Cook, J. Cook, and W. Alkohlani, "A statistical performance model of the opteron processor," *ACM SIGMETRICS Performance Evaluation Review*, 2011.

[32] B. C. Lee and D. M. Brooks, "Accurate and efficient regression modeling for microarchitectural performance and power prediction," in *ACM SIGOPS Operating Systems Review*, ACM, 2006.

[33] T. Tsuei and W. Yamamoto, "A processor queuing simulation model for multiprocessor system performance analysis," in *Proc. of 5th Workshop on Computer Architecture Evaluation using Commercial Workloads (CAECW)*, pp. 58–64, 2002.

[34] D. Meisner, J. Wu, and T. F. Wenisch, "Bighouse: A simulation infrastructure for data center systems," *ISPASS*, 2012.