DOI 10.7603/s40601-014-0021-9

High-level simulation of concurrency operations in microthreaded many-core architectures

Irfan Uddin

Received 16 Jun 2015 Accepted 01 Jul 2015

***Abstract*—Computer architects are always interested in analyzing the complex interactions amongst the dynamically allocated resources. Generally a detailed simulator with a cycle-accurate simulation of the execution time is used. However, the cycle-accurate simulator can execute at the rate of 100K instructions per second, divided over the number of simulated cores. This means that the evaluation of a complex application with complex concurrency interactions on contemporary multi-core machine can be very slow. To perform efficient design space exploration we present a co-simulation environment, where the detailed execution of concurrency instructions in the pipeline of microthreaded cores and the interactions amongst the hardware components are abstracted. We present the evaluation of the high-level simulation framework against the cycle-accurate simulation framework. The results show that high-level simulator is faster and less complicated than cycle-accurate simulator and has reasonable accuracy.**

Keywords*—*High-level simulation, many-core systems, concurrent systems.

I. Introduction

The detailed simulation of a complex MultiProcessor System-on-Chip (MPSoC) in software increases the wall-clock execution time of the simulator. This cost of wall-clock time in the cycle-accurate simulation of many-core architectures is becoming increasingly expensive in the future many-core systems. As the modern interpretation of Moore's law implies that: *The number of cores will double every next generation*. The number of cores in a system are scaling up to hundreds or thousands, we believe that the detailed simulation of the components in the core become less interesting. Instead, the focus is shifting towards the overall behavior of the chip, where the simulation of creation, communication and synchronization of concurrency are more important.

In this article, we explore specifically the microthreaded many-core architecture also known as the Microgrid [15], [17], [14], [19] which implements the microthreading model [16] in hardware. This requires special attention, because existing simulation techniques do not yet account for the combined use of multiple cores, data-flow scheduling and hardware multi-threading. Every core in the Microgrid supports fine-grained multi-threading and implements the concurrency management of the model in its instruction set (ISA). There exists a cycle-accurate simulator for the Microgrid, named as MGSim [20], [26], [19], [25], [20]. It simulates all the low-level details of the architecture and therefore becomes very slow in the execution of large applications. The detailed simulation of the architecture is not suitable for design space exploration [29] and therefore a high-level simulator is desirable as a complementing tool. The techniques presented in this paper for scheduling, resource management and abstraction may be used to simulate other many-core architectures e.g. Intel SCC, Sun Sparc Tx, Tile64 etc. The high-level simulator of the Microgrid is named as HLSim [34], [28], [33], [32], [31] and is developed to make quick and reasonably accurate design decisions in the evaluation of the architecture using multiple runs of benchmarks which can consist of billion of instructions execution.

The objective of HLSim is to evaluate applications and make system-level decisions to the architecture and if necessary these decisions can be validated using MGSim. The applications used for the evaluation expose dynamic behavior as they adapt to the resources available at run-time, in order to improve efficiency in the architecture or application. HLSim provides help to the designers of the Microgrid, in investigating the implementation of operating system services to support the dynamic adaptation, in particular dynamic resource allocations and mappings.

The rest of the paper is organized as follows. We will give a background to the Microgrid in section II. We present the high-level simulation technique for the Microgrid in section III. The high-level simulation of concurrency constructs is given in in section IV and the evaluation of the framework against MGSim is given in section V. The related work is presented in section VI and the paper is concluded in section VII.

II. Background

The Microgrid [15], [3], [13], [30] is a general-purpose, many-core architecture that implements hardware multi-threading using data-flow scheduling and a concurrency management protocol in hardware to create and synchronize threads within and across the cores on chip. The suggested concurrent programming model for this chip is based on fork-join constructs, where each created thread can define further concurrency tree hierarchically. This model is called the microthreading model and is also applicable to current multi-core architectures using a library of the concurrency constructs called *svp-ptl* [35] built on top of pthreads. Our work focus on the microthreaded architecture where each core contains a single issue, in-order RISC pipeline with an ISA similar to DEC/Alpha, and all cores are connected to an on-chip distributed memory network [14], [4]. Each core implements the concurrency constructs in its ISA and is

Published online: 13 December 2015

able to support hundreds of threads and their contexts, called microthreads and tens of families (i.e. ordered collections of indexed microthreads) simultaneously. The channels for the communication and synchronization of the family introduced in the microthreading model are implemented in registers of the Microgrid. The registers allocated to a thread are categorized as; globals, locals, shareds and dependents.

To program this architecture, we use a system-level language called SL [18] which integrates the concurrency constructs of the microthreading model as language primitives. It works as an intermediate language for high-level programming languages (such as SAC [12]), a compiler [11] or with little effort by the programmer.

III. High-level simulation

The high-level simulator for the Microgrid is implemented in C++ using POSIX threads. We have introduced separation of concerns between application and architecture models, meaning that we have separate application and architecture models and either can separately be modified in order to improve performance. The application model simulates microthreads in the microthreading model [16] on the host machine and the architecture model simulates resources of the Microgrid. HLSim is an implementation of a discrete-event simulation [22]. The threads in the application model generates events to represent the concurrency or the computation in the code. These events are mapped to the architecture model which simulates the cycles required by the events on the architecture and advances the simulated time. The cycles represent the load on the multi-processor based on per instruction timing weight. The framework of HLSim consists of three parts.

- Application model: It is the microthreaded program that executes on the host machine, but generates events with the estimates for concurrency and block of computational constructs. It provides fully functional execution of the program and there are parameters that can be changed for performance based on simulation results e.g. place size, window size etc.
- Mapping function: The events generated by the application model are stored in an ordered queue where the top of the queue represent the event with the least workload. This layer is responsible to pass timing information to the architecture model and can be changed implicitly with changing in the application model.
- Architecture model: This layer simulates the hardware of microthreaded many-core architecture. It provides location of execution, simulated time, distribution of threads on cores etc. to the application model. We are using different models for the architecture model; One-IPC HLSim, Signature-based HLSim, Cache-based HLSim and Analytical-based HLSim.

The events for the concurrency constructs are blocking and computational constructs are non-blocking. By blocking we mean that the application model stops execution until notified by the architecture model, and by non-blocking we mean that the application model keeps sending events without any acknowledgement until the next blocking event is generated and blocked by the architecture model.

In order to evaluate the performance of an instruction stream, we do not implement the instruction issue mechanism (as in MGSim), rather we estimate the number of cycles required for the execution of the instruction stream and simulate these cycles in the high-level simulator to demonstrate the behavior of threads executing in the Microgrid [34], [28], [33]. We estimate the performance using basic blocks in every thread in the application model. In order to extract concurrency constructs from the instruction stream, a basic block is logically divided into two parts:

- Computational constructs: The part of the basic block which consists of instructions that perform computation. These instructions are executed natively on the host machine and the estimated number of instructions in the basic block is sent in the form of events to the architecture model. The estimated number of instructions represent the number of cycles required for their computation. The technique of estimating the workload of instruction is given in [34], [33].
- Concurrency constructs: The part of the basic block which does not perform any computation but takes care of concurrency management known as concurrency constructs (i.e. *allocate*, *create*, *sync*, *read from shared*, *write to shared* etc.). The time taken by concurrency constructs are calculated based on the dynamic state of the system. For example a sync event will take time based on the number of cores where the family is distributed. The time is estimated and then mapped to the architecture model of HLSim which advances the simulated time.

IV. High-level simulation of concurrency constructs

The concurrency constructs in microthreaded many-core systems are explained in detail in [30]. In this section we describe the latency of concurrency constructs simulated in HLSim.

A. Allocation

The *allocate* message is sent from the parent core to the group of cores where the family is delegated. The message travels to all cores in the group to see if it can allocate at least one thread table entry, one family table entry and 31 registers in each core. When all the cores succeed the message goes backward from the last core to the first core in the group and these entries are allocated asynchronously. The allocate process for a family of four cores is shown in fig. 1.

The latency of the allocate message from the parent core to the first core is 14 cycles. The allocation to any additional core will increase the latency by 5 more cycles given in Eq: 1, where nc is the number of cores in the given place.

$$latency = 14 + 5 \times nc \qquad (1)$$

In case of *balanced* strategy (c.f. [30]), the allocate process has a latency of 2 cycles compared to 5 cycles, as the message

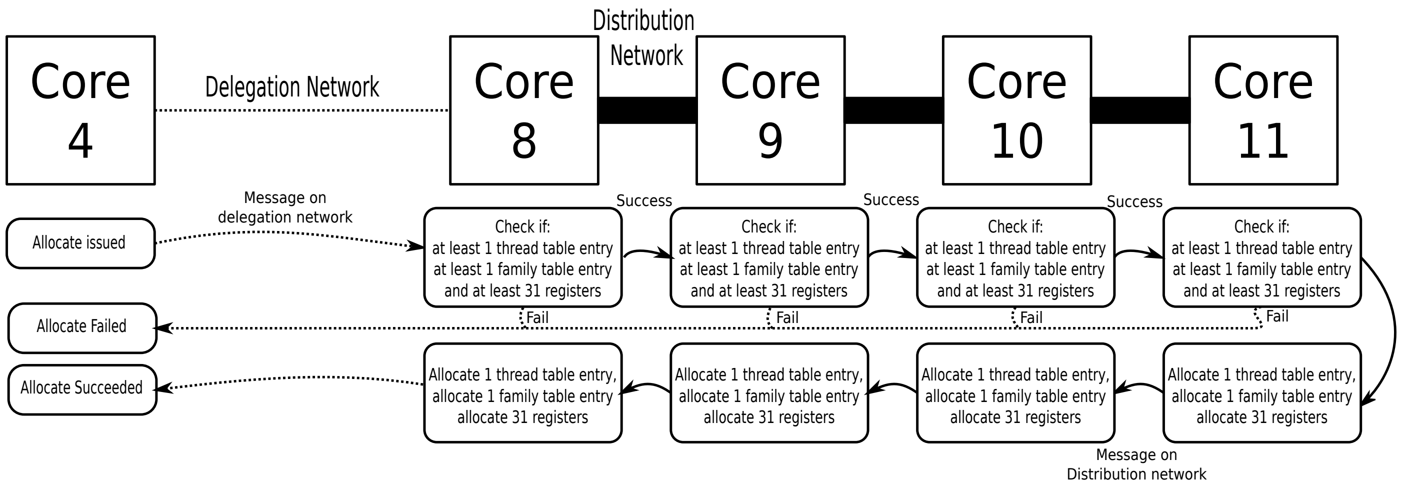
Core 4
Delegation Network
Core 8
Distribution Network
Core 9
Core 10
Core 11
Message on delegation network
Allocate issued
Check if: at least 1 thread table entry at least 1 family table entry and at least 31 registers
Success
Fail
Allocate Failed
Allocate Succeeded
Allocate 1 thread table entry, allocate 1 family table entry allocate 31 registers
Message on Distribution network

Fig. 1: The allocation process of a family on a group of four cores.

travels one way; from the parent core to the first core and then keeps on going until the last core. On the return the message goes directly to the selected core via delegation network, which is the least loaded and takes 3 more cycles. The allocation process takes one cycle on the parent core, and the rest of the latency can actually be tolerated given that there are enough threads in the pipeline.

B. Configuration

After the family is allocated, some parameters need to be passed to the first core for the configuration of the family. These parameters are *start*, *limit*, *step* and *window size*. Every parameter sends a separate message, but they are optional and in many cases there will be less than 4 messages. The latency of the configuration process depends on the parameterised creation. There are three possible cases:

- In the create construct, if no parameter or a constant or variable is used the default value is assumed. Then no message is passed for configuration and therefore no cycle is consumed.
- When some constants or variables are used, which are different than the default values. In that case 1 to 4 instructions create messages, where every message has the latency of one cycle.
- When some variables are used where the values are not known at compile time and not the same as default. Then for every parameter one message is passed, where the latency of every message is one cycle.

The configuration process completes asynchronously i.e. as soon as the configure messages are sent, the create process can start on the parent core. In the high-level simulation we do not consider the individual messages as per every parameter in the configuration. We group the 1-4 messages into a single high-level message which takes 4 cycles.

C. Creation

After the configuration messages have been sent on the delegation network, the creation process starts on the parent core. The parent core sends the create message to the first core in the group of cores where the family is delegated. As soon as the threads are created on the first core the parent core is notified, where the parent core can then send the message to write the shared and global channels. The latency of create message is independent of the number of cores allocated, mode or strategy and it is always 20 cycles. However it takes only one cycle on the parent core and the rest of the cycles can be tolerated.

D. Synchronization

The parent thread completes the creation process and continues with other instructions (if any). After that the synchronization message is issued, but depending on whether the threads of the family are still executing, the parent thread can be suspended. When all the threads of the family are terminated the synchronization is activated to synchronize the family.

Actually the time the sync message is issued by the parent thread, the sync message travels from the parent core to the first core. Every core knows the number of threads allocated from a family. As soon as all the threads allocated to the core terminate, the core passes this information to the next core. At the last core when all threads are terminated, the message is acknowledged to the parent core that the family can be synchronized. This process of synchronization is shown in fig. 2.

The synchronization process takes 14 cycles when the family is allocated to only one core. Any additional core takes two more cycles. This is given in Eq: 2, Where uc is the number of cores used by the family. The synchronization takes 1 cycle on the parent core, and the rest of the cycles can actually be tolerated given a sufficient amount of threads in the pipeline.

$$latency = 14 + 2 \times uc \tag{2}$$

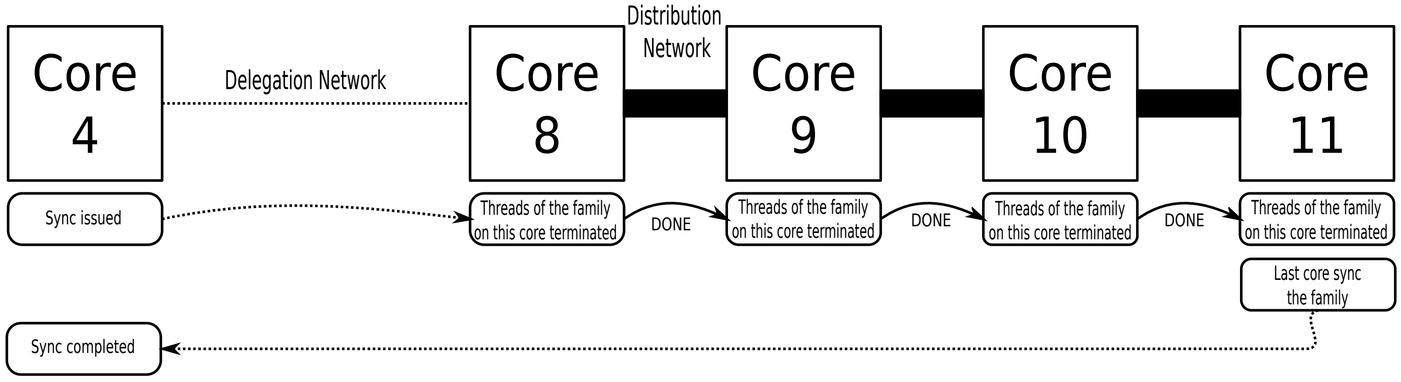
Core 4
Delegation Network
Core 8
Distribution Network
Core 9
Core 10
Core 11
Sync issued
Threads of the family on this core terminated
DONE
Threads of the family on this core terminated
DONE
Threads of the family on this core terminated
DONE
Threads of the family on this core terminated
Last core sync the family
Sync completed

Fig. 2: The synchronization of a family created on a group of four cores.

E. Release

Once the family is synchronized, the memory modified by the created thread is defined in the parent core and their context are released. A message can then be sent from the parent core to the first core on delegation network to release the resources. The message then travels from the first core to the last core on the distribution network. This process is completed asynchronously and therefore we do not simulate this timing in HLSim.

V. RESULTS

In this section we present results to evaluate One-IPC HLSim against MGSim.

A. Ratio in simulated time

We show the ratio of simulated time in One-IPC HLSim over MGSim in fig. 3. A ratio closer to 1 means that the simulated time in MGSim and One-IPC HLSim are converging. In the given graph, the ratio does not stay close to 1 and illustrates the inaccuracy in One-IPC HLSim. This inaccuracy is mainly because of the assumption in simulation that *every instruction takes one cycle to complete all the time*, while in MGSim the assumption can be true only if there are sufficient threads in the pipeline.

B. The effect of window size on simulated time

The simulated time of both simulators based on the *window size* is shown in fig. 4. The One-IPC HLSim always takes one cycle per instruction, and therefore the increased number of threads per family does not have any effect on the performance, resulting in a straight line for all window sizes. MGSim, however, shows an increase in the performance when the size of the window increases because of latency tolerance. However, the performance line in MGSim does not even touch the line of One-IPC HLSim. It means that MGSim never gets to the point where the latency of instructions is one cycle, because it has the overhead of concurrency and long latency operations. After a window size of 16, we do not see any change in cycles because the number of threads that can be created is reached. There are 256 threads in total (i.e. $2^8 = 256$) and concurrency is 128 threads (i.e. $256/2$), therefore the maximum number of threads per core is 16 (i.e. $128/8 = 16$).

C. Simulation time

In this experiment, we execute an approximation of the Mandelbrot set using different complex plane sizes and different number of cores. We show two experiments of simulation time; in the first experiment we execute a complex plane of different sizes using selected number of cores, and in the second experiment we execute a particular complex plane on different number of cores. In the first experiment the X-axis shows the size of the complex plane and Y-axis shows the simulation time in the range of program execution. In the second experiment the X-axis shows the number of simulated cores and the Y-axis shows the simulated time in the range of program execution. There is no particular reason to use Mandelbrot instead of FFT for simulation time, but to give a different application for evaluation.

The simulation of large number of cores in MGSim increases the complexity of the simulator and therefore has an impact on the simulation time, because of the interaction between simulated cores. But in HLSim, the complexity does not change with increasing the number of simulated cores, and therefore the simulation time remains close to a straight line shown in fig. 5. This is one of the key benefits of HLSim, that it does not effect the simulation time with the increased number of simulated cores. In design space exploration, we can use HLSim with a large number of cores on the chip to execute large applications with little impact on simulation time.

D. IPC - Simulation accuracy

Instructions Per Cycle (IPC) shows the efficiency (not performance, as that also depends on the clock frequency) of the architecture. For each core the IPC should be as close to the number of instructions the architecture is capable of issuing in each cycle. In case of the Microgrid, with single issue, the IPC of each core should be as close to 1 as possible. However, for c cores, the overall IPC may be up to c, i.e. each core may issue 1 instruction per cycle. We can measure the average IPC, i.e. sum of the IPC of c cores divide by c. Because of the pressure on memory and load balancing MGSim can perform with different efficiency level in different applications. The IPC in HLSim depends on the abstraction over the detailed instruction execution in MGSim. In order to see a direct comparison we can compute the percentage error of One-IPC HLSim to MGSim and this is shown in table I.

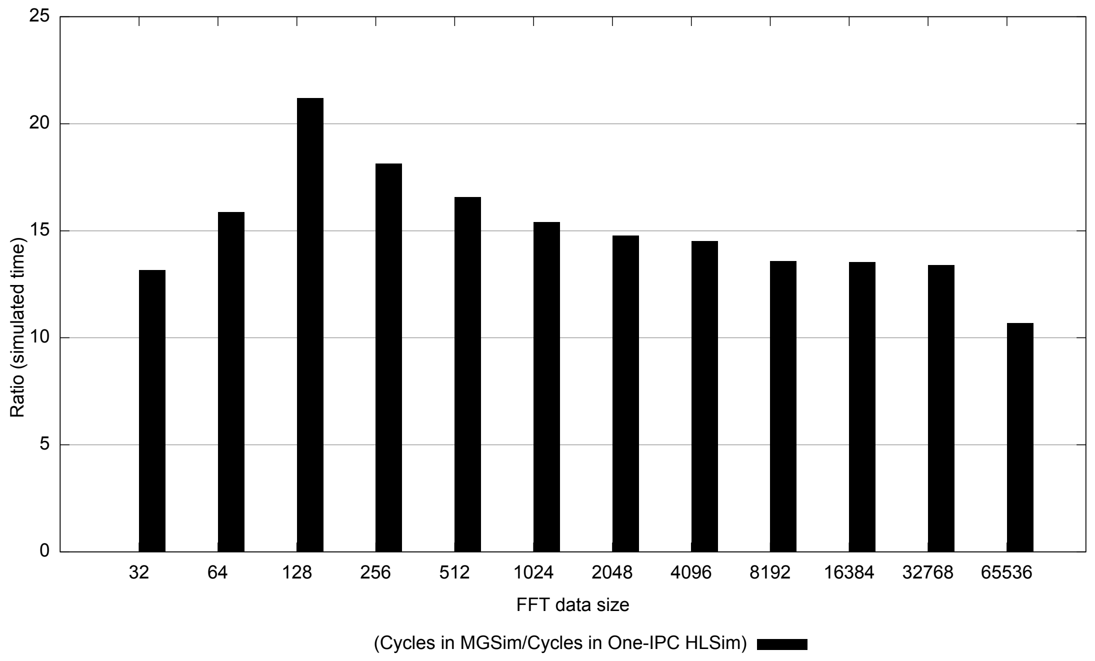

Fig. 3: Ratio in simulated time of FFT using different data sizes and executing on 64 simulated cores.

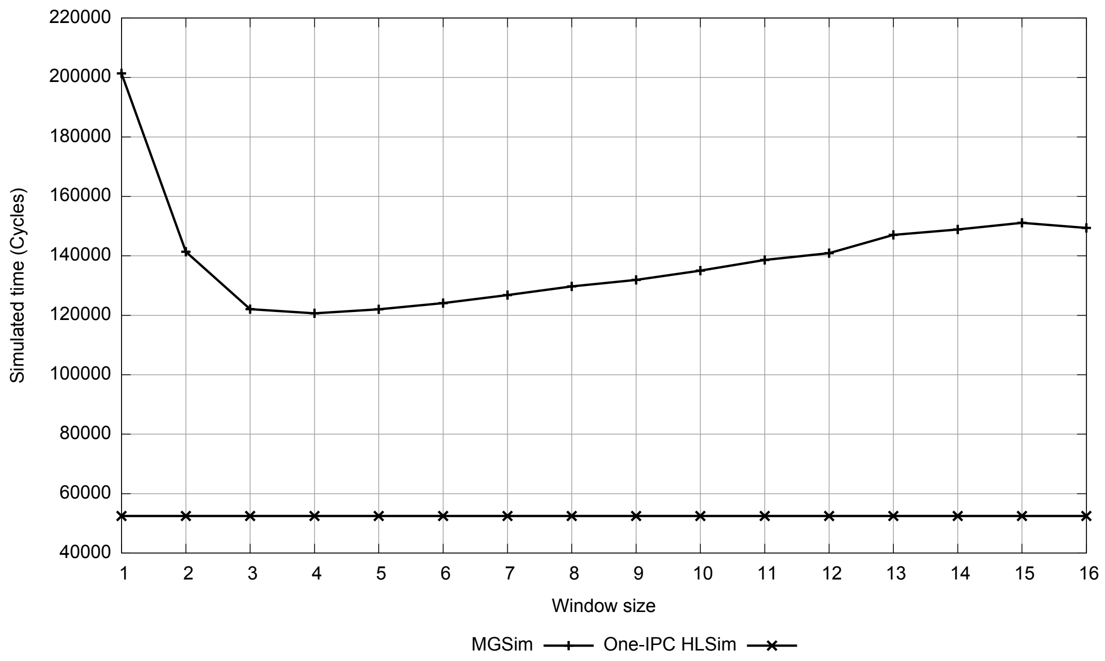

Fig. 4: The effect of changing the window size on the execution of FFT of size 2^8 2^3 cores.

Application	$\frac{One-IPC\,HLSim\,-\,MGSim}{MGSim} * 100$
GOL (Torus)	6.4%
GOL (Grids)	30.84%
FFT	58.39%
LMK7	42.66%
Mandelbrot	0.90%
MatrixMultiply	14.15%
Smooth	32.11%

TABLE I: Percent error in the average IPC of different cores in MGSim and One-IPC HLSim.

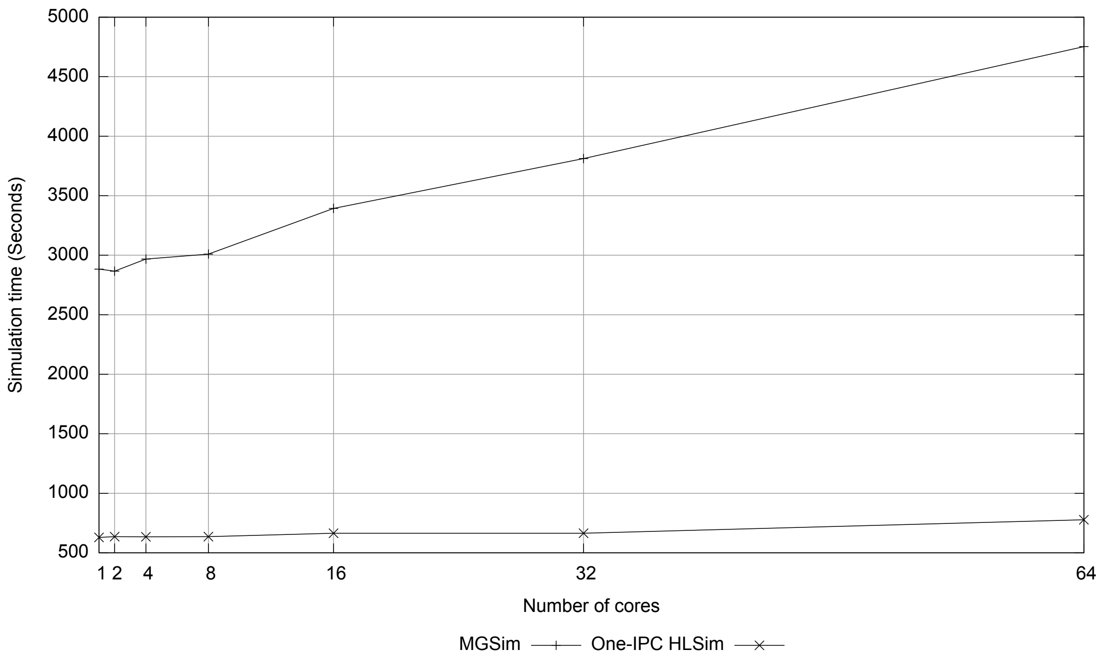

Fig. 5: Simulation time in the execution of Mandelbrot set (Complex plane: 1000×1000) on different number of cores of One-IPC HLSim and MGSim.

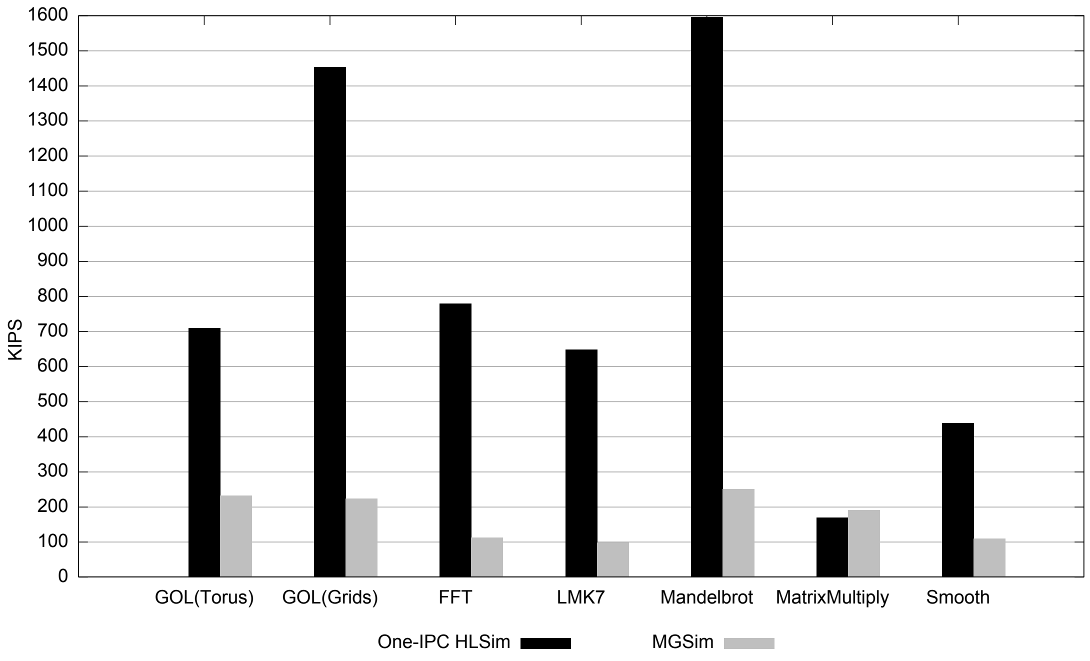

Fig. 6: The average IPS achieved by One-IPC HLSim and MGSim.

E. IPS - simulation speed

Instructions per second (IPS) is used to measure the basic performance of an architecture i.e. we measure the simulated instructions per second using a known contemporary processor. The average IPS (average across all the cores) achieved by One-IPC HLSim and MGSim is shown in fig. 6. We can see that the IPS of MGSim is approximately 100 KIPS, and the IPS of One-IPC HLSim is approximately 1MIPS. Different simulators used in industry and academia with their simulation speed in terms of IPS are; COTSon [1] executes at 750KIPS, SimpleScalar [2] executes at 150KIPS, Interval simulator [5] executes at 350KIPS and Sesame [8] executes at 300KIPS. and MGSim [4] executes at 100KIPS. Compared to the IPS of these simulators the IPS of HLSim is very promising. It should be noted that the referenced simulation frameworks given above simulate only small (2-4 cores) number of cores on the chip. In MGSim and HLSim we have simulated 128 cores on a single chip, given this large number of simulated cores on a chip, 1MIPS indicates a high simulation speed.

VI. Related work

High-level simulator avoids unnecessary details of execution and communication in the architectures at the expense of losing accuracy. High-level simulators are commonly used in the domain of embedded systems e.g. [21], [24], [27]. However, these type of simulators are rarely used in general-purpose multi- or many-core systems. Eeckhout et al. have worked on statistical simulation of single-core general-purpose processors [7], [6], [9] and multi-core processors [10]. Similarly, Nussbaum and Smith [23] has also worked on statistical approaches to provide high-level simulation of multi-core processors.

RAMP [36] is an open-source, community-developed, FPGA-based emulator of parallel architectures. It is not just a hardware architecture project, but the most important goal is to support the software community to take advantage of the potential capabilities of parallel microprocessors, by providing a platform through which the software community can collaborate with the hardware community. It provides multiple levels for evaluation, where one of the level provides the implementation of distributed memory network and hence a framework to experiment with the memory architecture.

In One-IPC HLSim we have focused on the abstraction of concurrency operations in the microthreaded many-core architecture. The detailed execution of concurrency operations require that all instructions are executed in the pipeline, which can be very slow, given the number of concurrency operations in a complicated application. The abstraction of these constructs reduces the execution of complexed applications in HLSim which is required in the early design space exploration of many-core architectures.

VII. Conclusion

We have presented a high-level simulation framework which integrates the architecture model to the application model in a concurrent system. The architecture model abstracts the instruction stream from the detailed execution of the instructions in the pipeline and the details of communication in the concurrency of the microthreading model. These abstractions improve the simulation speed of the high-level simulator. The architecture model is based on the assumption that every instruction takes one cycle to complete and can be true only when there are sufficient threads to tolerate latency. In the case of inefficient threads the latency of long latency operation can not be tolerated and hence One-IPC HLSim can not be considered as an accurate simulation model. The future research will address the areas to provide the simulation of fine-grained latency tolerance to improve the accuracy of HLSim.

References

[1] Eduardo Argollo, Ayose Falcón, Paolo Faraboschi, Matteo Monchiero, and Daniel Ortega. Cotson: infrastructure for full system simulation. *SIGOPS Oper. Syst. Rev.*, 43(1):52–61, 2009.

[2] Todd Austin, Eric Larson, and Dan Ernst. SimpleScalar: An Infrastructure for Computer System Modeling. *Computer*, 35(2):59–67, 2002.

[3] Thomas A. M. Bernard, Clemens Grelck, Michael A. Hicks, Chris R. Jesshope, and Raphael Poss. Resource-agnostic programming for many-core microgrids. In *Proceedings of the 2010 conference on Parallel processing*, Euro-Par 2010, pages 109–116, Berlin, Heidelberg, 2011. Springer-Verlag.

[4] K. Bousias, L. Guang, C. R. Jesshope, and M. Lankamp. Implementation and evaluation of a microthread architecture. *J. Syst. Archit.*, 55:149–161, March 2009.

[5] Trevor E. Carlson, Wim Heirman, and Lieven Eeckhout. Sniper: exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 52:1–52:12, New York, NY, USA, 2011. ACM.

[6] Lieven Eeckhout, Robert H. Bell Jr., Bastiaan Stougie, Koen De Bosschere, and Lizy K. John. Control flow modeling in statistical simulation for accurate and efficient processor design studies. *SIGARCH Comput. Archit. News*, 32(2):350, 2004.

[7] Lieven Eeckhout, Sebastien Nussbaum, James E. Smith, and Koen De Bosschere. Statistical simulation: Adding efficiency to the computer designer's toolbox. *IEEE Micro*, 23:26–38, September 2003.

[8] Cagkan Erbas, Andy D. Pimentel, Mark Thompson, and Simon Polstra. A framework for system-level modeling and simulation of embedded systems architectures. *EURASIP J. Embedded Syst.*, 2007:2–2, January 2007.

[9] Davy Genbrugge and Lieven Eeckhout. Memory data flow modeling in statistical simulation for the efficient exploration of microprocessor design spaces. *IEEE Trans. Comput.*, 57(1):41–54, 2008.

[10] Davy Genbrugge and Lieven Eeckhout. Chip multiprocessor design space exploration through statistical simulation. *IEEE Transactions on Computers*, 58:1668–1681, 2009.

[11] Clemens Grelck, Stephan Herhut, Chris Jesshope, Carl Joslin, Mike Lankamp, Sven-Bodo Scholz, and Alex Shafarenko. Compiling the functional data-parallel language sac for microgrids of self-adaptive virtual processors. In *14th Workshop on Compilers for Parallel Computing (CPC'09), IBM Research Center, Zurich, Switzerland*, 2009.

[12] Clemens Grelck and Sven-Bodo Scholz. Sac: off-the-shelf support for data-parallelism on multicores. In *Proceedings of the 2007 workshop on Declarative aspects of multicore programming*, DAMP '07, pages 25–33, New York, NY, USA, 2007. ACM.

[13] Chris Jesshope. A model for the design and programming of multi-cores. *Advances in Parallel Computing*, High Performance Computing and Grids in Action(16):37–55, 2008.

[14] Chris Jesshope, Mike Lankamp, and Li Zhang. The implementation of an svp many-core processor and the evaluation of its memory architecture. *SIGARCH Comput. Archit. News*, 37:38–45, July 2009.

[15] Chris R. Jesshope. Microgrids - the exploitation of massive on-chip concurrency. In Lucio Grandinetti, editor, *High Performance Computing Workshop*, volume 14 of *Advances in Parallel Computing*, pages 203–223. Elsevier, 2004.

[16] Chris R. Jesshope. Microthreading a model for distributed instruction-level concurrency. *Parallel Processing Letters*, 16(2):209–228, 2006.

[17] Raphael ‘kena’ Poss. *On the realizability of hardware microthreading—Revisting the general-purpose processor interface: consequences and challenges*. PhD thesis, University of Amsterdam, 2012.

[18] Raphael ‘kena’ Poss. SL—a “quick and dirty” but working intermediate language for SVP systems. Technical Report arXiv:1208.4572v1 [cs.PL], University of Amsterdam, August 2012.

[19] Raphael ‘kena’ Poss, Mike Lankamp, Qiang Yang, Jian Fu, Michiel W. van Tol, Irfan Uddin, and Chris R. Jesshope. Apple-CORE: harnessing general-purpose many-cores with hardware concurrency management. *Microprocessors and Microsystems*, 2013.

[20] Mike Lankamp, Raphael Poss, Qiang Yang, Jian Fu, Irfan Uddin, and Chris R. Jesshope. MGSim - Simulation tools for multi-core processor architectures. Technical Report arXiv:1302.1390v1 [cs.AR], University of Amsterdam, February 2013.

[21] Brett H. Meyer, Joshua J. Pieper, JoAnn M. Paul, Jeffrey E. Nelson, Sean M. Pieper, and Anthony G. Rowe. Power-performance simulation and design strategies for single-chip heterogeneous multiprocessors. *IEEE Trans. Comput.*, 54(6):684–697, 2005.

[22] Jayadev Misra. Distributed discrete-event simulation. *ACM Comput. Surv.*, 18:39–65, March 1986.

[23] Sebastien Nussbaum and James E. Smith. Statistical simulation of symmetric multiprocessor systems. In *SS ’02: Proceedings of the 35th Annual Simulation Symposium*, page 89, Washington, DC, USA, 2002. IEEE Computer Society.

[24] Andy D. Pimentel, Cagkan Erbas, and Simon Polstra. A systematic approach to exploring embedded system architectures at multiple abstraction levels. *IEEE Trans. Comput.*, 55:99–112, February 2006.

[25] Raphael Poss, Mike Lankamp, Irfan Uddin, Jaroslav Sýkora, and Leoš Kafka. Heterogeneous integration to simplify many-core architecture simulations. In *Proceedings of the 2012 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools*, RAPIDO ’12, pages 17–24, New York, NY, USA, 2012. ACM.

[26] Raphael Poss, Mike Lankamp, Qiang Yang, Jian Fu, Irfan Uddin, and Chris Jesshope. MGSim - A simulation environment for multi-core research education. *SAMOS*, 2013.

[27] Mendel Rosenblum, Edouard Bugnion, Scott Devine, and Stephen A. Herrod. Using the SimOS machine simulator to study complex computer systems. *ACM Trans. Model. Comput. Simul.*, 7(1):78–103, 1997.

[28] Irfan Uddin. High-level simulation of the Microgrid. Master’s thesis, University of Amsterdam, Amsterdam, the Netherlands, August 2009.

[29] Irfan Uddin. Design space exploration in the microthreaded many-core architecture. Technical report, University of Amsterdam, September 2013. arXiv Technical report.

[30] Irfan Uddin. Microgrid - The microthreaded many-core architecture. Technical report, University of Amsterdam, September 2013. arXiv Technical report.

[31] Irfan Uddin, Raphael Poss, and Chris Jesshope. Analytical-based high-level simulation of microthreaded many-core architectures. In *PDP*, February 2014.

[32] Irfan Uddin, Raphael Poss, and Chris Jesshope. Cache-based high-level simulation of microthreaded many-core architectures. *Journal of Systems Architecture*, 60(7):529 – 552, 2014.

[33] Irfan Uddin, Raphael Poss, and Chris Jesshope. Signature-based high-level simulation of microthreaded many-core architectures. *SIMULECH*, 2014.

[34] Irfan Uddin, Michiel W. van Tol, and Chris R. Jesshope. High-level simulation of SVP many-core systems. *Parallel Processing Letters*, 21(4):413–438, December 2011.

[35] M.W. van Tol, C.R. Jesshope, M. Lankamp, and S. Polstra. An implementation of the sane virtual processor using posix threads. *Journal of Systems Architecture*, 55(3):162–169, 2009. Challenges in self-adaptive computing (Selected papers from the Aether-Morpheus 2007 workshop).

[36] J. Wawrzynek, D. Patterson, M. Oskin, Shin-Lien Lu, C. Kozyrakis, J.C. Hoe, D. Chiou, and K. Asanovic. Ramp: Research accelerator for multiple processors. *Micro, IEEE*, 27(2):46 –57, march-april 2007.

AUTHOR’S PROFILE

Irfan Uddin is working as Assistant Professor at Al-Yamamah University, Riyadh, Kingdom of Saudi Arabia. His expertise include architectural simulation, high-level simulations, design space exploration and comparative analysis. He has been working on the high-level simulation for a particular type of many-core systems, known as microthreaded many-core architecture.

Published online: 13 December 2015