Thread- and Data-Level Parallel Simulation in SystemC, a Bitcoin Miner Case Study

Zhongqi Cheng, Tim Schmidt, Guantao Liu, Rainer Dömer
Center for Embedded and Cyber-Physical Systems
University of California, Irvine, USA

***Abstract*—The rapidly growing design complexity has become a big obstacle and dramatically increased the time required for SystemC simulation. In this case study, we exploit different levels of parallelism, including thread- and data-level parallelism, to accelerate the simulation of a Bitcoin miner model in SystemC. Our experiments are performed on two multi-core processors and one many-core Intel® Xeon Phi™ Coprocessor. Our results show that with the combination of data- and thread-level parallelism, the peak simulation speed improves by over 11x on a 4-core host, 50x on a 16-core host, and 510x on a 60-core host, respectively. The results confirm the efficiency of combining thread- and data-level parallelism for higher SystemC simulation speed, and can serve as a benchmark for future optimization of system level design, modeling, and simulation.**

I. INTRODUCTION

SystemC [1] is a widely used modeling language for Electronic System Level (ESL) design and also provides a simulation framework for validation and verification [2]. With the rapidly growing complexity of embedded systems, a tremendous challenge is imposed on the simulation time, which is a crucial factor affecting the time-to-market and thus the commercial success. Various studies have been proposed to accelerate the SystemC simulator, and parallelization is often the most common approach.

With the development of computer architecture, the parallelism mainly takes three forms [3], namely instruction-level parallelism (ILP), data-level parallelism (DLP) and thread-level parallelism (TLP). ILP is implicit. It is exploited automatically by the compiler and processor, without the interaction or awareness of the software developer. In contrast, DLP and TLP are explicit. The programmers are required to write parallel code and pragmas manually. In the SystemC based ESL design, TLP is achieved by the simulator, specifically, the parallel discrete event simulator. It can issue and run multiple simulation threads in parallel. On the other hand, exploiting DLP for faster SystemC simulation is a novel idea. It is first proposed in 2017 [4]. In this case study, we evaluate the effectiveness of this technique with a more computationally intensive SystemC design, the Bitcoin miner.

We exploit different level of parallelism, including TLP, DLP and the combination of both to accelerate the simulation of a Bitcoin miner model in SystemC. ILP is not considered because it is transparent to the programmer and automatically applied at the hardware level. Bitcoin miner is used as a case study due to its high potential for parallel execution.

The contributions of this case study are as follows:

- We have developed a SystemC model of the reference C++ Bitcoin miner project, which can serve as a suitable test bench for evaluating parallel SystemC models.
- We evaluate the performance of thread-level parallelism in the context of the RISC simulator [14]. Our results show that the speedup of the parallel simulation compared to the reference sequential simulator is proportional to the number of simulation threads. This confirms that RISC is an effective framework for parallel SystemC simulation.
- We exploit data-level parallelism on top of thread-level parallelism for fast SystemC simulation. SIMD pragmas are added to vectorize loops and functions. The results show that with the combination of DLP and TLP, a speedup of the magnitude of $N \times M$ is achieved, where N and M denote the thread- and data-level speedup factors, respectively. This confirms the results and conclusions of [4].
- We analyze the scalability of the parallel SystemC simulation on a many-core Xeon Phi™ Coprocessor. A speedup of over 510x is attained. This demonstrates that simulation of Bitcoin miner using the RISC simulator scales well on the Xeon Phi™ Coprocessor.

The rest of the paper [5] is organized as follows: Section II introduces Bitcoin miner and its modeling in SystemC. In Section III and IV, we take advantage of thread-level and data-level parallelism to accelerate the simulation. Results and evaluations are carried out in Section V. Finally, Section VI concludes this case study.

A. Related work

Various approaches have been proposed to speedup the simulation of SystemC models. Parallel discrete event simulation (PDES) is well studied in [6] and [7]. It is a very big step from the traditional discrete event simulation [8], where only one simulation thread can run at the same time. However, the absolute temporal barrier is still an obstacle towards highly parallel simulation.

Distributed parallel simulation [9][10] is an extension from the PDES. SystemC models are broken into small executable units and distributed to different host machines to run in parallel. This still suffers from the previously mentioned temporal barrier, and the network speed is another bottleneck limiting the performance.

978-1-5090-3997-5/17/$31.00 ©2017 IEEE

Time-decoupling [11] is an appealing approach for fast SystemC simulation. With parts of the model executing in an unsynchronized way, simulation gets much faster. However, this technique cannot guarantee an accurate simulation result. In other words, it is a trade-off between accuracy and simulation speed. [12] parallelizes the temporal decoupling approach, but some human efforts are required to manually partition and instrument the model.

Out-of-order parallel discrete event simulation (OoO PDES) [13] localizes the simulation time to individual threads, and handles event deliveries and data conflicts carefully with the use of a segment graph infrastructure. This approach achieves a 100% accurate simulation result. However, pointers cannot be effectively analyzed for data conflicts.

SystemC simulations on specialized hardware are also studied. [15] presented a FPGA board based SystemC simulation approach and [16] proposed a multi-threading SystemC simulation on GPU. Such approaches all face the difficulty of manual model partitioning to fit the heterogeneous simulator. In contrast to these approaches, we exploit data-level parallelism in the SystemC model to speedup the simulation without loss of any accuracy.

II. SYSTEMC MODELING OF BITCOIN MINER

In this section, we first introduce the architecture and algorithm of a Bitcoin miner application, and then develop a sequential implementation of Bitcoin miner in SystemC.

A. Bitcoin miner

Bitcoin is a new peer-to-peer digital asset and a decentralized payment system introduced by Satoshi Nakamoto in 2009 [17]. Without a third party, the transactions of Bitcoin are verified by the network nodes running the Bitcoin software, and are recorded into a public ledger, which serves to prevent the double-spending problem. The ledger is made up of Bitcoin blocks, where each *Bitcoin block* contains the validated transactions. Formally, the ledger is called *block chain*. The maintenance of the block chain is also performed by the network nodes.

1) Proof of work: To prevent malicious nodes from modifying the past blocks in the block chain, Bitcoin system requires each node to prove that it has invested a significant amount of work in its creation of the candidate Bitcoin block. This behavior is called *proof of work*. Once a new block is accepted by the Bitcoin network and is appended to the block chain, new Bitcoins will be created and paid as a reward together with some transaction fees to the node which found the block. The proof of work in the Bitcoin system is implemented with a cryptographic hash algorithm. Each computation node is required to find a number called *nonce*, such that when the *block header* (the compression of the whole block content) is hashed using the SHA-256 algorithm along with the nonce, the result is numerically smaller than the network's *difficulty target*. The difficulty target is a 256-bit value shared in the global network. Figure 1 demonstrates the workflow for proof of work of an individual node.

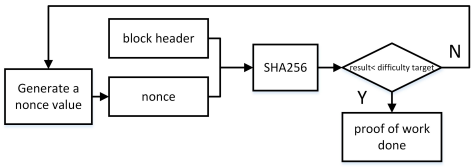

Fig. 1. Flow graph for proof of work

2) SHA-256 algorithm: SHA-256 [18] is a member of the SHA-2 family, which is a set of cryptographic hash functions designed by the National Security Agency (NSA). The SHA-256 function computes with 32-bit words, and has a digest size of 256 bits. Considering its collision-resistant properties, such functions are often used in digital signatures and password protection [19], and its computational complexity fits well the demands of proof of work for the Bitcoin system. The SHA-256 algorithm is briefly described as follows. It first performs the preprocessing, which pads the input message into a new message M with the length of a multiple of 512 bits, then parses M into N 512 bit blocks. The SHA-256 algorithm consists mainly of a loop, as shown in Figure 2. In each step, the 8 intermediate values are updated, and after 64 iterations, a result is generated by cascading them together. Details of the whole algorithm can be found in [18].

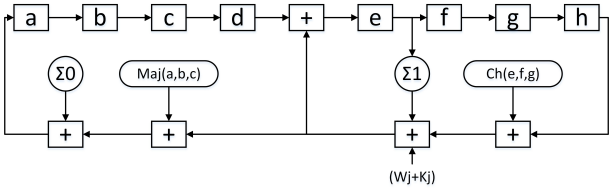

Fig. 2. Flow graph for SHA-256 algorithm [18]

B. Sequential SystemC Bitcoin Miner Model

Figure 3 shows the block diagram of the sequential SystemC Bitcoin miner. It mainly contains two parts. The *dispatcher* is responsible for data transfer and the *scanner* includes all the hashing computations. The design is based on a C reference implementation, CPUminer [20].

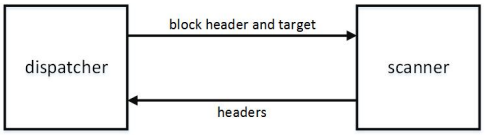

Fig. 3. Block diagram for solo Bitcoin miner

1) Design of the dispatcher for the sequential Bitcoin miner: The dispatch sends the block header and the difficulty target to the *scanner*. Since the main purpose of our research

is to analyze and exploit the parallelization potential of the computation, we design the *dispatcher* in a simple way, where the block header is generated as a fixed value, instead of packing the transactions and deriving the Merkle root hash [21]. This change is valid because the transactions are all unknown and independent, making the block header random to the *scanner*. The difficulty target is also user-defined, rather than obtaining it from the Internet.

Figure 4 depicts the implementation of the *dispatcher* block. In the loop, the *dispatcher* first sends the block header together with the difficulty target to the *scanner* via the output *sc_fifo*. Then it is blocked on the input *sc_fifo* for the result from the *scanner*. In conclusion, the *dispatcher* works both as the stimulus and the monitor in our model.

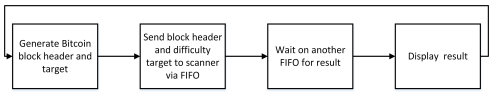

Fig. 4. Flow graph for reference dispatcher block

2) Design of the scanner for the sequential Bitcoin miner: The *scanner* module is the most computationally intensive part of the Bitcoin miner. As depicted in Figure 5, it receives the block header and difficulty target as the input, and then iterates through every possible value of the nonce and generates the corresponding hash. If the hash value is below the difficulty target, the *scanner* sends the nonce together with the hash to the *dispatcher*.

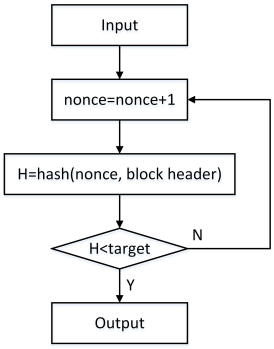

Fig. 5. Flow graph for reference scanner block

3) Test bench and benchmark configuration: The execution time is determined by two parameters in the model. The first is the number of total jobs dispatched to the scanner. The second is the difficulty target. Nonce range and block header have nothing to do with the simulation speed, but they are another two critical parameters in the model. In order to make the experiments reproducible, the four values are fixed on each simulation host, as shown in Table I. The difficulty target in our model is a 256-bit value, starting with 24-bit 0s and followed by 232-bit 1s. The block header is 80-bit wide. Note that the number of total jobs dispatched is 50/100/20 respectively on the three different host processors. This is because of their different CPU clock frequencies. If we were using the same number of jobs, either would one machine finish too early or another run for too long. Besides, the comparison of simulation speed on different host processors is not the focus of this work, so it is only necessary to fix the total number of jobs on each individual host.

TABLE I
CONFIGURATION OF THE BITCOIN MINER

total number of jobs	50/100/20
difficulty target (256 bits)	0x000000FF_FFFFFFFF_..._FFFFFFFF
nonce range	0x00000000 - 0xFFFFFFFF
block header (80 bits)	0x80000000_00000000_..._00000280

Table II lists some results of the simulation. As we can see, the resulting hash values are all smaller than the difficulty target. Furthermore, the C++ reference model also gets the same results. In conclusion, the correctness of our SystemC based Bitcoin miner model is confirmed.

TABLE II
TEST CASE RESULTS

#iterations	hash value
4044822	0x000000bcbe45bba7e37bc9c.....6a681c0fbb0038768fa
589033	0x000000ebc7a887f9a1ff961.....bfb8227597b77453935
9311051	0x000000d627c0a1fdccc2ea4.....cf8aaa68a4b6eeec8e52
6316947	0x000000872acb7cc530faa96.....099093c6b20f9b57f663

III. THREAD-LEVEL PARALLEL BITCOIN MINER MODEL

In the sequential implementation of Bitcoin miner, only one thread is running at the same time, which largely wastes the computation power of multi- and many-core modern processors. In this section, we will propose our thread-level parallel Bitcoin miner design. Thread-level parallelism is exploited by taking advantage of the RISC simulator [14].

A. Overall architecture

Based on the observation that the *scanner* module is the most complex, time-consuming, and computationally intensive block, its optimization is the main focus in our parallel implementation. In Figure 6, the parallel Bitcoin miner block diagram is shown. For simplicity, only two scanners are shown in this figure. It is worth noting that the overall architecture is similar to the previous one, except that there are multiple *scanners* and an additional *synchronizer* module. However, going more deeply into these blocks, important differences arise due to the essential synchronization behavior among the *scanners*. That is, when one *scanner* succeeds in finding a hash value below the given difficulty target, the other *scanners* can abort their current scanning job and start with a new one.

B. Design of the dispatcher for the parallel Bitcoin miner

The block diagram for the dispatcher module is shown in Figure 7. It has multiple outgoing ports and one incoming port. Each output port is connected to a single *scanner*, and

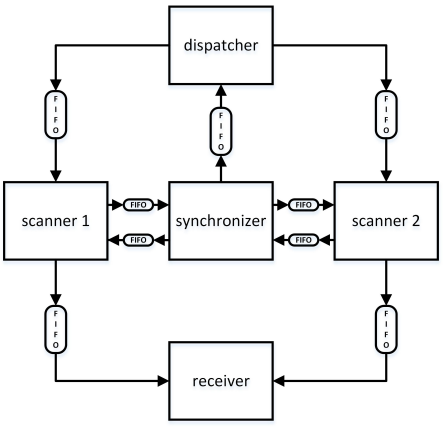

Fig. 6. Parallel Bitcoin miner model with two scanners

the input port is bound to the *synchronizer*. The functionality of the *dispatcher* is quite similar to the one in the sequential reference design. The main difference is that the *dispatcher* now assigns to the *scanners* also a starting point for scanning, so the *scanners*' work will not overlap with each other. After job dispatching, the *dispatcher* block waits on the input port until the *synchronizer* block wakes it up.

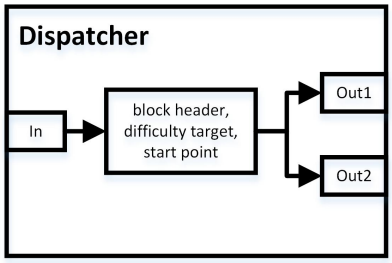

Fig. 7. Dispatcher for parallel Bitcoin miner

C. Design of the scanner for the parallel Bitcoin miner

With thread-level parallelism, multiple *scanners* can work simultaneously. As mentioned previously, the starting points for the scanning of different *scanners* are different. For instance, we consider the case that there are four *scanners*. Then the entire scanning range will be partitioned into four equal pieces, with the first *scanner* starting from 0x00000000, the second from 0x40000000, the third from 0x80000000, and the last one from 0xC0000000, as illustrated in Figure 8. To show the effectiveness of our design, we can assume that the successful nonce values (the nonce which would result in a hash value below the difficulty target when hashed together with the block header) distribute uniformly across the entire range. Based on this assumption, it is obvious that the probability of finding a successful nonce becomes N times larger with N *scanners*, because each *scanner* is independent. Synchronization is another important issue in the parallel *scanner* design. When one *scanner* succeeds in finding the nonce, others have to stop because further scanning on the current proof-of-work becomes meaningless. In order to solve this problem, the *scanners* are synchronized after every scanning step. After each scanning step, a Boolean value representing whether or not a successful nonce is found is sent to the central *synchronizer*, and then the *scanner* waits for a response. The flow graph for each *scanner* is illustrated in Figure 9. When a *scanner* succeeds in finding the nonce, the result hash value is sent to the *receiver* module via another *sc_fifo* channel.

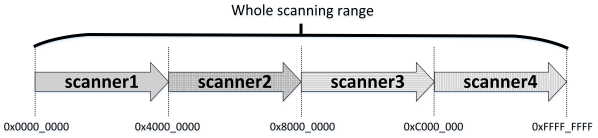

Fig. 8. Illustration of parallel scanning

D. Design of the synchronizer for the parallel Bitcoin miner

The *synchronizer* module serves as the central control block for synchronizing the multiple *scanners*. A loop in this module repeatedly checks the status of each *scanner*, as shown in Figure 10. With the use of *sc_fifo*'s blocking read, it is guaranteed that only when all the *scanners* are checked will the *synchronizer* generate a response back. Depending on the response, the *scanners* can decide to continue their current work or to start a new search.

E. Design of the receiver for parallel Bitcoin miner

The *receiver* block contains a busy waiting loop. On every loop step, one input port is checked to see if there is any value stored.

IV. DATA-LEVEL PARALLEL BITCOIN MINER MODEL

In this section, data-level parallelism (DLP) is applied to the thread-level parallel Bitcoin miner model to further improve the simulation speed.

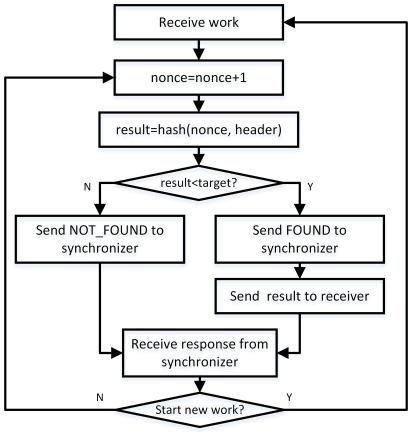

Fig. 9. Scanner for parallel Bitcoin miner

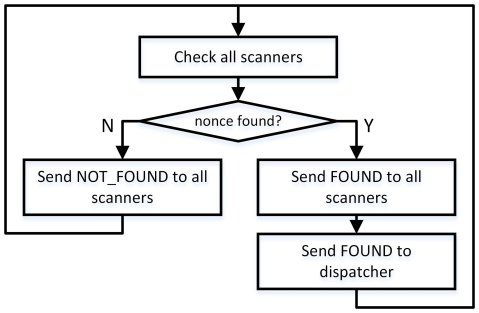

Fig. 10. Synchronizer for parallel Bitcoin miner

A. Basic idea for applying DLP

The *scanner* block is made up of three stages: work receiving, scanning loop and synchronization. In the reference and the thread-level parallel Bitcoin miner, one *scanner* instance only executes a single lane of the scanning iteration at the same time. In order to improve the performance, Single Instruction Multiple Data (SIMD) vectorization is exploited to execute multiple scanning lanes simultaneously. A comparison between the reference *scanner* module and the SIMD *scanner* module is shown in Figure 11. This idea is based on the

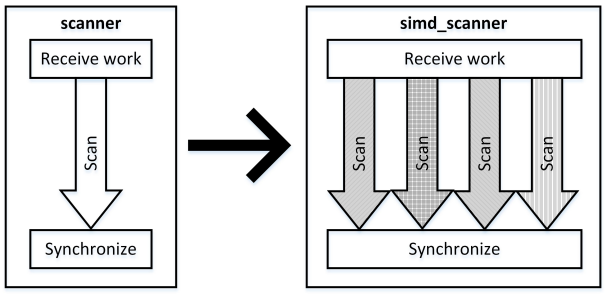

Fig. 11. Comparison between scalar and SIMD scanners

observation that the computation of the hash value inside each iteration step is independent with others. The hashing computation is performed on a constant block header value and an increasing nonce. The nonce value only relies on the loop index. However, because of the *if* and *break* statements in the scanning loop, the control graph becomes divergent, which makes the implementation of SIMD difficult

B. Design of the data-level parallel scanner

In the SIMD *scanner* design, we use SIMD pragma [1] to vectorize the scanning loop and the function calls inside the loop, which are *hash()* and *isSmaller()*.

The first step is to vectorize the *hash()* function. This function contains the SHA256 algorithm and some data padding operations. In its scalar implementation, it can only hash one data point at one time. In the vectorized loop, it is required to hash multiple data points simultaneously, and thus an efficient vectorization is critical to the overall performance.

In our first approach, nothing is changed except a *#pragma simd declare* statement is placed in front of the *hash()* function declaration. However, the result is not as expected. Vectorization is very inefficient. The run time for the Bitcoin miner doubles compared to the scalar one. The reason for that is because this function is too long for the compiler to automatically inline it in the scanning loop. To solve this problem, we manually inlined this function.

For the *isSmaller()* function call, since it only contains some comparison instructions, which is very short in length, the compiler can inline it automatically, and thus no manual modification is needed, except for adding the SIMD pragma in front.

The second step is to vectorize the scanning loop. According to the restriction for vectorization of loops [22], the loop should not contain any *break* statements, and should not modify the same variable (*nonce*, *result*, *found*) on different steps of the iteration. In order to solve this problem, an auxiliary bit array *flag_array* is introduced to replace the *flag* variable. The length of the array is equal to the scanning loop length *LOOP_LENGTH*. A corresponding bit in the array is set if a successful nonce is found on that iteration step, and is not set otherwise. The *result* variable is also changed to a temporary variable belonging to the loop. The *break* statement is removed as well. After the scanning loop, the bit array is checked to find if there is any bit set to true. If so, it means that the scanning loop has successfully found a hash value smaller than the target. The result variable is recalculated based on the position of the set bit.

V. EXPERIMENTAL RESULTS AND EVALUATION

In this section, we present a thorough evaluation of our proposed parallelization approaches on three different host processors. For the experimental setup, we first describe the compilation options and then show the target processor specifications in detail. Finally, we present and evaluate our experimental results on the different processor platforms.

A. Benchmark setup and reproducibility

Four versions of the Bitcoin miner model have been implemented with different parallelization techniques, referred to as sequential (SEQ), thread-level parallel (TLP), data-level parallel (DLP) and the combination of TLD and DLP (TLP+DLP), respectively. In order to evaluate all the implementations under the same conditions, we are using the same source file for the different designs. The number of *scanners* and the SIMD operations are controlled by compile-time macros. The source code is first instrumented by the Recoding Infrastructure for SystemC (RISC) compiler [14], and then compiled with Intel® C++ compiler (ICPC) [23], under optimization level *-O3*.

Execution time is measured with */usr/bin/time*. In our CentOS 6.8 64-bit Linux environment, this is a very precise time measuring tool, which can provide information regarding the system time, the user time and the elapsed time.

[1]The SIMD pragma is recognized by the Intel® compiler as a hint from the designer to vectorize the annotated function or loop

Our experiments are conducted on otherwise idle host processors with CPU frequency scaling turned off. File I/Os operations (i.e. printf()) are also disabled. These settings ensure the accuracy and reproducibility of the measurements. Hyper-threading is turned on so as to maximize parallelism. The setups are shown in Table III.

TABLE III
BENCHMARK SETUP

Linux host OS	CentOS 6.8 64-bit
Compile option	`-O3 -DNDEBUG`
Time measurement	`/usr/bin/time`
File I/O	disabled
CPU frequency scaling	disabled
Work load of other users	0
Hyper-threading	on

B. Processor specifications

The evaluations are performed on three different platforms, respectively the E3-1240 processor (4-core host), the E5-2680 processor (16-core host), and Intel® Xeon Phi™ Coprocessor 5110P (60-core host). Specifications of these processors are listed in Table IV. Here, the peak parallelism is calculated as $\#processors \times \#physical\ cores \times \#SIMD\ lanes$.
Hyper-threading is not in this equation because two hyper-threads on the same physical core only duplicate the status registers, but still share the main execution resources [24]. Our Bitcoin miner application is computational intensive (with integer operations) and has minor communications, making hyper-threading's advantages (reduced core idle time and efficient inter-thread communication) less useful for our application. The experimental results discussed below confirm this observation. Hyper-threading is not effective for Bitcoin mining.
The number of SIMD lanes is calculated as $SIMD\ register\ width\ /\ Operand\ data\ width$. In our Bitcoin miner application, all data types are *unsigned int* which is 32-bit wide. *#pragma simd vectorlengthfor(unsigned int)* statement is used explicitly to set the SIMD data type to *unsigned int*. In the AVX instruction set, the integer SIMD register width is 128-bit wide [2], and thus the number of SIMD lanes of the 4-core and 16-core host is 4. On other hand, the 60-core host uses 512-bit SIMD registers and thus has 16 integer SIMD lanes.

C. Experiments on 4-core host

Table V shows the experimental results with different parallelism techniques on the Xeon E3-1240 processor.
The run time of sequential scalar Bitcoin miner T_{SEQ} is used as the reference for speedup measurements. One thing we noticed is that the absolute execution time of the sequential model is different with different number of scanners. This is because of the random nature of Bitcoin mining, as discussed in Section 3.4. However, with the same number of scanners, the four models (SEQ, DLP, TLP and DLP+TLP) perform the same amount of work.

TABLE IV
PROCESSOR SPECIFICATIONS

	omicron	phi	mic0
processor type	E3-1240	E5-2680	Xeon Phi™ Coprocessor 5110P
#cores	4	8	60
#processors	1	2	1
#total cores	4	16	60
#threads per core	2	2	4
SIMD instruction set	AVX	AVX	AVX-512
integer SIMD register width	128 bits	128 bits	512 bits
#integer SIMD lanes	4	4	16
peak parallelism	16	64	960

The DLP speedup S_{DLP} is stable and approximately 2.96. This value is smaller than the naively expected maximum value 4 for two reasons. One is the needed overhead for SIMD lanes packing and unpacking. The other one is that there are still some sequential operations that cannot be vectorized, such as data padding and communication. According to Amdahl's law, the speedup of 2.96 is reasonable.
The TLP speedup S_{TLP} is naively expected to be equal to the maximum of *#physical cores* and N, where N is the number of scanners in the Bitcoin miner model. As shown in Table V, S_{TLP} is always smaller than the expected maximum. The maximum cannot be reached because of the synchronizations and the context switchings between threads. As we can see in Figure 12, S_{TLP} reaches a maximum of 3.71 when N equals to 4, and then stops increasing because of the limited number of physical cores in the host processor. It even decreases slightly because of the increasing contentions between threads. This also confirms our expectation that hyper-threading technology does not help in our Bitcoin miner application, as discussed in Section V-B.
Finally, by combining TLP and DLP together, we get a

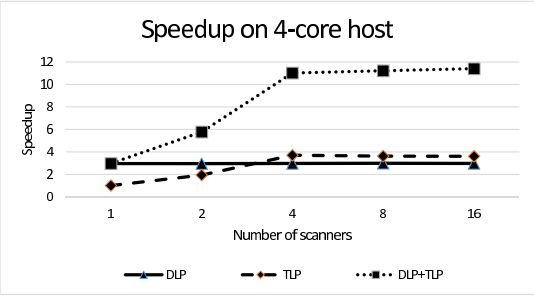

Fig. 12. Speedup on 4-core host with 4 SIMD lanes

maximum speedup of over 11x on the 4-core machine with 4 SIMD lanes. This is an impressive result. Our results confirm the orthogonality of DLP and TLP, and achieve the speedup $S_{DLP+TLP} = S_{DLP} \times S_{TLP}$ as proposed in [4], which shows that the combination of thread- and data-level parallelism can be very efficient to accelerate the SystemC

simulation.

Figure 12 gives an graphical overview of the three speedups. It can be seen that S_{DLP} is constant all the time, and S_{TLP} and $S_{DLP+TLP}$ first increase and then saturate when reaching the upper limit of #physical cores. From this figure, it is clear that combining DLP and TLP together results in a large improvement on the speedup.

D. Experiments on 16-core host

The same models are simulated also on a 2×8-core host, and similar results are achieved, as shown in Table VI.

First, S_{DLP} has a constant value, which agrees with the results on the 4-core host. However, the speedup value of 3.66 is higher. The difference is likely due to the different CPU architectures of the two host machines [25][26]. The 16-core host is advanced in this aspect.

Second, S_{TLP} increases with N, and is limited by the total number of physical cores. A maximum of over 13.7 is reported from the table.

Finally, $S_{DLP+TLP}$ is approximately the product of S_{TLP} and S_{DLP}, which agrees with the previous results on the 4-core host. This again confirms the effectiveness of combining DLP and TLP for faster SystemC simulation.

One thing to be noticed is that, as shown in Figure 13, the value of S_{TLP} at $N = 16$ is lower than expected. From Table VI we can see that it has a value of 11.82, which is only 73.85% of the upper limit. This is because there are two separate processors in this host machine, and the communication overhead between them is much higher than that of the intra-processor communication [27]. With the increase of communication overhead, speedup will decrease. For the same reason, $S_{DLP+TLP}$ is also lower than expected.

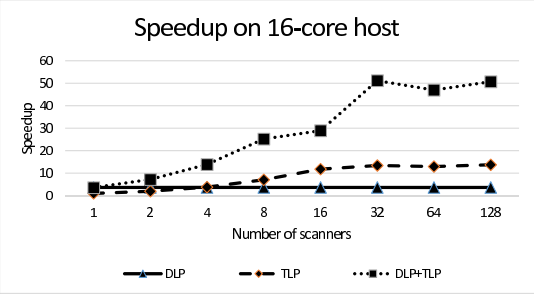

Fig. 13. Speedup on 16-core host with 4 SIMD lanes

E. Experiments on 60-core host

Finally, we simulated our Bitcoin miner model on the many-core Xeon Phi™ Coprocessor host. The results are shown in Table VII. With the use of the 512-bit wide vector registers, a constant DLP speedup S_{DLP} of 9.7 is achieved. On the other hand, TLP speedup S_{TLP} reaches a maximum of 61.73 on the 60-core machine. The upper limit is exceeded in this case. Such phenomenon is often referred to as super-linear speedup [28]. This happens when the working set of a problem is greater than the cache size when executed sequentially, but can fit in each available cache when executed in parallel [29]. Finally, by combining both DLP and TLP techniques together, an impressive speedup of more than 516.6 is reported. However, with $N = 256$, $S_{DLP} \times S_{TLP} = 600.52$, which is approximately +15% higher than $S_{DLP+TLP} = 516.6$. This is because of the well-known memory bandwidth bottleneck for Xeon Phi™ Coprocessor [27]. Overall, these results show the good scalability of parallel simulation on the many-core processor, and confirm the advantage of combining DLP and TLP for SystemC simulation.

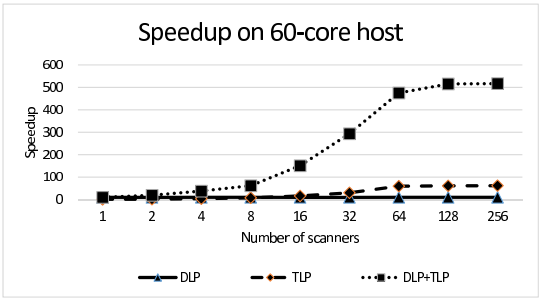

Fig. 14. Speedup on 60-core host with 16 SIMD lanes

VI. CONCLUSION

In this case study, we exploit different levels of parallelism to accelerate the simulation of SystemC based on a Bitcoin miner model. Our approaches include thread-level parallelism (TLP), data-level parallelism (DLP), and the combination of both. The Bitcoin miner is investigated as a case study due to its highly parallel potential and computational complexity. We demonstrate our experiments on a 4-core Intel® E3-1240 processor, a 16-core Intel® E5-2680 processor and a 60-core Intel® Xeon Phi™ Coprocessor. With the combination of DLP and TLP, we achieve a speedup of more than 11x, 50x and 510x, respectively, on the three hosts. These results confirm the advantage of accelerating SystemC simulation through the combination of thread- and data-level parallelism. Moreover, our results also confirm that $S_{DLP+TLP} = S_{DLP} \times S_{TLP}$, as recently shown in [4].

In future work, we plan to investigate these three parallel SystemC simulation techniques (DLP, TLP, DLP+TLP) on more SystemC models as well as other hardware platforms.

REFERENCES

[1] T. Grötker, S. Liao, G. Martin, and S. Swan. System Design With SystemC. 2002.

[2] Intel R Advanced Vector Extensions Programming Reference. https://software.intel.com/sites/default/files/4f/5b/36945.

[3] M. J. Flynn. Some Computer Organizations and Their Effectiveness. IEEE Transactions on Computers, C-21(9):948-960, 1972.

[4] T. Schmidt and R. Dömer. Exploiting Thread and Data Level Parallelism for Ultimate Parallel SystemC Simulation. In Proceedings of the 54th Annual Design Automation Conference 2017, Article No. 79, Austin, TX, USA June 18 - 22, 2017.

[5] Zhongqi Cheng. Thread- and Data-Level Parallel Simulation of a Bitcoin Miner Model in SystemC, 2017

TABLE V
RESULTS ON 4-CORE HOST WITH 4 SIMD LANES: RUNTIME(SECS)/SPEEDUP

#scanner	SEQ	DLP	TLP	DLP+TLP
1	361.68 / 1	122.06 / 2.96	361.69 /1.00	121.50 / 2.97
2	331.96 / 1	112.01 / 2.96	170.90 / 1.94	57.53 / 5.77
4	382.00 / 1	128.30 / 2.97	103.08 / 3.70	34.68 / 11.01
8	362.50 / 1	121.33 / 2.98	99.91 / 3.62	31.39 / 11.22
16	529.25 / 1	177.89 / 2.97	146.49 / 3.61	46.37 / 11.41

TABLE VI
RESULTS ON 16-CORE HOST WITH 4 SIMD LANES: RUNTIME(SECS)/SPEEDUP

#scanner	SEQ	DLP	TLP	DLP+TLP
1	1100.74 / 1	299.98 / 3.66	1098.36 / 1.00	305.57 /2.97
2	993.03 / 1	270.73 / 3.66	506.59 / 1.96	137.67 /5.77
4	1155.46 / 1	314.55 / 3.67	301.64 / 3.83	83.25 /11.01
8	1261.42 / 1	345.51 / 3.65	179.19 / 7.04	50.10 /11.22
16	1617.65 / 1	443.12 / 3.65	136.76 / 11.82	55.9/28.93
32	1954.14 / 1	535.79 / 3.64	146.30 / 13.38	38.21/51.13
64	3037.19 / 1	833.49 / 3.64	234.33 / 12.97	64.62/46.99
128	5030.56 / 1	1370.10 / 3.67	366.35 / 13.74	99.26/50.67

TABLE VII
RESULTS ON 60-CORE HOST WITH 16 SIMD LANES: RUNTIME(SECS)/SPEEDUP

#scanner	SEQ	DLP	TLP	DLP+TLP
1	2669.42 / 1	274.23 / 9.73	2540.91 / 1.05	273.75 / 9.75
2	2383.14 / 1	245.13 / 9.72	1245.76 / 1.91	124.66 / 19.12
4	3321.10 / 1	341.46 / 9.73	838.19 / 3.96	86.68 / 38.31
8	1911.22 / 1	196.38 / 9.73	238.76 / 8.00	30.81 / 62.03
16	3501.69 / 1	359.63 / 9.74	217.97 / 16.06	23.20 / 150.93
32	4843.24 / 1	498.15 / 9.72	160.52 / 30.17	16.51 / 293.35
64	6774.88 / 1	696.40 / 9.73	113.55 / 59.66	14.27 / 474.76
128	11252.35 / 1	1156.35 / 9.73	184.71 / 60.92	21.84 / 515.22
256	18744.61 / 1	1926.87 / 9.73	303.65 / 61.73	36.28 / 516.67

[6] R. Fujimoto. Parallel discrete event simulation. Commun. ACM, 33:3053, Oct. 1990.

[7] R. Dömer, W. Chen, and X. Han. Parallel discrete event simulation of transaction level models. In 17th Asia and South Pacific Design Automation Conference, pages 227-231,2012.

[8] G.S. Fishman. Principles of discrete event simulation. John Wiley and Sons,New York, NY, Jan 1978.

[9] J. Viitanen, P. Sjvöall, M. Viitanen, T. D. Hämäläinen, and J. Vanne. Distributed SystemC simulation on manycore servers. In 2016 IEEE Nordic Circuits and Systems Conference (NORCAS), pages 1-6, 2016.

[10] K. M. Chandy and J. Misra. Distributed Simulation: A Case Study in Design and Verification of Distributed Programs. IEEE Transactions on Software Engineering, SE-5(5):440-452, 1979.

[11] Systemc TLM-2.0. IEEE Standard 1666-2011, 2011.

[12] N. Ventroux and T. Sassolas. A new parallel SystemC kernel leveraging manycore architectures. In 2016 Design, Automation Test in Europe Conference Exhibition (DATE), pages 487-492, 2016.

[13] W. Chen, X. Han, C. W. Chang, G. Liu, and R. Dömer. Out-of-Order Parallel Discrete Event Simulation for Transaction Level Models. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 33(12):1859-1872, 2014.

[14] Tim Schmidt Guantao Liu and Rainer Dömer. RISC Compiler and Simulator, Beta Release V0.4.0:Out-of-Order Parallel Simulatable SystemC Subset. Technical Report CECS-17-07, July 2017.

[15] S. Sirowy, C. Huang, and F. Vahid. Online SystemC emulation acceleration. In Design Automation Conference, pages 30-35, 2010.

[16] M. Nanjundappa, H. D. Patel, B. A. Jose, and S. K. Shukla. SCGPSim: A fast SystemC simulator on GPUs. In 2010 15th Asia and South Pacific Design Automation Conference (ASP-DAC), pages 149-154.

[17] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008.

[18] Descriptions of SHA-256, SHA-384, and SHA-512. http://www.iwar.org.uk/comsec/resources/cipher/sha256-384-512.pdf.

[19] SHA-2. https://en.wikipedia.org/wiki/SHA-2.

[20] pooler, Jeff Garzik, ArtForz. Source code for cpuminer. https://github.com/pooler/cpuminer/issues/13.

[21] Bitcoin block header. https://bitcoin.org/en/developer-reference#block-headers.

[22] The Intel® Intrinsics Guide. https://software.intel.com/sites /landing-page/IntrinsicsGuide/.

[23] User and Reference Guide for the Intel R C++ Compiler 15.0. https://software.intel.com/en-us/compiler_15.0_ug_c.

[24] Hyper-threading Technology. https://en.wikipedia.org/wiki/Hyper-threading.

[25] Intel R E3-1240 Specification. https://ark.intel.com/products/52273/Intel-Xeon-Processor-E3-1240-8M-Cache-3_30-GHz.

[26] Intel R E5-2680 Specification. http://ark.intel.com/products/64583/Intel-Xeon-Processor-E5-2680-20M-Cache-2_70-GHz-8_00-GTs-Intel-QPI.

[27] G. Liu, T. Schmidt, R. Dömer, A. Dingankar, and D. Kirkpatrick. Optimizing thread to-core mapping on manycore platforms with distributed Tag Directories. In 20th Asia and South Pacific Design Automation Conference, pages 429-434, Jan 2015.

[28] Shamoon Jamshed. Commercial software benchmark. In Using HPC for Computational Fluid Dynamics: A Guide to High Performance Computing for CFD Engineers.

[29] Speckenmeyer, Ewald. Superlinear Speedup for Parallel Backtracking. Lecture Notes in Computer Science. 297: 985-993.