UNIVERSITY OF CALIFORNIA,
IRVINE

Design and Evaluation of a Bitcoin Miner SystemC Model with Thread and Data-Level Parallelism

Thesis

submitted in partial satisfaction of the requirements
for the degree of

MASTER OF SCIENCE

in Electrical and Computer Engineering

by

Zhongqi Cheng

Thesis Committee:
Professor Rainer Dömer, Chair
Professor Mohammad A. Al Faruque
Professor Pai Chou

2017

DEDICATION

To,
My parents

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I would first like to express my gratitude to my thesis advisor Professor Rainer Dömer. This was my first time doing academic research, and was a fully rewarding experience. His encouragement and support were really invaluable.

I would also like to thank Dr. Guantao Liu and Tim Schmidt for their supports during my research. They were kind enough to take time out of their busy schedule to help with my problems, for which I am most grateful.

I would also like to thank Professor Mohammad A. Al Faruque and Professor Pai Chou as the committee member of this thesis. I am gratefully indebted to them for their very valuable comments on this thesis.

Finally, I must thank my parents for their strong supports and continuous encouragements throughout my study. This accomplishment would not have been possible without them. Thank you.

ABSTRACT OF THE THESIS

Design and Evaluation of a Bitcoin Miner SystemC Model with Thread and Data-Level Parallelism

By

Zhongqi Cheng

MASTER OF SCIENCE in Electrical and Computer Engineering

University of California, Irvine, 2017

Professor Rainer Dömer, Chair

SystemC based Electronic System Level (ESL) design is one of the most efficient approaches for modeling, simulating, and validating of embedded system models. However, the rapid growing design complexity has become a big obstacle and dramatically increased the time required for simulation. This thesis focuses on exploiting different level of parallelism including data and thread-level parallelism to accelerate the simulation of SystemC based ESL design.

Bitcoin miner is chosen as a benchmark because of its high potential for parallel execution and computational complexity. The 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 data and thread-level parallelism for higher SystemC simulation speed, and can serve as a benchmark for future optimization of system level design and modeling.

Chapter 1

Introduction

SystemC [36] is a widely used modeling language for Electronic System Level (ESL) design and also provides a simulation framework for validation and verification [19]. With the rapid 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 [23], 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 [34].

In this paper, we exploit different level of parallelism, including TLP, DLP and the combina-

tion of both to accelerate SystemC simulation. ILP is not considered in our thesis because it is invisible to the programmers and automatically applied at the hardware level. Bitcoin miner is used as a case study due to its high potential for parallel execution.

1.1 Related work

Various approaches have been proposed to speedup the simulation of SystemC models. Parallel discrete event simulation (PDES) is well studied in [24] and [21]. It is a very big step from the traditional discrete event simulation [22], 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 [40][18] 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.

Time-decoupling [15] 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. [39] 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) [19] localizes the simulation timestamp to each individual thread, and handles event deliveries and data conflicts carefully with the use of segment graph infrastructure. This approach achieves a 100% accurate simulation result. However, pointers cannot be effectively analyzed for data conflicts.

SystemC simulations on specialized hardwares are also studied. [35] presented a FPGA board based SystemC simulation approach, and [32] proposed a multi-threading SystemC

simulation on GPU. Such approaches all faces the difficulty of 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 lose of any accuracy.

1.2 Bitcoin miner

Bitcoin is a new peer-to-peer digital asset and a decentralized payment system introduced by Satoshi Nakamoto in 2009 [31]. 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 avoid 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.

1.2.1 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 1.1 demonstrates the workflow for proof of work of an individual node.

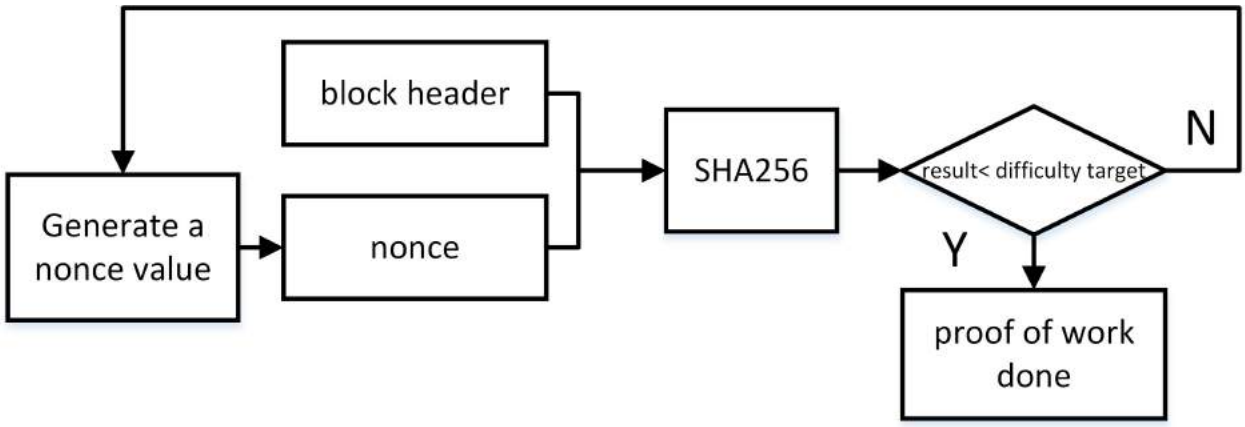

Figure 1.1: Flow graph for proof of work

1.2.2 SHA-256 algorithm

SHA-256 [6] 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 function are often used is digital signatures and password protection [11], 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 1.2. 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 [6].

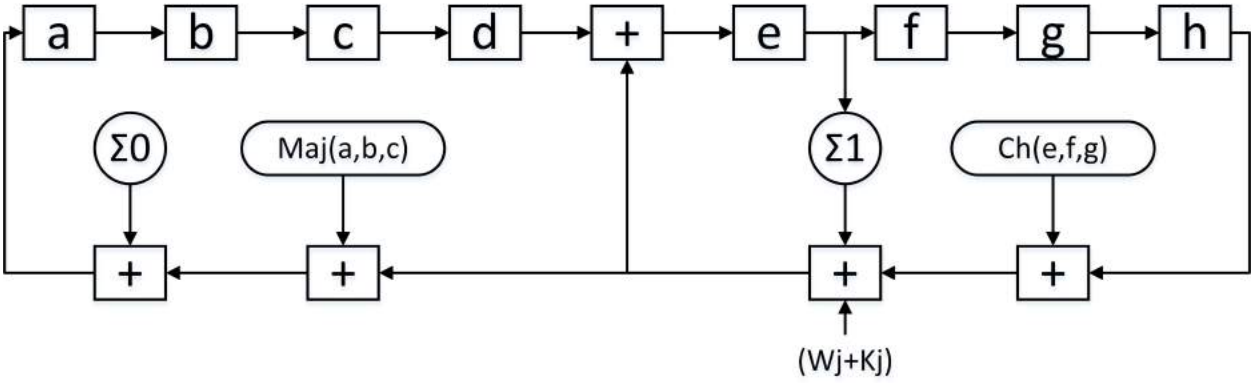

Figure 1.2: Flow graph for SHA-256 algorithm [6]

1.3 SystemC simulation

SystemC [36] is an Electronic System-Level (ESL) Design language and simulation framework which is widely used to facilitate the modeling and verification process in hardware and system design at the abstract specification level.

In this section, three popular simulation approaches are presented, namely discrete event simulation (DES), parallel discrete event simulation (PDES) and out-of-order parallel discrete event simulation (OoO PDES).

1.3.1 Discrete event simulation (DES)

Discrete event simulation [22] is the inherent simulation approach for the SystemC language [16]. It utilizes a central scheduler to manage multiple concurrent threads, which results in temporal barriers (namely time and delta cycle) in the SystemC simulation [20]. According to the cooperative multitasking semantics of the SystemC standard IEEE 1666-2011 [16],

most SystemC simulator implementations have only one thread active at the same time and thus cannot utilize the parallel computing resources available on multi-core (or many-core) processor hosts. This significantly limits the execution speed of SystemC simulation.

Figure 1.3 shows the algorithm for the traditional discrete event simulator [20]. The simulation is driven by events and time advances. At any time, the simulator runs a single thread from the *ready* queue. Once the ready queue is empty, the simulator walks through the *wait* queue to move the threads that should be waken up to the ready queue. If the ready queue is still empty after event delivery, time will be advanced and corresponding threads in the *waitfor* queue are moved to the ready queue. If the ready queue is still empty, then the simulation reaches the end.

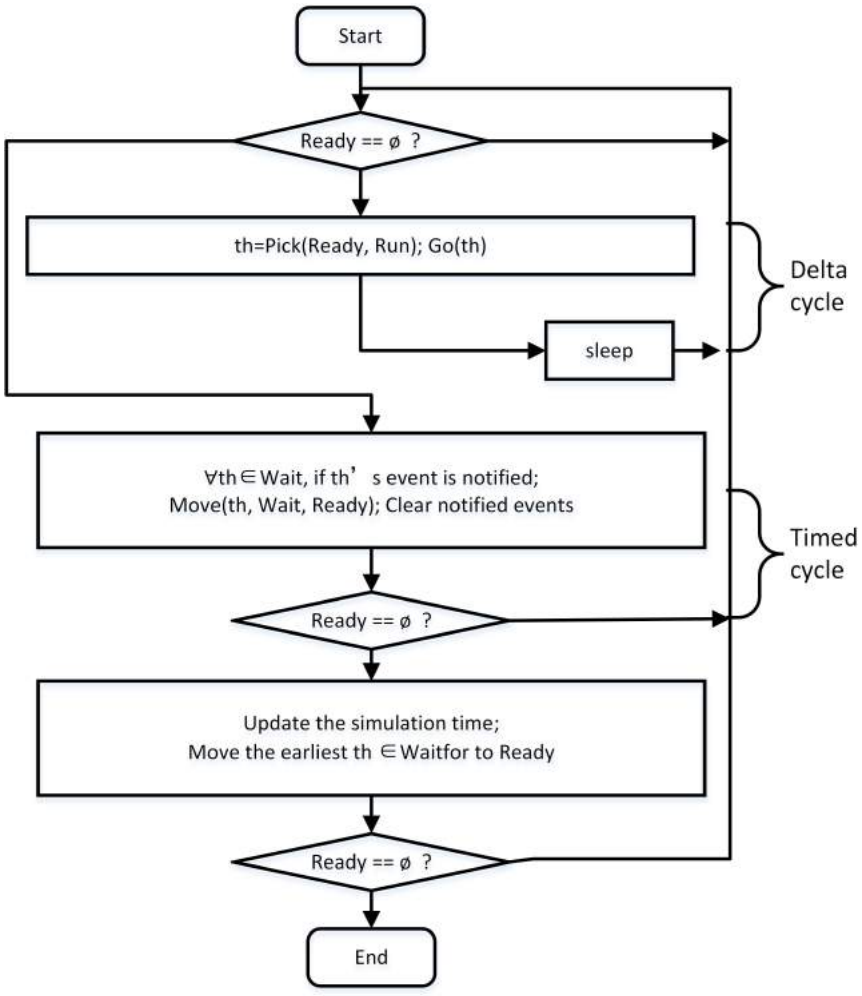

Figure 1.3: Discrete event simulation algorithm [20]

1.3.2 Parallel discrete event simulation (PDES)

In order to provide faster simulation and due to the inexpensive availability of parallel processing on todays multi-core (and many-core) processors, parallel discrete event simulation (PDES) has recently gained again significant attentions [24][21]. The PDES simulator issues multiple threads (i.e. SC_METHOD, SC_THREAD and SC_CTHREAD) at the same time and dispatches them onto the available cores in parallel. In turn, the simulation speed increases significantly. The scheduler in the parallel simulator works the same way as the sequential simulator with one exception. The main difference is that it picks multiple threads from the ready queue in each cycle, and runs them in parallel. The algorithm is shown in Figure 1.4.

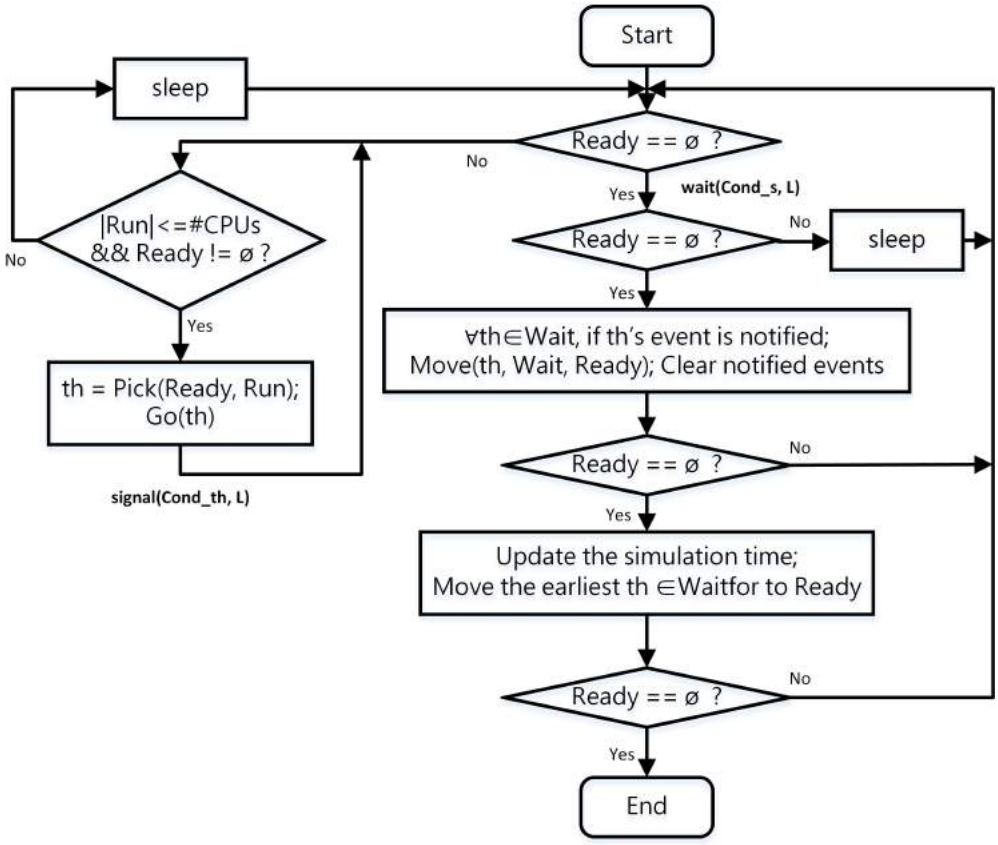

Figure 1.4: Parallel discrete event simulation algorithm [20]

1.3.3 Out-of-order parallel discrete event simulation (OoO PDES)

In the regular synchronous PDES, time advance happens globally. That is, earlier completed threads have to wait until all other running threads reach the simulation cycle barrier, even if they do not have any conflicts with each other in the future. The strict total order of time imposed by the synchronous PDES is still a limit to high performance parallel simulation. To solve this problem, out-of-order parallel discrete event simulation (OoO PDES) is proposed [19].

In the OoO PDES, the simulation time is local to individual threads and event delivery is carefully handled. With the partial order of time, the system model can be simulated without loss of accuracy and increases the utilization of the parallel computation infrastructure. Figure 1.5 depicts the algorithm of OoO PEDS. The RISC simulator [33] used in this work implements OoO PDES as its scheduling algorithm.

1.4 RISC compiler

The Recoding Infrastructure for SystemC (RISC) [33] is essential to realize the OoO PDES approach. Figure 1.6 shows the design flow using the RISC compiler and simulator. As shown in this figure, the input SystemC model file is sent to the RISC compiler, and the RISC compiler generates an instrumented intermediate model. Then this model is linked against the parallel RISC SystemC library by the target compiler (a regular C++ compiler) to produce the final executable file [25].

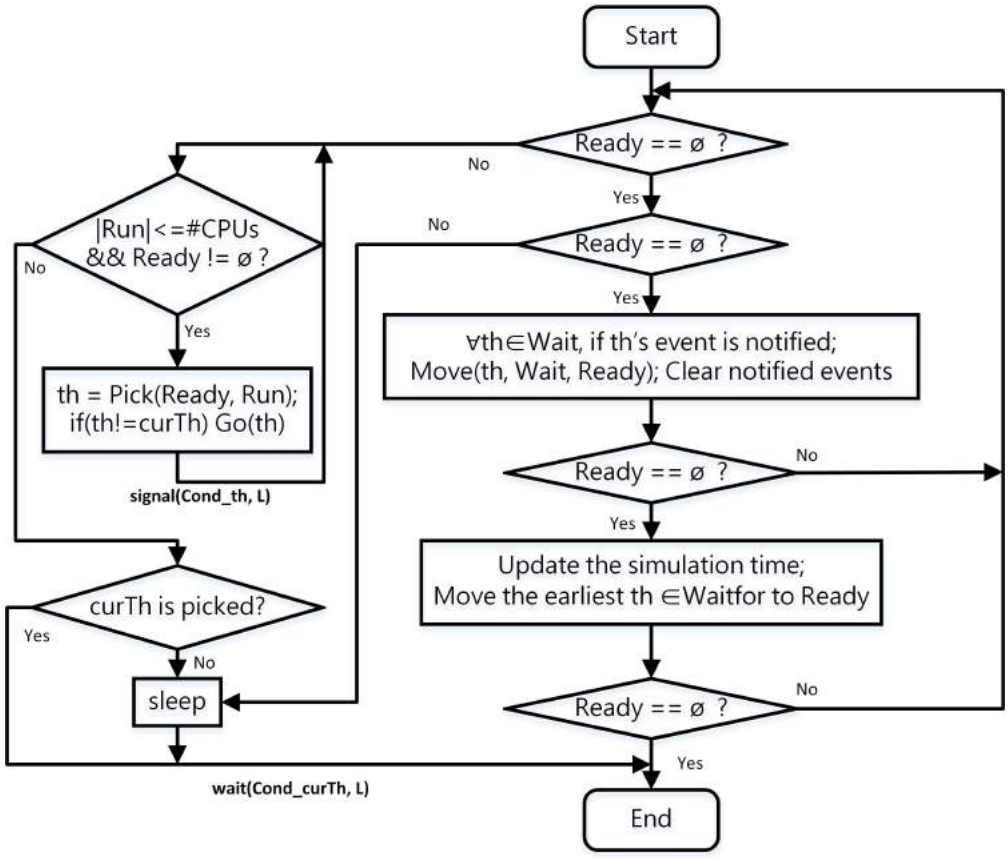

Figure 1.5: Out-of-order parallel discrete event simulation algorithm [19]

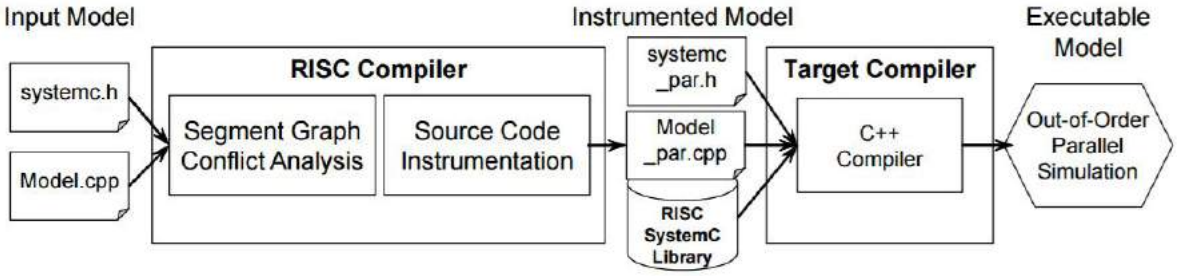

Figure 1.6: RISC compiler and simulator for OoO PDES of SystemC [25]

1.5 Single instruction, multiple data (SIMD)

Single Instruction, Multiple Data (SIMD) is a parallel execution model that performs the same instructions over a vector of input data points simultaneously, whereas traditional scalar operations only take a scalar value and yield a scalar result [28]. The most important feature of SIMD is that it exploits data level parallelism, but not concurrency, meaning that the computations are parallel but only one instruction is given at the same time. Thus, for all the input data points, the operations performed on them are exactly the same. For instance, if the SIMD system works by loading four data points at a time, the same vector operation will be applied to all the four values at once. Nowadays, most modern CPUs introduce special instruction sets to support the SIMD feature to improve the performance of computations. Common examples of such instruction sets include [28]

- Intel MMX (64 bit registers, since 1996)
- AMD 3DNow! (128 bit registers, since 1998)
- Intel Streaming SIMD Extensions (SSE, 128 bit, since 1999)
- Freescale/IBM/Apple AltiVec (also VMX, 128 bit, since 1999)
- ARM NEON (128 bit, since 2005)
- Intel Advanced Vector Extensions (AVX, 256 bit, since 2011)
- Intel Larrabee New Instructions (LRBni, 512 bit, since 2013)
- Intel AVX-512 (512 bit, since 2015)

1.5.1 Advantages of SIMD

Although compared with the scalar instructions, SIMD instructions operate on more registers, the execution time is approximately the same as its scalar counterpart. That is to say that, SIMD instructions can achieve a much higher throughput over the same time interval. So if the SIMD instructions are used properly, they can greatly increase the performance of the applications. Another benefit of the SIMD model is regarding energy efficiency, because it only requires one instruction fetch and decode phase for multiple data operations. [28]
The SIMD width defines the number of inputs that an SIMD instruction can process in parallel. It is derived from the number of values that can be stored in a single SIMD register. For instance, in the Intel® Xeon E3-1240 multi-core processor used in this thesis, the vector registers are 256-bit long each [34]. That is to say that the SIMD registers for this processor have four double-precision SIMD lanes, and thus the SIMD instructions on double values can operate on the four lanes in parallel.

1.5.2 Limitations for SIMD

Loops are often the most interesting part for vectorization in algorithms. However, SIMD instructions are not always easily applied to loops. [17] demonstrates the various limitations that exist to vectorize a loop.
First, each lane in the vectorization unit must perform the same operation. The control flow for each loop should not change. That is to say, jump and switch statements are not allowed. However, on the other hand, if-then-else statements are allowed depending on whether it can be transfered to a masked assignment.
Second, a loop can only be vectorized when the iterations of the loop is countable. This is because vectorization is often achieved through loop unrolling. Furthermore, the index

variable should not be modified in the loop body. And, no data-dependent exit conditions are allowed.

Finally, no backward carried data dependencies (e.g. $A[i] = A[i-2] + 2$) can be in the loop body. This allows consecutive iterations of the original loop to be executed simultaneously in a single iteration of the unrolled, vectorized loop.

1.6 Xeon Phi™ many-core coprocessor

The Intel® Xeon Phi™ Coprocessor is a bootable host processor that delivers massive parallelism and vectorization to support the most demanding high-performance computing applications. It is based on the Intel® Many Integrated Core (Intel® MIC) architecture [9], which scales with many small cores. Each contains a powerful 512-bit vector processing unit (SIMD unit).

In our thesis, we are using the Xeon Phi™ 5110P Coprocessor, which has the following specifications. [38]

- 60 cores, 240 threads (4 threads/core)
- 1.053 GHz clock frequency
- 1 TeraFLOP double precision theoretical peak performance
- 8 GB memory with 320 GB/s bandwidth
- 512-bit wide SIMD vector engine
- 32 KB L1, 512 KB L2 cache per core
- Fused Multiply-Add (FMA) support

The overall architecture is shown in Figure 1.7 [29]. With the 512-bit vector unit, the Xeon Phi^{TM} Coprocessor can process 16 single precision or 8 double precision data values using only one SIMD instruction. However, because of the low clock frequency, even a small amount of sequential work may cause a performance bottleneck. In order to achive high performance on the Xeon Phi^{TM} Coprocessor, applications need to efficiently exploit a high degree of thread level parallelism and use the wide SIMD registers [38].

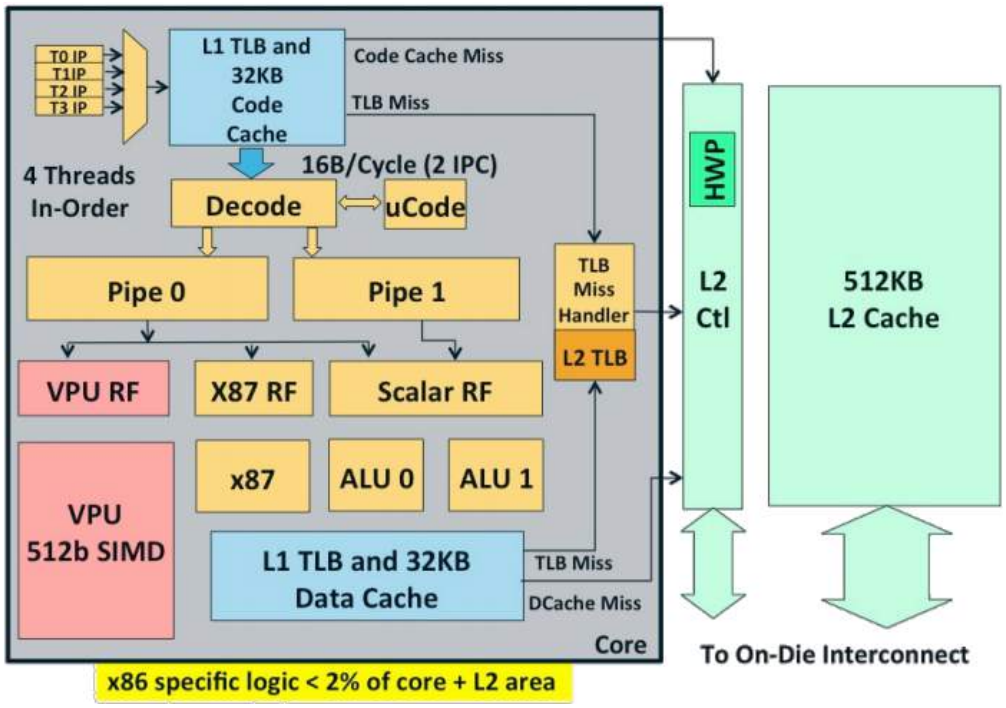

Figure 1.7: Architectural overview of an Intel® Xeon Phi^{TM} core [29]

1.7 Goals and contributions

This thesis aims to make the following contributions:

- We rewrite the reference C++ Bitcoin miner project into an appropriate SystemC model. Considering its high parallel potential and computational complexity, Bitcoin

miner is a very good test bench for evaluating parallel SystemC models.

- We evaluate the performance of thread-level parallelism in the context of the RISC simulator. We expect the speedup ratio to increase linearly with more simulation threads.

- We exploit data-level parallelism on top of thread- level parallelism for a fast SystemC simulation. SIMD pragmas are expected to effectively vectorize loops and functions.

- We analyze the scalability of the parallel SystemC simulation on a many-core Xeon Phi^{TM} Coprocessor. Considering the low performance of a single node in the MIC architecture, scalability becomes the most critical factor for high PDES speed.

Chapter 2

Sequential Bitcoin Miner Model

In this section, we first develop the implementation of a sequential Bitcoin miner in SystemC, which serves as a reference model of Bitcoin miner application and as an introduction to the two modules: *dispatcher* and *scanner*.

2.1 Solo Mining

Bitcoin mining today takes on two forms, solo mining and pooled mining. In this paper, the Bitcoin miner performs solo mining and is based on the reference implementation: CPUminer [27]. As shown in Figure 2.1, solo miner contains two parts. The software (*dispatcher*) is responsible for data transfer and the hardware (*scanner*) includes all the hashing computations.

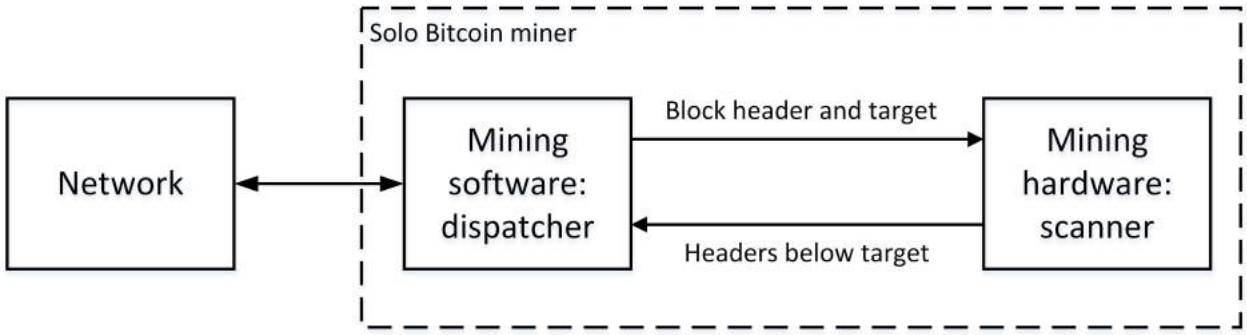

Figure 2.1: Block diagram for solo Bitcoin miner

2.2 Design of dispatcher for sequential Bitcoin miner

In our design, the software part is implemented in the *dispatcher* module. It sends the block header and the difficulty target to the hardware part, that is, the *scanner*. Since the main purpose of our research is to analyze and exploit the parallelization potential of 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 [4]. 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 2.2 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 FIFO. Then it is blocked on the input FIFO for the result from the *scanner*. In conclusion, the *dispatcher* works both as the stimulus and the monitor in our model.

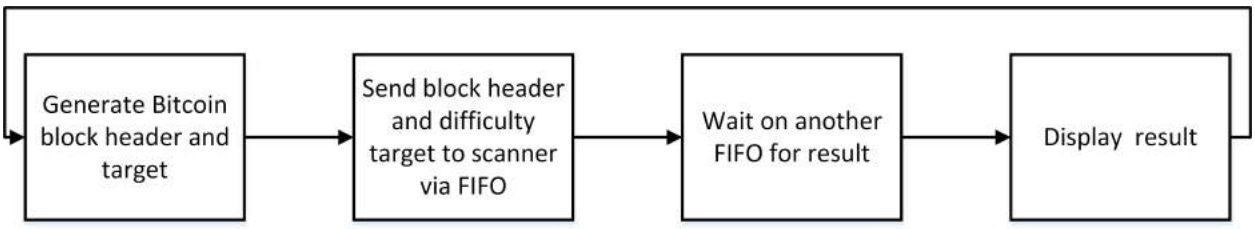

Figure 2.2: Flow graph for reference dispatcher block

2.3 Design of scanner for sequential Bitcoin miner

The *scanner* module is the most computational part for the Bitcoin miner. As depicted in Figure 2.3, 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 the *scanner* sends the nonce together with the hash to the *dispatcher*.

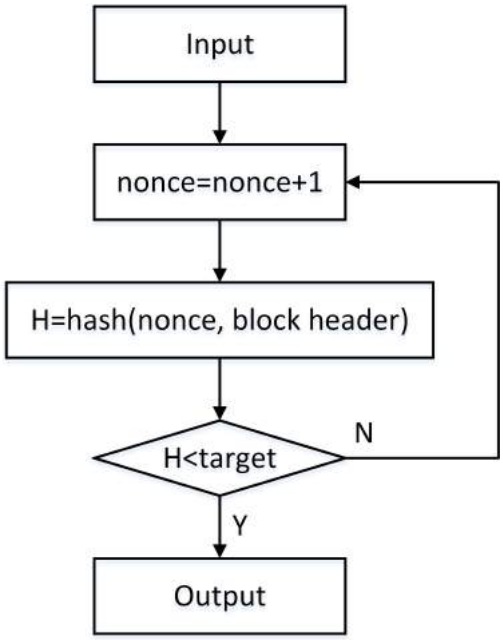

Figure 2.3: Flow graph for reference scanner block

2.4 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 machine, as shown in Table 2.1. 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. To be noticed, the numbers of total jobs dispatched are 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 machine.

Table 2.1: 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 2.2 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 can be confirmed.

Table 2.2: Test case results

#iterations	hash value
4044822	0x000000bcbe45bba7e39b5ef7bc9c839f1217c69736c476a681c0fbb0038768fa
589033	0x000000ebc7a887f9a1f5851ff961e030cd5e807ad9541bfb8227597b77453935
9311051	0x000000d627c0a1fdccc7d18d2ea4698c52d415aed798cf8aaa68a4b6eeec8e52
6316947	0x000000872acb7cc530f0f212aa961db20c77ba633f0d099093c6b20f9b57f663

Chapter 3

Thread-Level Parallel Bitcoin Miner Model

Bitcoin mining compromises three stages: *work dispatching*, *scanning* and *result receiving*. In the reference implementation in Chapter 2, only one thread is running at the same time, which largely wastes the computation power of the multi- and many-core modern processor. In this chapter, we will propose our parallel Bitcoin miner design. Thread level parallelism is exploited by taking the advantage of the state-of-art RISC simulator.

3.1 Parallel Bitcoin miner design

Based on the observation that the *scanner* module is the most complex, time-consuming, and computational intensive block, its optimization is the main concentration in our parallel implementation. In Figure 3.1, the parallel Bitcoin miner block diagram is shown. For simple illustrative purpose, only two scanners are in this figure. It is worth noting that the overall architecture is similar to the reference 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 behaviors among *scanners*. That is, when one *scanner* succeeded in finding a hash value below the given difficulty target, other *scanners* can abort their current scanning job and start with a new one.

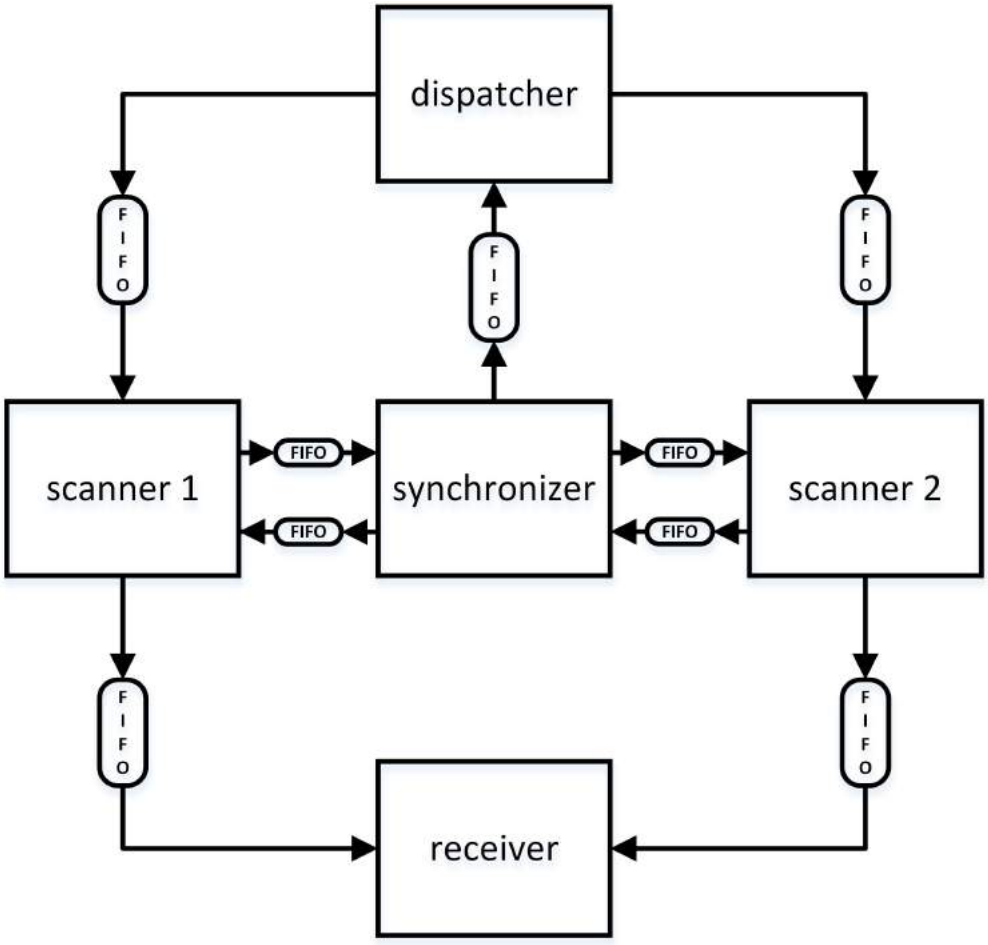

Figure 3.1: Parallel Bitcoin miner model with two scanners

3.2 Communication via sc_fifo

In SystemC based ESL design, communication between different blocks is performed via channels. In our parallel Bitcoin miner design, *sc_fifo* channels are used.

The *sc_fifo* is a predefined SystemC primitive channel designed to model the behavior of a

FIFO, that is, a first-in first-out buffer [5]. Each FIFO has a number of slots for storing data values. The number of slots is fixed when the channel is constructed. Default slots size is 16. In our implementation, the number is set to one because there is always no more than one value in the buffer.

3.3 Design of dispatcher for parallel Bitcoin miner

The block diagram for the dispatcher module is illustrated in Figure 3.2. It has multiple outgoing ports and one incoming port. Each output port is connected to a single *scanner*, and 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* as well 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.

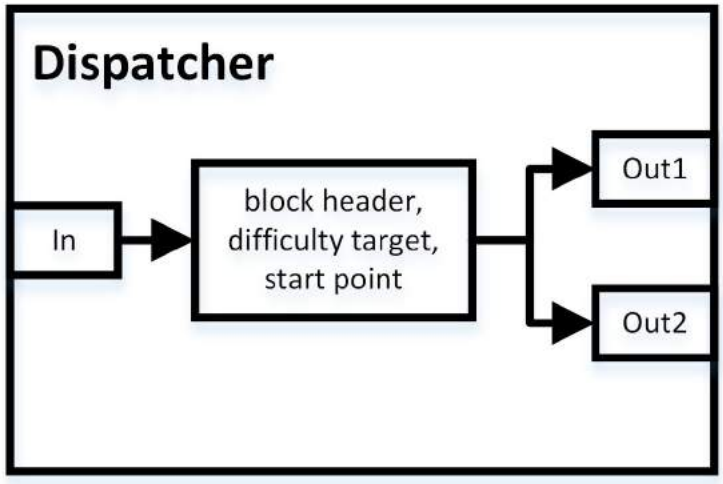

Figure 3.2: Dispatcher for parallel Bitcoin miner

3.4 Design of scanner for 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 3.3. 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 3.4. When a *scanner* succeeds in finding the nonce, the result hash value is sent to the *receiver* module via another FIFO channel.

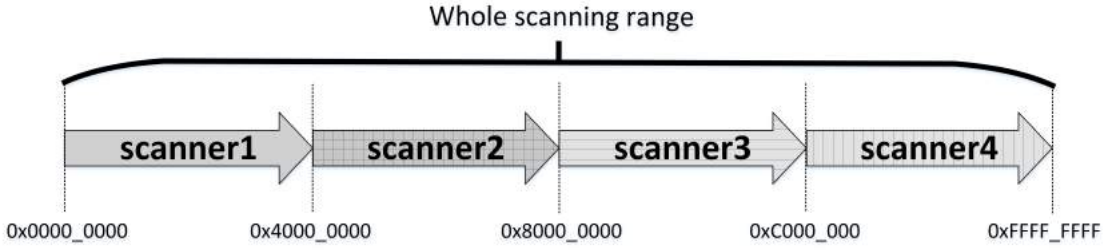

Figure 3.3: Illustration of parallel scanning

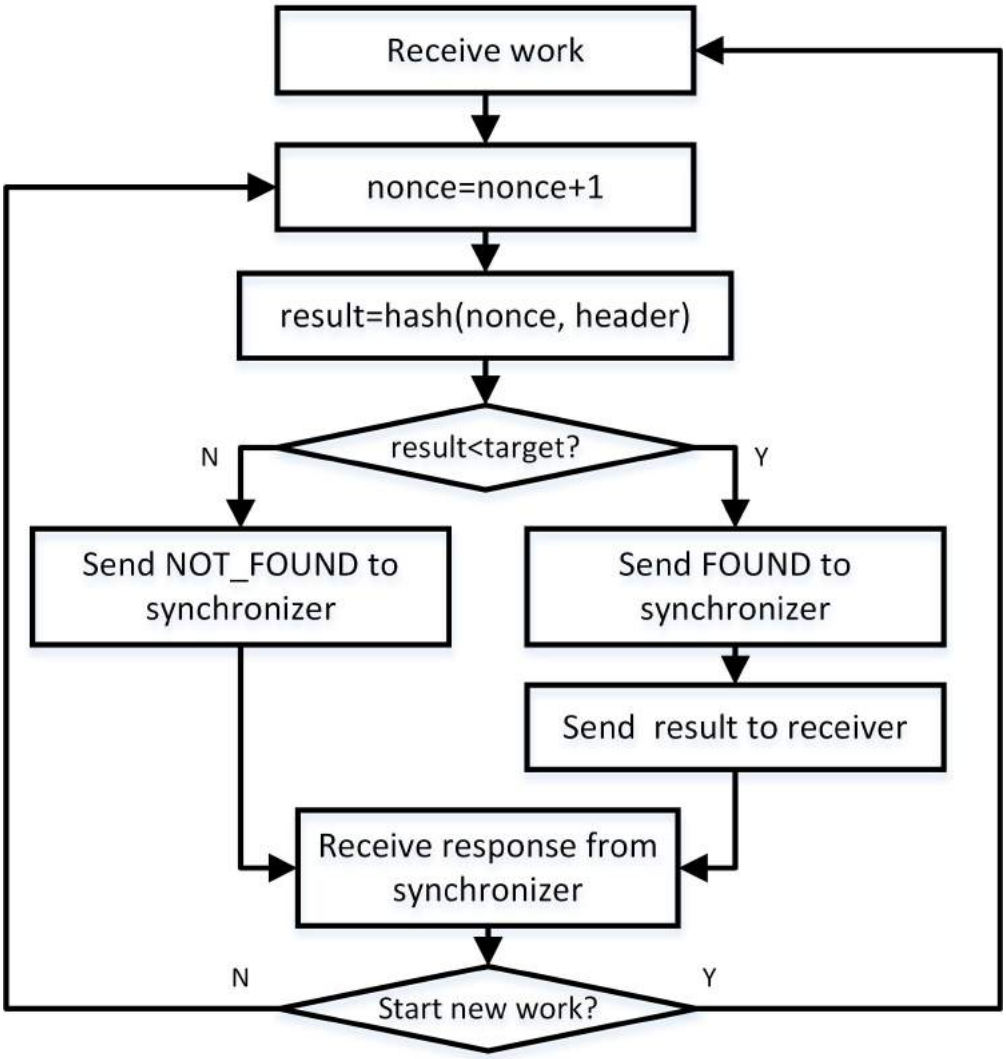

Figure 3.4: Scanner for parallel Bitcoin miner

3.5 Design of synchronizer for 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 3.5. 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.

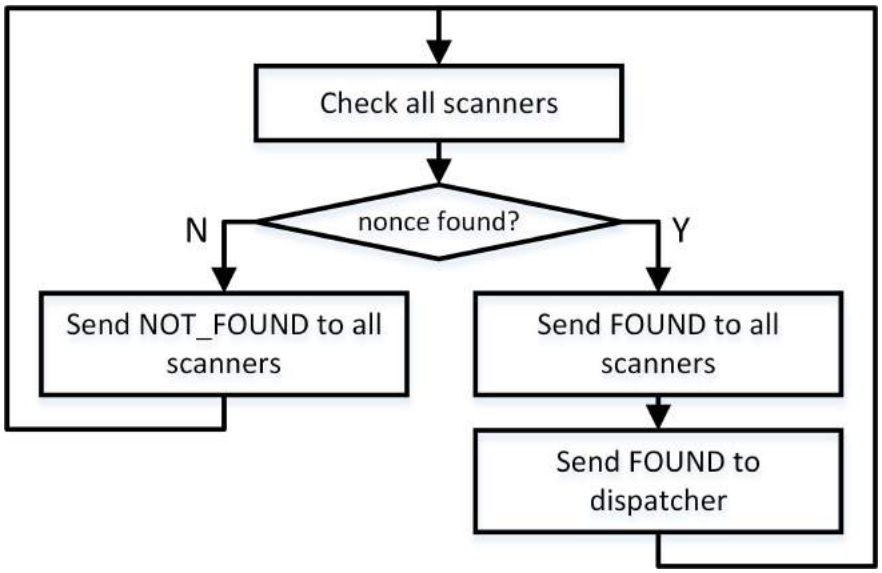

Figure 3.5: Synchronizer for parallel Bitcoin miner

3.6 Design of 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. Listing 3.1 shows the logic for the *receiver* module.

```
1 #define M(n) result_from_scanner##n
2 for(int i=0;i<N;i++){
3       if(M(i).num_available())      M(i).read(result);
4 }
```

Listing 3.1: Busy waiting loop for receiver module

3.7 Shortcomings of current model

Although the implementation described previously can basically perform parallel mining, it still suffers from a few shortcomings.

Firstly, there are too many FIFO channels, approximately four FIFOs per *scanner*, which makes the model complicated. On the other hand, it is a waste of memory to instantiate too many FIFO channels.

Secondly, the hard barrier introduced by synchronizing slows down the simulation. Each *scanner* is hung up after every iteration step, and can only continue when all the *scanners* have reached the barrier. Besides, the frequent context switching places a very high overhead on the simulation speed.

Thirdly, since the *receiver* block contains a busy waiting loop, one hardware thread is always assigned to it, which largely wastes the processors computation power. Furthermore, this does not simulate in discrete event SystemC simulator.

To solve the first two problems, the synchronization pattern is optimized, which includes modification of the *scanner* and the *synchronizer* module. A user-defined channel is implemented to solve the third problem.

3.8 Optimized parallel Bitcoin miner

In this section, an optimized parallel Bitcoin miner is proposed to overcome the three shortcomings mentioned previously. The block diagram for the optimized design is shown in Figure 3.6. We introduce a user-defined channel, namely *result submission channel* to replace the synchronization FIFOs. The details about each block are described in the following sections.

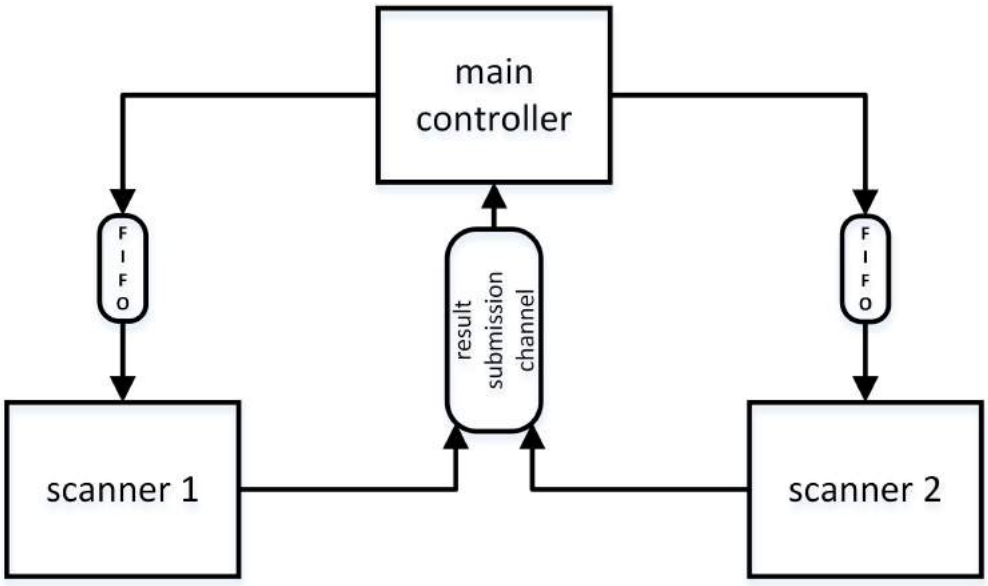

Figure 3.6: Optimized Parallel Bitcoin miner with two scanners

3.8.1 Main controller

In the optimized design, the *dispatcher*, *receiver* and *synchronizer* modules are combined into a new one, named *main controller*. This is based on the observation that *dispatcher* will never run in parallel with the other two. The *synchronizer* and the *receiver* both perform polling to check the status of the *scanners*. In other words, their functionalities overlap with each other.

Figure 3.7 illustrates the *main_controller*. In the first part, it sends the required data to the

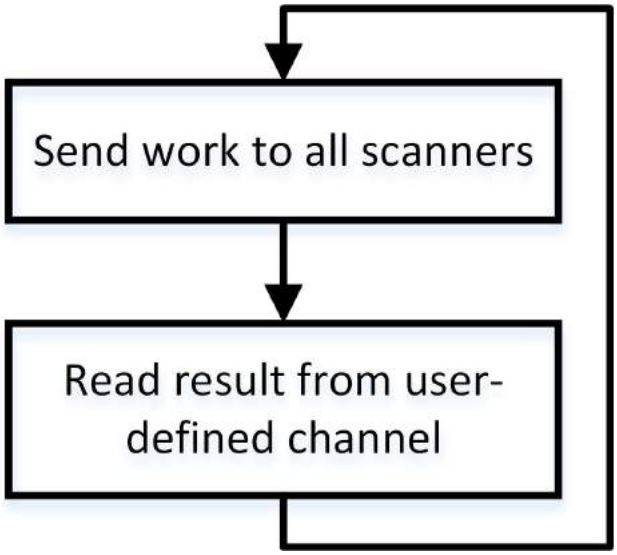

Figure 3.7: Optimized main controller for parallel Bitcoin miner

scanners through FIFOs. Then it waits on the *result submission channel* for result. The channel is described in detail in the next section.

3.8.2 User-defined channel for synchronization

We define an N-to-one channel to which all the *scanners* are writers whereas the *main controller* is the reader. The channel is for the purpose of non-blocking write and blocking read, and is named *result submission channel* in our design.

The implementation of the channel is shown in Listing 3.2. Inside the channel there is an array of length N, where N is the total number of *scanners*. When a *scanner* invokes the *write_result* function, it puts the value into the corresponding bucket of the array, such that the i^{th} *scanner* writes to the i^{th} bucket. Besides, it also sets the member variable *pos* to i, and *result_empty* to 0 so that when the *main controller* reads from the channel, it can determine which bucket to read. When *result_empty* is 1, meaning that the channel is empty, the *read_result* function will be blocked, until the event *result_written_event* is notified.

```
1  void read_result(SCAN_WORK& Y)
2  {
3          if(result_empty==1){
4                  wait( result_written_event );
5          }
6          Y=result_buf[pos];
7          result_empty=1;
8  }
9
10 void write_result(SCAN_WORK X, int n)
11 {
12         result_buf[n]=X;
13         pos=n;
14         result_empty=0;
15         result_written_event.notify(SC_ZERO_TIME);
16 }
```

Listing 3.2: read_result and write_result functions in the result submission channel

3.8.3 Optimized scanner

Figure 3.8 shows the implementation of the optimized *scanner*. One of the main differences is that synchronization is performed every *LOOP_LENGTH* scanning steps. The choice of *LOOP_LENGTH*'s value is a trade-off between the CPU utilization and the non-effective computations. On one hand, if *LOOP_LENGTH* is too small, more CPU time will be spent on context switchings and synchronizations. On the other hand, if *LOOP_LENGTH* is too large, when one *scanner* has succeeded, other *scanners* cannot get notified in time because

synchronization happens late, which leads to a waste of computation power. Based on the observation that with the choices of the four values in Table 2.1, a successful nonce is often found after millions of iteration steps in our implementation, $LOOP\_LENGTH$ is set to 10000 in our design.

Secondly, in order to remove the unnecessary timing barriers, synchronization is performed in another way. In the old design, synchronization comprises a blocking write and a blocking read. In the optimized implementation, the blocking write is removed, and the blocking read is replaced with a non-blocking one. When it detects that there is a new job in the input port, it aborts the current search and starts a new one. These changes reduce the number of the FIFO channels and the model complexity.

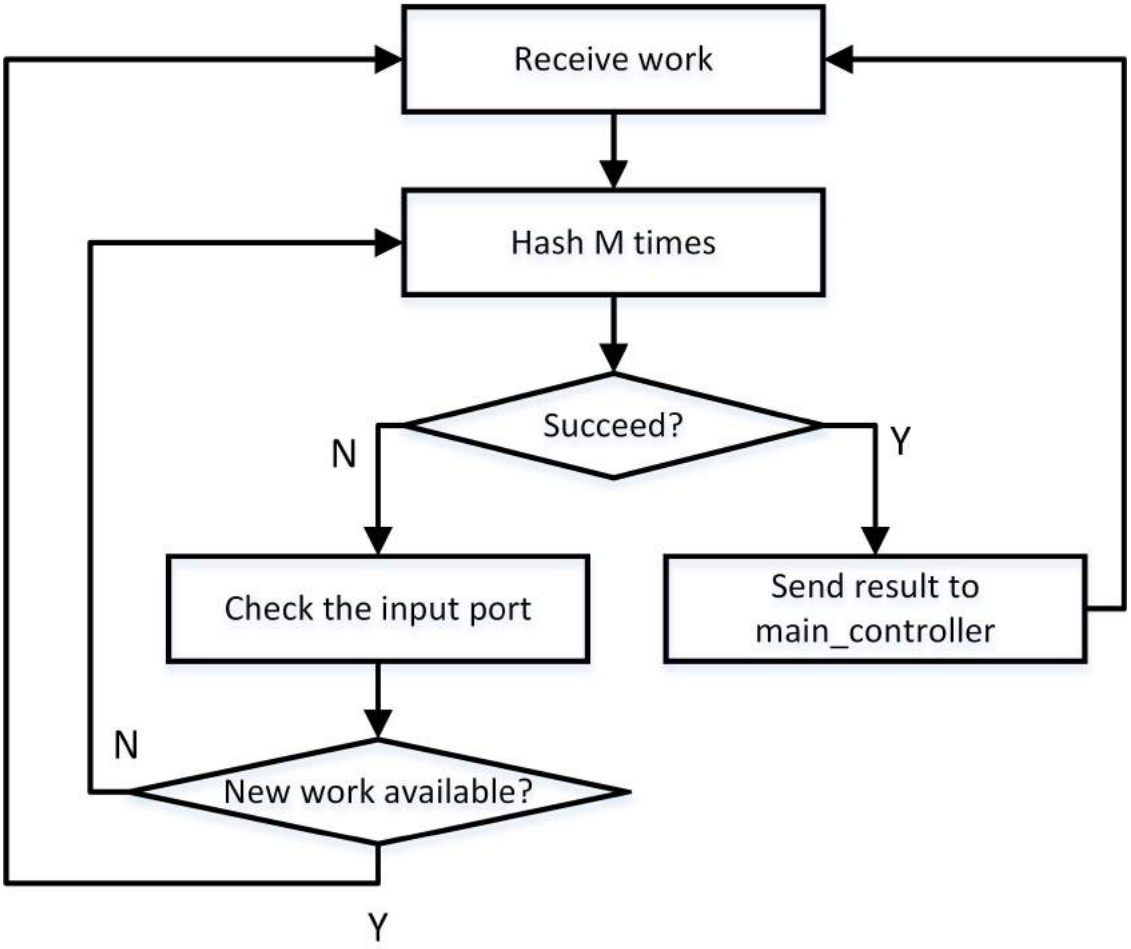

Figure 3.8: Optimized scanner for parallel Bitcoin miner

Chapter 4

Data-level parallel Bitcoin miner model

In this chapter, the data-level parallelism (DLP) is applied to the parallel Bitcoin miner model to further improve the performance.

4.1 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, SIMD technique is exploited to execute multiple scanning lanes simultaneously. A comparison between the reference *scanner* module and the SIMD *scanner* module is shown in Figure 4.1. This idea is based on the 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

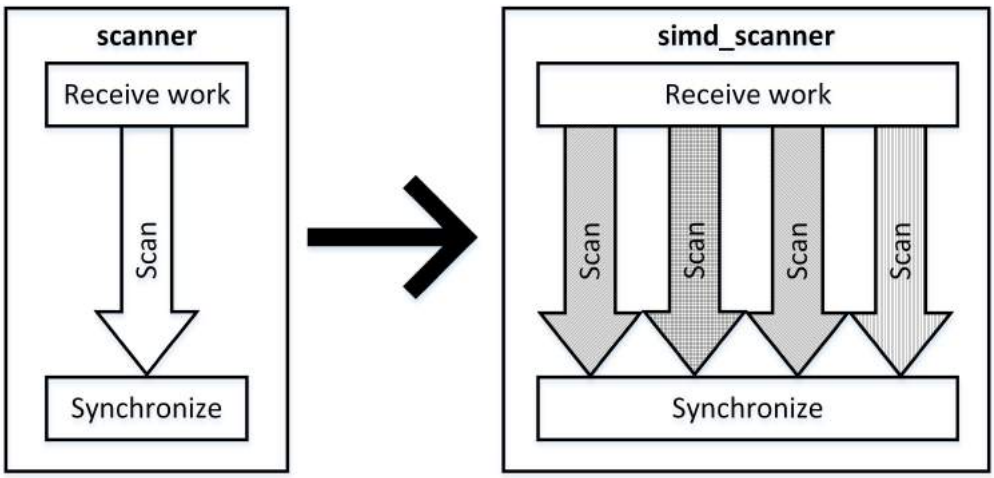

Figure 4.1: Comparison between scalar and SIMD scanners

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 [28].

4.2 Intrinsics and SIMD pragma

A common approach to exploit SIMD on the Intel architecture is to use the Intel intrinsics [14]. The intrinsics are a set of functions known by the Intel compiler that are mapped to a sequence of assembly instructions. With the friendly C/C++ interface to assembly instructions, the use of processor-specific enhancements becomes easier for the software developers [10]. Another advantage is that with the intrinsics, the compiler can manage things that the user would normally have to be concerned with, such as register names, register allocations, and memory locations of data.

However, the conversion of a scalar code into its SIMD counterpart is still quite complicated. Listing 4.1 shows an example code of a *for* loop [13]. In each iteration step, it performs an *add* and a masked *if* operation, which can be considered a simplified version of the scanning loop (by replacing the *hash()* function with an *add* operation). The vectorized function using

intrinsics is shown in listing 4.2.

```
1  #define SIZE 128
2  short int aa[SIZE],bb[SIZE],cc[SIZE],dd[SIZE];
3  void Branch_Loop(short int g)
4  {
5      int i;
6      for(i=0;i<SIZE;i++)
7      {
8          aa[i] = (bb[i]>0)?(cc[i]+2):(dd[i]+g);
9      }
10 }
```

Listing 4.1: Scalar *for* loop [13]

```
1  #define SIZE 128
2  #include <emmintrin.h>
3  #define TOVectorAddress(x) ((__m128i *)&(x))
4
5  void Branch_Loop(short int g)
6  {
7      __m128i a,b,c,d,mask,zero,two,g_broadcast;
8      int i;
9      zero = _mm_set1_epi(16);
10     two  = _mm_set1_epi16(2);
11     g_broadcast = _mm_set1_epi16(g);
12
13     for(i=0;i<SIZE;i+=8)
14     {
15         b = _mm_load_si128(ToVectorAddress(bb[i]));
16         c = _mm_load_si128(ToVectorAddress(cc[i]));
17         d = _mm_load_si128(ToVectorAddress(dd[i]));
18         c = _mm_add_epi16(c,tw0);
```

```
19            d = _mm_add_epi16(d, g_broadcast);
20            mask = _mm_cmpgt_epi16(b, zero);
21
22            a = _mm_and_si128(c, mask);
23            mask = _mm_andnot_epi16(mask, d);
24            a = _mm_or_si128(a, mask);
25           _mm_store_si128(ToVectorAddress(aa[i]), a);
26        }
27    }
```

Listing 4.2: Vectorized *for* loop using intrinsics [13]

This example shows the complexity of using intrinsics. Variables are first loaded explicitly to SIMD registers, and then SIMD instructions are applied to the registers. The results are stored back at the end. The code length grows around 2.5 times compared to the scalar function. Considering the over 500 lines of code in the Bitcoin scanning loop, it would be very time consuming and overwhelming to implement SIMD via the use of intrinsics.

Nowadays, the SIMD pragma extension proposed by Tian et al. in 2012 [37] is becoming more and more popular in C/C++ compilers: it performs automatic loop vectorization and can vectorize functions as well. Listing 4.3 shows the SIMD implementation with SIMD pragma statements for the same function in Listing 4.1. In order to vectorize the loop, the only modification is to place the *#pragma simd* statement in front of the *for* loop. This simple example demonstrates the effectiveness and efficiency of using the SIMD pragma extension for SIMD implementation. Thus, in our approach, instead of using intrinsics to rewrite the scanning loop, we use SIMD pragmas to vectorize the loop and the *hash()* function.

```
1  #define SIZE 128
2  short int aa[SIZE], bb[SIZE], cc[SIZE], dd[SIZE];
3  void Branch_Loop(short int g)
4  {
5  int i;
```

```
6  #pragma simd
7      for(i=0;i<SIZE;i++)
8      {
9          aa[i] = (bb[i]>0)?(cc[i]+2):(dd[i]+g);
10     }
11 }
```

Listing 4.3: Vectorized *for* loop using SIMD pragma

4.3 Design of data-level parallel scanner

In the SIMD *scanner* design, we use SIMD pragma 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, as shown from line 5 to line 7 in listing 4.5.

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. Listing 4.4 shows the scalar scanning loop. According to the restriction for vectorization of loops described in Section 1.5.2, the loop

should not contain any *break* statements, and should not modify the same variable (*nonce*, *result*, *found*) on different steps of 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. As shown in listing 4.5, now the operations performed in every iteration steps are the same, and the loop can be easily vectorized by *#paragma simd*. 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.

```
1 for(int i=0;i<LOOP_LENGTH;i++){
2     nonce=nonce+i;
3     result=hash(scan_work.header,nonce);
4     if(isSmaller(result,scan_work.target)){
5         found=true;
6         break;
7     }
8 }
9 synchronize(result,found);
```

Listing 4.4: Scalar scanning loop

```
1 #pragma simd
2 for(int i=0;i<LOOP_LENGTH;i++){
3     unsigned int* result;
4     n=nonce+i;
5 /* inlined hash function start */
6   ...557 lines of code here...
7 /* inlined hash function end   */
8     if(isSmaller(result,scan_work.target)){
```

```
9             found_array[i]=true;
10        }
11    }
12    nonce=nonce+LOOP_LENGTH;
13    for(int i=0;i<LOOP_LENGTH;i++){
14        if(found_array[i]) found=true;
15    }
16    unsigned int* result=hash(scan_work.header,i);
17    synchronize(result,found);
```

Listing 4.5: Vectorized scanning loop

Chapter 5

Evaluation

In this chapter, 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.

5.1 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 [33], and then compiled with Intel® C++ compiler (ICPC) [3], 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, the elapsed time.

Our experiments are conducted on idle host processors with CPU frequency scaling turned off. File I/Os (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 5.1.

Table 5.1: 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

5.2 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 5.2. 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 [1]. 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 dis-

cussed 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.

Table 5.2: Processor specifications

	omicron	phi	mic0
processor type	E3-1240	E5-2680	Xeon Phi^{TM} 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

5.3 Experiments on 4-core host

Table 5.3 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. On 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.

The DLP speedup S_{DLP} is stable and approximately 2.96. This value is smaller than the naively expected maximum value 4 due to 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 communications. 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 5.3, 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 5.1, 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 5.2.

Finally, by combining TLP and DLP together, we get a 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 [34], which shows that the combination of thread and data-level parallelism can be very efficient to accelerate the SystemC simulation.

Table 5.3: 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

Figure 5.1 gives an graphical overview of the three speedups. It can be seen that S_{DLP}

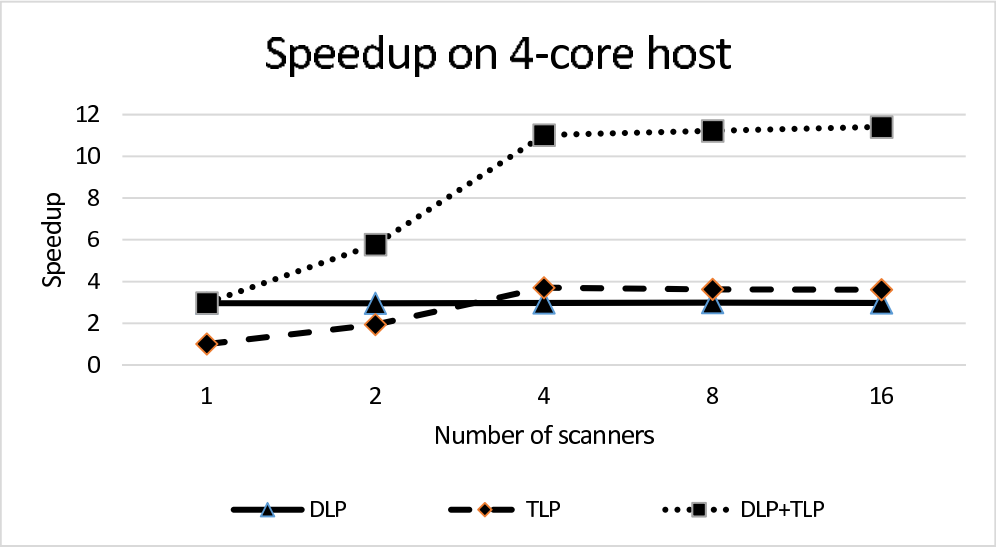

Figure 5.1: Speedup on 4-core host with 4 SIMD lanes

is constant all the time, and S_{TLP} and $S_{DLP+TLP}$ first increase and then become almost constant when reaching the upper limit of #physical cores. From this figure, it is obvious that combining DLP and TLP together results in a very large improvement on the speedup.

5.4 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 5.4.

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 should be because of the different CPU architectures of the two host machines [7][8]. 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 5.2, the value of S_{TLP} at $N = 16$ is lower than expected. From Table 5.4 we can see that it has a value of 11.82, which is only 73.85% of the upper limit. We suspect that 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 [30]. With the increase of communication overhead, speedup will decrease. For the same reason, $S_{DLP+TLP}$ is also lower than expected.

Table 5.4: 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

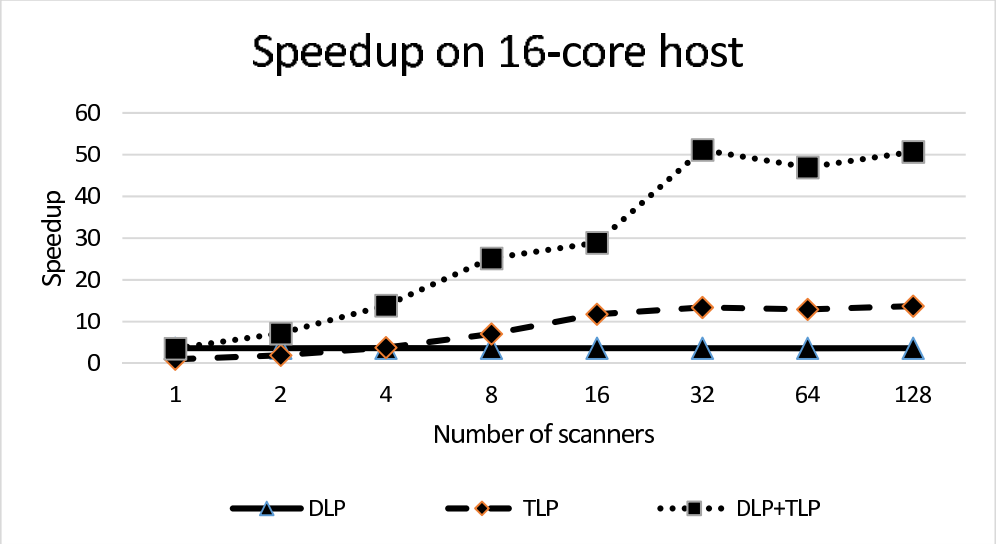

Figure 5.2: Speedup on 16-core host with 4 SIMD lanes

5.5 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 5.5. 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-liner speedup [26]. This happens when the working set of a problem is greater than the cache size when executed sequentially, but can fit nicely in each available cache when executed in parallel [12].

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 [30]. 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.

Table 5.5: 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

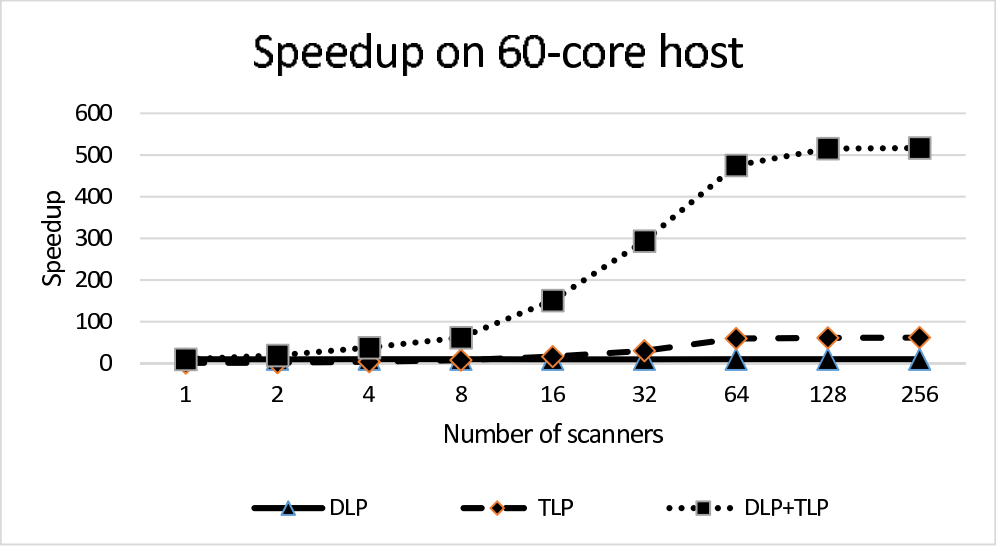

Figure 5.3: Speedup on 60-core host with 16 SIMD lanes

Chapter 6

Conclusion

In this thesis, 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 high 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 published in [34].
In summary, we made the following contributions in this thesis:

- We have changed the reference C++ Bitcoin miner project into an appropriate SystemC model, which can serve as a very good test bench for evaluating parallel SystemC models.
- We have evaluated the performance of thread-level parallelism in the context of the

RISC simulator. Our results show that the speedup of the RISC simulator compared to the reference sequential simulator was proportional to the number of simulation threads. This confirms that RISC is an effective framework for parallel SystemC simulation.

- We have exploited data level parallelism on top of thread level parallelism for fast SystemC simulation. SIMD pragmas are used 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 [34], respectively. This finding is of help to the performance optimization of SystemC simulation.

- We have analyzed the scalability of the parallel SystemC simulation on a many-core Xeon Phi™ Coprocessor. A speedup of over 510x is attained. The results demonstrate that simulation of Bitcoin miner using the RISC simulator scales well on the Xeon Phi™ Coprocessor.

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.

Bibliography

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

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

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

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

[5] Channels In SystemC Part IV. http://www.asic-world.com/systemc/channels4.html.

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

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

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

[9] Introduction to Xeon Phi™ family. http://www.intel.com/content/www/us/en/products/processors/xeon-phi.html.

[10] MMX, SSE, and SSE2 Intrinsics. https://msdn.microsoft.com/zh-tw/library/y0dh78ez(v=vs.100).aspx.

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

[12] Superlinear Speedup in Parallel Computation. https://pdfs.semanticscholar.org/d4db/ecc7c35e15a4b2f1238373f6485272adc3ef.pdff.

[13] The Intel® Intrinsics Example, howpublished = http://blog.chinaunix.net/uid-20385936-id-3902720.html.

[14] The Intel® Intrinsics Guide. https://software.intel.com/sites/landingpage/IntrinsicsGuide/.

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

[16] IEEE Standard for Standard SystemC Language Reference Manual. *IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005)*, pages 1–638, 2012.

[17] Intel® Corporation - Requirements for Vectorizable Loops. `https:software.intel.com/en-us/articles/requirements-for-vectorizable-loops`, 2016.

[18] 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.

[19] 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.

[20] W. Chen, X. Han, and R. Dömer. Multicore Simulation of Transaction-Level Models Using the SoC Environment. *IEEE Design Test of Computers*, 28(3):20–31, May 2011.

[21] 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.

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

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

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

[25] Tim Schmidt Guantao Liu and Rainer Dömer. RISC Compiler and Simulator, Beta Release V0.3.0:Out-of-Order Parallel Simulatable SystemC Subset. Technical Report CECS-16-06, September 2016.

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

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

[28] Ralf Karrenberg. *Automatic SIMD Vectorization of SSA-based Control Flow Graphs.* 2015.

[29] A. Keliris and M. Maniatakos. Investigating large integer arithmetic on Intel® Xeon Phi™ SIMD extensions. In *2014 9th IEEE International Conference on Design Technology of Integrated Systems in Nanoscale Era (DTIS)*, pages 1–6, 2014.

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

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

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

[33] T. Schmidt R Dömer, G. Liu. RISC v0.3.0 compiler. `http://www.cecs.uci.edu/~doemer/risc.html#RISC030`, 2016.

[34] T. Schmidt and R. Dömer. Exploiting Thread and Data Level Parallelism for Ultimate Parallel SystemC Simulation. In *2017 Design Automation Conference*, To be published in Austin, TX, 2017.

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

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

[37] X. Tian, H. Saito, M. Girkar, S. V. Preis, S. S. Kozhukhov, A. G. Cherkasov, C. Nelson, N. Panchenko, and R. Geva. Compiling C/C++ SIMD Extensions for Function and Loop Vectorizaion on Multicore-SIMD Processors. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops PhD Forum*, pages 2349–2358, 2012.

[38] X. Tian, H. Saito, S. V. Preis, E. N. Garcia, S. S. Kozhukhov, M. Masten, A. G. Cherkasov, and N. Panchenko. Practical SIMD Vectorization Techniques for Intel® Xeon Phi™ Coprocessors. In *2013 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum*, pages 1149–1158, 2013.

[39] 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.

[40] J. Virtanen, P. Sjvall, M. Viitanen, T. D. Hmlinen, and J. Vanne. Distributed SystemC simulation on manycore servers. In *2016 IEEE Nordic Circuits and Systems Conference (NORCAS)*, pages 1–6, 2016.