symmetry

MDPI

Article

Qinling: A Parametric Model in Speculative Multithreading

Yuxiang Li [1,†], Yinliang Zhao [1,*,†] and Bin Liu [2]

[1] School of Electronic and Information Engineering, Xi'an Jiaotong University, No 28, Xianning West Road, Xi'an 710049, China; liyuxiang19841203@163.com

[2] College of Information Engineering, NorthWest Agriculture and Forestry University, No 22, Xinong Road, Yangling 712100, China; liubin0929@nwsuaf.edu.cn

* Correspondence: zhaoy@mail.xjtu.edu.cn; Tel.: +86-130-7297-2287

† These authors contributed equally to this work.

Received: 22 July 2017; Accepted: 28 August 2017; Published: 2 September 2017

Abstract: Speculative multithreading (SpMT) is a thread-level automatic parallelization technique that can accelerate sequential programs, especially for irregular applications that are hard to be parallelized by conventional approaches. Thread partition plays a critical role in SpMT. Conventional machine learning-based thread partition approaches applied machine learning to offline guide partition, but could not explicitly explore the law between partition and performance. In this paper, we build a parametric model (*Qinling*) with a multiple regression method to discover the inherent law between thread partition and performance. The paper firstly extracts unpredictable parameters that determine the performance of thread partition in SpMT; secondly, we build a parametric model *Qinling* with extracted parameters and speedups, and train *Qinling* offline, as well as apply it to predict the theoretical speedups of unseen applications. Finally, validation is done. Prophet, which consists of an automatic parallelization compiler and a multi-core simulator, is used to obtain real speedups of the input programs. Olden and SPEC2000 benchmarks are used to train and validate the parametric model. Experiments show that *Qinling* delivers a good performance to predict speedups of unseen programs, and provides feedback guidance for Prophet to obtain the optimal partition parameters.

Keywords: parametric model; speculative multithreading; Prophet

1. Introduction

The emergence of the speculative multithreading (SpMT) model [1–6] in the last decade has provided significant breakthrough in non-numeric applications. Exploring the partition law in SpMT, however, is challenging due to the complexity of influence parameters. Program parallelization primarily includes two methods: compiler-based automatic parallelization and machine learning-based parallelization. Compiler-based automatic parallelization is a widely studied area and can potentially deliver significant speedups for sequential programs. The studies [7–10] focused on loops, and they decomposed loops into multiple code segments to achieve the performance improvement. These research works [2,11] partitioned the whole program into multiple threads to be executed in parallel.

Machine learning technology has been successfully introduced to SpMT, for program parallelism [5,12–17]. Wang et al. [15] developed an automatic compiler-based approach to map a parallelized program to multi-core processors using machine learning. Long et al. [13] presented a machine learning-based approach to parallel workload allocation in a cost-aware manner. Chen et al. [17] presented an adaptive open multiprocessing(OpenMP)-based mechanism capable of generating a reasonable number of representative multithreaded versions for a given loop, and selecting a suitable version at runtime to execute on a multicore architecture using machine learning.

Li et al. [6] used an artificial immune algorithm to obtain the optimal thread partition scheme. Liu et al. [18] used virtual sample generation and a K-nearest neighbor(KNN) algorithm to realize thread partition. Machine learning has recently also been investigated by a number of researchers in the area of compiler optimization. Much of the prior work in machine-learning based compilation relied on program feature-based characterization. For instance, Monsifrot et al. [19], Stephenson et al. [20], and Agakov et al. [21] all used static loop nest features. Cavazos et al. [22] considered a reaction-based scheme that used the sequence of transformations was applied to a program as an input to a learnt model. Wang et al. [5,23] together used dynamic features and machine learning method to exploit probably parallel legacy code.

Moreover, regression or statistics-based methods have also received much attention [24–27]. Lee et al. [26] applied a regression modeling to derive simulation-free statistical inference models, in order to reduce the number of required simulations. Cavazos et al. [28] developed a logistic regression technique that automatically selected the best set of optimizations for different sections of a program. Khan and Luk et al. [24,25] used a statistical machine learning and a fully automatic method, respectively, to map potential parallelisms onto threads in the context of SpMT.

In both cases, the benefits of statistical regression are highlighted in the following two aspects. On the one hand, the selection of best set of optimization and the mapping of potential parallelisms onto threads are all automatically completed; on the other hand, statistical regression is effective to learn and apply thread partition regular.

The paper develops a parametric model, namely multiple regression model(*Qinling*) to explore the inherent law between the influencing factors during thread partition and performance. We develop this model on Prophet [29,30] and automatically predict speedups according to partition parameter values of unseen programs. This is achieved by training *Qinling* offline on a set of training data, which then automatically learns inherent law.

2. Definitions and Motivations

2.1. Definitions

In this section, we present several definitions, so that abbreviations of them can be well understood.

Thread partition is a process, in which sequential programs are divided into many segments that are mapped to many processing elements to run. **SP** is abbreviation of spawning point, which is used to spawn a child thread. **CQIP** is abbreviation of control quasi-independent point, which is used to validate successor thread.

Spawning distance (SD) is the dynamic instruction count from spawning point to control quasi-independent point. SD represents the time difference between execution of predecessor thread and its successor thread.

P-slice is abbreviation of pre-computation slice, which is a simplified version of dependent instructions between predecessor and successor threads.

Thread granularity is the size of thread, which is generated by partitioning sequential programs.

2.2. Motivations

The purpose of multiple regression model is to exploit the inherent regular between influencing factors and speedups. Thus, finding the primary influencing factors during thread partition becomes the first issue to be handled.

2.2.1. Motivation from Partition Algorithms

In this paper, we refer to two partition algorithms: Algorithm 1 and Algorithm 2 [31] and time overhead analysis graph (in Figure 1) to describe the process of extracting influencing factors. Algorithm 1 gives the description of loop partition. While partitioning a function, the loop regions are first identified and partitioned. The profiling information about the number of iterations and the loop

body size are considered together to decide the partition of loop region. Data dependence count of successive iterations of the loop is also checked. Only when the thread spawning for the next iteration is profitable, then the thread is spawned. Between the 6_{th} to 10_{th} line in Algorithm 1, for loop with proper *granularity* and*data dependence count* of inter-iteration is small, each iteration is specified as a candidate thread. For small loops, they will not be parallelized otherwise the overhead of spawning thread offsets performance improvement by SpMT; instead, the loop is unrolled to increase parallelism. The pseudo code for non-loop partitioning is shown in Algorithm 2. This function "partition_thread" partitions the sequential code segments between two basic blocks into multiple threads by calling itself recursively. In Algorithm 2, "curr_thread" represents subgraph of the current candidate thread, and it consists of all the basic blocks between exit node of previous thread and "start_block". If "curr_thread" is NULL, it means that start_block is the exit node of previous thread, or "curr_thread" can not be the candidate thread, as "curr_thread" is too small or there are too much data dependence. "pdom_block", which acts as the control-independent point of current basic block, is the postdominator of "start_block". The "likely_path" is the most likely path between "start_block" and "pdom_block". The next function "find_optimal_dependence" keeps the optimal **data dependence counts** between current thread and the future thread below DEP_THRESHOLD. In order to get the best speedup at runtime, the lower and the upper limit of **thread granularity** should be limited to balance **thread granularity**. From the 7_{th} and 25_{th} line in Algorithm 2, the "curr_thread" whose **granularity** is within the limits and whose **dependenc**e with the successor thread is less than DEP_THRESHOLD can be partitioned and generates a new thread. If the **granularity** of "curr_thread" is too large, then the subgraph between "start_block" and its control-independent basic block will be further partitioned for potential candidate threads. Furthermore, if the "curr_thread" is too small even when including the basic blocks along the most likely path, then no new candidate thread will be created at the control-independent point, the "future_thread" will be simply added to "curr_thread".

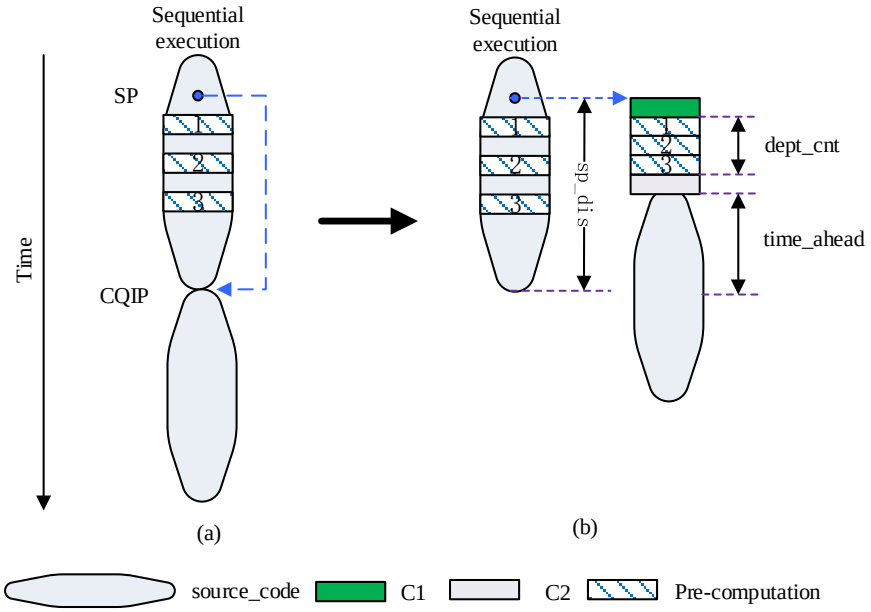

Figure 1. Time overhead analysis in speculative multithreading.

Algorithm 1 Loop partition.

Input: Loop L

Output: *curr_thread*

partition_loop(loop L){
1 *start_block* := entry block of loop *L*;
2 *end_block* := exit block of loop *L*;
3 *likely_path* := the most likely path from *start_block* to *end_block*;
4 *opt_ddc* := find_optimal_dependence(*start_block*, *end_block*, *likely_path*, &*spawn_pos*);
5 *loop_size* := the number of dynamic instructions in loop *L*;
6 if(*loop_size*<=**THREAD_LOWER_LIMIT**)
7 unroll(loop *L*);
8 else(*opt_ddc*<**DEP_THRESHOLD**)
9 *curr_thread* := create_new_thread(*end_block*, *spawn_pos*, *likely_path*);
10 end if
11 return *curr_thread*;}

Algorithm 2 Non-loop partition.

Input: *start_block*, *end_block*, *curr_thread*(candidate thread)

Output: *curr_thread*

partition_thread(*start_block*, *end_block*, *curr_thread*){
1 if(*start_block*==*end_block*) then
2 return *curr_thread*;
3 end if
4 *pdom_block*:= the nearest post dominator block of *start_block*;
5 *likely_path*:= the most likely path from *start_block* to *pdom_block*;
6 *opt_ddc*:=find_optimal_dependence(*pdom_block*, *curr_thread*, &*spawn_pos*);
7 **if(is_medium(*thread_size*) && *opt_ddc*<DEP_THRESHOLD) then**
8 *thread_size* := *curr_thread* + sizeof (path);
9 finish_construction(*curr_thread*);
10 *curr_thread* := create_new_thread(*pdom_block*, *spawn_pos*, *likely_path*);
11 *curr_thread* = partition_thread(*pdom_block*, *end_block*, *curr_thread*);
12 **else if(is_big(*thread_size*)) then**
13 *thread_size* := *curr_thread* + *path.first_block*;
14 opt_ddc := find_optimal_dependence(path.*first_block*, *curr_thread*, null,&*spawn_pos*);
15 **If(!is_small(*thread_size*) && *opt_ddc*< DEP_THRESHOLD) then**
16 *curr_thread* := *curr_thread*+path.*first_block*;
17 finish_construction(*curr_thread*);
18 *curr_thread* := create_new_thread(path.*first_block*,*spawn_pos*, *likely_path*);
19 *curr_thread* = partition_thread(path.*first_block*, *end_block*,*curr_thread*);
20 else
21 *curr_thread* := *curr_thread* + *path*;
22 *curr_thread* := *curr_thread* + *pdom_block*;
23 *curr_thread* := partition_thread(*pdom_block*, *end_block*, *curr_thread*);
24 end if
25 end if
26 return *curr_thread*;}

2.2.2. Motivation from *Time_Ahead*

Besides Algorithm 1 and Algorithm 2, we give a *time_ahead* analysis in Speculative Multithreading to introduce influencing factors. Shown in Figure 1, the length of precomputation-slice(*p-slice*) is represented with variable *p-slice*, spawning distance from predecessor thread to successor thread is *sp_dis*, the correlative instruction count along spawning path is *dep_cnt*, so

$$pslice = dep\_cnt + C, \tag{1}$$

where C represents the overhead to construct *pslice*. Thus, the reduced time (indicated by *time_ahead*) for speculative execution is shown in formula (2):

$$\begin{aligned} time\_ahead &= sp\_dis - pslice \\ &= sp\_dis - dep\_cnt - C, \end{aligned} \tag{2}$$

where *dep_cnt* is determined by *dependence count*, and C is also affected by many other factors. The whole *time_ahead* is determined by spawning distance and dependence count.

2.2.3. Determination of Influencing Factors

During the process of loop partition and nonloop partition, bold words in the above paragraphs as well as the bold statements in Algorithm 1 and Algorithm 2 suggest that three factors, including thread granularity, data dependence count, and spawning distance are the primary influencing factors during partition. In the process of *time_ahead* analysis, the *time_ahead* is mainly influenced by spawning distance and dependence count. We give a set of influencing factors, containing three factors: *spawning distance*, *dependence count*, and *thread granularity*.

In terms with Sections 2.2.1 and 2.2.2, we come to make a conclusion and get the independent variables and dependent variables in Table 1.

3. Speculative Multithreading

3.1. SpMT Execution Model

Speculative multithreading technique [6,32] is actually an aggressive program execution, and multiple code segments are executed in parallel simultaneously on multi-core to improve the speedups of sequential programs. In SpMT execution model, sequential programs are partitioned into multiple speculative threads; furthermore, each of the speculative threads executes a different part of the sequential program. There is a special thread called a non-speculative thread among concurrently executed threads. It is the only one allowed to commit its results to memory, while the other threads are speculative. A speculative thread is marked by a spawning instruction pair. When a spawning instruction is found during program execution and if the existing processor resources allow spawning, a parent thread will spawn a new speculative thread.

When the execution of the non-speculative thread is completed, it will verify its successor thread. If the validation is correct, the non-speculative thread will commit all the values, which the successor thread generates to memory and then the successor thread will become non-speculative. Otherwise, the non-speculative thread will revoke all speculative child threads and re-execute its successor threads.

On Prophet [29,30], a spawning instruction pair are composed of a Spawning Point (SP) and a Control-Quasi Independent Point (CQIP). The SP defined in parent thread can spawn a new thread to execute speculatively the code segment behind the CQIP during program execution. Thread-level speculative model is shown in Figure 2. A sequential program is mapped to a SP-CQIP, and the speculative multithreading program becomes a sequential program as shown in Figure 2a.

When an SP is found on program execution, the parent thread will spawn a new speculative thread and execute the code segment speculatively behind the CQIP, as shown in Figure 2(b).

Validation failure or Read-after-Write (RAW) violations will lead to fail. When validation fails, predecessor thread executes the speculative thread in a sequential manner as shown in Figure 2(c). When there is a violation in RAW dependence as shown in Figure 2(d), the speculative thread restarts itself on the current state.

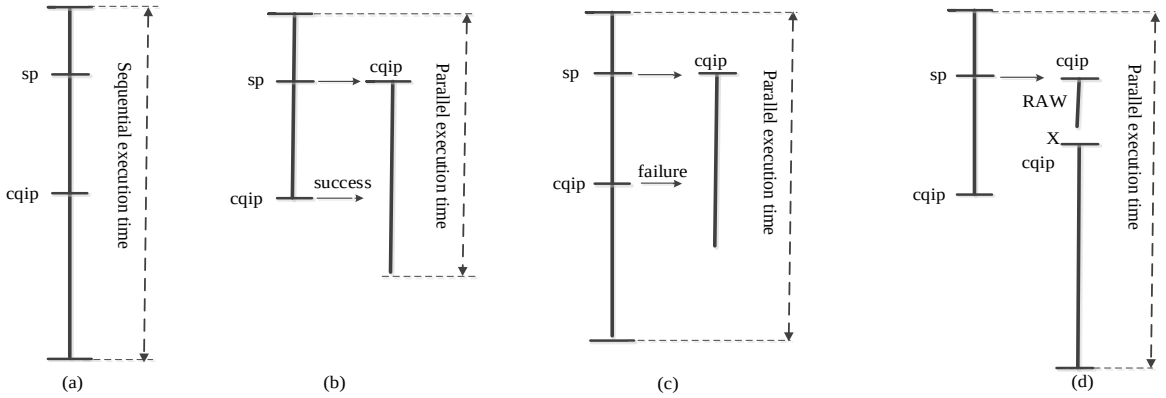

Figure 2. Thread-level speculative model (**a**) sequential execution; (**b**) parallel execution; (**c**) failed parallel execution; (**d**) RAW.

3.2. Pre-Computation Slices

In SpMT, the key is how to deal with inter-thread data dependences. Synchronization mechanism and value prediction have been applied so far. The synchronization approach imposes a high overhead when dependences are frequent and seriously affect the parallel performance. Value prediction has more potential if the values computed by one thread and consumed by another can be predicted. The consumer thread can be executed in parallel with the producer thread since these values are only needed for validation at later stages. On the Prophet compiler [29], in order to reduce inter-thread dependences, the speculative p-slices [1] are constructed and inserted at the beginning of each speculative thread. P-slices are used to calculate the live-ins (dependent variables that are generated by predecessor thread and consumed by a successor thread) of the new speculative thread, but they do not need to guarantee their correctness, since the underlying architecture can detect and recover from mis-speculations. The p-slices are extracted from the producer thread at compile time but triggered at run-time to pre-fetch the live-ins. The steps to build the p-slices for a given spawning pair are: (1) identifying the live-ins produced on the speculative path; and (2) generating the optimal p-slices.

3.3. Data Dependence Calculation

Data dependence [32] includes data dependence count (*DDC*) and data dependence distance (*DDD*). *DDC* is the weighted count of the number of data dependence arcs coming into a basic block from other blocks, while *DDD* between two basic blocks B1 and B2 models the maximum time that the instructions in block B2 will stall for instructions in B1 to complete, if B1 and B2 are executed in parallel. *DDC* and *DDD* are, respectively, achieved in formula (3) and formula (4). Among *DDC* and *DDD*, we select *DDC* as the counted dependence criteria. *DDC* models the extent of data dependence that this block has on other blocks. In Figure 3, we give a description of data dependence between two blocks. The values of x, y in B3 rely on the ones from B1 and B2. The dotted lines represent the dependences. If the dependence count is small, then this block is more or less data independent from other blocks and we can start a thread at the beginning of that basic block. While counting the data dependence arcs, the compiler gives more weights to the arcs coming from blocks that belong to threads that are closer to the block under consideration. The motivation is that dependences from

distant threads are likely to be resolved earlier and hence the current thread is less likely to wait for data generated there:

$$\begin{aligned} &Dependence\_Count(T) = \sum(W_n * A_n); \\ &A_n \textit{ is dependence edges } from \textit{ } T_n \textit{ to } T; \end{aligned} \tag{3}$$

$$\begin{aligned} &Y = MAX(dependence\_distance(A_i)); \\ &Dependence\_Distance(T) = Y. \end{aligned} \tag{4}$$

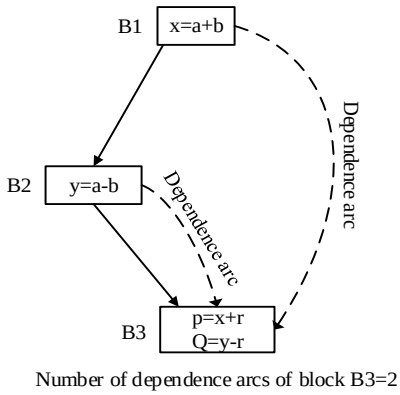

Figure 3. Data dependence arcs between basic blocks.

Furthermore, the compiler gives less weightage to the data dependence arcs coming from the less likely paths. The rationale behind using the data dependence count are twofold: firstly, it is simple to compute; secondly, if the processing elements do out of order execution then the data dependence distant model may not be very accurate because it assumes serial execution within each thread. However, in practice, due to out of order execution, instructions that are lower in the program order can be executed before the earlier instructions inside the threads. Thus, data dependence count tries to model the extent of data dependence in the presence of out of order execution.

4. Model Building

4.1. Deployment

We use a multiple regression method to build a parametric model, the specific steps are concluded as follows:

- Determination of influence factors;
- Setting of index variables;
- Gathering of statistical data;
- Determine the mathematical form of theoretical regression model;
- Estimation of model's parameters;
- Validation and modification of model;
- Application of model.

4.2. Heuristic rules

SP can be anywhere in programs and as far as possible behind function call instruction. CQIP is at the entrance of basic block in non-loop region. In loop region, CQIP is located in front of the loop branch instruction in the last basic block of the loop. SP-CQIPs are located in the same function or in the same loop. The number of dynamic instructions between SP and CQIP must be greater than the

lower limit of thread granularity (**LLoTG**) and less than the upper limit of thread granularity(**ULoTG**). Spawning distance between candidate threads must be greater than the lower limit of spawning distance(**LLoSD**) and is less than the upper limit of spawning distance(**ULoSD**). Data dependence of two consecutive candidate threads must be less than the data dependence count(**DDC**). Function call instructions between SP and CQIP are less than CALL_LOWER.

4.3. Index Variable Setting

As shown in Section 2.2.3, the influence factors are *spawning distance*, *dependence count*, and *thread granularity*. During thread partition, the specific determinant is the values of these factors. In accordance with heuristic rules [6](shown in the section 4.2), we extract the specific variables and determine the final index variables.

Usually, given the execution time of a parallelized program on N cores T_p, and of the original sequential program T_s, the absolute speedup (shown in formula (5)) is defined as T_s/T_p [33]:

$$Speedup = T_s/T_p \times 100\%. \tag{5}$$

Speedup (*Sp*) is a time ratio between the runtime spent for a task to run in a single processing unit and the time cost in processing the same task in p processing units. According to heuristic rules, what affects speedups are five parameters: (*DDC*), *LLoTG*, *ULoTG*, *LLoSD*, *ULoSD*. These five parameters are independent variables, and speedup(*Sp*) is dependent variable, and all of them are listed in Table 1.

Table 1. Set of independent variables and dependent variables.

Influencing Factors	Independent Variables	Dependent Variables
spawning distance, *dependence count*, and *thread granularity*.	*DDC*, *LLoTG*,*ULoTG*, *LLoSD*,*ULoSD*.	*Speedup*

4.4. Gathering of Statistical Data

In order to achieve a credible and truthful regression model, we build the model on the foundation of statistical data. After ensuring the dependent variables and independent variables, we then gather and organize data from a hybrid sample set. Conventionally, the heuristic rules-based (HR-based) sample generation approach is efficient, but it is just one-size-fits-all way, and can not generate the optimal samples for all applications. Then, a hybrid sample generation approach is proposed. With this method, we firstly generate samples which are mips codes, consisting of spawning points (SPs) and control quasi-independent points (CQIPs) by heuristic rules on Prophet [30], and then manually adjust the positions of SPs and CQIPs and rebuild precomputation slice (p-slice) to obtain the best sample set. During the implementation of hybrid sample generation, three mechanisms: bias weighting, preservation of optimal solutions, summary of greedy rules are carried out. In this way, hybrid samples own the optimal partition positions.

We use the manual statistical methods to obtain values of five independent variables and dependent variables from the hybrid sample set.

4.5. Determination of Mathematical Form

Let us then consider an appropriate mathematical form to describe the relation among variables, the conventional method mainly used the scatter diagram to describe the relation between independent variables and dependent variables, to guide the building of regression model. After that, we will give a description of five independent variables: ***A*** (data dependence count), ***B*** (the lower limit of thread granularity), ***C*** (the upper limit of thread granularity), ***D*** (the lower limit of spawning distance), ***E*** (the upper limit of spawning distance), as well as a dependent variable: ***Sp*** (speedup). In order

to determine the final relation between *A*, *B*, *C*, *D*, *E* and *Sp*, we adopt "other-fixed-one-change" mechanism. For example, if we build the relation between *Sp* and *A*, we just change *A* and fix *B*, *C*, *D*, *E*.

Seen from Figure 4 to Figure 8, we can conclude that the distribution of sample points and dependent variable *Sp* have a linear relation. Due to the value precision of variable *C*, speedup has little change in the area of data selection, but we can also see that its relation is essentially linear dependent.

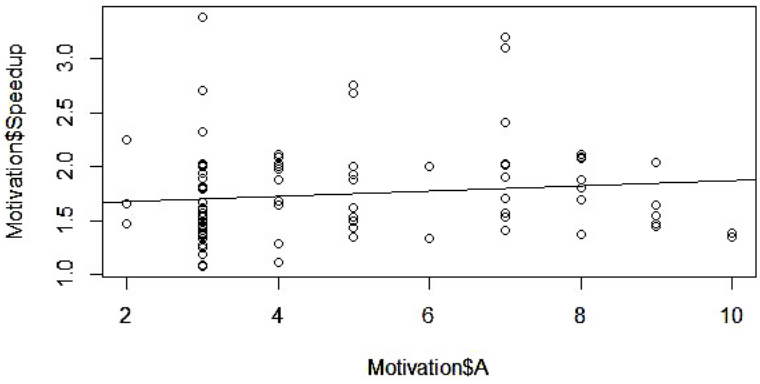

Figure 4. Statistical values between *A* and speedups.

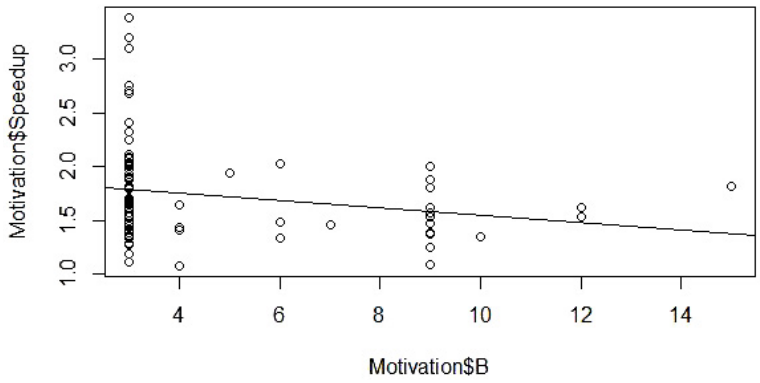

Figure 5. Statistical values between *B* and speedups.

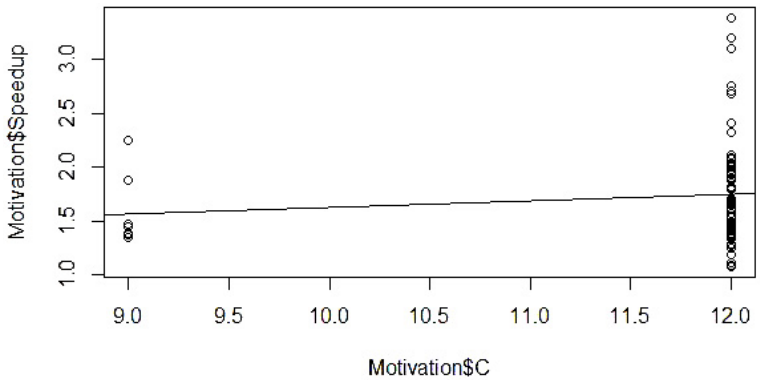

Figure 6. Statistical values between *C* and speedups.

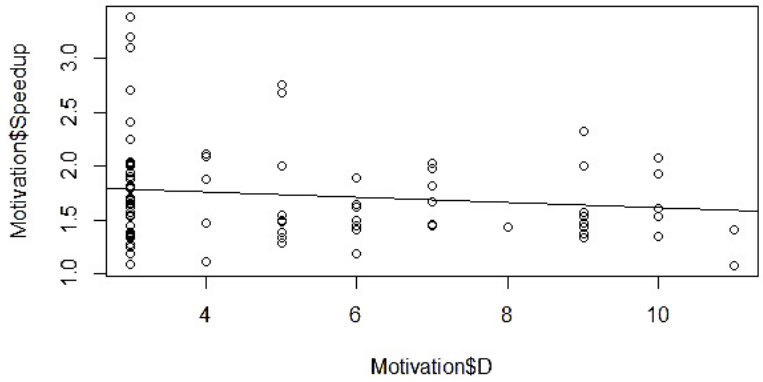

Figure 7. Statistical values between *D* and speedups.

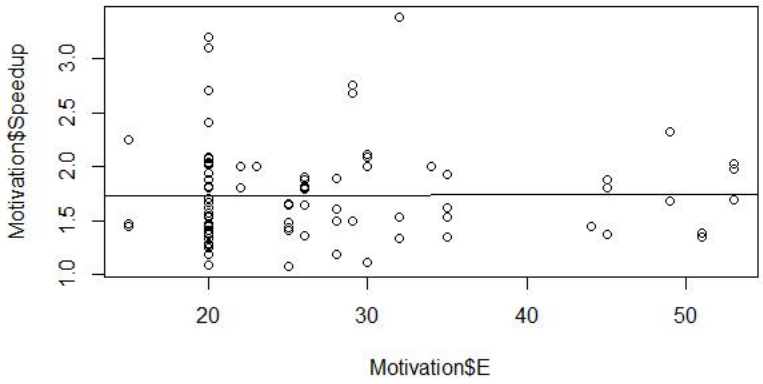

Figure 8. Statistical values between *E* and speedups.

After detecting the relations between every affecting factors and speedup, we then assume a multiple linear correlation model, which is in accordance with the relations. Through model building and validation, we validate the correctness of the model. We assume that the relation between speedup and five factors (described in Section 5.3) can be expressed in formula (6), where *A*,*B*,*C*,*D*,*E* are five influencing factors and β_1, β_2, β_3, β_4, β_5 are coefficients of linear representation:

$$\begin{aligned} &S_p = \beta_0 + \beta_1 A + \beta_2 B + \beta_3 C + \beta_4 D + \beta_5 E + \varepsilon, \\ &A = DDC, B = LToTG, C = ULoTG, D = LLoSD, E = ULoSD, \\ &B <= C, D <= E, \end{aligned} \tag{6}$$

where β_1, β_2, β_3, β_4, β_5 are unknown parameters.

4.6. Model Parameter Estimation

Positional parameters, in multiple regression model, are usually estimated by the least square method. The processing of obtaining the estimated values of parameter β is shown in the formula (7):

$$Q(\beta) = (y - \beta X)^T * (y - \beta X). \tag{7}$$

By getting the minimum value of β, we can ensure the least square estimation of β from formula (8):

$$\beta^{\wedge} = X^T * X_{-1} * X^T * y. \tag{8}$$

In the actual process, we use parallel computer to implement the process of the least square method, and then get the estimation values of model parameters.

4.7. Validation and Modification

After getting the estimated values of unknown parameters, we set up a regression model. Then, we need verify the model to make model$'$s accuracy be proved, and modify the model to be more accurate. Among all the validation methods of regression functions, significance validation is one of the most commonly used methods.

Significance validation method of regression function is listed as formula (9):

$$\begin{aligned} &H_0 : b_0 = b_1 = ...b_p = 0, \\ &H_1 : b_0, b_1, ..., b_p, J = y_0. \end{aligned} \tag{9}$$

When H_0 sets up, statistics magnitude is shown in the following:

$$F = (SS_R/p)/(SS_E/(n-p-1)) \ \sim \ F(p, n-p-1);$$

where,

$$\begin{aligned} &SS_R = \sum_{I=1}^{n} (\hat{y} - \bar{y})^2, SS_E = \sum_{I=1}^{N} (y_i - \hat{y}_i)^2; \\ &\bar{y} = \frac{1}{n} \sum_{i=1}^{n} y_i; \\ &\hat{y}_i = \hat{\beta}_0 + \hat{\beta}_1 x_{i1} + \hat{\beta}_2 x_{i2} + ... + + \hat{\beta}_p x_{ip}. \end{aligned}$$

Usually, we regard SS_R to be regression square sum, and SS_E to be the square sum of residual errors. Once significance α is given, the refusal domain of validation is shown in formula (10):

$$F > F\alpha(p, n-p-1). \tag{10}$$

The results of program show that the obtained regression function is of statistical significance.

4.8. Application of Model

The whole model (shown in Figure 9) is divided into two parts: training stage and application. Once training programs are inputted, we use heuristic rules-based thread partition approaches to partition input programs and figure out the optimal values of partition performance values. Then, we assign the independent variables and dependent variables with profiling values and speedups, which are obtained by heuristic rules-based partition approach. Finally, we start to train our multiple regression model. Once the model is trained, we come to the application stage in which similarity comparison between tested program and the trained one is firstly made, and then we apply the trained regression model to predict the performance of testing programs.

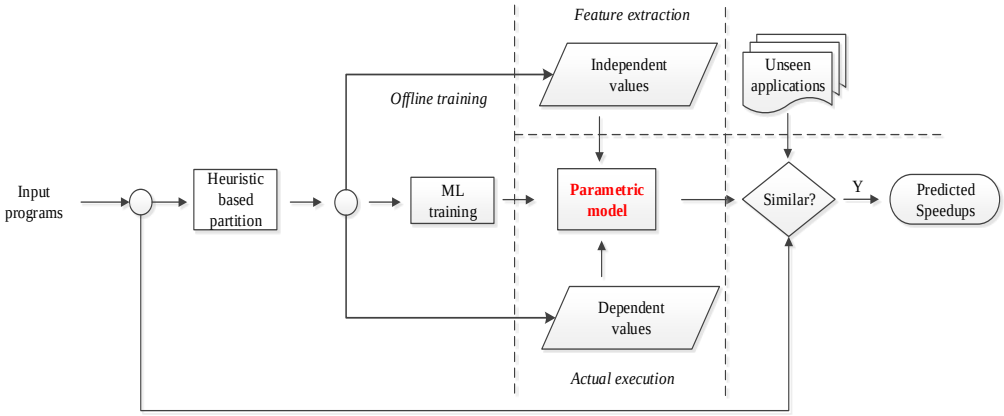

Figure 9. Two stages of model: training and application.

5. Experiment

In this section, we introduce our experimental setup, providing details of the Prophet simulator [29,30] and benchmarks used throughout the evaluation. In the end, we analyze and discuss our results.

5.1. Experiment Configuration

We have implemented the execution model and machine learning(ML)-based thread partitioning algorithms on Prophet [30], which is developed based on SUIF/MACHSUIF [34] and Weka [35]. All the compiler analysis is performed at the high-level intermediate representation (IR) of SUIF. A profiler is implemented to produce profiling information from SUIF-IR in forms of annotations. The profiler interprets and executes SUIF programs and provides information such as control flow, path prediction, data value prediction, the number of dynamic instructions of loops and subroutines. The Prophet simulator [17] models a generic SpMT processor with sixteen pipelined million instructions per second(MIPS)-based R3000 processing elements (PEs). The simulator is an execution-driven simulation and executes binaries generated by Prophet compiler. Each PE has its own program counter, fetch unit, decode unit, and execution unit, and it can fetch and execute instructions from a thread. Each PE can issue up to four instructions per cycle in an in-order fashion. Each PE also has private multiversioned L1 cache with two cycles access latency. Multiversion L1 cache is used to buffer the speculation results for each PE as well as performs cache communication, and the sixteen PEs share a write-back L2 cache via a snoopy bus. Table 2 shows the simulation parameters similar to those listed in a recent publication on Hydra [36]. Figure 10 shows the Prophet framework, and Prophet simulator is the software abstract of implementation scheme based on MIPS processing element in Prophet framework.

Table 2. Prophet simulator configuration.

Configuration Parameter	Value
Fetch, In-order Issue and Commit bandwidth	4 Instructions
Pipeline Stages	Fetch/Issue/Ex/WB/Commit
Architectural Registers	32 int and 32 fp
Function Units	16 int ALU (1 cycle) int Mult/Div (3/12 Cycles) fp ALU (2 Cycles) fp Mult/Div (4/12 Cycles)
L1-Cache(Multiversion)	4-Way Associative 64KB (32B/Block) Hit Latency 2 LRU Replacement

Cont.

Spec. Buffer Size	Fully Associative 2KB (1 Cycle)
L2-Cache(Share)	4-Way Associative 2MB (64B/block) 5 hit latency, 80 cycles(miss) LRU replacement
Spawn Overhead	5 Cycles
Validation Overhead	15 Cycles
Local Register	1 Cycle
Commit Overhead	5 Cycles

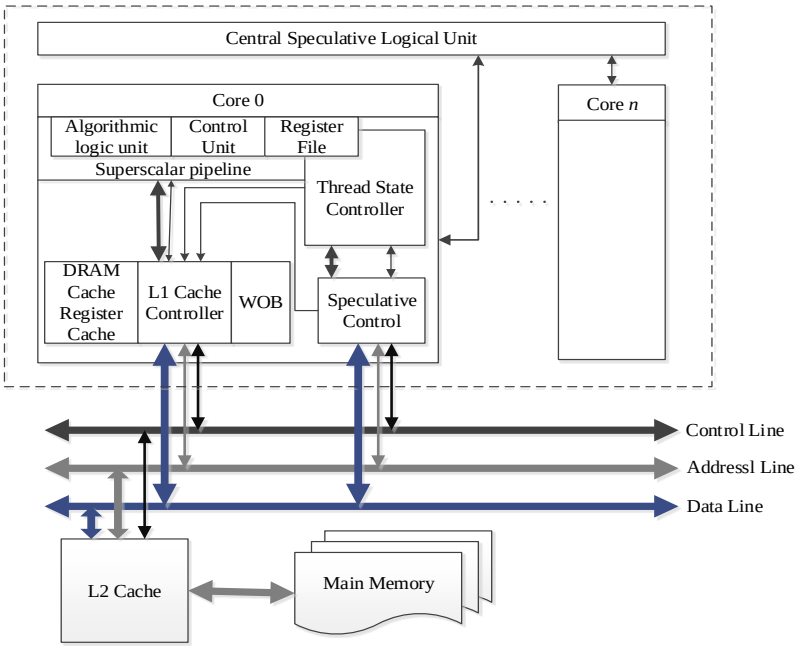

Figure 10. Prophet framework.

In this section, we use Olden benchmarks [37] to evaluate our approach. Olden benchmarks are popular benchmarks for the study of irregular programs, and they process complex control flow and irregular, pointer-intensive data structures. These programs have dynamic structures such as trees, lists and DAGs so that they are hard to be parallelized by the conventional approaches.

5.2. Experiment Assumption

Figure 9 gives the description on how *Qinling* is trained and applied. When a sequential program comes, the program is firstly converted into a SUIF intermediate representation (SUIF IR). The IR programs pass through our developed profiler analysis module. The profiler collects execution statistics such as the number of dynamic instructions of a loop body and subroutine, and the branch probability of each branch instruction. The annotated SUIF IRs are partitioned into multithread programs by the heuristic-based thread partitioner. The MachSUIF [38] back-end and Linker take threaded SUIF IR as input and generate threaded MIPS programs. Then, the MIPS programs are evaluated at simulator to generate speedups.

Before we construct the parametric model *Qinling* which extracts parameter values from partitioner, we assume that some thread overheads are ignored. *Qinling* is trained offline, and applied to predict speedups of unseen applications, and the specific process is shown in the Section 5.4. We use leave-one-out cross-validation to evaluate our approach. This means that we remove the program to be predicted from the training samples and then build a regression model (shown in Figures 9 and

Figure 11), also called prediction model based on the remaining programs. This guarantees that our regression model has not seen the target program before. The prediction model is used to generate speedups for the removed programs. We repeat this process for each program in turn. It is a standard evaluation methodology, providing an estimation for the generalization ability of a regression model for predicting unknown programs. There are several assumptions to make. First, emphasis is not placed on the process of heuristic-based partition. Second, the similarity comparison between training samples and testing samples is directly inferred from other papers. Third, this paper focuses on building and application of a multiple regression model.

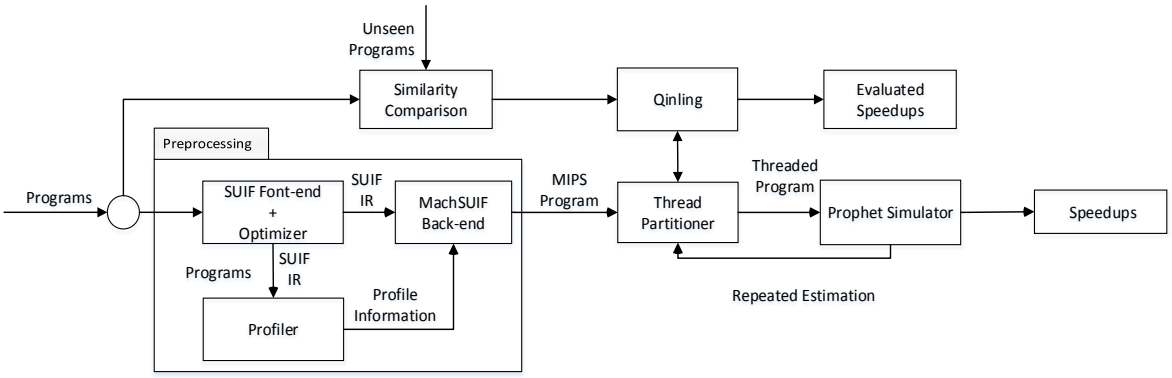

Figure 11. Training and application flow of Qinling.

5.3. Model Building

Table 3 presents an example of extracted data from Olden benchmarks. The 1_{st} column is list of benchmarks, and the 2_{nd} and 3rd column show values of (A, B, C, D, E) in formula (11), and the general speedups. The total item count is 97, which is larger than $2^5 = 32$ (five is the count of independent variables).

Table 3. Extracted data from olden benchmarks.

Benchmark	Item Count	Speedups
voronoi	17	1.92052
treeadd	3	1.33622
power	17	2.08553
perimeter	11	1.26772
mst	12	1.36551
health	13	1.64411
em3d	14	2.27579
bh	10	1.99481

We divide 97 items into five groups, and get their average values, producing five linear equations.

$$\begin{aligned}
388\beta 1 + 433\beta 2 + 1189\beta 3 + 394\beta 4 + 2383\beta 5 &= 154;\\
472\beta 1 + 394\beta 2 + 1183\beta 3 + 416\beta 4 + 2661\beta 5 &= 153;\\
505\beta 1 + 495\beta 2 + 1164\beta 3 + 423\beta 4 + 2522\beta 5 &= 135;\\
471\beta 1 + 424\beta 2 + 1183\beta 3 + 394\beta 4 + 2547\beta 5 &= 179;\\
367\beta 1 + 417\beta 2 + 1194\beta 3 + 411\beta 4 + 2306\beta 5 &= 135;
\end{aligned} \tag{11}$$

where

$$M = \begin{Bmatrix} 388 & 433 & 1189 & 394 & 238 \\ 472 & 394 & 1183 & 416 & 2661 \\ 505 & 495 & 1164 & 423 & 2522 \\ 471 & 424 & 1182 & 394 & 2547 \\ 367 & 417 & 1194 & 411 & 2306 \end{Bmatrix}; \tag{12}$$

$$\beta = [\beta 1, \beta 2, \beta 3, \beta 4, \beta 5]^T; \tag{13}$$

$$Y = [154, 153, 135, 179, 135]^T. \tag{14}$$

Formula (11) can be expressed as formula (15):

$$Y = M * \beta. \tag{15}$$

According to Cramer's Rules [39],

$$\beta_i = \frac{\mid M_i \mid}{\mid M \mid}, i = 1, 2, 3, 4, 5. \tag{16}$$

where M is the determinant of matrix M, and M_i is the replaced matrix, whose i_{th} column is replaced with:

$$\begin{cases} \beta 1 = 0.445; \beta 2 = -0.212; \\ \beta 3 = 0.582; \beta 4 = -1.209; \\ \beta 5 = -0.060. \end{cases} \tag{17}$$

From formula (5), we can deduce the values of $\beta 1$, $\beta 2$, $\beta 3$, $\beta 4$, $\beta 5$ and obtain the final parametric model (shown in the formula (18)):

$$S_p = 0.445A - 0.212B + 0.582C - 1.209D - 0.060E + \xi. \tag{18}$$

5.4. Model Validation

Once we get the multiple linear regression Equation (18), we will apply it to predict speedups. The purpose of our model can be classified into two headings: speedup prediction, and feedback guidance for Prophet.

5.4.1. Speedup Prediction

We select 516 testing samples from Olden benchmark randomly, using the trained model to predict their speedups, and complete a comparison between predicted speedups and real speedups obtained from a simulator (shown in Figure 12). Table 4 shows forty values of *A*–*E*, and real speedups obtained from a simulator. Via *Qinling* (shown in the formula (18)) and values of *A*–*E*, we get the predictive values. Then, we compare predictive speedups with real speedups. Figure 12 shows the comparison results between predictive results and real results. From Figure 12, we can find that there exist gaps between predictive results and real results. The reasons can be classified into two headings: first, the applied parametric model ignores the error (ξ); second, no adequate similarity comparisons between training samples and testing samples are performed. Figure 13 shows the predictive results and real results for part of Standard Performance Evaluation Corporation(SPEC)2000 benchmarks on different cores. Figure 14 presents the speedup comparisons for SPEC2000 between our predictive model and Mitosis [2]. In Figure 14, the red boxes denote the speedups of predictive model.

The next step is to classify input applications according to the similarities among them. The samples in the same class will be applied a fine parametric model, while the samples in a different class use different models.

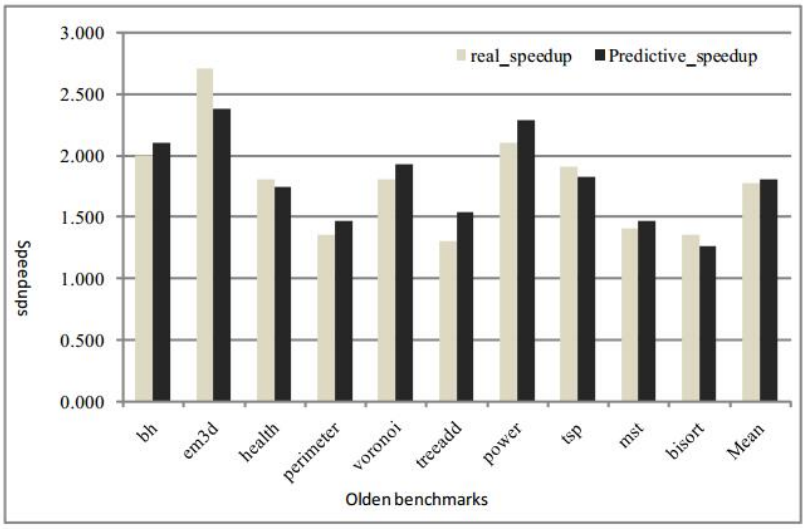

Figure 12. Comparisons between predictive speedups and real speedups.

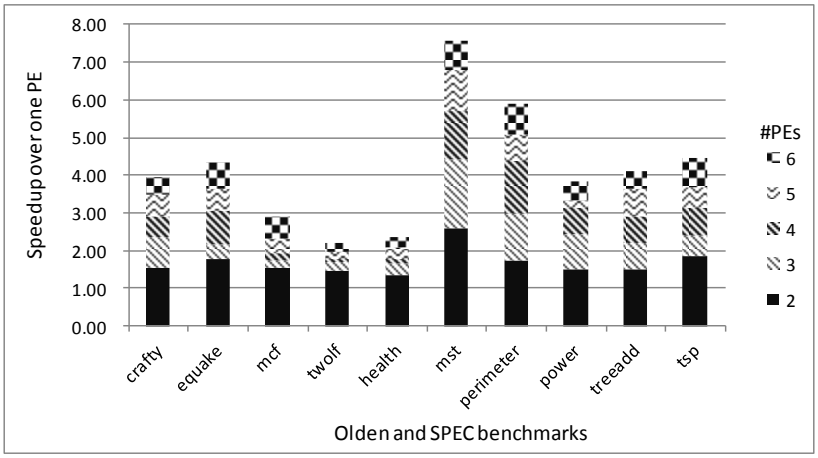

Figure 13. Predictive speedups and real speedups on different cores.

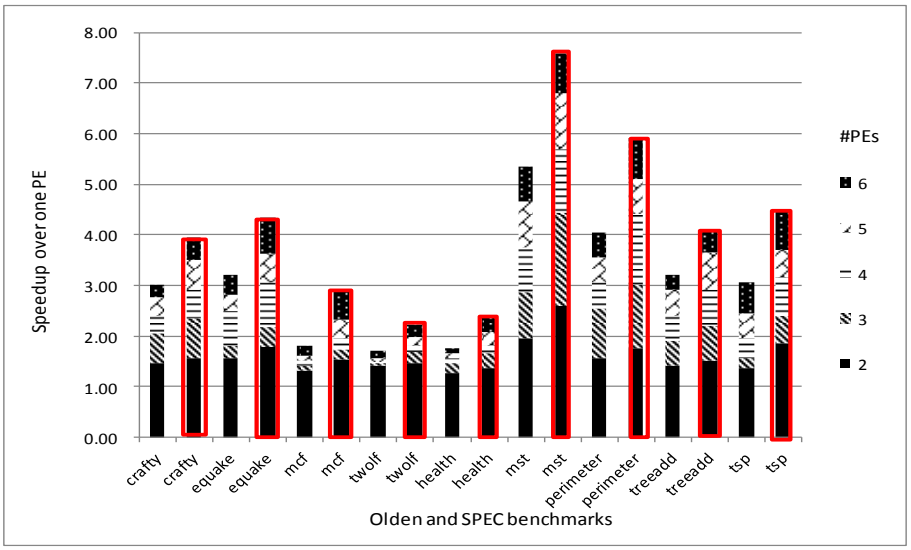

Figure 14. Speedup comparisons between predictive model and mitosis for Olden, SPEC2000 benchmarks.

Table 4. Testing samples (*Sp* is real speedups from simulator).

Testing Samples	Parameter Values					
	A	***B***	***C***	***D***	***E***	***Sp***
compute_nodes	3	4	12	3	24	1.57143
initialize_graph	5	3	12	8	29	1.8
lrand48	3	5	12	3	20	1.8
drand48	3	3	12	4	23	1.25
srand48	3	3	12	3	20	2.59048
init_random	3	3	12	3	20	1.45711
gen_number	3	3	12	3	20	1.34615
gen_signed_number	3	3	12	3	20	1.29032
gen_uniform_double	3	3	12	3	20	1.33333
check_percent	3	3	12	5	24	1.34146
fill_table	5	7	12	5	21	1.48148
make_neighbors	6	5	11	5	38	1.42958
update_from_coeffs	5	8	12	7	26	1.46672
fill_from_fields	5	4	12	3	24	1.25116
dealwithargs	6	3	12	3	25	2.7734
print_graph	3	5	12	3	24	1.26829
dealwithargs	6	4	10	3	23	2.75573
my_rand	4	3	12	6	34	2.3292
generate_patient	5	6	12	6	21	1.57143
put_in_hosp	3	3	12	3	22	1.8
addList	3	5	12	5	25	2.0231
removeList	3	4	12	5	25	1.81421
sim	6	3	12	3	32	2.54631
check_patients_inside	3	5	12	5	23	1.83007
check_patients_assess	3	3	12	3	22	2.70452
check_patients_waiting	7	3	12	3	24	2.01973
get_results	7	3	12	3	31	1.22222
alloc_tree	7	3	12	3	34	1.38029
main	3	3	12	6	21	1.67238

5.4.2. Feedback Guidance for Prophet

Once *Qinling* is built and trained, we can also make a feedback guidance for Prophet, used in reality to partition sequential programs and get speedups via simulator. Within the primary influencing factors, A denotes *DDC*, which is objective and can not be changed, while B, C, D, E are four variables that form a solution space. We regard formula (18) as an objective function. Take the *lrand48* (in Table 4) as an example, and we use parameter model (formula (18)) to obtain the optimal parameter values. Table 5 shows a segment of Matlab code, which is used to search the optimal combination of $<B,C,D,E>$. During the process of searching, $S(p) = -0.212\times B + 0.582\times C - 1.209\times D - 0.060\times E$ is regarded as an objective function.

To do feedback guidance for Prophet, we firstly build a solution space with four dimensions. In the solution space, every point is a possible solution. The scale of solution space is $30^4 = 8.1 \times 10^5$. During the process of traversing all the combination points, there exists a basic restriction, namely $B<C\&\&D<E$. Then, we obtain all the objective values of all possible combination points, and find the maximum as well as its corresponding combination of $< B,C,D,E>$, which are shown in Table 6. Note that the speedup shown in Table 6 is not the final result, as the objective function does not conclude the part 0.445 A, so it is just an intermediate result.

Table 5. Matlab code for feedback guidance.

Segment
Input: Predicted Sp
Output: Optimal values of B,C,D,E
clear; clc; p = 1;T = zeros(810000,4); for B = 1:1:30 for C = 1:1:30 for D = 1:1:30 for E = 1:1:30 if(B<C && D<E) Sp = –0.212*B + 0.582*C – 1.209*D – 0.060*E; T(p,1) = B; T(p,2) = C;T(p,3) = D; T(p,4) = E; p = p + 1; end; end; end; end; end; y = max(S) T(find(y==S(:)),:)

Table 6. Optimal solution.

Speedups	**Optimal Solution**			
	B	***C***	***D***	***E***
15.919	1	30	1	2

With these values, we set partition parameters (A,B,C,D,E) in the partition model of Prophet, so that Prophet can apply the predicted results for loop partition (Algorithm 3) and nonloop partition (Algorithm 4).

Algorithm 3 Applying predicted results for loop partition.

Input: Loop L

Output:*curr_thread*

partition_loop(loop L){
1 *start_block* := entry block of loop *L*;
2 *end_block* := exit block of loop *L*;
3 *likely_path* := the most likely path from *start_block* to *end_block*;
4 *opt_ddc*:= find_optimal_dependence(*start_block*, *end_block*, *likely_path*, &*spawn_pos*);
5 *loop_size* := the number of dynamic instructions in loop *L*;
6 if(*loop_size*<=**B**)
7 unroll(loop *L*);
8 else((*opt_ddc*<**A**)&&(**D**<*spawning_distance*<**E**)&&(**B**<*thread_size*<**C**))
9 *curr_thread* := create_new_thread(*end_block*, *spawn_pos*, *likely_path*);
10 end if
11 return *curr_thread*;}

Algorithm 4 Applying predicted results for nonloop partition.

Input:*start_block*, *end_block*, *curr_thread*(candidate thread)

Output:*curr_thread*

partition_thread(*start_block*, *end_block*, *curr_thread*){
1 if(*start_block* == *end_block*) then
2 return *curr_thread*;
3 end if
4 *pdom_block* := the nearest post dominator block of *start_block*;
5 *likely_path* := the most likely path from *start_block* to *pdom_block*;
6 *opt_ddc* := find_optimal_dependence(*pdom_block*, *curr_thread*, &*spawn_pos*);
8 if((**B**+0.25(**C**-**B**)<*thread_size*<**C**-0.25(**C**-**B**)) && (*opt_ddc*<**A**)) then
9 *thread_size* := *curr_thread* + sizeof (path);
10 finish_construction(*curr_thread*);
11 *curr_thread* := create_new_thread(*pdom_block*, *spawn_pos*, *likely_path*);
12 *curr_thread* := partition_thread(*pdom_block*, *end_block*, *curr_thread*);
13 else if(**C**-0.25(**C**-**B**)<*thread_size*<**C**) then
14 *thread_size*:= *curr_thread* + *path.first_block*;
15 opt_ddc:=find_optimal_dependence(path.*first_block*, *curr_thread*, null,&*spawn_pos*);
16 If((**B**+0.25(**C**-**B**)<*thread_size*<**C**) && (*opt_ddc*< **A**)) then
17 *curr_thread*:= *curr_thread*+path.*first_block*;
18 finish_construction(*curr_thread*);
19 *curr_thread* := create_new_thread(path.*first_block*,*spawn_pos*, *likely_path*);
20 *curr_thread* := partition_thread(path.*first_block*, *end_block*,*curr_thread*);
21 else
22 *curr_thread* := *curr_thread* + *path*;
23 *curr_thread* := *curr_thread* + *pdom_block*;
24 *curr_thread* := partition_thread(*pdom_block*, *end_block*, *curr_thread*);
25 end if
26 end if
27 return *curr_thread*;}

6. Conclusions

In this paper, we have presented and evaluated a parametric model *Qinling*, in order to explicitly explore the inherent law between thread partition factors and performance. *Qinling* makes use of a multiple regression model to predict speedups and to do feedback guidance for Prophet. It does so by three steps: first, it exploits linear relations between every primary influencing factors during thread partition and speedups; then, it builds and trains a multiple regression model offline, as well as predicting speedups of unseen applications; finally, by ways of building solution space and searching overall space to find the optimal solution, it searches offline for the optimal combination of thread partition so as to guide Prophet to achieve real speedups online.

The key characters of parametric model can be concluded: (1) primary influencing factors of thread partition are correlated with performance (speedup) by a parametric model *Qinling*; (2) the inherent law between thread partition and speedup is explicitly expressed; and (3) both offline prediction of speedups and online guidance of thread partition are realized.

Two future research works will be done based on *Qinling*: (1) training and validating programs will be classified so that different classes of programs use more fine parametric models; and (2) *Qinling* will be enhanced to meet the requirements of classifying applications.

Acknowledgments: We thank our colleagues for their collaboration and the present work. We also thank all the reviewers for their specific comments and suggestions. This work is supported by National Natural Science Foundation of China through grants No.61640219, National Natural Science Foundation of China through grants No.61602388, Doctoral Fund of Ministry of Education of China under Grant No.20130211110012,Natural Science

Basic Research Plan in Shaanxi Province of China under grant No.2017JM6059, China Postdoctoral Science Foundation under grant No.2017M613216, Postdoctoral Science Foundation of Shaanxi Province of China under Grant No.2016BSHEDZZ121 and the Fundamental Research Funds for the Central Universities under grants No.2452015194 and No.2452016081.

Author Contributions: Yuxiang Li contributed significantly to proposing idea, doing experiment, and manuscript preparation and revision; Yinliang Zhao provided a research project; Bin Liu helped performed the analysis with constructive discussions.

Conflicts of Interest: We declare that we have no financial and personal relationships with other people or organizations that can inappropriately influence our work, there is no professional or other personal interest of any nature or kind in any product, service and/or company that could be construed as influencing the position presented in, or the review of, the manuscript entitled, "Qinling: A Parametric Model in Speculative Multithreading".

References

1. Quiñones, C.G.; Madriles, C.; Sánchez, J.; Marcuello, P.; González, A.; Tullsen, D.M. Mitosis compiler: An infrastructure for speculative threading based on pre-computation slices. *ACM Sigplan Notices* **2005**, *40*, 269–279.
2. Madriles, C.; García-Quiñones, C.; Sánchez, J.; Marcuello, P.; González, A.; Tullsen, D.M.; Wang, H.; Shen, J.P. Mitosis: A speculative multithreaded processor based on precomputation slices. *IEEE Trans. Parallel Distrib. Syst.* **2008**, *19*, 914–925.
3. Ooi, C.L.; Kim, S.W.; Park, I.; Eigenmann, R.; Falsafi, B.; Vijaykumar, T. Multiplex: Unifying conventional and speculative thread-level parallelism on a chip multiprocessor. In Proceedings of the 15th International Conference on Supercomputing, Sorrento, Italy, 18–23 June 2001; pp. 368–380.
4. Liu, W.; Tuck, J.; Ceze, L.; Ahn, W.; Strauss, K.; Renau, J.; Torrellas, J. POSH: A TLS compiler that exploits program structure. In Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, New York, NY, USA, 29–31 March 2006; pp. 158–167.
5. Tournavitis, G.; Wang, Z.; Franke, B.; OBoyle, M.F. Towards a holistic approach to auto-parallelization: Integrating profile-driven parallelism detection and machine-learning based mapping. *ACM Sigplan Not.* **2009**, *44*, 177–187.
6. Li, Y.; Zhao, Y.; Gao, H. Using artificial neural network for predicting thread partitioning in speculative multithreading. In Proceedings of the 2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems, New York, NY, USA, 24–26 August 2015; pp. 823–826.
7. Zheng, B.; Tsai, J.Y.; Zang, B.; Chen, T.; Huang, B.; Li, J.; Ding, Y.; Liang, J.; Zhen, Y.; Yew, P.C.; et al. Designing the agassiz compiler for concurrent multithreaded architectures. In Proceedings of the International Workshop on Languages and Compilers for Parallel Computing, La Jolla, CA, USA, 4–6 August 1999; Springer: Berlin, Germany, 1999; pp. 380–398.
8. Gao, L.; Li, L.; Xue, J.; Yew, P.C. SEED: A statically greedy and dynamically adaptive approach for speculative loop execution. *IEEE Trans. Comput.* **2013**, *62*, 1004–1016.
9. August, D.I.; Huang, J.; Beard, S.R.; Johnson, N.P.; Jablin, T.B. Automatically exploiting cross-invocation parallelism using runtime information. In Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), Shenzhen, China, 23–27 Feburary 2013; IEEE Computer Society: Washington, DC, USA, 2013; pp. 1–11.
10. Wang, S.; Yew, P.C.; Zhai, A. Code transformations for enhancing the performance of speculatively parallel threads. *J. Circuits Syst. Comput.* **2012**, *21*, 1240008.
11. Sohi, G. Multiscalar: Another fourth-generation processor. *Computer* **1997**, *30*, 72–72.
12. Grewe, D.; Wang, Z.; O'Boyle, M.F. A workload-aware mapping approach for data-parallel programs. In Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers, Heraklion, Greece, 24–26 January 2011; pp. 117–126.
13. Long, S.; Fursin, G.; Franke, B. A cost-aware parallel workload allocation approach based on machine learning techniques. In Proceedings of the IFIP International Conference on Network and Parallel Computing, Dalian, China, 18–21 September 2007; Springer: Berlin, Germany, 2007; pp. 506–515.

14. Wang, Z.; O'Boyle, M.F. Partitioning streaming parallelism for multi-cores: A machine learning based approach. In Proceedings of the 19th international conference on Parallel architectures and compilation techniques, Vienna, Austria, 11–15 Sepetember 2010; pp. 307–318.
15. Wang, Z.; O'Boyle, M.F.P. Mapping parallelism to multi-cores: A machine learning based approach. In Proceedings of the ACM Sigplan Symposium on Principles and Practice of Parallel Programming, PPOPP 2009, Raleigh, NC, USA, 14–18 February 2009; pp. 75–84.
16. Singer, J.; Yiapanis, P.; Pocock, A.; Lujan, M.; Brown, G.; Ioannou, N.; Cintra, M. Static java program features for intelligent squash prediction. In Proceedings of the Statistical and Machine learning approaches to ARchitecture and compilaTion (SMART'10), Pisa, Italy, 24 January 2010; p. 14.
17. Chen, X.; Long, S. Adaptive multi-versioning for OpenMP parallelization via machine learning. In Proceedings of the 2009 15th International Conference on Parallel and Distributed Systems (ICPADS), Shenzhen, China, 9–11 December 2009; pp. 907–912.
18. Liu, B.; Zhao, Y.; Zhong, X.; Liang, Z.; Feng, B. A novel thread partitioning approach based on machine learning for speculative multithreading. In Proceedings of the 2013 IEEE 10th International Conference on High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing, Zhangjiajie, China, 13–15 November 2013; pp. 826–836.
19. Monsifrot, A.; Bodin, F.; Quiniou, R. A machine learning approach to automatic production of compiler heuristics. In Proceedings of the International Conference on Artificial Intelligence: Methodology, Systems, and Applications, Varna, Bulgaria, 4–6 September 2002; Springer: Berlin, Germany, 2002; pp. 41–50.
20. Stephenson, M.; Amarasinghe, S. Predicting unroll factors using supervised classification. In Proceedings of the International Symposium on Code Generation and Optimization, San Jose, CA, USA, 20–23 March 2005; pp. 123–134.
21. Agakov, F.; Bonilla, E.; Cavazos, J.; Franke, B.; Fursin, G.; O'Boyle, M.F.; Thomson, J.; Toussaint, M.; Williams, C.K. Using machine learning to focus iterative optimization. In Proceedings of the International Symposium on Code Generation and Optimization, Manhattan, NY, USA, 26–29 March 2006; IEEE Computer Society: Washington, DC, USA, 2006; pp. 295–305.
22. Cavazos, J.; Dubach, C.; Agakov, F.; Bonilla, E.; O'Boyle, M.F.; Fursin, G.; Temam, O. Automatic performance model construction for the fast software exploration of new hardware designs. In Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems, Seoul, Korea, 22–27 October 2006; pp. 24–34.
23. Wang, Z.; Powell, D.; Franke, B.; OBoyle, M. Exploitation of GPUs for the parallelisation of probably parallel legacy code. In Proceedings of the International Conference on Compiler Construction, Grenoble, France, 5–13 April 2014; Springer: Berlin, Germany, 2014; pp. 154–173.
24. Luk, C.K.; Hong, S.; Kim, H. Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In Proceedings of the 2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), New York, NY, USA, 12–16 December 2009; pp. 45–55.
25. Khan, S.; Xekalakis, P.; Cavazos, J.; Cintra, M. Using predictive modeling for cross-program design space exploration in multicore systems. In Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques, Brasov, Romania, 15–19 September 2007; IEEE Computer Society: Washington, DC, USA, 2007; pp. 327–338.
26. Lee, B.C.; Brooks, D.M. Accurate and efficient regression modeling for microarchitectural performance and power prediction. *ACM SIGOPS Oper. Syst. Rev.* **2006**, *40*, 185–194.
27. Yang, S.; Shafik, R.A.; Merrett, G.V.; Stott, E.; Levine, J.M.; Davis, J.; Al-Hashimi, B.M. Adaptive energy minimization of embedded heterogeneous systems using regression-based learning. In Proceedings of the 2015 25th International Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS), Bahia, Brazil , 1–4 September 2015; pp. 103–110.
28. Cavazos, J.; O'boyle, M.F. Method-specific dynamic compilation using logistic regression. *ACM Sigplan Not.* **2006**, *41*, 229–240.
29. Chen, Z.; Zhao, Y.L.; Pan, X.Y.; Dong, Z.Y.; Gao, B.; Zhong, Z.W. An overview of Prophet. In Proceedings of the International Conference on Algorithms and Architectures for Parallel Processing, Taipei, Taiwan, 8–11 June 2009; Springer: Berlin, Germany, 2009; pp. 396–407.
30. Dong, Z.; Zhao, Y.; Wei, Y.; Wang, X.; Song, S. Prophet: A speculative multi-threading execution model with architectural support based on CMP. In Proceedings of the 2009 International Conference on Scalable

Computing and Communications; Eighth International Conference on Embedded Computing, Dalian, China, 25–27 September 2009; pp. 103–108.
31. Liu, B.; Zhao, Y.; Li, Y.; Sun, Y.; Feng, B. A thread partitioning approach for speculative multithreading. *J. Supercomput.* **2014**, *67*, 778–805.
32. Bhowmik, A.; Franklin, M. A general compiler framework for speculative multithreading. In Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures, Winnipeg, MB, Canada, 11–13 August 2002; pp. 99–108.
33. Cao, Z.; Verbrugge, C. Mixed model universal software thread-level speculation. In Proceedings of the 2013 42nd International Conference on Parallel Processing, Lyon, France, 1–4 October 2013; pp. 651–660.
34. Wilson, R.P.; French, R.S.; Wilson, C.S.; Amarasinghe, S.P.; Anderson, J.M.; Tjiang, S.W.; Liao, S.W.; Tseng, C.W.; Hall, M.W.; Lam, M.S.; et al. SUIF: An infrastructure for research on parallelizing and optimizing compilers. *ACM Sigplan Not.* **1994**, *29*, 31–37.
35. Holmes, G.; Donkin, A.; Witten, I.H. Weka: A machine learning workbench. In Proceedings of the 1994 Second Australian and New Zealand Conference on Intelligent Information Systems, Brisbane, Australia, 29 November–2 December 1994; pp. 357–361.
36. Hammond, L.; Hubbert, B.A.; Siu, M.; Prabhu, M.K.; Chen, M.; Olukolun, K. The stanford hydra CMP. *IEEE Micro* **2000**, *20*, 71–84.
37. Carlisle, M.C. Olden: Parallelizing Programs with Dynamic Data Structures on Distributed-Memory Machines. Ph.D. Thesis, Princeton University, Princeton, NJ, USA, 1996.
38. Compiler, M.S.B. *The Machine-SUIF 2.1 Compiler Documentation Set*; Harvard University: Cambridge, MA, USA, 2000.
39. Chen, Y. A cramer rule for solution of the general restricted linear equation? *Linear Multilinear Algebra* **1993**, *34*, 177–186.