IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received March 1, 2017, accepted March 15, 2017, date of publication March 20, 2017, date of current version April 24, 2017.

Digital Object Identifier 10.1109/ACCESS.2017.2684129

Toward Emotion-Aware Computing: A Loop Selection Approach Based on Machine Learning for Speculative Multithreading

BIN LIU, (Member, IEEE), JINRONG HE, YAOJUN GENG, LVWEN HUANG, AND SHUQIN LI
College of Information Engineering, Northwest A&F University, Yangling 712100, China

Corresponding author: S. Li (lsq_cie@nwsuaf.edu.cn)

This work was supported in part by the National Natural Science Foundation of China under Grant 61602388, in part by the Postdoctoral Science Foundation of Shaanxi Province of China under Grant 2016BSHEDZZ121, in part by the Fundamental Research Funds for the Central Universities under Grant 2452015194 and Grant 2452016081, and in part by the Yangling Demonstration Zone Science and Technology Planning Project under Grant 2016NY-31.

ABSTRACT Emotion-aware computing can recognize, interpret, process, and simulate human affects. These programs in this area are compute-intensive applications, so they need to be executed in parallel. Loops usually have regular structures and programs spend significant amounts of time executing them, and thus loops are ideal candidates for exploiting the parallelism of sequential programs. However, it is difficult to decide which set of loops should be parallelized to improve program performance. The existing research is one-size-fits-all strategy and cannot guarantee to select profitable loops to be parallelized. This paper proposes a novel loop selection approach based on machine learning (ML-based) for selecting the profitable loops and paralleling them on multi-core by speculative multithreading (SpMT). It includes establishing sufficient training examples, building and applying prediction model to select profitable loops for speculative parallelization. Using the ML-based loop selection approach, an unseen emotion-aware sequential program can obtain a stable, much higher speedup than the one-size-fits-all approach. On Prophet, which is a generic SpMT processor to evaluate the performance of multithreaded programs, the novel loop selection approach is evaluated and reaches an average speedup of 1.87 on a 4-core processor. Experiment results show that the ML-based approach can obtain a significant increase in speedup, and Olden benchmarks deliver a better performance improvement of 6.70% on a 4-core than the one-size-fits-all approach.

INDEX TERMS Emotion-aware computing, loop selection, machine learning, speculative multithreading.

I. INTRODUCTION

Emotion-aware computing is that computer has ability to recognize the emotional state of humans and gives an appropriate response for these emotions. In recent years, emotion-aware computing could offer benefits and plays an important role in an almost limitless range of applications. For example, in intelligent recommender systems, the computer can assess emotional offset quantitatively to revise user ratings for improving the objectivity of context data [1], [2]. In information retrieval systems, the computer can exploit emotions in short films and automatically extract affective context from user comments for emotion-aware film retrieval [3]. In healthcare cyber-physical systems, the computer can recognize patient's emotion and adjust the treatment plan by collecting the information published on the social networks [4]. In audio-visual emotion recognition, the computer can identify a user's emotion and behavior for providing a rich user experience [5]. In mobile cloud computing, the computer can offer emotion care for improving people's health status by providing personalized emotion-aware services [6]. Other applications include community activity prediction [7], human–computer interaction [8] and cloud gaming [9]. Because the emotion-aware computing is a compute-intensive task, the most existing emotion-aware applications need to be executed in parallel for quickly getting the results of the analysis.

With the rapid development of computer architecture, multi-core processors are the only way to build high-performance microprocessors now [10]. To improve speedups of emotion-aware applications on multi-core processors, these sequential programs must be reconstructed so that they can be executed in parallel [11]. SpMT is an auto-parallelization technology that depends on out of order execution on multi-core processors. Examples of

SpMT include SEED [12], DOE [13], DOMORE [14] and Prophet [15].

The programs usually spend a lot of time executing loops, consequently, loops are being the target of parallel computing. These research of [16]–[19] partitioned the loops into multithreads to improve program performance. Some existing loop selection schemes only decomposed innermost loops or outermost loops into multithreads, in which each thread includes an outermost loop or innermost loop. Research in [20]–[23] was still based on heuristics or simple evaluation model to select loops and only indirectly estimated the speculative multithreaded execution. The existing one-size-fits-all approaches obtain some performance improvements, but they apply the same strategies to select loops with different characteristics to be parallelized. These approaches are one-size-fits-all solutions and only fit for one kind of loops, while the ML-based approach proposes a loop-aware selection scheme for speculative parallelization according to the characteristics of the target loops.

In this article, a novel ML-based loop selection approach is proposed to select profitable loops to be executed in parallel. Firstly sufficient training examples are provided for machine learning model. The profiler is developed to collect the profiling information. Using profiling information, the Prophet compiler employs the thread partitioner to partition loops in turn into multithreaded programs that are estimated on the Prophet simulator. Finally these classification labels of loops and themselves form the training examples. Furthermore, features of these loops and their classification labels are put together to form training samples. Then, these training samples are used to build a K Nearest Neighbor (KNN) model for predicting the classification label for unseen loops in emotion-aware applications.

The novel ML-based loop selection approach is implemented in Prophet Compiler [24] based on SUIF research complier development framework. Experimental results show that the ML-based approach can select profitable loops for speculative parallelization and fully exploit the inherent parallelism of loops. Finally, the ML-based approach achieves an average speedup of 1.87 in SpMT system and provides a performance improvement of 6.70% for Olden benchmarks than the one-size-fits-all approach.

The remainder of this article is organized as follows. In Section II, the SpMT execution model of Prophet is first briefly described. In Section III, based on program profiling technology and the loop thread partitioning algorithm, the sufficient training examples are established. Section IV describes the ML-based loop selection framework, including feature extraction, training sample generation, training and applying prediction model to predict classification label for an unseen loop in emotion-aware application. Section V analyzes experimental results provided by the ML-based loop selection approach. In Section VI, related work is introduced and summarized. Finally, we conclude this article in Section VII.

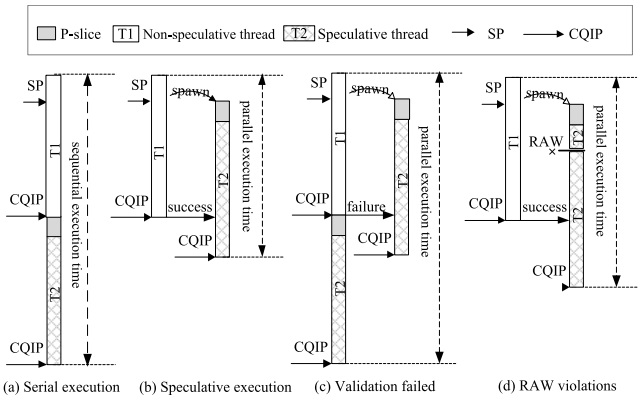

FIGURE 1. Speculative mutithreading execution model of Prophet.

II. PRELIMINARY

Speculative multithreading is an effective technology for automatic parallelization of sequential programs. SpMT execution model of Prophet is shown in Fig. 1 [25], serial program is decomposed into multiple speculative threads to be executed in parallel, and a different part of the program is executed by each speculative thread. During program runtime, only one non-speculative thread is allowed to commit calculation results to memory, and the results of all the other speculative threads should be validated before committing. On Prophet, spawning instruction pairs are used to label speculative threads and each of them is composed of Spawning Point (SP) and Control Quasi Independent Point (CQIP) in the program. If the sequence of SP-CQIPs is ignored, the program is sequential as shown in Fig. 1(a). When an SP defined in parent thread is identified during program runtime, as shown in Fig. 1(b), the speculative thread following the CQIP will be spawned by the parent thread. In SpMT System, the data generated by speculative threads need to be verified by the non-speculative thread. If it fails validation, non-speculative thread will continue to execute serially the speculative threads as shown in Fig. 1(c). Otherwise, non-speculative thread rewrites the memory value and the speculative thread becomes non-speculative thread. In addition, Read after Write (RAW) dependence between non-speculative thread and speculative thread occurs as shown in Fig.1 (d), speculative child thread will restart itself on current state.

III. ESTABLISHING TRAINING EXAMPLES

In order to select the ideal loops of sequential program for speculative parallelization, loop selection is cast as the problem of machine learning. The paper wishes to train a prediction model which, given the feature vector X_{orig} of the loop, predicts its classification label Y that whether it can be parallel by speculative multithreading. However, a particular problem encountered is that few sequential programs can satisfy the requirements for establishing training samples. To solve this problem, in our previous research [26], after

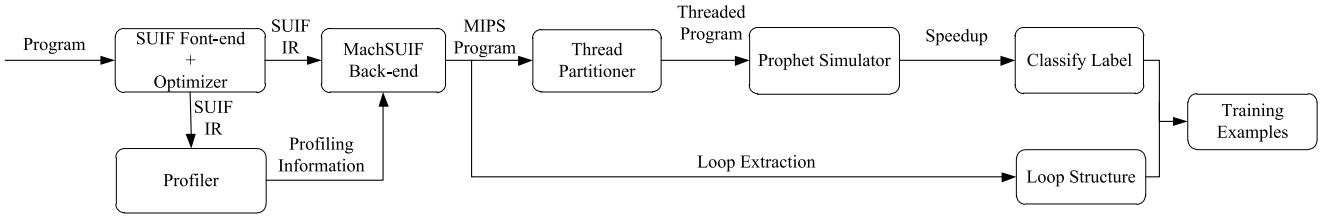

FIGURE 2. Generating training examples.

analyzing features affecting program's speedup, these essential features are chosen to represent a sequential program. Then, a feature set is built based on Olden benchmarks and disturb it to generate a new feature set. Furthermore, using the feature set, sufficient virtual programs are generated as a supplementary to the existing Olden benchmarks. These virtual programs will allow us to train the prediction model on larger training examples.

Training examples are generated as shown in Fig. 2. The serial program is transformed into SUIF intermediate representation (IR). The IR is then optimized by applying various optimizations available in SUIF. Then, the profiler is used to analyze and collect executive information such as the average dynamic instruction counts of loop bodies, the average dynamic instruction counts of called functions and the branch probabilities of branch instructions. The MachSUIF back-end takes SUIF IR with the profiling information and generates low-SUIF IR with annotated MIPS program. The loops in annotated MIPS programs are partitioned into multithreads by thread partitioner and the threaded program is estimated on the Prophet simulator for each loop. If the speedup of loop in speculative parallelization greater than 1, the classification label is YES, otherwise NO. Finally, classification label of the loop and itself are chosen as a training example.

A. PROFILER

Program profiling is a method that gathers and analyzes the runtime information during program execution in the past. Once the profiling information is collected, it is fed back to compiler which performs the predictive optimization. In theory, if the compiler can figure out all the precise program behavior, it can generate perfect binary code with the best performance on any platform. However, in fact, the compiler only obtains partial and imprecise program behavior, such as calling frequency of functions, branch probability, etc. Based on the profiling information, the compiler can eliminate uncertainty factors and obtain more precise program behavior than static analysis. In this paper, each program has been executed multiple times in advance by using multiple different input data sets. A profiler has been realized for generating these profiling information such as control flow and structure information that are also fed back into the Prophet compiler for guiding the loop thread partitioning.

Based on the control flow information, the Prophet compiler uses profiler to determine the most likely path of program by branch probability. Then, the control-based approach is used to generate multithreads on the path. Through the selection of the most likely path, it can eliminate some data dependences with low probabilities and perform data speculations.

For the sake of minimizing load imbalance among threads, the size of the thread should be considered, so Prophet compiler needs to know that how many instruction cycles a piece of code will occupy during program execution. Because there are unpredictable control flows, interrupts and nested function calls, dynamic instruction is difficult to be captured by the static analysis. In this section, program is first executed multiple times, and the profiler is used to collect dynamic instructions of functions and loops. Because a thread is composed of several basic blocks, Prophet compiler can estimate the execution time of a thread. For a function, the number of calls and the number of dynamic instructions are counted to calculate the average dynamic instruction count. For a loop, the profiler records its iteration times and dynamic instruction count of loop body in every iteration to get the average dynamic instruction count of its body.

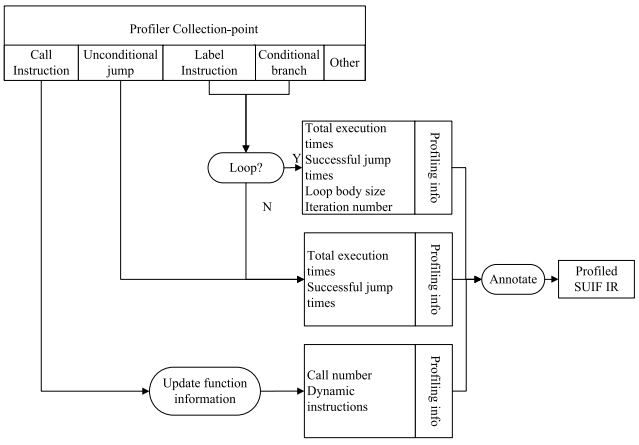

FIGURE 3. Profiling information collector.

Synthesizing the above analyses, the profiler is developed to collect the following profiling information on SuifVM that is a virtual machine and can execute the SUIF IR instructions, the workflow of which is shown in Fig. 3. When a SUIF IR instruction is executed, the different instruction types are processed in different ways:

- branch probability
  The profiler monitors all the branch instructions and counts the execution times of each branch instruction E and the execution times of each branch instruction when its criterion condition is true T, then the profiler can calculate the probability of a conditional jump instruction with true $P = T/E$.
- average dynamic instruction count of a function
  The profiler collects the number of current function calls C and the number of current dynamic instructions S. Then, the average dynamic instruction count is calculated: $N_C = (N_{C-1} * (C - 1) + S)/C$, where N_{C-1} is the average dynamic instruction count in the last invocation.
- average dynamic instruction count of a loop
  The profiler records the iteration times D and the dynamic instruction count M in current iteration. Then, the average dynamic instruction count of loop is $L_D = (L_{D-1} * (D - 1) + M)/D$, where L_{D-1} is the average dynamic instruction count in the last iteration.

B. THREAD PARTITIONER

In this section, the Prophet compiler uses the thread partitioner to partition loops into multithreads for program parallelization. The thread partitioner is composed of control flow graph (CFG), control flow analysis, structure analysis, data flow analysis, data dependence model and thread partitioning algorithm.

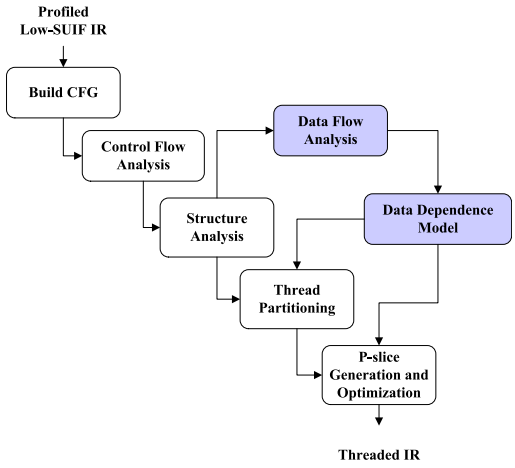

FIGURE 4. Thread partitioner.

As shown in Fig. 4, the source program is preprocessed into profiled low-SUIF IR. The low-SUIF program is analyzed by Prophet compiler and the compiler builds its CFG. Then Prophet compiler constructs the weighted control flow graph (WCFG) with profiling information. Next step, a structural analysis identifies the loop and non-loop regions. After that, based on data dependence model and data flow analysis, the loop thread partitioning algorithm partitions the loops into multithreaded programs. Hereto, the partitioner has finished all the work and generates threaded IR.

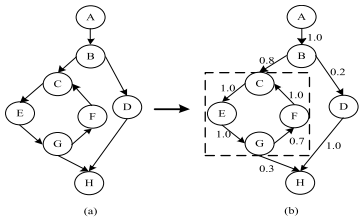

FIGURE 5. Program structure preprocessing. (a) CFG. (b) WCFG.

1) STRUCTURE PREPROCESSING

Before loop thread partitioning, the low-SUIF program is preprocessed, the process has three major phases:

First, CFG is a directed graph of a program structure, so CFG is used to represent a program for analyzing control structures among basic blocks in high-level presentation. Based on CFG library provided by MachSUIF, instruction sequences can be divided into basic blocks. Fig. 5 (a) shows a CFG of a program. In CFG, the basic blocks are represented by vertices and the edges show the flow of control among the basic blocks.

Second, profiler is used to collect branch probability, function and loop profiling information, and the profiling information is annotated to CFG. The CFG of program and the corresponding profiling information form a WCFG and The WCFG is realized as shown in Fig. 5 (b).

Third, based on structural analysis, the Prophet compiler traverses the WCFG of each function and identifies the loop regions.

2) DATA DEPENDENCE MODEL

In SpMT, all speculative multithreads are executed aggressively in parallel and the inter-thread ambiguous data dependences are permitted. At runtime, all the data miss-speculations can be detected and the Prophet underlying system can recover from data miss-speculations to ensure the correctness of the program. However, excessive data dependences among threads may lead to thread squash frequently.

In Prophet compiler, a data dependence model is implemented based on data dependence counts to quantify the data dependence degree between successive threads. If the two successive threads have a small count value, it means that successor thread has somewhat data independent on its predecessor thread and is a good candidate thread to be executed in parallel with its predecessor thread.

Definition 1: The data dependence counts DDC (T_i, T_{i+1}) are the number of data dependence arcs coming into a successor thread T_i from predecessor thread T_{i+1}.

3) LOOP THREAD PARTITIONING ALGORITHM

Thread partitioning algorithm is crucial to the performance improvement in SpMT. Therefore, good thread partitioner needs to consider control flow, thread size, and data dependence model as a basis for thread partitioning algorithm.

The loop thread partitioning algorithm is described by Algorithm 1. While partitioning a loop, the Prophet compiler

Algorithm 1 Algorithm for Loop Thread Partitioning

Input: Loop region of each program for thread partitioning *loop*

Output: Multithreded program of *loop*

```
loop_level_thread_partitoning(Loop region loop)
{
 begin_block := loop.entry_block;
 exit_block := loop.exit_block;
 possible_executed_path := the possible executed path
  between begin_block to exit_block;
 opt_dep := compute_optimization_dependence
 (begin_block, exit_block, possible_executed_path,
 &spawn_location);
 dynamic_thread_size :=dynamic_thread_size
 (possible_executed_path);
 if(loop_size <= LOOP_SIZE_ THRESHOLD )
    unwinding(loop);
 else (opt_dep <OPTIMAL_DATA_DEPENDENCE_
 THRESHOLD)
    new_thread := generated_next_thread(exit_block,
    spawn_location, possible_executed_path);
 end if
 return multithreaded loop;
}
```

checks the profiling information on the average times of iterations and the number of dynamic instructions of loop. In general, small loops that has few dynamic instructions or iterates fewer times are not be partitioned into multithreads for speculative execution. On the contrary, the small loops are unwinding to increase the loop size, after which these loop bodies can be further partitioned into multithreaded program. For moderate loops, the Prophet compiler considers the data dependence counts OPTIMAL_DATA_DEPENDENCE_THRESHOLD between two successive iterations. If the new thread spawned at the next iteration is profitable, then the Prophet compiler will generate the thread for speculative parallelism.

4) AN EXAMPLE OF LOOP THREAD PARTITIONING

The virtual programs generated by our research and genuine programs are equal for loop thread partitioning. Although the actual meaning of virtual programs does not already exist, program features influencing speedup such as data and control dependences are still retained. Before loop thread partitioning, CFG is first constructed for each function, and then all the loops of programs are identified by the Prophet compiler using data and control flow analysis.

Figure 6 shows the example of loop partitioning. Only if the thread size of loop body is proper and data dependence counts between successive iterations is less than a fixed threshold OPTIMAL_DATA_DEPENDENCE_THRESHOLD, the SP instruction is kept at the beginning basic block of loop body, and then a CQIP instruction is placed at the end basic block of loop body. When the program is executed

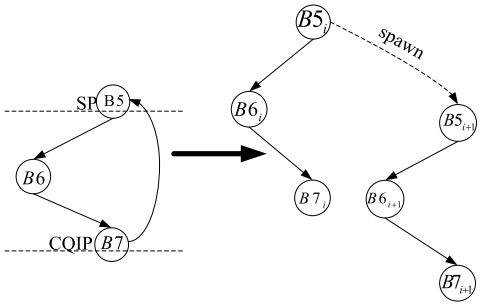

FIGURE 6. An example of loop partitioning.

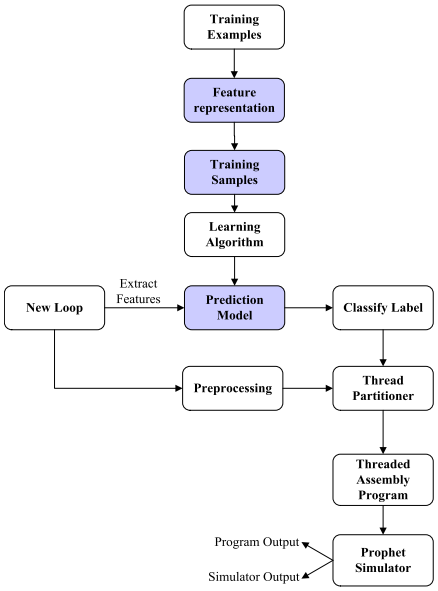

FIGURE 7. The ML-based Loop selection approach.

speculatively, these loop bodies can be executed in parallel by the SP-CQIP instruction pair.

C. TRAINING EXAMPLES GENERATION

Our training examples are generated from prior knowledge by off-learning. These Olden benchmarks and virtual programs run on the Prophet simulator for collecting training examples. Using the loop partitioning algorithm (as shown in Algorithm 1), all the loops of training programs are broken down into multiple speculative threads, and the paper employs the Prophet simulator to evaluate the multithreaded programs. By the experimental evaluation, these profitable loops that can obtain good parallel performance are positive examples, other loops are negative examples. The generation process of training examples take two days by using eight Prophet simulators.

IV. ML-BASED LOOP SELECTION APPROACH

In this section, an ML-based loop selection approach is used to select loops for speculative parallelization and it mainly includes feature representation and extraction, training samples generation, and ML-based loop selection as illustrated in Fig. 7.

TABLE 1. Features description.

Features	Category	Descriptions
Block count	Static feature	The number of basic blocks in a loop
Iteration count		Loop iteration count
Static instruction count		The number of static instructions in a loop
Load instruction count		Static load instruction count in a loop
Store instruction count		Static store instruction count in a loop
Loop depth		The depth of the loop in the loop tree
Dynamic instruction count	Dynamic feature	dynamic instruction count in a loop
Loop probability		The probability that the condition is true
DDC		The number of data dependence between two iterations
DDD		The distance of data dependence between two iterations

A. FEATURE REPRESENTATION AND EXTRACTION

One of the key aspects in ML-based loop selection approach is selecting the appropriate features to represent the loop. As we know, if a loop has the similar features with another loop, the two loops will have the same classification label for speculative parallelization. As shown in Table 1, the paper extracts these features from a loop and uses them to characterize the loop.

When extracting the loop features, the SUIF-IR (high-level intermediate representation of SUIF) of program is firstly constructed, and then all feature analyses are performed on SUIF-IR. Static features are counted and extracted from training programs by Prophet compiler, while dynamic features are dealt with by program profiling. For a given loop, these features make up a fixed-length feature vector $X_{orig} = [x_1, x_2, \cdots, x_n]$ that is used to characterize the loop.

B. TRAINING SAMPLES GENERATION

The feature vector X_{orig} of each loop and itself classification label Y_{orig} constitute a training sample. Sample dataset is composed of features of all the loops and their classification labels. Feature vector of each loop $X_{orig} = [x_1, x_2, \cdots, x_n]$ corresponds to a point in the n-dimensional space R^n, the classification label Y_{orig} is an eigenvalue representing whether the loop can be speculatively parallelized.

C. LOOP SELECTION APPROACH BASED ON KNN

In this section, we focus on predicting classification label for an unseen loop using prediction model, then the preprocessing module will be in charge of converting a real program into MIPS program that is taken as input for thread partitioner in SUIF.

Classification label of an upcoming predicted loop is equivalent to its nearest samples, which is the premise whereon the ML-based loop selection approach partitions the loop into parallel threads. Once sufficient training samples are generated, an ML-based loop selection approach can be built. This prediction model is based on a KNN classification. The loop selection algorithm is described by Algorithm 2. When program is executing on Prophet, the feature vector X_{orig} of each loop and its label Y_{orig} are collected dynamically as a training sample. Once all the training program is running over, all training samples $< x_{org-i}, y_{orig}(x_{org-i}) >$ are put together to form the KNN model.

Algorithm 2 Loop Selection Algorithm Based on KNN

Input: training examples $<X, Y>$(X represents feature vectors and Y is a set of classification labels), Xq (feature vector of new, unseen loop)

Output: Y(Xq) (predicted classification label for Xq)

knn_classifier (training examples $<X, Y>$, feature vector Xq)

{

All the training samples $<x, y(x)>$ are put together to form *training_examples*;

while (feature vector x_q that need to be predicted) **do**

Find and locate the k-nearest neighbor examples $x_1 \ldots x_k$ of the unseen feature vector Xq from *training_examples*;

end while

return $\hat{y}(x_q) = \arg\max_i^k \delta(\hat{y}_i(x_i) == y_i),\ (1 \le i \le k)$

$$\delta(p) = \begin{cases} 0 & \text{if } p \text{ is true} \\ 1 & \text{if } p \text{ is false.} \end{cases}$$

}

KNN is a simple and effective classifier. KNN-based learning methods only simply store the training samples, and the generalizing beyond these samples is postponed until a new unseen sample must be classified. When a new sample should be dealt with, the k nearest neighbor training samples are retrieved from memory and voting among its k nearest neighbors is used to classify the new sample.

In order to solve this problem, the ML-based approach firstly extracts features of the unseen loop x_q, and then its k nearest training samples are found and located by the formula 1.

$$d(x_q, x_i) = \sqrt{\sum_{r=1}^{n} (x_q^r - x_i^r)^2} \tag{1}$$

where n is the dimension of feature vector, x_q^r is the r^{th} attributes of feature vector x_q and x_i^r is the r^{th} attributes feature vector x_i, respectively. Once the k nearest training samples are found, the predicted classification label can be obtained by a simple majority vote to the K nearest training samples for guaranteeing model approximation and generalization.

Because the objective function is a one-dimensional feature vector Y, predicted result of the new sample x_q is the

TABLE 2. Prophet simulator parameters (per PE).

Configuration item	Value
Fetch, issue and commit width	4 instructions
Pipeline stages	Fetch/Issue/Ex/WB/Commit
Architectural registers	32 int and 32 fp
Function units	4 int ALU (1 cycle) 2 int mult/div (3/12 cycles) 2 fp ALU (2 cycles) 2 fp mult/div (4/12 cycles)
L1-cache(multiversion)	4-way associative 64KB (32B/block) hit latency 2 LRU replacement
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

label that is voted by K nearest training samples.

$$\hat{y}(x_q) = \arg\max_i^k \delta(\hat{y}_i(x_i) == y_i), \quad (1 \le i \le k)$$
$$\delta(p) = \begin{cases} 0 & \text{if } p \text{ is true} \\ 1 & \text{if } p \text{ is false} \end{cases} \tag{2}$$

When the prediction model is built as described above, the classification label of the unseen loop can be predicted as illustrated in Fig. 7. Once the classification label is predicted successfully, given a new sequential program, the predicted classification label is employed by the thread partitioner to decide whether the loop should be partitioned into multi-threads for speculative parallelism.

V. EXPERIMENTAL EVALUATION

In this section, experimental setup is introduced and details of the Prophet simulator and benchmarks are provided. At last, experimental results are analyzed and discussed.

A. EXPERIMENTAL SETUP

The execution model and the ML-based loop selection algorithm are implemented on Prophet. The compiler analyses and optimization of all the programs have been done at the SUIF-IR. We have developed a profiler to execute SUIF-IR programs and generate profiling information such as the most likely path of control flow graph, the average dynamic instruction count of loop iterations and called function, data value prediction. The Prophet simulator [27] models a multi-core processor with speculative multithreading technology. Each processing elements (PEs) has its own program counter, fetch unit, decode unit, and execution unit and it can fetch and execute instructions from a thread. Each PE has a private multi-versioned L1 cache with 2 cycles access latency and it is employed to cache the speculation results for each PE and performs cache communication. The prophet simulator uses MIPS ISA and the configuration parameters are shown in Table 2.

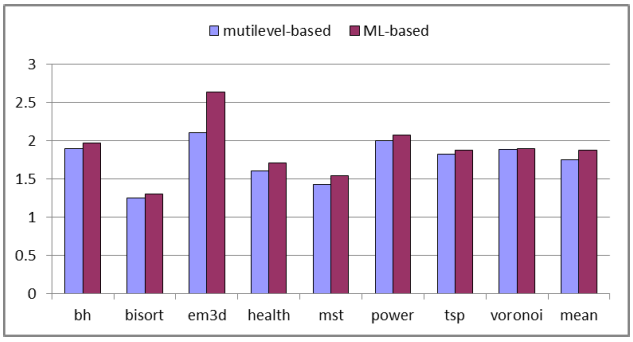

FIGURE 8. Speedup performance.

This paper uses Olden benchmarks to evaluate the ML-based selection approach. Olden benchmarks are written in C and they are classic benchmarks of emotion-aware applications. These programs usually manipulate complex data structure such as lists and trees, so they are difficult to be paralleled.

In this section, leave one out cross validation (LOOCV) is used to estimate the ML-based approach. That is, the program that its loops will be predicted for speculative parallelization is removed from the training samples and then a prediction model based on the remaining programs is built. This guarantees that the prediction model has not seen the target program before. The prediction model is used to generate selection scheme for the removed program. This process is repeated for each program in turn. It is a standard evaluation methodology, providing an estimation of the generalization ability of an ML-based model for predicting an unseen program.

B. EXPERIMENTAL RESULTS AND ANALYSES

In this section, Olden benchmarks are tested on the Prophet simulator by using the multilevel-based loop selection approach [28] (it is a representative of one-size-for-all approach) and the ML-based loop selection approach.

From Fig. 8, the experimental results show that, using the ML-based loop selection approach, all the programs of Olden benchmarks have gained better performance than multilevel-based approach, particularly ***em3d*** and ***health*** have obtained major performance improvement. Only program ***voronoi*** exhibits a meager increase. For further analyses, increase rate is defined as:

$$rate\_inc = \frac{ML\_speedup - multilevel\_speedup}{multilevel\_speedup} \times 100\% \tag{3}$$

As shown the right column in Table 3, the variation of the increasing rate is from 0.61% to 24.97%. Compared with the multilevel-based approach, experiment results show that the ML-based loop selection approach can select profitable loop for speculative parallelization and exploit more speculative parallelism. And further explains that the ML-based loop selection approach is valid and can get better performance improvement than the multilevel-based approach.

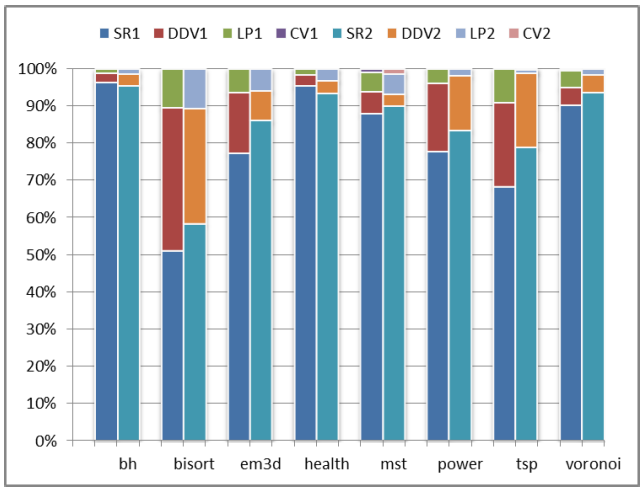

FIGURE 9. Breakdown of speculative execution.

The comparison of speculative execution breakdown between multilevel-based approach with 1 and ML-based loop selection approach with 2 is shown in Fig.9. SR represents the success rate of speculative execution. DDV refers to the failure rate of speculative execution caused by data dependence violations. CV is the failure rate of speculative execution caused by control dependence violation. LP means the failure rate of speculative execution caused by low priority of speculative threads.

Program ***bh*** is a solver of the classic n-body problem. From Table 4 and 5, it is obvious that program ***bh*** adopts heterogenous octree as the main data structures, there are complex data dependences in loops, at same time the average number of iterations and loop body size are 9.14 and 200.45 (Table 5), respectively. Compared with the multilevel-based approach, the ML-based approach employs profitable loops to execute in parallel, resulting that the speculative success rate is decreased. However, program ***bh*** has some extent of increase in the number of the spawned threads, so the spawned thread count of speculative execution is increased (Table 3), finally the ML-based approach provides a performance improvement of 3.35% for program ***bh***.

Program ***bisort*** implements forward and backward sort algorithms. From Table 5, it is obvious that there are 3 loop structures with fewer dynamic instructions and only 2 loops are executed in fact. Because program ***bisort*** uses binary tree data structure to store the numbers and has complex data dependences (Figure 9), both the multilevel-based selection approach and ML-based approach obtain lower speedup of 1.26 and 1.30 respectively. Compared with the multilevel-based approach, the ML-based approach chooses profitable loops that have larger data dependence counts and thread size to exploit all possible parallelism in program. Due to the increase of successful speculative threads, eventually the ML-based approach achieves a performance improvement of 3.87%.

Program ***em3d*** models the transmission of electromagnetic waves in a three-dimensional object. As shown in Table 5, there are many larger loops that contain 265.99 average dynamic instructions and the average iteration times are 8.73, so program ***em3d*** has larger loop-level parallelism and obtains the highest speedup by the multilevel-based approach and the ML-based approach (Figure 8). Furthermore, using the ML-based loop selection approach, program ***em3d*** gains the optimal performance improvement. All nested loops are parallelized by multilevel loop selection approach whereas the ML-based approach has only selected profitable loops for speculative parallelization according to the characteristics of loops in program ***em3d***. As shown in Table 3, both the success rate of speculative execution and the spawned thread count of program ***em3d*** have some increase, so that successful threads have also increase, resulting that the ML-based loop selection approach exploits more speculative threads to participate in parallel computation and the speedup obtains improvement of 24.97%.

Program ***health*** uses a four-way tree to model the multilevel health-care system that includes multiple hospitals. ***Health*** has 14 loops, just like program ***em3d***, loop-level parallelism is the major source of program ***health***. Although program ***health*** has many loops, there is only one nested loop, so there are increased faintly in the success rate of speculative execution and the spawned thread count. Previous multilevel-based loop selection approach chooses some unprofitable loops for speculative execution while the ML-based approach selects all profitable loops to be executed in parallel by machine learning to exploit loop-level parallelism. As shown in Table 3, compared with the multilevel-based approach, the successfully spawned threads is increased, hence, the ML-based approach allows more speculative threads to take part in speculative execution and has gained a performance improvement of 6.39%.

Program ***mst*** is a kind of graph algorithm that calculates the minimum spanning tree. ***Mst*** employs singly linked lists to solve problem and has complex data structures. Meanwhile, the multilevel-based approach selects 5 loops with larger thread size for speculative execution while the ML-based approach chooses all the profitable loops. Compared with the multilevel-based approach, the ML-based approach predicts the classification label for every loop by predicting model, resulting the spawned thread count and the success rate of spawned threads increased. At last, program ***mst*** spawns a lot of the successfully spawned threads for speculative parallelization and causes a performance improvement of 8.06%.

Program ***power*** is a solver of power pricing system problem. ***Power*** has similar data structure with ***mst***, it adopts multi-way tree and single-linked lists as data structure and has complex data dependencies. As shown in Table 5, program ***power*** contains 20 loop structures and loop body size is big. Because of complex data dependences, compared with the multilevel-based approach, the ML-based approach picks up all the profitable loops and partitions them into multithreads to executed in parallel. Through analysis, the parallelism mainly comes from loop-level parallelism, especially in function main and *optimize_node*. From Table 3 and Fig. 9, it is

TABLE 3. The dynamic information statistics of the olden benchmarks.

Olden benchmarks	Spawned Threads		Speculative Success Rate		Successful Threads		Performance Improvement
	Multilevel	ML	Multilevel	ML	Multilevel	ML	
bh	119920	122887	0.963426	0.955195	115534	117381	3.35%
bisort	64577	60361	0.5106	0.583986	32973	35250	3.87%
em3d	2275	3807	0.774066	0.860783	1761	3277	**24.97%**
health	3684	3688	0.9541	0.9542	3515	3519	6.39%
mst	490	1014	0.8796	0.9375	431	951	8.06%
power	121083	122461	0.800525	0.835335	96930	102296	3.61%
tsp	198304	182394	0.68258	0.78612	135358	143384	2.77%
voronoi	111149	109972	0.924129	0.937484	102716	103097	0.61%
MEAN	70352	70616	0.8152	0.850644	58924	60779	6.70%

TABLE 4. The static characterization statistics of the olden benchmarks.

Olden benchmarks		***bh***	***bisort***	***em3d***	***health***	***mst***	***power***	***tsp***	***voronoi***	**mean**	**variance**
Multilevel-based method	Thread size	129.93	67.00	353.50	51.33	39.5	168.02	30.44	143.18	122.86	11346.34
	Slice/Thread	0.05	0.10	0.02	0.08	0.10	0.04	0.10	0.06	0.07	0.00
	Live-ins	5.10	6.00	5.63	3.22	3.2	4.87	1.75	7.55	4.67	3.43
ML-based method	Thread size	42.00	43.00	38.67	47	40	58.41	28.58	133.85	53.94	1112.38
	Slice/Thread	0.13	0.15	0.11	0.10	0.10	0.10	0.09	0.04	0.10	0.00
	Live-ins	4.28	5.50	3.33	3.55	3	4.71	1.83	3.92	3.77	1.25

TABLE 5. Chararistic of olden benchmarks.

Olden	***bh***	***bisort***	***em3d***	***health***	***mst***	***power***	***tsp***	***voronoi***
Main pointer data structures	Heterogenous Octree	Binary tree	Single-linked lists	Double-linked lists, Quadtree	Array of singly linked lists	Multiway tree, Single-linked lists	Binary tree, Linked lists	Binary tree
Input parameters	64-2	512-1	10-3-30	2-16-1	8-1	1-1	100-1	128-1-0
# excuted /sum	24/28	2/3	13/15	13/14	11/13	17/20	6/7	3/4
Average number of iterations	9.14	5.87	8.73	3.51	3.92	2 .88	94.5	1.3
Loop body size	200.45	41.42	265.99	172.88	830.59	2625.27	1267.25	137

obvious that the spawned success rate and the spawned thread count are increased, so the success spawned threads have also increased. Finally, the ML-based approach obtains a respectable performance improvement of 3.61%.

Program ***tsp*** implements the traveling salesman problem in a planargraph. As shown in Table 5, only 6 loops are executed in fact while there are 7 loop structures. However, the average iteration times of loop are 94.5 and the average dynamic instruction count of loop bodies is 1267.25. Previous multilevel-based approach selects the 6 loops for speculative execution while the ML-based approach chooses 4 loops. Compared with previous approach, although the ML-based approach generates a fewer spawned threads, it improves the success rate and increases the successfully spawned threads. Finally, the ML-based approach achieves a performance improvement of 2.77%.

Program ***voronoi*** generates a voronoi digram for random points. There are some loops in program ***Voronoi***, but almost no nested loops. As shown in Table 5, although loop body size is 137, the iteration times are very small, only 1.3, therefore loops contains fewer parallelism. Compared with the multilevel-based approach, the ML-based loop selection approach predicts the classification label for each loop by prediction model and then the loop thread partitioner decomposes them into multithreads to be executed in parallel. As illustrated in Table 3, although the spawned thread count is reduced, the corresponding success rate of speculative execution is increased. As a result, the successful spawned thread count is increased. The results indicate that the ML-based loop selection approach doesn't appear to be quite different from the multilevel-based approach, so the speedup only obtains a performance improvement of 0.61%.

As shown in Table 4, it is obvious that the mean and variance of thread size generated by the ML-based loop selection approach are lesser than multilevel-based approach. It indicates that modest thread size can gain higher speedup.

Furthermore, some programs (***bh***, ***em3d***, ***power*** and ***voronoi***) exploit the parallelism by coarse-grained threads (thread size is greater than 100 instructions), while some programs (***bisort***, ***health***, ***mst***, and ***tsp***) use fine-grained threads to exploit the parallelism.

Through the above analyses, two conclusions can be concluded: (1) The ML-based loop selection approach can choose profitable loops and partition them into multithreads for speculative execution by coarse-grained or fine-grained threads. Which thread size is helpful to improve the speedup? This is determined by the loop's characteristics; (2) Although the sacrifice of a certain success rate of speculative execution will increase the time overhead of thread squash, compared to the multilevel-based approach, the ML-based loop selection approach can pick up more speculative threads for speculative parallelism; (3) The speedup of multithreaded program is related to not only success rate and the spawned thread count but also load imbalance and thread size. Whether the speedup of program is improved depends on benefit is greater than the overhead caused by speculative parallelism. In conclusion, the ML-based approach achieves an average speedup of 1.87 and provides a performance improvement of 6.70% for Olden benchmarks than the multilevel-based approach. The experimental results show that the ML-based loop selection approach can find the optimal classification label of loop for speculative parallelization and is more stable approach in different programs.

VI. RELATED WORK

Emotion-aware computing is computing that the computer is capable of understanding and responding to human emotions. Most existing emotion-aware applications in this area are compute-intensive sequential programs and they need to be executed in parallel for quickly obtaining these results of emotion-aware computing. Because loops hold most of the computation time during the execution, they are the target of parallel computing. Liu *et al.* [29] proposed a loop selection approach based on performance prediction. Basing on profiling information and various speculative factors, they established a prediction model for selecting loops that are executed in parallel. Shen *et al.* [20], using profiling information, proposed an evaluation method to choose loops for maximizing program performance. Liu *et al.* [30] leveraged the code structure of the loop iterations to generate tasks and used profiling information to discard ineffective tasks by estimation. Liu *et al.* [28], using expert experiences, parallelized all the profitable nested loops by multilevel spawning strategies. Li and Zhao [17] developed the thread counter to predict the runtime performance of threads. Based on the analysis of different thread behaviors, loops are selected automatically to be performed in parallel. Using misspeculation cost model, Johnson *et al.* [31] decomposed loop structure into multiple threads. Gayatri *et al.* [18] used task-based programming model to parallel all while-loops. Aldea *et al.* [19] augmented the OpenMP capabilities by using thread level speculation technology and supported the all nested loops for speculative execution. Hirata *et al.* [23] used the expert experience to select loops and executed their iterations speculatively in parallel. Bhattacharyya *et al.* [22] proposed a framework that can use two different heuristics to find loops for speculative parallelization. These studies have obtained some acceleration effect for loop-level parallelism, but there are still some limitations: First, these works select loops for speculative execution based on simple cost model and not all of them can obtain good parallel performance. Second, the existing methods adopt one-size-fits-all strategies to select loops, only partial loops are selected for speculative execution. In contrast, this paper uses a loop-aware characteristic approach to select loops for speculative parallelization.

VII. CONCLUSION

The paper proposes a novel ML-based loop selection approach to choose the profitable loops for speculative parallelization in emotion-aware applications. For the sake of providing sufficient training examples for prediction model, the profiler is developed to collect the profiling information that reflects more precise program behavior than static analysis. Using profiling information, the Prophet compiler employs the thread partitioner to partition loops into multithreaded programs, and then these programs are estimated by using the Prophet simulator. Finally these classification labels of loops and themselves form the training examples. Furthermore, features of these loops and their classification labels are put together to generate sufficient training samples. The paper uses training samples to build automatically a model that is used to predict the classification label for unseen loops in emotion-aware sequential program, allowing for a quick search for the thread solution space.

The ML-based loop selection approach is implemented on the Prophet compiler. The results show that the ML-based approach can accurately select profitable loops for speculative parallelization. From an overall perspective, using Olden benchmarks, the results are satisfactory and the ML-based approach achieves performance improvement by an average of 6.70% on a 4-core processor than the one-size-fits-all approach. In conclusion, the ML-based approach employs a loop-aware selection strategy and delivers a significant increase in speedup for each of unseen emotion-aware sequential programs.

ACKNOWLEDGMENT

The authors are grateful for anonymous reviewers' hard work and comments that allowed us to improve the quality of this paper.

REFERENCES

[1] Y. Zhang, "GroRec: A group-centric intelligent recommender system integrating social, mobile and big data technologies," *IEEE Trans. Serv. Comput.*, vol. 9, no. 5, pp. 786–795, Oct. 2016.

[2] Y. Zhang, M. Chen, D. Huang, D. Wu, and Y. Li, "iDoctor: Personalized and professionalized medical recommendations based on hybrid matrix factorization," *Future Generat. Comput. Syst.*, vol. 66, pp. 30–35, Jan. 2017.

[3] C. Orellana-Rodriguez, E. Diaz-Aviles, and W. Nejdl, "Mining affective context in short films for emotion-aware recommendation," in *Proc. 26th ACM Conf. Hypertext Soc. Media*, 2015, pp. 185–194.
[4] Y. Zhang, M. Qiu, C. W. Tsai, M. M. Hassan, and A. Alamri, "Health-CPS: Healthcare cyber-physical system assisted by cloud and big data," *IEEE Syst. J.*, vol. 11, no. 1, pp. 88–95, Mar. 2017.
[5] M. S. Hossain, G. Muhammad, M. F. Alhamid, B. Song, and K. Al-Mutib, "Audio-visual emotion recognition using big data towards 5G," *Mobile Netw. Appl.*, vol. 21, no. 5, pp. 753–763, 2016.
[6] M. Chen, Y. Zhang, Y. Li, S. Mao, and V. C. M. Leung, "EMC: Emotion-aware mobile cloud computing in 5G," *IEEE Netw.*, vol. 29, no. 2, pp. 32–38, Mar./Apr. 2015.
[7] Y. Zhang, M. Chen, S. Mao, L. Hu, and V. Leung, "CAP: Community activity prediction based on big data analysis," *IEEE Netw.*, vol. 28, no. 4, pp. 52–57, Jul./Aug. 2014.
[8] Y. Fu, H. V. Leong, G. Ngai, M. X. Huang, and S. C. F. Chan, "Physiological mouse: Towards an emotion-aware mouse," in *Proc. IEEE 38th Annu. Int. Comput., Softw. Appl. Conf. Workshops*, Jul. 2014, pp. 258–263.
[9] H. S. Hossain, G. Muhammad, B. Song, M. M. Hassan, A. Alelaiwi, and A. Alamri, "AudioŰvisual emotion-aware cloud gaming framework," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 25, no. 12, pp. 2105–2118, Dec. 2015.
[10] Y. Luo and A. Zhai, "Dynamically dispatching speculative threads to improve sequential execution," *ACM Trans. Archit. Code Optim.*, vol. 9, no. 3, pp. 1–31, 2012.
[11] X. Sun and Y. Chen, "Reevaluating Amdahl's law in the multicore era," *J. Parallel Distrib. Comput.*, vol. 70, no. 2, pp. 183–188, 2010.
[12] L. Gao, L. Li, J. Xue, and P.-C. Yew, "SEED: A statically greedy and dynamically adaptive approach for speculative loop execution," *IEEE Trans. Comput.*, vol. 62, no. 5, pp. 1004–1016, May 2013.
[13] M. Sharafeddine, K. Jothi, and H. Akkary, "Disjoint out-of-order execution processor," *ACM Trans. Archit. Code Optim.*, vol. 9, no. 3, pp. 1–32, 2012.
[14] H. Jialu, T. B. Jablin, S. R. Beard, N. P. Johnson, and D. I. August, "Automatically exploiting cross-invocation parallelism using runtime information," in *Proc. IEEE/ACM Int. Symp. Code Generat. Optim.*, Feb. 2013, pp. 1–11.
[15] X. Wang, Y. Zhao, Y. Wei, S. Song, and B. Han, "Prophet synchronization thread model and compiler support," in *Proc. Int. Symp. Parallel Distrib. Process. Appl.*, 2010, pp. 81–87.
[16] K. Ootsu, T. Yokota, and T. Baba, "Performance improvement of hot-path based thread partitioning technique by unifying loop parallelization," in *Proc. Parallel Distrib. Computing Syst.*, 2011, pp. 110–121.
[17] M. Li and Y. Zhao, "A dynamically adaptive approach for speculative loop execution in SMT architectures," in *Proc. 16th IEEE Int. Conf. High Perform. Comput. Commun.*, Jun. 2014, pp. 1024–1031.
[18] R. Gayatri, R. M. Badia, and E. Aygaude, "Loop level speculation in a task based programming model," in *Proc. IEEE Int. Conf. High Perform. Comput.*, Mar. 2013, pp. 39–48.
[19] S. Aldea, A. Estebanez, D. R. Llanos, and A. Gonzalez-Escribano, "An OpenMP extension that supports thread-level speculation," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 1, pp. 78–91, Jan. 2016.
[20] L. Shen, F. Xu, and Z. Wang, "Optimization strategies oriented to loop characteristics in software thread level speculation systems," *J. Comput. Sci. Technol.*, vol. 31, no. 1, pp. 60–76, 2016.
[21] M. Li, Y. Zhao, Y. Tao, and Q. Wang, "A static greedy and dynamic adaptive thread spawning approach for loop-level parallelism," *J. Comput. Sci. Technol.*, vol. 29, no. 6, pp. 962–975, 2014.
[22] A. Bhattacharyya and J. N. Amaral, "Automatic speculative parallelization of loops using polyhedral dependence analysis," in *Proc. Int. Workshop Code Optim. Multi Many Cores*, 2013, pp. 1–9.
[23] H. Hirata, A. Nunome, and K. Shibayama, "Speculative memory: An architectural support for explicit speculations in multithreaded programming," in *Proc. IEEE Int. Conf. Comput. Inf. Sci.*, Oct. 2016, pp. 1–7.
[24] Z. Li, Y. Zhao, and D. Yanning, "Design and implementation of the prophet speculative multithreading system," *Comput. Sci.*, vol. 38, no. 2, pp. 296–301, 2011.
[25] B. Liu, Y. Zhao, X. Zhong, Z. Liang, and B. Feng, "A novel thread partitioning approach based on machine learning for speculative multi-threading," in *Proc. IEEE Int. Conf. High Perform. Comput. Commun.*, Oct. 2014, pp. 826–836.
[26] B. Liu, Y. Zhao, M. Li, Y. Liu, and B. Feng, "A virtual sample generation approach for speculative multithreading using feature sets and abstract syntax trees," in *Proc. 13th Int. Conf. Parallel Distrib. Comput., Appl. Technol.*, Mar. 2012, pp. 39–44.
[27] Z. Dong, Y. Zhao, Y. Wei, X. Wang, and S. Song, "Prophet: A speculative multi-threading execution model with architectural support based on CMP," in *Proc. Int. Conf. Scalable Comput. Commun.*, Sep. 2009, pp. 103–108.
[28] B. Liu, Y. Zhao, Y. Li, Y. Sun, and B. Feng, "A thread partitioning approach for speculative multithreading," *J. Supercomput.*, vol. 67, no. 3, pp. 778–805, 2014.
[29] B. Liu *et al.*, "A loop selection approach based on performance prediction for speculative multithreading," *J. Electron. Inf. Technol.*, vol. 36, no. 11, pp. 2768–2774, 2014.
[30] W. Liu *et al.*, "POSH: A TLS compiler that exploits program structure," in *Proc. 11th ACM SIGPLAN Symp. Principles Pract. Parallel Program.*, 2006, pp. 158–167.
[31] T. A. Johnson, R. Eigenmann, and T. N. Vijaykumar, "Speculative thread decomposition through empirical optimization," in *Proc. 12th ACM SIGPLAN Symp. Principles Pract. Parallel Program.*, 2007, pp. 205–214.

BIN LIU was born in Shaanxi in 1981. He received the B.S. degree in computer science and technology from the Shaanxi University of Science and Technology, China, in 2004, and the M.Sc. degree in technology with a major in parallel computing and machine learning from Yunnan University, China, in 2010, and the Ph.D. degree in electronic and information engineering from Xi'an Jiaotong University, China, in 2014. His research interests focus on emotion-aware computing, parallel computing, and machine learning.

Since 2014, he has been an Assistant Professor with the College of Information Engineering, Northwest A&F University, China. He is also a Post-Doctoral Fellow with the College of Mechanical and Electronic Engineering, Northwest A&F University. He currently serves as a Reviewer for the IEEE TRANSACTIONS ON COMPUTERS, the *Journal of Supercomputing*, and so on.

JINRONG HE was born in Gansu in 1981. He received the B.S. degree in science of information and computation, the M.Sc. degree in computational mathematics from the Wuhan University of Technology, China, in 2007 and 2010, respectively, and the Ph.D. degree in computer software and theory from the State Key Laboratory of Software Engineering, Wuhan University, China, in 2014. He is currently an Assistant Professor with Northwest A&F University. His research interests include affective computing and machine learning.

YAOJUN GENG was born in Shanxi in 1982. He received the B.S., M.sc., and Ph.D. degrees in computer science and technology from Xidian University. Since 2013, he has been an Assistant Professor with the College of Information Engineering, Northwest A&F University, China. His research interests focus on emotion-aware computing, machine learning, and bioinformatics.

LVWEN HUANG was born in Hunan in 1976. He received the B.Eng. degree in automation engineering in 1999, the M.Sc. degree in signal and information processing from the Xi'an University of Technology, China, in 2005, and the Ph.D. degree in agricultural electrification and automation from Northwest A&F University, China, in 2013. Since 2005, he has been an Assistant Professor with the College of Information Engineering, Northwest A&F University. His research interests lie in emotion-aware computing, intelligent system, and image processing.

SHUQIN LI was born in Shaanxi in 1965. She received the B.S. and M.Sc. degrees from Northwest Agricultural University, China, in 1986 and 1989, respectively. Since 2003, she has been a Professor with the College of Information Engineering, Northwest A&F University. Her research interests include intelligent information system and emotion-aware computing.