The Journal of Supercomputing (2019) 75:7366–7390
https://doi.org/10.1007/s11227-019-02943-1

# ProCTA: program characteristic-based thread partition approach

**Yuxiang Li[1] · Zhiyong Zhang[1] · Lili Zhang[1] · Danmei Niu[1] · Changwei Zhao[1] · Bin Song[1] · Liuke Liang[2]**

Published online: 3 July 2019
© Springer Science+Business Media, LLC, part of Springer Nature 2019

## Abstract

As a thread-level automatic parallelization technique, thread-level speculation (TLS) can partition irregular serial programs into multiple threads and implement these threads in parallel on multi-core architectures to improve the performance of programs. To tackle the problem that the conventional heuristic rule-based (HR-based) thread partition approach partitions programs of different characteristics with the same scheme and several programs have bad partition results, this paper proposes a program characteristic-based thread partition approach (ProCTA), which uses a machine learning method to learn the knowledge of thread partition from TLS sample set and predicts thread partition schemes for unknown programs in accordance with programs' characteristics and finally applies the schemes to thread partition. In Prophet compilation system, Olden benchmarks are used to evaluate ProCTA, and a comparison is made between ProCTA and conventional heuristic rules-based partition approach. The experimental results show that the proposed approach can deliver an average 18.24% speedup improvement than HR-based thread partition approach.

**Keywords** Thread-level speculation · Thread partition · Program characteristics · Partition scheme

✉ Zhiyong Zhang
xidianzzy@126.com

Yuxiang Li
liyuxiang@haust.edu.cn

[1] Henan Joint International Research Laboratory of Cyberspace Security Applications, Henan University of Science and Technology, Luoyang, People's Republic of China

[2] Luoyang Normal University, Luoyang, People's Republic of China

## 1 Introduction

With development of semiconductor evolving from the single core era to the multi-core era, how to accelerate the serial programs and make full use of the rich multi-core resources becomes a research focus. Thread-level speculation (TLS) [10, 26, 27] can automatically partition one serial program into multiple threads in an aggressive way, and allows the existence of fuzzy control dependence and data dependence between threads, and allows multiple threads to simultaneously execute on a Chip Multiprocessor (CMP), so to exploit thread-level parallelism for programs. Various schemes of TLS emerge as the time requires. The products of TLS's development can be classified into three headings, including:

1. systems: Hydra [7], Multiscalar [5], POSH [17], STAMPede [25], Mitosis [18, 21], Pinot [19], and Prophet [2, 4, 24];
2. algorithms [26, 27];
3. related products [9–11, 14, 15]

Thread partition approach is of vital importance to improve the speedups of programs. Conventional thread partition methods usually adopted heuristic rules to perform thread partition. The paper [8] used a series of balanced minimum cuts to partition programs under consideration of various overheads of thread partition in total, and adjusted the edges' weights of programs' control flow graphs after every partition. The paper [3] compared different partition algorithms quantitatively under the given computing architecture and proposed a new dynamic partition algorithm:Mex-slicing, which outperformed other dynamic partition algorithms, but could not reach a balance between predictability and cost. In conclusion, these heuristic rules-based (HR-based) thread partition methods have the "one-size-fits-all" limitations, namely uniform heuristic rules are utilized to partition all programs, which lead to that it is hard to get the optimal thread partition scheme for every program.

To solve the shortcomings of HR-based thread partition method [18, 22], machine learning methods are successfully introduced into TLS. The paper [2] proposed a method that the portable compiler used the machine learning method to match the parallelism of a program to a multi-core processor. This paper developed two predictors, one data-sensitive and one data-insensitive to select the best matches for parallel programs. The paper [6] developed a method for predicting the optimal thread data in the presence of additional load for data parallel applications and developed a comprehensive model to reduce the effects of additional loads and to increase the average speedup. The paper [16] adopted a machine learning method to learn thread partition knowledge and applied it to direct thread partition, in which parameters of thread partition are set by expert experience (also called expert experience-based (EE-based) thread partition). The paper [13] presented a novel graph-based thread partition approach, firstly to characterize programs by graphs, integrating static and dynamic features, as well as data and control information; secondly to learn partition knowledge and predict partition for unseen programs.

Compared with conventional heuristic rules-based thread partitions, this paper applies a program characteristic-based thread partition approach (Pro CTA) to learn thread partition knowledge from TLS sample set and predicts the partition scheme of unseen programs in accordance with their characteristics. The advantages of ProCTA can be summarized to two aspects: (1) programs' partition schemes are generated in accordance with their characteristics; (2) partition knowledge can be learnt by machine learning methods so to facilitate the partition of unknown programs. The remaining parts of this paper are organized as follows: in Sect. 2, we first briefly describe the SpMT execution model and motivation of ProCTA; the process of thread partition is detailed in Sect. 3; in Sect. 4, thread partitions (loop partition and unloop partition) are presented; Sect. 5 presents experimental evaluation; conclusion and future work are shown in Sect. 6; final section presents acknowledgement.

## 2 Execution model and motivation

### 2.1 Execution model

TLS parallelizes serial programs and performs parallel execution on multi-core platforms to improve speedup performance. Figure 1 presents the TLS execution model [11, 12], in which the spawning point (SP) and the control quasi-independent point (CQIP) instructions map the serial programs into multi-threaded programs. According to serial semantics, there is only one thread that allows data to be submitted to memory at each moment. This thread is called a definite thread, and the other threads are regarded as speculative threads. Every speculative thread consists of two parts, including precomputation-slice (p-slice) [18] and serial program code. P-slice is a small piece of code that is generated by the compiler based on slicing techniques to predict the live-ins used in speculative threads (a set of variables to be referenced before the value is defined).

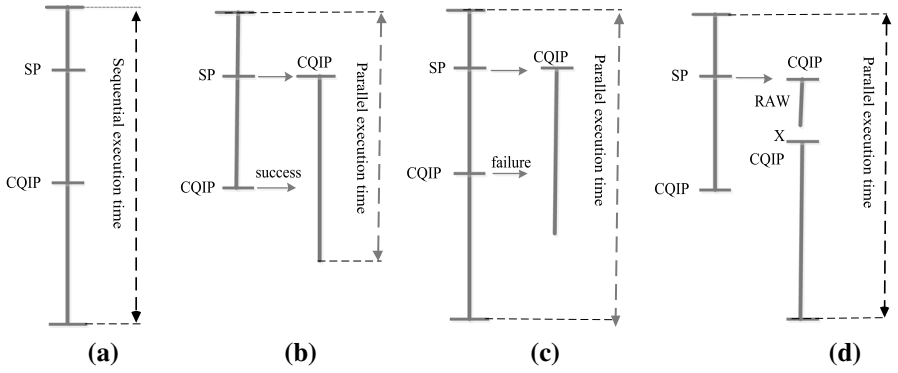



**Fig. 1** Model of thread-level speculation: **a** sequential execution; **b** successful parallel execution; **c** failed parallel execution; **d** RAW

Figure 1 shows four cases of SpMT execution. In Fig. 1a, it is assumed that a multi-threaded program is equivalent to a serial execution program because it ignores SP-CQIP. Figure 1b shows the successful speculative execution: when the thread T1 encounters sp, if idle cores exist, the new speculative thread T2 is spawned; otherwise, the T2 is not spawned. When T1 encounters CQIP, it will validate the live-ins used by T2 in p-slice. If the validation is correct, T1 submits the execution results and releases the core resource. Then, the execution permission is transferred from T1 to the successor thread of T1; Fig. 1c presents the state that validation of T2 fails, and speculative execution fails so to withdraw T2, and p-slice is not performed; Fig. 1d illustrates the situation of restarting the thread in the current state when read-after-write (RAW) violation happens.

### 2.2 Motivation

Figure 2 shows 13 partitioning schemes and their corresponding speedups for the same subprocedure of the *health*() in Olden benchmarks [1]. The x-axis represents the 13 partition schemes of the same subprocedure, while the vertical axis represents their corresponding speedups. The partition scheme is represented by a five-dimensional vector whose composition is present in Chapter 4. The motivation denoted by Fig. 2 is that the procedures' partition schemes influence achieved speedups after their partition. How to predict the partition scheme based on programs' characteristics has become a research goal of this paper.

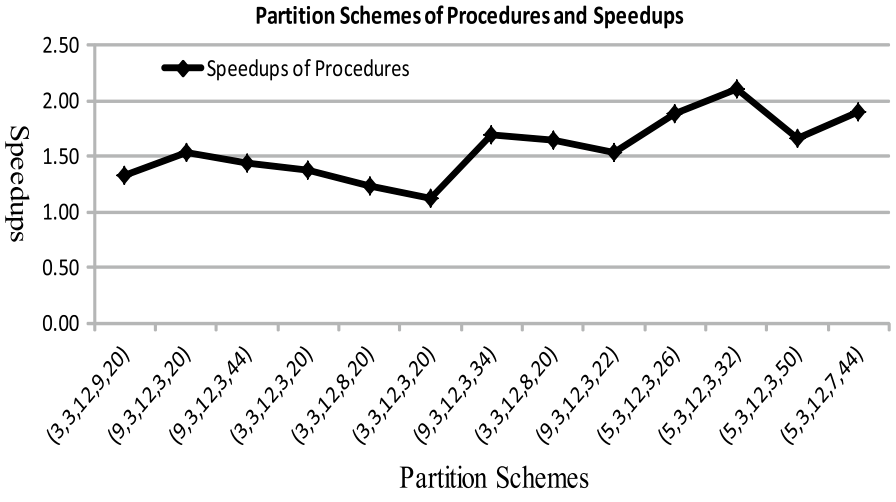



**Fig. 2** 13 Partition schemes and speedups of procedures in *health*()

## 3 Process of thread partition

This paper adopts a program characteristic-based thread partition approach (ProCTA), which partitions a serial program into multi-threaded programs that can be executed in parallel. ProCTA firstly extracts programs' features, obtaining their optimal partition schemes according to their characteristics, and then performs their partitions according to their partition schemes.

Figure 3 shows the framework of thread partition based on programs' characteristics. In this figure, the dashed box indicates the process that the K Nearest Neighbor (KNN) algorithm learns partition knowledge of the training samples, and then builds a predictive model. Once the prediction model is constructed, characteristics of the program to be partitioned are regarded as the input of prediction model, and the partition scheme of an unknown program is predicted, and partition of the program is guided in the Prophet system [4, 24] to generate the multi-threaded programs. Then the evaluation is completed in the Prophet simulator, outputting the programs' results and speedups.

The framework consists of five main components, namely:

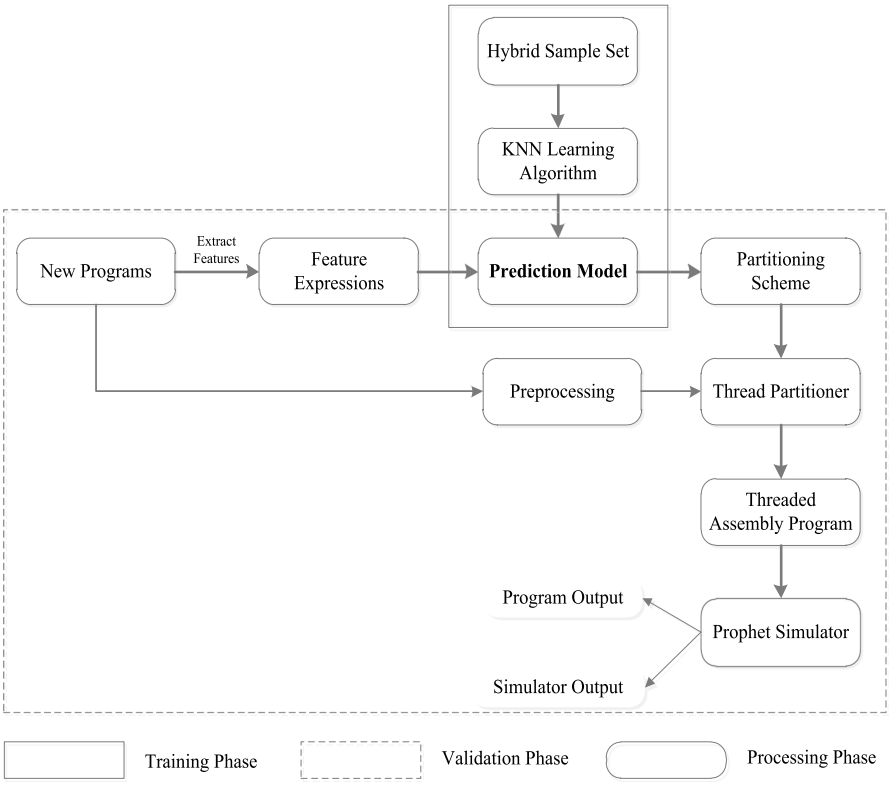



**Fig. 3** Scheme of thread partition based on program characteristics

1. Expression of programs' characteristics;
2. Expression of partition schemes;
3. Construction of prediction model;
4. Thread partition;
5. Experimental evaluation.

The following sections describe the five components in the partition scheme.

### 3.1 Expression of program's features

ProCTA obtains programs' features by using a system's profiler in Prophet [2] to collect programs' running information. If the instruction is a type of jump instruction, the jump is processed according to whether or not the jump is a loop jump, and if the jump is normal, the current jump is updated. If the jump is a back edge, then the average number of dynamic instructions $L_D$ is calculated in accordance with the dynamic instruction number of current jump $M$, the average number of dynamic instructions in the history record $L_{D-1}$, the total iteration count $D$ and the average number of dynamic instructions $L_D$; Second, if the instruction is the procedure's return instruction, then the average number of dynamic instructions $N_C$ is calculated in accordance with the current number of dynamic instructions $S$, the average number of dynamic instructions in the history record $N_{C-1}$, the total called number $C$; Finally, the results will be annotated with the corresponding information domain of information probe function.

Figure 4 shows the initial storage structure of characteristics extracted by a profiler, where $c$ is the number of dynamic instructions, $l$ is the branch probability, $b$ is the number of basic blocks, *d1* is the number of branch instructions, *d2* is the number of jump instructions, *d3* is the number of add/subtract instructions, *d4* is the number of multiplication/division instructions, $n$ is the number of programs' basic blocks. Every element in the matrix $M$ is a feature vector, which includes dynamic instructions, probabilities of loop branches, probabilities of branches, number of branch instructions, number of jump instructions, number of addition/subtraction instructions, number of multiplication/division instructions, and number of programs' basic blocks.

In this paper, static and dynamic features are used to represent the input sample program. To facilitate the calculation of the Euclidean distance, the eight variables $x1 \sim x8$ are chosen to represent features of the sample programs. The specifically used characteristic variables and their descriptions are shown in Table 1.

### 3.2 Expression of partition scheme

The partition scheme is extracted from the nonloop partition algorithms and loop partition algorithms in Prophet [1]. The execution flow of partition algorithm (nonloop partition and loop partition) can be described by the flow diagram in Fig. 5. Seen from Fig. 5, the upper limit of thread granularity (*ULoTG*), the lower limit of the thread size (*LLoTG*), data dependent count (*DDC*), the upper limit of spawning

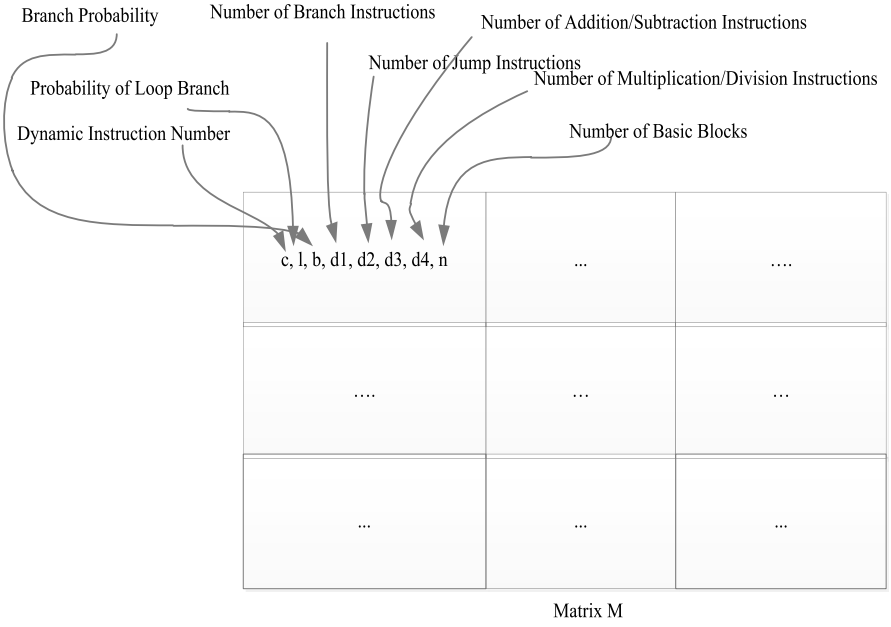



**Fig. 4** Expression of program characteristics

**Table 1** Used characteristics and their descriptions

| Variables | Descriptions of characteristics |
|---|---|
| *x1* | Number of basic blocks |
| *x2* | Number of dynamic instructions |
| *x3* | Number of branch instructions |
| *x4* | Probabilities of branches |
| *x5* | Probabilities of loop branches |
| *x6* | Number of addition and subtraction |
| *x7* | Number of multiplication and division |
| *x8* | Number of loop instructions |

distance (*ULoSD*), the lower limit of spawning distance (*LLoSD*) directly affect the process of thread partition, in which the $1_{st}$ decision and $2_{nd}$ decision refer to two different decisions (such as: spawning new threads, loop unrolling and repartition). If thread partition can be reduced to a decision problem, then these five thresholds are the direct factors determining the answer to the problem. It can be concluded that the changes of the five thresholds directly affect the decision of partition, thus affecting the result of partition. Therefore, this paper selects these five thresholds as the optimized parameters.

Partition scheme is expressed by $H =< h1, h2, h3, h4, h5 >=< LLoTG, ULoTG, DDC, LLoSD, ULoSD >$. Where, *LLoTG*, *ULoTG*, *DDC*, *LLoSD*, *ULoSD* are the five primary parameters affecting partition effects during thread partition.

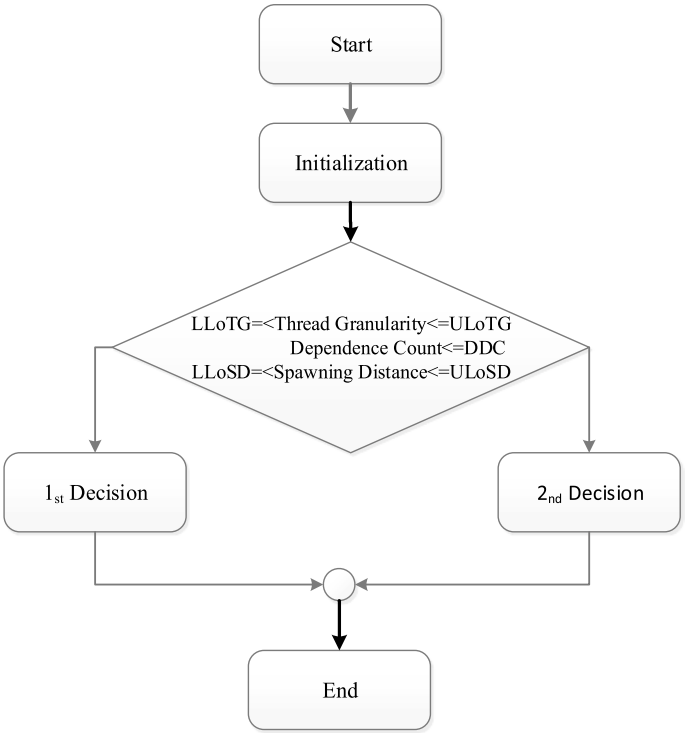



**Fig. 5** Flow chart of extracting partition schemes

### 3.3 Construction of prediction model

The foremost process of predicting the partition scheme is usually to obtain the sample set of thread-level speculation. When a new program is to be partitioned, its characteristics are extracted, and then its partition scheme is predicted from the sample set using the KNN method, in which the partition scheme of program to be partitioned is determined by the $k$ nearest similar samples.

The sample set can be formally represented as $T = \{X_{orig-i}, H_{par-i}\}$ ($i \in 1, 2, ...,$ N). In the sample set, every training sample composes of the characteristic vector $X_{orig}$ and the corresponding approximately optimal partition scheme $H_{par}$. Each characteristic vector $X_{orig} = [x_1, x_2, \ldots, x_n]$ corresponds to a point in the N-dimensional space. Every approximate optimal partition structure $H_{par}$ composes of a set of parameters, that are, $H_{par} = [h1, h2, h3, h4, h5]$, where $h1$ represents the lower limit of the thread granularity, $h2$ represents the upper limit of the thread granularity, $h3$ represents the data dependence number, $h4$ represents the lower limit of the spawning distance, and $h5$ represents the upper limit of the spawning distance.

When two procedures have the same characteristics, then they have the same partitioning scheme, which is a prerequisite for predicting the partitioning scheme. Based on the KNN-based partitioning scheme, we first need to build a predictive model. All the training samples $< X_{orig-i}, H_{par-i} >$ are stored in the

training samples to form a KNN prediction model. The KNN-based classification method simply stores all the training samples, and the classification of the samples is delayed until a new sample needs classification. When a new program needs to be partitioned, it first extracts the program' characteristics and then finds the K closest samples of it. The distance is calculated by the Euclidean distance, as shown in formula (1).

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

where $n$ denotes the dimensionality of a characteristic vector, $x_j^r$ and $x_i^r$, respectively, represent the $r_{th}$ attribute values of $x_j$ and $x_i$.

When we find $k$ nearest neighbor samples, in order to ensure the correctness of the prediction model, the class labels of input procedures are obtained by multiplying the class labels of the $k$ nearest neighbors and their weights and adding together to get the tag. The closer the procedure is to be partitioned, the greater the weight is given to it, and with the distance between the characteristic vectors increasing, the weights must be attenuated rapidly, but the total weight must be 1. In order to assign weights to the class labels of $k$ nearest neighbors, formula (2) is introduced, which is the Maclaurin series of exponential functions $e^x$.

$$e^x \approx 1 + x + \frac{x^2}{2!} + \cdots + \frac{x^n}{n!}, n \in N \tag{2}$$

When $x = 1$, then the formula (3) comes.

$$1 \approx \frac{2}{e} + \frac{1}{e * 2!} + \frac{1}{e * 3!} + \frac{1}{e * 4!} + \cdots + \frac{1}{e * n!}, n\epsilon N \tag{3}$$

Based on formula (3), we can deduce the distribution weight formula as (4).

$$weight(i) = \begin{cases} \frac{2}{e} & (i = 1) \\ \frac{1}{i! * e} & (i > 1) \end{cases} \tag{4}$$

Formula (4) indicates the sequence label of the procedure to be separated from the $k$ nearest neighbor samples. The greater the similarity distance is, the greater the weight is. As Olden benchmarks are used, and the number of sample set is not large, the value of $k$ is set to 10, the nearest 10 procedures close to the procedure which is to be partitioned are considered, and weights are summed to partition schemes of procedures which are to be partitioned.

Since the objective function $H_{par}$ is a thread partition scheme, its class label corresponds to five thresholds. For a procedure $x_q$ which is to be partitioned, its class label is the sum of all multiplications between the $k$ nearest neighbors and their respective weight values, as shown in equation (5).

$$\hat{h}_j(x_q) = \sum_{i=1}^{i=10} h_j(x_i) * weight(i) \quad (1 \le j \le 5) \tag{5}$$

where $\hat{h}_j(x_q)$ represents the $j_{th}$ threshold of the procedure to be partitioned, $h_j(x_i)$ represents the $j_{th}$ threshold of the $i_{th}$ procedure in the training samples. The $weight(i)$ is the weight formula of the $i_{th}$ procedure.

The prediction algorithm of the KNN-based partition scheme is shown in Table 2.

## 4 Thread partition

For a program to be partitioned, once the partition scheme has been obtained, then the partition scheme is used to partition the program into multiple threads. Thread partition is performed on the CFG (control flow graph) of a program. First of all, it is necessary to partition the loops of the program, as the loop partition is complicated. Because the loops can be divided into inner loops and outer loops, it is necessary to divide them so to facilitate their respective partitions. When the loops complete their partition, the next is to sum up all loops to superblocks, then CFGs of the original programs turn into SCFGs (super control flow graphs). After the most likely path in SCFG is selected according to the cost evaluation model, nonloops are partitioned. Finally, we need construct precomputation-slice for every thread that has been partitioned. The acceleration process of thread partition is shown in Fig. 6.

### 4.1 Cost evaluation model

When setting up a cost model, the assumption that there is an unlimited number of processor cores exists, so the problem of thread withdrawing due to a lack of processor cores need not be taken into account. In the Prophet compilation system, every instruction occupies only one clock cycle, so the execution time of the program segment can be expressed by the number of instructions contained in the segment. When a thread

**Table 2** K Nearest Neighbor Classifier

| |
|---|
| **Input:** training examples represented by $<X,H>$ ($X$ denotes feature vectors and $H$ represents a set of partition schemes), $x_q$ (characteristic vector of unseen program) |
| **Output:** $\hat{h}(x_q)$(predicted partition scheme for $x_q$) |
| k_nearest_neighbor ($<X,H>$, $X_q$) { |
| 1 for every training sample $<x, h(x)>$ do |
| 2 store the sample to the list *training_examples*; |
| 3 end for |
| 4 Let $x_1 \sim x_k$ be the $k$ nearest samples to $x_q$ from train sample; |
| 5 Let $weight(1) \sim weight(k)$ be the weights for the $k$ nearest neighbors($k \in N$); |
| 6 define an unseen vector $x_q$; |
| 7 for(int j=1;j≤5;j++){ |
| 8 $\hat{h}_j(x_q) \leftarrow \sum_{i=1}^{i=10}\{h_j(x_i) * weight(i)\}$; |
| 9 }; |

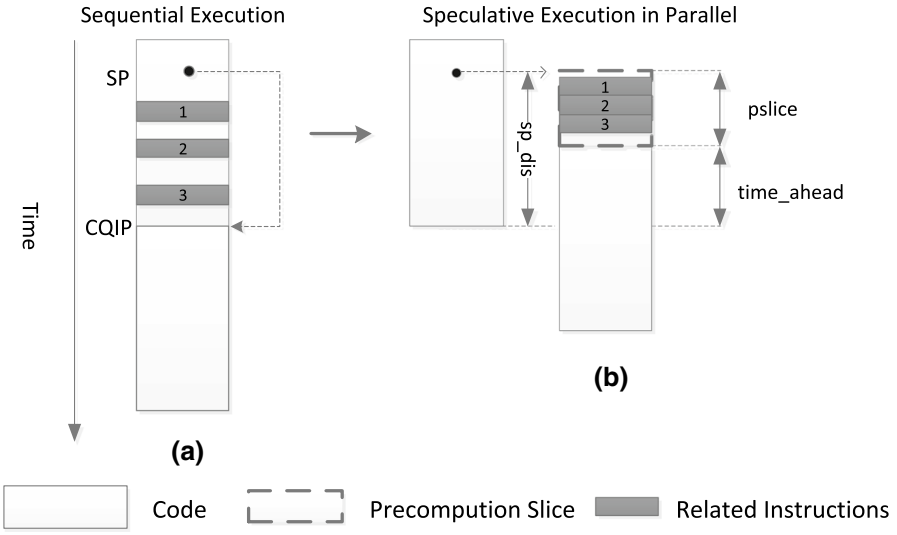



**Fig. 6** Effect of thread-level speculation

is executed, a new thread is spawned and there are three related instructions along the spawning path, as shown in Fig. 6. When two threads are speculatively executed in parallel, that is, the parent thread is executed on one processor core and its sub-thread is executed on another processor core, the sub-thread first needs to execute the precomputation-slice, whose length is expressed as *p-slice*. The distance from the parent thread to the sub-thread is *sp_dis*, and the number of related instructions contained in the spawning path is *dep_cnt*. Then, *p-slice* can be represented by *dep_cnt* + C, where C represents the overhead for creating a precomputation-slice. From Fig. 6, you can obtain the execution time of sub-threads, as shown in formula (6):

$$time\_ahead = sp\_dis - pslice = sp\_dis - dep\_cnt - C \tag{6}$$

Formula (6) can only show the local acceleration effect. If you want to get the global acceleration effect, you need to get the acceleration effect of each instruction, so you need to divide the advance time by the instruction number of speculative threads. That is, the evaluation definition of partition for nonloops is shown in Formula (7).

$$evaluate = \frac{sp\_dis - dep\_cnt}{thread\_size} \times w \tag{7}$$

where $w$ is the weight factor, *thread_size* is the instruction number of speculative threads, and *evaluate* is the evaluation value.

For the partition of nonloops, the features of every procedure are firstly extracted. Then, the machine learning method is used to predict the optimized partitioning scheme of every procedure in programs from the already built TLS sample set, and the predicted partition scheme consists of *LLoTG*, *ULoTG*, *DDC*, *LLoSD*, *ULoSD*. The partition scheme is a set of constraints on the candidate threads. During the process of program partition, the candidate threads need to satisfy the constraints

of thread granularity, data dependence and spawning distance. When the constraints are satisfied, the points with the maximum *evaluate* value during every partition are selected as the boundaries of the thread.

### 4.2 Loop partition

Before starting to partition the loops, loops are firstly induced into superblocks, as shown in Fig. 7. Every node in Fig. 7a represents a basic block, and nodes $5- > 6- > 8- > 7- > \cdots - > 5- > \cdots$ are a loop structure that can be induced to a supernode (i.e., gray node $5^{'}$). After the loop nodes are induced, this paper performs loop partitions. In order to fully exploit the parallelism of loops, loops are partitioned into several parts. In this paper, loops' partition is divided into two categories, namely loop nested partition and loop internal partition. For the nested iteration of loops, this paper uses sequential spawning to partition it, that is, the $i_{th}$ loop iteration spawns the $(i+1)_{th}$ loop iteration ($i \in \mathrm{N}$). If the loop contains a large number of instructions, it will be partitioned by use of nonloop partition approach. The algorithm of loop partition is present in Table 3.

Figure 8 is a diagram of loop partition, which is used to explain the partition algorithm of loops. If the loop body is greater than the upper limit of thread granularity *h2*, the loop is partitioned with nonloop partition approach. If it is a nested loop, the most likely path is found first, the optimal dependence *opt_ddc* is calculated, the number of instructions that express the inner loop is *size_of_loop*, and the spawning distance is *spawn_dis* ; If *opt_ddc* is less than the dependency threshold *h3* and *size_of_loop*

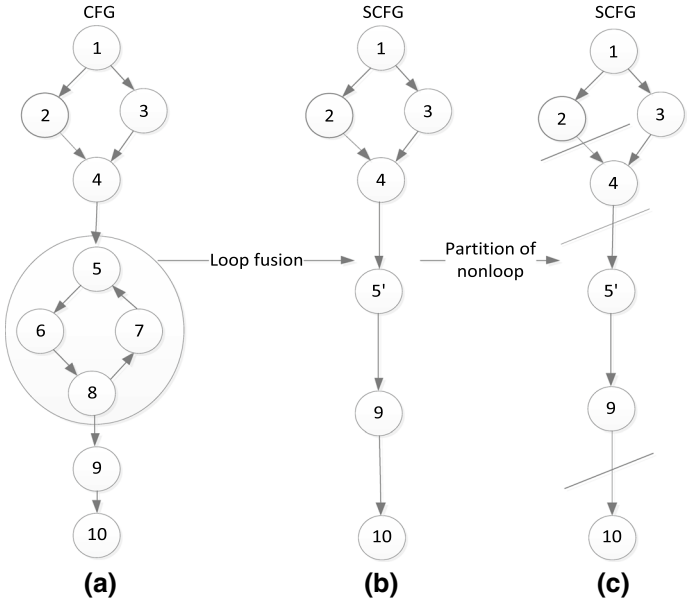



**Fig. 7** Progress of thread partition

**Table 3** Algorithm 1: Pseudocode of loop partition

| **Algorithm 1** Loop Partition |
|---|
| **Input:** Loop L |
| **Output:** Parallel threads set PTS |
| **Loop_Partition(loop *L*)**{ |
| define the starting block of $L$: *start_block*;<br>define the ending block of $L$: *end_block*;<br>define the lower limit of thread granularity: $H_1$;<br>define the upper limit of thread granularity: $H_2$;<br>define the lower limit of data dependence: $H_3$;<br>define the lower limit of spawning distance: $H_4$;<br>define the upper limit of spawning distance: $H_5$;<br>set the likely path starting from *start_block* to *end_block* to be *likely_path*;<br>define the *loop_size* with dynamic instructions' number in $L$;<br>define good dependence *opt_ddc* with *find_good_dependence*(*start_block*, *end_block*, *likely_path*,&*spawn_pos*);<br>while(*loop_size* $<= H_1$){<br>*unroll*($L$);<br>*update*(*loop_size*)}<br>if(($H_1 \leq$ *thread_size* $\leq H_2$)&($H_4 \leq$ *spawning_distance* $\leq H_5$)&(*opt_ddc* $< H_3$))<br>create a new thread *create_new_thread*(*end_block*, *spawn_pos*, *likely_path*),<br>and assign it to *curr_thread*<br>end if<br>record new thread *curr_thread* into PTS } |

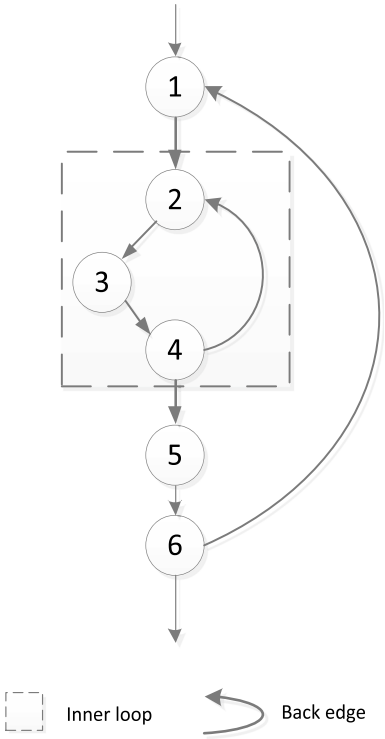
is within [*h1*, *h2*] and *spawn_dis* is within [*h4*, *h5*], a new thread is built. Figure 8 contains the inner loop and the outer loop, loop nested iterations are displayed in the dashed box, and the outer loops contain more instructions. In order to explore the potential parallelism of the program, ProCTA partitions loops in many points. If the body of the loop is too small, the loops need to be unrolled, and then the unrolled loops are partitioned according to the nonloop partition approach. It then identifies the back edges, induces the loops as a supernode, and examines data dependence among successive iterations in the loops. If it is beneficial to start a thread at the next iteration of the loop, a new thread is created at the next iteration of the loop.

The dashed box in Fig. 8 represents loops, which can be induced into a node, resulting in a starting node of loops with node 1 and an outer loop region with a back edge of $6- > 1$. If you remove the back edge $6- > 1$, you get a nonloop, which can use nonloop partition approach to perform the thread partition.

### 4.3 Nonloop partition

In this code (shown in Table 4), *curr_thread* represents the subgraph of the recent candidate thread, which is the set of basic blocks between the current thread's *start_block* and the last candidate thread's *end_block*. The subgraph in *curr_thread* cannot be used as a single thread because of too much data dependence or too small threads; *curr_thread* is empty if *start_block* coincides with the *end_block* of the previous thread. The post-dominator block of the *start_block* is represented with *pdom_block*, and the subgraph of speculative path between *start_block* and *pdom_block* is

**Fig. 8** Diagram of loop partition




represented by *path*. Thread partition refers to granularity and spawning distance of thread subgraph. According to the maximum threshold *h2* and the minimum threshold *h1*, the granularity of a thread subgraph is divided into three types: moderate granularity, large granularity, and small granularity. In Fig. 9, it is assumed that the critical path of thread partitioning is 0-1-3-4-5-6.

1. If the granularity of the current thread (0-1) is within the interval [*h1*, *h2*] and the spawning distance is within the interval [*h4*, *h5*] and the data dependence between it and the succeeding thread is not greater than the upper limit of data dependence *h3*, a new thread will be built, as shown in Fig. 9a.

2. If thread granularity of the current thread is too large, that is, the granularity is in the interval [*h2*, +), you need to partition program code between *start_block* and *pdom_block*. For the subgraph consisting of basic blocks 0, 1 and 2, if thread granularity of basic block 0 is within [*h1*, *h2*] and the data dependence count between basic block 1 and *future_thread* is less than *h3*, a new thread is built at the starting of basic block 1, as shown in Fig. 9b.

3. f the granularity of current thread is small, that is, the granularity is in the interval (0, *h1*], then the path and *pdom_block* are added to the subgraph of *curr_thread*, and program code between post-dominate nodes and *end_block* is analyzed and partitioned. Granularity and data dependence are met, a new thread is built at the beginning of basic block 5, as shown in Fig. 9c.

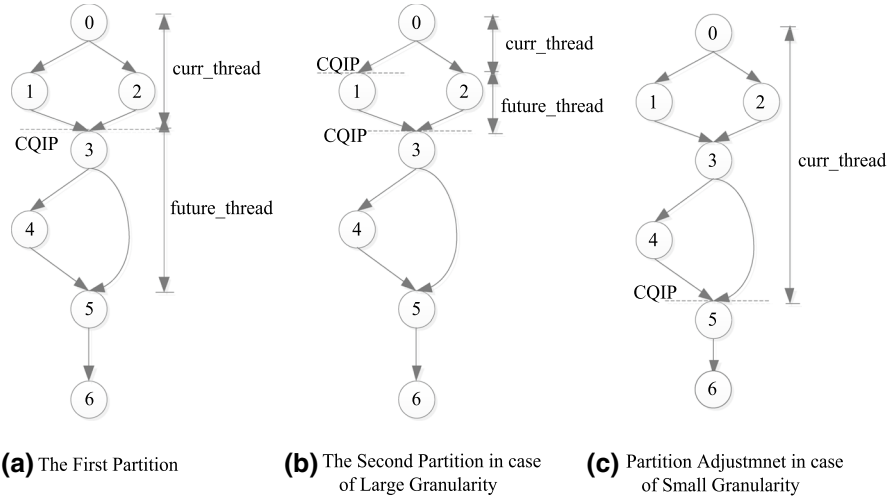



**Fig. 9** Diagram of nonloop partition

**Table 4** Algorithm 2: Pseudocode of nonloop partition

| **Algorithm 2** Non-loop Partition |
|---|
| **Input:** Non-loop $L$ |
| **Output:** *Curr_thread* |
| **Non-loop Partition(nonloop $L$)**{ |
| define the starting block of $L$: *start_block*;<br>define the ending block of $L$: *end_block*;<br>define the lower limit of thread granularity: $H_1$;<br>define the upper limit of thread granularity: $H_2$;<br>define the lower limit of data dependence: $H_3$;<br>define the lower limit of spawning distance: $H_4$;<br>define the upper limit of spawning distance: $H_5$;<br>set the likely_path starting from *start_block* to *end_block* to be *likely_path*;<br>set the nearest post dominator block of *start_block* to be *pdom_block*;<br>set the optimal dependence to be *opt_ddc*;<br>define the current thread to be *curr_thread*;<br>calculate the optimal dependence *opt_ddc* with *find_good_dependence*(*start_block*, *end_block*, *likely_path*,&*spawn_pos*);<br>if(*start_block*==*end_block*) then<br>return the value of *curr_thread*;<br>end if<br>if(($H_1$+0.25*($H_2$-$H_1$)≤*thread_size*≤$H_2$-0.25*($H_2$-$H_1$))&($H_4$≤*spawning_distance*≤$H_5$) &(*opt_ddc*≤$H_3$))<br>then create a new thread with function *create_new_thread*() in accordance with *start_block*,*end_block*, and the *likely_path*;<br>else if (($H_2$-0.25*($H_2$-$H_1$)≤*thread_size*≤$H_2$)&($H_4$≤*spawning_distance*≤$H_5$)&(*opt_ddc* ≤$H_3$)) then update *thread_size* with *curr_thread*+*path.first_block*;<br>if (($H_1$≤*thread_size*≤$H_1$+0.25*($H_2$-$H_1$))&(*opt_ddc*<$H_3$))<br>Partition threads in accordance with *first_block*,*end_block*, and *curr_thread*, and assign new thread to *curr_thread*;<br>end if end if<br>return *curr_thread*;} |

## 5 Experimental evaluation

In this section, the experimental setup is introduced, to provide details of the Prophet simulator as well as used benchmarks during the evaluation. In the last, we present the results' analysis and discussions.

### 5.1 Configuration of experiment

We perform the implementation of the execution model as well as thread partition algorithm on the platform: Prophet (its module chart is shown in Fig. 10), which is based on SUIF/MACHSUIF [28]. At the level of SUIF's intermediate representation (IR), we complete the compiler analysis. The profiling information is produced from SUIF-IR in the form of annotation by profiler of Prophet. The SUIF programs which are interpreted and executed by profiler provide information, including dynamic instruction number, prediction of control flow path, and prediction of data values. The Prophet simulator can simulate 1–64 pipelined mips-based R3000 processing elements (PE) and we run ProCAT with 4 PEs or 8 PEs. This simulating process is an execution-driven simulation, which performs the execution of binaries generated by Prophet compiler. Every PE fetches and executes instructions from one thread, and orderly issues 4 instructions per cycle. Every PE owns a private multiversioned L1 cache, which has latency of 2 cycles. Speculative results of PEs are buffered and cache communication is performed via multiversioned L1 caches. With a snoopy bus, a write-back L2 cache is shared by the 8 PEs. The parameter configuration of simulator is shown in Table 5.

Olden benchmarks [23] and SPEC2000 [20] are used to evaluate ProCTA. As a popular benchmarks of studying irregular programs, Olden benchmarks process complex control flows, pointer-intensive, as well as irregular data structures. The benchmarks own dynamic structures, e.g., trees, lists, and DAGs, et al, which are all difficult to get parallelized using conventional approaches.

ProCTA makes use of one leave-one-out cross-validation method to perform its results' evaluation. It means that the program which is to be partitioned is firstly moved from training set, and based on the left programs a prediction model is built. The method has an advantage that the prediction model never sees the programs to be partitioned before. The partition schemes for the left programs are built by

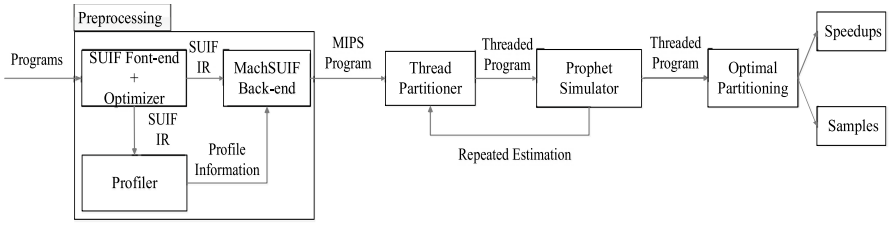



**Fig. 10** Module chart of Prophet

**Table 5** Configuration of Prophet simulation (per PE)

| Parameters of configuration | Value |
|---|---|
| Function units | 4 int ALU (1 cycle) |
| | 4 int Mult/Div (3/12 cycles) |
| | 4 fp ALU (2 cycles) |
| | 4 fp Mult/Div (4/12 cycles) |
| Spec. buffer size | Fully associative 2KB (1 cycle) |
| Bandwidth for Fetch,In order issue | 4 Instructions |
| and commit pipeline stages | Fetch/issue/Ex/WB/commit |
| Architectural registers | 32 int and 32 fp |
| L1-Cache (multiversioned) | 4-Way associative 64 KB (32B/Block) |
| | Hit latency 2 |
| | LRU replacement |
| L2-cache | 4-way associative 2MB (64 B/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 |
| $k$ | 5 |
| Similarity threshold | 0.5 |

applying the prediction model. Every program is performed with this process in turn.

The paper uses multi-version caches to solve memory dependence and uses register files to solve register data dependence.

### 5.2 Experiment results

In ProCTA, it firstly needs to extract the characteristics of every procedure in programs, and then learns the partition schemes in according to the built sample set. Finally, ProCTA applies the learnt partition schemes to partitioning programs. We use every procedure in Olden benchmarks as a sample to get the characteristic vector of every procedure by constructing characteristics of every procedure in the program and then use the explicit partition method based on expert experience to obtain the optimal partition scheme for every procedure, so that the sample is expressed in the form of "characteristic vector+ partition scheme". All the samples are constructed together to form a sample set.

In this paper, KNN algorithm is used to obtain the partition scheme of every procedure in the program to be partitioned. Because the validation set and the training set use the same sample set, a "leave-one-out" approach is used to validate the effectiveness of ProCTA. The so-called "leave-one-out" method is that when testing a sample, the sample can not be obtained directly from the sample set and can only be

estimated from other samples by machine learning. The following will take Olden's *mst* benchmark as an example to show how to predict the partition scheme of every procedure in the *mst* program (the number of inter-thread dependence, the lower limit of thread granularity, the upper limit of thread granularity, the lower limit of spawning distance, and the upper limit of spawning distance), which are shown in Table 6. Firstly, the sample of every procedures in the *mst* program is removed from the sample, avoiding the optimal partition scheme of the *mst* program. Then, the features of every procedure in the *mst* program are extracted and the predicted partition scheme of every procedure is learned from the sample set (Table 7).

Next, we will apply the predicted partitioning scheme to thread partition. For convenience, we use the procedure called *HashDelete* in the *mst* benchmark as an example to show the results of thread partition. According to the predicted partition scheme, the maximum data dependence count is 3, the lower limit of thread granularity is 9, the upper limit of thread granularity is 45, the lower limit of spawning distance is 4 and the upper limit of spawning distance is 20, as shown in Table 6.

### 5.3 Analysis of time complexity

The time spent in the ProCTA can be divided into two phases: training and prediction. During the training phase, a machine learning approach is used to learn the knowledge of thread partition, while the machine learning method is applied to predicting thread partition during prediction phase.

#### 5.3.1 Time complexity of training phase

In the worst case, selecting the $k$ ($k \in N$) nearest neighbors from the $n1$ samples needs to implement $n1$ comparisons of similarity, so the time complexity of training phase is $\Theta(n1)$, where $n1 \in N$.

**Table 6** Configuration of Prophet simulation (per PE)

| Name of procedure | Actual partition scheme | Predicted partition scheme |
|---|---|---|
| *HashLookup* | (3 9 35 3 20) | (3 6 43 4 20) |
| *BlueRule* | (3 9 30 3 20) | (3 8 37 4 20) |
| *ComputeMst* | (3 9 32 3 20) | (3 3 32 4 20) |
| *Dealwithargs* | (5 5 45 3 51) | (4 9 35 4 50) |
| *MakeGraph* | (4 9 35 3 45) | (8 9 42 4 40) |
| *AddEdges* | (3 7 43 3 20) | (3 5 34 4 20) |
| *HashInsert* | (3 9 36 3 20) | (3 5 37 4 20) |
| *MakeHash* | (3 9 31 3 20) | (3 3 45 4 20) |
| *HashDelete* | (4 8 40 3 20) | (3 9 45 4 20) |
| *Main* | (6 9 45 3 45) | (9 3 31 4 20) |

**Table 7** Partition diagram of *HashDelete* in *mst*

| **Source Programs before Partition** |
|---|
| void *HashDelete*(*int key*,*Hash hash*){ |
| HashEntry **ent*; |
| int *j* = (*hash*-> *mapfunc*)(*key*); |
| for (*ent*=&(*hash*->*array*[j]);(*ent) && (**ent*)->*key*!=*key*; *ent*=&((**ent*)-> next)); |
| *assert*(4,**ent*); |
| HashEntrytmp = **ent*; **ent* =(**ent*)->*next*; *localfree*(*tmp*);} |
| **MIPS Expression after Partition** |
| **Spawn HashDelete.L11** |
| la $sp,-152($sp) |
| ust |
| sw $fp,0($sp) |
| sw $ra,4($sp) |
| sw $s0,8($sp) |
| sw $s1,12($sp) |
| move $s0,$a0 |
| move $s1,$a1 |
| li $v1,4 |
| addu $t0,$s1,$v1 |
| lw $t0,0($t0) |
| **cqip HashDelete.L11** |
| HashDelete.L11: |
| **Pslice_entry HashDelete.L11** |
| li $v1,4 |
| addu $t0,$s1,$v1 |
| lw $t0,0($t0) |
| **Pslice_exit HashDelete.L11** |
| move $a0,$s0 |
| move $fp,$sp |
| fst $sp |
| jalr $t0,$ra |
| nop |
| move $t1,$zero |
| addu $t2,$s1,$t1 |
| lw $t3,0($t2) |
| lw $a2,0($t5) |
| li $a3,8 |
| lw $v0,0($t5) |
| la $v1,0($zero) |
| ... |
| lw $v1,0($t6) |
| la $t0,0($zero) |
| sne $v0,$v1,$t0 |

### 5.3.2 Time complexity of prediction phase

Once the nearest *k* samples are selected, the next step is to implement the prediction of partition schemes for unknown programs. The prediction process can be divided into two subprocedures: searching the partition positions and inserting partition instructions (i.e., spawning point (SP) and control quasi-independent point(CQIP)).

Of the two subprocedures, the time spent in inserting partition instructions can be ignored. Assume the number of partition instructions (sp and cqip) is $k$, then the time spent in inserting SP and CQIP is:

$$n \times (n-1) + (n-2) \times (n-3) + (n-4) \times (n-5) + \cdots + (n-k) \times (n-k-1) = \Theta(n^2)$$

As $n1 \ll n$, so the time complexity is $\Theta(n^2)$.

### 5.4 Performance comparisons and analysis

In the process of thread partition based on programs' characteristics, the first is to obtain the programs' characteristics, and then machine learning methods are used to obtain partition schemes (i.e., a set of thresholds) from sample set. Every procedure in the program has a different set of thresholds, and then this set of thresholds are used for thread partition, and finally the partitioned threads are implemented on the Prophet simulator with 4 cores to get the speedup of every program. Table 8 shows the comparison results among the heuristic rule-based (HR-based) partition approach, the expert experience-based (EE-based) partition approach, and ProCTA.

In order to show the effectiveness of ProCTA, this paper makes a comparison between ProCTA and HR-based thread partition. As EE-based thread partition is an explicit partition method, it can select the appropriate thread boundaries according to the control dependence and data dependence, and can adjust the positions of SP,CQIP several times, so can get better speedups than implicit partition. So in Table 8 the acceleration effect of EE-based thread partition approach is better than HR-based thread partition and ProCTA. But EE-based thread partition needs to manually select the thread boundary, so fully using this approach to partition all the serial irregular programs is basically impossible. Olden benchmarks set don't contain too many programs, so manually adjusting thread boundaries is feasible, we can use this kind of approach to obtain the better partition scheme of threads to construct TLS sample set. So, when we analyze the experimental results, we only compare the

**Table 8** Comparison results of speedups

| Olden benchmarks | Speedups (HR) | Speedups (EE) | Speedups (Pro) | Increase ratio (with HR) (%) |
|---|---|---|---|---|
| *bh* | 1.96693 | 2.21235 | 2.35125 | 19.54 |
| *em3d* | 2.60674 | 2.90956 | 2.95012 | 13.17 |
| *health* | 1.61705 | 1.88107 | 1.91244 | 18.27 |
| *perimeter* | 1.31302 | 1.53643 | 1.55243 | 18.23 |
| *voronoi* | 1.89448 | 1.95454 | 2.45089 | 29.37 |
| *treeadd* | 1.2461 | 1.53145 | 1.51012 | 21.19 |
| *power* | 2.08593 | 2.21734 | 2.35345 | 12.82 |
| *tsp* | 1.8205 | 1.97292 | 1.95123 | 7.18 |
| *mst* | 1.43446 | 1.58382 | 1.75238 | 22.16 |
| *bisort* | 1.27188 | 1.46015 | 1.62128 | 27.47 |
| *Mean* | 1.72571 | 1.92596 | 2.040559 | 18.24 |

performance of the original HR-based thread partition approach and ProCTA, and then we will analyze the experimental results, in which we only select several program analysis in the Olden benchmarks.

The main data structure in program *bh* is a heterogeneous *octree*, which has very complex data dependence. Its parallelisms exist in and out of loop structures. For the heuristic rules, the same partition scheme is used to partition all the procedures in the *bh* program, and for the ProCTA, the optimal partition scheme matching with the characteristic of every procedure in the program can be predicted, and then the partition scheme is applied to the threads. However, due to the existence of more dependence, ProCTA gains 19.54% performance improvement.

The main data structure of the program *em3d* is a single linked list, in which the loop structure occupies most of the total, and all the parallelism of program *em3d* comes mainly from the loop structure. Although ProCTA can obtain the partition scheme suitable for its own characteristics, the characteristic extraction of the loop is not enough. Finally, compared with the HR-based partition approach, 13.17% performance improvement is achieved.

The main data structure of the program *health* is a two-way linked list, which contains both loop and nonloop structure. In which, the loop structure is the main source of parallelism, and compared with the HR-based partition approach, you can obtain the partition scheme of *health* suitable for its characteristics. During the partition of loop partition, although the loops occupy most of the program, but it has a large loop body and simple data dependence, so *health* gets 18.27% speedup improvement.

The main data structure of program *perimeter* is four fork tree, the program primarily contains loop structure, rather than nonloop structure. The parallelism of program mainly comes from the decomposition of function into multi-threading. Because it is difficult to predict the return value of the function, the acceleration effect of these two approaches is not good. Compared with HR-based partition approach, ProCTA selects the suitable partition scheme in line with its own characteristics, and the partition scheme is not affected by loops. The assessment models adopted by nonloops are used to find the better thread partition boundary for the current program, so the final execution performance improves 18.23%.

The main data structure of program *treeadd* is two fork tree, which is a simple program structure. In this structure, only four procedures are included, and the program does not contain any loop structure, so the parallelism comes from the nonloops. ProCTA can select the appropriate partition scheme for every procedure, but there are many recursive function calls and data dependence in *treeadd*, and finally the program achieves 21.19% performance improvement.

The main data structure of the program *bisort* is two fork tree. Through the analysis of the source code, we can see that there are only three loops in the program, and only two loops are executed, and the granularity of the loop is relatively small. Then the parallelism of program is mainly from the nonloops, although the program has a certain number of data dependence, but mining the potential parallelism from the application program can be performed based on the ProCTA in every procedure. ProCTA selects the suitable partition scheme for every procedure, finally obtains 27.47% performance improvement.

Figure 11 shows the speedup comparisons between HR-based and ProCTA. Seen from Fig. 11, the speedups obtained by ProCTA in Olden benchmarks have a certain improvement than the speedups gotten by using HR-based thread partition approach. However, different programs have obvious differences in the speedup improvement. Overall, the HR-based approach obtains an average speedup of 1.725, while ProCTA gets an average speedup of 2.040, so the average speedup improves by 18.24%, indicating that ProCTA has a good effect on the program partition. Figure 12 shows the speedups of some SPEC2000 and Olden benchmarks on different number of cores.

In order to show the effectiveness of ProCTA, this paper compares the ProCTA with HR-based thread partition approach and EE-based thread partition approach. Because

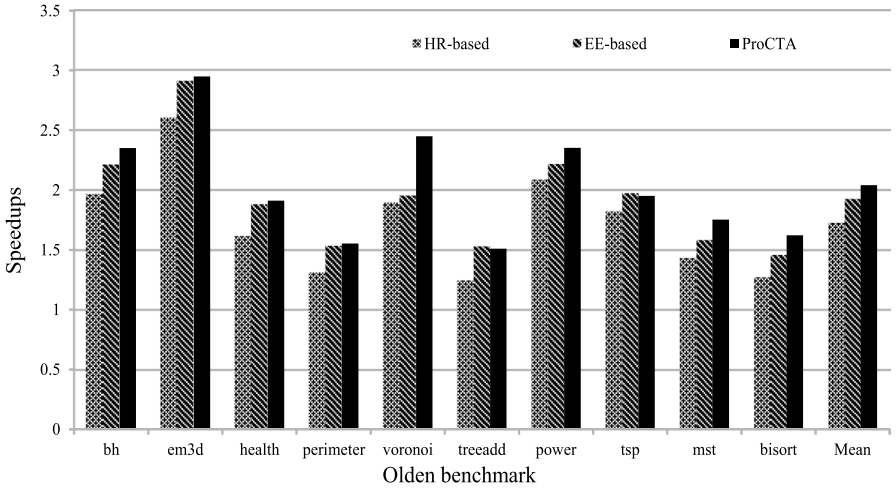



**Fig. 11** Comparison diagram of speedups for olden benchmarks

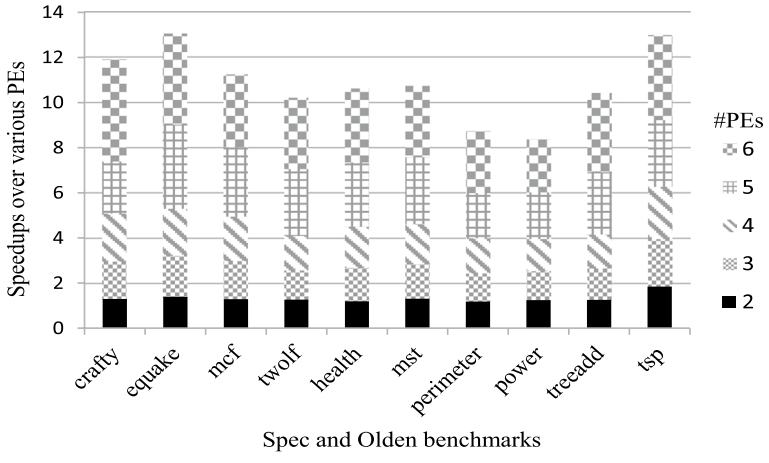



**Fig. 12** Speedup comparison diagram of olden and SPEC2000 benchmarks over different PEs

EE-based thread partition approach is an explicit partition approach, it can select the appropriate thread boundaries according to the control dependence and data dependence, and can adjust SP, CQIP points several times, getting better speedups than implicit partition, so speedups obtained by EE-based in Table 8 are better than HR-based and ProCTA. But EE-based thread partition approach needs to manually select the thread boundary, so it is basically impossible to partition all the serial irregular programs. Olden benchmarks contain not too much programs, thread boundary through manual annotation is feasible, we can use this kind of approach which is used to obtain better partition scheme of thread to construct TLS sample set. So, when we analyze the experimental results, we only compare the performance of the original HR-based approach with ProCTA, and then we will analyze the experimental results, in which we only select several programs in the Olden benchmarks to perform analysis.

## 6 Conclusion and future work

### 6.1 Conclusion

Based on the Prophet system, this paper proposes a program characteristic-based thread partition approach (ProCTA), and applies a machine learning method to thread partition. According to programs' characteristics, thread partition schemes are predicted from the sample set, and the programs are partitioned by the predicted partition scheme. Finally, the program is executed on the Prophet simulator to verify its execution performance. The research contents and conclusions are as follows:

#### 6.1.1 Construction of sample set

This paper proposes a method for constructing TLS sample set. TLS sample set are composed of characteristics and partition schemes. The characteristics are extracted from the speculative path of the control flow graph in every procedure of the program. For the acquisition of partition schemes in TLS samples, an explicit partition method based on expert experience is used to explicitly partition the program, and the optimal partition scheme of every procedure in the program is calculated.

#### 6.1.2 Thread partition based on programs' characteristics

This paper uses the KNN classification algorithm in machine learning to obtain a better partition scheme for every procedure, which is to be partitioned from the constructed TLS sample set.

#### 6.1.3 Prediction of partition scheme

This paper proposes an approach to partition programs based on predicted partition scheme. In this paper, the nonloop partition and loop partition are carried out, respectively. An evaluation model is established for the nonloops of the program, and the nonloops are iteratively partitioned according to the evaluation model. For the loops,

the nested iterations and the inner loops are, respectively, partitioned; for nested loops, sequential spawning and nested iteration are used for thread partition; for the inner loops, nonloop partition approach is adopted for thread partition.

#### 6.1.4 Validation using olden benchmarks

Using Olden benchmarks, ProCTA verifies its effect on Prophet system, and its effect is compared with the original partition results. The experimental results show that ProCTA obtains 18.24% speedup increasement than HR-based approach on average.

### 6.2 Future work

In the aspect of platform, the development of TLS is gradually developing from single machine multi-core component to distributed platform, such as parallelism of decompression algorithm with thread-level speculation in [26] on Spark platform. In order to overcome the limitation that simulated annealing algorithm (SA) is of low efficiency as the conventional SA algorithm still runs with low parallelism on new platforms and the computing resource cannot be fully utilized, Wang raised a speculative parallel SA algorithm [27] based on Apache Spark to expand the algorithm's parallelism and to enhance its efficiency. So the future work focuses on the parallelism of algorithms (i.e., crawling algorithm) on Hadoop or Apache Spark.

**Acknowledgements** We thank all members of Henan Joint International Research Laboratory of Cyberspace Security Applications for their great support and give our best hope to them for their collaboration. We also thank reviewers for their careful comments and suggestions. This work is supported by National Natural Science Foundation of China Grant Nos. 61772174 and 61370220, and Plan For Scientific Innovation Talent of Henan Province Grant No. 174200510011, as well as Program for Innovative Research Team (in Science and Technology) in University of Henan Province Grant No. 15IRTSTHN010. This work is also supported by National Key R&D Plan under Grant No. 2016YFE0104600.

## References

1. Carlisle MC (1996) Olden: parallelizing programs with dynamic data structures on distributed-memory machines. PhD thesis, Princeton University
2. Chen Z, Zhao YL, Pan XY, Dong ZY, Gao B, Zhong ZW (2009) An overview of prophet. In: International Conference on Algorithms and Architectures for Parallel Processing. Springer, pp 396–407
3. Codrescu L, Wills DS (1999) On dynamic speculative thread partitioning and the MEM-slicing algorithm, pp 40–46
4. Dong Z, Zhao Y, Wei Y, Wang X, Song S (2009) Prophet: a speculative multi-threading execution model with architectural support based on CMP. In: International Conference on Scalable Computing and Communications; Eighth International Conference on Embedded Computing. SCALCOM-EMBEDDEDCOM'09. IEEE, pp 103–108
5. Franklin M (1995) Multiscalar processors. ACM Sigarch Comput Archit News 23(2):414–425
6. Grewe D, Wang Z, O'Boyle MFP (2011) A workload-aware mapping approach for data-parallel programs. In: Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers. ACM, pp 117–126
7. Hammond L, Hubbert BA, Siu M, Prabhu MK (2000) The stanford hydra CMP. IEEE Micro 20(2):71–84

8. Johnson TA, Eigenmann R, Vijaykumar TN (2004) Min-cut program decomposition for thread-level speculation. In: ACM Sigplan Notices, vol 39. ACM, pp 59–70
9. Li D-C, Lin Y-S (2006) Using virtual sample generation to build up management knowledge in the early manufacturing stages. Eur J Oper Res 175(1):413–434
10. Li Y, Zhao Y, Sun L, Shen M (2017) Optimizing partition thresholds in speculative multithreading. ICIC Express Lett 11(6):1053–1061
11. Li Y, Zhao Y, Shi J (2016) A hybrid samples generation approach in speculative multithreading. In: 2016 IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS). IEEE, pp 35–41
12. Li Y, Zhao Y, Sun L, Shen M (2017) A hybrid sample generation approach in speculative multithreading. J Supercomput 3:1–33
13. Li Y, Zhao Y, Qiangsheng W (2017) Gba:a graph based thread partition approach in speculative multithreading. Concurr Comput Pract Exp 29(21):e4294
14. Liu B, Zhao Y, Li M, Liu Y, Feng B (2012) A virtual sample generation approach for speculative multithreading using feature sets and abstract syntax trees. In: 2012 13th International Conference on Parallel and Distributed Computing, Applications and Technologies. IEEE, pp 39–44
15. Liu B, Zhao Y, Li Y, Sun Y, Feng B (2014) A thread partitioning approach for speculative multithreading. J Supercomput 67(3):778–805
16. Liu B, Zhao Y, Li Y, Sun Y, Feng B (2014) A thread partitioning approach for speculative multithreading. J Supercomput 67(3):778–805
17. Liu W, Tuck J, Ceze L, Ahn W, Strauss K, Renau J, Torrellas J (2006) Posh: a TLS compiler that exploits program structure. In: Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. ACM, pp 158–167
18. Madriles C, García-Quiñones C, Sánchez J, Marcuello P, González A, Tullsen DM, Wang H, Shen JP (2008) Mitosis: a speculative multithreaded processor based on precomputation slices. IEEE Trans Parallel Distrib Syst 19(7):914–925
19. Ohsawa T, Takagi M, Kawahara S, Matsushita S (2005) Pinot: speculative multi-threading processor architecture exploiting parallelism over a wide range of granularities. In: Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture. IEEE Computer Society, pp 81–92
20. Prabhu MK, Olukotun K (2005) Exposing speculative thread parallelism in spec2000. In: Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. ACM, pp 142–152
21. Quinones CG, Madriles C, Sanchez J, Marcuello P, Gonzalez A, Tullsen DM (2005) Mitosis compiler: an infrastructure for speculative threading based on pre-computation slices. In: ACM Sigplan Notices, vol 40. ACM, pp 269–279
22. Quinones CG, Madriles C, Sanchez J, Marcuello P, Gonzalez A, Tullsen DM (2005) Mitosis compiler: an infrastructure for speculative threading based on pre-computation slices. In: ACM Sigplan Conference on Programming Language Design & Implementation
23. Rogers A, Carlisle MC, Reppy JH, Hendren LJ (1995) Supporting dynamic data structures on distributed-memory machines. ACM Trans Program Lang Syst 17(2):233–263
24. Song S, Zhao Y, Feng B, Wei Y, Wang X, Zhao H (2010) Prophet+: an extended multicore simulator for speculative multithreading. J Xian Jiaotong Univ 44(10):13–15
25. Steffan JG, Colohan C, Zhai A, Mowry TC (2005) The stampede approach to thread-level speculation. ACM Trans Comput Syst 23(3):253–300
26. Wang Z, Zhao Y, Liu Y, Chen Z, Lv C, Li Y (2017) A speculative parallel decompression algorithm on apache spark. J Supercomput 73(9):1–30
27. Wang Z, Zhao Y, Liu Y, Lv C (2018) A speculative parallel simulated annealing algorithm based on apache spark. Concurr Comput Pract Exp 1:e4429
28. Wilson RP, French RS, Wilson CS, Amarasinghe SP, Anderson JM, Tjiang SWK, Liao S-W, Tseng C-W, Hall MW, Lam MS et al (1994) SUIF: an infrastructure for research on parallelizing and optimizing compilers. ACM Sigplan Notices 29(12):31–37

**Publisher's Note** Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.