Frontiers of Information Technology & Electronic Engineering
www.zju.edu.cn/jzus; engineering.cae.cn; www.springerlink.com
ISSN 2095-9184 (print); ISSN 2095-9230 (online)
E-mail: jzus@zju.edu.cn

Optimization of thread partitioning parameters in speculative multithreading based on artificial immune algorithm*

Yu-xiang LI, Yin-liang ZHAO‡, Bin LIU, Shuo JI

(*Department of Computer Science and Technology, Xi'an Jiaotong University, Xi'an 710049, China*)

E-mail: liyuxiang@stu.xjtu.edu.cn; zhaoy@mail.xjtu.edu.cn; liubin2010@gmail.com; jishuo@stu.xjtu.edu.cn

Received May 13, 2014; Revision accepted Sept. 10, 2014; Crosschecked Jan. 28, 2015

Abstract: Thread partition plays an important role in speculative multithreading (SpMT) for automatic parallelization of irregular programs. Using unified values of partition parameters to partition different applications leads to the fact that every application cannot own its optimal partition scheme. In this paper, five parameters affecting thread partition are extracted from heuristic rules. They are the dependence threshold (DT), lower limit of thread size (TSL), upper limit of thread size (TSU), lower limit of spawning distance (SDL), and upper limit of spawning distance (SDU). Their ranges are determined in accordance with heuristic rules, and their step-sizes are set empirically. Under the condition of setting speedup as an objective function, all combinations of five threshold values form the solution space, and our aim is to search for the best combination to obtain the best thread granularity, thread dependence, and spawning distance, so that every application has its best partition scheme. The issue can be attributed to a single objective optimization problem. We use the artificial immune algorithm (AIA) to search for the optimal solution. On Prophet, which is a generic SpMT processor to evaluate the performance of multithreaded programs, Olden benchmarks are used to implement the process. Experiments show that we can obtain the optimal parameter values for every benchmark, and Olden benchmarks partitioned with the optimized parameter values deliver a performance improvement of 3.00% on a 4-core platform compared with a machine learning based approach, and 8.92% compared with a heuristics-based approach.

Key words: Speculative multithreading, Thread partitioning, Artificial immune algorithm
doi:10.1631/FITEE.1400172 **Document code:** A **CLC number:** TP311

1 Introduction

It is one of the definitive challenges in computer science and engineering over the last several decades to reduce the completion time of a single computational task. The primary means of reducing execution time, besides decreasing the clock period and the memory latency, has hinged on exploiting the inherent parallelism present in programs. Parallelism of programs has been a success for scientific applications, but not for nonnumeric applications. In the last decade, the emergence of the speculative multithreading (SpMT) model (Tsai and Yew, 1996; Akkary and Driscoll, 1998; Krishnan and Torrellas, 1999; Marcuello and González, 1999; Olukotun *et al*., 1999; Bhowmik and Franklin, 2002) has provided a much awaited breakthrough for nonnumeric applications. The hardware support that this model provides for speculative thread execution makes it possible for the compiler to parallelize sequential applications without being constrained by the data dependences and control dependences present in programs. When programs are parallelized at the thread level, it is necessary to preserve the dependencies between threads in order to guarantee the correct execution results. However, the dependences between threads are ambiguous because of the existence of the

‡ Corresponding author
* Project supported by the National Natural Science Foundation of China (No. 61173040) and the Doctoral Fund of Ministry of Education of China (No. 2013021110012)
ORCID: Yu-xiang LI, http://orcid.org/0000-0001-7271-3841

conditional branches within the program. Therefore, in conventional (non-speculative) multithreading, when a dependency is ambiguous, it is necessary to conservatively (and often too excessively) synchronize the related threads for (possibly useless) data communication, assuming that there exists a dependency between threads. This causes a problem that the execution performance is not improved or rather degraded because of the occurrence of the unnecessary waiting time. To solve the problem, speculative multithreading is considered to be quite effective, and various research on speculative multithreading has been performed (Sohi *et al.*, 1995; Tsai *et al.*, 1999; Bhowmik and Franklin, 2002; Quiñones *et al.*, 2005; Madriles *et al.*, 2009).

In speculative multithreading, a program is decomposed into multiple threads. The execution performance of the program is improved by predicting the next thread to be executed with the highest possibility, and by speculatively executing the predicted thread in parallel. In speculative multithreading, the execution time is shortened when the speculation succeeds. On the contrary, the execution time increases when the speculation fails. As we know, a program usually consists of many basic blocks. During the process of partition, the granularity, dependence distance, and spawning distance seriously affect the effect after its parallelism. How to effectively choose the optimal values for the thread size, dependence distance, and spawning distance is an issue which needs to be solved. Heuristic rules (parameters are listed in the following five items) exhibit a way to set their value ranges. Besides the heuristic rules, we are still able to optimize them using multi-objective optimization algorithms (Wang *et al*., 2000).

1. A spawning point (SP) can be anywhere in programs and as far as possible behind the function call instruction. Control quasi-independent points (CQIPs) are at the entrance of the basic block in the non-loop region. In the loop region, CQIPs are located in front of the loop branch instruction in the last basic block of the loop.

2. SP-CQIPs are located by the same procedure or in the same loop.

3. The dynamic instruction number between SP and CQIP must be greater than the minimum LOWER_LIMIT and less than the maximum UPPER_LIMIT.

4. Data dependences of two candidate threads must be less than the threshold DEP_THRESHOLD.

5. The number of function call instructions between SP and CQIP is less than CALL_LOWER.

2 Speculative multithreading

2.1 SpMT execution model

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

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

On Prophet, a spawning instruction pair is composed of an SP and a CQIP. The SP defined in the parent thread can spawn a new thread to execute speculatively the code segment behind the CQIP during program execution. The thread-level speculative model is shown in Fig. 1. The sequential program is mapped to an SP-CQIP, and the speculative multithreading program becomes a sequential program (Fig. 1a).

When an SP is found on program execution, the parent thread will spawn a new speculative thread and execute speculatively the code segment behind the CQIP (Fig. 1b). Validation failure or read after write (RAW) violations will lead to failures. When validation

fails, the predecessor thread executes the speculative thread in a sequential manner (Fig. 1c). When there is a violation in RAW dependence (Fig. 1d), the speculative thread restarts itself in the current state.

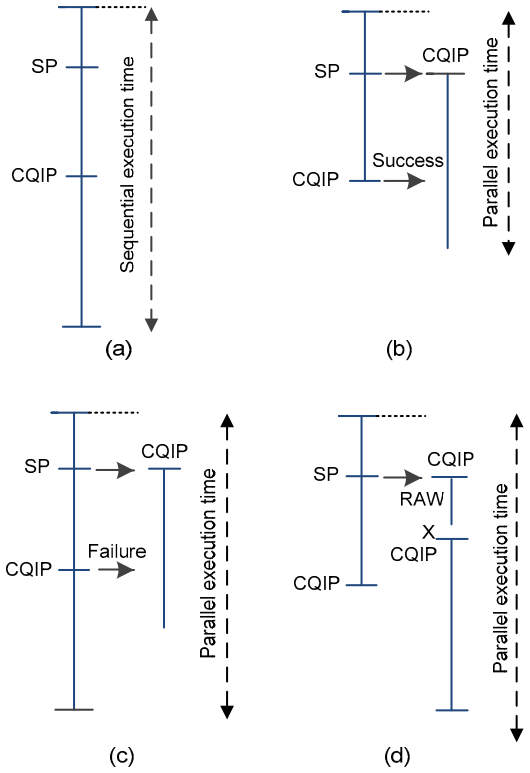

Fig. 1 Thread-level speculative model

(a) Sequential execution; (b) Parallel execution; (c) A failure of parallel execution; (d) RAW (X denotes RAW)

2.2 Pre-computation slices

In SpMT, the key is how to deal with inter-thread data dependences. Synchronization mechanism and value prediction have been applied so far. The synchronization approach imposes a high overhead when dependences are frequently and seriously affecting the parallel performance. Value prediction has more potential if the values computed by one thread and consumed by another can be predicted. The consumer thread can be executed in parallel with the producer thread since these values are needed only for validation at later stages. On the Prophet compiler, to reduce inter-thread dependences, the speculative *p*-slices (Quiñones *et al.*, 2005) are constructed and inserted at the beginning of each speculative thread. *p*-slices are used to calculate the live-ins of the new speculative thread but they do not need to guarantee their correctness, since the underlying architecture can detect and recover from miss-speculations. The *p*-slices are extracted from the producer thread at compile time but triggered at runtime to pre-fetch the live-ins. The steps to build the *p*-slices for a given spawning pair are: (1) identifying the live-ins produced on the speculative path, and (2) generating the optimal *p*-slices.

2.3 Data dependence calculation

Data dependence (Bhowmik and Franklin, 2002) includes the data dependence count (DDC) and data dependence distance (DDD). DDC is the weighted count of the number of data dependence arcs coming into a basic block from other blocks, while DDD between two basic blocks B_1 and B_2 models the maximum time that the instructions in block B_2 will stall for instructions in B_1 to complete, if B_1 and B_2 are executed in parallel.

Between DDC and DDD, we select DDC as the counted dependence criterion. DDC models the extent of data dependence a block has on other blocks. In Fig. 2, we give a description of data dependence between two blocks. The values of x and y in B_3 rely on the ones from B_1 and B_2. If the dependence count is small, then this block is more or less data independent of other blocks and we can begin a thread at the beginning of that basic block. While counting the data dependence arcs, the compiler gives more weight to the arcs coming from blocks that belong to threads closer to the block under consideration. The motivation is that dependences from distant threads are likely to be resolved earlier and hence the current thread is less likely to wait for data generated there.

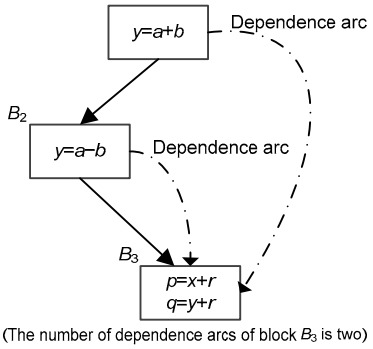

Fig. 2 Data dependence arcs between basic blocks

The dotted lines represent the dependence arcs

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

3 Artificial immune algorithm and its applications

3.1 Basic idea

The concept of immune (Dasgupta, 1999; Timmis, 2000; Wang *et al*., 2000; de Castro and Timmis, 2002) is inspired by biological natural science, and introduced with a new operator, namely the immune operator, based on the original framework of standard genetic algorithms. Similar to the immune theory in life science, the immune operator has two types, namely full immunization (full immunity) and target immune (target immunity), which correspond to the non-specific immunity in life sciences and specific immunity, respectively. For full immunization, each individual is dealt with in an immune operator after the genetic operator; for target immunity, individuals have immune operators at the point of action. The prior operation is primarily used for the initial stage of individual evolution, and does not play a role in the evolutionary process. Or it will produce the 'assimilative phenomenon'. The target immunity, which is accompanied by the group evolution process, is a basic operator of immune.

3.2 Basic concepts

The concepts in the algorithm (the specific process is not given in this paper) are listed as follows:

Definition 1 (Population) Population is a collection of individuals, which is a gathering of possible solutions.

Definition 2 (Individuals) Individuals are possible solutions to the problem. Correspondingly, the combinations of thresholds for every benchmark are mapped to be individuals.

Definition 3 (Antigen) Antigen is the problem to be solved.

Definition 4 (Antibody) Antibody is the solution to the handled problem. The optimal combination of partition parameters is the antibody.

Definition 5 (Vaccine) Vaccine is the best individual in a certain generation of the population. The best combination of thresholds in every turn is the vaccine.

Definition 6 (Vaccination) A process in which the corresponding genic bit of one individual is replaced with the one from the vaccine.

3.3 Data structure

We use a linear list to save five parameters (Fig. 3). These five parameters DT, TSL, TSU, SDL, and SDU represent the dependence threshold, lower limit of thread size, upper limit of thread size, lower limit of spawning distance, and upper limit of spawning distance, respectively.

Note that we use binary codes to express the linear structure (Fig. 4), so that they can be easily operated in the algorithm (which will be introduced later). We use eight binary bits to express every parameter, which ranges from 0 to 255.

Fig. 3 Data structure

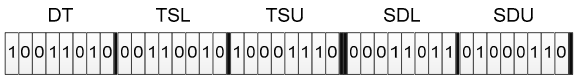

Fig. 4 Binary linear structure for the five parameters

3.4 Objective function

Similar to other parallel algorithms, our objective function takes speedup as the sole objective:

$$Y=\text{speed\_up}(\text{DT}, \text{TSL}, \text{TSU}, \text{SDL}, \text{SDU}), \quad (1)$$

where Y denotes the output of the objective function, speed_up() is the speedup function, which is determined by the five parameters, i.e., DT, TSL, TSU, SDL, and SDU. As we know, speedup is originally expressed by the parallel time (T_{para}) and serial time

(T_{ser}), where T_{para} is correlated with the five parameters. Thus, we can deduce Eq. (1) from

$$\text{speed\_up} = \frac{T_{ser}}{T_{para}}. \tag{2}$$

3.5 Algorithm model

During the specific operational processes in the artificial immune algorithm, the solved problem is firstly analyzed, and the basic feature information (vaccine) is extracted. Then the feature information is handled and converted to a scheme. The set obtained in accordance with this scheme is collectively referred to as the antibody. Our primary work is to find the antibody, i.e., the specific solution to the issue.

3.5.1 Vaccination

Assume x and y are two individuals (x and y have the data structure as illustrated in Fig. 3), and x (to be vaccinated) is changed in certain bits, to reach high fitness with a high probability. Two cases are taken into consideration. In the first case, assume that the information in every bit of y is false (that is, every bit is different from the one in the best individual). Then the probability for x to be converted to y is 0. In the second case, assume that every bit of the gene is correct (that is, x is already the optimal individual); thus, the probability for x to be transformed to itself equals 1. In addition, assume a population $C=(x_1, x_2, \ldots, x_n)$. Population C being vaccinated means that a number of individuals ($n\_\alpha=\alpha n$, $0<\alpha\leq 1$) among C are extracted and operated. Vaccines are extracted from prior knowledge for issues, and their contained information and accuracy play an important role in the performance of the algorithm.

3.5.2 Immune selection

The first step is immune detection, i.e., detecting the vaccinated individuals. If the fitness of a vaccinated individual is less than its parents' fitness, which denotes that serious degradation occurs during the process of crossover and mutation, the individual will be replaced by its parents. If the fitness of a vaccinated individual is higher than its parents' fitness, then the above operation will be done. The second step is annealing selection (Wang *et al*., 2000); i.e., the individual x_i is selected from progeny groups $E_k=(x_1, x_2, \ldots, x_n)$ to enter a new species with the following probability:

$$P(x_i) = \frac{\exp(f(x_i)/T_k)}{\sum_{i=1}^{n_0}\exp(f(x_i)/T_k)}, \tag{3}$$

where $f(x_i)$ is the fitness of x_i and T_k is an element of a temperature control sequence $\{T_k|k=1, 2, \ldots\}$.

3.5.3 Algorithm flow

1. Initialization is the process of initializing the population. The population is denoted with C, and is expressed as

$$c=(x_1, x_2, \ldots, x_n), \quad n\geq 1, n\in\mathbb{N}, \tag{4}$$

where individual x_i has the data structure as expressed in Fig. 3.

2. Fitness calculation is to calculate the fitness of individuals using an objective function. In this study the fitness calculation is terminated with the assistance of the objective function. Note that T_{para} and T_{ser} are obtained from our experiment platform Prophet (Fig. 5).

Fig. 5 shows a thread spawning process and the corresponding time statistics process. The serial execution time is

$$T_{ser} = \sum_{i=0}^{3} t_i, \tag{5}$$

where t_i (i=0, 1, 2, 3) is the execution time of the ith thread. The parallel execution time T_{para} is obtained from the statistics of Prophet. Moreover, T_{ser} does not contain overheads of p-slice information, overheads of thread validation and data submission, as well as overheads of dependence violation detection.

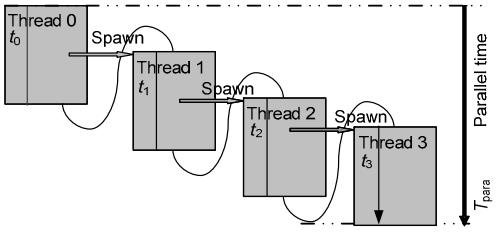

Fig. 5 Statistic model of Prophet

t_0 is the main thread and t_1–t_3 are speculative threads. Their control dependence is expressed with serial speculation

3. Condition judgment is to judge whether the result of the objective function has reached the desired criteria. If the criteria have been reached, go to the next step; otherwise, stop. The criteria we set contain two kinds: the first is that our objective function values stay almost unchanged after certain iterations; the second is that the iteration number has reached a fixed value.

As to crossover operation, we use a single-point crossover and set the crossover rate to 0.6. Similarly, the mutation rate is set to 0.01.

4. Immune operators involve two aspects, vaccination and immune selection, which have been mentioned and explained above.

$$\begin{cases} x_1 = [01010101 0.....01], \\ x_2 = [11011101 1.....01]. \end{cases} \tag{6}$$

During the mutation process, a seldom bit of individual x_1 switches with the corresponding bit of another random individual x_2:

$$\begin{cases} x_1 = [0101011 10.....01], \\ x_2 = [0101010 10.....01]. \end{cases} \tag{7}$$

During the mutation process, a seldom bit of individual x_1 changes to the opposite value (0 or 1).

3.5.4 Solution space

Assume parameters DT, TSL, TSU, SDL, and SDU have upper and lower limits. DT_{max}, TSL_{max}, TSU_{max}, SDL_{max}, and SDU_{max} represent their upper limits, and DT_{min}, TSL_{min}, TSU_{min}, SDL_{min}, and SDU_{min} represent their lower limits, satisfying

$$\begin{cases} DT \in [DT_{min}, DT_{max}], \\ TSL \in [TSL_{min}, TSL_{max}], \\ TSU \in [TSU_{min}, TSU_{max}], \\ SDL \in [SDL_{min}, SDL_{max}], \\ SDU \in [SDU_{min}, SDU_{max}], \\ TSU > TSL, \\ SDU > SDL. \end{cases} \tag{8}$$

From Table 1 and Eq. (8), we know that the solution space (10^{20}) is large. It is difficult for conventional methods to find the optimal solution. The artificial immune algorithm (AIA), as a novel intelligent algorithm, has its specific advantages in dealing with the issues whose solutions are diversified in a large space.

Table 1 Configurations of Prophet simulation

Optimized parameter	Value range	Step length
DT	1–10 000	1
TSL	1–10 000	1
TSU	(TSL+1)–10 000	1
SDL	1–10 000	1
SDU	1–10 000	1

Value range and step length are set empirically, rather than fixed

Fig. 6 shows the optimization flowchart with AIA. It comprises two main parts. The left part shows the flowchart of AIA, while the right one indicates the optimization process of AIA. Initial populations are built through initialization of DT, TSL, TSU, SDL, and SDU to establish a solution space. Vaccine extraction corresponds to the extraction of DT, TSL, TSU, SDL, and SDU. We calculate the object function to evaluate the fitness of individuals. Then we use crossover, mutation, and selection operations to iteratively optimize solutions and select the optimal one. The optimal solution corresponds to the optimal combination of thresholds.

4 Experiment

4.1 Simulation environment

We use Prophet (Dong *et al.*, 2009), which is configured in Table 2, to partition threads with thresholds and Matlab to run AIA. We realize the combination of them, so that we can partition

Table 2 Parameters list and value setting

Configuration item	Item value
Fetch, in-order issue, and commit bandwidth	4 instructions
Number of processing units	4
L1-cache (share)	4-way associative 64 KB hit latency: 2
Spawn overhead	5 cycles
Validation overhead	15 cycles
Local register	1 cycle
Access memory	5 cycles
Commit overhead	5 cycles

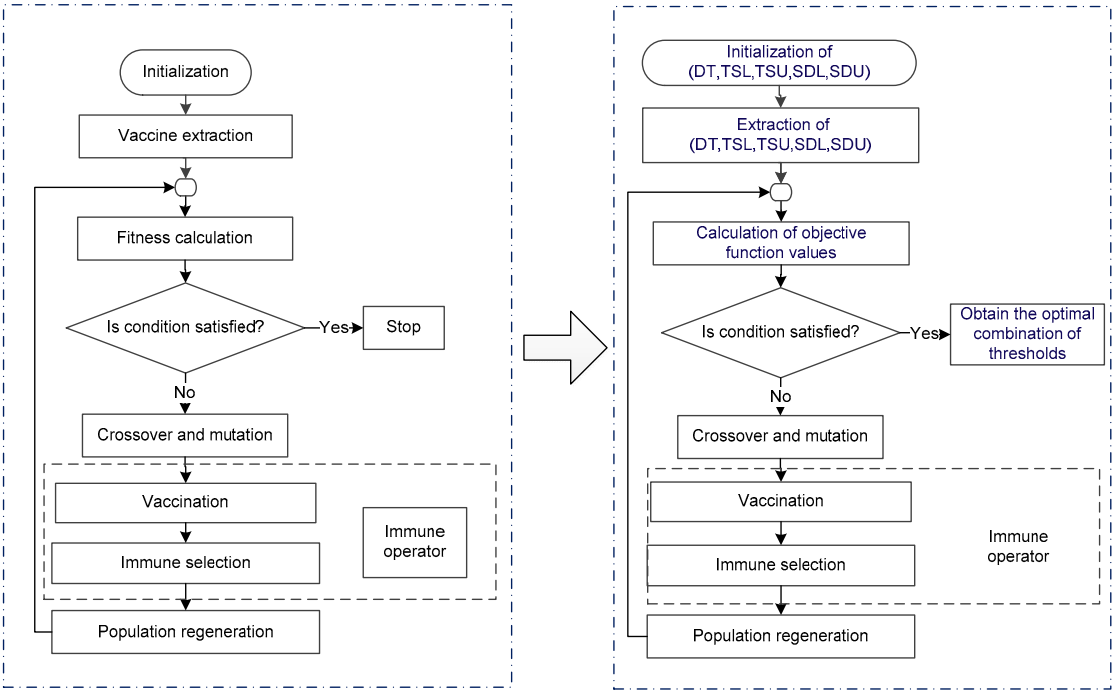

Fig. 6 Flowchart of optimization with the artificial immune algorithm

programs with specific thresholds during the optimization.

We have implemented the algorithm and searching processes on Matlab (R2010a) and the Prophet compiler (Chen *et al.*, 2009; Dong *et al.*, 2009) based on SUIF/MACHSUIF (Wilson *et al.*, 1994). The Prophet simulator (Dong *et al.*, 2009) models a generic SpMT processor to evaluate our SpMT model. Each processing element (PE) has its own program counter, fetch unit, decode unit, and execution unit that can fetch and execute instructions from a thread. In our evaluation, we assume that the architecture is fast enough so that spawning and squashing can be finished in one cycle. The simulator uses the MIPS ISA (Heinrich, 1994). The library code is not parallelized while the serial execution of which provides a conservative treatment to our parallelism values. For each input parameter, we observe the trend of the real speedup obtained with the Prophet simulator for an increasing number of processors. Until the speedup value is no longer increased, we obtain the maximum speedup and take it as the real speedup for the corresponding input parameter. The output of our cost estimation model is an estimate of the real value of the speedup.

4.2 Simulation design

4.2.1 Flowcharts of experimental results for Olden benchmarks vs. variable optimized parameters

Fig. 7 shows the parameter values for each experiment. During the starting period of experimental design, we deliberately set the step length of every parameter for one, just for careful observation of experimental results. After the 6th experiment, we add the parameter values with a random step length. In the beginning, DT, TSL, TSU, SDL, and SDU are assigned {8, 15, 38, 5, 33} empirically. Then they are increased with one in every step during the first six times, while the five parameters change

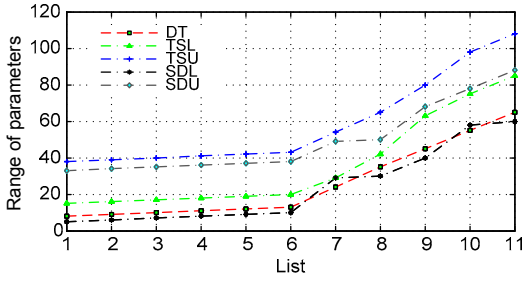

Fig. 7 Ranges of parameters (DT, TSL, TSU, SDL, SDU) of Olden benchmarks

randomly in the next five times. Fig. 7 gives only the changing process for these five parameters. In the next step we will take into consideration effects based on their changing processes. Accordingly, the initial individual is expressed in Fig. 8.

Fig. 9 illustrates the experimental results of Olden benchmarks corresponding to 10 combinations of parameters (DT, TSL, TSU, SDL, SDU).

In the Prophet simulator, we simulate four processors. If CPU_i is squashed, its failure count is added with one. The 5th index is speed_up, which is the primary experiment result. The 6th index is success_ratio, which denotes the successive spawning rate. In this study, we consider more about speedup for every benchmark, and less for the other five indexes. However, we give other experimental indexes just to show more information to indicate that threshold modification can affect experimental results in many ways.

Note that the 10 collections of thresholds are shown in Table 3. Each row, except the 1st row, is founded on the above one. Every threshold value is later changed with a random increment.

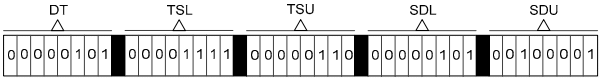

Fig. 8 Expression of the initial individual

4.2.2 Threshold distribution

Fig. 10 shows a curve between speedups and corresponding combinations of thresholds for benchmark em3d. Combinations of thresholds include five elements, namely DT, TSL, TSU, SDL, and SDU. These five elements form a five-dimensional space. We aim to find the maximum speedup by running every benchmark through searching this space. During the process of obtaining the speedups, we pick

Table 3 Collection of thresholds corresponding to Fig. 7

List	Collection of parameters
1	{DT=8; TSL=15; TSU=38; SDL=5; SDU=33}
2	{DT=9; TSL=16; TSU=39; SDL=6; SDU=34}
3	{DT=10; TSL=17; TSU=40; SDL=7; SDU=35}
4	{DT=11; TSL=18; TSU=41; SDL=8; SDU=36}
5	{DT=12; TSL=19; TSU=42; SDL=9; SDU=37}
6	{DT=13; TSL=20; TSU=43; SDL=10; SDU=38}
7	{DT=24; TSL=29; TSU=54; SDL=29; SDU=49}
8	{DT=35; TSL=42; TSU=65; SDL=30; SDU=50}
9	{DT=45; TSL=63; TSU=80; SDL=40; SDU=68}
10	{DT=55; TSL=75; TSU=98; SDL=58; SDU=78}

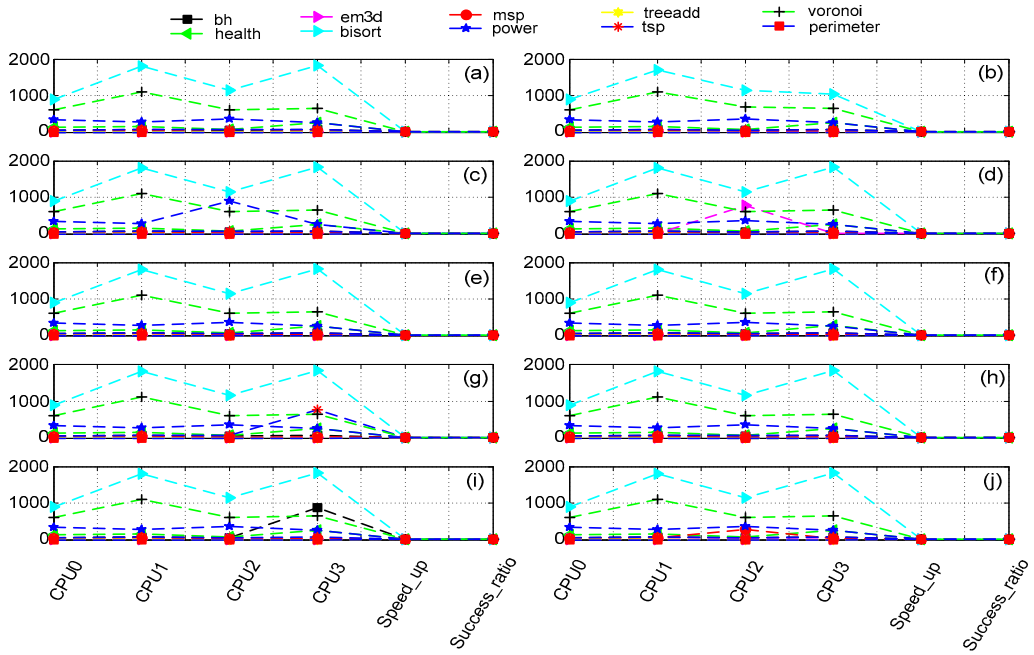

Fig. 9 Experimental results of Olden benchmarks corresponding to 10 combinations of parameters (DT, TSL, TSU, SDL, SDU)

(a)–(j) correspond to the collections of parameters from list 1 to list 10 in Table 3. The vertical coordinates represent the numbers of failures of CPU0, CPU1, CPU2, CPU3, speedup, and successive spawning rate, respectively

up 10000 combinations of thresholds. These inputs are randomly picked from the sample space, built by varying every threshold from minimum to maximum with the step length of one. The benchmark reaches the maximum speedup while running the 9648th combination. Using the values of the 9648th combination to set the initial thresholds, we can obtain an optimal speedup 2.7897 for em3d.

Figs. 11a and 11b are both a random distribution of five thresholds. Different from Fig. 11a, Fig. 11b displays thresholds within the lower and upper limits. As the solutions in the solution space are randomly distributed, we build the initial solutions in accordance with this principle. Fig. 11 shows the specific built solutions. Five kinds of points are contained in this figure, and they add up to 800. To do so, we divide it into three steps: First, set the initial ranges for every threshold based on Table 1. Then use a seldom number generation function to generate threshold values. Finally, check every row of thresholds, and delete the illegal values.

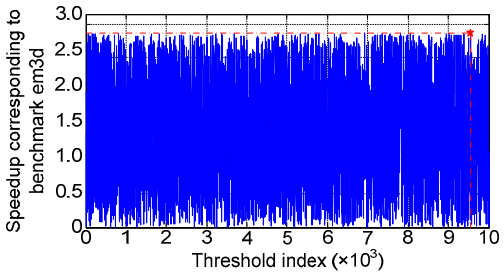

Fig. 10 Speedups of em3d corresponding to collections of thresholds (DT, TSL, TSU, SDL, SDU)

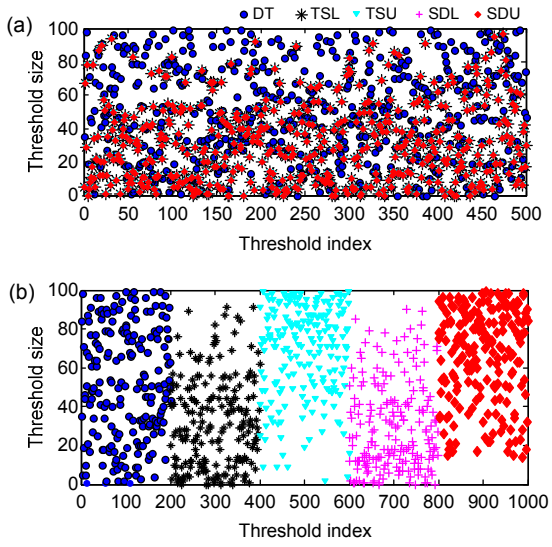

Fig. 11 Random distributions of 1000 parameters (DT, TSL, TSU, SDL, SDU)

(a) Mixed points of five parameters (DT, TSL, TSU, SDL, SDU); (b) Points separately shown in different sections

4.2.3 Solution search

Based on the generated threshold space, we use our experimental platform and artificial immune algorithm to search for the optimal threshold combination for every benchmark. We set the iteration number to be 10000, and the iteration will not be terminated until the ideal speedup is found.

Figs. 12 and 13 show that benchmarks bh and health obtain their own largest speedups, while running the 2564th and 9648th combinations of thresholds, respectively. Using values of the 2564th and 9648th combinations to set the initial thresholds, we can obtain optimal speedups 2.0897 and 1.6212 for them, respectively. In addition, Fig. 13 shows the speedup of the benchmark health. The benchmark health obtains the largest speedup 1.3962 while setting the 768th combination of thresholds.

Fig. 14 gives an overview of speedups for the benchmark perimeter. The function obtains the largest

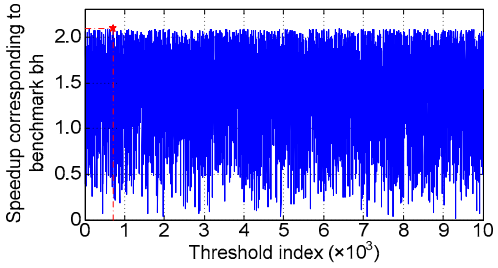

Fig. 12 Speedups of bh corresponding to collections of thresholds (DT, TSL, TSU, SDL, SDU)

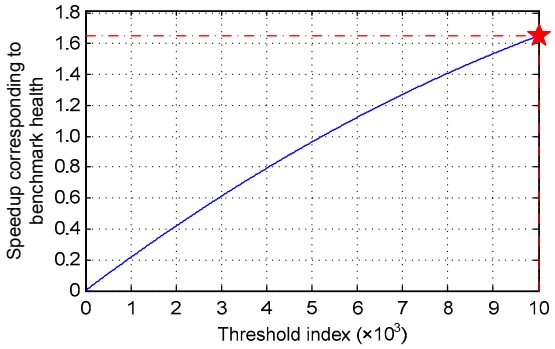

Fig. 13 Speedups of health corresponding to collections of thresholds (DT, TSL, TSU, SDL, SDU)

speedup 1.3900 while setting the 5000th combination of thresholds.

Similarly, Fig. 15 gives an overview of speedups for the benchmark voronoi. The function obtains the largest speedup 1.8001 while setting the 5566th combination of thresholds. Fig. 16 gives an overview of speedups for the benchmark treeadd. The function obtains the largest speedup 1.5605 while setting the 9846th combination of thresholds. Fig. 17 gives an overview of speedups for the benchmark power, and the function obtains the largest speedup 1.56 while setting the 5546th combination of thresholds. Fig. 18 gives an overview of speedups for the benchmark tsp. The function obtains the largest speedup 1.80 while setting the 506th combination of thresholds. Fig. 19 gives an overview of speedups for the benchmark msp. The function obtains the largest speedup 1.47 while setting the 2342nd combination of thresholds. Fig. 20 gives an overview of speedups for the benchmark bisort. The function obtains the largest speedup 1.4054 while setting the 3598th combination of thresholds.

Note that we obtain the optimal speedup for every benchmark, but the initial combinations of thresholds at every turn are all different and random. During the process of searching for the optimal speedup for a benchmark, we aim to find the best

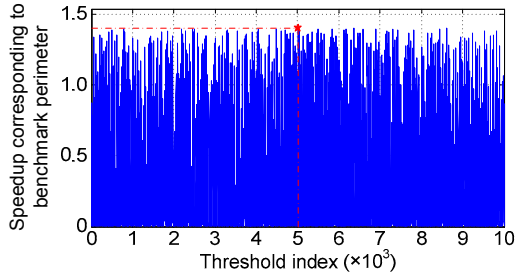

Fig. 14 Speedups of perimeter corresponding to collections of thresholds (DT, TSL, TSU, SDL, SDU)

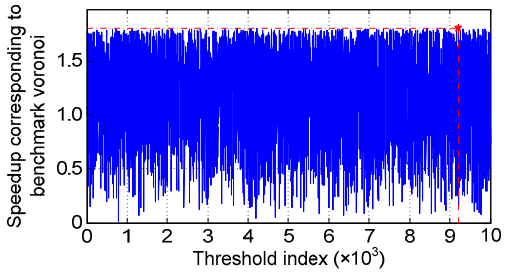

Fig. 15 Speedups of voronoi corresponding to collections of thresholds (DT, TSL, TSU, SDL, SDU)

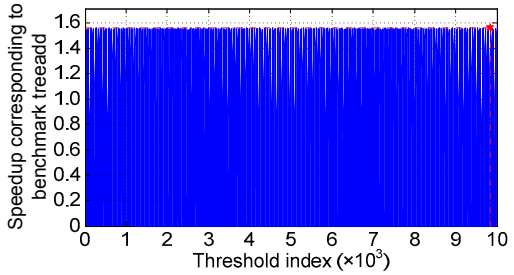

Fig. 16 Speedups of treeadd corresponding to collections of thresholds (DT, TSL, TSU, SDL, SDU)

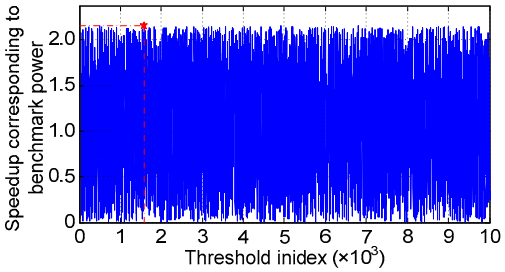

Fig. 17 Speedups of power corresponding to collections of thresholds (DT, TSL, TSU, SDL, SDU)

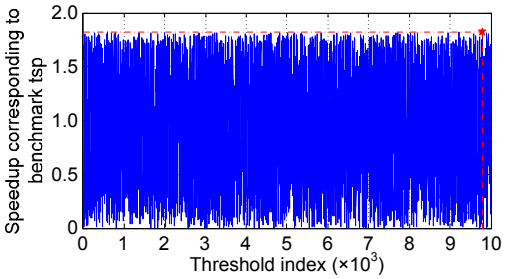

Fig. 18 Speedups of tsp corresponding to collections of thresholds (DT, TSL, TSU, SDL, SDU)

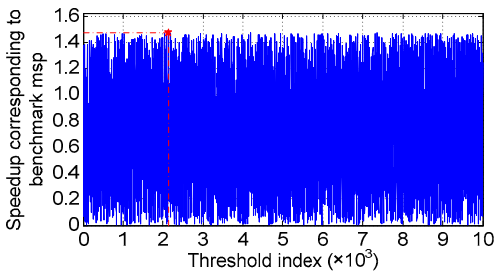

Fig. 19 Speedups of msp corresponding to collections of thresholds (DT, TSL, TSU, SDL, SDU)

combinations in the random distribution space of thresholds.

Table 4 shows the optimal combinations of thresholds. Every row shows the best combination of five thresholds for every benchmark. For every function, we enquiry the table, and set the values of thresholds over every function with the corresponding value. Thus, we can obtain better speedups for them.

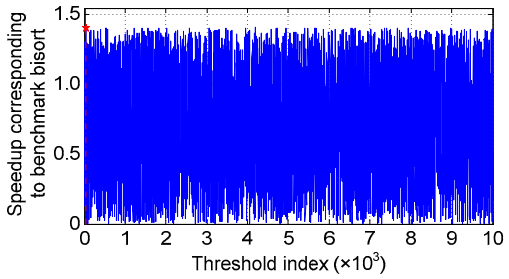

Fig. 20 Speedups of bisort corresponding to collections of thresholds (DT, TSL, TSU, SDL, SDU)

Table 4 The optimal combinations of parameters

Benchmark	DT	TSL	TSU	SDL	SDU
em3d	7	10	48	9	39
bh	7	16	48	4	32
health	10	17	34	4	31
perimeter	4	26	49	6	31
voronoi	5	16	46	1	30
treeadd	4	21	44	6	39
power	8	15	44	2	43
tsp	4	16	42	8	31
msp	12	23	39	4	38
bisort	6	19	40	8	45

4.2.4 Comparison of experimental results

Fig. 21 shows the speedups of the different approaches on the 4-core platform. Compared with the heuristic- and ML-based approaches, most of the programs gain higher speedups in different percentages and the increasing rates of performance improvement vary from 25% to −8.0%. In particular, programs em3d, bh, and treeadd have gained significant performance improvement. These surprising performance gains can be attributed to two important factors. First, our approach provides the optimal thread partitioning thresholds for every program according to optimized combinations. Second, because these irregular programs have more inherent parallelism, the parallelism can be adequately exploited in the case of enough core resources. However, in Fig. 22, programs health and perimeter exhibit a decrease, and voronoi shows a zero growth.

Although our approach can find the best combination of thresholds in the random distribution threshold space for these programs, they have complex control and data dependency. Also, we can never pick out all points in the space. Even if processor resources are sufficient and computing speedup is

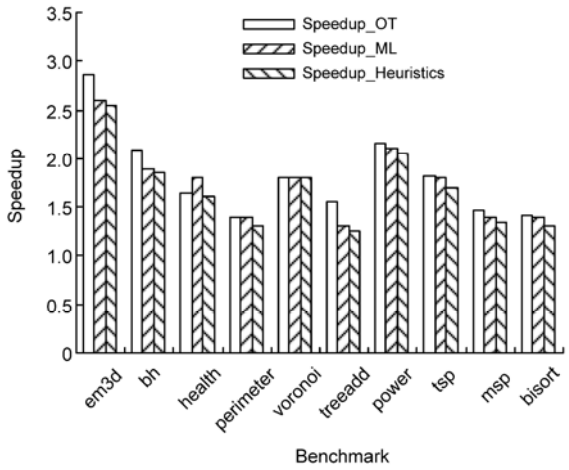

Fig. 21 Comparison of experiment results among Speedup_OT, Speedup_ML, and Speedup_Heuristics
Speedup_OT: speedups obtained by optimizing thresholds; Speedup_ML: speedups obtained with the method of machine learning; Speedup_Heuristics: speedups obtained by heuristics. Speedup_ML and Speedup_Heuristics are referred to the literature (Liu *et al*., 2014)

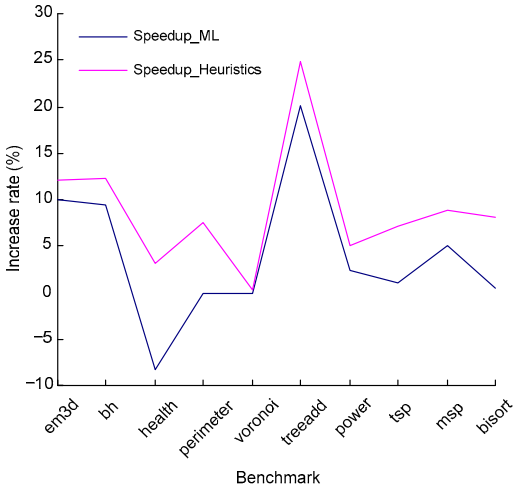

Fig. 22 Increase rates over machine learning based and heuristics-based approaches

infinite, we are not able to find the most precise thresholds for all the programs.

5 Conclusions

In this paper, we have proposed and evaluated a novel parameter optimizing approach for thread partition in SpMT based on an artificial immune algorithm. Multithreads partitioned with optimized parameters can better reach an improvement than the ones partitioned with parameters from heuristic rules. To obtain much higher speedup, we treat parameter optimization as a single-objective optimization problem. First, we extract five parameters, including the dependence threshold (DT), lower limit of thread size (TSL), upper limit of thread size (TSU), lower limit of spawning distance (SDL), and upper limit of spawning distance (SDU). Second, we set value ranges for every parameter based on heuristic rules. Third, we set the step size for every parameter. Finally, we use an optimization algorithm to search for the optimal combination of parameters. After the evaluation process for every benchmark, we obtain the best parameter combination for every benchmark.

However, there are still some issues to be handled. Rules for parameter granularity need to be summarized, and the evaluation functions should be extended to more benchmarks, not just Olden benchmarks.

References

Akkary, H., Driscoll, M.A., 1998. A dynamic multithreading processor. Proc. 31st Annual ACM/IEEE Int. Symp. on Microarchitecture, p.226-236.

Bhowmik, A., Franklin, M., 2002. A general compiler framework for speculative multithreading. Proc. 14th Annual ACM Symp. on Parallel Algorithms and Architectures, p.99-108. [doi:10.1145/564870.564885]

Chen, Z., Zhao, Y., Pan, X., *et al.*, 2009. An overview of Prophet. Proc. 9th Int. Conf. on Algorithms and Architectures for Parallel Processing, p.396-407. [doi:10.1007/978-3-642-03095-6_38]

Dasgupta, D., 1999. Artificial Immune Systems and Their Applications. Springer Berlin Heidelberg. [doi:10.1007/978-3-642-59901-9]

de Castro, L.N., Timmis, J., 2002. Artificial Immune Systems: a New Computational Intelligence Approach. Springer.

Dong, Z., Zhao, Y., Wei, Y., *et al.*, 2009. Prophet: a speculative multi-threading execution model with architectural support based on CMP. Proc. 8th Int. Conf. on Embedded Computing, and Int. Conf. on Scalable Computing and Communications, p.103-108. [doi:10.1109/EmbeddedComScalCom.2009.128]

Heinrich, J., 1994. MIPS R4000 Microprocessor User's Manual (2nd Ed.). MIPS Technologies, Inc., Mountain View, CA.

Krishnan, V., Torrellas, J., 1999. A chip-multiprocessor architecture with speculative multithreading. *IEEE Trans. Comput.*, **48**(9):866-880. [doi:10.1109/12.795218]

Liu, B., Zhao, Y., Li, Y., *et al*., 2014. A thread partitioning approach for speculative multithreading. *J. Supercomput.*, **67**(3):778-805. [doi:10.1007/s11227-013-1000-1]

Madriles, C., Lopez, P., Codina, J.M., *et al.*, 2009. Anaphase: a fine-grain thread decomposition scheme for speculative multithreading. Proc. 18th Int. Conf. on Parallel Architectures and Compilation Techniques, p.15-25. [doi:10.1109/PACT.2009.27]

Marcuello, P., González, A., 1999. Clustered speculative multithreaded processors. Proc. 13th Int. Conf. on Supercomputing, p.365-372. [doi:10.1145/305138.305214]

Olukotun, K., Hammond, L., Willey, M., 1999. Improving the performance of speculatively parallel applications on the Hydra CMP. Proc. 13th Int. Conf. on Supercomputing, p.21-30. [doi:10.1145/305138.305155]

Quiñones, C.G., Madriles, C., Sánchez, J., *et al*., 2005. Mitosis compiler: an infrastructure for speculative threading based on pre-computation slices. Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation, p.269-279. [doi:10.1145/1064978.1065043]

Sohi, G.S., Breach, S.E., Vijaykumar, T.N., 1995. Multiscalar processors. Proc. 22nd Annual Int. Symp. on Computer Architecture, p.414-425. [doi:10.1145/223982.224451]

Timmis, J., 2000. Artificial Immune Systems: a Novel Data Analysis Technique Inspired by the Immune Network Theory. Available from https://kar.kent.ac.uk/21989/.

Tsai, J., Yew, P., 1996. The superthreaded architecture: thread pipelining with run-time data dependence checking and control speculation. Proc. Conf. on Parallel Architectures and Compilation Techniques, p.35-46. [doi:10.1109/PACT.1996.552553]

Tsai, J., Huang, J., Amlo, C., *et al*., 1999. The superthreaded processor architecture. *IEEE Trans. Comput.*, **48**(9):881-902. [doi:10.1109/12.795219]

Wang, L., Pan, J., Jiao, L., 2000. The immune algorithm. *Acta Electron. Sin.*, **28**(7):74-78 (in Chinese).

Wilson, R.P., French, R., Wilson, C.S., *et al.*, 1994. The SUIF Compiler System: a Parallelizing and Optimizing Research Compiler. Technical Report No. CSL-TR-94-620, Computer Systems Laboratory, Stanford University, CA.