ICIC Express Letters
Volume 11, Number 6, June 2017
ICIC International ©2017 ISSN 1881-803X
pp. 1053–1061

OPTIMIZING PARTITION THRESHOLDS IN SPECULATIVE MULTITHREADING

Yuxiang Li, Yinliang Zhao, Liyu Sun and Mengjuan Shen
School of Electronic and Information Engineering
Xi'an Jiaotong University
No. 28, Xianning West Road, Xi'an 710049, P. R. China
liyuxiang19841203@163.com; zhaoy@mail.xjtu.edu.cn; { 858115052; 980624675 }@qq.com

Received December 2016; accepted March 2017

Abstract. *Thread-level speculation (TLS) is an automatic parallelization technique for serial programs on multicore platforms, and it permits to generate multiple threads during compiling as well as to run them at runtime. Thread partition plays an important role in TLS. Heuristic rules-based (HR-based) partition and machine learning-based (ML-based) partition are two commonly used approaches. During the partition of these two approaches, what seriously affect the partition effect are five thresholds, i.e., upper limit of spawning distance (ULoSD), lower limit of spawning distance (LLoSD), data dependence count (DDC), upper limit of thread granularity (ULoTG), lower limit of thread granularity (LLoTG). This paper proposes to optimize these five thresholds with a level-based traversal method, in order to find the optimal partition thresholds for every procedure, so as to obtain the optimal partition. Prophet, which consists of an automatic partition compiler and a simulator, is used to perform partition and achieve speedups. Experimental results show that 11 Olden benchmarks obtain speedup improvements.*
Keywords: Thread-level speculation, Partition threshold, Optimization

1. Introduction.

Speculative multithreading [1,2] (SpMT), also called thread-level speculation (TLS), is a promising technique which allows to parallelize sequential codes aggressively, without considering too much about the success of execution, which is guaranteed by hardware. Compared to manual parallelization, SpMT accelerates irregular sequential programs with lower cost and fewer user interactions, as well as has a wide range of applications [3].

Thread partition plays an important role in SpMT. Liu et al. [2] used a machine learning method to perform thread partition. Tang et al. [4] presented a new heuristic algorithm based on an interesting extension of the classical list scheduling algorithm. Based on a cost model, the algorithm groups instructions into threads by considering the trade-offs among parallelism, latency tolerance, thread switching costs and sequential execution efficiency. Li et al. [5] made use of artificial neural network to predict thread partition. Compared to the thread partitions using machine learning, cost model, etc, heuristic rules-based thread partition has the advantages: simplicity, easy to handle, strong practicability.

As the commonly used partition approach, HR-based thread partition and ML-based thread partition severely depend on the specific thresholds of thread granularity, data dependence count, spawning distance, etc. Conventional approaches set the specific thresholds solely by experience, easily resulting in missing the optimal values. This paper overcomes this problem with the level-based traversal method, in which we traverse almost all the points in the solution space, which is extended by the five thresholds, so as to find the optimal point.

The remaining paper is structured as follows. Section 2 gives the motivation, Section 3 shows the overall framework, and Section 4 shows the experiment and analysis. The conclusion and future work are given in the Section 5.

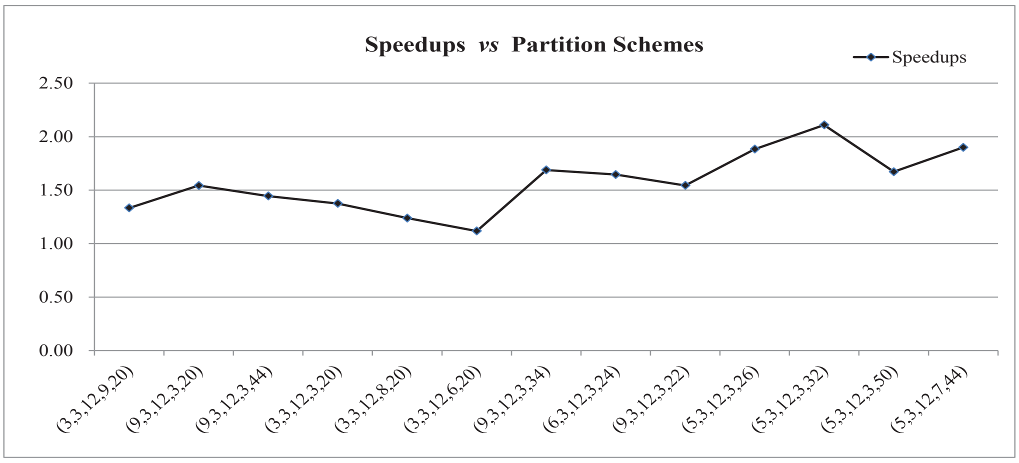


FIGURE 1. Speedups *vs* 13 partition schemes in *health*(). This figure shows that it is important to set the right partition thresholds.

2. Motivations.

This section illustrates that optimizing the partition thresholds in SpMT has a significant impact on performance.

In Figure 1, the x-axis shows 13 groups of partition thresholds ($ULoSD$, $LLoSD$, DDC, $ULoTG$, $LLoTG$) for benchmark *health*(), while the y-axis shows the speedups obtained for these partition thresholds. We see from this figure that changes of partition thresholds can lead to changes of speedups. The maximum speedup is 2.1, while the minimum speedup equals 1.1, so the speedup obtained from the optimal partition thresholds is almost two times the speedup from the worst partition thresholds.

This simple example illustrates that selecting the optimal partition thresholds has a significant performance impact, and obtaining the optimal partition thresholds needs a searching process. Prior researches have already managed to find a group of better partition thresholds for a specific benchmark by experience. Therefore, it is crucial to apply the optimal partition thresholds for a specific program, and we apply a level-based traversal method to finding them.

3. Framework.

3.1. Basic idea.

To provide the separate optimal partition thresholds for different programs, we use the level-based traversal method to build the generator, which is used together with Prophet compiler and simulator simultaneously. Figure 2 shows the overall framework of optimizing partition thresholds on Prophet. The bold words illustrate the part of level-based traversal method, while the other parts are classified into Prophet. This section describes how the thresholds of thread partition can be optimized in Prophet.

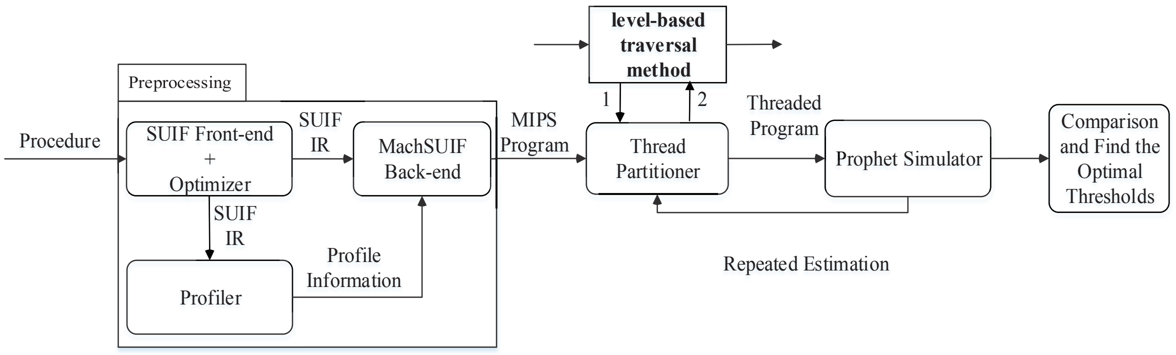


FIGURE 2. Overall framework of optimizing partition thresholds on Prophet [6, 7]

Once a procedure is entered, the first step is preprocessing, which primarily contains SUIF front-end, optimizer, MachSUIF back-end as well as Profiler. After preprocessing, mips codes are generated for thread partitioner, providing which thresholds seriously affect thread partitions. Moreover, level-based traversal method provides all possible partition thresholds for partitioner. After partitioning, threaded codes are entered into Prophet simulator to obtain speedups. We reserve and compare all possible partition thresholds and their corresponding speedups, so as to find the optimal partition thresholds which correspond to the biggest speedup.

3.2. **Optimization code.** Figure 3 shows the simple implementation code of level-based traversal method. During this code, five *for* loops are used to traverse all possible points in the solution space, which is built by these five thresholds. Within the code, *ULoSD*, *LLoSD*, *DDC*, *ULoTG*, *LLoTG* are partition thresholds to be optimized. Note that the maximum values and minimum values of five thresholds are given by experience, and shown in Table 1.

```
For(LLoSD=3;LLoSD<=9;LLoSD+=1);do
For(ULoSD=3;ULoSD<=15;ULoSD+=1);do
For(DDC=12;DDC<=32;DDC+=1); do
For(LLoTG=3;LLoTG<=9;LLoTG+=1);do
For(ULoTG=20;ULoTG<=50;ULoTG+=1);do
Compiler//perform compiling and execute the whole procedure,
obtaining the speedup of every sub-procedure, updating the
speedup of whole procedure.
End End End End End
```

Figure 3. Simple implementation code of level-based traversal method

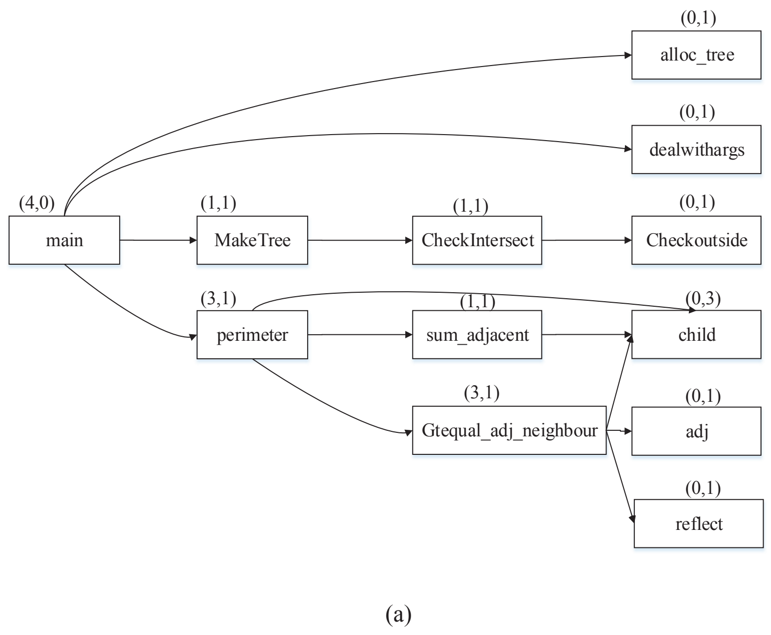
Table 1. Traverse range of partition thresholds

threshold name	**lower limit**	**upper limit**
DDC	3	9
LLoTG	3	15
ULoTG	12	32
LLoSD	3	9
ULoSD	20	50

3.3. **Optimization sequence.** We adopt the level-based optimization method, in which we firstly build a function call tree, and then traverse all nodes from the leaf to root. Figure 4(a) shows the function call graph of benchmark *perimeter*(). The arrays in the tree represent function call relations. The procedure *main*() is the root node, while *alloc_tree*(), *dealwithargs*(), *Checkoutside*(), *child*(), *adj*(), and *reflect*() are all leaf nodes.

In order to obtain an overall optimal speedup for a procedure, we need use the level-based traversal method to realize that every sub-procedure obtains their optimal partition thresholds. In the level-based traversal method, the most important point is the optimization sequence of all sub-procedures. To handle this issue, we adopt a level-based statistic. Figure 4(b) shows the four levels of *perimeter*(), and the thick black arrays indicate the optimization sequence. All the sub-procedures with the same level have no fixed sequence. The sub-procedures in level 1 are firstly optimized, and then level 2, level 3, level 4.

The leaf nodes are firstly optimized, and then their optimization thresholds are all reserved. While partitioning all sub-procedures (in leaf nodes), we make use of their respective optimization thresholds. Then, we start to optimize the partition thresholds



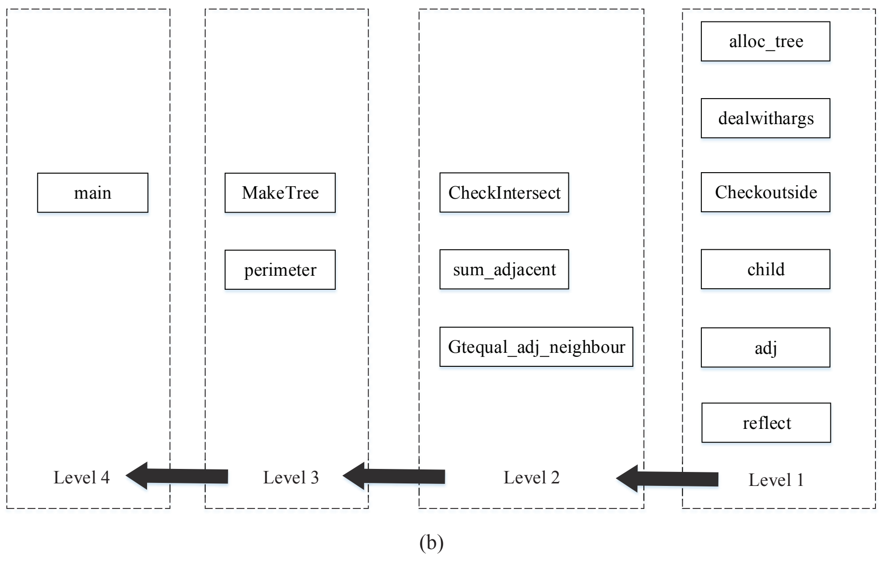


FIGURE 4. (a) Function call graph of *perimeter*(), the numbers in brackets are separately calling time and called time of sub-procedures; (b) levels of *perimeter*()

of all sub-procedures (in the level 2) while not changing the reserved optimization results in the level 1. Finally, we obtain the overall optimal thresholds of all levels.

3.4. **Optimization times.** According to the basic framework shown in Figure 2, we need implement all the modules in this figure to complete a traversal. However, it is a waste of time to finish all traversals. Table 1 shows the traversal ranges of five thresholds in benchmark $bh()$. If we set step size 1, the traversal time of five thresholds is calculated in Formula (1):

$$\begin{aligned}
&(\max(LLoSD) - \min(LLoSD) + 1) * (\max(ULoSD) - \min(ULoSD) + 1) \\
&\quad * (\max(ULoTG) - \min(ULoTG) + 1) * (\max(LLoTG) - \min(LLoTG) + 1) \\
&\quad * (\max(DDC) - \min(DDC) + 1) \qquad (1) \\
&= (9 - 3 + 1) * (50 - 20 + 1) * (15 - 3 + 1) * (32 - 12 + 1) * (9 - 3 + 1) \\
&= 7 * 31 * 13 * 21 * 7 = 414687
\end{aligned}$$

Table 2. Level and execution times of sub-procedures in *perimeter*()

sub-procedure name	**level**	**execution times**
main()	4	1
perimeter()	3	245
MakeTree()	3	245
Gtequal_adj_neighbour()	2	1088
CheckIntersect()	2	245
sum_adjacent()	2	120
adj()	1	1056
reflect()	1	840
child()	1	920
Checkoutside()	1	1305
alloc_tree()	1	245
dealwithargs()	1	1

There are overall four levels, so the whole traversal times are $414687 * 4 = 1658748$. If the time spent in running a time of $bh()$ is 70.2 seconds, then the time taken to finish the 4 levels is about 1347.7 days. This result is difficult to accept. To deduce the execution time as well as not to miss the optimal thresholds, we adopt a level-based traversal method:

- We manage to control the step size of every threshold, to make the consuming time of overall execution acceptable;
- Near the approximate optimal thresholds, we start to conduct a precision search with the level-based traversal method.

4. **Experiment and Analysis.**

4.1. **Experiment configuration.**

We implement the execution model as well as thread partition algorithm on Prophet [6,7], which is based on SUIF/MACHSUIF [8]. We complete the compiler analysis at the level of SUIF's intermediate representations. The profiling information is extracted from SUIF-IR in the form of annotations. The SUIF programs which are interpreted and executed by profiler provide information, including dynamic instruction number, control flow path prediction, and data value prediction. The Prophet simulator simulates 8 pipelined mips-based R3000 processing elements (PEs). This simulating process is an execution-driven simulation, which executes 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 multiversion L1 cache, which has latency of 2 cycles. Speculative results of PE are buffered and cache communication is performed via multiversion L1 caches. With a snoopy bus, a write-back L2 cache is shared by the 8 PEs. The simulator's parameter configuration is shown in Table 3.

4.2. **Step setting.**

To optimize the partition thresholds as well as reduce the execution time of level-based traversal method, we set step size in accordance with the called times of every sub-procedure, which are the 2nd numbers in the brackets of Figure 4. In order to optimize partition thresholds, we take three measures: numbering every sub-procedure, ranking, and setting steps.

4.2.1. *Numbering every sub-procedure.*

To set an appropriate step size for every sub-procedure (shown in Figure 4), we need firstly number every procedure. We assign every procedure with an ID from 1 to n ($n \in N$), which are shown in the 1st column of Table 4.

Table 3. The configuration of Prophet (per PE)

Parameters of configuration	**Value**
Bandwidth for Fetch, In-order Issue and Commit	4 Instructions
Pipeline Stages	Fetch/Issue/Ex/WB/Commit
Architectural Registers	4 int and 4 fp
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)
L1-Cache (Multiversioned)	4-Way Associative 64KB (32B/Block)
	Hit Latency 2
	LRU Replacement
Spec. Buffer Size	Fully Associative 2KB (1 Cycle)
L2-Cache	4-Way Associative 2MB (64B/Block)
	5 hit latency, 80 Cycles (miss)
	LRU replacement
Spawn Overhead	5 Cycles
Validation Overhead	15 Cycles
Local Register	1 Cycle
Commit Overhead	5 Cycles

Table 4. Level, called time, rank, and step size of sub-procedures in *perimeter*()

ID	**sub-procedure name**	**level**	**called time**	**rank**	**step size**
1	*child*()	1	3	1	(1,1,1,1,1)
2	*perimeter*()	3	1	2	(2,2,2,2,2)
3	*MakeTree*()	3	1	2	(2,2,2,2,2)
4	*Gtequal_adj_neighbour*()	2	1	2	(2,2,2,2,2)
5	*CheckIntersect*()	2	1	2	(2,2,2,2,2)
6	*sum_adjacent*()	2	1	2	(2,2,2,2,2)
7	*adj*()	1	1	2	(2,2,2,2,2)
8	*reflect*()	1	1	2	(2,2,2,2,2)
9	*Checkoutside*()	1	1	2	(2,2,2,2,2)
10	*alloc_tree*()	1	1	2	(2,2,2,2,2)
11	*dealwithargs*()	1	1	2	(2,2,2,2,2)
12	*main*()	4	0	3	(3,3,4,3,3)

4.2.2. *Ranking.* Table 4 shows the step setting of every sub-procedure. In the 2nd column of the table, 12 sub-procedures of Figure 4 are shown. After numbering every sub-procedure, we start to rank all sub-procedures in accordance with the called time. The sub-procedure with maximum called time is *child*(), which owns 3 called time, so we assign *child*() with rank 1. Similarly, we assign sub-procedures *perimeter*(), *MakeTree*(), *Gtequal_adj_neighbour*(), *CheckIntersect*(), *sum_adjacent*(), *adj*(), *reflect*(), *Checkoutside*(), *alloc_tree*(), *dealwithargs*() with rank 2, as they have 1 called time. Finally, we assign *main*() with rank 3, as it has 0 called time.

4.2.3. *Setting steps.* According to rank number i ($i = 1, 2, 3, \ldots$) of every sub-procedure, we set the step size of the jth ($j = 1, 2, 3, 4, 5$) thresholds with the ith common divisor (the 1st common divisor is 1) of difference value between the maximum value of the jth threshold and the minimum value of the jth threshold.

```
For((currentlevel=1;currentlevel<=mainlevel;currentlevel+=1));do
step0=f(0);step1=f(1);step2=f(2);step3=f(3);step4=f(4);
For(LLoSD=3;LLoSD<=9;LLoSD+=step0); do
For(ULoSD=3;ULoSD<=15;ULoSD+=step1);do
For(DDC=13;DDC<=32;DDC+=step2);do
For(LLoTG=3;LLoTG<=9;LLoTG+=step3);do
For(ULoTG=20;ULoTG<=50;ULoTG+=step4);do
compile
End End End End End End
```

FIGURE 5. Optimization code

Assume the step size of the jth ($j = 1, 2, 3, 4, 5$) threshold of sub-procedure with rank number i is $Y(j)$, and then the calculation of $Y(j)$ is in Formula (2).

$$\begin{cases} Y(j) = \textit{the ith common divisor of difference value between maximum of the jth} \\ \textit{threshold and minimum of the jth threshold, where}, i = 1, 2, 3, \ldots;\ j = 1, 2, 3, 4, 5. \end{cases} \tag{2}$$

Taking sub-procedure *dealwithargs*() for example, it has 1 called time. Then we calculate the difference values of five thresholds, i.e., 6 ($9 - 3 = 6$), 12 ($15 - 3 = 12$), 20 ($32 - 12 = 20$), 6 ($9 - 3 = 6$), 30 ($50 - 20 = 30$). Moreover, we calculate the 2nd common divisor of every difference value since *dealwithargs*() has rank number 2, and obtain the results: (2, 2, 2, 2, 2). Namely the step sizes of every threshold in *dealwithargs*() are respectively 2, 2, 2, 2, 2.

4.3. **Optimization process.**

4.3.1. *Optimization setting.* Taking benchmark *perimeter*() for example, we have obtained the level, step size of every threshold. Table 4 shows the step size and specific execution times for every sub-procedure.

4.3.2. *Optimization code.* As the threshold changes are correlated with level and step size, we design the level-based traversal method, which considers these two factors. Before traversing all the sub-procedures of one level, we control the step sizes in this level so that traversing time of every level decreases with the rising of level number. Moreover, we emphasize on traversing the lowest two levels. The specific optimization code is shown in Figure 5, in which *step0*, *step1*, *step2*, *step3*, *step4* are the optimized step sizes of five thresholds.

4.4. **Results of optimization.**

Table 5 shows the level, step size, and execution times of sub-procedures in *health*(). *th1*, *th2*, *th3*, *th4*, *th5* respectively denote *DDC*, *LLoSD*, *ULoSD*, *LLoTG*, *ULoTG*. The bold part in Table 5 represents the final speedup after optimization, while the italic part represents the initial speedup before optimization. Similar to *perimeter*(), we perform optimizations of partition thresholds for other benchmarks, and show speedup comparisons in Figure 6.

From Table 5, we can see that *health*() has speedup improvement when all sub-procedures have speedup improvements, as every sub-procedure has an optimal partition threshold. In Figure 6, all benchmarks obtain speedup improvements after optimization of partition thresholds.

TABLE 5. Step size of every threshold, and speedups of sub-procedures in *health*()

sub-procedure Name	**th1**	**th2**	**th3**	**th4**	**th5**	**speedups**
dealwithargs()	3	3	12	9	20	*1.33333*
my_rand()	9	3	12	3	21	1.54245
generate_patient()	9	3	12	3	44	1.44414
put_in_hosp()	3	3	12	3	21	1.37478
addList()	3	3	12	3	20	1.23843
removeList()	3	5	12	3	20	1.1175
sim()	8	3	12	3	45	1.69066
check_patients_inside()	3	5	12	3	20	1.64539
check_patients_assess()	7	3	12	3	20	1.5439
check_patients_waiting()	6	3	12	3	22	1.88305
get_results()	8	3	12	3	45	1.67068
main()	5	3	12	7	44	**1.89326**

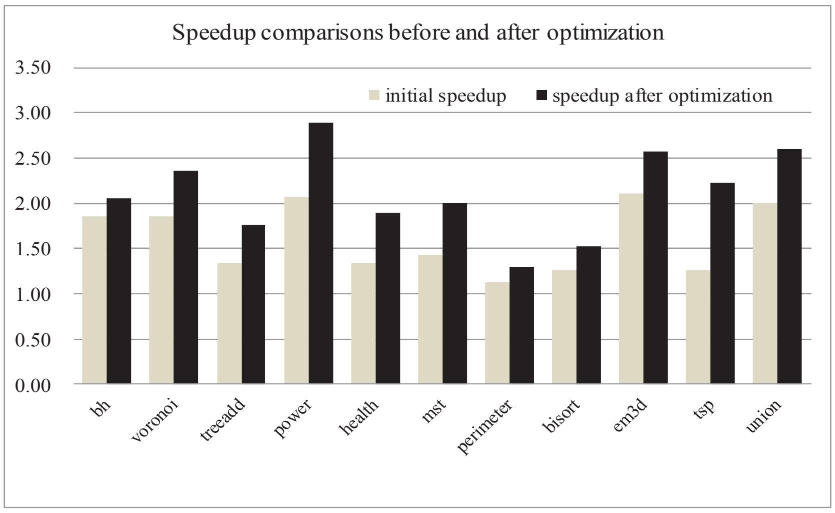


FIGURE 6. Speedup comparisons before and after optimization of partition thresholds

5. Conclusion and Future Work.

5.1. Conclusion.

In this paper, we propose an optimization approach of thread partition thresholds for HR-based and ML-based thread partitions, to improve the efficiency of thread partition. The novelties of this paper can be concluded as follows.

- Five thresholds, which seriously influence thread partition are extracted.
- A level-based traversal method is proposed to realize the optimization of partition thresholds.
- To improve the optimization efficiency, we adopt a method of adaptive step setting.

In conclusion, all Olden benchmarks obtain performance improvements.

5.2. Future work.

In this paper, the speedups obtained by threshold optimization can be used to evaluate the merits of the obtained thresholds, and the obtained speedups can be seen as the speedups of samples. We need to use the machine learning algorithm to make the actual speedups of unknown programs be as close as the speedups of samples. The future work is summed up in the following aspects:

- Better search algorithms are needed to reduce the time of obtaining the optimal thresholds;
- The rules for changing thresholds will be summarized, so that future work of adjusting thresholds can follow these rules.

Acknowledgement. We thank our 3C laboratory for their great support during our work. We give our best hope to all our colleagues of laboratory for their collaboration. We also give our thanks to reviewers for their careful comments and suggestions. This work is supported by Doctoral Fund of Ministry of Education of China under Grant No. 2013021110012 and National Natural Science Foundation of China through Grant No. 61173040.

REFERENCES

[1] A. Estebanez, D. R. Llanos and A. Gonzalez-Escribano, A survey on thread-level speculation techniques, *ACM Computing Surveys (CSUR)*, vol.49, no.2, 2016.

[2] B. Liu, Y. Zhao, X. Zhong, Z. Liang and B. Feng, A novel thread partitioning approach based on machine learning for speculative multithreading, *The 15th IEEE International Conference on High Performance Computing and Communications & IEEE International Conference on Embedded and Ubiquitous Computing (HPCC_EUC)*, pp.826-836, 2013.

[3] D. Bader, *Analyzing Massive Social Networks Using Multicore and Multithreaded Architectures*, Springer-Verlag, 2010.

[4] X. Tang, J. Wang, K. Theobald and G. R. Gao, Thread partition and schedule based on cost model, *Proc. of the 9th Annual ACM Symposium on Parallel Algorithms and Architectures*, Newport, Rhode Island, 1997.

[5] Y. Li, Y. Zhao and H. Gao, Using artificial neural network for predicting thread partitioning in speculative multithreading, *The 17th IEEE International Conference on High Performance Computing and Communications (HPCC), the 7th IEEE International Symposium on Cyberspace Safety and Security (CSS), and the 12th IEEE International Conference on Embedded Software and Systems (ICESS)*, pp.823-826, 2015.

[6] Z. Chen, Y. Zhao, X. Pan, Z. Dong, B. Gao and Z. Zhong, An overview of Prophet, *International Conference on Algorithms and Architectures for Parallel Processing*, pp.396-407, 2009.

[7] S. Song, Y. Zhao, B. Feng, Y. Wei, X. Wang and H. Zhao, Prophet+: An extended multicore simulator for speculative multithreading, *Journal of Xi'an Jiaotong University*, vol.44, no.10, pp.13-15, 2010.

[8] R. P. Wilson, R. S French, C. S. Wilson, S. P. Amarasinghe, J. M. Anderson, S. W. K. Tjiang, S.-W. Liao, C.-W. Tseng, M. W. Hall, M. S. Lam et al., SUIF: An infrastructure for research on parallelizing and optimizing compilers, *ACM Sigplan Notices*, vol.29, no.12, pp.31-37, 1994.