An Adaptive Thread Partitioning Approach in Speculative Multithreading

Yuxiang Li[1,2][0000−0002−3758−7162], Zhiyong Zhang[1,2,4], and Bin Liu[3]

[1] Henan University of Science and Technology, Henan 471023, China
[2] Henan International Joint Laboratory of Cyberspace Security Applications, Henan, 471023, China xidianzzy@126.com
http://www.sigdrm.org/~zzhang/
[3] Northwest Agriculture & Forestry University, Yangling, 471023, China
liubin0929@nwsuaf.edu.cn

[4]

Abstract. Thread partition is a core part of Speculative Multithreading (SpMT) technique. The existing thread partition approaches mostly adopt one unique thread partitioning scheme for unknown programs, resulting in high misspeculation ratio, restricting the programs′ speedup improvement due to inappropriate partitioning schemes. This paper which introduces an adaptive thread partition approach (AdapTPA), takes the relationship between program complexity and thread partitioning scheme as the research entry point, and uses the irregular programs as the research carrier, and utilizes formal analysis, probability statistics, mathematical modeling and simulation experiments to reveal the rule that program′s characteristics affect speedup performance, and generates a compound thread partitioning scheme for one program, and selects and executes the most suitable thread partitioning scheme according to the runtime context and the program′s complexity, so to achieve the expected maximum speedups. With the method of path statistics on one program′s control flow graph, the program′s complexity calculation model is set up; A candidate thread partitioning scheme set is constructed on the foundation of classical thread partitioning approaches; Using expert knowledge to guide production rules, a scheme selection mechanism that complies with program complexity is explored. Compared to the heuristic rules-based (HR-based) thread partitioning method, the experiment results show that AdapTPA delivers an average 18.24% performance improvement.

Keywords: thread partition approach· Speculative Multithreading· expert knowledge

[4] Contact Email:xidianzzy@126.com

1 Introduction

Thread-Level Speculation (TLS), which is a speculative multi-threading technique [1, 2], allows data execution between concurrent units to be aggressively executed in parallel, overcoming the weakness that the traditional parallelization methods cannot effectively eliminate thread-level fuzzy dependence. Thread partitioning is a key step in TLS, as it takes charge of the insertion of thread partitioning statements. It is the core of programs′ speculative parallelization, which directly affects the speedup performance [3]. Therefore, the research on thread partitioning approach is urgent. The existing thread partition methods mainly include: thread partition method based on heuristic rules [4] and thread partition methods based on machine learning [5], and et al [6–10]. The former determines the granularity of threads generated after the program is partitioned, the data dependency between threads, and the spawning distance according to heuristic rules, so as to determine the partition flags ((spawning point, sp) and (control quasi-independent point, cqip)); The latter uses machine learning to learn the knowledge of thread partition in the sample set, and predicts its partition scheme based on the characteristics of the new input program. However,these two types of thread partition methods consider one program as a partition unit. A unified thread partition scheme is used for the procedure in the program. The lack of a personalized thread partition scheme for the procedures in the program results in the incomplete parallelism of some procedures in the program, resulting that the performance of the program after parallelization can not be maximized. Therefore, it is of great significance to carry out thread partition with the procedures in the program as the object, which can overcome the shortcomings of traditional thread partition methods.

Based on the previous work, this paper intends to use an adaptive mechanism to adaptively select the most suitable thread partitioning scheme based on the program features and context, which can ensure the maximum performance of the serialization program after parallelization, and can provide a new method for multi-core processor design.

The remaining parts of this paper are organized as follows. In section 2, we first briefly present the motivation of AdapTPA; Section 3 presents the overall framework of AdapTPA; Implementation of AdaTPA is shown in section 4; Section 5 presents experiment and analysis; Section 6 shows conclusion and future work; Section 7 shows acknowledgement.

2 Motivation of AdapTPA

This paper brings an adaptive mechanism, proposing an adaptive thread partition ing approach(AdapTPA) for irregular programs, aiming at achieving the overall research goal of maximizing the speedup performance, and providing the pos-

sibility for the wide application and healthy development of emerging parallel technologies.

3 Overall Framework

The research framework to be adopted is shown in Fig.1. The research framework regards the irregular serial program as the input, and establishes the program complexity calculation model, the generation of candidate thread partition scheme, and the expert knowledge-based partition scheme selection as the main research points, and selects the most suitable thread partitioning scheme to perform the thread partition. The results are run on Prophet simulator to obtain speedups and programs′ results.

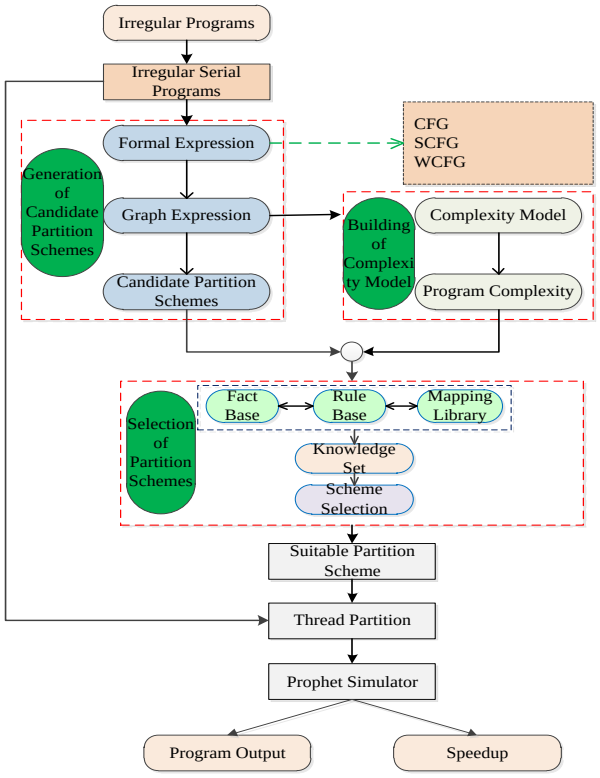

Fig. 1. Overall Research Framework. The framework includes three main parts:generation of candidate partition schemes, building of complexity model, and selection of partition schemes

3.1 Feature Extraction

Conventionally, compiler researchers have used fixed-length representations of the program's source code features or intermediate representations [11]. They are extracted from programs and collected during compilation time. Afterwards, we apply graph-based features to build WCFG for GbA. The feature graphs are generated from profiling pass, which extracts static and dynamic features. Figure 4 gives a simple description of feature extraction. The input programs are Olden benchmarks [12]. Thread granularity, load balance, data dependence, and control dependence are the main influence factors on program speedup. Hence, we take dynamic instruction number, DDD, DDC, loop branch probability, and critical path into account and regard them as program features. The specific features and descriptions are given in Table 1.

Table 1. Extracted Features and Descriptions

Features	**Descriptions**
Instruction Number	Actual number of instructions in a basic block
DDC	Data dependence count between two basic blocks
DDD	Data dependence distance between two basic blocks
Loop branch Probability	Probability for loop to jump to testing part of code
Branch Probability	Probability for control flow to pass through a branch

3.2 Knowledge Expression

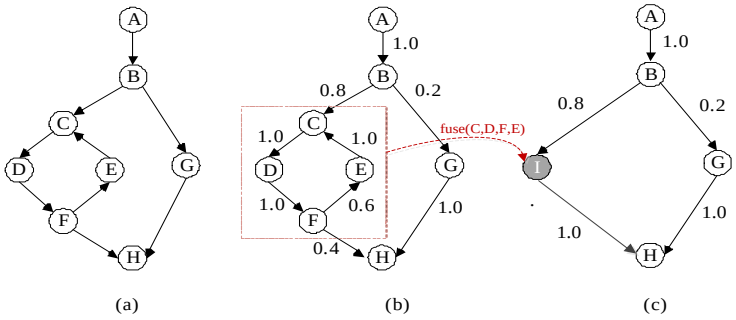

Fig. 2. Transformation from CFG (a) to WCFG (b), and Finally to SCFG (c)

Deriving from the same input, train set and validation set are divided. Firstly, we use partition compiler (in the Prophet) to generate control flow graph (CFG) after an intermediate pass. Then, profiled feature information generated

by profiling model are annotated to the CFG with a structural analysis method, so to generate WCFG. The weights of each edge are denoted with the relative branch probabilities. Then, a structural analysis traverses the CFGs of programs to WCFGs and also identifies loop regions. Then, the loop regions are induced into a super node with one entry and one exit node and WCFG traverses to super control flow graph (SCFG), where loop region is represented as an abstract node. Each node in the SCFG is either basic block or super basic block, which represents loop region. Fig.2(a) shows a CFG. After structural analysis as well as loop region induction, basic blocks:C, D, F and E in the dashed box of Fig.2(b) are induced into a super basic block I (shown in Fig.2(c)). AdapTPA represents the

Table 2. Heuristic Rules for Thread Partition

1.SP can appear anywhere in programs and behind a function call instruction as far as possible. In non-loop region, CQIP is located in the beginning of a basic block. CQIP is located in front of loop branch instructions in the last basic block in loop regions.
2.SP-CQIP pairs are located within the same loop or function. The number of dynamic instructions from SP to CQIP is between THREAD_LOWER_LIMIT and THREAD_UPPER_LIMIT.
3.Between two successive candidate threads, spawning distance is bigger than minimum DIS_LOWER_LIMIT.
4.Data dependence between two consecutive candidate threads is less than threshold DEP_THRESHOLD.
5.Between SP and CQIP, the number of function call instructions is less than the threshold CALL_LOWER.

desired thread partition scheme with a vector H=[H_1, H_2, H_3, H_4, H_5] and the sample partition scheme with h_i=[h_{i1}, h_{i2}, h_{i3}, h_{i4}, h_{i5}]($i \in N$),which all include five thresholds: the upper limit of thread granularity ($ULoTG$), the lower limit of thread granularity ($LLoTG$), data dependence count (DDC), the upper limit of spawning distance ($ULoSD$), and lower limit of spawning distance ($LLoSD$). As these five parameters determine the effectiveness of thread partition, and the partition scheme is represented by [$ULoTG$, $LLoTG$, DDC,$ULoSD$, $LLoSD$]. For example, one partition scheme could be [50, 10, 18, 30, 20]. These values indicate that during thread partition thread granularity is set from 10 to 50, and data dependence count is less than 18, and spawning distance ranges from 20 to 30.

A novel research result is successful construction of samples [13]. Based on generated samples, we obtain the partition scheme $h_i(i \in N)$ of every sample by means of mathematical statistics. Although we use a graph to denote every sample, the node in the graph is represented by the first part of T=$\{X, H\}$, where X represents program features, and H denotes the optimal partition scheme,

which are composed by five partition thresholds, namely *ULoTG*, *LLoTG*, *DDC*, *ULoSD*, *LLoSD*.

4 Implementation of AdapTPA

4.1 Building of Complexity Calculation Model

There are many program features that affect thread partitioning, such as data dependency, control dependency, number of branches, number of basic blocks, number of average dynamic instructions, nesting level of loop structure, number of procedure calls, and so on. The values of these features reflect the complexity of the program. Most of the existing thread partitioning methods can not fully consider the influence of program complexity on thread partitioning. Only the program features are selected as the input of the thread partitioning method. It is easy to cause the program features selected by different thread partitioning methods to be inconsistent, and the generated thread partitioning scheme is not accurate enough.

In the proposed program′s complexity calculation model, the formal expression is firstly constructed, and the CFG diagram of the program is constructed with the basic block as the analysis unit. The feature values obtained by the program analysis are added to the CFG diagrams in the form of annotations to form the weighted control flow (WCFG); based on probability statistics and graph traversal, the complexity (sub-complexity) of possible paths on WCFG is calculated; finally, the overall complexity of the program is obtained by integrating sub-complexities. Fig.3 shows the flow chart for program complexity calculation, and Table 3 shows the pseudocode of complexity calculation.

In Fig.3, P represents the input irregular serial program, *G(P)* stands for WCFG, *F1*∼*Fn* (n∈N) stand for program features, and *f1()*∼*fn()* (n∈N) stand for transfer function, *Comp1()*∼*Compn()* (n∈N) represent the complexity of each path, and *Comp* represents the total complexity of P. In the model, first, the unknown program P is formalized and converted into WCFG, i.e. *G(P)*; Secondly, feature extraction is performed on each possible path (from the head node to the tail node) in *G(P)* respectively, using *F1*∼*Fn* (n∈N); Thirdly, using the conversion function *f1()*∼*fn()*(n∈N) to achieve the mapping of eigenvalues to complexity, for example, the complexity of basic blocks x is $0.01\times x$, the complexity of loops y is $0.2\times y$, *etc*; then, the complexity *Comp1()*∼*Compn()* of each path in *G(P)* is calculated separately; Finally, for each path, the complexity is summarized to get the complexity of the program P.

4.2 Building of Candidate Thread Partition Scheme Set

The candidate thread partitioning scheme set is constructed with the method of fusing program context and program features. Firstly, the initial candidate set

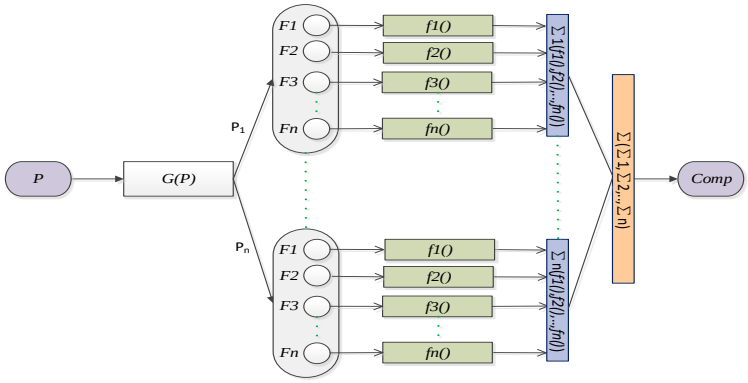

Fig. 3. Flow Diagram of Complexity Calculation.The graph P is firstly represented by $G(P)$, the the complexity of different part of P is separately calculated, then aggregated into $Comp$

Table 3. Computation of Complexity

Input: irregular program P
Output: Complexity of program P
$G(P)$ = formalize(P); Extract and express the characteristics of program′s the i_{th} possible path (possibility P_i in Fig.4) with F1∼Fn; Set $f_1()$∼$f_n()$ to be n transfer functions; Set W_1∼W_n to be n weight parameters for F_1∼F_n; Use function $Sub\_complexity(f_1,f_2,f_3,...,f_n)$ to compute the complexity of every possible path; According to every Sub_complexity, compute the final complexity computation with function $Comp$ = $Complexity(Sub\_complexity1,Sub\_complexity2,...,Sub\_complexityn)$;

is constructed based on the program features. Based on this, the program context parameter values are used to filter the initial candidate thread partitioning scheme set, so to generate the final scheme set. Fig.4 shows the construction process of the candidate thread partitioning scheme set.

In Fig.4, P stands for an irregular serial program, $F1$∼Fn stand for program feature, $Formal(P)$ stands for formal expression of P, $M1$∼$Mn(n∈N)$ stand for n classical thread partitioning approaches, and $Schem1$∼$Schemn$ stand for n thread partitioning schemes. The number of thread partition method is set as follows:HR-based thread partition method ($M1$) is numbered 1, ML-based thread partition method ($M2$) is numbered 2, and the critical path-based thread partition method($M3$) is numbered 3, the full path-based thread partition method ($M4$) is numbered 4, the hybrid thread partition method ($M5$) is numbered 5, and so on. The path numbers are set as follows:the critical path number is numbered 1, and the other non-critical path numbers are 2∼n. The thread partition scheme consists of five main parameters:number of thread par-

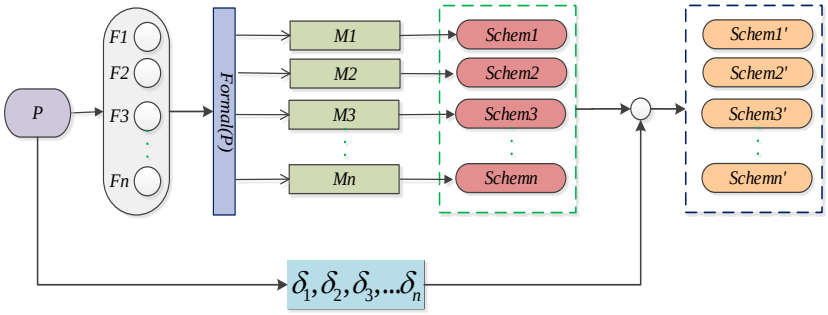

Fig. 4. Flow Diagram of Building Candidate Thread Partition Scheme Set. The graphs of input programs are firstly formalized, then generated into different partition schemes

tition method, path number and thread partition algorithm (the five parameters are: Upper Limit of Spawning Distance ($ULoSD$), Lower Limit of Spawning Distance, ($LLoSD$), Data Dependence Count (DDC), Upper Limit of Thread Granularity ($ULoTG$), and Lower Limit of Thread Granularity ($LLoTG$)). By introducing the context parameter $\delta_1 \sim \delta_n$(n∈N), the thread partitioning method in this topic is context-aware, and the candidate thread partitioning scheme set can also capture the change of the program state.

4.3 Construction of Thread Partitioning Scheme Selection Mechanism in line with Program Complexity

After calculating the program complexity and constructing the candidate thread partitioning scheme set respectively in the technical routes (1) and (2), based on the expert knowledge, the mapping rule set of scheme selection of "program complexity->thread partitioning scheme" is established; according to mapping rule and program complexity, execution context, the most suitable thread partitioning scheme in the candidate set is selected. Fig.5 shows the flow chart of thread partitioning scheme selection.

Rule sets are used to store expert knowledge for reasoning. In the rule set, the expert knowledge of the thread partitioning scheme selection mechanism is represented by a production rule (also called a mapping rule). The production rule divides the knowledge representation into two parts: premise and conclusion. The general form of expert knowledge production rule representation is IF <condition>, THEN <conclusion>, for example:

1. IF <$Comp$∈[0.8,1.0]>, THEN <select $Schem1'$>;
2. IF <$Comp$∈[0.6,0.8)>, THEN <select $Schem2'$>;

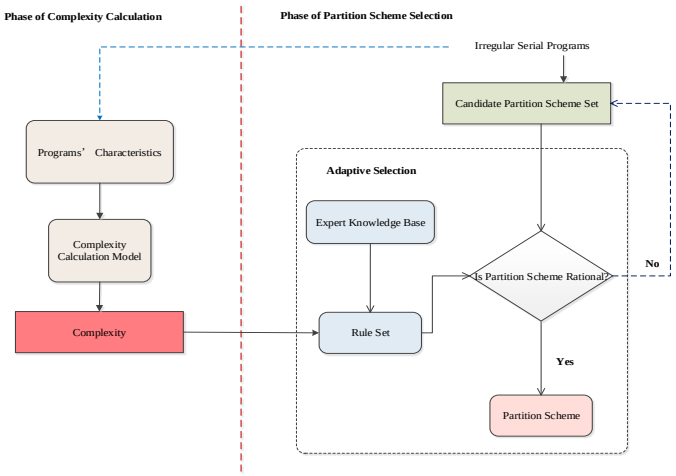

Fig. 5. Flow Graph of Thread Partition Scheme Selection.The complexity of input program is firstly calculated, then partition schemes are generated by using expert knowledge

3. IF $<Comp \in [0.4,0.6)>$, THEN <select $Schem3^{'}$>;
4. IF $<Comp \in [0.2,0.4)>$, THEN <select $Schem4^{'}$>;
5. IF $<Comp \in [0.0,0.2)>$, THEN <select $Schem5^{'}$>;

where, $Schem1^{'} \sim Schem5^{'}$ is a partitioning scheme selected by the candidate thread partitioning scheme generated by the technical route (2), which is determined by the complexity and rules of the program. Some examples of generating mapping rules are given above.

5 Experiment and Analysis

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.6), which is based on SUIF/MACHSUIF [14]. 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 4.

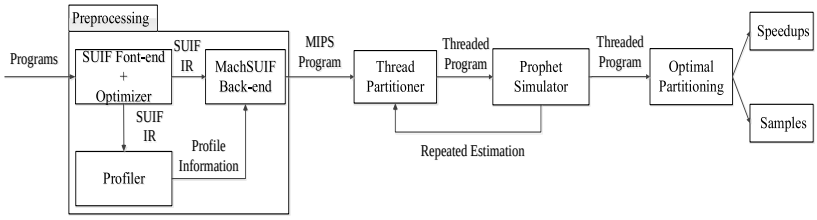

Fig. 6. Module Chart of Prophet. Programs are firstly transformed into MIPS codes, then partitioned into threaded programs, then run on Prophet simulator

Olden benchmarks [15] and SPEC2000 [16] 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.

AdapTPA 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 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.

Table 4. 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 64KB (32B/Block)
	Hit Latency 2
	LRU Replacement
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
k	5
Similarity Threshold	0.5

5.2 Experimental Configuration

In order to show the effectiveness of AdapTPA, this paper makes a comparison between AdapTPA and HR-based thread partition. Olden benchmarks [17] which have complex data dependence and control dependence among basic blocks, are selected as the inputting programs. When we analyze the experimental results, we only compare the performance of the original HR-based thread partition approach and AdapTPA, 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 AdapTPA, the optimal partition scheme matching with the characteristic of every procedure in the program can be selected, and then the partition scheme is applied to the threads. However, due to the existence of more dependence, AdapTPA 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 AdapTPA can obtain the partition scheme suitable for its own characteristics, the characteristic extrac-

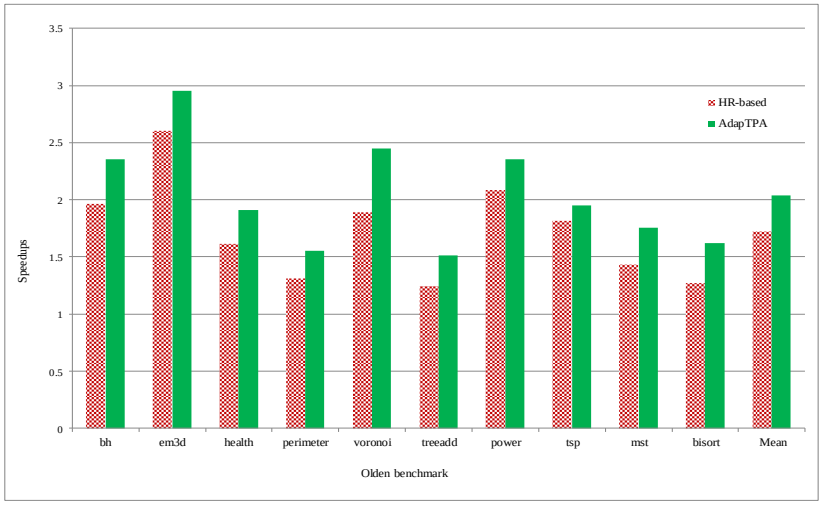

Fig. 7. Comparison Diagram of Speedups for Olden Benchmarks

tion 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 *health*, 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, 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 multithreading. Because it is difficult to predict the return value of the function, the acceleration effect of these two approaches are not good. Compared with HR-based partition approach, AdapTPA 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. AdapTPA can select the appropriate partition scheme for every

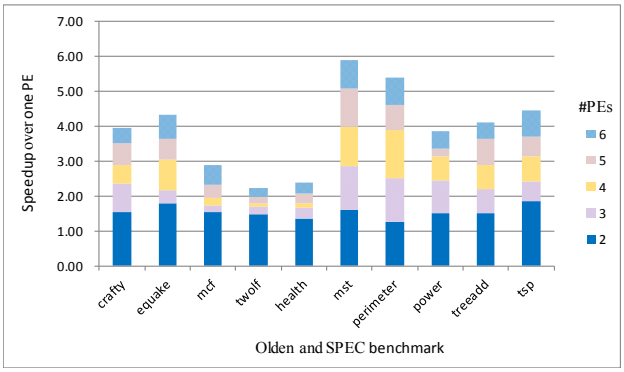

Fig. 8. Speedup Comparison Diagram of Olden and SPEC2000 Benchmarks over Different PEs

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 AdapTPA in every procedure. AdapTPA selects the suitable partition scheme for every procedure, finally obtains 27.47% performance improvement.

Fig.7 shows the speedup comparisons between HR-based and AdapTPA. Seen from Fig.7, the speedups obtained by AdapTPA 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 AdapTPA gets an average speedup of 2.040, so the average speedup improves by 18.24%, indicating that AdapTPA has a good effect on the program partition. Fig.8 shows the speedups of some SPEC2000 and Olden benchmarks on different number of cores.

6 Conclusion and Future Work

Based on the Prophet system, this paper proposes an Adaptive Thread Partition Approach (AdapTPA), and brings an adaption mechanism into thread partition. According to programs′ characteristics, the complexity of unknown program is calculated, candidate thread partition scheme set is built, and the most suitable partition scheme is selected in accordance with programs′ characteristics

and running context. Finally, the program is executed on the Prophet simulator to verify its execution performance. Thread Level Speculation has been evolving many years, showing great advantages in making use of multicore resources. AdapTPA is proposed to handle the issue that conventional partitioning approaches can not generate the best partitioning scheme for unknown programs. The trend of adaptive thread partition falls on two parts:1.more detailed candidate thread partitioning schemes are designed; 2.adaptive thread partition is implemented on hardwares.

7 Acknowledgement

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. The work was sponsored by National Natural Science Foundation of China Grant No.61972133,Project of Leading Talents in Science and Technology Innovation for Thousands of People Plan in Henan Province Grant No.204200510021, Henan Province Key Scientific and Technological Projects Grant No.192102210130 and No.202102210162, and Key Scientific Research Projects of Henan Province Universities Grant No.19B520008.

Bibliography

[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, p. 22, 2016.

[2] C. Hammacher, K. Streit, A. Zeller, and S. Hack, "Thread-level speculation with kernel support," 2016.

[3] L. I. Yu-Xiang, Y. L. Zhao?, B. Liu, and J. I. Shuo, "Optimization of thread partitioning parameters in speculative multithreading based on artificial immune algorithm," *Frontiers of Information Technology & Electronic Engineering*, vol. 16, no. 3, pp. 205–216, 2015.

[4] C. Madriles, C. Garcia-Quinones, J. Sanchez, P. Marcuello, A. Gonzalez, D. M. Tullsen, H. Wang, and J. P. Shen, "Mitosis:a speculative multithreaded processor based on precomputation slices," *IEEE Transactions on parallel and distributed systems*, vol. 19, no. 7, pp. 914–925, 2008.

[5] Y. Li, Y. Zhao, and Q. Wu, "Gba: A graphbased thread partition approach in speculative multithreading," *Concurrency & Computation Practice & Experience*, vol. 29, no. 21, p. e4294, 2017.

[6] M. Qiu and E. H. M. Sha, "Cost minimization while satisfying hard/soft timing constraints for heterogeneous embedded systems," *ACM Transactions on Design Automation of Electronic Systems*, vol. 14, no. 2, p. 25, 2009.

[7] M. Qiu, W. Dai, and A. V. Vasilakos, "Loop parallelism maximization for multimedia data processing in mobile vehicular clouds," *IEEE Transactions on Cloud Computing*, vol. 7, no. 1, pp. 250–258, 2019.

[8] H. Qiu, H. Noura, M. Qiu, Z. Ming, and G. Memmi, "A user-centric data protection method for cloud storage based on invertible dwt," *IEEE Transactions on Cloud Computing*, pp. 1–1, 2019.

[9] J. Li, Z. Ming, M. Qiu, G. Quan, X. Qin, and T. Chen, "Resource allocation robustness in multi-core embedded systems with inaccurate information," *Journal of Systems Architecture*, vol. 57, no. 9, pp. 840–849, 2011.

[10] M. Qiu, Z. Chen, J. Niu, Z. Zong, G. Quan, X. Qin, and L. T. Yang, "Data allocation for hybrid memory with genetic algorithm," *IEEE Transactions on Emerging Topics in Computing*, vol. 3, no. 4, pp. 544–555, 2015.

[11] A. Monsifrot, F. Bodin, and R. Quiniou, "A machine learning approach to automatic production of compiler heuristics," in *International Conference on Artificial Intelligence: Methodology, Systems, and Applications*. Springer, 2002, pp. 41–50.

[12] B. Olden, "benchmark suite v," 2010.

[13] Y. Li, Y. Zhao, L. Sun, and M. Shen, "A hybrid sample generation approach in speculative multithreading," *Journal of Supercomputing*, no. 3, pp. 1–33, 2017.

[14] R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarasinghe, J. M. Anderson, S. W. 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.

[15] A. Rogers, M. C. Carlisle, J. H. Reppy, and L. J. Hendren, "Supporting dynamic data structures on distributed-memory machines," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 17, no. 2, pp. 233–263, 1995.

[16] M. K. Prabhu and K. Olukotun, "Exposing speculative thread parallelism in spec2000," in *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*. ACM, 2005, pp. 142–152.

[17] M. C. Carlisle, "Olden: parallelizing programs with dynamic data structures on distributed-memory machines," Ph.D. dissertation, Princeton University, 1996.