JCSE Journal of Computing Science and Engineering

Regular Paper
Journal of Computing Science and Engineering,
Vol. 9, No. 1, March 2015, pp. 20-28

Speculative Parallelism Characterization Profiling in General Purpose Computing Applications

Yaobin Wang*
Department of Computer Science and Technology, Southwest University of Science and Technology, MianYang, China
wyb1982@mail.ustc.edu.cn

Hong An
Department of Computer Science and Technology, University of Science and Technology of China, Hefei, China
han@ustc.edu.cn

Zhiqin Liu, Li Li, Liang Yu, and Yilu Zhen
Department of Computer Science and Technology, Southwest University of Science and Technology, MianYang, China
lzq@swust.edu.cn, lili@swust.edu.cn, yuliang@swust.edu.cn, zhenyilu@swust.edu.cn

Abstract
General purpose computing applications have not yet been thoroughly explored in procedure level speculation, especially in the light-weighted profiling way. This paper proposes a light-weighted profiling mechanism to analyze speculative parallelism characterization in several classic general purpose computing applications from SPEC CPU2000 benchmark. By comparing the key performance factors in loop and procedure-level speculation, it includes new findings on the behaviors of loop and procedure-level parallelism under these applications. The experimental results are as follows. The best gzip application can only achieve a 2.4X speedup in loop level speculation, while the best mcf application can achieve almost 3.5X speedup in procedure level. It proves that our light-weighted profiling method is also effective. It is found that between the loop-level and procedure-level TLS, the latter is better on several cases, which is against the conventional perception. It is especially shown in the applications where their 'hot' procedure body is concluded as 'hot' loops.

Category: Embedded computing

Keywords: Multicore; Thread level speculation; General purpose computing; Profiling; Data dependence

I. INTRODUCTION

General purpose computing is taking an irreversible step toward multicore chip era. The primary problem is that creating legacy parallelized code is difficult, since many current general purpose programs have been written in serial algorithms. Even with a good tool chain including profilers and parallel compilers, automated parallelization has been proven to be a very difficult problem. Although parallel compilers have made significant efforts, they still fail to automatically parallelize general purpose single-threaded programs which have complex data dependence

Open Access http://dx.doi.org/10.5626/JCSE.2015.9.1.20 http://jcse.kiise.org

Received 12 April 2014; **Revised** 5 February 2015; **Accepted** 4 March 2015
*Corresponding Author

structures caused by non-linear subscripts, pointers, or function calls within code sections [1, 2].

To alleviate this problem, one promising way is the use of thread level speculation (TLS) [3-5]. Unlike parallelization for traditional multiprocessors, which requires conservative synchronization for preserving the program semantics, TLS can potentially achieve higher levels of parallelism by exploiting the dynamic parallelism. TLS hardware provides support for speculative threads, which can be executed in parallel; but, it dynamically rolls back and is re-executed if dependences exist between threads. It allows the compiler to automatically parallelize portions of code in the presence of statically ambiguous data dependences, thus extracting parallelism between whatever dynamic dependences actually exist at run-time. Speculative threads are thus not limited by the programmer's or the compiler's ability to find the guaranteed parallel threads. Furthermore, speculative threads have the potential to outperform even the perfect static parallelization by exploiting dynamic parallelism, unlike a multiprocessor which requires conservative synchronization to preserve correct program semantics [6-8].

General purpose computing applications have been accelerated in many researches, but they have not yet been thoroughly explored in the procedure level speculation, especially in the light-weighted profiling way [6, 9-11]. It is therefore important to quantitatively characterize the program behavior in procedure level speculation of these applications, in order to provide insights for future design and research of multicore systems.

In this paper, we propose a speculation model in order to analytically present the parallelism limit of general purpose computing applications, including speculative execution model for procedure & loop, evaluation metrics, and light-weighted profiling mechanism. It takes several of the typical general purpose computing applications from SPEC CPU2000 benchmark to compare their potential speculative speedup, thread size, coverage parallelism, inter-thread data dependence feature in loop, and procedure level speculation. The rest of this paper is organized as follows. Related work is discussed in Section II. The procedure & loop level speculation model and profiling mechanism are described in Sections III and IV, respectively, followed by experiment analysis in Section V. Finally, we conclude in Section VI.

II. RELATED WORK

Difficulty in accelerating general purpose computing applications on multicore platform requires exploiting and making use of more parallelism. There are two traditional ways of obtaining more parallelism. One way is the use of shared memory parallel programming model and language, such as OpenMP. It uses explicit user guidance to exploit thread-level parallelism, and the locking and synchronization variables to achieve the synchronization between the threads. This method is very limited as follows. 1) User guidance can only exploit limited coarse granularity thread-level parallelism, and the fine granularity parallelism cannot be exploited because the complex data dependence makes it difficult for users to recognize the finer parallelism. 2) Coarse-grained locks could synchronize a large amount of data, and they would lead the unrelated code to only run in the serial sequence so that it makes less use of parallelism. Fine-grained locks bring additional system overhead and make parallel programming and debugging become extremely difficult. Another way is to automatically parallelize the serial program by using a parallel compiler. This approach requires the compiler to carefully deal with a lot of data dependencies between the threads. However, in order to maintain the serial semantics in program itself, the compiler has to use relatively conservative parallelism and synchronization strategy, thus greatly affecting the implementation of the thread execution concurrency. Practice has shown that apart from a few scientific computing programs, automatic parallelization for serial program is unfit to exploit parallelism from a large number of general purpose computing applications.

Recent TLS researches have focused on various accelerating technologies in speculation, and most of them [1, 6, 7, 10, 12, 13] pointed out that speculating loop structure is a better choice in TLS. Sohi et al. [14] of the Wisconsin University first used the thread-level speculative technology to accelerate the serial programs in the Multi-Scalar project. Prabhu and Olukotun [6] evaluated the performance potential of a multiprocessor-based approach through a radical manual optimization way, but a purely manual way could not reduce the programmers' workload. Oplinger et al. [15] proposed to treat procedure structure as thread candidates supplement in the TLS. Hydra [13] moved speculative data into each core's private space and leveraged the cache coherence protocol for disambiguation. Research by the CMU STAMPede team [12] proposed to implement scalable conflict detection and version management for TLS, but their cache coherence protocols were shown to be very complicated. Du et al. [16] focused on the software value prediction technology to improve the speculative parallel performance in desktop benchmarks. Johnson et al. [1] showed that speculative thread partition is performed by a profile embedded into the program. Our preliminary work has also focused on the online profiling technology in loop level speculation [17].

Meanwhile, general purpose computing applications have been accelerated in many new researches, such as by software runtime system [11], GPU [10], or FPGA [9], etc. However, they have not yet been explored light-weightedly in procedure level speculation.

III. SPECULATION MODEL

A. Speculative Execution Model for Procedure

A procedure's boundaries often separate the fairly independent computations, and the local variables would not violate the outer program. Multiple calls would be executed in parallel by ignoring thread boundaries, until the call returns.

The procedure level execution model is shown in Fig. 1. To speculate at the procedure level in our model, we concurrently executed the called procedure with the code following the return of the procedure. Notice that it is the latter that executes speculatively. We propose to use a new thread to execute the called procedure and have the original thread execute the rest of the caller code, speculatively. A data dependence violation occurs if the code following the return reads a location before the callee thread writes to that location. The same mechanism that is used for loop-level parallelism can be used to ensure that the data dependencies are satisfied.

B. Speculative Execution Model for Loop

Loop iterations always carry out similar operations to the same data set, and are independent from each other. The data dependence between iterations is regular. They provide a runtime sequence of predictable and naturally load-balanced threads, leaving dependence as the primary overhead to potentially interfere with speculation.

The loop level execution model is shown in Fig. 2, for comparison. Fig. 2(a) shows the traditional execution model and Fig. 2(b) shows the parallel execution model. At the beginning of the parallel execution, the main processor informs all of the other processors (we call it speculative processor) to load and execute different iterations of the loop, by sending a 'Loop_Start' signal to them. In the process of parallel execution, only the head processor can directly write to memory, and all of the other processor's memory references will be cached in its speculative buffer. The next processor will become the new head processor after the current head processor is committed. A new iteration will be loaded and executed after a processor has committed its result into memory. When a processor finds that the exit condition of the loop becomes true, a 'Loop_End' signal would be sent to all of the other processors to finish the parallel execution of the specific loop structure; and only the main processor will continue running the code following the loop.

Our model is platform-independent to focus more on the program behavior itself. It incurs no overhead in executing the parallel threads, and it can delay the computation of a thread to perfectly avoid the need to rollback any of the computation. We can use the optimal model to derive an upper bound on the performance achievable by using any possible synchronization optimizations.

IV. PROFILING MECHANISM

A. Evaluation Metrics in Profiling

Coverage parallelism: According to Amdahl's Law, it is low parallelism coverage necessarily resulting in poor performance. So, it is the first evaluation metric in our profiling. We refer to the percentage of code executed under parallel execution as the parallelism coverage.

Inter-thread data dependence violation: It is known that all of the overheads for conflict detection, synchronization, rollback, and restart are caused by this. So, it is the most important evaluation metric. The lower Inter-thread data dependence violations will give the higher performance.

Thread size: Different thread size would cause load imbalance problem in speculation. Short thread cannot payoff the overhead of speculative execution. Long thread may lead to speculative buffer overflow, which must stall

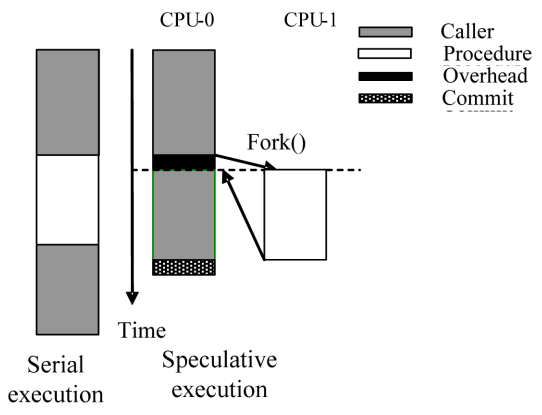


Fig. 1. Speculative execution model for procedures.

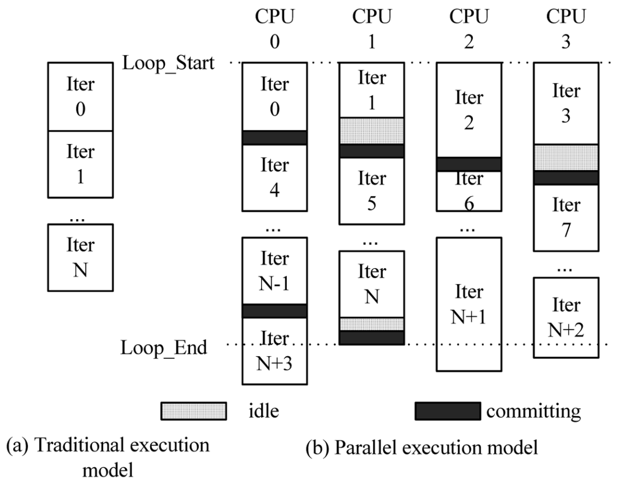


Fig. 2. Speculative execution model for loops.

the execution of the thread. Proper and similar thread size is good for TLS performance. So, it is also an important evaluation metric.

Inter-thread control dependence feature: The predication rate for inter-thread control flow shows the control dependence violations feature among the threads. If the predication is wrong, the speculative thread has to restart, which badly affects the performance. Fortunately, inter-thread control dependence between speculative threads can be easily resolved by stride-value predication technology [15]. So, we did not analyze the Inter-thread control dependence feature in profiling.

B. Memory Access Type in Speculation

Memory dependence violations have a tremendous impact on the TLS performance. There are three categories arising from this, as follows: the access in the Global Data segment (including initialized and uninitialized global data access), Heap segment, and Stack segment. Corresponding to the variable visits in C language, global variables and static variables visits represent the access in the Global Data segment; local function variables visits take place in the Stack segment, as well as visits in the procedures context preservation and recovery (mainly occurred during the function calls). The visit in the memory space allocated by dynamic allocation function '`malloc()`' refers to the Heap access, and here, we call it as dynamic variables visits.

Through the scope judgment of the memory access addresses, the memory access type can be determined at runtime. Only the global and dynamic variables visited in procedure would violate data dependence with its subsequent code, so they're the profiling objects in procedure level speculation. For loop level speculation, conservatively, the profiling objects are the global variables, dynamic variables, and the current variables in Stack segment.

C. Analysis Method for Data Dependence Violations

Definition 1. Produce-distance: the instruction numbers, from the beginning of the thread to the last, write operation for a specific memory address.

Definition 2. Consume-distance: the instruction numbers from the beginning of the thread to the first read operation for a specific memory address.

To speculate either at procedure or loop level, the inter-thread data dependence can be abstracted as a producer/consumer model. The memory write operations play the role of producing data, while the memory read operations are in charge of consuming data. To describe the data dependence violation, we introduced the two terms here, 'produce-distance' and 'consume-distance', as shown in

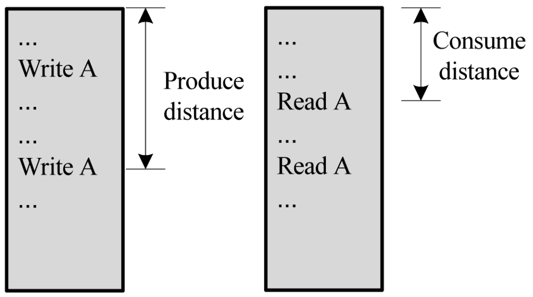


Fig. 3. Produce-distance and consume-distance.

Fig. 3. The produce-distance means the instruction numbers from the beginning of the thread to the last write instruction, for a specific memory address; and consume-distance means the instruction numbers from the beginning of the thread to the first read instruction, for a specific memory address.

We should point out in the definition that we used the instruction numbers to represent the running cycles for our profiling tool to execute one instruction per cycle. By definition, we can see that either the produce-distance or the consume-distance is a dynamic concept; in other words, it is a concept relative to program's one specific run. In different runtime environment, the different branches will be executed in the same thread. So, either of them is calculated for specific data, and both of them must be calculated at running time. For thread *i* and its successor thread *i*+1, starting at almost the same time, if the latter's consume-distance is less than the former's produce-distance, there will be a dependence violation.

In this paper, we selected the ratio of consume-distance to produce-distance (α) to evaluate the inter-thread data dependence pattern. There will be a violation when $\alpha<1$. Smaller ratio number means lower performance. For example, when α is close to 0, it means that the speculative threads are running serially, even with perfect synchronization strategy.

D. Profiling Framework

We used the GUN Prof (Gprof) tool to choose the 'hot' area in the program, which can be used to analyze the running time proportion of procedures in the given program and to generate the function call graphs. In our study, we picked out the 'hot' procedures and loops that occupy more than 5% of the total program execution time, and then these procedures were treated as the further analysis input.

The profiling tools we developed in our investigation are named as ProFun and ProLoop [18], and all of them were extended from sim-fast, the fasted simulator of SimpleScalar tool set which executes one instruction per cycle. ProFun is used to profile the procedure, and ProLoop is for Loop.

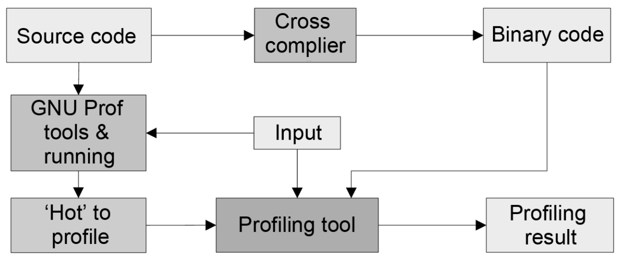


Fig. 4. Profiling framework.

The profiling framework is shown in Fig. 4.

(1) After the source code was profiled by Gprof, a list of the 'hot' procedures will help us to find out the 'hot' parts for ProFun and ProLoop each.

(2) Through an input file with a 'hot' procedures list for ProFun or adding the flags to 'hot' loop body for ProLoop, the tool can get the memory access addressing the range of these 'hot' fragments.

(3) The binary code is transformed into the disassembled instructions by objdump tool.

(4) Then, we conduct the profiling by capturing 'load' and 'store' operations to work out the experimental results.

The key data structure and calculation mechanism in profiling are shown in following sections.

E. Key Data Structure and Calculation Mechanism

Fig. 5 shows two key list structure named 'call_list' and 'p_write_list' in ProFun design. They are used together to record and compute all of the profiling data in speculation. A similar mechanism is also used in ProLoop.

The 'call_list' is used to reserve the relevant profiling data for each speculative thread (procedure calls or loop iterations). As shown in Fig. 5, the 'call_list' is composed of 'call_list_entry_t', which will record the function id, start time to identify the only call itself, and end time for computing the profiling goals.

Another 'p_write_list' structure is used to remember the writes by the speculative threads for every memory address. The 'p_write_hash_t' has 65,536 items as the memory space has 2^{32} address space; 'p_write_list' is composed of the 'p_write_hash_entry_t', which is used to record the function id, start time to identify the specific procedure calls and access address, write time, write time in TLS parallelism to compute the dependence distance and speedup.

Memory dependence distance in speculation is gained in the following way:

(1) When the profiling tool enters a 'jal' or 'jalr' instruction followed by a name in the input file, with 'hot' names, it will create a 'call_list_entry_t'.

(2) For store operations, it will record them in 'p_write_list'; and for load operations, it will find the item with the same access address in 'p_write_list'.

(3) Then, it would compute the memory dependence distance with the relative call's information in 'call_list', using the analysis method described in Section IV-C.

There is a search for the load operation to find its relative call in 'call_list', and it will decrease the profiling performance if too many calls for one procedure are searched serially. So, we defined the 'call_hash_list' to accelerate the search speed as shown in Fig. 5.

We can get the speculative speedup as follows:

(1) When a speculative thread fork is encountered, the start time in speculation is stored, and the header thread executes.

(2) When the later speculative thread(s) begins to execute, its start time is set as the same as the header's.

(3) When dependence violation happens between the two speculative threads, the strategy is to delay the computation of a thread to perfectly avoid the need to rollback any of the computation.

(4) Finally, the speedup is calculated by the total execution time in both speculation and serial mode.

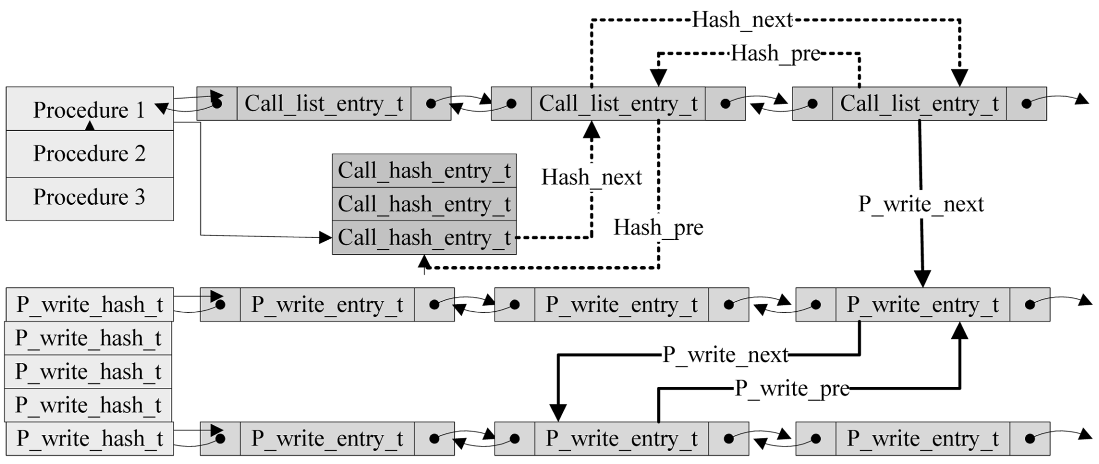


Fig. 5. Key data structure in ProFun.

Table 1. SPEC CPU2000 description

Program	Input size	Description
gzip	lgred	Compression
vpr	lgred	FPGA circuit placement and routing
gcc	lgred	C programming language compiler
mcf	lgred	Combinatorial optimization
parser	lgred	Word processing
bzip2	lgred	Compression
twolf	lgred	Place and route simulator

Table 2. Average thread size in loop and procedure level speculation

Benchmark	Loop	Procedure
gcc	4.50E+03	3.30E+05
bzip2	9.50E+03	4.60E+07
vpr	2.90E+03	3.60E+02
mcf	2.30E+04	8.00E+06
parser	1.90E+06	1.20E+02
towlf	2.30E+04	4.00E+05
gzip	9.80E+01	1.30E+07

V. EXPERIMENT ANALYSIS

For TLS used to accelerate traditional serial codes, we have chosen gzip, vpr, gcc, mcf, parser, bzip2, and twolf that are written in C language as the representative general purpose computing applications from SPEC CPU2000 benchmark suite. They're classic and more suitable for TLS performance evaluation. The description of SPEC CPU2000 is shown in Table 1.

Then, by running the profiling tools, we will focus on analyzing the coverage parallelism, thread size, inter-thread data dependence feature and speedup in each benchmark. All experiments are done in Linux on the ×86 platform. The destination ISA in the tool is PISA, and the cross compiler from the gcc-2.7.2.3 with reconstructed backend is provided by SimpleScalar tool set.

A. Speedup

Fig. 6 shows the compared speedup in loop and procedure level speculation. It shows the max speedup potential under infinite core numbers in TLS. Disappointingly, most of their speedups are so low that they cannot reach 2 in loop level speculation. The best gzip application can only get a 2.4X speedup. Similar low speedup results are also available from prior studies [6, 15]. It is proven that our light-weighted profiling mechanism is also effective.

The results are different in procedure level speculation in two aspects. (1) First, it shows that parser, gcc, bzip2, gzip, and vpr's speedups are also low in procedure level speculation. (2) However, it is surprising that mcf and twolf get much higher speedups than in loop level speculation. The best mcf application can get almost 3.5X speedup in the procedure level speculation. In other words, the mcf is more suitable for procedure level speculation, rather than in loop level.

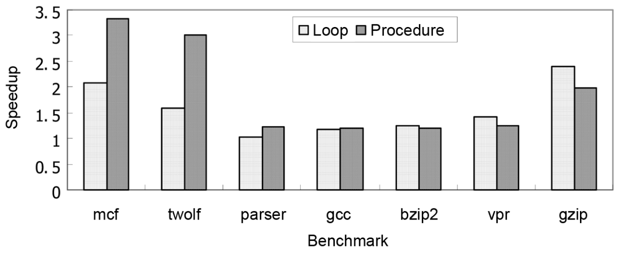


Fig. 6. Speedup in loop and procedure level speculation.

The performance improvement in mcf and twolf is so interesting; and the conventional perception is that speculative performance in loop level is better than that in procedure level. Then, we will analyze the applications from their thread size, parallelism coverage, and inter-thread data dependence feature (α), according to the metrics described in Section IV-A.

B. Average Thread Size

From Table 2, it is also to be noticed that the speculative thread size of mcf and twolf in procedure level speculation is bigger in Table 2. We will discuss it further by the assembled code analysis, as follows:

The 'hot' procedures in mcf are 'refresh_potential', 'price_out_impl', and 'primal_bea_mpp'. The 'refresh_potential' and 'price_out_impl' take up nearly 86% of the total running time. In the assemble code analysis, it is found that these two 'hot' procedure concludes some 'hot' loops in them. So its size is bigger than that in loop level speculation. In this way, former inter-thread data dependences in loop level speculation are held in a single procedure level thread.

The reason in twolf is similar. It is found that the four hot loops in program are respectively included in four hot procedures ('term_newpos_b', 'new_dbox', 'term_newpos_a', and 'new_dbox_a'), and these four hot procedures are included in the main loop body. By speculating at the proper procedure level that the loop level could not, the inherently higher independence in procedure structure brings less inter-thread data dependence violations. Therefore, they obtain higher speedup in procedure level speculation.

C. Parallelism Coverage

Table 3 shows the compared parallelism coverage

Table 3. Parallelism coverage in loop and procedure level speculation

Benchmark	Loop (%)	Procedure (%)
gcc	25	29
bzip2	79	77
vpr	49	42
mcf	92	95
parser	43	47
towlf	89	79
gzip	84	84

Table 4. Average α value in loop and procedure level speculation

Benchmark	Loop	Procedure
gcc	0.33	0.37
bzip2	0.24	0.34
vpr	0.29	0.31
mcf	0.37	0.67
parser	0.18	0.16
towlf	0.20	0.70
gzip	0.22	0.31

results. It is to be noticed that the coverage ratios for each application are similar in both loop and procedure level speculation. It shows that the 'hot' procedure is either in a 'hot' loop or holds one or some 'hot' loop structures. So, it is pointed out that the loop structure is the origin of effective speculative level parallelism.

Also, it has been observed that coverage ratios for gcc, vpr, and parser are all below 50%, so their ideal max speedup is less than 2, computed by Amdahl's law. It means that low coverage ratio leads to inevitably low speedup.

D. Inter-thread Data Dependence Feature

The compared inter-thread data dependence feature (α) is shown in Table 4. This is the point why vpr, gcc, bzip2, gzip, and parser's max potential speedups are so low. In the loop level speculation, the reason is that each of their average α is too small. Most of their α are less than 0.3, which means the data dependence violations can badly hurt the TLS performance. They get low speedups for the similar reasons in procedure level speculation. It shows that severe inter-thread data dependence violations can also badly affect speculative procedure level parallelism. In overall, it is pointed out that the inherent ubiquitous inter-thread data dependences in general computing applications can badly affect their parallelizing performance in TLS, whether it is in loop level or in procedure level speculation.

In Table 4, it has been observed that the biggest change in mcf and twolf is their average α number. For them, bigger α number means lower inter-thread data dependence violations affect in speculation. That leads them to obtain higher speedup in procedure level speculation.

As a result, it is found that because of the inherently better independence in procedure structure, some general purpose computing applications may obtain higher speedup in procedure level speculation than that in loop level. This is especially in applications that their 'hot' procedure body concludes 'hot' loops.

VI. CONCLUSIONS

This paper presents ways to analyze speculative parallelism characterization in general purpose computing applications. It makes the following main contributions:

• We explored the loop and procedure level speculative potential in general purpose computing applications and showed that most applications are not suitable for TLS.

• It is found that the procedure-level TLS would perform better than loop-level TLS, when the 'hot' procedure body concludes 'hot' loops.

• It is suggested that by speculating the proper procedure level that the loop-level TLS could not, the inherently higher independence in procedure structure would bring less inter-thread data dependence violations.

ACKNOWLEDGMENTS

This work is supported financially by the National Natural Science Foundation of China grants (No. 61202044), the National Basic Research Program of China under contract (No. 2011CB302501), the National Hi-Tech Research and Development Program of China under contracts (No. 2012AA010902 and 2012AA010303), the National Science & Technology Major Projects (No. 2011ZX01028-001-002-3), the Research Fund of Southwest University of Science and Technology (No. 12zxwk08).

REFERENCES

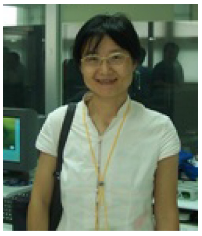
1. T. A. Johnson, R. Eigenmann, and T. N. Vijaykumar, "Speculative thread decomposition through empirical optimization," in *Proceedings of the 12th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, San Jose, CA, 2007, pp. 205-214.
2. C. Tian, M. Feng, and R. Gupta, "Speculative parallelization using state separation and multiple value prediction," *ACM SIGPLAN Notices*, vol. 45, no. 8, pp. 63-72, 2010.
3. A. Munir, S. Ranka, and A. Gordon-Ross, "High-perfor-

mance energy-efficient multicore embedded computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, no. 4, pp. 684-700, 2012.

4. D. Prountzos, R. Manevich, K. Pingali, and K. S. McKinley, "A shape analysis for optimizing parallel graph programs," *ACM SIGPLAN Notices*, vol. 46, no. 1, pp. 159-172, 2011.
5. A. Raman, H. Kim, T. R. Mason, T. B. Jablin, and D. I. August, "Speculative parallelization using software multi-threaded transactions," *ACM SIGARCH Computer Architecture News*, vol. 38, no. 1, pp. 65-76, 2010.
6. M. K. Prabhu and K. Olukotun, "Exposing speculative thread parallelism in SPEC2000," in *Proceedings of the 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Chicago, IL, 2005, pp. 142-152.
7. A. Kejariwal, X. Tian, W. Li, M. Girkar, S. Kozhukhov, H. Saito, et al, "On the performance potential of different types of speculative thread-level parallelism," in *Proceedings of the 20th Annual International Conference on Supercomputing (ICS)*, Cairns, Australia, 2006, p. 24.
8. K. Selvamani, and T. M. Taha, "Estimating critical region parallelism to guide platform retargeting," in *Proceedings of the 43rd ACM Southeast Regional Conference*, Kennesaw, GA, 2005, pp. 168-173.
9. J. Auerbach, D. F. Bacon, I. Burcea, P. Cheng, S. J. Fink, R. Rabbah, and S. Shukla, "A compiler and runtime for heterogeneous computing," in *Proceedings of the 49th Annual Design Automation Conference*, San Francisco, CA, 2012, pp. 271-276.
10. M. Samadi, A. Hormati, J. Lee, and S. Mahlke, "Paragon: collaborative speculative loop execution on GPU and CPU," in *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units*, London, UK, 2012, pp. 64-73.
11. P. Yiapanis, D. Rosas-Ham, G. Brown, and M. Lujan, "Optimizing software runtime systems for speculative parallelization," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 9, no. 4, article no. 9, 2013.
12. J. G. Steffan, C. Colohan, A. Zhai, and T. C. Mowry, "The STAMPede approach to thread-level speculation," *ACM Transactions on Computer Systems(TOCS)*, vol. 23, no. 3, pp. 253-300, 2005.
13. L. Hammond, B. A. Hubbert, M. Siu, M. K. Parbhu, M. Chen, and K. Qlukolun, "The Stanford Hydra CMP," *IEEE Micro*, vol. 20, no. 2, pp. 71-84, 2000.
14. G. S. Sohi, S. E. Breach, and T. N. Vijaykumar, "Multiscalar processors," in *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA'95)*, Barcelona, Spain, 1995, pp. 414-425.
15. J. T. Oplinger, D. L. Heine, and M. S. Lam, "In search of speculative thread-level parallelism," in *Proceedings of International Conference on Parallel Architectures and Compilation Techniques (PACT'99)*, Newport Beach, CA, 1999, pp. 303-313.
16. Z. H. Du, C. C. Lim, X. F. Li, C. Yang, Q. Zhao, and T. F. Ngai, "A cost-driven compilation framework for speculative parallelization of sequential programs," in *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, Washington DC, 2004, pp. 71-81.
17. Y. Liu, H. An, B. Liang, and L. Wang, "An online profile guided optimization approach for speculative parallel threading," in *Advances in Computer Systems Architecture*, *Lecture Notes in Computer Science vol. 4697*, Heidelberg: Springer, pp. 28-39, 2007.
18. Y. Wang, H. An, B. Liang, L. Wang, & R. Guo, "OpenPro: a dynamic profiling tool set for exploring thread-level speculation parallelism," in *Proceedings of the International Conference on Computer and Electrical Engineering (ICCEE)*, Phuket Island, Thailand, 2008, pp. 256-260.

Yaobin Wang

Yaobin Wang has obtained his Master's degree in Computer Science and Ph.D. in Computer Architecture both from the University of Science and Technology of China. Currently, he is working as an associate professor at Southwest University of Science and Technology, Mianyang. His research interests are focused on computer architecture and parallel computing.



Hong An

Hong An has obtained her Master's degree in Computer Science and Ph.D. in Computer Architecture both from the University of Science and Technology of China. Currently, she is working as a professor at University of Science and Technology of China, Hefei. Her research interests are focused on computer architecture and parallel computing.

Zhiqin Liu

Zhiqin Liu has obtained her Master's degree in Computer Science from Southwest University of Science and Technology. Currently, she is working as a professor at Southwest University of Science and Technology, Mianyang. Her research interests are focused on computer network and high performance computing.

Li Li

Li Li has obtained his Ph.D. degree in Computer Science from Sichuan University. Currently, he is working as a lecturer at Southwest University of Science and Technology, Mianyang. His research interests are focused on computer hardware.

Liang Yu

Liang Yu has obtained his Master's degree in Computer Science from Waseda University. Currently, he is working as a lecturer at Southwest University of Science and Technology, Mianyang. His research interests are focused on computer architecture.

Yilu Zhen

Yilu Zhen has obtained her Master's degree in Computer Science from Southwest University of Science and Technology. Currently, she is working as a lecturer at Southwest University of Science and Technology, Mianyang. Her research interests are focused on computer engineering.