Balancing Thread Partition for Efficiently Exploiting Speculative Thread-Level Parallelism

Yaobin Wang, Hong An, Bo Liang, Li Wang, Ming Cong, and Yongqing Ren

Department of Computer Science and Technology,
University of Science and Technology of China, Hefei 230026, China
Key Laboratory of Computer System and Architecture,
Chinese Academy of Sciences, Beijing 100086, China
wyb1982@mail.ustc.edu.cn, han@ustc.edu.cn,
{boliang,wangliii,mcong,renyq}@mail.ustc.edu.cn

Abstract. General-purpose computing is taking an irreversible step toward on-chip parallel architectures. One way to enhance the performance of chip multiprocessors is the use of thread-level speculation (TLS). Identifying the points where the speculative threads will be spawned becomes one of the critical issues of this kind of architectures. In this paper, a criterion for selecting the region to be speculatively executed is presented to identify potential sources of speculative parallelism in general-purpose programs. A dynamic profiling method has been provided to search a large space of TLS parallelization schemes and where parallelism was located within the application. We analyze key factors impacting speculative thread-level parallelism of SPEC CPU2000, evaluate whether a given application or parts of it are suitable for TLS technology, and study how to balance thread partition for efficiently exploiting speculative thread-level parallelism. It shows that the inter-thread data dependences are ubiquitous and the synchronization mechanism is necessary; Return value prediction and loop unrolling are important to improve performance. The information we got can be used to guide the thread partition of TLS.

1 Introduction

We have witnessed that chip multiprocessors (CMPs), or multi-core processors, have become a common way of reducing chip complexity and power consumption while maintaining high performance. General-purpose computing is taking an irreversible step toward on-chip parallel architectures [1]. The ability to place multiple cores or many cores on the same chip will significantly increase the communication bandwidth and decrease the communication latency seen by threads executing on different processing cores. This enables the exploitation of finer-grained thread-level parallelism on a multicore chip as compared to a conventional symmetric multiprocessor (SMPs). But current parallel software is limited since many programs have been written using serial algorithms.

On the one hand, software transformations are a possible way for extracting some parallelism from these codes. Unfortunately, although parallel compilers have

M. Xu et al. (Eds.): APPT 2007, LNCS 4847, pp. 40–49, 2007.
© Springer-Verlag Berlin Heidelberg 2007

made significant efforts, they still fail to automatically parallelize general-purpose single-threaded programs which have complex data dependence structures caused by non-linear subscripts, pointers, or function calls within code sections [2,3,4]. On the other hand, many applications may still turn out to have a large amount of parallelism, but are still only hand-parallelizeable with state-of-the-art parallel programming models. Manual parallelization can provide good performance, but typically requires not only a different initial program design but also programmers with additional skills and efforts. In a word, the primary problem is that creating parallelized versions of legacy code is difficult.

Can simple hardware support on multicore chip help to parallelize general-purpose programs? To parallelize these codes, researchers have proposed Thread-Level Speculation (TLS) that allows to parallelize regions of code in the presence of ambiguous data dependence, thus extracting parallelism whatever dynamic dependences actually exist at run-time [5,6,7]. Speculative CMPs use hardware to enforce dependence, allowing a parallelizing compiler to generate multithreaded code without needing to prove independence. In these systems, a sequential program is decomposed into threads to be executed in parallel; dependent threads cause performance degradation, but do not affect correctness. Speculative threads are thus not limited by the programmer's or the compiler's ability to find guaranteed parallel threads. Furthermore, speculative threads have the potential to outperform even perfect static parallelization by exploiting dynamic parallelism, unlike a multiprocessor which requires conservative synchronization to preserve correct program semantics. But for performance reasons, thread decomposition is expected to reduce the run-time overheads of data dependence, inter-thread control-flow misprediction, and load imbalance. Unfortunately, these kinds of threads are very hard to find, especially in non-numerical programs. Identifying the points where the speculative threads will be spawned becomes one of the critical issues of this kind of architectures.

Several hardware designs have been proposed for this speculative thread-level parallelism (STP) model [5,6,7,9,10], but so far the speedup achieved on large general-purpose code has been limited. The decision on where to speculate can make a large difference in the resulting performance. If the performance is poor, we gain little insight on why it does not work, or whether it is the parallelization scheme or machine model (or both) that should be improved. As a consequence, poor results may not reflect any inherent limitations of the STP model, but rather the way it was applied.

The goal of this paper is to propose a criterion for selecting the region to be speculatively executed and to identify potential sources of speculative parallelism in general-purpose programs. We also evaluate whether a given application or parts of it are suitable for TLS technology, and study how to balance thread partition for efficiently exploiting speculative thread-level parallelism.

The rest of this paper is organized as follows. In Section 2 we describe the STP models for subroutine and loop level speculation. The analysis method is described in Section 3, followed by experiment analysis in Section 4. Finally we conclude in Section 5.

2 Speculative Thread-Level Parallel Execution Model

2.1 Candidate Threads

The thread partition is based on the control flow information, usually choose loop and subroutine structures as the candidate threads. For subroutine, its boundaries often separate fairly independent computations, the local variables wouldn't violate with the outer program; and for the loop body, every iteration does the similar operations to the same data set, and is independent each other. The data dependence between iterations is regular. Both of them are good choices for candidate threads.

2.2 Speculative Execution Model for Loops

The speculative execution model for loops is shown in Fig.1, for comparing, Fig.1(a) shows the traditional execution model and Fig.1(b) shows the speculative execution model. At the beginning of the speculative execution, the main processor informs all the other processors to load and execute different iterations of the loop by sending a "Loop_Start" signal to them. In the process of speculative execution, only the head processor can write to memory directly, and all the other speculative 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 committed. A new iteration will be loaded and executed after a processor committed its result into memory. When a processor found that the exit condition of the loop becomes true, a "Loop_End" signal would be send to all the other processors to finish the speculative execution of the specific loop structure, and only the main processor continue running the code followed the loop.

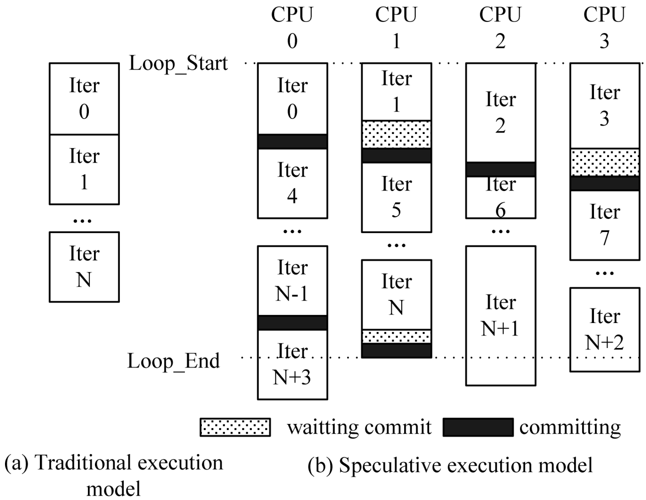

Fig. 1. Speculative execution model for loops

2.3 Speculative Execution Model for Subroutines

As shown in Fig.2, when a speculative subroutine call takes place, a new processor will be selected to run the code followed the call speculatively with the predicted return value and the old processor concurrently run the subroutine. The new processor's memory references will be cached in its speculative buffer while the old processor can write directly into memory. After the old processor complete the execution of the subroutine, the real return value come into being and compared with the prediction value, if miss prediction detected, the new processor must rollback to correct the execution.

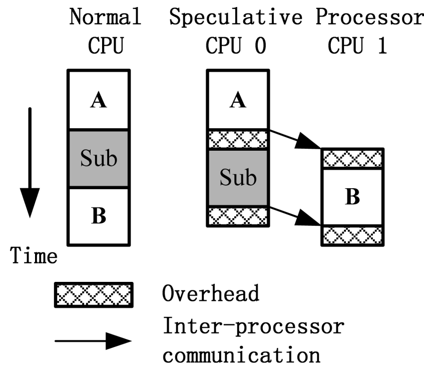

Fig. 2. Speculative execution model for subroutines

3 Analysis Method

3.1 Basic Criterion for Selecting Threads

Granularity and inter-thread data dependence pattern is the most important criterions for selecting candidate threads. Long thread may lead to speculative buffer overflow which must stall the execution of the thread, while short thread cannot payoff the overhead of speculative execution. Different from subroutine, loop slicing and unrolling can be used to control the granularity of a loop. Inter-thread data dependence pattern is the other basic criterion for both loop and subroutine, and we will propose two new concepts to describe this issue in section 3.2. Besides granularity and inter-thread data dependence pattern, there are some other distinguished criterions, such as type of return value and return value prediction rate that should be used to choose thread from subroutine structure. In all of them, value predication rate, data dependence, thread granularity are foremost in the TLS parallelism, the reasons are as follows:

The value predication rate shows the control dependence violations among the threads, the data dependence would cause the violations, and the granularity shows the problem about the thread balance.

3.2 Analysis Method for TLS Parallelism

The inter-thread data dependence can be abstracted as a producer/consumer model, write operation is data producing while read operation is data consuming. To describe the data dependence violation, we introduce two terms here: "produce- distance" and "consume-distance", as shown in 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. Either of them is a concept relative to program's one specific execution and both of them must be calculated at running time.

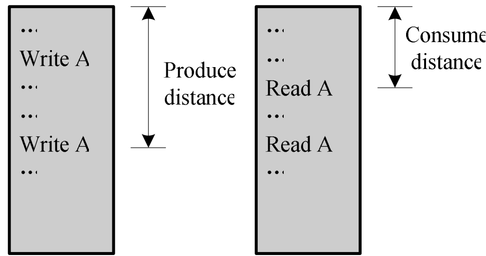

Fig. 3. Produce-distance & Consume-distance

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 under the assumption of that all processor execute instructions at a same speed. In this paper we select the ratio of consume-distance to produce-distance to evaluate the inter-thread data dependence pattern. To facilitate the description, we call a inter-thread data dependence as a Deadly Dependence if the ratio is less than 1.0, Dangerous Dependence for the ratio between 1.0 and 2.0 and Safe Dependence for the ratio larger than 2.0.

4 Experiment Analysis

4.1 Experimental Environment and Tools

The profiling tools we used in our investigation named ProFun, ProRV and ProLoop, and all of them are extended from sim-fast, the fasted simulator of SimpleScalar tool set which execute one instruction per cycle. ProFun and ProRV are used to profile the subroutines and ProLoop is for Loop. All the tests were achieved on an x86 machine running Linux system, the compiler we used is modified from gcc-2.7.2.3, and the benchmarks are selected from SPEC CPU2000.

Firstly, we pick out subroutines that occupy more than 5% of the total program execution time, as the inputs of ProRV and ProFun by using Gprof tool, and the input

loop structures of ProLoop were selected in the subroutines acquired above. And then by running the profiling tools, we profiled execution time distribution and the return value prediction rate of subroutines with different return value types, the granularity distribution and the inter-thread data dependence pattern of both subroutines and loops, and the ideal speedup achieved by speculative execution.

4.2 Experiment Results

4.2.1 Return Value Prediction Rate

The “sparse int” subroutine always returns zero except it errors and can be well predicted (e.g. boolean type), it is necessary to separate it from int type and we named it as sparse int. As shown in Fig.4, we found that the last-value prediction scheme is better then the stride, the “sparse int” type achieve a prediction rate about 80%, and the prediction rate of float is almost zero. For the “void” and “sparse int” subroutines take up most of the execution time and they’re easy to predicate, we can say that the source of speculative thread-level parallelism is abundant in general-purpose applications.

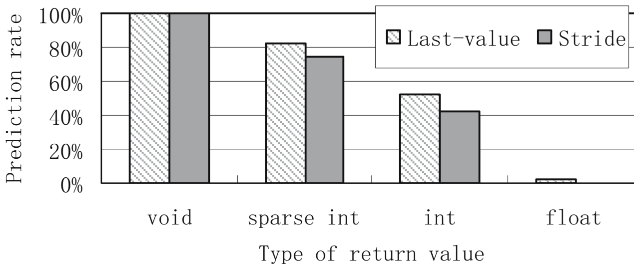

Fig. 4. Prediction rate of different return value types

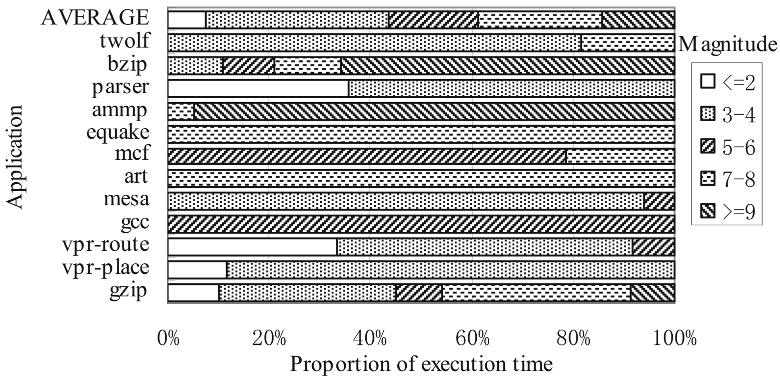

Fig. 5. The thread granularity for subroutines

Fig. 6. The thread granularity for loops

4.2.2 The Thread Granularity

Figure 5 shows the execution time distribution of subroutines with different instruction numbers, and from it we can see that the execution time distribution of them varied widely. For only the subroutines with a length of 10^3-10^6 instructions are suitable for speculative execution and luckily we found that they take about 55% of total execution time.

Figure 6 shows the execution time distribution for Loop structure, and we found that most of the loop iteration is shorter than 10^4 instructions. It means that loop unrolling should be frequently used to achieve a larger iteration.

4.2.3 Memory Dependence Distribution

Figure 7 and Fig.8 show the memory dependence distribution for subroutine and loop structures. From Fig.7 we can see that the distribution characteristic is quite varied,

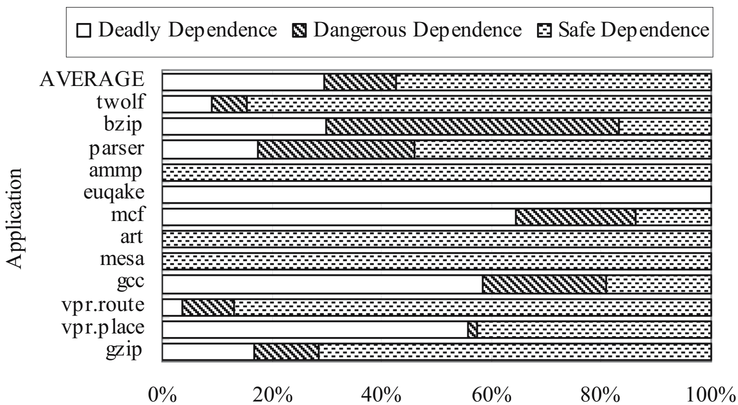

Fig. 7. Memory dependence distribution for subroutine speculation

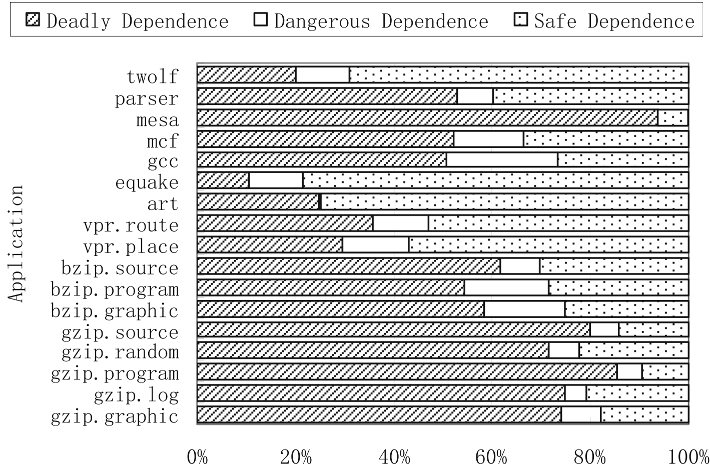

Fig. 8. Memory dependence distribution for loop speculation

and in average, there are about 30% dependences are deadly dependence, 12% are dangerous dependence and about 58% are safe dependence. Only ammp, art and mesa almost have no deadly dependence. The situation for loop structure is more pessimistic, all the applications have deadly dependence as shown in Fig.8. It means that for application of SPEC CPU2000, the data dependences are ubiquitous; to achieve a better performance, synchronization mechanism is quite necessary.

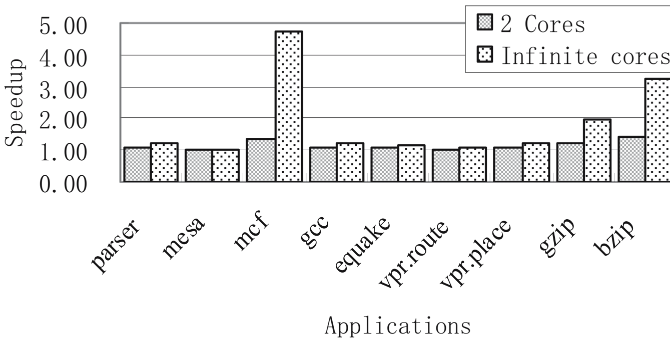

Fig. 9. Speedup of subroutine speculation

4.2.4 Speedup

Figure 9 and Fig.10 show the potential speedup of speculative execution for subroutine and loop structures when using different core numbers. As we can see in Fig.9, even the highest speedup we can achieve by speculative execution of subroutines using infinite cores, can not exceed 5, and most of the applications can only achieve the speedup lower than 3 even using infinite cores. The situation is more awful when limited the number of cores to 2. The similar situation was also appeared in Fig.10: the highest speedup can only be about 5.3 even for infinite cores and allowing speedup nest loop. This pessimistic result is consistent with the analysis mentioned in section 4.2.3.

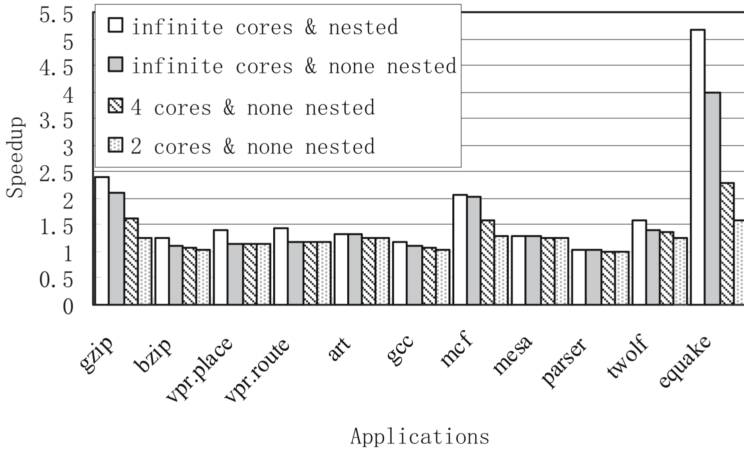

Fig. 10. Speedup of loop speculation

5 Conclusions

In this paper, a criterion for selecting the region to be speculatively executed is presented to identify potential sources of speculative parallelism in general-purpose programs. The dynamic profiling method has been provided to search a large space of parallelization schemes. We analyze the key factors impacting speculative thread-level parallelism of SPEC CPU2000, such as the return value prediction rate of subroutines with various prediction methods, the memory dependence, the granularity of loops and subroutines, and so on. We evaluate whether a given application or parts of it are suitable for TLS technology, and study how to balance thread partition for exploiting speculative thread-level parallelism. It shows that the source of speculative thread-level parallelism is abundant in general-purpose applications, the value predication and loop unrolling technology can greatly improve the TLS performance.

Acknowledgement

This work has been supported by the grant from Intel (PO#4507176412), the National Natural Science Foundation of China (60373043 and 60633040) and the National Basic Research Program of China (2005CB321601).

References

1. Asanovic, K., Bodik, R., et al.: The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report, No.UCB/EECS-2006-183, UC Berkeley (2006)
2. Zhai, A., Colohan, C.B., Steffan, J.G., et al.: Compiler optimization of scalar value communication between speculative threads. In: ASPLOS-10, San Jose, California (2002)
3. S.W. Liao, et al.: SUIF Explorer: An Interactive and Interprocedural Parallelizer. In: PPoPP 1999 (1999)

4. Miller, B.P., et al.: The Paradyn Parallel Performance Measurement Tools. IEEE Computer 11, 37–46 (1995)
5. Sohi, G.S., Breach, S.E., Vijaykumar, T.N.: Multiscalar Processors. In: 22nd Annual International Symposium (1995)
6. Hammond, L., Willey, M., Olukotun, K.: Data Speculation Support for a Chip Multiprocessor. In: ASPLOS-VIII, San Jose, CA (1998)
7. Steffan, J.G., Mowry, T.: The potential for using thread-level data speculation to facilitate automatic parallelization. In: HPCA-4, Las Vegas, NV (1998)
8. Oplinger, J.T., Heine, D.L.: In Search of Speculative Thread-Level Parallelism. In: Malyshkin, V. (ed.) Parallel Computing Technologies. LNCS, vol. 1662, Springer, Heidelberg (1999)
9. Akkary, H., Driscoll, M.A.: A Dynamic Multithreading Processor. MICRO-31, Dallas, TX (1998)
10. Krishnan, V., et al.: Hardware and Software Support for Speculative Execution of Sequential Binaries on a Chip- Multiprocessor. In: Supercomputing 1998, Melbourne, Australia (1998)