ChinaRxiv

AI translation · View original & related papers at
chinarxiv.org/items/chinaxiv-201807.00060

Postprint: Loop-Level Thread Speculative Parallelism Analysis in Embedded Applications

Authors: Bu Deqing, Wang Yaobin, Li Ling, Yang Yang, Cheng Yiming, Liu Zhiqin, Wu Yadong

Date: 2018-07-09T00:00:00+00:00

Abstract

How to effectively utilize the abundant transistor resources provided by multi-core processors to accelerate the execution of sequential programs is a hot topic in current research. Thread-level speculation (TLS) aims to fully utilize multi-core resources and maximally exploit the potential parallelism inherent in sequential code. Currently, TLS technology has been effectively utilized in the parallelization of various sequential applications, but embedded applications have not yet been effectively analyzed in terms of speculative parallelization. Therefore, this paper selects eight representative embedded applications to explore their performance improvement potential and runtime characteristics (data dependencies, thread granularity, and parallel coverage) in loop-level speculative parallelization. Experimental results show that: a) the acceleration effect of parallelizing embedded applications using thread-level speculation is superior to instruction-level parallelism techniques, with the maximum speedup in the experiments reaching 13.29; b) in the embedded application domain, this technology can effectively utilize the computational resources of 4 to 8 cores.

Full Text

Profiling Loop-Level Speculative Parallelism in Embedded Applications

Bu Deqing[1] , Wang Yaobin[1] †, Li Ling[1] , Yang Yang[2], Cheng Yiming[1] , Liu Zhiqin[1] , Wu Yadong[1]

[1](a. School of Computer Science & Technology; b. Sichuan Civil-military Integration Institute, Southwest University of Science & Technology, Mianyang Sichuan 621010, China; [2]Sichuan Institute of Computer Sciences, Chengdu 610041, China)

Abstract: How to effectively utilize the abundant transistor resources provided by multi-core processors to accelerate serial program execution is a hot research topic. Thread-Level Speculation (TLS) aims to fully exploit multi-core resources and maximize the potential parallelism inherent in serial code. While TLS has been effectively applied to parallelize various serial applications, embedded applications have not yet been thoroughly analyzed for speculative parallelism.

This paper therefore selects eight representative embedded applications to investigate their performance improvement potential and runtime characteristics (data dependencies, thread granularity, and parallel coverage) under loop-level speculative parallelization. Experimental results demonstrate that: (a) thread-level speculative parallelization of embedded applications achieves superior speedup compared to instruction-level parallelism, with a maximum speedup of 13.29 observed in our experiments; (b) in the embedded application domain, this technology can effectively utilize computational resources spanning 4 to 8 cores.

Keywords: thread-level speculation; multi-core; embedded application; data dependency

0 Introduction

With the advent of the multi-core era, effectively utilizing the abundant transistor resources on processor cores has become a focal research issue. Since superscalar technology can no longer effectively extract additional parallelism, thread-level parallelism (TLP) techniques have experienced rapid development, particularly for chip multiprocessors (CMP) [1~3]. Currently, most applications are designed for single-core processors and require substantial rewriting to execute on CMP architectures. Thread-Level Speculation (TLS), also known as speculative parallelization (SP), serves as a key methodology for exploiting thread-level parallelism in serial programs by enabling multiple threads to execute speculatively on different cores to enhance program parallelism [4].

Early processor performance evaluation relied on comparing Million Instructions Per Second (MIPS), a metric that proves difficult to standardize across different instruction set architectures. In contrast, the benchmark suite evaluation methodology proposed in the 1980s has gained widespread industry acceptance due to its more comprehensive approach to measuring machine performance. MiBench, a free embedded application benchmark suite developed by the University of Michigan, provides publicly available source code and has been widely adopted in academic research [5]. Consequently, this benchmark suite has received broad recognition in the academic community since its release.

Current evaluation and analysis of the MiBench embedded application benchmark primarily focus on GCC compiler optimization [6], machine learning algorithms [7], and code mutation [8], without thorough analysis from a thread-level

speculation perspective. The objective of this paper is to investigate the performance improvement potential and runtime characteristics of embedded applications under loop-level speculative parallelization. By selecting eight representative programs from the MiBench embedded application suite and the bzip2 program from SPEC CPU, we designed a dynamic profiling mechanism—including core data structures such as hash tables for recording memory addresses—to analyze dependency characteristics, thread granularity, parallelizable coverage, and the impact of core count on speedup during speculative parallelization, thereby identifying a balance between hardware resources and program performance.

1.1 Typical TLS Schemes

Although the Multiscalar [9] architecture did not explicitly propose the TLS concept, its execution process closely resembles current TLS methodologies. The Hydra [10] scheme implemented TLS mechanisms through a hardware-software co-design approach, simplifying hardware system design and advancing TLS development. More recently, Wang et al. [11] proposed a novel TLS model with write-operation-replicated caches, achieving average speedups of 5.69× to 10.04× for general applications. Cao et al. [12] introduced an automatic buffer mechanism selection approach to improve TLS performance, enhancing most programs by over 75%. Our prior work [13] also conducted in-depth analysis of thread-level speculative parallelism in back-propagation neural network applications. In summary, TLS technology has been effectively utilized in parallelizing various types of serial applications.

TLS thread partitioning schemes typically rely on program control flow characteristics, selecting loop structures and subroutine structures as parallelization targets [15]. From a system architecture perspective, programs consist of machine instructions with primary structures including sequential and branch constructs, where loop control forms the core of branch structures and critically impacts program execution performance [16]. Currently, loops serve as the primary parallelization target [17], making them the focus of this study. In our work, loop selection—i.e., determining parallelizable region coverage—employs the GNU Prof tool to collect runtime information from previous executions, selecting loops within subroutines that exhibit large time proportions.

Thread granularity and data dependencies constitute key influencing factors in TLS [18]. Thread granularity refers to the number of dynamic instructions contained within a thread. Excessively fine granularity increases thread count, causing thread management overheads that offset parallelization benefits. Conversely, overly coarse granularity reduces thread count, increasing speculatively cached data and raising dependency conflict probability, thereby decreasing parallelism. Data dependencies refer to value dependencies between child and parent threads. To characterize this, literature [19] proposed the

producer-consumer distance ratio, where $\alpha = A/B$ (producer distance/consumer distance), with A representing the distance from the last write to program start and B representing the distance from the first read to program start. When $\alpha > 1$, parallelism potential is favorable (Figure 2(b)). When $\alpha < 1$, execution resembles serial programs (Figure 2(c)). In this paper, we classify $\alpha < 1$ as fatal dependencies, $1 < \alpha < 2$ as hazardous dependencies, and $\alpha > 2$ as safe dependencies.

1.2 Related Optimizations for Embedded Applications

Current acceleration evaluations for MiBench primarily include: an optimized GCC compiler architecture [6] that improves MiBench speedup through compiler optimization selection algorithms and advanced hybrid elimination algorithms; machine learning algorithms [7] that reduce MiBench' s time and space complexity to improve speedup; and a novel code mutation technique [8] that transforms applications into complex reverse engineering problems, altering dynamic storage and control flow behavior to change performance characteristics. However, MiBench has not yet been effectively analyzed from a thread-level speculation perspective.

---

2 Speculation Model and Profiling Mechanism

During the initial chip design phase, hardware simulation serves as the most effective technique for comprehensive performance evaluation based on specific hardware system models, enabling analysis and trade-offs of key performance factors. Compared to direct hardware development, simulation not only validates hardware design effectiveness with high precision but also enables low-cost comprehensive comparison and adjustment of various design alternatives. Therefore, this paper designs the following speculation model and profiling mechanism to analyze speculative parallelism in serial code.

2.1 Speculation Model

TLS technology can partition a single program thread into multiple automatically parallelizable threads. The TLS execution process proceeds as follows: the first thread executes sequentially, subsequent threads execute speculatively, and commits occur according to the original serial program order. If conflicts arise during parallel execution, conflicting threads are rolled back and re-executed [14]. As shown in Figure 1 [Figure 1: see original paper], Figure 1(a) illustrates traditional sequential thread execution on a core—the current execution model for serial programs. Figure 1(b) demonstrates how the system decomposes the program into multiple threads at runtime and distributes them across cores. When execution begins, the first core notifies other cores to load programs and

variables, then dispatches start signals. Upon receiving the signal, other cores commence speculative execution. After the first core completes its commit, the second core continues committing computed values to maintain sequential semantics. When the final thread completes execution and commits its last value, it signals program termination to other cores.

TLS thread partitioning schemes typically select program loop and subroutine structures based on control flow characteristics [15]. From a system architecture perspective, programs consist of machine instructions with primary structures including sequential and branch constructs, where loop control forms the core of branch structures and critically impacts program execution performance [16]. Currently, loops serve as the primary parallelization target [17], making them the focus of this study. In our work, loop selection—i.e., determining parallelizable region coverage—employs the GNU Prof tool to collect runtime information from previous executions, selecting loops within subroutines that exhibit large time proportions.

Thread granularity and data dependencies constitute key influencing factors in TLS [18]. Thread granularity refers to the number of dynamic instructions contained within a thread. Excessively fine granularity increases thread count, causing thread management overheads that offset parallelization benefits. Conversely, overly coarse granularity reduces thread count, increasing speculatively cached data and raising dependency conflict probability, thereby decreasing parallelism. Data dependencies refer to value dependencies between child and parent threads. To characterize this, literature [19] proposed the producer-consumer distance ratio, where $\alpha = A/B$ (producer distance/consumer distance), with A representing the distance from the last write to program start and B representing the distance from the first read to program start. When $\alpha > 1$, parallelism potential is favorable (Figure 2(b)). When $\alpha < 1$, execution resembles serial programs (Figure 2(c)). In this paper, we classify $\alpha < 1$ as fatal dependencies, $1 < \alpha < 2$ as hazardous dependencies, and $\alpha > 2$ as safe dependencies.

2.2 Profiling Mechanism

Our loop profiling tool follows this design methodology: (a) identify "hotspot" code fragments with substantial computation and high parallelization potential as preliminary candidates; (b) profile these hotspot regions to quantitatively analyze key performance factors (data dependencies, thread granularity, and parallel coverage); (c) output quantitative analysis results.

As shown in Figure 3 [Figure 3: see original paper], we first employ Linux's built-in GNU Prof tool for preliminary program analysis. This tool roughly analyzes each subroutine' s runtime proportion and function call relationships during execution. Based on this information, we select hotspot fragments exceeding 3% time proportion as preliminary speculative candidate regions. We then insert profiling markers into loops within these candidates, cross-compile them into

executable binary files using a compiler, and finally enable the profiling tool to automatically identify hotspot regions in the binary through assembly instruction recognition for profiling and quantitative analysis based on key performance factors.

As illustrated in Figure 4 [Figure 4: see original paper], for automatic marker identification, we use the objdump4pisa disassembly tool supporting the PISA instruction set to disassemble binary code into assembly. The loop_{id} is compiled and stored in register 4, while the loop variable address is compiled and stored in register 5. The profiling tool then identifies hotspot regions by recognizing jal instructions and loop markers in the assembly file.

Based on this profiling flow, Figure 5 [Figure 5: see original paper] depicts the core data structure of our loop profiling mechanism. For programs containing multiple loops, each loop is decomposed into n iteration_{{{list}}*{{entry}}}t structures, with each iteration{{{list}}*{{entry}}}t further decomposed into N iteration_t structures representing iterations. Hash{list} records memory write operations, and producer-consumer distances can be calculated from iteration_{{{list}}*{{head}}} and iteration{{{list}}*{{tail}}} in Hash{list}. Each iteration_t contains a unique loop id from loop unrolling, start time, pointers to next and previous loop unrolling addresses, hash_{next} recording the next loop unrolling' s memory operation address pointer, and hash_{pre} recording the previous loop unrolling' s memory operation address pointer. When subsequent code in the profiling portion reads a memory location, the profiling tool intercepts it and retrieves the last write time for that memory unit by searching this structure. If the current time is less than the last write time, the current time is set as the last write time for that memory address. The system also maintains a serial version runtime, and the ratio of these two times yields the desired speedup.

---

3 Experimental Data and Analysis

This section describes our experimental environment (Table 1) and presents detailed analysis of selected applications.

Table 1 Experimental Environment Description

Environment Component	Configuration
Development Platform	Linux-based Ubuntu 12.04
Simulation Framework	Simplescalar toolset
Compiler	Modified and extended gcc-2.7.3.3 from Simplescalar
Simulator	Modified and extended sim-fast from Simplescalar

We selected sim-fast from the Simplescalar toolset for modification because this

simulator executes one instruction per cycle, making profiling time well-suited for compiler overhead requirements.

MiBench contains 35 benchmark programs; we selected eight typical applications for analysis and also included SPEC CPU's Bzip2 for comparison with literature [6]. Table 2 lists the selected applications.

Table 2 Selected Applications and Descriptions

Application	Category	Description
Dijkstra	Network	Dijkstra algorithm implementation
Bzip2	Consumer	Lossless compression algorithm implementation
Bitcount	Automotive	Counts number of 1s in integer arrays
Susan	Automotive	Image recognition toolkit
Patricia	Network	Patricia Trie for sparse leaf nodes
Blowfish	Security	Blowfish encryption/decryption algorithm

3.1 Time Comparison

Table 3 compares our best runtime (mm) with the best runtime from literature [6], where A represents the program' s runtime without acceleration, B represents the best accelerated runtime from literature [6], and C represents our best runtime.

Table 3 Runtime Comparison

Application	A (Baseline)	B (Literature [6])	C (Our Work)
Bzip2	1200	850	780
Bitcount	980	720	650
Susan	1500	1100	950
Dijkstra	2100	1400	1200
Patricia	890	670	680
Blowfish	1100	800	720

Except for Patricia (where our runtime is slightly higher than literature [6]), all other programs achieve better runtime than the best results reported in literature [6], demonstrating that our thread-level speculation technique outperforms optimized GCC compiler results.

3.2.1 Dijkstra

Figure 6 [Figure 6: see original paper] shows the speedup from profiling Dijkstra. As core count increases, the program exhibits linear speedup scaling, plateauing between 13.10× and 13.29× beyond 64 cores. Source code analysis reveals no data dependencies between loops and high parallelizable region coverage with

predominantly safe dependencies, enabling continuous speedup improvement with increasing core count until saturating at 64 cores.

3.2.2 Bzip2

Profiling Bzip2 yields the speedup shown in Figure 7 [Figure 7: see original paper]. Thread-level parallel analysis reveals speedup improvements of 0.39× and 0.48× on 4-core and 8-core configurations, respectively. Further hardware resource increases show diminishing returns compared to 4-core and 8-core configurations, indicating that speedup becomes relatively stable despite continued slow improvement, ultimately wasting hardware resources.

3.2.3 Bitcount

As shown in Figure 8 [Figure 8: see original paper], Bitcount exhibits linear speedup growth as core count increases from 2 to 8 cores, after which speedup ceases to improve. This demonstrates that rational core utilization is a critical consideration.

3.2.4 Susan

Figure 9 [Figure 9: see original paper] reveals a slow linear speedup trend as core count increases from 2 to 32 cores, with 60% and 80% speedup improvements at 4-core and 8-core configurations, respectively. Source code analysis shows the program primarily performs image processing with basic logical operations. Consequently, limited loop count results in low parallelizable region coverage, restricting overall speedup potential.

3.2.5 Jpeg

As illustrated in Figure 10 [Figure 10: see original paper], both encode and decode programs show limited speedup before 16 cores, with only the encode program showing improvement beyond 16 cores. As compression and decompression applications involving extensive pointer-based data, these programs suffer from severe data dependencies that hinder ideal speedup even with increasing core counts.

3.2.6 Patricia

Patricia' s speedup profile in Figure 11 [Figure 11: see original paper] indicates it is unsuitable for speculative multithreading. Source code analysis reveals pointer-heavy implementations causing severe data dependencies between loops and subsequent code, resulting in stable but poor speedup.

3.2.7 Blowfish

Figure 12 [Figure 12: see original paper] shows Blowfish encode and decode speedups remain relatively stable before 8 cores, only beginning to improve

afterward and achieving merely 1× speedup at 64 cores. Since increasing core count incurs unlimited cost, finding a balance between core count and speedup is crucial. The encode and decode programs exhibit consistent thread granularity, parallel coverage, and data dependency characteristics, yielding similar results.

3.2.8 Data Analysis

Given that each program involves different influencing factors and yields distinct speedups, we normalized experimental results by comparing each speedup against the 64-core speedup, as shown in Figure 13 [Figure 13: see original paper].

Figure 13 reveals that except for Dijkstra (which is unsuitable for thread-level parallel acceleration), all other programs exhibit diminishing speedup growth rates as core count increases. Bitcount represents an extreme case where speedup growth stabilizes beyond 4 cores. Most programs reach a performance-hardware resource balance point between 4-core and 8-core configurations.

3.3 Analysis of Influencing Factors

Since thread granularity, parallelization coverage, and dependency characteristics all influence thread-level speculation, we analyzed these factors for the selected applications.

Thread granularity distribution appears in Figure 14 [Figure 14: see original paper]. Most programs exhibit thread granularity between 10^6 and 10^8 instructions. As embedded application speculative thread granularity is typically larger than general-purpose applications, speculative cache capacity in embedded-oriented multicore processor designs should be increased by one order of magnitude to avoid cache overflow.

Parallelizable coverage is shown in Figure 15 [Figure 15: see original paper]. Most programs achieve coverage between 30% and 50%. According to Amdahl' s law, higher parallelizable coverage enables more effective thread-level speculation development. Consequently, low parallel region coverage contributes to limited speedup.

Dependency characteristic α distribution appears in Figure 16 [Figure 16: see original paper]. Most programs exhibit fatal dependencies exceeding 50%, with three programs reaching approximately 90%. Higher α values ($\alpha > 1$) indicating safe dependencies facilitate thread-level speculation development. Thus, severe fatal dependencies represent the primary reason for limited speedup in most embedded applications.

Combined source code and performance factor analysis reveals three main causes for limited parallel speedup in embedded applications: First, large loop iteration thread granularity increases dependency conflict probability between threads—the primary cause of limited speedup in our selected applications. Therefore,

programmers should decompose large loop bodies to reduce thread granularity. Second, most programs exhibit low parallelizable region coverage with severe fatal dependencies. To improve loop parallelization potential and reduce fatal dependencies, programmers should minimize pointer usage in loop bodies or replace pointer data with alternatives, thereby achieving greater benefits from loop-level thread parallelism and improving parallelization potential.

---

4 Conclusion

Through comparative analysis of speedups obtained from thread-level speculative parallelization of most MiBench programs and SPEC' s Bzip2, we draw the following conclusions:

a) Thread-level speculation acceleration for embedded applications outperforms optimized GCC compiler results. Therefore, in the multi-core era, thread-level speculation technology provides effective acceleration for embedded applications.

b) In MiBench, when core count increases to between 4 and 8 cores, most programs achieve speedups between 1.65× and 4.89×, representing performance improvements of 63.8% to 84.5%. Thus, when designing multicore processors for embedded applications, a core count between 4 and 8 effectively improves hardware utilization while achieving optimal acceleration.

c) Data dependency is the primary factor affecting loop-level thread speculation. In loop iterations, appropriately modifying loop body or control variables can eliminate inter-iteration data or control dependencies, or increase the α ratio (consumer distance/producer distance), thereby improving loop parallelization potential and achieving higher speedup.

As hardware simulation represents the first stage of chip development—requiring successful 准入式评估 (admission evaluation) before subsequent more accurate hardware prototype verification and real hardware system evaluation—our proposed approach focuses on deriving theoretical parallel speedup limits. Compared to actual hardware system execution (particularly processor memory system behavior), certain differences remain. Future work will conduct register-level hardware prototype simulation for these embedded applications to obtain more accurate evaluation data. Additionally, we will further profile and analyze the thread-level speculative parallelization potential of this benchmark suite when speculating on subroutine structures, comparing it with loop-level speculative parallelization potential to provide design guidance for embedded speculative multicore chip development.

References

[1] Luo Yangchun, Zhai A. Dynamically dispatching speculative threads to improve sequential execution [J]. *ACM Trans on Architecture and Code Optimization*, 2012, 9(3): 1-31.

[2] Schmidt T, Liu Guantao. Exploiting thread and data level parallelism for ultimate parallel systemc simulation [C]// *Proc of the 54th Annual Design Automation Conference*. New York: ACM Press, 2017: article No. 79.

[3] Cai Yong, Li Guangyao, Liu Wenyang. Parallelized implementation of an explicit finite element method in many integrated core (MIC) architecture [J]. *Advances in Engineering Software*, 2018, 116: 50-59.

[4] Salamanca J, Amaral J N, Araujo G. Using hardware-transactional-memory support to implement thread-level speculation [J]. *IEEE Trans on Parallel & Distributed Systems*, 2018, PP(99): 1-14.

[5] Guthaus M R, Ringenberg J S, Ernst D, et al. MiBench: a free, commercially representative embedded benchmark suite [C]// *Proc of IEEE WWC*. Austin, Texas: IEEE Press, 2001: 3-14.

[6] Andrews J. Analysis of Mibench Benchmark Applications Using GCC Compiler [C]// *Proc of the 3rd International Conference on Computer Science and Information Technology*. 2013: 34-37.

[7] Andrews J, Sasikala T. Evaluation of various compiler optimization techniques related to mibench benchmark applications [J]. *Journal of Computer Science*, 2013, 9(6): 749-756.

[8] Ertvelde L V, Eeckhout L. Dispersing proprietary applications as benchmarks through code mutation [C]// *Proc of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*. New York: ACM Press, 2008: 201-210.

[9] Vijaykumar T N, Sohi G S. Task Selection for a multiscalar processor [C]// *Proc of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*. New York: ACM Press, 1998: 81-92.

[10] Hammond L, Hubbert B A, Siu M, et al. The Stanford Hydra CMP [J]. *IEEE Micro*, 2000, 20(2): 71-84.

[11] Wang Qiong, Wang Jialong, Shen Li, et al. A software-hardware co-designed methodology for efficient thread level speculation [C]// *Proc of the 17th IEEE International Conference on Computer and Information Technology*. New York: ACM Press, 2017: 184-191.

[12] Cao Zhen, Verbrugge C. Reducing memory buffering overhead in software thread-level speculation [C]// *Proc of the 25th International Conference on Compiler Construction*. New York: ACM Press, 2016: 12-22.

[13] Wang Yaobin, An Hong, Liu Zhiqin, et al. Parallelizing Back Propagation Neural Network on Speculative Multicores [C]// *Proc of the 22nd International Conference on Parallel and Distributed Systems*. Piscataway, NJ: IEEE Press, 2017: 902-907.

[14] Li Xiang. Research on the key technology for parallel serial program based on multi-core [D]. Xi' an: Xi' an University of Technology, 2014.

[15] Goossens B, Parello D, Porada K, et al. Computing on many cores [J]. *Concurrency and Computation Practice and Experience*, 2017, 29(4): 1-20.

[16] Zou Yu, Xue Xiaoping, Zhang Fang, et al. Error detection algorithm of program loop control [J]. *Journal of Computer Applications*, 2015, 35(12): 3450-3455.

[17] Li Yingying, Pang Jianmin, Li Yanbing, et al. Multi-dimensional parallelism recognition method of nested loop for many-core processors [J/OL]. *Application Research of Computers*, 2018, 35(11). (2017-11-10) [2018-03-10]. http://www.arocmag.com/article/02-2018-11-001.html.

[18] Liu Bin, Zhao Yinliang, Han Bo, et al. A loop selection approach based on performance prediction for speculative multithreading [J]. *Journal of Electronics and Information Technology*, 2014, 36(11): 2768-2774.

[19] Liang Bo. Study on the key technologies of thread-level speculation on multi-core platform [D]. Hefei: University of Science and Technology of China, 2008.

Note: Figure translations are in progress. See original paper for figures.

Source: ChinaXiv —Machine translation. Verify with original.