EFFICIENT SPECULATIVE PARALLELIZATION ARCHITECTURE FOR OVERCOMING SPECULATION OVERHEADS

SudhakarKumar[1], Sunil K.Singh[2], NaveenAggarwal[3] and KritiAggarwal[4]

[1]*Chandigarh College of Engineering and Technology, Chandigarh, India*
[2]*Chandigarh College of Engineering and Technology, Chandigarh, India*
[3]*University Institute of Engineering and Technology, Chandigarh, India*
[4]*Chandigarh College of Engineering and Technology, Chandigarh, India*

Abstract

Today, with the advancement in technology, development of efficient smart and autonomous systems have become a priority. These systems use complex calculations which are hardware demanding and time consuming. Parallel processing systems hence, provide a way of improving the computational performance as a whole. Parallelizing sequential code automatically entails converting it to multithreaded code without supervision. Parallelization in the ideal case would allow programmers to make full use of available hardware resources, resulting in optimal performance. This isn't possible due to a wide array of program dependencies. The technique of speculative parallelization is one of the most favorable ways to automatically parallelize a loop when the system cannot determine dependencies at compile time. Therefore, the need for manual parallelization is eliminated. However, due to the use of additional hardware or software architectures, there are some speculative overheads. The present paper describes a suitable speculative parallelization algorithm that is capable of providing optimal performance. Moreover, suitable hardware architecture is also suggested which can further reduce the overheads for speculative parallelism.

Keywords

Automatic Parallelization, Speculative Parallelization, Speculation Overheads, Transaction Memory

1. Introduction

Automatic parallelization, often known as auto parallelization, is the process of transforming sequential code into multi-threaded code in order to employ many processors concurrently in a multi-core shared-memory chip architecture [1]. Some of the most popular use of automatic parallelization architectures include artificial intelligence, machine learning, seismic surveying, computational astrophysics, video color correction, climate modeling, financial risk management, drug discovery, medical imaging and computational fluid dynamics [2-6]. With the advent of smart technologies especially in the field of architectures [26-28] and data science, including artificial intelligence and deep learning, automatic parallelization techniques are being extensively used to improve the computation speed of large datasets [7-9]. However, despite its popularity, parallelizing sequential programs fully automatically is challenging because it requires complex analysis, and the ideal solution could depend on unknown parameter values when compiling [10].

Speculative parallelization [11] is a strategy which automatically parallelizing loops when the system cannot detect dependencies at build time. This strategy, also known as thread-level speculation, assumes that the system can execute all iterations of a particular loop in parallel. Speculative

International Conference on Smart Systems and Advanced Computing (Syscom-2021), December 25–26, 2021
EMAIL: sudhakark324@gmail.com (A. 1); sksingh@ccet.ac.in (A. 2); , navgg@gmail.com (A. 3) ; kritiaggarwwal@gmail.com (A.4)
ORCID: 0000-0001-7928-4234 (A. 1); 0000-0003-4876-7190 (A. 2); 0000-0003-1549-531X (A. 3) ; 0000-0002-7415-0103 (A. 4)


parallelization has the advantage that it can automatically parallelize loops in a sequential program even if dependency patterns are unknown at build time.

In this way, it can speed up a parallel, multithreaded computer without requiring manual parallelization, which incurs development costs. This takes only a few basic changes to the original sequential code, which are all well within the capability of contemporary compilers. Implementing custom functions for scheduling threads, executing speculative loads, and initiating commits when a thread completes successfully.

However, there are some overheads as a result of the utilization of additional hardware. Overheads [12] are all distinct extra activities that must be performed with the goal of resolving difficulties caused by mis-speculation in speculative parallelism. Thread start and commit overhead is one sort of mandatory overhead. Other forms of overhead include roll-back overhead, squash overhead, load imbalance overhead, hardware overhead, communication overhead, and cache replacement overhead, which are non-compulsory[13].

Despite the overheads caused by mis-speculation, automated speculative parallelism with multicore chip design requires less time to execute a speculative loop than a sequential loop on a single processor. As a result, the current research investigates several automated parallelization strategies for optimum sequential code translation. As a result, the authors devise a suitable algorithm capable of producing near-optimal outputs for dependency-driven applications. The authors also proposed transactional memory-based technology for the same purpose. The primary goal of creating such hardware is to minimize overhead as much as feasible.

2. Literature Survey

This section surveys various automatic parallelization algorithms and compares their advantages and disadvantages. The paper also surveys various software and hardware overheads that are encountered while realizing the speculative parallelism algorithms.

2.1. Automatic Parallelization Algorithms

There have been several ways developed for parallelizing sequentially coded algorithms [14]. Most the algorithms utilize the relationship between loop iterations. This is because loops take most of the program execution time. The parallelizing compilers only execute these iterations in separate threads if they have removable data dependency. When any of the iterations is reliant on other iterations (despite independent iterations), no compiler can parallelize the loop.

The publications in [15] provide an overview of some of the most significant contributions in the field of automatic parallelization. Popular techniques like DOALL (Figure 1) and DOACROSS (Figure 1) [16] techniques have been employed to better use multicore architectures through thread level parallelism (TLP). A loop iteration in DOALL can be executed in parallel because it is independent from any other loop iteration. On the other hand, DOACROSS is capable of extracting parallelism from more complicated loops with loop-borne dependencies. Because the loop carried dependency must be communicated between cores every time the algorithm is iterated, DOACROSS performance is a function of inter-core latency of the system. Figure 1 shows doall and doacross execution in comparison to the sequential execution.

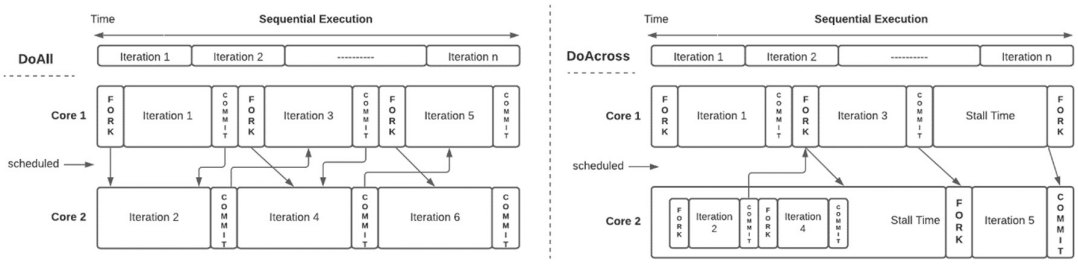

Figure 1: Doall and Doacross Loop Execution

However, applying these algorithms is a difficult and error prone task due to inter-thread data and control dependencies. Also, compilers sometimes act very conservative whenever they can't assure the absence of any type of dependence. Moreover, as per the Amdahl's law [17], only few fractions of whole sequential workload are parallel and hence only those parallel parts can be allocated to different cores not whole program.

Speculative parallelism or thread level speculation is an optimistic multi-threading technique that automatically parallelizes the implicit (speculative or dependent) threads keeping in view of original sequential semantic of the program. Parallelization using thread-level speculation (TLS) has been studied using software [20, 21] and hardware [18, 19] architecture. Rauchwerger et al [22] originally proposed it as a way to parallelize loops with independent data access, primarily arrays. There is a common feature among TLS implementations in that they largely rely on loops to parallelize, they mostly change the cache coherence protocols, and the parallel sections are generally small.

2.2. Overhead encounter in Speculative Parallelization

The identification of speculative threads can greatly contribute to overall performance. Architectures like Hydra, Atlas, Trace, Mitosis etc use dedicated compilers for the parallelization of sequential code [23, 24]. For the control flow and/or speculative values, however, they use software-based value prediction. This as compared to the proposed method is not applicable for all software code and applications.

Several thread-level speculative approaches have been proposed for extracting parallelism from legacy code by using Transactional Memory. They split an application into sections and run them speculatively on parallel threads. Each thread can either buffer its state or expose it. If the code is unsafe, the changes are reverted, and the execution is restarted. Some efforts have combined TLS and TM into a unified model to benefit from both technologies. However, most of the models require manual efforts to ensure memory violations do not occur.

The proposed architecture and algorithm uses speculation for improving the performance of automatic parallelization algorithm. However, as discussed in section 2.1. and section 2.2. Speculation requires additional hardware or software support which results in overheads. The present paper hence proposes an efficient, computation aware architecture that minimizes some of the non-compulsory speculation overheads.

3. Architecture Framework & Design

The present paper proposes an architecture for conversion of sequential code into paralleled threads or jobs speculatively. For the same program is first converted into LLVM IR through the preprocessing step. Then via the code profiling step hot loops are identified. These loops then pass through the code transformer and generator phase of the architecture. The next step is the multi-threaded code generation step as described in the section 3.1. The partitioned code is scheduled onto the transaction memory-based runtime architecture. The scheduling of jobs is said to be successful when there are no dependency violations. The detailed architecture scan be seen in the Figure 2.

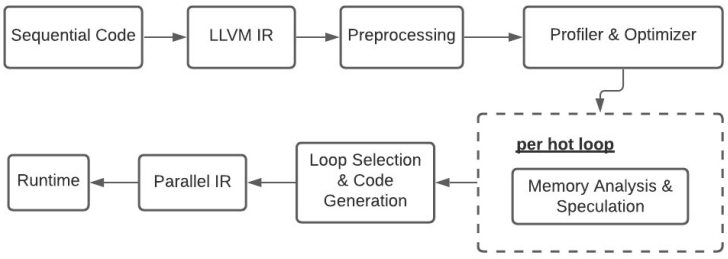

Figure 2: Architectural Framework

1. **Preprocessing**: It involves preparing the program's intermediate representation (IR) for parallelization.
2. **Profiling:** The code profiling employed in this design is based on hotspots in the original IR code.
3. **Memory analysis & Code Generation**: The result of the memory analysis phase is fended into the code transformation and generation mechanism. The code generation algorithm used in the architecture is defined in Figure 3.

Algorithm (Hot Loop ℓ, Thread t)

1. Compute inter- iteration [live values]
2. Speculated values are calculated using value predicter.
3. if memory dependency then
   - 3.1 return, no transformation possible
4. else create thread copies using thread portioning
5. For each thread t_i
   - 5.1 Build Dependency Graph G
   - 5.2 SCCS identification for Graph G
   - 5.3 If pipeline possible,
6. L then apply cyclic pipelining
7. Generate communication code
8. Apply mis- speculation mechanism

Figure 3: Suitable Code Generation Algorithm

4. **Runtime & Hardware Architecture**: The model uses a batch wise process batch approach. The multi-threaded code generation algorithm produces parallel code in the form worker threads (or jobs) which are executed in batches. These are then implemented as memory transactions in the model. Each memory transaction has its own private copy of the memory that has been accessed. The purpose of transactional memory systems is to support portions of code identified as transactions in a visible manner by guaranteeing atomicity, consistency, and isolation. By allowing programmers to wrap their procedures behind transactional blocks, transactional memory provides a high-level programming abstraction.
5. **Validation & Recovery:** To support transactional memory access, the model additionally includes batch-wise validation tests In the event that a dependence violation conflict is found, tasks within the batch are performed up to the last valid checkpoint before waiting for the Recovery program to terminate transactions in the higher chronological order

4. Evaluation

Implementations of the current architecture use the LLVM compiler infrastructure, which has been tested using clang in full system mode on a quad-core out-of-order processor. The operating system

used in for testing and implementing the model is Linux OS (Ubuntu Version 21) [25]. The algorithm phase is implemented in the LLVM compiler infrastructure. LLVM provide analysis modules and passes which can be manipulated in order to perform code optimizations and transformations.

Speedup of the hottest loop of each benchmark speedup is compared to the sequential execution time in Figure 4.

052.alvinn is a back propagation based neural networking algorithm in C from the SPECfp92 benchmark suite. K-means or 120.kmean is a clustering algorithm in C++ form the SPEC ACCEL benchmark suite. Labyrinth algorithm from STAMP benchmark is used in grid copying operations for the removal of read sets. Swaptions is PARSEC based algorithm used in heath-jarrow-Morton framework. Bzip algorithm is an opensource burrows-Wheeler Feature comparison algorithm.

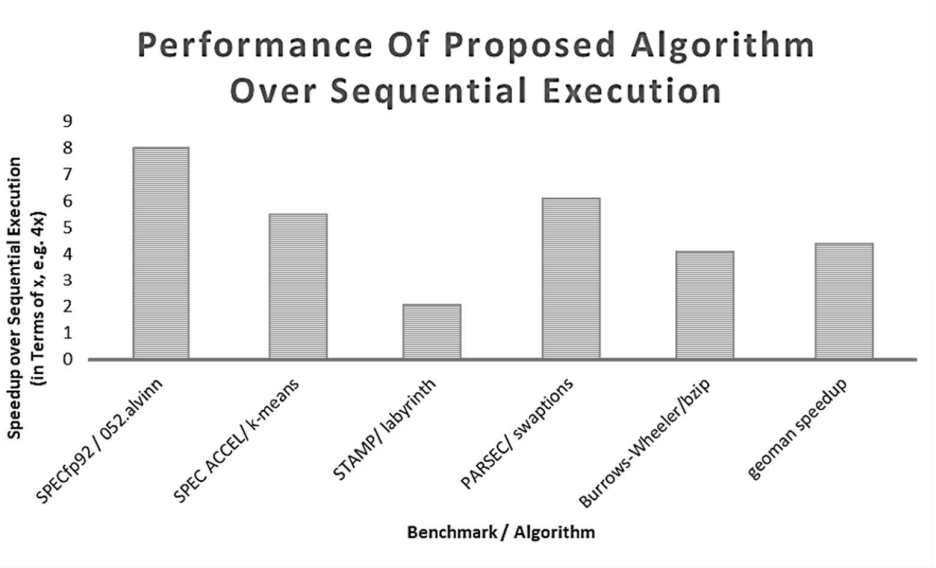

Figure 4 : Speedup of designed algorithm over sequential code

Figure 4, represents the geometric speedup of the benchmark codes on the proposed model. The proposed model gives on overall geomean speedup of 4.3x on the tested benchmark algorithms. Hence, applying speculative parallelization algorithm is much more efficient as compared to the sequential code execution.

5. Conclusion

The current research paper discussed automatic parallelization approaches for the conversion of parallel codes into threads. Although several parallelization algorithms have been devised over the year, speculative parallelization techniques provide better speedup. This is because they speculatively predict values at compile time. This helps in code optimization and hence, code generation. However, speculative algorithms require additional overheads. Hence, in the present paper a suitable speculative algorithm has been devised which produces high speed up as compared to sequential algorithms. Further, transactional memory-based hardware architecture for the same has also been discussed. The hardware suggested, is capable of producing near-optimal outputs for dependency-driven applications. The future research avenues entails improving the parts of speculative parallelization architecture, in order to ensure better dependency identification and removal. The use of LLVM in development infrastructure as a whole will not only help in increasing the performance of proposed algorithms but also facilitate the application of algorithm to various other speculation techniques as well.

6. References

[1] Yehezkael, Rafael (2000). "Experiments in Separating Computational Algorithm from Program Distribution and Communication". Lecture Notes in Computer Science of Springer Verlag. Lecture Notes in Computer Science. 1947: 268–278.

[2] Aggarwal, K., Singh, S. K., Chopra, M., & Kumar, S. (2022). Role of Social Media in the COVID-19 Pandemic: A Literature Review. In B. Gupta, D. Peraković, A. Abd El-Latif, & D. Gupta (Ed.), Data Mining Approaches for Big Data and Sentiment Analysis in Social Media (pp. 91-115). IGI Global. http://doi:10.4018/978-1-7998-8413-2.ch004.

[3] Chopra, M., Singh, S. K., Aggarwal, K., & Gupta, A. (2022). Predicting Catastrophic Events Using Machine Learning Models for Natural Language Processing. In B. Gupta, D. Peraković, A. Abd El-Latif, & D. Gupta (Ed.), Data Mining Approaches for Big Data and Sentiment Analysis in Social Media (pp. 223-243). IGI Global. https://www.igi-global.com/gateway/chapter/293158.

[4] Do, P., Phan, T. H., et al. (2021). Developing a Vietnamese tourism question answering system using knowledge graph and deep learning. Transactions on Asian and Low-Resource Language Information Processing, 20(5), 1-18.

[5] Gupta, B. B., Agrawal, D. P., Yamaguchi, S., & Sheng, M. (2020). Soft computing techniques for big data and cloud computing. Soft Computing, 24(8), 5483-5484.

[6] Gupta, B. B., Tewari, A., Cvitić, I., Peraković, D., & Chang, X. (2021). Artificial intelligence empowered emails classifier for Internet of Things based systems in industry 4.0. Wireless Networks, 1-11.

[7] AlZu'bi, S., Hawashin, B., Mujahed, M., Jararweh, Y., et al. (2019). An efficient employment of internet of multimedia things in smart and future agriculture. Multimedia Tools and Applications, 78(20), 29581-29605.

[8] Adil, K., Jiang, F., Liu, S., Grigoriev, A., et al (2017). Training an agent for fps doom game using visual reinforcement learning and vizdoom. International Journal of Advanced Computer Science and Applications, 8(12).

[9] Manepalli Ratna Sri, Surya Prakash and T Karuna (2021) Classification of Fungi Microscopic Images – Leveraging the use of AI, Insights2Techinfo, pp.1

[10] Fox, Geoffrey; Roy Williams; Paul Messina (1994). Parallel Computing Works! Morgan Kaufmann. pp. 575, 593. ISBN 978-1-55860-253-3

[11] M. K. Prabhu and K. Olukotun, "Using Thread-Level Speculation to Simplify Manual Parallelization," in Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 2003, pp. 1–12.

[12] M. B. Radulović, M. V Tomašević, and V. M. Milutinović, "Chapter One - Register-Level Communication in Speculative Chip Multiprocessors," vol. 92, A. Hurson, Ed. Elsevier, 2014, pp. 1–66.

[13] Zima, H., Supercompilers for Parallel and Vector Computers, ACM Press, New York, 1990.

[14] Doreen Y Cheng. 1993. A survey of parallel programming languages and tools. Computer Sciences Corporation, NASA Ames Research Center, Report RND-93-005 March (1993)

[15] Sudhakar Kumar, Sunil Kr Singh, Naveen Aggarwal, Kriti Aggarwal, "Evaluation of automatic parallelization algorithms to minimize speculative parallelism overheads: An experiment", pp 1517-1528, 2021 Journal of Discrete Mathematical Sciences and Cryptography, Volume 24, Issue 5 , Taylor & Francis, (2021)

[16] M. D. Hill and M. R. Marty, "Amdahl's Law in the Multicore Era," Computer (Long. Beach. Calif)., vol. 41, no. 7, pp. 33–38, Jul. 2008

[17] Michael K Chen and Kunle Olukotun. 2003. The Jrpm system for dynamically parallelizing Java programs. In Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on. IEEE, 434–445.

[18] Lance Hammond, Mark Willey, and Kunle Olukotun. 1998. Data Speculation Support for a Chip Multiprocessor. SIGOPS Oper. Syst. Rev. 32, 5 (Oct. 1998), 58–69.

[19] J Greggory Steffan, Christopher B Colohan, Antonia Zhai, and Todd C Mowry. 2000. A scalable approach to thread-level speculation. Vol. 28. ACM.

[20] Francis Dang, Hao Yu, and Lawrence Rauchwerger. 2001. The R-LRPD test: Speculative parallelization of partially parallel loops. In Parallel and Distributed Processing Symposium., Proceedings International, IPDPS 2002.

[21] Wei Liu, James Tuck, Luis Ceze, Wonsun Ahn, Karin Strauss, Jose Renau, and Josep Torrellas. 2006. POSH: a TLS compiler that exploits program structure. In Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming. ACM, 158–167.

[22] Lawrence Rauchwerger and David A Padua. 1999. The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization. Parallel and Distributed Systems, IEEE Transactions on 10, 2 (1999), 160–180.

[23] M.B. Radulovic, M.V. Tomaševic, On reducing overheads in CMP TLS integrated protocols, IPSI Trans. Internet Res. 3 (1) (2007) 11–17.

[24] M.B. Radulovic, M.V. Tomaševic, Towards an improved integrated coherence and speculation protocol, in: IEEE EUROCON 2007 Conference, Sept9-12, Warsaw, Poland, 2007, pp. 405–412.

[25] Singh, S.K. (2021). Linux Yourself: Concept and Programming (1st ed.). Chapman and Hall/CRC. https://doi.org/10.1201/9780429446047

[26] Singh, S. K., Singh, R. K., & Bhatia, M. P. S. (2011). CAD Optimization Technique in Reconfigurable Computing System using Hybrid Architecture. *International Journal of Computer Applications*, 24(4), 50-54.

[27] Singh, S. K., Singh, R. K., Bhatia, M. P. S., & Madan, R. (2011). Multi FPGA Based Novel Reconfigurable Hybrid Architecture for High Performance Computing. *International Journal of Computer Science Issues (IJCSI)*, *8*(4), 335.

[28] S. K. Singh, A. Madaan, A. Aggarwal and A. Dewan, "Design and implementation of a high-performance computing system using distributed compilation," 2013 International Conference on Advances in Computing, Communications and Informatics (ICACCI), 2013, pp. 1352-1357, doi: 10.1109/ICACCI.2013.6637374.