Multi-thread Weak Deterministic Method Based on Cache Optimization

Kaiyu Wang
Harbin institute technology
Computer science and technology department
Harbin China
Wang_kaiyu2103@126.com

Zhenzhou Ji
Harbin institute technology
Computer science and technology department
Harbin China
jizhenzhou@hit.edu.cn

***Abstract*—Aiming at the problem of multi-thread uncertainty and the large system overhead of the existing deterministic implementation, this paper proposed to enter from the cache level. Research on weak deterministic methods for row optimization. First, for the problem that the LRU replacement algorithm does not work as well for multithreading as LIRS. The replacement strategy is optimized, followed by the weak deterministic method of ignoring data competition, and the deterministic order is obtained for the thread synchronization point. The competition that occurs with the card's synchronous operation is improved using a deterministic approach. We use of isolated thread memory in data competition, to change the thread into a lightweight process, and use experiments to prove the feasibility of the method**.

Keywords-component, Cache replacement, Multithreading, LIRS, Weak Determinism

I. Introduction

With the rapid development of microelectronics technology, multi-core processors have become the mainstream computing platform and research hotspot. Compared with previous single-core processors, multi-core processors have exploded in hardware performance, but traditional serial programs have cannot play its performance, parallel programming is the key to fully exploit its multi-core performance. It is enable to mainstream applications from multi-core CPU. The only way to program the benefits of performance. In order to achieve the goal of software parallelism, it is necessary to provide a hardware platform for parallel programs. The core architecture can be divided into shared memory and distributed memory based, Pthread for shared memory and The MPI of distributed memory is the two most common standards[1]. In general, the current parallel deterministic technology is still not mature enough, there are still many problems, the certainty of pure software implementation Parallel systems are generally inefficient and can achieve 2 to 10 times performance overhead[12]. This makes the technology difficult to be used in practical applications accept. Therefore, Olszewski and other scholars from another perspective, assuming that there is no data competition in the program, by sacrificing one Partial determinism in exchange for performance improvement which the concept of weak determinism is proposed. That is the system only guarantees the execution of program synchronization statements make the order is deterministic.

II. Multi-thread Weak Deterministic Method

A. Deterministic Technology

Deterministic parallel technology has always been a key issue in the research of universities and research institutions. The current deterministic programs and achievements are mainly reflected in the following aspects:

1) Design Method

Joseph D, Brandon L, Luis C, Mark O et al. proposed the DMP system[2]. The RCDC system[3] proposed by Devietti and Joseph of the University of Washington et al is implemented in parallel at the thread level by both hardware and software. The DMP system converts the system from parallel to serial execution in the event of a memory conflict, guaranteed by sacrificing parallelism. The RCDC system was further modified on the basis of DMP, which weakened the memory consistency and used DRF (Data Race Free) relaxes the consistency model by tracking the happens-before relationship due to thread synchronization. Locking operations that do not have a happens-before relationship can be performed in parallel while satisfying certainty. Calvin system It can only be implemented by hardware, which can ensure the certainty of the whole system, but it is difficult to guarantee a single program because it is realized from the hardware point of view. The certainty of the program is therefore not widely used [4].

2) Design Level

dOS and Determinitor and DDOS[5,6,11] both implement parallel determinism issues as an operating system, embedded in Small-scale real-time operating system on the platform, due to fewer applications, the system kernel is simple and easy to write and schedule in the system. The other ways to achieve certainty, while most other systems are implemented as a runtime system, such as CoreDet, Tern And DTHREAD etc[7,8,9].

3) Paralle Level

The system that implements deterministic parallelism is divided into a multi-threaded implementation of shared memory and a multi-process implementation without shared memory. The DMPREAD and Kendo [10] mentioned above, and the hardware-implemented DMP are thread-level deterministic systems. DDOS and dOS achieve determinism at the process level.

B. Multi-threaded Weak Determinism

Multi-threaded weak determinism is a method of guaranteeing certainty based on weak deterministic thinking at the multi-thread level. The multi-thread standard POSIX threads set the running standard for multi-threaded programs, but the standard does not meet the deterministic requirements. Weak determinism begins with the perspective of synchronous competition, the threads are isolated from each other. The threads are run in units of transactions, and the internals are divided into two phases: parallel and serial. Data competition that may be deterministic may be resolved by means of thread memory isolation. Because thread memory is isolated, memory as the last-level shared resource of the thread loses its original role, which is also to achieve the deterministic cost, so this article replaces LRU from the cache level by using the LIRS replacement algorithm that has better performance in multi-threading algorithm.

III. Multi-thread Weak Deterministic Method Design

A. Structure of Thread Weak Deterministic

In the system control module, the thread initialization and completion state settings are first completed. The thread initialization needs to set the thread state information, including the initialization state, the protection state, the number of child threads, the number of locks held, and whether the token is obtained. The protection state defines a bool variable that can determine whether the current thread is protected by memory. When there is only one thread in the system, the system will turn off the memory protection to avoid protection overload. After the state is initialized, the thread number of the current thread can be obtained, and the thread is added to the token queue, waiting to acquire the token. The thread running mode is shown in Figure 1.

According to Pthreads, in a multithreaded program, a thread can create a thread. After a new child thread is created, it needs to be registered. The relationship with the parent thread has been established to facilitate sharing information between threads. When the child thread registers, it needs to obtain the global index of the thread and associate it with the parent thread. Newly registered child threads cannot lock and get tokens to avoid deadlocks. For the newly registered child thread fence is always valid, after waiting for the synchronization point, you need to wait for the token to get through the fence to enter the next round of events.

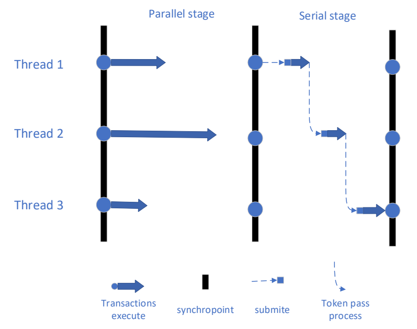

Fig.1 Deterministic thread running mode

B. Memory Configuration Based On Weak Determinism

Compared with strong determinism, weak determinism has the characteristics of small system overhead, good deterministic effect and high efficiency. However, the concept based on weak determinism makes this method ignore data competition and only handle thread synchronous competition, and weak deterministic processing. Data competition is associated with memory access. In the parallel phase of thread execution, each thread can only access the last round of saved backups. In the serial phase, the respective backups are written in the order of acquiring tokens. Into the memory, after the serial phase is over, each thread will back up the current memory. Therefore, the thread needs to know at any time whether the state of the memory is accessible. After the start of a new round of thread execution, the memory information acquired by the thread is also initialized to match the order of the tokens obtained in the current serial phase. In order to back up the memory information, each thread is assigned its own stack. At the beginning of each new round of transactions, you need to call begin() to complete the initialization, and determine whether the content submitted by the previous round of the transaction still remains on the stack. If not, the stack is cleaned up, and the current round of transactions is serialized. Phase commits are ready to set protection for global pages and stacks. When a new thread is created, the child thread needs to complete registration to establish a relationship with the parent thread. At the same time, by calling the setThreadIndex() function, it establishes its index in the global page and stack response position, and completes the relationship with the memory. After the new thread allocates memory, it needs to find the real thread number tid through the global index of the thread to calculate the usage of the sub-heap. When the sub-heap is occupied, the system needs to apply for more memory. See Figure 2 for deterministic memory usage.

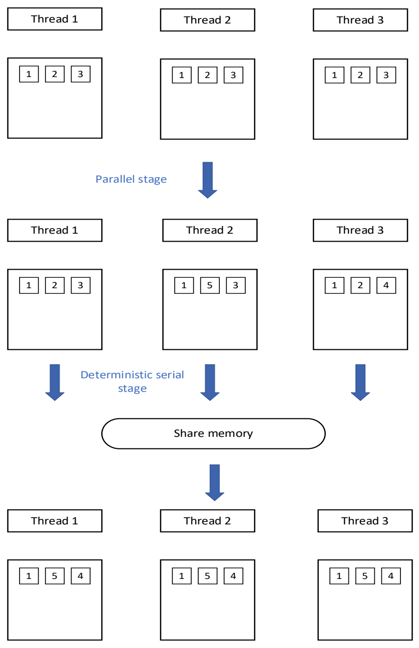

Fig.2 Deterministic Memory Usage

IV. Atomic Transaction Synchronization Control

Synchronization control mainly includes synchronization point waiting, fence placement and clearing, and mutex lock acquisition. In this topic, in order to control thread execution, the concept of transaction is added. The thread runs in transaction units, and the transaction is divided into parallel phase and serial phase. In the parallel phase, each thread does not interfere with the parallel execution, and saves the respective running results to the sub-heap of the initialization function allocation when the thread is created. The contents of these sub-heaps all come from the copy of the shared memory after the last round of the transaction (Share memory), the purpose of setting the synchronization point has the following two points:

1) After the parallel phase, let the thread wait for the token to be submitted and commit to the shared memory

2) After the serial phase, copy the contents of the shared memory to each thread and start the next round of transaction execution.

At the synchronization point, each thread submits to the shared page in the order of acquiring the token, and correspondingly writes less, increments the version number of the page, and when all submissions are completed, the shared page is released. The setting of the synchronization point interrupts the execution of the thread, providing an opportunity for the thread to compare and submit the mirror page, allowing each thread to submit in the order of acquiring the token, ensuring certainty.

The purpose of setting the synchronization point is to let the thread know when to execute in the parallel phase and when it needs to stop and commit. When the thread runs to the synchronization point, the system sets the fence to block all threads that reach the synchronization point, performs synchronization operations, and when all threads complete the commit, clears the fence release, and the system needs to notify the system to re-declare the index when the fence is closed. When there is only one active thread in the system, the thread's runnability and commit are not blocked by the fence. The system uses a single global token to ensure the sequence certainty and atomicity of the serial phase. When the thread acquires the token, it needs to apply for the lock on the submitted page when it submits like shared memory. First, the system should check whether the current thread has obtained any locks. When the thread wants to acquire the lock, it must wait for the token to be passed to itself. If the lock has been acquired by other threads, the thread must pass the order.

Until the next serial phase condition allows the lock to apply to prevent the occurrence of deadlock, as shown in Figure 3.

T1 trys to occupy the R1 and T2 trys to occupy R2 in the same time. Both threads are waiting the other one to release the resource and hole the current resource with occupy state, so here is a dead lock situation

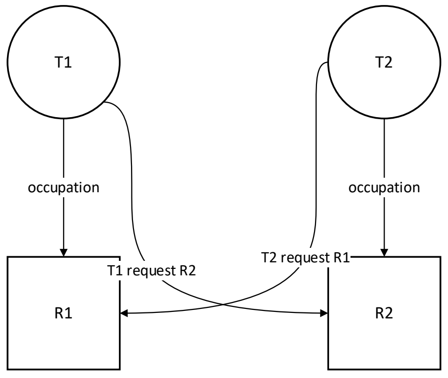

Fig.3 dead lock illustration

V. Deterministic Execution Scheduling

When the thread finishes executing in the parallel phase, the arrival syncpoint is blocked by the fence and joins itself to the active list, which means that the thread is ready to wait for the token to start the serial phase. The token is delivered in a first-come-first-served algorithm, which is passed in order of the threads in the active list. The delivery of the token is an important measure for the system to ensure that the execution is deterministic. In the above, the design thread acquires the lock, and the commit modification needs to be performed under the premise of acquiring the token. The active list of threads is designed as a circular linked list. Every time you operate on a linked list, you must first determine whether the linked list is empty. The system determines whether the head pointer of the circular linked list points to null (NULL). When inserting an entry into a linked list, if the linked list is empty, both the head and tail pointers point to the current entry. If the linked list is empty when the withdrawal operation is performed on the linked list, the token is passed, the fence is released, and the next round of transactions is started.

VI. LIRS Cache Policy

Multi-threaded systems must be paid for system resources or time in order to obtain certainty. Therefore, the actual use of many deterministic systems is not ideal. The strong determinism of software implementation is too restrictive for threads and memory. Nearly 10 times the overhead, which makes the application of this technology very limited. Therefore, optimizing system performance from thread resource scheduling and improving system performance have become a better choice for improving deterministic systems.

LIRS is a replacement by dynamically using the distance between the same page twice (this distance refers to how many non-repeating blocks are accessed in the middle) as a measure to dynamically sort the access pages. The work of all page replacement algorithms relies on the existing locality principle. The main difference between the various replacement algorithms is how to quantify the locality [13]. LIRS uses the distance that the same page is accessed twice, that is, between consecutive two visits to the page, how many non-repeating pages are accessed to quantify locality. If a page is first accessed, its reuse distance is infinity, and at the same time, the LIRS algorithm uses a page's recent access time to quantify this locality. In order to take into account the latest access history information, the implementation of the LIRS algorithm uses the larger value of the revisit interval and the recent access time of the page to measure this locality, using RD-R representation (Reuse Distance-Recency) to revisit the interval and the newest The concept of access time is shown in Figure 4 below

The LIRS algorithm classifies cache blocks according to IRR values, and is divided into two types, LIR and HIR, to set a stack S and a queue Q respectively. All LIR blocks and newly replaced non-resident memory blocks are placed in stack S. The newly replaced cache block is placed on the top of the stack. The queue Q is used to place all the resident HIR blocks. The distance from the block in the stack S to the bottom of the stack represents the IRR value of the cache block. Replace all blocks that reside in the cache.

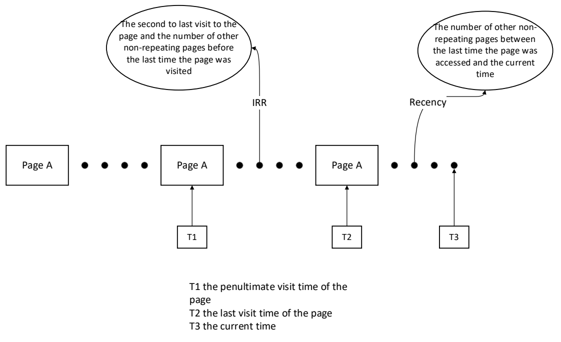
The second to last visit to the page and the number of other non-repeating pages before the last time the page was visited
The number of other non-repeating pages between the last time the page was accessed and the current time
IRR
Recency
Page A
Page A
Page A
T1
T2
T3
T1 the penultimate visit time of the page
T2 the last visit time of the page
T3 the current time

Fig.4 LIRS Cache strategy illustration

VII. Testing and Verification

A. Environment

This experiment uses Phytium FT-1500A/16 domestic 16-core experimental platform, clocked at 1.5GHz, 16G memory, equipped with Linux version 4.4.13-20161128. kylin.5. server operating system, using standard test set PARSEC-3.0

B. Test Results

The figure below shows the function compared by our system and pthreads, the input file size is simsmall.

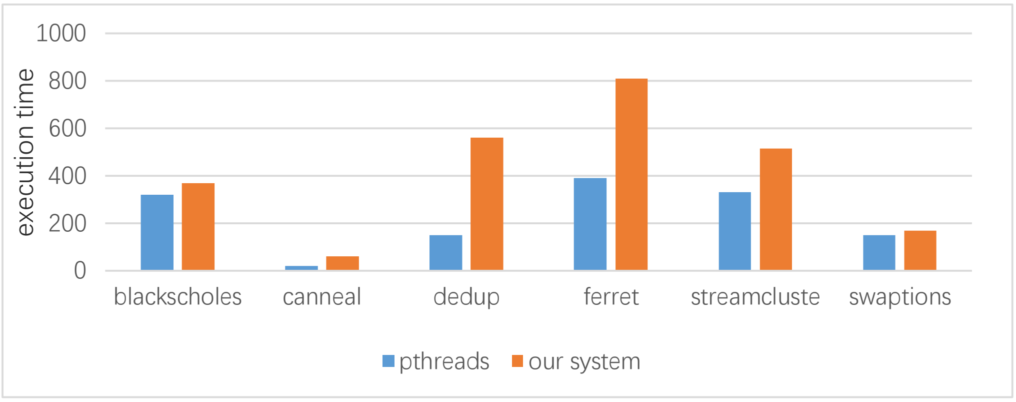

Fig.5 function compare between our system and pthreads

VIII. Conclusion

The weak deterministic implementation ignores data competition. Compared with other deterministic methods, it has advantages in system performance overhead. It can be seen from the performance test chart that when the test set program parallelism is high, the performance overhead of the two is very close. However, when the degree of parallelism of the program is low and the degree of serialization is high, the system often has several times of overhead in order to obtain parallelism, which is also a problem to be solved in future research.

References

[1] Xu Zhou, et al. Chinese Journal of Computers, 2015,(5):974-983.

[2] Joseph D, Brandon L, Luis C, Mark O, DMP: Deterministic shared memory multiprocessing//Proceedings of the 14 th International Conference on Architectural Support for Programming Languages and Operating Systems. Washington, USA, 2009:85-96.

[3] Devietti J, Nelson J, Bergan T, et al. RCDC:A relaxed consistency deterministic computer// Proceedings of the 16 th International Conference on Architectural Support for Programming Languages and Operating Systems. Newport Beach, USA, 2001:67-78.

[4] Hower D R, Dudnik P, Hill M D, Wood D A. Calvin: Deterministic or not? Free will to choose//Proceedings of the 17 th International Symposium on High Performance Computer Architecture, Washington ,USA, 2011:333-334.

[5] .Bergan T, Hunt N, Ceze L, Gribble S D. Deterministic process groups in dOS//Proceedings of the 9 th USENIX Conference on Operating Systems Design and Implementation. Vancouver, Canada, 2010:1-16.

[6] Amittai a, Shu-Chun W, Sen H, Bryan F. Efficient system enforced deterministic parallelism// Proceedings of the 9 th USENIX Conference on Operating Systems Design and Implementation. Vancouver, Canada, 2010:1-16.

[7] Bergan T, Anderson O, Devietti J, et al. CoreDet: A compiler and runtime system for deterministic multithreaded execution//Proceedings of the 15 th ASPLOS on Architectural Support for Programming Languages and Operating Systems, Pittsburgh, USA, 2010:53-64.

[8] Cui H, Wu J, Tsai C C, et al. Stable deterministic multithreading through schedule memoization[C]// Usenix Symposium on Operating Systems Design and Implementation, OSDI 2010, October 4-6, 2010, Vancouver, Bc, Canada, Proceedings. DBLP, 2010:207-221.

[9] Liu T, Curtsinger C, Berger E D. Dthreads:efficient deterministic multithreading[C]// ACM Symposium on Operating Systems Principles 2011, SOSP 2011, Cascais, Portugal, October. DBLP, 2011:327-336.

[10] Olszewski M, Ansel J, Amarasinghe S. Kendo: efficient deterministic multithreading in software[C]// International Conference on Architectural Support for Programming Languages and Operating Systems. ACM, 2009:97-108.

[11] Aviram A, Weng S C, Hu S, et al. Efficient system-enforced deterministic parallelism[J]. Communications of the Acm, 2010, 55(5):111-119.

[12] Lu K, Zhou X, Wang X, et al. RaceFree:an efficient multi-threading model for determinism[J]. Acm Sigplan Notices, 2013, 48(8):297-298..