2010 3rd International Conference on Computer and Electrical Engineering (ICCEE 2010)
IPCSIT vol. 53 (2012) © (2012) IACSIT Press, Singapore
DOI: 10.7763/IPCSIT.2012.V53.No.2.12

Investor: A Light-Weight Speculation Scheme for the Open Internet

Huiba Li+, Shengyun Liu, Yuxing Peng and Xicheng Lu

National Laboratory for Parallel and Distributed Processing

National University of Defense Technology

Changsha, China, 410073

Abstract—Investor is a novel speculation scheme that targets at the open Internet. It makes use of standard HTTP and *speculation by convention* to fit into this environment. Investor is a runtime technique that doesn't modify the languages, compilers or binary formats of the applications, so it is compatible well with existing programs or libraries. Investor can overcome some form of control dependency and data dependency. Preliminary experiments show that Investor can significantly improve the overall performance of the applications.

Index Terms—Speculation, Distributed Systems, Internet, HTTP

1. Introduction

Speculation has been an important approach to exploit po- tential concurrency opportunities with uncertainty. For exam- ple, modern high-performance processors make extensive use of speculation in some forms, such as out-of-order execution, branch target prediction. As a qualitative analysis, we run the SPEC2000 benchmark suit in Simple Scalar 3.0 default configuration with speculation turned on and off respectively. The results shows various speedups ranging from 1.57 to 2.30. Speculation has also achieved great success in some other centralized systems, such as database management systems and parallel simulation. Introducing speculation into dis- tributed systems is recently becoming a hot topic [4–6, 8–10]. According to these works, speculation can bring significant performance gains. However none of these works try to consider speculation in the open Internet environment.

The open Internet environment has distinct properties that are new challenges for speculation. First of all, the Internet tends to have much greater latencies than LANs, so the system design and protocol must try to minimize the need for commu- nication. Secondly, the Internet incorporates a large number of administrative domains with various security policies, so the system must allow for gradual adoption, in order that existing applications can easily evolve to make use of speculation. Thirdly the system design should be simple enough so that third-party implementations are easily available, even in other programming languages. Last but not least, it is extraordinarily desirable to be compatible with the web, so as to inherit the success of the web, as well as the vast amount of existing resource on it.

These challenges inspire the development of Investor, a light-weight speculation scheme for the open Internet. To achieve this goal, Investor relies on standard HTTP as the communication protocol, and it introduces speculation to HTTP through *speculation by convention*. This terminology in fact doesn't refer to a new idea. Instead, it is really an idea borrowed from the area of computer architecture. The design of memory and its controller are not much influenced by the speculative behave or of CPUs. They are not aware of speculation at all. Investor parallels this principle, with HTTP servers being the memory

+ Corresponding author.
E-mail address: lihuiba@163.com

modules, HTTP being the memory bus, HTTP clients being the processors and the Internet being inter-processor network.

HTTP consists of the following methods: GET, HEAD, POST, PUT, DELETE, TRACE, CONNECT and OPTIONS. A convention has been established that “the GET and HEAD methods SHOULD NOT have the significance of taking an action other than retrieval”[2]. So they behave much like read operations. On the other hand, POST, PUT and DELETE are supposed to change the state of the server, so they can be taken as write operations. Other methods are not frequently used, so they are currently not taken into account by Investor.

Investor makes use of the convention to speculate “read” operations, while ensuring not to change the order of any “write” operations and the relative order of any “read”-“write” pair, if they access the same resource.

Speculation by convention not only addresses the challenges discussed above, it’s also a generic approach that can be applied in many different applications. However, speculation by convention still attaches some limitations to Investor. For example, it’s not equipped with a consistency protocol working among the nodes, so speculative execution may not produce a result exactly the same with non-speculative execution. For another, applications that doesn’t (or cannot) use HTTP are not able to use Investor. These limitations are not crucial to Investor, because HTTP has been the most widely used protocol in the world, and we believe most applications can make only use of HTTP. And, consistency problem is inherent to the Internet environment itself, so applications running in it should relax the requirement for consistency, no matter what middleware to use.

Investor relies on the following preconditions to result in benefits: (1) the hardware has spare resources that can be used to do speculative execution; (2) the communication responses can be correctly predicted with high probability; and finally (3) the overhead of checkpointing and rollback is low. Hardware resources usually grow exponentially, and many applications are not able to fully occupy CPU, RAM and network bandwidth. According to a research on return- value prediction[12], the accuracy of prediction can be as high as 20%-60%, which suffices for Investor. The overhead of checkpointing and rollback is, according to our experiments, usually less than a few milliseconds, which is at least an order less than Internet latencies. It is also the reason why we call it a “light-weight” scheme. So these facts promises that Investor is likely to result in benefits.

Our implementation of Investor is based on the virtual machine of the Lua programming language. We choose Lua because of its simplicity and efficiency, though the approach is general enough to be applied in many other languages. Our primary contributions in this paper are: (a) the scheme for speculative execution in the open Internet environment; (b) the design, implementation and evaluation of such scheme.

This paper is organized as follow: section 2 gives out related works; section 3 discusses design considerations; section 4 discusses implementation; section 5 evaluates the approach; and section 6 concludes the whole paper.

2. Related Work

A wide range of techniques can be understood as applica- tions of speculation [7], such as optimistic concurrency control in databases, and more recently in transactional memory, prefetching in memory and file systems, branch prediction and speculative execution in modern CPUs. Speculation in distributed systems has just become a hot topic for re- search. Existing works can be generally divided into two category: (1)distributed speculation for a specific purpose; and (2)general-purpose distributed speculation.

Speculation is very important to parallel and distributed simulation [4]. It allows each logical process to progress without knowing whether there will be any violations, and it allows the processes to rollback when a violation eventually occurs. Jefferson’s Time Warp[6] mechanism was the first and remains the most well-known speculation algorithm in this field. Many of the fundamental concepts and mechanisms in speculation such as rollback, anti-messages, and Global Virtual Time (GVT) first appeared in Time Warp. However, these systems and algorithms are specially designed for simulation, and they can be hardly applied to other applications.

Transactions in distributed database systems are usually speculative executed in parallel, and checked against for conflicts when they commit. The two-phase commit[5] pro- tocols are the most fundamental approach for commitment coordination. They realize the effects of atomic committing by insisting that all sites involved in the execution of a distributed transaction agree to commit the transaction before its effects are made permanent.

Nightingale in [9] proposed speculator, a scheme for spec- ulative execution in a distributed file system, and achieved great results. He summarized three facts that support his speculation: first, the clients can correctly predict most results of many operations; second, the time to take a checkpoint is less than a round-trip delay; finally, the client has spare resources to execute the speculative process. Speculator allows multiple processes to share speculative state by tracking causal dependencies propagated through inter-process communica- tion. It guarantees correct execution by preventing speculative processes from externalizing output, e.g., sending a network message or writing to the screen, until the speculations on which that output depends have proven to be correct. For PostMark and Andrew-style benchmarks, speculative execu- tion results in a factor of 2 performance improvement for NFS over LANs and an order of magnitude improvement over WANs. Speculator enables the Blue File System to provide the consistency of single-copy file semantics and the safety of synchronous I/O, yet still outperform current distributed file systems with weaker consistency and safety.

The above-mentioned works are all aiming at specific areas, and none of them can be applied directly out of their scope. Cosmin Oancea in [10] declared to be the first to propose dis- tributed speculation for general purpose systems. He extended Generic IDL [11] — a distributed component system — to support speculation. This approach is indeed a general-purpose one, but it has some limitations. Firstly, it confines the users to its specific implementation, and the large majority of existing systems are hard to gradually adopt this technology. Secondly, its design is based on generic programming techniques, which is originally a very complex construct in C++, and it is not well support by other languages. Thirdly, its protocol is complex and not easily standardizable, making third-party implementations or middle-ware inter-operations hard to be available. Finally its improper abstraction makes it impossible to implement an application-independent cache, which is an important optimization in real life applications [1, 3].

As the behavior of speculation relies on some stochastic factors, it can make applications run more fast, but not neces- sarily more efficient. As long as the running application comes across a rollback operation, the corresponding speculation can be considered as a “waste”.

3. Overview

Communications in distributed systems are essentially inter- task dependencies, which require the tasks to block and wait for the corresponding replies, otherwise inconsistencies may occur. For example, a write operation to a distributed file should block the task until the server acknowledges it, or the application will be risking corruption or even loss of data.

But in fact, it is not always necessary to wait that long before the task can continue, because some replies are not immediately used, or some of them can be predicted with high probability. For example, Listing 1 shows an imaginary e-commerce code segment that accesses user information in Key-Value stores. This segment first updates the time of last login, and then loads various pieces of user information. The execution of this code segment will be blocked at every line until a reply is received. But in fact, the results of these operations are not immediately used, and it is highly

```
//uid is the user’s identifier, and
//services is the facade for Key-Value stor
try {
services.lastLogin.put(uid, now);
Cart cart=services.carts.get(uid);
Message msg=services.messages.get(uid);
  Notification noti=services.notifications.get(uid);
} catch(Exception e) { ... }

...
```

Listing 1. An imaginary e-commerce code segment that accesses user information in Key-Value stores.

Predictable that these operations are either likely to complete successfully (in the case that everything is OK), or likely to fail (in the case that something is wrong). And some return values can even be predicted with high probabilities by proper mechanisms.

This example illustrates two types dependencies usually found in applications: (1)control dependency (wait for a suc- cess or failure); and (2)data dependency (wait for a return value). This section will show how Investor overcomes these dependencies with speculative execution.

3.1. Architecture

Investor supports the client-server architecture, and it allows a client continues to (speculatively) execution after sending a request. It actually provides two levels of speculation: the first is to speculate that the request will be delivered and processed successfully, so the client doesn't need to wait for the response, if it not immediately used; and the second level is to speculate that the response can be predicted correctly, so the client can use the predicted response before the actual one arrives. The simplest prediction may be a cache, which returns the recent response for a request. This topic will be further discussed in section III-B. If there's no enough information to predict the response of a given request, or the prediction has been proved to be inaccurate, Investor just bypasses the second level speculation for the current request. Each time when Investor is about to speculate, it makes a checkpoint of current execution. As soon as a speculation is subsequently proved to be right, Investor will release the corresponding checkpoint, otherwise it will rollback the execution to the checkpoint.

When there are multiple consecutive requests, the execution paths will be interlaced with one another. Investor records the correct order of the checkpoints, and whenever an rollback occurs, Investor knows what checkpoints to process. When an rollback needs to cancel some unfinished requests, Investor simply closes the corresponding connections. Rollbacks don't process finished speculative request, they just abandon the responses. But the responses can be still in the cache, which may be helpful for following requests. The fact that Investor doesn't cancel finished requests can bring some side effects, so we require these effects, if any, to be benign. This requirement is moderate, and it is also found in other speculation systems. For example, a prefetch in memory or file system can turn out to be unnecessary, and the corresponding request needs not to be canceled, if it is finished.

3.2. Prediction and Predictive Cache

The nature of response prediction is similar to that of return value prediction, which has been studied previously in the area of Thread-Level Speculation. Ref. [12] carried out detailed evaluations on 14 predictors and 7 benchmarks, and the results are promising. Most predictors can achieve an accuracy of 20% ~ 60%, and some even approach 80%. Most predictors consume memory less than a few mega- bytes, and consume CPU cycles less than 3 times of normal execution. In communication-intensive applications that have spare hardware resources, predictor executions can be masked behind communications, so these numbers suggest that pred- ication is good enough to guarantee real benefits, because communication latencies are several orders higher than the overhead of checkpointing, prediction and rollback.

The simplest prediction in distributed systems may be the cache. It is not a prediction in processors, because it grantees 100% accuracy. But it is a prediction in Investor (HTTP, and many other distributed systems), because there's no coherence protocol running among the nodes, so it may provide stale responses. Existing HTTP cache can be seen as a variation of the Last Value predictor mentioned in [12]. It relies on an expiration mechanism to get a cache item invalidated. But the expiration can be difficult to set, as a short one would waste the potential performance, and a long one would increase the risk of data incoherence. The expiration mechanism won't exactly reflex the actual life cycle of the data item. We denote the unexpired but incoherent cache items as *false positives*; and the expired but still coherent items as *false negatives*.

Based on speculation, we propose Predictive Cache (Pred- Cache) that minimize the false positive and false negative problems in cache. The main idea is that when a request hits the PredCache, but the PredCache find the response may have become stale, then the PredCache checkpoints the whole application and returns the cached response as a prediction, and at the same time it asks the server for a validation. If the cached response does have expired, the PredCache will rollback the application and feed it again with the right response. Or if the cached response is proved to be fresh, the PredCache will release the checkpoint.

4. Design and Implementation

Investor consists of many runtime services, such as check- pointing, rolling-back and speculation. These services can be implemented as library routines, and programmers will be in charge of inserting invocations to these services on proper occasions. That design is repetitive, mechanical and error- prone, so we try another one: embedding the services into the runtime system. We design and implement a prototype of Investor based on the virtual machine of Lua, which is a dynamically typed scripting language. A virtual machine is not essential to Investor. It is introduced only for ease of application developing.

4.1. Checkpointing and Rollback

Investor implements checkpointing by performing a copy- on-write fork of the process, so any operations performed in the main process won't affect those in the checkpoint. Checkpoints are created on the demand of the application through system calls, and there should be no pending signals on this occasion, so Investor doesn't need to save them, unlike what has been done in Speculator [9].

In order to make the checkpoint process "freezed", Investor make it attempt to read a few bytes from a pipe as soon as it is created. The pipe is shared between the checkpoint and the running process, and it will be written by the running process only when the corresponding speculation turns out to be a failure, so the checkpoint will get blocked until resumed by a rollback, or killed by a release or a crossover rollback.

Checkpoints are organized in a linked list in time order.

Investor implements rollback and release by killing the unnec- essary process(es). When a speculation is proved to be right, Investor release the corresponding checkpoint by killing the checkpoint process and removes it from the list. When rolling- back to a checkpoint, say x, Investor releases all checkpoints that comes after x, and resumes x by writing data to the corresponding pipe, informing x what checkpoints that comes before it have been released. The running process then kills itself.

The process of checkpoint x then becomes the running process. It gets information from the pipe, and removes from the list any checkpoints that has been released previously. Killings are not necessary because they should have been killed already.

4.2. Speculation

The semantics of rollback is to cancel any modifications made to the memory, but not to undo any file writings or console printings. Investor requires these activities to be buffered until the speculations on which they depend are all proved to be right. This occasion is called *commit*.

Any readings on files are assumed to have no side-effects in Investor by default. This assumption implies that if a file is read in speculative state, then the process is rolled-back and the file is read again, there should be no difference in the file content, or the difference is acceptable. Although this assumption is right in common cases, it is not in some cases. For example, the file may be removed by other programs between the two identical read operations. For another, stream- typed files should not be read speculatively, such as stdin. Application developers hold the liability to judge whether the default behavior of Investor is correct or not, and take necessary measures. They can insert invocations to a special function `wait_specultion()` to wait any prior specula- tions to commit, or they can set the the file type to a stream so that Investor will always invoke `wait_specultion()` before performing actual reading.

Investor maintains a separate commit queue for each spec-ulation segment. Being a prototype, the commit queues cur- rently only deal with file writes, but there should be noessential difficulties to support other types of commitment. Each item in the queues consists of file descriptor, position and data that can describe a complete write operation. An item with empty data represents a close operation.

We have modified all related I/O functions in the standard library of Lua to make use of speculation. So print(), write() and close() will buffer their actions in specula- tive mode, instead of performing them immediately. Investor also exposes an convenient interface to C programs, so that third-party native functions can buffer their output if needed.

4.3. Networking

It is a bit surprising that networking service is running in a dedicated process created by forking when the VM get initialized, because it shouldn't be affected by checkpointing and rollback. For example, suppose there are several concur- rent TCP connections receiving responses from server when a checkpoint x is created. If some time the process needs to roll back to x, data transferring state is not to go back, as the server is not aware of speculation.

The main process cooperates with the networking service process as follow: (1) the main process send a request to the networking service process via request pipe; and (2) the main process immediately receive a predictive response via response pipe$_1$, and the service process may send a request to the server to verify its prediction; finally (3) the service process has received a response from the server, so it send a signal (SIGUSR1) to the main process and write the validation result to response pipe$_2$.

Signal handlers have a lot of restriction on its behavior, so as to avoid race conditions, so the main process only atomically increase a global counter by 1 that represents the number of validation results to be read from response pipe$_2$. Each time when the virtual machine of Lua is to execute the next virtual instruction, it checks the counter and receive any validation results from the pipe. It always receives all the results before it performs rollback, if needed. And it always rollback to the earliest checkpoint, if there are multiple wrong speculations.

5. Evaluation

To evaluate Investor, we write a small tool *WSave* that recursively download a whole web site. We run this tool in LAN, instead of the Internet environment, so as to avoid the randomness factor. We setup a server that serves a mirror site of http://www.lua.org, through a network with various latency settings.

5.1. WSave

The state of PredCache significantly influences the perfor- mance of WSave, and there is no standard initial state for such benchmark. Considering that cache performance is not a primary focus of this paper, we set it to be initially empty in this test, so Investor is only to exploit the first level speculation in WSave.

For each page that has been fetched, WSave parses its content to extract a full set of URLs and pictures in it. For each such set, WSave saves its elements through 2 loops, in which one is to issue **http_get()** requests, and the other is to save the responses one by one. Investor is able to concurrently execute the first sequentially-written loop, but may block on any iteration of the second loop, to wait for the corresponding response.

For comparison, we write a manually optimized event- driven version of WSave in standard Lua with luasocket extension. this version is expected to be more efficient than In- vestor, because sub-tasks in it are organized so that they won't block one another. We also ran WSave in non-speculative (sequential) mode, which is expected to be less efficient.

Figure 1 shows the results of running the 3 WSaves.

This suggests that Investor exploits considerable amount of potential concurrency in WSave.

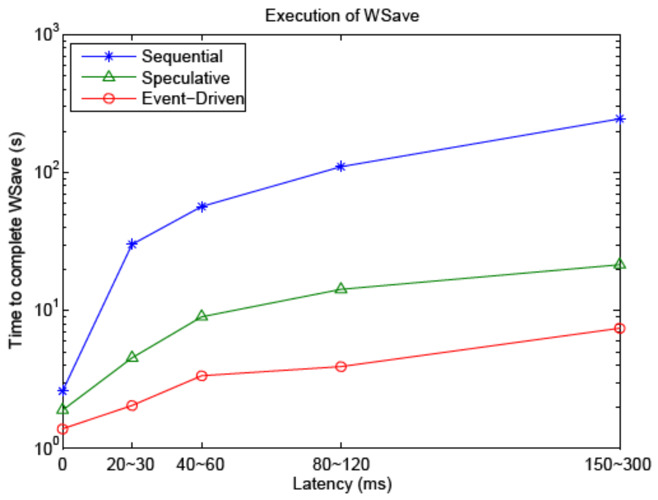

Fig. 1. Running 3 WSaves with different latencies

5.2. Checkpointing, Releasing and Rollback

We perform a micro benchmark on checkpointing, releasing and rollback. Theoretically, the overhead of these actions can be approximately modeled as $c + n * CoW$, where c is a fixed cost payed on the invocation, and n is the number of copy-on-write (CoW) actions.

In this test, we control the number of memory pages written between a checkpoint-rollback pair, or a checkpoint-release pair. Figure 2 shows the results, which highly confirms the above model.

6. Conclusion

Speculation is an old technique and this paper tries to place it in a new environment — the open Internet. This environment tends to have much higher latency than memory access or local I/O device, thus applications usually have to execute concurrently to achieve reasonable performance. Concurrency control, however, is often a nightmare. Investor provides the developers with a new simple means to control concurrency. Although it has limitations, Investor is much simpler than handcrafted concurrent code and it is much faster than sequential execution. We expect Investor to suit for many cloud computing applications and large-scale enterprise computing applications as well.

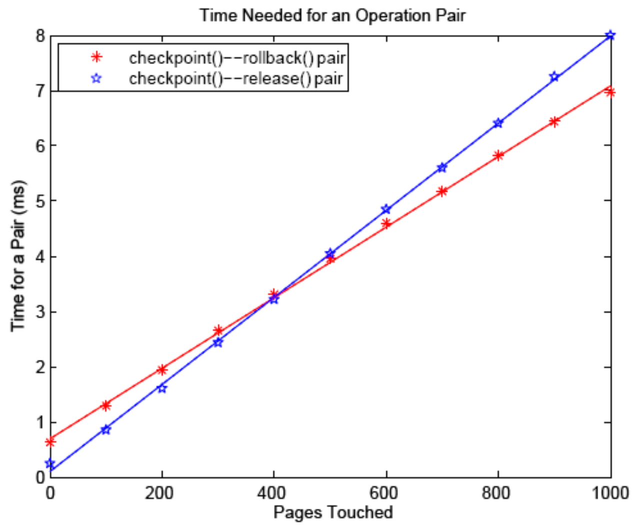

Fig. 2. The Overhead of Checkpointing Operations

7. Acknowledgements

This work is partially sponsored by the National Basic Research Pro- gram of China under Grant No. 2005CB321801, and the National High Technology Research and Development Program of China under Grant No. 2009AA01Z142.

8. References

[1] decandia , g., hastorun , d., jampani , m., kakula pati , g., lakshman , a., pilchin , a., sivasubramanian , s., vosshall , p., and vogels , w. Dynamo: amazon's highly available key-value store. Sigops oper. Syst. Rev. 41, 6 (2007), 205–220.

[2] fielding , r., gettys , j., mogul , j., frystyk , h., ma sinter , L., leach , p., and berners -lee , t. Hypertext transfer protocol- http/1.1.

[3] FIELDING, R. T. *Architectural styles and the design of network-based software architectures.* PhD thesis, 2000. Chair-Taylor,, Richard N.

[4] FUJIMOTO, R. Parallel simulation: parallel and distributed simulation systems. In *Proceedings of the 33nd conference on Winter simulation* (2001), IEEE Computer Society, p. 157.

[5] GRAY, J., AND REUTER, A. *Transaction processing: concepts and techniques.* Morgan Kaufmann Pub, 1993.

[6] EFFERSON, D., AND SOWIZRAL, H. Fast Concurrent Simulation Using the Time Warp Mechanism. Part I. Local Control, 1982.

[7] LAMPSON, B. W. Lazy and speculative execution in computer systems. In *ICFP '08: Proceeding of the 13th ACM SIGPLAN international conference on Functional programming* (New York, NY, USA, 2008), ACM, pp. 1–2.

[8] LIU, J., GEORGE, M., VIKRAM, K., QI, X., WAYE, L., AND MYERS, A. Fabric: a platform for secure distributed computation and storage. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles* (2009), ACM, pp. 321–334.

[9] NIGHTINGALE, E. B., CHEN, P. M., AND FLINN, J. Speculative execution in a distributed file system. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles* (New York, NY, USA, 2005), ACM, pp. 191–205.

[10] OANCEA, C., SELBY, J., GIESBRECHT, M., AND WATT, S. Distributed models of thread-level speculation. In *Proceedings of the 2005 Inter- national Conference on Parallel and Distributed Processing Techniques and Application* (2005), Citeseer, pp. 920–927.

[11] OANCEA, C., AND WATT, S. Generic library extension in a hetero- geneous environment. In *Library Centric System Design Workshop (LCSD06)* (2006), Citeseer.

[12] PICKETT, C., VERBRUGGE, C., AND KIELSTRA, A. Adaptive Software Return Value Prediction. Tech. rep., Technical Report SABLE-TR- 2009-1, Sable Research Group, School of Computer Science, McGill University, Montréal, Québec, Canada, 2009.