Safe Parallel Programming in an Interpreted Language

Chen Ding, Brian Gernhardt, Pengcheng Li
Department of Computer Science
University of Rochester
Rochester, New York

Matthew Hertz
Department of Computer Science
Canisius College
Buffalo, New York

1. Introduction

Parallel programming is increasingly important with the advent of multicore processors. However, modern software is difficult to parallelize because of the high degree of modularization. It is unclear whether a piece of code is parallel if it calls other functions. Dynamic languages such as Ruby, Python, and Matlab represent modularization to the extreme. A program, also known as a script, requires interpretation for every statement, calling a whole host of functions from parsing to execution. Because of the complexity of the interpreter and the dynamic nature of script code, scripting languages are among the most difficult to parallelize.

In this paper, safe parallelization means to parallelize a sequential program with the guarantee that the parallelized version produces the same result as the original sequential code. In other words, safety means *sequential equivalence*.

There are generally three ways to ensure safety. Consider a sequential loop as the target program to parallelize. The first way is static. A compiler analyzes the loop and determines whether it is parallel. Static parallelization means that the loop is parallel in all executions. The second way is dynamic. Immediately before executing the loop, a run-time library determines whether the loop is parallel. The third way is speculative. The loop is run in parallel speculatively. If conflicts are detected in the parallel execution, the conflicting part is rolled back and re-run sequentially. This paper uses the third way and describes a system that speculatively parallelizes a Ruby script to ensure safety.

In the past, most speculation systems were for automatic parallelization and were not user programmable [2, 5, 10]. Then a number of interfaces appeared, including *safe future* in Java [13], ordered transactions using TLS hardware [12], and possibly parallel region (*PPR*) in the *BOP* (behavior-based parallelization) system [4, 6, 14], and multi-threaded transactions [9]. These are effectively hints of parallelism. Dependence hints were introduced in the *BOP* system as a way to suggest possible dependences between operations in different *PPR*s [7]. In this paper, we adapt the *BOP* interfaces for use in Ruby.

2. Safe Parallel Ruby

Ruby is an object-oriented scripting language with first-class functions. It is "pure" in that almost all programmable entities, including constants and first-class functions, are objects. Every object has a class, which is itself an object.

Safe parallel Ruby adds the following two constructs to Ruby:

- *PPR*{ X }: X is a code block called a *PPR* block. The *PPR* block is possibly parallel with the code after it.
- Ordered{ Y }: Y is a code block called an ordered block. An ordered block is likely dependent on the result of earlier ordered blocks (as ordered in would-be sequential execution).

Note that the parallelism happens between *PPR* blocks, not inside a *PPR* block.

As an example, consider a program that searches two places, A, B. Let's assume that the two searches are independent, but the parallelism is conditional — the second search is needed only if the first returns nil. We can parallelize them using two *PPR* blocks as follows.

Listing 1: Conditional parallel search

```
done = false
PPR{
  r = search( A )
  done = true if r != nil # stop search
}
PPR{
  r = search( B ) if not done
}
```

In the first *PPR* block, we search A and assign a flag variable if the result is not nil. In the second *PPR* block, we check the flag variable before searching B. When executed, safe parallel Ruby runs the two *PPR* blocks in parallel. After they finish, it will check for conflicts, that is, if there is an object written by the first *PPR* block and read by the second *PPR* block. In the case of the flag variable, the correctness check would pass if the flag is not set after the first search.

Then the second search is needed, and the speculation is correct. If the flag is set, however, the correctness check would fail, and the second *PPR* would be rolled back and re-executed.

To demonstrate the use of an ordered block, we consider another example. It searches two places A, B (unconditionally) and combines the results in an array. We use two *PPR* blocks for the two searches. Inside each *PPR*, we use an ordered block to collect the result. The parallelized code is as follows:

Listing 2: Parallel probe but serial sum

```
hits = []
PPR{
  r = search( A )
  Ordered{ hits << r }
}
PPR{
  r = search( B )
  Ordered{ hits << r }
}
```

Safe parallel Ruby guarantees the same output as the sequential execution of the code without hints. In the first example, the result r is the result of the conditional search. In the second example, the $hits$ array stores the two results in the sequential order of the search. Speculation and parallel execution have absolutely no effect on the final result. The only aspect it can change is the execution time.

A common question about the interface is what happens to the code between *PPR* blocks. We may call it the inter-*PPR* code. The inter-*PPR* code is suggested to run sequentially with the *PPR* block after it but in parallel with the *PPR* block before it. A *PPR* block similar to a *future*, invented in Multilisp to specify parallelism between the future and the continuation [8]. `PPR{ X }Y` means that X is possibly parallel with Y. In the two examples, the second *PPR* block is actually unnecessary. The first *PPR* already expresses the parallelism.

3. Implementation Using BOP

Safe parallel Ruby is implemented using the *BOP* system, which supports speculative parallelization in C programs [3, 4, 6, 7, 14]. Ruby *PPR* blocks use *BOP* C functions, which implements *PPR* tasks as a process. Using processes, *BOP* implements speculation in a two-step strategy. The first is copy-on-write in *PPR* tasks to isolate them from each other. This is done automatically by the OS. The second is serial commit after a group of *PPR*s finish. The second step is to merge concurrent changes and resolve conflicts. It is done by the *BOP* run-time using data access information provided by Ruby.

Before merging the changes, *BOP* checks for parallel conflicts. The checking is based on the dependence theory, i.e. the fundamental theory of dependence, which states that a reordered execution produces the same result if the reordered execution preserves all dependences [1]. The copy-and-merge strategy changes dependence checking in two ways [4]. First, copy-on-write removes all false dependences, i.e. write-after-read and write-after-write. The only type of dependences to check is flow dependence, which happens when a predecessor task writes a value, and a successor task reads it. The second is value-based speculation. If the predecessor does not change the original value, or it alters the value but changes it back at the end, the successor, when running in parallel with the predecessor, still reads the correct value in spite of the flow dependence violation. Value checking is more permissive than value speculation because the value is not necessarily a constant (during *PPR*).

If a conflict occurs in the parallel execution, the speculative *PPR*s are abandoned by killing their processes, and the canceled *PPR*s are re-executed sequentially. Tian et al. called their system CorD to indicate that the speculation results are either copied or discarded [11]. To combine the acronyms, we may say that the strategy is first COW (copy-on-write) and then CorD. The step after COW may also be called merge or discard (MorD, or MORD for ministry of rural development).

4. Conclusion

We have implemented a prototype called *RubyBOP* from the MRI 1.8.7 Ruby interpreter. *RubyBOP* includes classes for the interface, multiple heaps for speculative allocation, and access monitoring for objects, arrays, and hashes. We have also parallelized three programs to analyze the cost and scalability of the design. On a multicore, multiprocessor machine we observed three to four times the sequential speed for two of the tests simply by adding a few hints to the sequential scripts.

For the programmer, safe parallel Ruby reduces the complexity of parallel programming. It just suggests parallel tasks and their dependences. There are no concurrent, asynchronous, or out-of-order task finishes for the programmer to worry about. There is no data synchronization needed, no coherence problem (software or hardware), and no deadlock or livelock.

Using a dynamic language, testing is simple and interactive. There is no intermediate step of recompilation and linking. An interpreter stores enough information for run-time reflection. The internal states, for example, the identity of shared objects, can be represented and understood at the source level.

References

[1] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann Publishers, Oct. 2001.

[2] M. H. Cintra and D. R. Llanos. Design space exploration of a software speculative parallelization scheme. *IEEE Transactions on Parallel and Distributed Systems*, 16(6):562–576, 2005.

[3] C. Ding and L. Liu. Access annotation for safe program parallelization. In *Proceedings of the IFIP International Conference on Network and Parallel Computing*, pages 13–26, 2013.

[4] C. Ding, X. Shen, K. Kelsey, C. Tice, R. Huang, and C. Zhang. Software behavior oriented parallelization. In *Proceedings of PLDI*, pages 223–234, 2007.

[5] M. Gupta and R. Nim. Techniques for run-time parallelization of loops. In *Proceedings of SC'98*, page 12, November 1998.

[6] Y. Jiang and X. Shen. Adaptive software speculation for enhancing the cost-efficiency of behavior-oriented parallelization. In *Proceedings of ICPP*, pages 270–278, 2008.

[7] C. Ke, L. Liu, C. Zhang, T. Bai, B. Jacobs, and C. Ding. Safe parallel programming using dynamic dependence hints. In *Proceedings of OOPSLA*, pages 243–258, 2011.

[8] R. B. Osborne. Speculative computation in Multilisp. In *Proceedings of LISP and Functional Programming*, pages 198–208, 1990.

[9] A. Raman, H. Kim, T. R. Mason, T. B. Jablin, and D. I. August. Speculative parallelization using software multi-threaded transactions. In *Proceedings of ASPLOS*, pages 65–76, 2010.

[10] L. Rauchwerger and D. Padua. The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization. In *Proceedings of PLDI*, La Jolla, CA, June 1995.

[11] C. Tian, M. Feng, V. Nagarajan, and R. Gupta. Copy or Discard execution model for speculative parallelization on multicores. In *Proceedings of MICRO*, pages 330–341, 2008.

[12] C. von Praun, L. Ceze, and C. Cascaval. Implicit parallelism with ordered transactions. In *Proceedings of PPoPP*, pages 79–89, Mar. 2007.

[13] A. Welc, S. Jagannathan, and A. L. Hosking. Safe futures for Java. In *Proceedings of OOPSLA*, pages 439–453, 2005.

[14] C. Zhang, C. Ding, X. Gu, K. Kelsey, T. Bai, and X. Feng. Continuous speculative program parallelization in software. In *Proceedings of PPoPP*, pages 335–336, 2010. *poster paper*.