Runtime, Speculative On-Stack Parallelization of For-Loops in Binary Programs

Marwa Yusuf, Ahmed El-Mahdy, Erven Rohou

Runtime, Speculative On-Stack Parallelization of For-Loops in Binary Programs

Marwa Yusuf, Ahmed El-Mahdy and Erven Rohou

Abstract—Nowadays almost every device has parallel architecture, hence parallelization is almost always desirable. However, parallelizing legacy running programs is very challenging. That is due to the fact that usually source code is not available, and runtime parallelization, without program restarting is challenging. Also, detecting parallelizable code is difficult, due to possible dependencies and different execution paths that undecidable statically. Therefore, speculation is a typical approach whereby wrongly parallelized code is detected and rolled back at runtime. This paper considers utilizing processes to implement speculative parallelization using on-stack replacement, allowing for generally simple and portable design where forking a new process enters the speculative state, and killing a faulty process simply performs the roll back operation. While the cost of such operations are high, the approach is promising for cases where the parallel section is long and dependency issues are rare. Also, our proposed system performs speculative parallelization on binary code at runtime, without the need for source code, restarting the program or special hardware support. Initial experiments show about 2x to 3x speedup for speculative execution over serial one, when three fourth of loop iterations are parallelizable. Also, maximum measured speculation overhead over pure parallel execution is 5.8 %.

Index Terms—Compilers, Runtime, Optimization, Parallelization, Binary, Pthreads, On-Stack Replacement, Speculation.

1 Introduction

NOWADAYS almost every computer has a multicore architecture. However, not every software is designed to utilize this feature. From one side, writing a parallel software is not always an easy task. From the other side, there exist legacy software, possibly running, that were designed for older, serial architectures, and the development cycle has a high cost. Automatic parallelization in these cases may be the only solution. However, there are a number of challenges. First, the source code may not be available (like the case of proprietary software) and/or the program may be already running. Second, the program may have possible dependencies and complex execution paths that cannot be decided until runtime.

In this paper, we propose a speculative parallelization technique that parallelizes for-loops in binary programs at runtime. Our paper assumes that the dependence information for the loop is available through the application of a runtime binary analyzer. The analyzer will essentially mark a parallelizable (dependence-free) execution superpath (which is the union of a set of execution paths) in the for-loop, along with exit points from this superpath indicating possible dependence violation that would require rolling back. Based on these information, our mechanism parallelizes the for-loop, but with trampoline jumps added at each exit point. Whenever an exit point is reached, parallel execution stops, state-rolled and the program continues serially. Our approach relies on system processes to implement speculation; fork starts a "backup" version of the current serial loop; the current loop is executed in parallel, and whenever an exit point is reached, the state is recovered by switching to the backup process and killing the parallel process. Moreover, checkpointing is used to decrease the amount of wasted work. This is done through periodic forking of new correct states. The rationale behind this approach is to provide for a simple portable approach. Even though the cost of forking and rolling back is higher, this can be amortized by using more elaborate dependence analysis as well as using loops with rare dependence violations. Moreover, this technique does not require source code information making it directly applicable to binary codes.

Our mechanism is implemented as an extension to the Padrone platform [1]. Also, this work is an extension to our previous parallelization technique [2].

To summarize, this paper introduces the following contributions:

1) Running a binary for-loop iterations in parallel, speculatively, using processes.
2) Selective partial parallelization of the remaining loop iterations.
3) Conducting an initial performance investigation of the proposed mechanism.

The rest of this paper is organized as follows: Section 2 discusses related work. Section 3 provides a brief background about the Padrone system, and the parallelization technique used. Section 4 presents the proposed mechanism design and implementation. Section 5 presents the experiments and discusses the results. Finally, Section 6 concludes the paper and suggests future work.

2 Related Work

Estebanez et al. [3] made a comprehensive survey about speculation. They made a taxonomy of speculation's different techniques and contexts, e.g. hardware vs. software based, control vs. data speculation, and binary vs. source code speculation. They also discussed different design questions needed for any speculation system, e.g. lazy vs. eager version management, and lazy vs. eager conflict detection. Based on these different classifications, our system is a software-based control speculation system for running binary code using processes with eager conflict detection and eager version management.

While most speculation systems use threads, we use processes instead for speculation and use threads only to parallelize within the speculative process. This helps in separating the state of the speculative parallel and the serial processes, without the need for special mechanism for buffering state or for committing or rolling back the state later on. While creating and killing processes are expensive

- *Marwa Yusuf and Ahmed El-Mahdy are with the Department of Computer Science and Engineering, Egypt-Japan University of Science and Technology, Alexandria, Egypt.*
  E-mail: {marwa.yusuf, ahmed.elmahdy}@ejust.edu.eg.
- *Marwa Yusuf is on leave from Benha University, Egypt*
- *Ahmed El-Mahdy is on leave from Alexandria University, Alexandria, Egypt*
- *Erven Rohou is with Inria, France*
  E-mail: erven.rohou@inria.fr.

operations, this cost is compensated by the gain from parallelization and the lack of need to buffer and commit the execution state. Ding et al. [4] introduce a speculation technique based on processes, also. However, their system needs compiler cooperation to insert markers in the code to identify speculative regions of code, to be run in parallel speculatively, and this requires recompiling the program. Our system works on binary code directly at runtime, with no need for source code or even restarting the program. Also, their system runs several processes, with one lead and the others speculative, and they execute different parts of code each, and combine the states at commit. Our system has only one executing process at a time, either parallel one that fully utilizes the underlying architecture, or the serial one. Execution continues from only the right one of them, hence no need for synchronizing state at commit. Synchronization is done implicitly at each checkpoint by the forking of a new process that automatically inherits the current state.

Hertzberg and Olukotun [5] introduce RASP, an automatic speculative parallelization system that uses their dynamic binary translation system, DBT86 [6] to speculatively parallelize running binary code at runtime, without the need for source code or recompilation. However, their system assumes special features needed in the underlying architecture, e.g. the ability to restore the register file to checkpoint state, the ability to buffer speculative state in memory, and the ability to detect dependency violations. In contrast, our system assumes no specific needs from the underlying hardware.

To conclude, our system combines the advantages of runtime automatic speculative parallelization without the need for source code, debugging information, compiler cooperation or execution restarting. Hence, it can be applied directly on a running program. And this is done without the need for special hardware features, which makes it applicable to current available architectures. Also, our system eliminates the need for buffering and synchronizing state through using processes. The cost of managing processes is amortized by the speedup gained from parallelization. Also, using Padrone system, enables doing all the preparation steps (like profiling, analyses and code generation) during target process execution. Only code injection requires stopping the process. Hence, minimal overhead is achieved.

3 Background

Padrone system is a platform that provides an API to create clients for profiling, analyzing and optimizing running binary codes. These clients can be run without the need for either source code, debugging information or restarting the program. Using the `ptrace` system call, Padrone attaches to (and detaches from) the running target process and reads (and writes) its memory registers. The distinctive characteristic of Padrone is that it executes in a different process from its target, hence it minimizes its effect on the target behaviour, while analyses and optimizations can be applied during the target process execution, it needs to be stopped only during new code injection (a mere memcopy). Also, Padrone's code generation does not rely on basic blocks that need to be combined into traces to reduce overhead, as Pin [7] or DynamoRio [8] do. As such, it does not degrade performance of unmodified code.

This work is expanding our previous parallelization mechanism [2]. This previous work introduces a parallelization mechanism of dependence-free for-loops in running binary programs. It works by extracting the considered loop into a separate function into a code cache, (*loopy*). Then it changes the start of the original loop to jump to a previously prepared function (*parallel*). *Parallel* function creates a number of threads, copies the modified stack of the considered loop into the stacks of these threads, and starts them with the extracted loop function *loopy* as the start routine. *Parallel* distributes the remaining loop iterations among threads. After the threads complete their work, the execution continues from after the original loop address. This mechanism needs no source code, no debugging information, no restarting of the program or uplifting it to a higher representation. It works directly on the running binary.

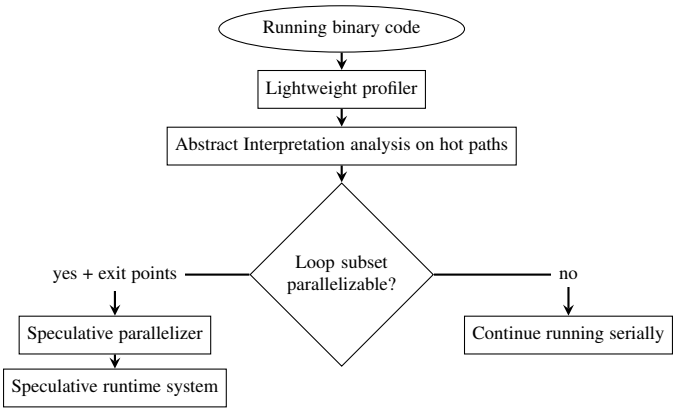

Fig. 1: Full system overview

In this paper, we extend this mechanism to add speculation. The new speculative parallelization mechanism deals with possibly not-parallelizable codes, and adapts at runtime.

4 Proposed Design

This work is a part of a bigger system, that is explained briefly in Section 4.1. However, due to page limit, this paper's scope is the speculative parallelization subsystem, which is explained thoroughly in Section 4.2.[1]

4.1 Full System Design

Fig. 1 shows the flowchart of the full system. A lightweight profiler pass is applied on the input running binary code to determine the hot function and for loop that is candidate for parallelization with the frequent superpath. A superpath is the union of execution paths. Then, an abstract interpretation analysis pass is applied to this hot execution superpath to decide if it is parallelizable or not, based on the dependencies detected in this superpath. However, if the execution exits this superpath, the correctness of parallel execution is not guaranteed. Hence, with the approval of parallelization, the set of possible exit points from this superpath are input to our speculation system in the form of an array passed to the following speculative parallelization step. Each exit point is an address of a conditional jump instruction with the decision that causes exit (whether the exit is the *taken* jump or *notTaken* jump). Our system, using these information, parallelizes the running binary superpath with instrumentation at each exit point. Finally, the instrumented code is run speculatively. If the analysis decision is not to parallelize, the code is left intact to run serially.

4.2 Speculative Parallelization Subsystem Design

Fig. 2 is an overview of the design of our proposed speculation system. The modification to the running code is done in a Padrone client. The client performs the following steps:

1) A code cache is created to accommodate for the following steps' results.
2) The target for-loop code is extracted into a separate function (*loopy*) (Procedure 1), into the code cache. This function is parameterized with the loop boundaries. The prologue of the function prepares a stack that is an extended version of the

1. The details of profiling and analyses steps are beyond the scope of this paper.

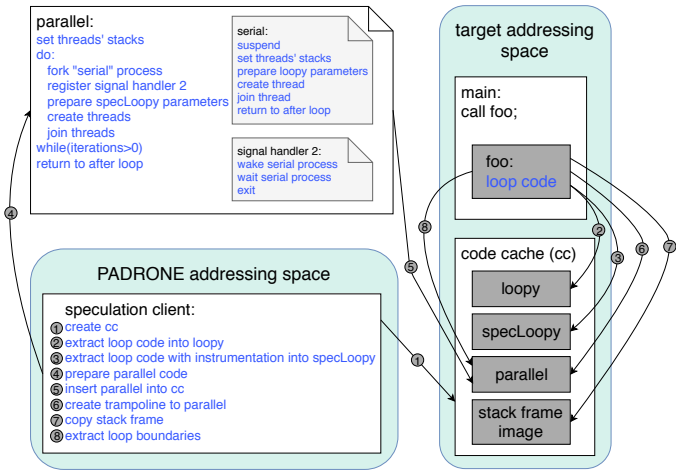

Fig. 2: System Overview

original function stack, to accommodate for the parameters. Then, the loop head is modified to take the function parameters as the initialization and condition of the loop. Also, jumps are modified as needed. Then, only the loop body and back edge are copied to this function, as they are. Only jumps and calls' addresses are modified. Finally, a return is added. This function is used later for serial run, as will be explained shortly.

Procedure 1 loopy function

```
procedure LOOPY(lower, higher)
    for i ← lower, higher do
        loop body
    end for
end procedure
```

3) Another similar but instrumented version of *loopy* is created in the code cache, (*specLoopy*). In this function, each conditional jump instruction that is marked as an exit point is instrumented. The exit could be a *taken* jump or *notTaken* jump. In both cases, the instrumentation code (Procedure 2) is added after the jump instruction. The instrumentation code simply fires an abort signal to the running parallel process. This abort signal causes the parallel process to wake the serial process, waits for it to finish, and then exits. This will be explained in more detail shortly. If the exit is a *taken* jump, the jump is inverted (*greater than* becomes *less than or equal*, for example), and the jump target is modified to the address of the instruction directly after instrumentation code (that follows the jump in the original code).

Procedure 2 instrumentation code

```
procedure INSTRUMENTATION
    pid ← parallel process pid
    sig ← signal number
    call abort(pid, sig)
end procedure
```

4) A previously prepared function code (*parallelSpec*) is compiled (using Padrone special compilation) and inserted into the code cache. This function code is a part of the proposed system. However, its compilation is deferred till the actual run to be fed with the actual runtime addresses of *loopy*, *specLoopy* and other required system functions. *ParallelSpec* takes as parameters both loop remaining iterations boundaries, and the size and contents of the original function stack. If the remaining iterations are so few, *parallelSpec* just runs them serially. Otherwise it starts speculative parallel run. *ParallelSpec* creates phases of parallel execution (using a loop), each executes a chunk of the remaining loop iterations. Each phase start is a check point. At each check point, a new serial process is forked and suspended. This is to guarantee that new serial process has the execution state at this check point. Then, the original parent process (which is now the parallel process) creates a number of threads, populates their stacks with the extended original function's stack, and distributes the selected iterations among them. This distribution is done through the parameter passed to the thread start routine. The start routine is *specLoopy*. The parallel process waits for the threads to finish. If the threads finish normally, then the commit of this phase is done by simply killing the serial process and continuing to the next phase. If, however, an abort signal is received, parallel process stops execution, wakes the serial process, waits for it to finish, then exits. Upon wake up, the serial process starts executing the remaining iterations from the last committed check point till the end serially, by creating only a single thread and calling *loopy* as the start routine. Then it returns. Whether the parallel process continues to the end of iterations or the serial process finishes the last iterations, *parallelSpec* function returns to the instruction address that follows the original loop directly (*afterLoopAddress*). Hence, original program continues after the parallelized loop normally.

5) *ParallelSpec* is inserted into code cache.

6) The start of the original loop is modified to a jump to *ParallelSpec* address in code cache. Hence, in the first coming loop iteration, execution switches to *parallelSpec*.

7) Original function's stack frame is copied to code cache. The address of this copy is copied to right parameter registers, to be passed to *parallelSpec*.

8) Original loop remaining iterations' boundaries are copied also to right parameter registers, to be passed to *parallelSpec*.

Fig. 3 gives an arbitrary execution scenario of the speculative parallel runtime of 2mm kernel. In this scenario, the loop has 1600 iterations and parallelization is started after 12 iterations. The parallelization is done on phases, each phase executes 128 iterations on 4 threads. Phases 1 to 9 are executed in parallel. At phase 10, an exit point is encountered at iteration 1200 in thread 2. This triggers an abort which is performed by waking the serial process and dropping the parallel one. The serial process continues from the last consistent state till the end of the loop. Then execution continues after the loop normally.

4.3 Discussing memory and time costs

Time costs are due to three main operations: process fork, kill, and copy-on-write (COW) which happens at first write on a memory page in the parallel process. Generally, these operations have relatively high cost. However, they are masked by the speedup gained from parallel operation. The only possible case where there is no speedup at all and only overhead is there, is when abort happens from the first parallel execution phase (before first checkpoint). This is very unlikely, due to the profiling and analysis phase. On the other hand, the simplicity of commit and abort operation, which is only to kill and fork, without memory copying or rolling back, minimizes the overhead greatly.

This simplicity comes by the extra cost in memory, due to redundant serial process. However, as the underlying architecture is fit for parallelized version of the target application, having an extra (idle) serial process is most probably not a problem.

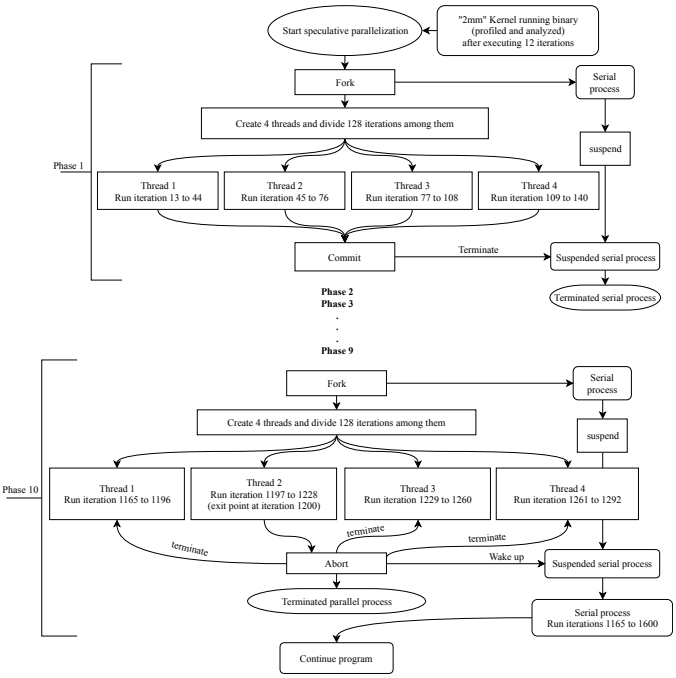

Fig. 3: Example Kernel

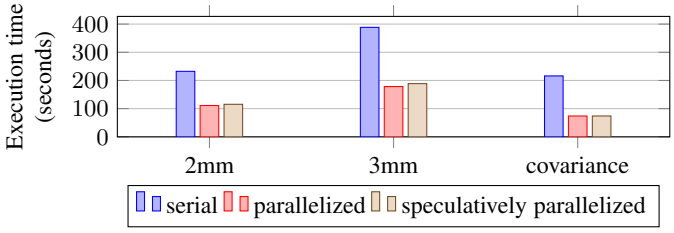

Fig. 4: Performance of our speculative parallelization mechanism (parallelization using 4 threads)

5 Experiments, Results and Analysis

To provide an initial performance study, three benchmarks from the Polybench benchmark suite [9] are used (listed in Table 1, with the number of hot for-loops to parallelize shown for each kernel). Polybench kernels are simple and consist mainly of for-loops. However, to assess speculation, we modified kernels by adding an exit point in each for loop. This forces serial execution of the last quarter of the loop iterations. We run the speculative parallelization mechanism during the execution of the kernel function.

An Optiplex 980 Dell desktop with Intel Core i7 CPU 860 @ 2.80 GHz and 15.6 GiB and running Ubuntu 16.04 is used for experiment. The kernel programs are compiled using gcc 5.4 using -O0 (no optimization). Parallelization is done over 4 threads, to reflect the underlying architecture with four cores.

For each benchmark three runs are performed. First, serial run; the whole loop is run serially. Second, parallel run; the loop is run in parallel except for the last quarter, without any speculation. Finally, speculative run; the whole loop is optimistically run in parallel, until exit point is met, where speculation turns execution into serial. Comparing between speculative and serial runs gives the speculation speedup, while comparing between speculative and parallel runs gives the speculation process overhead. This overhead is mainly due to fork and kill of processes and the time of faulty process squash.

Fig. 4 shows the results. Comparing speculative run to serial run, 2× speedup for 2mm and 3mm benchmarks and 3× speedup for covariance benchmark are achieved. This is for parallelizing only three fourth of the iterations. Comparing speculative run to parallel run, the overhead due to speculation is maximum 5.8%, which is almost negligible.

TABLE 1: Polybench kernels used in the experiments

Benchmark	Description	# loops
2mm	2 Matrix Multiplications (D=A.B; E=C.D)	2
3mm	3 Matrix Multiplications (E=A.B; F=C.D; G=E.F)	3
covariance	Covariance Computation	1

As the number of parallelized loops in an application increases, the speedup is expected to increase accordingly, because larger parts of the application would run in parallel. I.e, the percentage of execution time spent in loops that can benefit from speculative parallelization increases, the whole application performance would enhance greatly.

6 Conclusion and Future Work

In this paper, we presented an initial design of a speculative parallelization runtime system. The system creates a serial process and suspends it at each checkpoint, to save work done so far and continue execution in parallel, until committing at the next checkpoint, or switching to serial if an abort happens. Using processes for separation between serial and parallel executions simplifies managing state of speculative execution. Our system needs no source code, debugging information, compiler or hardware support, or restarting the program. Our system achieves from 2× to 3× speedup on selected Polybench kernels, with maximum 5.8 % speculation overhead.

In future work, we intend to extend the system to be applicable to more kinds of loops. We intend to experiment and analyze its performance with more benchmarks, like SPEC. Also, we intend to integrate with other project modules (profiling and analyses) and experiment the whole system performance. Also, we could monitor how execution goes and adjust parallelization per chunk. For example, the number of threads may vary from chunk to another, or even parallelization could be turned off if not profitable.

Acknowledgments

This research is supported by a Ph.D. scholarship from the Egyptian Ministry of Higher Education (MoHE). Also, this work is partially funded by the PHC IMHOTEP 35236SM project.

References

[1] E. Riou, E. Rohou, P. Clauss, N. Hallou, and A. Ketterlin, "Padrone: a platform for online profiling, analysis, and optimization," in *DCE 2014-International workshop on Dynamic Compilation Everywhere*, 2014.

[2] M. Yusuf, A. El-Mahdy, and E. Rohou, "Runtime on-stack parallelization of dependence-free for-loops in binary programs," *Manuscript submitted for publication.*, 2018.

[3] A. Estebanez, D. R. Llanos, and A. Gonzalez-Escribano, "A survey on thread-level speculation techniques," *ACM Computing Surveys (CSUR)*, vol. 49, no. 2, p. 22, 2016.

[4] C. Ding, X. Shen, K. Kelsey, C. Tice, R. Huang, and C. Zhang, "Software behavior oriented parallelization," in *ACM SIGPLAN Notices*, vol. 42, no. 6. ACM, 2007, pp. 223–234.

[5] B. Hertzberg and K. Olukotun, "Runtime automatic speculative parallelization," in *Code Generation and Optimization (CGO), 2011 9th Annual IEEE/ACM International Symposium on*. IEEE, 2011, pp. 64–73.

[6] ——, "DBT86: A dynamic binary translation research framework for the CMP era," *PESPMA 2009*, p. 41, 2009.

[7] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Conference on Programming Language Design and Implementation*. ACM, 2005.

[8] D. Bruening and S. Amarasinghe, "Efficient, transparent, and comprehensive runtime code manipulation," Ph.D. dissertation, MIT, 2004.

[9] L.-N. Pouchet, "Polybench: The polyhedral benchmark suite," *URL http://web.cse.ohio-state.edu/ pouchet.2/software/polybench*, 2018.