Towards Automatic Binary Runtime Loop De-Parallelization using On-Stack Replacement

Marwa Yusuf, Ahmed El-Mahdy, Erven Rohou

Towards Automatic Binary Runtime Loop De-Parallelization using On-Stack Replacement

Marwa Yusuf[ab*], Ahmed El-Mahdy[ac], Erven Rohou[d]

[a] *Computer Science and Engineering Department, Egypt-Japan University for Science and Technology, Alexandria, Egypt*

[b] *On leave from Benha University, Egypt*

[c] *On leave from Alexandria University, Alexandria, Egypt*

[d] *Univ Rennes, Inria, CNRS, IRISA, France*

Abstract

Runtime compilation has opportunities to parallelize code which are generally not available using static parallelization approaches. However, the parallelized code can possibly slowdown the performance due to unforeseen parallel overheads such as synchronization and speculation support pertaining to the chosen parallelization strategy and the underlying parallel platform. Moreover, with the wide usage of heterogeneous architectures, such choice options become more pronounced. In this paper, we consider an adaptive form of the parallelization operation, for the first time. We propose a method for performing on-stack de-parallelization for a parallelized binary loop at runtime, thereby allowing for rapid loop replacement with a more optimized one. For this paper, we consider a loop parallelization strategy and propose a corresponding de-parallelization method. The method relies on stopping the execution at safe points, gathering threads' states, producing a corresponding serial code, and continuing execution serially. The decision to de-parallelize or not is taken based on the anticipated speedup. To assess the extent of our approach, we have conducted an initial study on a small set of programs with various parallelization overheads. Results show up to 4× performance improvement for a synchronization intense program on a 4-core Intel processor.

[*] Corresponding author

Email addresses: marwa.yusuf@ejust.edu.eg (Marwa Yusuf[ab]), ahmed.elmahdy@ejust.edu.eg (Ahmed El-Mahdy[ac]), erven.rohou@inria.fr (Erven Rohou[d])

Keywords: compilers, parallelization, optimization, on-stack replacement, binary

1. Introduction

Parallel execution is the default nowadays, and hence automatic parallelization. Moreover, automatic parallelization at runtime provides better optimization opportunity than static parallelization, as more information is available at runtime. However, there are situations where the parallelized program may prove—after starting execution—to be slower than the serial one or at least can be parallelized better. These situations mainly arise due to unpredictable execution behaviour that may result in excessive synchronization operations, false-sharing, changes in the underlying system load (or resources in case of dynamic infrastructures), thereby hurting parallel performance, yielding better performance for the original serial program [4].

In this paper, we consider on-stack replacement (OSR) as a potential solution for the above problem. Typically OSR allows replacing a serially executing function with a more optimized version without requiring returning from the function [5], as well as deoptimizing a speculatively optimized function when speculation assumption fails. Existing OSR techniques rely on the compiler cooperation to insert marked points in code to activate OSR. In our proposed design, OSR is applied directly on running binary parallel code at an arbitrary execution time, and replaces it with a serial one, without the need for compiler cooperation or recompilation. Later on, this generated serial binary can be re-parallelized better to be more suitable to the environment. This allows adaptive parallelization optimizations.

The rest of this paper is organized as follows: Section 2 discusses related work. Section 3 explains the proposed design. Section 4 conducts an initial study on a small set of programs assessing the extent of our method. Finally, Section 5 concludes and suggests possible future work.

2. Related Work

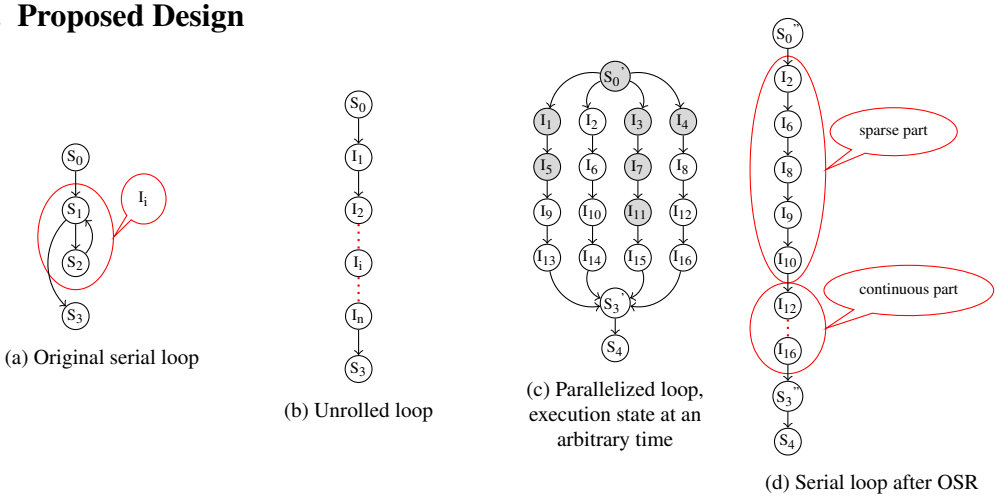
OSR is typically used to switch from a running function to another dynamically or statically generated function. This usually requires extracting the stack state from the old function and injecting it with the proper mapping into the new function, either by generating prologue code that creates the new stack or by creating the new stack directly.

OSR is used for dynamic optimization [1, 5, 9, 10, 8, 11]. Also, it is used for dynamic deoptimization [6, 12, 3]. Another usage of OSR is for obfuscation [13]. Some work are made to introduce frameworks with well defined APIs for OSR [7, 2].

All of the above related work considered performing OSR using compiler co-operation to insert instrumentation points in code to activate OSR and to compile alternative functions either statically or dynamically. In our proposed technique, no compiler cooperation is needed, as OSR is applied on binary code at runtime, and the dynamically generated code is binary.

Also, as far as we know, all other related work deoptimize from where serial state is available. In our proposed work, the running function is de-parallelized at an arbitrary point of execution, with the extra challenge of gathering parallel state from multiple threads, and serialising the execution accordingly.

3. Proposed Design



(a) Original serial loop

(b) Unrolled loop

(c) Parallelized loop, execution state at an arbitrary time

(d) Serial loop after OSR

Figure 1: An example on deparallelization process

Figure 1 is an example to explain how de-parallelization is done. The example is an arbitrary loop whose sequence is shown in Figure 1a, where 'S' is a block of statements and 'I' is an iteration. Figure 1b represents the real sequence of execution of loop iterations. Figure 1c is an example of a parallelized loop of 16 iterations distributed over 4 threads in round robin way. This figure represents the program at an arbitrary instant of time, where gray nodes are already executed and white ones are not. If at this point the OSR de-parallelization decision is taken, the program is paused, its state is gathered, the remaining iterations from each thread

are collected and sewed together in their original serial order to produce a serial code. Figure 1d shows the resulting execution after de-parallelization.

Figure 2 shows our proposed design. Our system is designed as a client that attaches to the target running program at runtime. It starts by calling an external runtime binary profiler and analyzer that detects that the program has a parallel for-loop, and the performance is expected to be better if the rest of the loop is executed in serial. An extended, more detailed model of the model in [5] is used to decide de-parallelization:

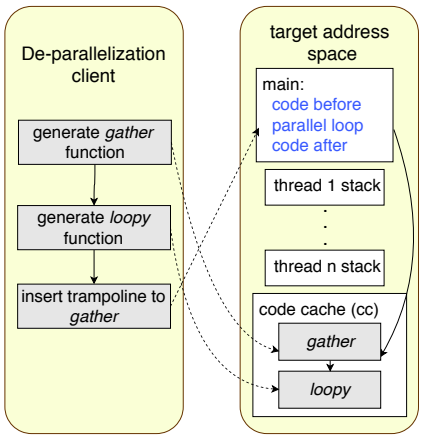


Figure 2: System Overview

$$T_{\text{de-parallel}} = T_{\text{OSRcompile}} + T_{\text{remainingSerial}} + T_{\text{runtimeOverhead}}$$

where the terms are defined as follows:

- $T_{\text{OSRcompile}}$: OSR processing time that includes state gathering and generating de-parallelized code; this is a pre-set constant based on previous compiler behaviour relative to the size of the stack and the number of threads of the given program;
- $T_{\text{remainingSerial}}$: execution time of the remaining program after de-parallelization; this is expected by the profiler after calculating the execution time of one iteration without blocking for other iterations and multiplying this time by the number of remaining iterations;
- $T_{\text{runtimeOverhead}}$: overhead during runtime due to extra checks and jumps at already executed iterations; this part is resulting from the sparse part of iterations at the start of executing the remaining iterations (will be clarified shortly), and it is computed based on the extent of sparsity of the remaining iterations;

Also, define T_{parallel} to be the remaining parallel execution time; expected by the profiler (based on profiling the executed iterations so far and given the number

of remaining iterations), then de-parallelization decision is taken if $T_{\text{de-parallel}} < T_{\text{parallel}}$, i.e, if the estimated remaining program time is longer than the OSR process time + the estimated remaining serialized program time (including the runtime overhead) then de-parallelization is done. Otherwise, the program continues as is.

If de-parallelization is decided, the running program is paused and the client creates a code cache into the target address space to accommodate for the generated code. Then, the state is captured by the generated *gather* function, the code is de-parallelized into the generated *loopy* function and execution continues. Details of these client steps are as follows:

1. A code cache is created in the target address space.
2. The de-parallelization process is started at the execution of next iteration after OSR decision is fired. This is done by replacing the first instruction of the loop with code that checks the thread id of each thread executing this code, to pause them all except for one thread, in which case it continues to a trampoline that jumps to the address of the code cache start, which is the address of *gather* function.
3. *Gather* function is generated, which gathers the threads' status from threads' stacks and combine them in a single unified stack. Threads' status includes shared variables and private variables. Shared variables are synchronized by nature, so no special action is needed for them. Private variables' values are taken from the latest executing iteration. The resulting stack map is used later in generating *loopy* function; Specifically, the iteration index value for each thread is used in generating the prologue of *loopy* function, as will be explained shortly.
4. *Loopy* function is generated, which has two parts; the prologue that is responsible for executing the part of iterations that is sparse, and the second part which executes the rest of the loop. The prologue works as shown in procedure 1, lines 2-16. The remaining iterations to be executed are inspected to find the areas where iterations' execution is sparse, i.e executed and un-executed iterations are mixed. The code runs from the minimum iteration to the maximum one in the mentioned area (iterations 2 to 10 in the example). A loop is created to iterate over all iterations in a level by level of threads. Each thread is checked for remaining iteration to be executed in this level. As an optimization, if a thread has no iterations to be executed in the sparse area (like third thread), no condition is created for it. After finishing all iterations in the sparse area, a normal loop is created to execute

the rest of iterations normally, without extra checks. Constraining the special code on the sparse area only allows for better later parallelization to be applied on the rest of the loop and makes the technique applicable on both for and while loops. This code can be optimized later on and parallelized if needed. As the same sequential execution order, defined by original serial code, is maintained; the correctness of the program and that no deadlock would happen are guaranteed. Of course, preserving this order comes at a cost (more checks for the already executed parts). This cost changes with how much remaining iterations are sparse. However, this cost would be calculated during the OSR decision step, hence it would be amortized. The second part (lines 30-35) of the function is just the original loop, executing the rest (non-sparse) of the iterations to the end. Finally, the function ends by returning to the instruction address that follows the loop in the original function code.

Procedure 1 *loopy* function

```
 1: procedure LOOPY
 2:     global_i ← smallest remaining iteration No.
 3:     max_i ← biggest remaining iteration No.
 4:     step ← No. of threads
 5:     threads_i[step] ← smallest iteration No. in each thread
 6:     th ← thread No. with the smallest remaining iteration No.
 7:     while global_i ≤ max_i do
 8:         for j ← 1, step do
 9:             if threads_i[th] ≤ max_i then
10:                 do iteration threads_i[th] work
11:                 threads_i[th] ← threads_i[th] + step
12:             end if
13:             th ← (th + 1) mod step
14:             global_i ← global_i + 1
15:         end for
16:     end while
17:     i ← next iteration No.
18:     while i ≤ max do
19:         do iteration work
20:         i ← i + 1
21:     end while
22: end procedure
```

It is possible to inject a barrier at the end of the sparse area and continue executing in parallel till this point and them perform OSR to continue serially. This

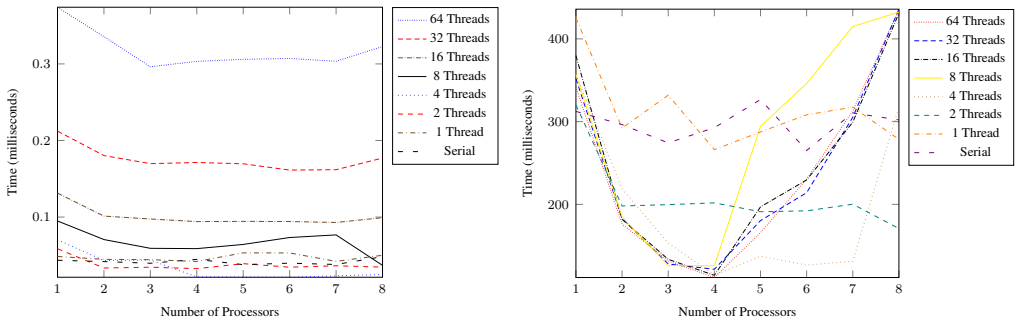
simplifies the problem greatly. However, it may happen that one thread is very advanced in execution (may be finished), which results in parallel execution of the whole loop, which was decided that it performs badly. Thus, in this proposed work, we choose to perform OSR immediately.

4. Case Study

In this section we take two example programs to show how the serial code may be better in performance than the parallel one. Section 4.1 inspects the effect of changing input size, while Section 4.2 inspects the case of a program with heavy synchronization overheads. For both cases, OpenMP is used for parallelization and the experiments are run on Intel Core i7-2670QM CPU @ 2.20 GHz (4-cores).

4.1. Input Size

In this example, we use simple matrix multiplication C++ program. Matrix multiplication is a good example to choose parallelization parameters, like threads granularity, and observe thread parallelization (scheduling) overheads with changing the input size. This case may happen when executing a binary parallel program on different input sizes, where there is no option to modify the program.



(a) 32×32 matrix multiplication

(b) 512×512 matrix multiplication

Figure 3: Matrix multiplication with different matrix sizes

Figure 3a shows the execution time (in milliseconds) of 32×32 matrix multiplication on different number of processors using different number of threads, while Figure 3b shows the same for 512×512 matrix. As it appears clearly from the figures, as the size of the matrix gets smaller, the parallelization tends to actually hurt performance, due to parallelization overheads, while the parallelization overheads are compensated with the amount of work needed for calculation in case of larger matrix.

4.2. Synchronization Overheads

In this example, a simple summation C++ program that use locks heavily is used. The summation variable is shared among threads and is protected by a lock. The first half of the program is run in parallel, while the other half is sequential. As it appears from Figure 4, the sequential part is more than 4 times faster than the parallel one.

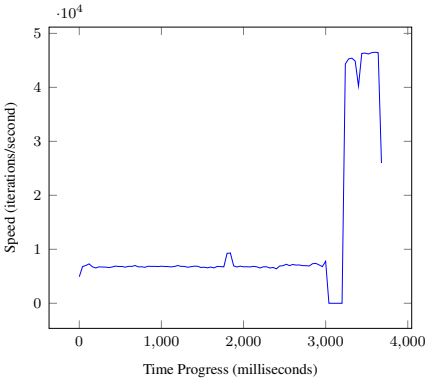


Figure 4: Speed of parallel and serialized over time

5. Conclusion and Future Work

In this paper, we consider, for the first time (as much as we know), using OSR in parallel binary code, without compiler instrumentation, to address the issue of poor performance that may arise during runtime due to unpredicted overheads or environment changes. In this initial study, we target simple loops. We experimented with a set of a corresponding microbenchmarks; The experiments showed a $4\times$ performance improvement at heavy synchronized code case. We proposed an initial design of the technique, and the algorithm used for serialized code generation.

As a future work, the design will be implemented, tested and evaluated on different use cases and different architectures. The technique can be extended to cover more general types of loops as well as recursive kernels. A more detailed proof of correctness will be considered.

Acknowledgement

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 project.

References

[1] Chambers, C., Ungar, D., 1991. Making pure object-oriented languages practical. In: ACM SIGPLAN Notices. Vol. 26. ACM, pp. 1–15.

[2] D'Elia, D. C., Demetrescu, C., 2016. Flexible on-stack replacement in llvm. In: Proceedings of the 2016 International Symposium on Code Generation and Optimization. ACM, pp. 250–260.

[3] Duboscq, G., Würthinger, T., Stadler, L., Wimmer, C., Simon, D., Mössenböck, H., 2013. An intermediate representation for speculative optimizations in a dynamic compiler. In: Proceedings of the 7th ACM workshop on Virtual machines and intermediate languages. ACM, pp. 1–10.

[4] Eyerman, S., Du Bois, K., Eeckhout, L., 2012. Speedup stacks: Identifying scaling bottlenecks in multi-threaded applications. In: Performance Analysis of Systems and Software (ISPASS), 2012 IEEE International Symposium on. IEEE, pp. 145–155.

[5] Fink, S. J., Qian, F., 2003. Design, implementation and evaluation of adaptive recompilation with on-stack replacement. In: Proc. of the int. symp. on CGO: feedback-directed and runtime optimization. IEEE Computer Society, pp. 241–252.

[6] Hölzle, U., Chambers, C., Ungar, D., 1992. Debugging optimized code with dynamic deoptimization. In: ACM Sigplan Notices. Vol. 27. ACM, pp. 32–43.

[7] Lameed, N. A., Hendren, L. J., 2013. A modular approach to on-stack replacement in LLVM. In: ACM SIGPLAN Notices. Vol. 48. ACM, pp. 143–154.

[8] Paleczny, M., Vick, C., Click, C., 2001. The Java hotspot TM server compiler. In: Proc. of the 2001 Symp. on JVM Research and Technology Symp.-Volume 1. USENIX Association, pp. 1–1.

[9] Soman, S., Krintz, C., 2006. Efficient and General On-Stack Replacement for Aggressive Program Specialization. In: Software Engineering Research and Practice. Citeseer, pp. 925–932.

[10] Suganuma, T., Yasue, T., Nakatani, T., 2003. A region-based compilation technique for a Java just-in-time compiler. In: ACM SIGPLAN Notices. Vol. 38. ACM, pp. 312–323.

[11] Süsskraut, M., Knauth, T., Weigert, S., Schiffel, U., Meinhold, M., Fetzer, C., 2010. Prospect: A compiler framework for speculative parallelization. In: Proc. of the 8th annual IEEE/ACM int. symp. on CGO. ACM, pp. 131–140.

[12] Würthinger, T., Wimmer, C., Mössenböck, H., 2009. Array bounds check elimination in the context of deoptimization. Science of Computer Programming 74 (5), 279.

[13] Yusuf, M., El-Mahdy, A., Rohou, E., 2013. On-stack replacement to improve JIT-based obfuscation a preliminary study. In: JEC-ECC, 2013 Japan-Egypt Int. Conf. on. IEEE, pp. 94–99.