Runtime On-Stack Parallelization of Dependence-Free For-Loops in Binary Programs

Marwa Yusuf, Ahmed El-Mahdy and Erven Rohou

Abstract—With the multicore trend, the need for automatic parallelization is more pronounced, especially for legacy and proprietary code where no source code is available and/or the code is already running and restarting is not an option. In this paper, we engineer a mechanism for transforming at runtime a frequent for-loop with no data dependencies in a binary program into a parallel loop, using on-stack replacement. With our mechanism, there is no need for source code, debugging information or restarting the program. Also, the mechanism needs no static instrumentation or information. The mechanism is implemented using the Padrone binary modification system and pthreads, where the remaining iterations of the loop are executed in parallel. The mechanism keeps the running program state by extracting the targeted loop into a separate function and copying the current stack frame into the corresponding frames of the created threads. Initial study is conducted on a set of kernels from the Polybench workload. Experiments results show from 2× to 3.5× speedup from sequential to parallelized code on four cores, which is similar to source code level parallelization.

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

1 Introduction

While multicore processors are the current trend, a lot of sequential software still exist. Hence, parallelization is highly demanded. However, parallelization is a challenging task. While many approaches have considered parallelizing programs, most of them assume the availability of source code, where higher level of semantics exists, they introduce the burden of decompiling binary into some intermediate format to analyze, transform and recompile or at least they require restarting the program. Parallelization at the binary level directly at runtime without restarting has the advantage of being more general, allowing for speeding up legacy long-running code (specially real-time applications where the cost of restarting is high), with minimal time overhead. However, transformation of loops into parallel form is much challenging due to low level of semantics available in such programs. Also, transformation at runtime requires a way to manage execution state.

In this paper, we introduce a novel mechanism for automatic parallelization transformation of for-loops with no loop-carried dependencies[1] in binary running programs at runtime. For-loops are very common in a lot of applications, especially scientific applications. Runtime parallelization, rather than static parallelization, makes use of the actual runtime information, like the underlying architecture, the available memory and processors, the runtime variables' values that affect conditions, etc. This information enables more realistic and effective parallelization without the need for too much conservativeness that is needed in case of static parallelization. Our mechanism parallelizes binary directly with no need for source code. This enables the parallelization of legacy code, proprietary code and when the cost of development cycle is too high.

- *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.
- *Ahmed El-Mahdy is on leave from Alexandria University, Alexandria, Egypt*
- *Marwa Yusuf is on leave from Benha University, Egypt*
- *Erven Rohou is with Univ Rennes, INRIA, CNRS, IRISA (France). E-mail: erven.rohou@inria.fr.*

1. Dependence analyses are out of the scope of this paper. Hence, we assume that the dependence analyses for such a loop is known at runtime. Examples for this situation is when an earlier binary analyses phase is conducted and the loops are proven to have no dependencies.

Our mechanism is implemented as an extension to the Padrone platform [1]. Padrone works by attaching itself to the targeted program and then profiles and analyzes the program. Our mechanism then parallelizes the remaining loop iterations using pthreads, and then continues execution after the loop normally. The main advantages of this mechanism are that it applies directly to running binary code. This means that there is no need for source code, debugging information, raising the code to a higher level, or restarting the program. Also, it assumes neither a specific parallel architecture (like GPU for example) nor a specific parallel application.

In summary, our main contributions are:

1) Extracting and parallelizing a general for-loop with no loop carried dependence at runtime to a separate parameterized function that accepts loop boundaries as parameters.
2) Allowing for parallelizing the remaining iterations of a for-loop through on-stack replacement of the serial function with the parallel one.
3) Conducting an initial performance investigation of the proposed mechanism.

The rest of this paper is organized as follows: Section 2 provides a brief background about the Padrone system. Section 3 explains the introduced design. Section 4 conducts an initial study on a small set of Polybench kernels assessing the applicability and performance of our mechanism. Section 5 discusses related work. Finally, Section 6 concludes and suggests possible future work.

2 Background

Binary rewriting is a technique that helps in dealing with executable directly without the need for source code. This is specially important in case of legacy and proprietary software. There are a number of tools which use binary rewriting for different purposes, like profiling, analyzing, optimizing or obfuscating existing binaries [2], [3], [4]. Padrone [1] is a platform to profile, analyze and optimize running binary codes without the need for either source code, debugging information or restarting the program. As Figure 1 shows, the platform provides an API to create clients (represented by *user code* section) that perform any of the mentioned tasks. Padrone achieves very lightweight detection of hotspots thanks to the use of the hardware performance monitoring unit (PMU). Contrarily to many other dynamic

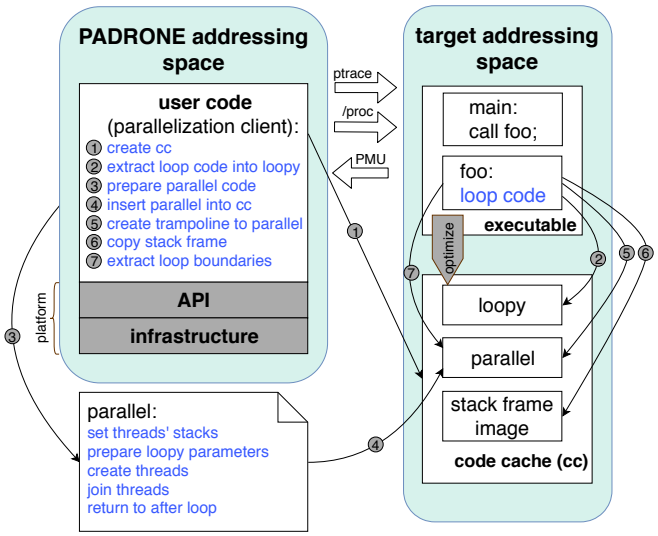

Fig. 1: System Overview (The items in "user code" block explain the action arrows

binary optimizers, Padrone is not linked to its target and does not share its address space. Instead, Padrone executes in a different process from its target, and interacts through Linux mechanisms. Attaching and detaching are done with the `ptrace` system call, as well as reading/writing target registers. The memory is accessed through the `/proc/PID/mem` pseudo-filesystem. As opposed to popular tools such as Pin or DynamoRIO which generate traces for all executed code, Padrone executes most of the target code in-place, and relies on a code cache only for modified code (instrumented or optimized), thus favoring performance and transparency, i.e. minimizing the impact on the behavior of the target.

3 Proposed Design

Figure 1 provides an overview of our proposed system. Our system is a parallelization client that is implemented using Padrone API and runs within its own addressing space, in addition to some new functions added to Padrone API. The input to our system is a running binary program (target addressing space) along with profiling and analyses information. Profiling and analyses determine the currently executing hot function containing a for-loop candidate for parallelization (with no carried dependencies). These information can be obtained from an external dynamic binary profiler and analyzer, like Prospector [5][2]. After the above preprocessing, the parallelization client starts the parallelization process (without interrupting the running target binary program) by creating a code cache, in the target address space. The client then promotes the selected for-loop as a function (a tranformation named *outlining*), prepares parallelization functions, and inserts them into the code cache. Finally, the client traps the loop (at an iteration's start), extends the stack frame and copies it into code cache, and inserts a jump to the parallelization code. The following subsections explain the loop outlining, parallelization function preparation, and stack copy operation in detail.

3.1 Loop Outlining

The client considers only regularly structured for-loops. It extracts the loop iterator's stack location, the loop head parts (initialization, condition and increment), and the loop body code. It generates a corresponding new function, (*loopy*, Proc. 1), and stores it into the code cache. *Loopy* function starts by a prologue that accepts as a parameter a `struct` (*para*) that contains the loop boundaries, and sets the loop head accordingly, i.e. the initialization and condition of the loop are set to *start* and *end* fields of *para* struct, respectively.

Following the prologue, *loopy* function has the loop body with the increment and back edge. The original back edge is modified to jump to the new loop head. Padrone takes care of adjusting global addresses relative to the code cache address.

Procedure 1 Oulined Function Structure (*loopy*)

```
procedure LOOPY(para)
    for i ← para.start to para.end do
        loop body
    end for
end procedure
```

3.2 Parallelization Code

The parallelization code itself (Proc. 2) is a code file containing a special function (*parallel*). This function takes the remaining iterations' boundaries (*lower*, and *higher*) and the address of the stack frame data in the code cache (explained more in the following section) as parameters. The *parallel* function creates a number of threads as needed, and set stacks for these threads by copying the extended stack frame from code cache into them. Also, it creates an equal number of *para* parameters that contain loop boundaries to send to *loopy* function. The remaining iterations are divided between threads. The return address of this function is set to the address of the instruction directly following the original loop, *afterLoopAddress*. Then, this file is compiled at runtime by the Padrone system, providing all runtime actual addresses of the targeted function, *afterLoopAddress*; and any required library functions. The resulting object file is inserted into the code cache. A long jump instruction is inserted at the start of the original loop to the address of *parallel* function in the code cache. Thus, at the start of the next loop iteration, the execution transfers to *parallel* function, and after completion, execution returns to *afterLoopAddress* instruction.

Procedure 2 Parallelization Code *parallel*

```
procedure PARALLEL(lower, higher, stackFrame)
    share ← ( higher - lower ) / numOfThreads
    for i ← 1 to numOfThreads do
        para.start ← lower
        para.end ← lower + share
        threadStack ← stackFrame
        call createThread ( Loopy, para )
        lower ← lower + share + 1
    end for
    for i ← 1 to numOfThreads do
        call joinThread
    end for
    return to afterLoopAddress
end procedure
```

3.3 Stack Frame

The targeted loop is not necessarily the only part of the function, i.e. the function may contain work before and after the loop. Our mechanism only parallelizes the targeted loop, and leaves the remaining function code executing intact. That is done by copying the function's stack frame at the start of the loop to the code cache. Commonly, the stack is defined by current stack and base pointers' values. The mechanism is to copy the memory area between these two addresses. There is no need to recover the stack map as loop body code is not changed, and all loop local variables are private[3]. We may copy

2. Due to the manuscript page-limit, profiling and dependency analyses are out of scope of this paper. Hence, we rely on external tools.

3. Currently, we are only considering local variables to be allocated into the stack; however, future work, can include register allocated variables, by simply copying the whole register set in the corresponding function.

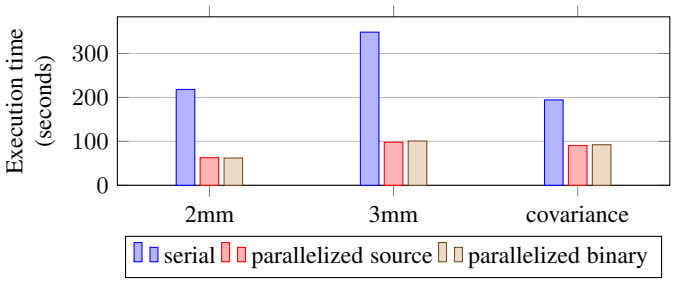

Fig. 2: Parallelization Mechanism Performance (using 4 threads)

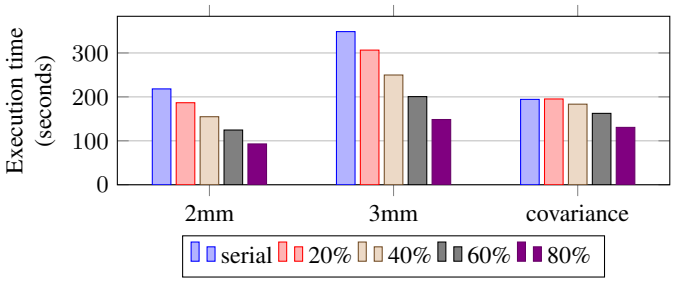

Fig. 3: Speedup with Different Amount of Iterations Parallelized

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

unused stack entries, however, this does not affect correctness.

The stack frame in code cache is extended to accommodate for the *loopy* parameter (*para*). Then, this extended stack frame is copied into individual stacks of every created thread. Hence, each thread executing *loopy* function finds an image of the stack with all the original function variables plus the parameter *para*. Also, *afterLoopAddress* is passed to the parallelization code, to be used as the return address. This way, execution continues after the parallelized loop normally.

4 Experiments, Results and Analysis

To present initial performance results, we consider three kernels from the Polybench benchmark suite [6]. Polybench contains a number of benchmarks that are simple and have for-loops with no loop carried dependencies; and each benchmark is contained in one file that is easy to tune during compilation for the experiment. The kernels used are *2mm*, *3mm* and *covariance*, listed in Table 1, with the number of hot for-loops to parallelize listed for each kernel. The parallelization client is run during the execution of each loop within each kernel function. The reported execution times represent only the execution time of the kernel function. The experiments are done on an Optiplex 980 Dell desktop with Intel Core i7 CPU 860 @ 2.80 GHz and 15.6 GiB and running Ubuntu 16.04. The kernel programs are compiled using gcc 5.4 using -O0 (no optimization).

In Figure 2 we measure (for each benchmark) the execution time for the serial kernel function, and for the parallelized kernel function with four threads both when the source code is parallelized and when the binary code is parallelized using our mechanism. We consider four threads to reflect the underlying architecture with four cores. The figure shows the speedup of the three selected kernels over the corresponding serial versions and parallelizing almost the

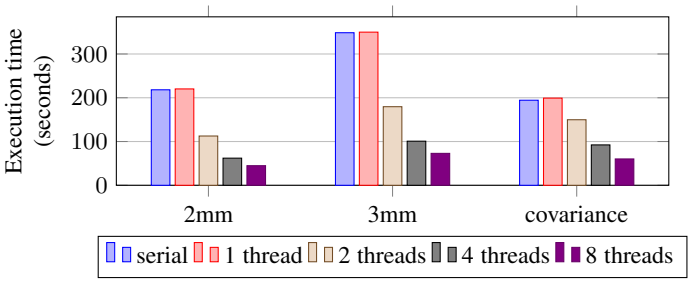

Fig. 4: Speedup with Different Numbers of Execution Threads

TABLE 2: Mechanism Overhead when Parallelizing with 1 Thread

kernel	2mm	3mm	covariance
overhead (second)	0.9	0.4	2.5

whole program (above 98 % of iterations). The results show about 3.5× speedup for *2mm* and *3mm* kernels, and about 2× speedup for *covariance* kernel. The lower speedup in case of *covariance* kernel is due to load imbalance due to the nature of the loop code (iterations differ in the amount of work, where the first iteration performs much more work than the last one). The measured time includes the parallelization process itself (overhead), the profiling time and the small number of iterations that are executed serially before applying parallelization. The same figure also reports the performance of kernels parallelized by the compiler from source code. It shows that our mechanism's performance is almost identical to source code parallelization performance.

In Figure 3 we show the execution time when parallelizing 20 %, 40 %, 60 % and 80 % of iterations, respectively, to assess how much our mechanism may reduce execution time according to the amount of work remaining in the program. As the speedup concluded from Figure 2 for *2mm* kernel is almost perfect, the linear behaviour here is expected. However, in case of *covariance* kernel, speedup is not achieved with small amount of iterations parallelized. Also, this figure accounts for the overhead of any processing (profiling and analysis) before actual parallelization, as it shows that there is considerable speedup gained even when only small part of the iterations is parallelized.

In Figure 4 we show the execution time when parallelizing with different number of threads. The performance of one thread parallelization, compared to serial execution, reflects the overhead of the parallelization process, as it includes only the parallelization client work (profiling, outlining loop, injecting parallelization code, etc.) and fork-join of one thread, without any gain from parallelization. As shown in the figure, the overhead is almost negligible. Table 2 lists this overhead for each kernel. The performance almost doubles from one to two threads and from two to four threads in case of *2mm* and *3mm* kernels. This behaviour changes from four to eight threads, due to the fact that there are only four cores, hence contention starts to affect speedup gained.

5 Related Work

In our mechanism, we perform binary code parallelization at runtime and we keep the running program state by copying the stack frame. As much as we know, this is the first mechanism that can apply parallelization on binaries at runtime, automatically without any static information, or hardware support and without the need to restart the running program.

5.1 Binary parallelization

Most of automatic binary parallelization systems work by rewriting binaries statically [7], [8]. Another set of approaches recompile

and instrument binaries statically which requires restarting the program [9]. In our work we rewrite binaries dynamically at runtime, hence allowing for optimizing long running applications, without the need for restart. Some techniques require uplifting binary into intermediate representation (IR) to perform analysis and preparations for parallelization (and possibly optimization itself) statically [10], [11], [12]. While this simplifies the transformation process, it adds extra time overhead for uplifting process. In our work, optimization is applied on the binary directly. Another kind of approaches parallelize at runtime with the help of compile-time generated information [13], [14]. Our mechanism does not need any compiler cooperation, hence, it is applicable for executables where source code is not available. Also, there are hardware supported approaches [15]. Our mechanism needs neither hardware support, nor specific features of hardware.

The most similar approach to our work is that of Nakamura et al. [16], where they have introduced an automatic vectorization system that operates on binaries at runtime. However, it works only on three specific kinds of programs; linear search, loops that have super word level parallelism and data level parallelism. Whenever the system identifies one of these cases during the analysis phase, a special function is generated using extracted data from targeted binary code, and a long jump is inserted into the start of the targeted function that jumps to the newly generated function. As the jump is at the start of the function, they need to wait for another call to the function, which may never happen in some programs, hence no optimization is gained. In our mechanism, the jump is inserted at the start of the loop. This is more challenging as it requires keeping the function stack state so executing the parallelized loop does not damage the original function execution, but it adds the benefit of parallelizing during function execution, which is crucial in case of long running functions, and also in case that this long function is called only once.

5.2 OSR

OSR has been used for different usages: deferred compilation and/or optimization [17], [18], [19], [20], [21], deoptimizing code for debugging [22], for speculation rollback [23], [24] or for obfuscation [25]. However all these techniques assume the availability of the source code and/or some compile time preparations. Our mechanism performs OSR on binary code at runtime, without any static preparations. Also, in contrast to most techniques, our mechanism transfers the stack state to several threads dynamically, and execution transfers to theses threads during function execution, not at the next call.

6 Conclusion and Future Work

In this paper we presented a mechanism that parallelizes for-loops with no dependencies dynamically at runtime. Our mechanism does not need source code, debugging information or restarting the program. It parallelizes binary directly with no uplifting to higher representation, minimizing overhead. The speedup measured for the selected kernels is about $2\times$ to $3.5\times$ on four cores. Also, the speedup is almost identical to that achieved by source-level parallelization with negligible overhead. The mechanism uses OSR for managing the state of the parallelized loop with no need to restart the targeted function.

As a future work, we will generalize the mechanism more to work on other types of loops and optimization levels and experiment with more benchmark kernels. Also, we will integrate the mechanism into a full system that profiles and analyzes binary on the run. At a later step, we will add speculation to account for the code that contains possible dependencies.

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 *International Workshop on Dynamic Compilation Everywhere (DCE)*, 2014.

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

[3] 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.

[4] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," in *ACM Sigplan notices*, vol. 42, no. 6. ACM, 2007.

[5] M. Kim, H. Kim, and C.-K. Luk, "Prospector: Discovering parallelism via dynamic data-dependence profiling," in *USENIX Workshop on Hot Topics in Parallelism (HOTPAR)*, vol. 10, 2010.

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

[7] A. Kotha, K. Anand, M. Smithson, G. Yellareddy, and R. Barua, "Automatic parallelization in a binary rewriter," in *International Symposium on Microarchitecture (MICRO)*. IEEE Computer Society, 2010.

[8] B. Pradelle, A. Ketterlin, and P. Clauss, "Polyhedral parallelization of binary code," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 8, no. 4, 2012.

[9] R. Zhou, "Guided automatic binary parallelisation," Ph.D. dissertation, University of Cambridge, 2018.

[10] E. Yardımcı and M. Franz, "Dynamic parallelization and mapping of binary executables on hierarchical platforms," in *Conference on Computing Frontiers*. ACM, 2006.

[11] N. Hallou, E. Rohou, and P. Clauss, "Runtime vectorization transformations of binary code," *International Journal of Parallel Programming*, vol. 45, no. 6, 2017.

[12] D.-Y. Hong, Y.-P. Liu, S.-Y. Fu, J.-J. Wu, and W.-C. Hsu, "Improving simd parallelism via dynamic binary translation," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 17, no. 3, 2018.

[13] A. Jimborean, P. Clauss, J.-F. Dollinger, V. Loechner, and J. M. Martinez Caamaño, "Dynamic and speculative polyhedral parallelization using compiler-generated skeletons," *International Journal of Parallel Programming*, vol. 42, no. 4, 2014.

[14] J. M. M. Caamaño, A. Sukumaran-Rajam, A. Baloian, M. Selva, and P. Clauss, "APOLLO: Automatic speculative polyhedral loop optimizer," in *International Workshop on Polyhedral Compilation Techniques (IMPACT)*, 2017.

[15] J. Yang, K. Skadron, M. L. Soffa, and K. Whitehouse, "Feasibility of dynamic binary parallelization," in *USENIX conference on Hot Topics in Parallelism*, 2011.

[16] T. Nakamura, S. Miki, and S. Oikawa, "Automatic vectorization by runtime binary translation," in *International Conference on Networking and Computing (ICNC)*. IEEE, 2011.

[17] M. Paleczny, C. Vick, and C. Click, "The Java Hotspot™ server compiler," in *Symp. on JVM Research and Technology Symp.*, 2001.

[18] S. Soman and C. Krintz, "Efficient and General On-Stack Replacement for Aggressive Program Specialization." in *Software Engineering Research and Practice*, 2006.

[19] C. Chambers and D. Ungar, "Making pure object-oriented languages practical," in *Conference on Object-oriented Programming Systems, Languages, and Applications*, 1991.

[20] S. J. Fink and F. Qian, "Design, implementation and evaluation of adaptive recompilation with on-stack replacement," in *Intl. Symp. on CGO: feedback-directed and runtime optimization*, 2003.

[21] M. Süßkraut, T. Knauth, S. Weigert, U. Schiffel, M. Meinhold, and C. Fetzer, "Prospect: A compiler framework for speculative parallelization," in *Intl Symposium on Code Generation and Optimization*, 2010.

[22] U. Hölzle, C. Chambers, and D. Ungar, "Debugging optimized code with dynamic deoptimization," in *Conference on Programming Language Design and Implementation*, 1992.

[23] C. Wimmer, V. Jovanovic, E. Eckstein, and T. Würthinger, "One compiler: deoptimization to optimized code," in *International Conference on Compiler Construction*. ACM, 2017.

[24] O. Flückiger, G. Scherer, M.-H. Yee, A. Goel, A. Ahmed, and J. Vitek, "Correctness of speculative optimizations with dynamic deoptimization," *Programming Languages (POPL)*, vol. 2, 2017.

[25] M. Yusuf, A. El-Mahdy, and E. Rohou, "On-stack replacement to improve JIT-based obfuscation a preliminary study," in *Japan-Egypt Int. Conf. on ECC*. IEEE, 2013.