

# Binary Analysis for Missed Vectorization Opportunities Detection

AMOS HERZ and ALESSANDRO LEGNANI, ETH Zürich, Switzerland

## 1 INTRODUCTION

Modern compilers are often able to automatically vectorize code using SIMD instructions. Take, as an example, the following code snippet:

```
1 void copy(long *restrict a, long *restrict b, unsigned long n) {
2     for (unsigned long i = 0ul; i < n; i++) {
3         a[i] = b[i];
4     }
5 }
```

Listing 1. copy.c

We can compile it with the following set of compiler flags

- -O3: tells the compiler to use the highest level of optimization available.
- -fno-tree-loop-distribute-patterns: prevents replacing the loop with a call to memcpy
- -fno-tree-vectorize: prevents vectorization

Which will produce the following assembly code:

```
1 .L3:
2     movq    (%rsi,%rax,8), %rcx
3     movq    %rcx, (%rdi,%rax,8)
4     addq    $1, %rax
5     cmpq    %rax, %rdx
6     jne     .L3
```

Listing 2. copy.c compiled with vectorizations disabled

However, by compiling without the -fno-tree-vectorize flag, the compiler will produce the following vectorized code (note the use of wider instructions and registers):

```
1 .L4:
2     movdqu  (%rsi,%rax), %xmm0
3     movups  %xmm0, (%rdi,%rax)
4     addq    $16, %rax
5     cmpq    %rcx, %rax
6     jne     .L4
```

Listing 3. copy.c compiled with vectorizations enabled

Nonetheless, autovectorization is *difficult*, compilers tend to miss many optimizations (as shown by Feng et al. [2]), and more than often vectorizing a small piece of code requires large changes to the whole code base (as was done for example by Chen et al. [1]). The main goal of this project is to develop a method for identifying missed opportunities for vectorization in existing code. That is, given an existing binary, we want to identify loops that could be vectorized but are not.

## 2 APPROACH

### 2.1 Dataflow Analysis

Let's look at Listing 2: there, we can realize that the load and store instructions are completely independent (from each other, but also from other instructions of the same type) and could, theoretically, be computed in parallel, therefore vectorized. This can be concluded by simply looking at the dataflow.

Since all the necessary data to build a dataflow graph can be obtained from the trace of the program, the idea is to use a dynamic analysis framework (such as Intel Pin<sup>1</sup>, DynamoRIO<sup>2</sup> or angr<sup>3</sup>) to perform a dynamic analysis that traces an execution and identifies vectorizable code. Given the trace of execution of a program (instructions and memory accesses), we can perform a dataflow analysis to identify "parallel" computations and thus, vectorization opportunities.

### 2.2 Test Inputs

To test our results, we will use an already existing autovectorization benchmark such as TSVC<sup>4</sup>. Other random program generators such YARPGen<sup>5</sup> could be used to generate more, less specific, test inputs.

## REFERENCES

- [1] Yishen Chen, Charith Mendis, and Saman Amarasinghe. 2022. All you need is superword-level parallelism: systematic control-flow vectorization with SLP. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) (*PLDI 2022*). Association for Computing Machinery, New York, NY, USA, 301–315. <https://doi.org/10.1145/3519939.3523701>
- [2] Jing Ge Feng, Ye Ping He, and Qiu Ming Tao. 2021. Evaluation of Compilers' Capability of Automatic Vectorization Based on Source Code Analysis. *Scientific Programming* 2021 (30 Nov 2021), 3264624. <https://doi.org/10.1155/2021/3264624>
- [3] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*.

<sup>1</sup><https://software.intel.com/sites/landingpage/pintool/docs/98830/Pin/doc/html/index.html>

<sup>2</sup><https://dynamorio.org/>

<sup>3</sup><https://angr.io/>, Shoshitaishvili et al. [3]

<sup>4</sup>[https://github.com/UoB-HPC/TSVC\\_2](https://github.com/UoB-HPC/TSVC_2)

<sup>5</sup><https://github.com/intel/yarpgen>