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:

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

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:

```
.L3:
    movq  (%rsi,%rax,8), %rcx
    movq  %rcx, (%rdi,%rax,8)
    addq  $1, %rax
    cmpq  %rax, %rdx
    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):

```
.L4:
    movdqu  (%rsi,%rax), %xmm0
    movups  %xmm0, (%rdi,%rax)
    addq    $16, %rax
    cmpq    %rcx, %rax
    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.

Authors' address: Amos Herz, amherz@ethz.ch; Alessandro Legnani, alegnani@ethz.ch, ETH Zürich, Zürich, Switzerland, 8092.

2 APPROACH

The idea is to use a dynamic analysis framework (such as Intel Pin[1], DynamoRIO[2] or angr[3])to performs 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.

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

[1] https://software.intel.com/sites/landingpage/pintool/docs/98830/Pin/doc/html/index.html
[2] https://dynamorio.org/
[3] https://angr.io/