Speculative SuperOptimization: Boosting Performance via Speculation-Driven Dynamic Binary Optimization

A Technical Report submitted to the Department of Computer Science

Presented to the Faculty of the School of Engineering and Applied Science

University of Virginia • Charlottesville, Virginia

In Partial Fulfillment of the Requirements for the Degree

Bachelor of Science, School of Engineering

Muhammad Abdullah

Spring, 2022

Ashish Venkat, Department of Computer Science

Speculative SuperOptimization: Boosting Performance via Speculation-Driven Dynamic Binary Optimization

Introduction

Recently, the software computing industry has seen rapid growth (Software - Worldwide, 2021), and new challenges have emerged regarding market needs and security. In response to that, modern compilers and JIT engines have made significant progress in generating carefully tuned and optimized machine code (Hall, 2009). This, however, has seen limited impact as a considerable chunk of wasteful computation persists due to compilers' inability to adapt execution to changing workload patterns and hardware resource availability. In addition to that, the progress in the ISA domain has been further limited by the market risks associated with legacy software. Legacy software forces the ISA to not majorly change, causing software and hardware engineers to consider inefficient workarounds**.** All of this has made apparent the absolute need to more aggressively and seamlessly super-optimize machine code that can adapt to dynamic execution environments even post-compilation and post-deployment.

This project intended to do just that by taking advantage of the existing hardware like the micro-op translation interface and existing speculation techniques like branch and value prediction. It also intends to use Pin, which is an intel tool that allows the user to instrument instructions or basic blocks dynamically, and thus change or gather information in regards to the program as it runs. Through the use of these features, this project intended to super-optimize loops by detecting them, and then potentially vectorizing them after running an analysis to determine their eligibility. The project was completed in two parts, with the first focusing on detecting a loop through a stream of instructions, and the second narrowing down the number of loops to vectorizable after studying their instruction mix, dependency analysis, and predictability.

The project was completed under Dr. Ashish Venkat, a Computer Architecture Professor in the Department of Computer Science, and a Ph.D. student Logan Moody.

Loop Detection Tool

The first step in the research was writing an intel pin tool to detect loops. Intel pin is a dynamic binary instrumentation framework that allows you to gain information regarding the instructions running and perform analysis on them dynamically. The loop detection pin tool written took advantage of this fact and attempted to detect loops dynamically by instrumenting each basic block encountered in execution. A general overview of the algorithm used for loop detection is provided below.

1) Keep a stack of tail addresses for each basic block encountered. Tail address here means the address of the last instruction in the basic block.
2) When a basic block is encountered, check if the tail address is already on the stack and whether a loop is being processed currently. If a loop is not being processed currently, that means we have found a new loop.
3) At this point, create a vector of basic blocks that are within the loop using the information on the stack.
4) Next, clear the stack and add the current tail address to it so that only the address where the loop is first detected remains on it. Also, set the current status inLoop to true.
5) If a tail address is already on the stack, and inLoop is true, that means the program is at the end of an iteration for the current loop being processed. At this point, update statistics in regards to the loop such as instructions and iterations. Again, clear the stack and add only the current tail address to it.

6) If a basic block is encountered that is not on the stack, and inLoop is true, that means either this is a basic block in the middle of the loop or this is the end of the loop. In order to check which one it is, check if this basic block is on the vector of basic blocks created when the loop was initially found.
7) If it is not on the list, that means we are at the end of the loop. At this point, check if a LoopStream object with the same basic block list and iterations exists. If yes, then increment the entry count for that LoopStream object. Otherwise, create a LoopStream Object, storing all the necessary statistics, and add it to the loops found vector.
8) Reset all variables. This includes clearing the stack and basic block vector. It also includes resetting inLoop to false, and instruction count to 0.
9) One special case that needs to be considered is if at the end of an iteration it is noted that the just-finished iteration had different basic blocks than the previously run, then it should create a separate LoopStream object for the first part and the second part of the loop.

The above algorithm should detect all regular loops but in the case of nested loops, it only detects the innermost loop.

Analysis Tool

In the analysis script, all the LoopStream objects found were analyzed to determine their eligibility for vectorization. Two of the main considerations included the instruction mix and the dependency analysis. For the instruction mix, each instruction in the LoopStream object was analyzed and then characterized in one of the following categories: 1) Reads Memory 2) Writes to Memory 3) Control Flow 4) Arithmetic. These values were then stored within the LoopStream object. Next, for dependency analysis, the regular live-in live-out analysis algorithm was used in order to determine if a certain iteration of the loop depended on the previous iteration. This analysis was limited to registers, and not extended to memory due to time limitations.

Preliminary Results

The two tools were run on a microbenchmark created to test their implementation. The microbenchmark was curated to test different sorts of loops that could possibly be encountered. The test file is provided in Appendix A. Types of loops included in the test file include:

1) Regular* single loop
2) Regular single loop that calls a function that contains a loop
3) Double nested* regular loop
4) Triple Nested* regular loop
5) Double nested loop that calls a method that contains a loop
6) Triple nested loop that calls a method that contains a loop
7) Regular loop with dependency
8) Double nested loop that calls two functions back to back*, each function contains a loop

* Regular corresponds to first and the second loop in the test

* Double nested loop corresponds to the loop within a loop given in the test

* Triple nested loop corresponds to a loop within a loop within a loop as shown in the test

* Loop within a Loop that first call sum1() and them sum2() in the inner-most loop as shown in the test.

The tools successfully analyzed and detected the loops they were intended to. In the case of nested loops, it was intended that only the innermost loop be detected so only that was checked for. Out of the 8 different loops tested, all 8 were successfully detected. Sample results are provided for LoopDetection in Figure A and for Analysis in Figure B.

Figure A

Id	Iterations	Entries*	Instructions/Iteration
1	50	1	10
2	50	1	10
3	50	6020000	14
4	50	6012000	7
5	300	10000	7

* Entries: How many times a loop was entered to be run. Meaning how many different times was the loop started

Figure B

Id	Arithmetic	Read Memory	Write Memory	Control Flow
1	2	4	2	2
2	2	4	2	2
3	4	7	1	2
4	1	3	1	2
5	1	3	1	2

If all these loops were vectorized, there would be a significant speedup that could be achieved. Since these preliminary results were based on a microbenchmark with only loops, the speed-up effects cannot be quantified yet. These could be quantified once the tools are refactored to run on SPEC benchmarks.

Related Work

Some of the other work done in the area includes a paper by Moseley, Grunwald, Connors, et al. that also provided a tool for detecting loops and creating a profile of them. Additionally, a paper by Porpodas, Magni, and Jones intended to introduce new techniques to vectorize code and potentially get a speed up. Another paper by Maleki, Gao, et al. studied the effects of vectorizing loops and it also concluded that a speedup can be achieved by implementing vectorization. However, both of the last papers studied vectorization and its effects only at the compiler level, whereas this project intends to study vectorization at runtime post compilation and post deployment.

Conclusion

Loop detection and analysis pin tools were successfully written in order to find and determine the eligibility of loops for vectorization. This information provides insight into the potential speed up that could be reached if loops were vectorized post compilation and post-deployment. The next step would be to realize this speed up and prove its existence by implementing similar mechanisms in gem5, a hardware simulator. The pin tools written could be directly translated to some extent to the gem5 implementation, and could also serve as a verification script for gem5.

References

Hall, M., Padua, D., & Pingali, K. (2009). Compiler research. Communications of the ACM, 52(2), 60–67. https://doi.org/10.1145/1461928.1461946

Software - Worldwide. (n.d.). Retrieved October 26, 2021, from https://www.statista.com/outlook/tmo/software/worldwide

Appendix A

```
#include <iostream>
#include <cstdlib>

uint64_t a[100], b[100], c[100];

void sum1() {
    for (int i = 0; i < 100; i+=2) {
        c[i] = a[i] | b[i];
    }
}
void sum2() {
    for (int i = 1; i < 100; i+=2) {
        c[i] = a[i] | b[i];
    }
}

int main() {
    for (int i = 0; i < 100; i+=2) {
        a[i] = 3;
        b[i] = 4;
    }
    for (int i = 1; i < 100; i+=2) {
        a[i] = 4;
        b[i] = 5;
    }
    for (int i = 0; i < 5000; i++) {
        sum1();
    }
    for (int i = 0; i < 2000; i++) {
        sum2();
    }
    /*for (int i = 0; i < 100; i++) {
        std::cout << c[i] << "\n";
    }*/
    for (int i = 0; i < 100; i++) {
        for (int j = 0; j < 100; j++) {
            for (int k = 0; k < 300; k++) {
                a[j] = 3;
            }
        }
    }
    for (int j = 0; j < 100; j++) {
            for (int k = 0; k < 50; k++) {
                a[k] = 3;
            }
    }
    int a = 10;
    int b = 10;
    int c;
    for(int i=0;i <10;i++) {
        c = a+b;
        b = c*i;
    }
    for (int j = 0; j < 100; j++) {
            for (int k = 0; k < 50; k++) {
                sum1();
            }
    }

    for (int j = 0; j < 100; j++) {
            for (int k = 0; k < 50; k++) {
                sum2();
            }
    }

    for (int j = 0; j < 100; j++) {
            for (int k = 0; k < 50; k++) {
```

```
            sum1();
        }
    }

    for (int j = 0; j < 100; j++) {
        for (int k = 0; k < 50; k++) {
            sum2();
            sum1();
        }
    }

    for (int i = 0; i < 100; i++) {
        for (int j = 0; j < 100; j++) {
            for (int k = 0; k < 300; k++) {
                sum1();
            }
        }
    }

    for (int i = 0; i < 100; i++) {
        for (int j = 0; j < 100; j++) {
            for (int k = 0; k < 300; k++) {
                sum2();
            }
        }
    }
    for (int i = 0; i < 100; i++) {
        for (int j = 0; j < 100; j++) {
            for (int k = 0; k < 300; k++) {
                sum1();
                sum2();
            }
        }
    }
}
```