CPSC 3300

Project 2 - Loop Optimizations

**Grading Weights:**

48% description of the techniques

4% description of loop-carried dependencies

4% description of aliasing

30% performance improvement examples

10% bibliographic references and citation style

4% section headers

This is an individual or team of two project assignment.

**Summary of the assignment**

You will write a 3 to 5 page paper (6 to 10 pages for teams of two) to

1. describe the loop optimization techniques listed below (at least 3 working individually or at least 6 as teams of two). You shall use example codes to show the transformations before and after the optimization,
2. describe two hurdles to loop optimization: loop-carried dependencies and aliases, and
3. run experiments and report on performance benefits obtained by applying loop optimization techniques available in gcc or g++ to the whetstone benchmark.

**Details:**

**List of loop optimization techniques:**

* loop fusion
  + This optimization will take several different loops and fuse them into one loop reducing execution time and any loop overhead.
* loop invariant code motion
  + The computation inside of the loop is averted, and in turn the computation overhead on the compiler is also averted.
* loop unrolling
  + The body of the loop is pulled out and executed as stand-alone code. So, the code will be the the loop body written the number of times that the loop would have run. This saves on execution time by removing the need for loop control and test instructions.
* Loop interchange:
  + The inner loops are swapped with the outer loops. This will allow loop variables to index into an array, improving locality of reference.
* Loop peeling:
  + Simplifies a loop that has a tricky first iteration by extracting that iteration and placing it before the loop in the code.

**Loop-carried dependencies**

Loop-carried dependencies in pipelined processors are pipeline hazards because they can create stalls in the program.

These hazards can be classified as one of the following three main types of dependencies:

1. Control Dependency

2. Structural Dependency

3. Data Dependency

Control dependencies occur when control instructions like BRANCH, JMP, or CALL are transferred. The processor will have no way of knowing the target address of the instructions when it needs to insert a new instruction in the pipeline. As a result, some potentially unwanted instructions could be inserted into the pipeline. In order to avoid control dependencies, the instruction fetch must be halted until the processor receives the target address of the branch instruction. A delay slot can be added to the pipeline until it receives the target address it needs. Unfortunately, the delay solution will create a stall in the pipeline. To remedy this, branch prediction is used to eliminate the stalls created by the control dependencies. First, predict which branch will be taken. The stakes are low because branch penalty (the number of stalls that are introduced in the pipelined processor during branch operations) is zero. A general equation is used for this:

If the target address exists after the kth stage, then the pipeline will have (k – 1) stalls in it. Finally, the total number of stalls that are created in the pipeline as a result of branch operations or instructions is equal to the branch frequency multiplied by the branch penalty.

Structural dependencies occur if there is a resource conflict within the pipeline. Usually a resource conflict is a specific situation when more than one instruction tries to access the same resource while inside the same cycle. Resources can be memory, the arithmetic-logic unit, or simply a register. In order to avoid resource conflict in structural dependencies, the instruction requesting a resource that is already in use must wait until that resource becomes available. This is where the pipeline will stall. In order to reduce structural dependency stalls on the pipeline, use the hardware mechanism renaming. Renaming will split the resource into two independent modules. Then, the independent modules are used to store the program instructions and data in separate modules, Code memory and Data memory.

Data dependencies, also known as data hazards, are the final of the three main types of dependencies. These occur when an instruction writes something after a second instruction has already read it, when an instruction reads something before the second instruction overwrites it, or when both instructions write to the same location in memory. In this case, operand forwarding can be used to prevent stalls in the pipeline. The registers that exist in between the stages of instructions hold intermediate output allowing the dependent instruction to access the new value from the register directly.

If any of the above dependencies occur in between loop iterations then they are called “loop-carried dependencies.” Therefore, any iterations of the loop cannot be executed in parallel if the loop is carrying dependencies.

**Array element aliasing**

Aliasing of array elements can completely prevent the loops from running in parallel. Element aliasing occurs if two possible references are for the same location in memory. Array element aliasing can happen in C because array referencing and array definition can both be done using pointer arithmetic. One solution is to use restricted pointers to allow the compiler to perform parallel execution of a loop efficiently.

Be sure to provide citations for any source you use. (Don't rely entirely on Wikipedia for the descriptions; double check with other sources.)

**Hands-on test and results report for GNU compilers on the Whetstone code.**

There are many techniques available in GNU compilers (gcc, g++, or gfortran) and they can impact the performance of programs, e.g., the Whetstone code. Use command man gcc to get the manual.

Note that the compilers have several levels of optimizations, i.e., -O0, -O1, -O2, etc. The higher the number, the more optimization techniques are applied. These levels directly control some optimizations, but not all. Again read the gcc manual. On the other hand, most optimizations are only enabled if an -O level is set on the command line. Otherwise they are disabled, even if individual optimization flags are specified. For example, if you would like to turn on  -funroll-loops, you can use gcc -O2 -funroll-loops. You can also refer to this online page <https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html>.

Start with the Whetstone code and test the various compiler options. (I would recommend that you write a Makefile to compile the program and a script to run the experiments.)

Report the hardware/OS platform and the version of gcc/g++ you use for experiments; also give the exact compiler invocation flags for each test for Whetstone looping for 200,000 times. Report the execution time in a table with each optimization technique enabled.

Hardware/OS Platform – Ubuntu 20.04.2 LTS 64-bit

Processor – Intel Core i7=10510U CPU @ 1.80GHz \* 8

GNOME Version – 3.36.8

Version of gcc – gcc (Ubuntu 9.3.0-17ubuntu1~20.04) 9.3.0

Loop Fusion - “-fschedule-fusion -O1”

Loop Unrolling – “-funroll-loops -O1”

Loop Peeling - “-fpeel-loops -O1”

First Run:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Base Case | Loop Fusion | Loop Unrolling | Loop Peeling | All 3 Optimizations |
| Task-Clock | 2516.13 msec | 1458.5 msec | 1459.82 msec | 1,463 msec | 767.08 msec |
| CPUs Utilized | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 |
| Time Elapsed | 2562.764567 msec | 1458.9 msec | 1,460.07 msec | 1,463 msec | 767.40 msec |
| Context-Switches | 59 | 31 | 5 | 24 | 20 |
| CPU Migrations | 0 | 0 |  | 1 |  |
| Page Faults | 74 | 75 | 75 | 72 | 71 |
| Cycles | 10,991,697,599 | 6,322,867,329 | 6,337,511,867 | 6,327,314,711 | 3,345,551,656 |
| Instructions | 23,234,140,793 | 9,674,854,601 | 8,278,093,408 | 9,674,748,380 | 3,366,565,547 |
| Branches | 2,335,997,271 | 1,514,132,002 | 1,074,778,059 | 1,514,110,413 | 519,374,005 |
| Branch Misses | 55,976 | 18,384 | 14,886 | 16,343 | 12,530 |

Second Run:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Base Case | Loop Fusion | Loop Unrolling | Loop Peeling | All 3 Optimizations |
| Task-Clock | 2583.36 msec | 1,461.21 msec | 1,396.84 msed | 1,467.79 | 768.59 |
| CPUs Utilized | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 |
| Time Elapsed | 2,583.68 msec | 1,461.61 msec | 1,397.04 | 1,468.07 | 768.76 |
| Context-Switches | 15 | 37 | 5 | 25 | 20 |
| CPU Migrations | 0 | 0 | 0 | 1 | 0 |
| Page Faults | 78 | 74 | 75 | 72 | 71 |
| Cycles | 11,004,657,283 | 6,326,320,692 | 6,054,189,319 | 6,325,521,523 | 3,337,354,269 |
| Instructions | 23,233,745,266 | 9,674,900,457 | 8,278,024,876 | 9,674,775,567 | 3,366,594,203 |
| Branches | 2,335,914,737 | 1,514,142,859 | 1,074,767,380 | 1,514,115,882 | 519,379,550 |
| Branch Misses | 51,373 | 17,469 | 15,037 | 16,528 | 13,218 |

Third Run:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Base Case | Loop Fusion | Loop Unrolling | Loop Peeling | All 3 Optimizations |
| Task-Clock | 2,568.30 msec | 1,469.96 msec | 1,395.65 msec | 1,460.48 msec | 763.07 msec |
| CPUs Utilized | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 |
| Time Elapsed | 2,568.93 msec | 1,470.22 msec | 1,396.00 msec | 1,460.77 msec | 763.32 msec |
| Context-Switches | 14 | 10 | 13 | 6 | 6 |
| CPU Migrations | 0 | 0 | 0 | 0 | 0 |
| Page Faults | 74 | 72 | 73 | 74 | 72 |
| Cycles | 11,006,691,930 | 6,331,158,750 | 6,056,096,388 | 6,320,214,811 | 3,337,297,489 |
| Instructions | 23,233,664,312 | 9,674,631,873 | 8,278,075,418 | 9,674,574,119 | 3,366,412,587 |
| Branches | 2,335,900,630 | 1,514,086,698 | 1,074,777,817 | 1,514,074,547 | 519,343,782 |
| Branch Misses | 52,025 | 16,944 | 16,123 | 15,930 | 12,277 |

Averages:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Base Case | Loop Fusion | Loop Unrolling | Loop Peeling | All 3 Optimizations |
| Task-Clock |  |  |  |  |  |
| Time Elapsed |  |  |  |  |  |
| Context-Switches |  |  |  |  |  |
| Cycles |  |  |  |  |  |
| Instructions |  |  |  |  |  |
| Branches |  |  |  |  |  |

References

|  |  |
| --- | --- |
| [1] | K. Allen, “CS 293S Parallelism and Dependence Theory (PowerPoint Slides),” in *Optimizing Compilers for Modern Architectures: A Dependence-based Approach (Ch. 9)*, San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001. |
| [2] | Oracle Solaris Studio, “Oracle Solaris Studio 12.3 C User’s Guide,” *Oracle*, 2012. [Online]. Available: https://docs.oracle.com/cd/E24457\_01/html/E21990/bjafa.html. [Accessed: 08-Apr-2021]. |
| [3] | A. Saggu, “Loop Optimization Techniques | Set 2,” *GeeksforGeeks*, 02-Jun-2020. [Online]. Available: https://www.geeksforgeeks.org/?p=421410. [Accessed: 08-Apr-2021]. |
| [4] | H. Agarwal, “Loop Unrolling,” *https://www.geeksforgeeks.org/loop-unrolling/*, 19-Feb-2018. [Online]. Available: https://www.geeksforgeeks.org/loop-unrolling/. [Accessed: 08-Apr-2021]. |
| [5] | R. Pandey, “Basic Code Optimizations in C,” *GeeksforGeeks*, 28-Jun-2019. [Online]. Available: https://www.geeksforgeeks.org/basic-code-optimizations-in-c/. [Accessed: 08-Apr-2021]. |
| [6] | P. Patel, “Loop Optimization in Compiler Design,” *GeeksforGeeks*, 21-Nov-2019. [Online]. Available: https://www.geeksforgeeks.org/loop-optimization-in-compiler-design/. [Accessed: 08-Apr-2021]. |