CPSC 3300

Project 2 - Loop Optimizations

Reagan Leonard

April 3, 2020

**Loop Optimization Technique Descriptions**

Loop Fission

Loop fission is an optimization technique that breaks a single loop into multiple loops (that iterate over the same index range) where each of the multiple resulting loops covers only a specific section of the original loop's body [1]. The goal of loop fission is to break down a large loop body into several smaller ones to achieve better utilization of what is known as “locality of reference”---the tendency of a processor to access the same set of memory locations repetitively over a short period of time. This optimization results in the greatest increase in efficiency when implemented in computers with multi-core processors that are able to split a task into multiple tasks for each processor [2]. A simple example of loop fission in the C programming language is shown below:

Before fission:

**int i, a[100], b[100];**

**for (i = 0; i < 100; i++){**

**a[i] = 1;**

**b[i] = 2;**

**}**

After fission:

**int i, a[100], b[100];**

**for (i = 0; i < 100; i++){**

**a[i] = 1;**

**{**

**for (i = 0; i < 100; i++){**

**b[i] = 2;**

**}**

These two blocks of code perform the exact same function (both set every element of arrays a and b to 1 and 2 respectively), however the 2nd block does so much more efficiently by splitting it into 2 loops. This is the purpose of loop fission.

Loop Fusion

Loop fusion is an optimization technique that combines multiple loops into a single loop that indexes over the same range as the two original loops [3]. This is only possible when the two original loops iterate over the same range and do not reference each other's data [4]. Loop fusion is not guaranteed to always improve runtime. On certain computer architectures, leaving the original two loops separate might actually perform better than fusing them together into one because there may be increased data locality within each loop [2]. A simple example of loop fusion in the C programming language is shown below:

Before fusion:

**int i, a[300], b[300];**

**for (i = 0; i < 300; i++){**

**a[i] = i + 3;**

**}**

**for (i = 0; i < 300; i++){**

**b[i] = i + 4;**

**{**

After fusion:

**int i, a[300], b[300];**

**for (i = 0; i < 300; i++){**

**a[i] = i + 3;**

**b[i] = i + 4;**

**}**

Again, these two blocks of code are identical in the task that they perform the exact (both set every element of arrays a and b to i + 3 and i + 4 respectively), however the 2nd block does so using less lines of code (and hopefully improving runtime) by combining the 2 loops into one because they cover the same “area” (both are indexing from 0-299). This is the purpose of loop fusion.

Loop Peeling

Loop peeling is a special case of loop splitting (similar to loop fission, but breaks a loop into multiple loops which have the same bodies but iterate over different contiguous portions of the index range) which splits any problematic first (or last) few iterations from the loop and performs them outside of the loop body or in a separate loop [5]. This optimization technique is mostly helpful when there are outlier iterations (or edge cases) that could cause a significant increase in runtime execution or could even potentially break the loop. A simple example of loop peeling is shown below [6]:

Before peeling:

**int i, a[10];**

**for (i = 0; i < 10; i++){**

**a[i] = a[i-1] + 3;**

**}**

After peeling:

**int i, a[10];**

**for (i = 0; i < 1; i++){**

**a[i] = i + 3;**

**}**

**for (i = 1; i < 10; i++){**

**a[i] = a[i-1] + 3;**

**{**

The intent of the first loop is to set all 10 elements in the array *a* equal to the first 10 multiples of 3 (3,6,9,12…30). However, the first loop iteration will cause a problem because of how the loop is written. Because each element is set to the previous element’s value + 3, the first iteration will throw an error: there is no element prior to the first one. To fix this, we “peel” off the first iteration and place it into a loop all by itself [7]. We will now end up with what we want: array *a* will be multiples of 3 from 3 through 30.

**Loop-carried dependencies**

A loop-carried dependence is essentially when an operation in a certain iteration of a loop (say, iteration *i*) depends on information or an operation from another iteration of that same loop (say, *i-1* or *i+1* or some other iteration) [8]. A simple instance of this happening is shown in the short block of code below:

**for (i = 0; i < 10; i++){**

**a[i] = x \* i;**

**b[i] = a[i-1];**

**}**

The code above shows a simple instance of a loop-carried dependence: the second line in the body of the loop relies on the first line of the loop that occurred in the previous iteration (*i-1*). In other words, the first line of the loop (a[i] = x \* I;) would have to be executed in the *0th* iteration of the loop before the second line (b[i] = a[i-1];) could execute in the *1st* iteration of the loop. This is because the second line needs to know what a[i-1] equals before it can set b[i] to that value [9]. In the case mentioned above, the compiler would be searching for the value stored in a[0] (equal to a[i-1] in the 1st iteration).

**Array element aliasing**

Aliasing refers to a situation that arises when a location in computer [memory](https://en.wikipedia.org/wiki/Memory_(computers)) is able to be accessed using symbolic names within a program. Due to this, it is possible for variables and array elements to often be changed by accident or unknowingly, making it particularly difficult to understand, analyze, and optimize programs or loops within programs [11].

A simple example of this is that most implementations of one of the most common programming languages---C---do not perform any sort of bounds checking on arrays [10]. Because of this, it is possible to exploit said implementation of C via the compiler and the computer architecture's assembly language conventions in order to achieve the effect of aliasing by writing to a data location outside of the bounds of the array. This is essentially a form of buffer overflow. Doing this causes what is called “undefined behavior” (according to the C language manual). However, what often actually happens with many implementations of C is the behavior described in the paragraph and code shown below.

The only way the following example could occur and cause an error in a program or loop is if the data we’re concerned with is stored in contiguous/adjacent locations in the computer’s memory [12]. For instance, if an array is created on the stack with a variable laid out in memory directly beside that array, it would be possible to index outside the bounds of the array and directly change the variable by changing the relevant array element. For example, if there is an int array of size 3 (for this example's sake, we’ll call it *array*), next to another int variable (we’ll call it *i*), *array[3]* (i.e., the 4th element) would be aliased to *i* if they are adjacent in memory. This example is shown by the following code snippet:

**# include <stdio.h>**

**int main()**

**{**

**int array[3] = { 2, 4, 6 };**

**int i=5;**

**/\* Write beyond the end of array. Undefined behavior in standard C, will write to i in some implementations. \*/**

**array[2] = 10;**

**printf("element 0: %d \t", array[0]); // this outputs 2**

**printf("element 1: %d \t", array[1]); // this outputs 4**

**printf("element 2: %d \t", array[2]); // this outputs 6**

**printf(“element 3: %d \t”, array[3]); // this outputs 10, if array element aliasing occurred**

**printf("i: %d \t\t", i); // this might also output 10 instead of 5 due to aliasing, but the compiler might have i stored in a register and print 5**

**/\* array size is still 3. \*/**

**printf("array size: %d \n", (sizeof(array) / sizeof(int)));**

**}**

The above example is possible in some implementations of C because an array is a block of contiguous memory and array elements are merely referenced by offsets off the address of the beginning of that block multiplied by the size of a single element. Since C has no bounds checking, indexing and addressing outside of the array is possible.

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

|  |  |  |  |
| --- | --- | --- | --- |
| **OS platform** | **Gcc/g++** | **Compiler flags** | **Execution time** |
| Linux | Gcc | None | 3.422 seconds |
| Linux | Gcc | -O1 | 1.997 seconds |
| Linux | Gcc | -O2 | 1.391 seconds |
| Linux | Gcc | -O3 | 1.349 seconds |
| Linux | Gcc | -O3 -funroll-loops | 1.262 seconds |
| Linux | Gcc | -O4 | 1.348 seconds |
| Linux | Gcc | -O5 | 1.357 seconds |
| Linux | Gcc | -O6 | 1.348 seconds |
| Linux | Gcc | -O7 | 1.352 seconds |
| Linux | Gcc | -O8 | 1.349 seconds |
| Linux | Gcc | -O9 | 1.370 seconds |
| Linux | Gcc | -O10 | 1.349 seconds |

As shown in the table above, in general, the more compiler flags that are used, the greater the speedup in execution time. However, there is clearly a principle of diminishing returns also being shown here. That is, beyond about 3 optimization techniques being applied (-O3), the speedup difference is much less and often even increases by a few hundredths of a second. The only exception to this is when combining the “-O3” flag with the “-funroll-loops” flag, which resulted in the overall lowest execution time by almost 1/10th of a second.

**References**

[1] “Loop Fission,” *sciencedirect.com*, [Online]. Available: <https://www.sciencedirect.com/topics/computer-science/loop-fission> [Accessed March 25, 2020].

[2] “Loop Fission and Fusion,” *Wikipedia.com,* [Online]. Available: <https://en.wikipedia.org/wiki/Loop_fission_and_fusion> [Accessed March 25, 2020]

[3] “Loop Fusion,” *compileroptimizations.com,* [Online]. Available: <https://compileroptimizations.com/category/loop_fusion.htm> [Accessed March 25, 2020]

[4] “Loop Fusion,” *sciencedirect.com,* [Online]. Available: <https://www.sciencedirect.com/topics/computer-science/loop-fusion> [Accessed March 25, 2020]

[5] “First example of loop peeling,” *researchgate.net,* [Online]. Available: <https://www.researchgate.net/figure/First-example-of-loop-peeling_fig1_4001066> [Accessed March 25, 2020]

[6] “Loop Peeling,” *Wikipedia.com,* [Online]. Available: <https://en.wikipedia.org/wiki/Loop_splitting> [Accessed March 25, 2020]

[7] “A technique for variable dependence driven loop peeling,” *ieee.org,* [Online]. Available: <https://ieeexplore.ieee.org/document/1173607> [Accessed March 25, 2020]

[8] “Loop-independent vs. loop-carried dependencies,” *people.engr.ncsu.edu,* [Online]. Available: <https://people.engr.ncsu.edu/efg/506/s10/www/lectures/notes/lec5.pdf> [Accessed March 30, 2020]

[9] “Dependence Analysis,” *engineering.purdue.edu,* [Online]. Available: <https://engineering.purdue.edu/~milind/ece573/2011spring/lecture-14.pdf> [Accessed March 30, 2020]

[10] “Aliasing and Parallelization,” *oracle.com,* [Online]. Available: <https://docs.oracle.com/cd/E24457_01/html/E21990/bjafa.html> [Accessed March 30, 2020]

[11] “Aliasing (computing),” *Wikipedia.com,* [Online]. Available: <https://en.wikipedia.org/wiki/Aliasing_(computing)> [Accessed March 30, 2020]

[12] “Aliasing of C-language arrays,” *tamivox.org,* Jan. 3, 2014. [Online]. Available: <http://tamivox.org/redbear/array_alias/index.html> [Accessed March 30, 2020]