The Linux Scheduler: Complicatedly Simple Calculations

Kurt Slagle, Dustin Hauptman
{kslagle,dhauptman}@ku.edu
University of Kansas, USA

***Abstract*—The Linux scheduler is written to be adaptive, flexible, and customizable. However, with this modularity comes complexity. As the kernel cannot easily perform floating point computations, there are present, inside the scheduler's per-task computations, a number of operations that are seemingly overly complex in an attempt to reproduce the floating point operations using fixed-point operations. This paper looks in-depth at one function called repeatedly inside the kernels Completely Fair Scheduler, and presents a statistical analysis of the values passed to and from this function, as well as the accuracy of the performed computations.**

I. Introduction

For newcomers to the Linux kernel, it would not be difficult to find oneself struggling to understand why a piece of code is written the way that it is. Comments in the source code are often detailed, but do not always explain the choices that influenced the implementation of a function. Floating point calculations are disabled inside the kernel, and enabling them is not free from overhead, so the code in the kernel is written to avoid using them. This code can often be difficult to follow, and it is not often clear why the code is written in that way.

When exploring the implementation of the Completely Fair Scheduler (CFS) in the Linux kernel, we were intrigued by the code used in *__calc_delta* in $fair.c$. The concept of the Completely Fair Scheduler (CFS) is deceptively simple. The source code for this inside the Linux kernel appears, initially, far more complex than would seem necessary. We suspected this code is written this way to help ensure the result of the calculation is correct on a variety of machines. The code is optimized to perform well for a number of special cases, but this comes at the sacrifice of readability.

We hypothesize that this implementation could be simplified if it was assumed it was to be run on a 64-bit system. In this paper, we present an discussion of the changes made, an analysis of the performance impacts of those changes, and an analysis of the accuracy of the values obtained from both implementations.

II. Background

The Completely Fair Scheduler seeks to be "fair" to all processes running, weighting processes to provide more CPU to those processes with a high priority without starving lower-priority tasks. Each task maintains a virtual time, which is equivalent to the execution time divided by its weight. The focus of this paper is on the code inside *__calc_delta*, namely on the calculation of $delta\_exec * 1/weight$. Floating point math cannot be used to simplify this, since floating point operations are disabled in the kernel. They can be enabled, but doing so would likely only prove to be a detriment to performance.

What can be considered is the removal of the special-case optimizations to improve readability. *__calc_delta* is intended to compute $delta\_exec * 1/weight$, and the code for this in the default Linux kernel is shown in Listing 1. The code shown simply calculates the new $delta\_exec$ by multiplying it with the weight and inverted load weight. The purpose of implementing it in this way is not immediately clear and not elaborated on in detail. We seek to analyze the benefits gained or lost by implementing the code in this way.

For the aforementioned code, the accuracy of the calculation is determined by tracking the value computed by the kernel and the value computed by using floating point operations in user space. This will be compared to the accuracy achieved by using a simpler version of the code, seen in Listing 2. Because speed is critical inside the scheduler, the time spent inside both versions of the code is compared. Since the kernel will perform different operations when using cgroups and not using cgroups for processes, the same analysis is done with benchmarks run with and without using cgroups.

Listing 1: __calc_delta inside fair.c

```
static u64 __calc_delta(u64 delta_exec,
unsigned long weight, struct load_weight *lw)
{
        u64 fact = scale_load_down(weight);
        int shift = WMULT_SHIFT;

        __update_inv_weight(lw);

        if (unlikely(fact >> 32)) {
                while (fact >> 32) {
                        fact >>= 1;
                        shift--;
                }
        }

        /* hint to use a 32x32->64 mul */
        fact = (u64)(u32)fact * lw->inv_weight;

        while (fact >> 32) {
                fact >>= 1;
                shift--;
        }

        return mul_u64_u32_shr(delta_exec,
        fact, shift);
}
```

Listing 2: Altered __calc_delta in fair.c

```
static u64 __calc_delta(u64 delta_exec,
unsigned long weight, struct load_weight *lw)
{
        u64 val;

        val = (!lw->weight)
            ? (delta_exec * (u64)weight * (u64)
                WMULT_CONST)
            : (delta_exec * (u64)weight / (u64)lw->
                weight);

        return val;
}
```

III. Your System

The Linux kernel uses a plethora of macros whose values are configuration-dependent, and thus it should not be assumed that the analysis presented here is reflective of how the changes would perform on other system configurations. The system on which this was tested is a Dell Latitude 5580, with an Intel Core i7 7820HQ @ 2.9 GHz (turbo up to 3.9 GHz), 16GB of DDR4 @ 2400 MHz memory, and 500GB HDD @ 7200 rpm. To keep the CPU running at a relatively constant rate, dynamic frequency scaling was disabled.

IV. Evaluation

Once the changes were made in the kernel we needed to see if there was any noticeable performance loss at both the micro and macro level. For the micro level analysis, the number of cycles taken to execute were found using the RTDSC instruction present on Intel chips. The cycles, along with all input and output data, were then printed out using $trace\_printk$.

There were two sets of data that were gathered using this method. One was Once we obtained the cycle counts and input data we started to analyze both the accuracy of both code versions along with the number of cycles each version takes.

	Min	Max	Mean	Variance	STDDEV
Old Kernel	0.846281	1	0.996221	1.20622e-05	0.00347307
New Kernel	0.999685	1	0.999978	4.807e-10	2.19249e-05

TABLE I: Accuracy w/o CGROUP

	Min	Max	Mean	Variance	STDDEV
Old Kernel	0.994311	1	0.997727	4.17965e-06	0.00204442
New Kernel	0.99984	1	1	5.17154e-12	2.2741e-06

TABLE II: Accuracy w/ CGROUP

...

V. Conclusion

...