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. All information that was needed from the function was then printed out using $trace\_printk$.

For the micro level analysis, the number of cycles were found using the RTDSC instruction present on Intel chips.

At the macro level two sets of data, one using CGROUPS and the other without, were gathered for both kernel versions. When implementing the CGROUPS, six cpu_hog programs (Listing 3) were run and relegated to a single core. The following hierarchy for the CGROUPS were then used $/cgroup1/cgroup1\_1$, $/cgroup1/cgroup1\_2$, and $/cgroup2$. Each group was then given two cpu_hog programs. When not using CGROUPS, the benchmark hackbench was used and allowed to run on all cores. The macro level data was used for both accuracy calculations and timing analysis.

Once we obtained the cycle counts and input data we looked at if there were any noticeable slowdowns and if the accuracy had degraded or not.

A. Accuracy

When calculating the accuracy of the code in both kernel versions, we are assuming that the "ideal" value that could be calculated is the $delta\_exec * 1/weight$ using floating point arithmetic. Since floating point operations are disabled in the kernel by default, we reimplemented all necessary kernel functions in user space and compared them against a floating point version of the ideal calculation. The results of this can be seen in both Table I and Table II. Not surprising is that our $\_\_calc\_delta$ closely mirrors the floating point operation, as the same computation will be performed. More surprising is the that the old $\_\_calc\_delta$ can generate results that are only around 84.63% accurate, where ours never looses accuracy beyond 99.97%.

B. Cycles

The total cycle counts for both kernels can be seen in Table III and Table IV. It is of interesting note that while we aren't able to always outperform, our mean and standard deviation are lower which would suggest that our code produces more consistent results and on average perform better.

C. Timing Analysis

To see if there was any performance perceptible at the macro level, we used hackbench as a scheduler benchmark. There were four different setups that were used for hackbench. These can be seen in Table V and Table VI. Each setup was run 100 times and then the statistics were calculated based on the 100 runs.

```
int main(void) {
        while(1);
        return;
}
```

Listing 3: cpu_hog program

	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

	Min	Max	Mean	Variance	STDDEV	Count
Old Kernel	19	3156	72.7423	889.85	29.8304	40427
New Kernel	22	909	46.9553	378.735	19.4611	40428

TABLE III: Total Cycles w/o CGROUP

	Min	Max	Mean	Variance	STDDEV	Count
Old Kernel	18	4295	75.1081	2854.71	53.4295	77997
New Kernel	19	1642	37.9399	964.117	31.0502	77997

TABLE IV: Total Cycles w/ CGROUP

	Min	Max	STDDEV	Mean
hackbench	0.113	0.156	0.00976	.12828
hackbench -p -T	0.075	0.117	0.008319	0.09316
hackbench -s 1024 -l 500 -g 30 -f 50 -P	15.191	17.054	.30808	15.72765
hackbench -s 1024 -l 500 -g 10 -f 50 -p -T	6.791	7.756	0.191834	7.24506

TABLE V: Hackbench average timing results on new kernel

	Min	Max	STDDEV	Mean
hackbench	0.113	0.154	0.009195	0.13028
hackbench -p -T	0.075	0.119	0.008978	0.09234
hackbench -s 1024 -l 500 -g 30 -f 50 -P	15.132	16.519	0.230227	15.51134
hackbench -s 1024 -l 500 -g 10 -f 50 -p -T	6.742	7.773	0.192186	7.16223

TABLE VI: Hackbench average timing results on old kernel

...

V. Conclusion

...