# APPENDIX

## A. TRIPLING THE WORK

Besides the experiments with Program 1 and Program 2, those presented in this paper, six similar programs also contributed to the discovery of the RAW hiccup. Program 2 was our attempt to double the work of Program 1 by duplicating its single inner loop, the remaining four programs were our attempt to multiply the work of Program 1 by 3 trough 8, also by replicating the inner loop. Figure 12 shows what Program 3 looks like.

Recall that the first clue towards the discovery of the RAW hiccup was that number of L1 data cache misses did not increase when the amount of work was doubled. However, when we *tripled the work* of Program 1 *the number of L1 data cache misses did double*. Adding a fourth inner loop again left the number of L1 data cache misses unchanged. This step-wise increase continued for all eight programs, as is shown in Figure 13. The number of cycles stalled because of pipeline hazards on the other hand increased linearly with the number of inner loops. When considering these data relative to the number of inner loops, as is shown in Figure 14, we see that consecutive programs alternate between being less efficient and more efficient.

At the level of C-source code, this translates to one inner loop suffering from the RAW hiccup for every two inner loops in the program ($\lceil \frac{\#\text{inner loops}}{2} \rceil$). Since all inner loops are compiled into the same instruction sequences (Figure 5) the memory layout must be causing the RAW hiccup. Consider a layout of variables where $sum0$ and $j0$ share a cache line as in line 2 of Figure 9. Introducing a new inner loop consequently introduces also two new variables (e.g. $sum1$ and $j1$) that do not share a cache line. Again adding two new variables (e.g. $sum2$ and $j2$) gives rise to a new RAW hiccup because $sum2$ and $j2$ do share a L1 data cache line. This explains why the number of cache misses only increases when adding a new inner loop to a program with an even number of inner loops.

```
1  for (  long i = 0; i < N; ++i ) {
2    long sum0 = 0;
3
4    for (  long j0 = 0; // init
5           j0 < i;      // test
6           ++j0 ) {     // incr
7
8      sum0 += j0;       // body
9
10   }
11   total += sum0;
12
13
14 }
```

```
1  for (  long i = 0; i < N; ++i ) {
2    long sum0 = 0;
3
4    for (  long j0 = 0; // init
5           j0 < i;      // test
6           ++j0 ) {     // incr
7
8      sum0 += j0;       // body
9
10   }
11   total += sum0;
12
13   long sum1 = 0;
14
15   for (  long j1 = 0; // init
16          j1 < i;      // test
17          ++j1 ) {     // incr
18
19     sum1 += j1;       // body
20
21   }
22   total += sum1;
23
24   long sum2 = 0;
25
26   for (  long j2 = 0; // init
27          j2 < i;      // test
28          ++j2 ) {     // incr
29
30     sum2 += j2;       // body
31
32   }
33   total += sum2;
34
35
36 }
```

Figure 12: Program 3 (right) triples the work of Program 1 (left).

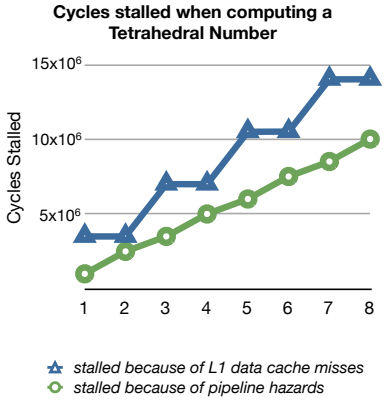


Figure 13: The number of cycles stalled caused by pipeline hazards increases linearly with the amount of work. Remarkably, the number of stalls caused by L1 data cache misses increases in steps.

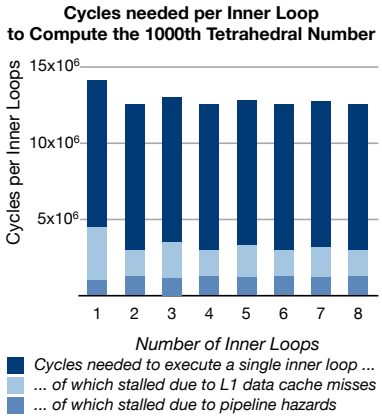


Figure 14: Comparing number of cycles needed to compute a single inner loop shows that programs with an even number of inner loops are more efficient because they suffer from less L1 data cache misses..