| Times event occurred | Benchmark_basic | Benchmark_blocked | Benchmark_col | Benchmark_naïve | Benchmark_rb | Benchmark_row |
|---|---|---|---|---|---|---|
| **L1-dcache-load-misses** | 9,171,473,641 | 1,613,791,542 | 19,900,779,655 | 9,118,146,855 | 1,983,350,659 | 2,445,143,401 |
| **L1-dcache-store-misses** | not supported | not supported | not supported | not supported | not supported | not supported |
| **LLC-loads** | 3,701,733,487 | 180,204,364 | 6,105,213,874 | 3,668,680,840 | 185,354,781 | 480,515,059 |
| **LLC-load-misses** | 4,446,108 | 7,654,430 | 6,747,619 | 3,101,752 | 2,812,196 | 1,448,631 |

Benchmark_row has the lowest count of LLC-misses

Benchmark_blocked has the lowest count of L1-dcache-load-misses

Benchmark_blocked has lowest count of LLC-loads

Benchmark_col has the highest count of L1-dcache-load-misses

Benchmark_col has the highest count LLC-loads

Benchmarl_blocked has the highest count of LLC-load-misses

The reason Benchmark_col has so many misses and loads is because it does not take advantage of the ways are stored in memory and has to jump around; meaning the cache rarely has what it needs.

Benchmark_blocked benefits from its optimizations as shown by the low count of misses and loads

```
void square_dgemm(int N, double A[N][N], double B[N][N], double C[N][N])
{
    int i, j, k;
    for (k = 0; k < N; k++) {
```

```
      for (i = 0; i < N; i++) {
         double tmp = A[i][k];
         for (j = 0; j < N; j ++) {
            C[i][j] += tmp * B[k][j]; ******
         }
      }
   }
}
```

****** -> this line is the source of all the cache misses because there is not enough room to hold the entire matrix

This code pulls segments of the matrix and runs the operations of them. When it is done, it loads in the next section based on the cache blocks.

```
void dgebb_subblock_opt(int bk,
          int Astride, double A[][Astride],
               int Bstride, double B[][Bstride],
              int Cstride, double C[][Cstride])
{
   double a, blocal[RJ], clocal[RI][RJ];
   int i, j, k;
   for (i = 0; i < RI; i++)
      for (j = 0; j < RJ; j++)
         clocal[i][j] = C[i][j];

   for (k = 0; k < bk; k++) {
      for (j = 0; j < RJ; j++) {
          blocal[j] = B[k][j];
      }
      for (i = 0; i < RI; i++) {
```

```
            a = A[i][k];
            for (j = 0; j < RJ; j++) {
                clocal[i][j] = clocal[i][j] + a * blocal[j]; ******
            }
        }
    }
    for (i = 0; i < RI; i++) {
        for (j = 0; j < RJ; j++) {
            C[i][j] = clocal[i][j];
        }
    }
}
```

****** -> this line is the source of most of the events because every time clocal[i][j] changes itself it must put itself back into the cache