Original code from assessment.

```
dhow@compy:~/repos/Optimization$ gcc main.c -o Optimization -lm
main.c:44:1: warning: return type defaults to ‘int’ [-Wimplicit-int]
   44 | main()
      | ^~~~

dhow@compy:~/repos/Optimization$ time ./Optimization

real    3m42.048s
user    3m41.750s
sys     0m0.084s
```

Same code with fancier compile flags.

```
dhow@compy:~/repos/Optimization$ gcc -O3  main.c -o Optimization -lm
main.c:44:1: warning: return type defaults to ‘int’ [-Wimplicit-int]
   44 | main()
      | ^~~~
dhow@compy:~/repos/Optimization$ time ./Optimization

real    1m54.287s
user    1m54.169s
sys     0m0.040s
```

Time to beat ~2min

Initial profile data before changes,

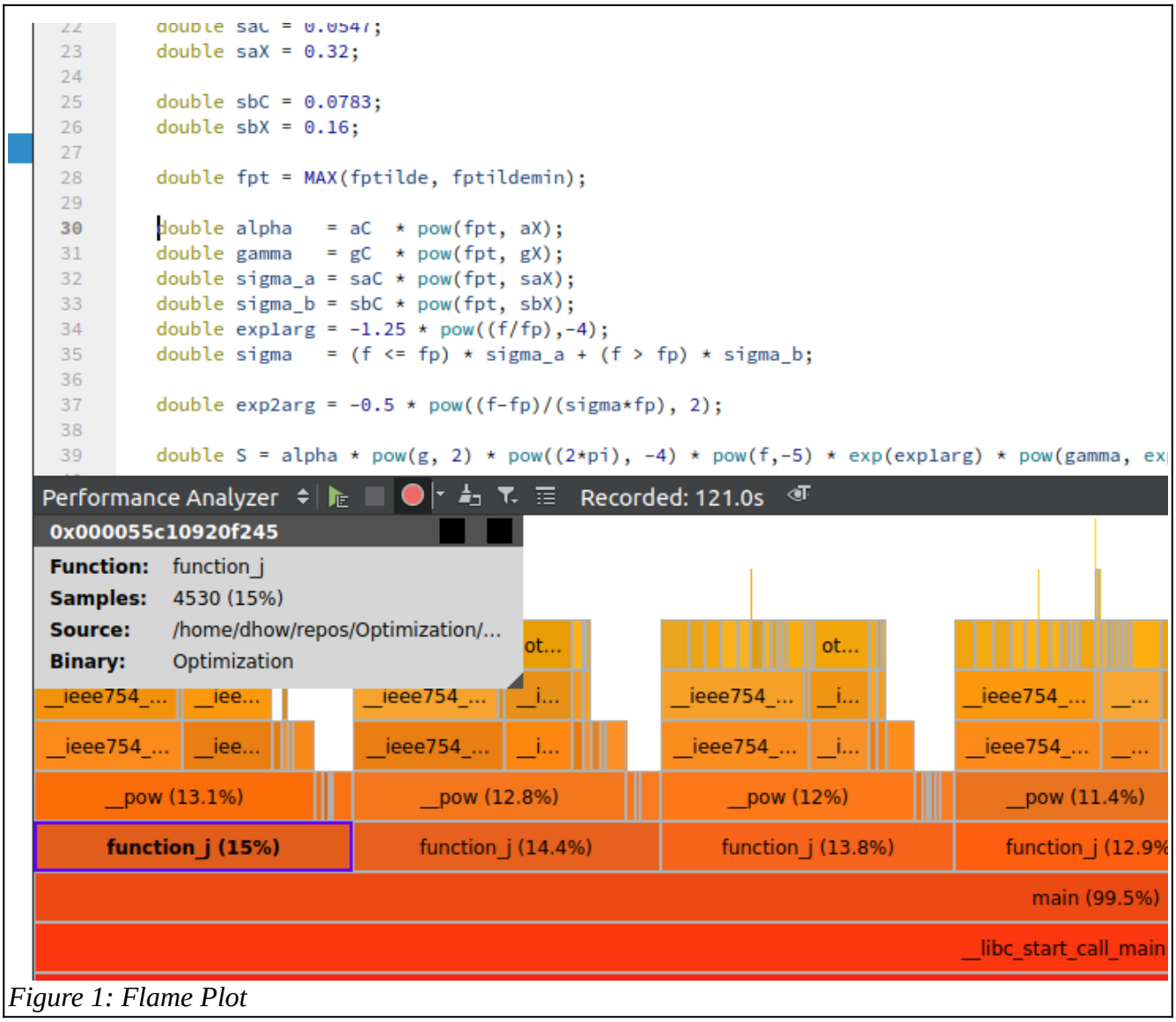


Figure 1: Flame Plot

__pow (Power) function is root of all troubles.

To reduce turnaround time and build a ground truth file of output for pre-optimized data, modify loop step to be settable from command line:

```
for (f = -5.; f <= 5.; f += step) {
 for (fp = 0.; fp <= 10.; fp += step) {
   for (fptilde = 0.; fptilde <= 10.; fptilde += step)
```

Default subsampling step size set to: step = 0.1
results in total calls to function_j of 1030301 which gives a representative sample of full data field, but can be profiled much faster.

The line:

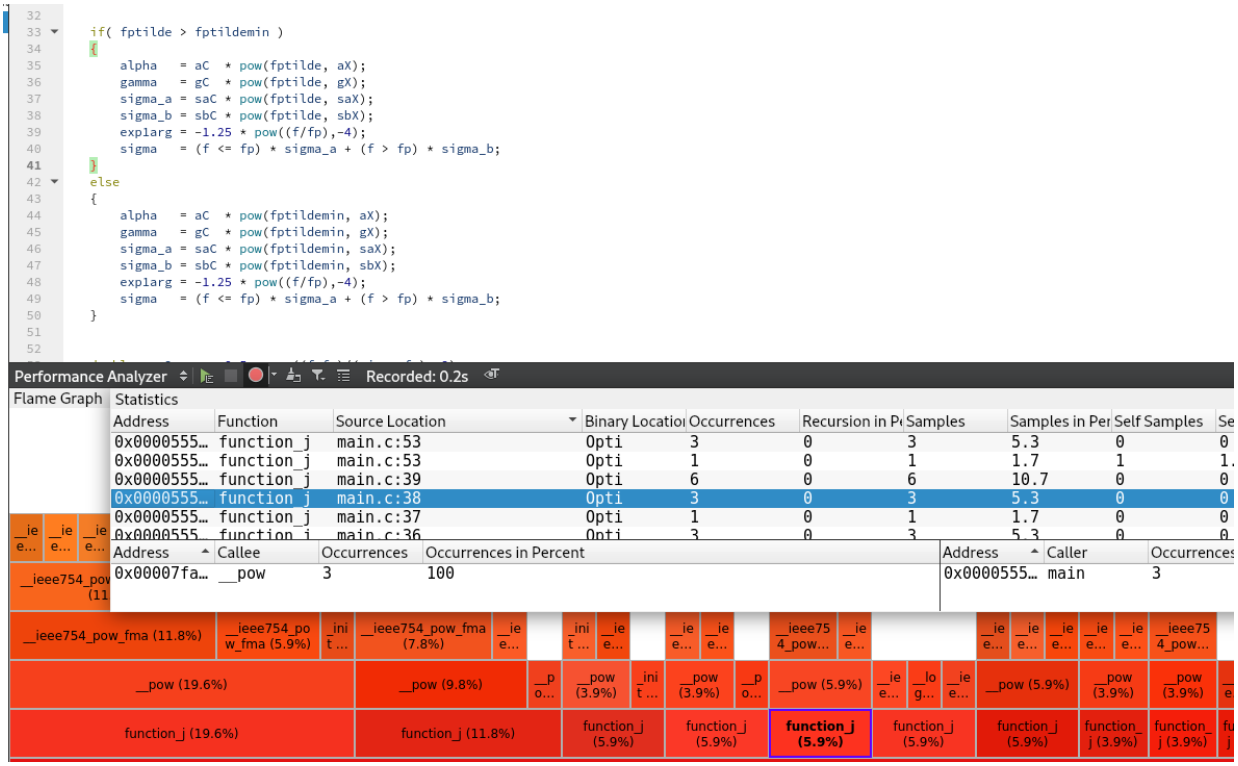
```
double fpt = MAX(fptilde, fptildemin);
```

creates two code paths, one of which (fptildemin > fptilde) is entirely dependent on known compile time constants. This would seem to be a worthwhile area to optimize (and -O3 might well automatically optimize it out), but it ought to be measured before investing effort.

Splitting single line MAX expression into conventional if ... else clauses allows profiler to measure each branch:

```
if( fptilde > fptildemin )
{
    alpha   = aC  * pow(fptilde, aX);
    gamma   = gC  * pow(fptilde, gX);
    sigma_a = saC * pow(fptilde, saX);
    sigma_b = sbC * pow(fptilde, sbX);
    exp1arg = -1.25 * pow((f/fp),-4);
    sigma   = (f <= fp) * sigma_a + (f > fp) * sigma_b;
}
else
{
    alpha   = aC  * pow(fptildemin, aX);
    gamma   = gC  * pow(fptildemin, gX);
    sigma_a = saC * pow(fptildemin, saX);
    sigma_b = sbC * pow(fptildemin, sbX);
    exp1arg = -1.25 * pow((f/fp),-4);
    sigma   = (f <= fp) * sigma_a + (f > fp) * sigma_b;
}
```

Calls in first if .. clause are 2nd,3rd , and 5th most costly __pow reference



While those in else clause lines 44-49 are not sampled enough to appear in this graph. This is due to the small fraction of times fptilde is less than fptildemin.

Optimizing computation or reducing number of times called for alpha, gamma, and sigma's in more common case holds more potential for improvement.

This sub sampled version of the objective function can save a text file of result values to be compared later to more optimized versions to be sure faster versions produce the same output. This is archived in the file smalltruth.txt.

The non-optimized, not printing program times at around .125 second:

```
dhow@compy:~/repos/Opti$ time ./Opti
real    0m0.131s
user    0m0.126s
sys     0m0.004s
dhow@compy:~/repos/Opti$ time ./Opti

real    0m0.125s
user    0m0.121s
sys     0m0.004s
```

Refactoring to create a structure to hold state and create named functions for different parts of application. Commit: ed329006d95700. This is designed to allow prevent recalculations of intermediate results whose underlying parameters haven't changed, and ultimately to facilitate multi-threaded implementation.

```
#ifndef JSTATE_H
#define JSTATE_H
typedef struct
{
    double f, fp, fptilde;

    bool f_lte_fp;

    double alpha, gamma, sigma;

    double exp1arg, exp2arg;
} JState;

double func_j( JState *state );

void update_exp1(JState *state);

void update_exp2(JState *state);

void update_sigma(JState *state);

void update_ag(JState *state);

void set_parameters( JState *state,  double f, double fp, double fptilde );

void init_state( JState *state, double f, double fp, double fptilde );

#endif // JSTATE_H
```

Comparing output of refactored version with original downsampled output yields no differences at all.

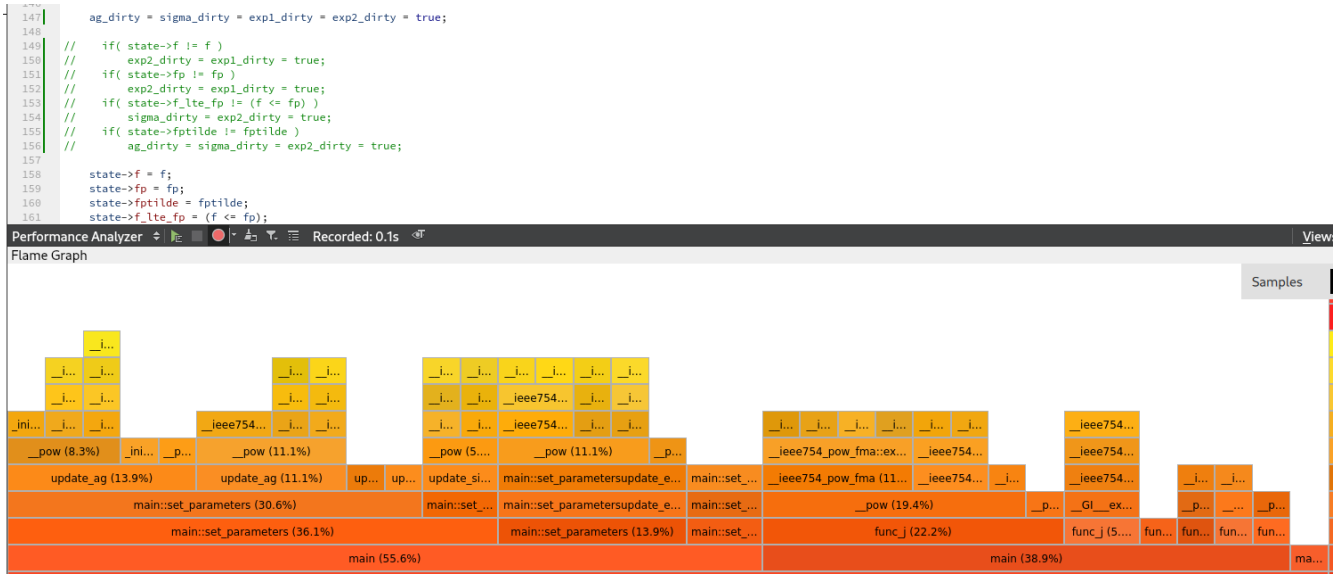
```
dhow@compy:~/repos/Opti$ ./Opti > smalltest
dhow@compy:~/repos/Opti$ diff smalltest smalltruth.txt
```

Speed is similar to original when recalculations checks are turned off.

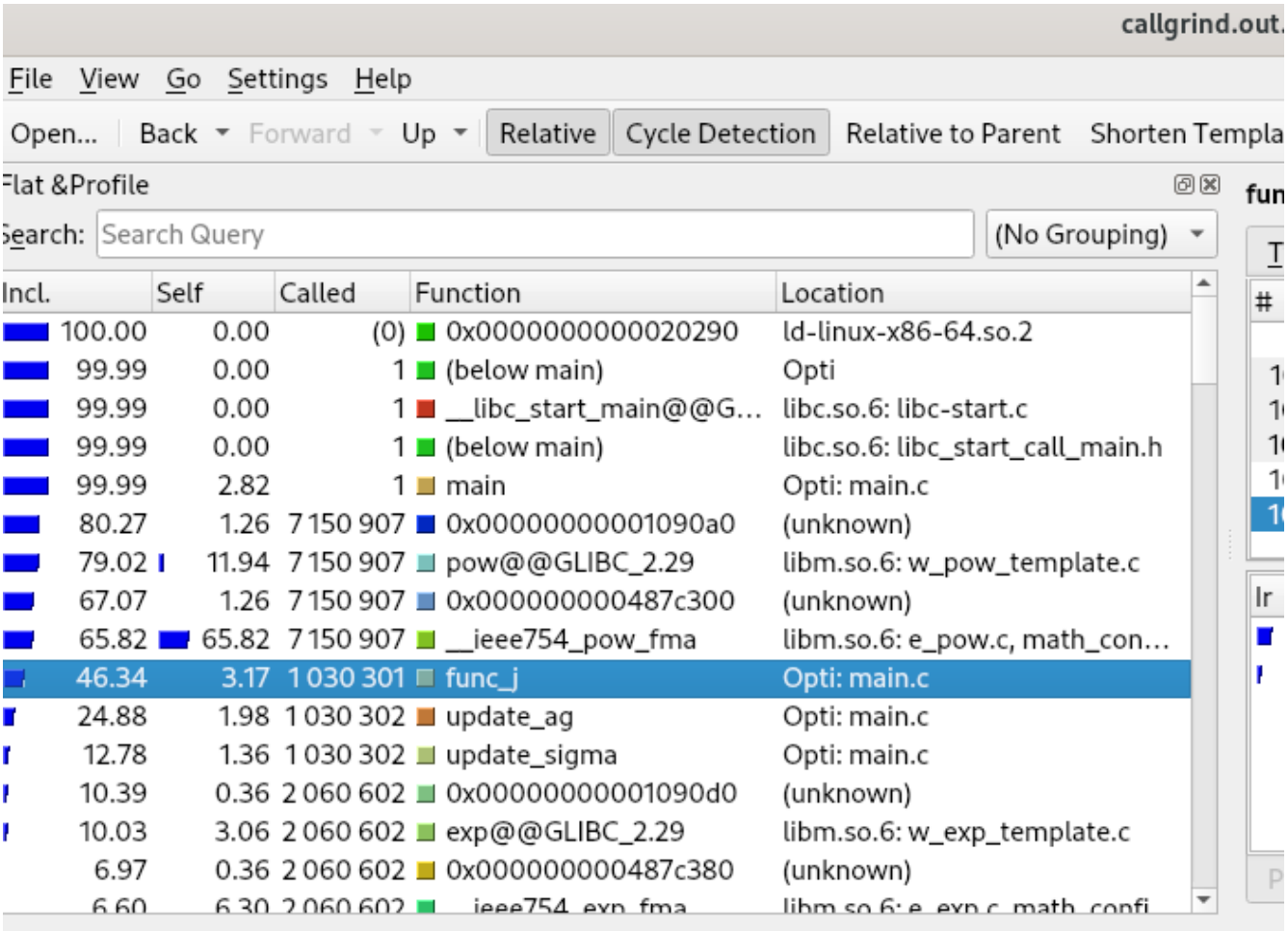
```
void set_parameters( JState *state,  double f, double fp, double
fptilde )

{
    bool ag_dirty, sigma_dirty, exp1_dirty, exp2_dirty;

    ag_dirty = sigma_dirty = exp1_dirty = exp2_dirty = true;

//    if( state->f != f )
//        exp2_dirty = exp1_dirty = true;
//    if( state->fp != fp )
//        exp2_dirty = exp1_dirty = true;
//    if( state->f_lte_fp != (f <= fp) )
//        sigma_dirty = exp2_dirty = true;
//    if( state->fptilde != fptilde )
//        ag_dirty = sigma_dirty = exp2_dirty = true;

    state->f = f;
    state->fp = fp;
    state->fptilde = fptilde;
    state->f_lte_fp = (f <= fp);

    if(ag_dirty)
        update_ag(state);
    if(exp1_dirty)
        update_exp1(state);
    if(sigma_dirty)
        update_sigma(state);
    if(exp2_dirty)
        update_exp2(state);
}
```

```
dhow@compy:~/repos/Opti$ time ./Opti

real    0m0.122s
user    0m0.118s
sys     0m0.004s
dhow@compy:~/repos/Opti$ time ./Opti

real    0m0.123s
user    0m0.122s
sys     0m0.000s
```

We can see the largest time sectionin flame graph is parameter setting:



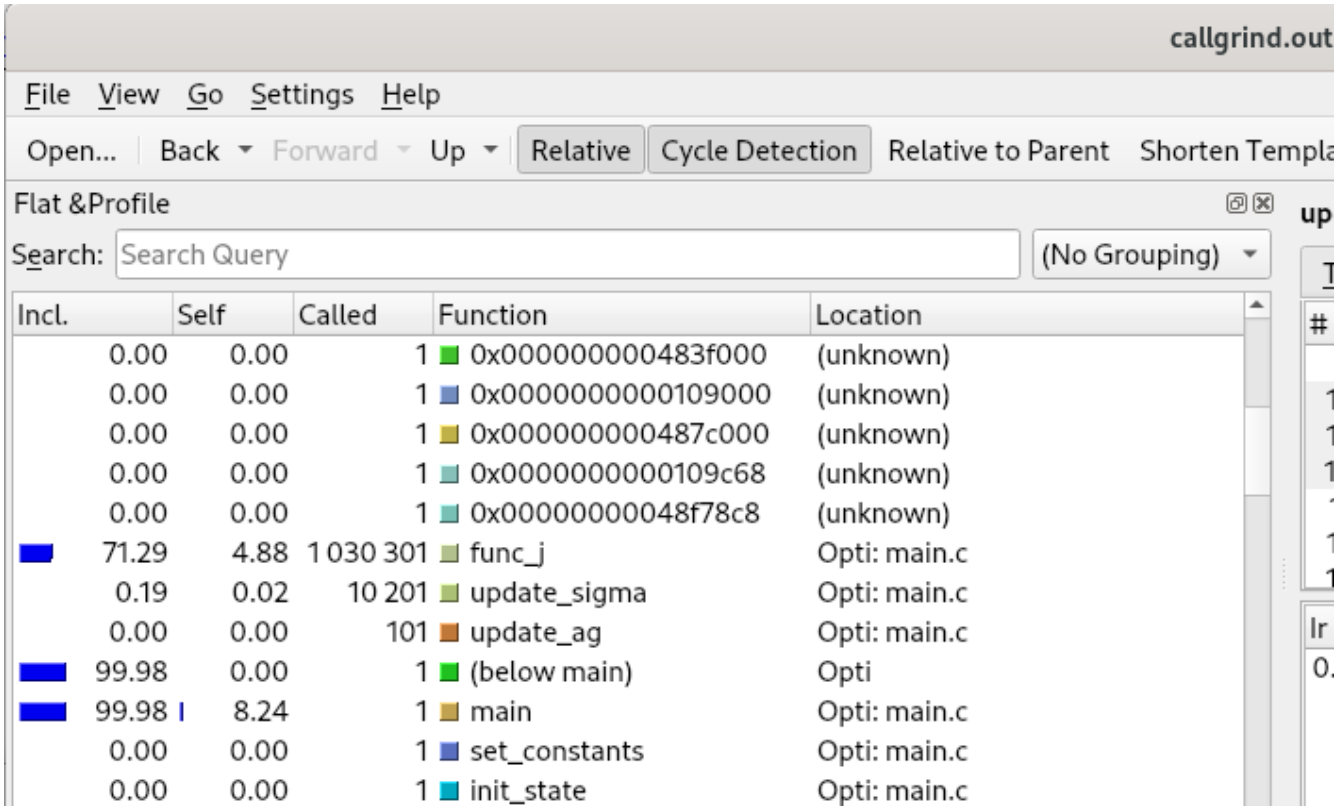
Looking at Calgrind output we see 1030301 calls to each parameter set subroutine as well as the final ultimate function. (one for each sample point in reduced set)



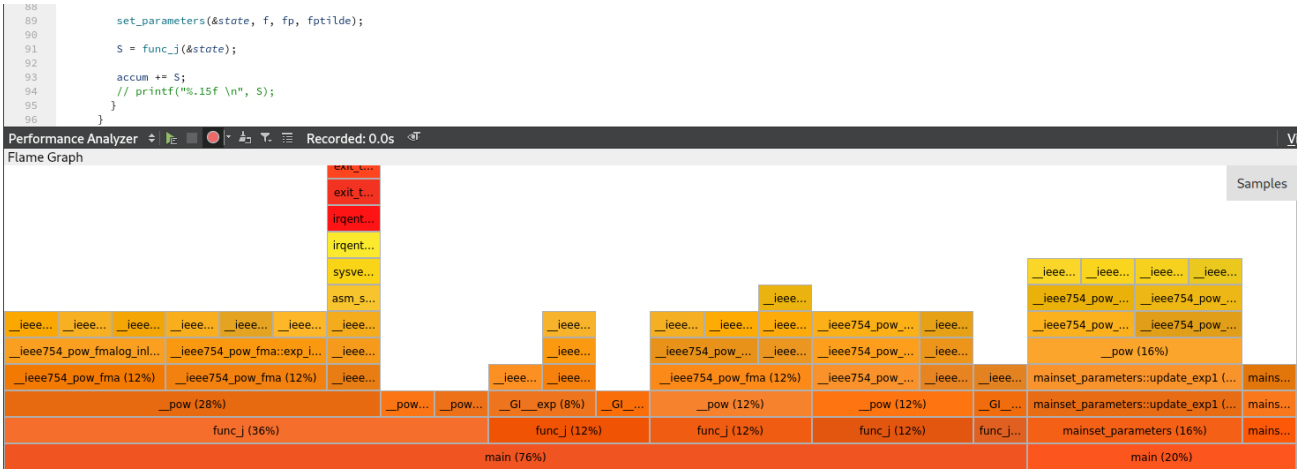
If we uncomment checking to prevent resetting of parameters and rearrange loops so the expensive fptilde parameter changes most slowly we see this improvement (just less than 2x):

```
dhow@compy:~/repos/Opti$ time ./Opti

real    0m0.079s
user    0m0.078s
sys     0m0.000s
dhow@compy:~/repos/Opti$ time ./Opti

real    0m0.077s
user    0m0.077s
sys     0m0.000s
```

We can see calls to methods triggered by updates of fptilde are much lower than number of sample points now, but order of output values is permuted and will no longer match test data file exactly.



Flame plot shows parameter setting is no longer top time sink.



To resore order and match output we create a cache of fptilde triggered parameters.

Commit: b0b9cd4811e.

Speed improves ~ another 2x.

```
dhow@compy:~/repos/Opti$ time ./Opti

real    0m0.029s
user    0m0.029s
sys     0m0.000s
dhow@compy:~/repos/Opti$ time ./Opti

real    0m0.030s
user    0m0.029s
sys     0m0.000s
dhow@compy:~/repos/Opti$ time ./Opti

real    0m0.029s
user    0m0.029s
sys     0m0.000s
```

But output is no longer exactly the same:

```
dhow@compy:~/repos/Opti$ ./Opti > smalltest
dhow@compy:~/repos/Opti$ diff smalltest smalltruth.txt | head
317026c317026
< -0.000332530223660
---
> -0.000332530223661
408886c408886
< -0.002835289984435
---
> -0.002835289984436
418963c418963
< -0.002631329959615
```

Shown changes are very small however. Try a quantitative comparison that takes this into account. Compute and sort in reverse order all the relative differences divided by target value:

```
dhow@compy:~/repos/Opti$ ./Opti > smalltest
dhow@compy:~/repos/Opti$ paste smalltruth.txt smalltest  | awk &apos;{ print ($1!
=0) ? ($1-$2)/$1: ($1-$2)/0.0000001; }&apos; | awk &apos;{ print
sqrt($1*$1); }&apos; | sort -k1 -gur | head
3.00712e-12
3.79897e-13
3.52722e-13
8.84412e-14
7.2343e-14
6.75741e-14
4.69306e-14
4.6243e-14
2.97012e-14
2.85492e-14
```

The worst case error is ~ 0.00000000003%

Now that the parameter setting is reduced to second most time consuming attention can be paid to function_j itself. Parameter called exp2arg is only ever referenced as exp(exp2arg) so create a cached value of this as exp_exp2arg. Also several pow() function with constant floating point exponents can be converted to logarithm and only combined in the end with a single exp() call. Pow calls with integer exponents are converted to repeated multiplications. (Also smarter more flexible make file)

```
dhow@compy:~/repos/Opti$ make release
dhow@compy:~/repos/Opti$ time release/Optimization

real    0m0.022s
user    0m0.022s
sys     0m0.000s
dhow@compy:~/repos/Opti$ time release/Optimization

real    0m0.017s
user    0m0.017s
sys     0m0.000s
dhow@compy:~/repos/Opti$ time release/Optimization

real    0m0.022s
user    0m0.022s
sys     0m0.000s
```

Concerning data matching:

```
dhow@compy:~/repos/Opti$ make test

gcc -c  -DNDEBUG -DTEST  -O3 -I/usr/lib/x86_64-linux-gnu/glib-2.0/include
-I/usr/include/glib-2.0 -o test/main.o main.c
gcc  -DNDEBUG -DTEST  -O3 -o test/Optimization  test/main.o -lm -lglib-2.0
dhow@compy:~/repos/Opti$ test/Optimization > smalltest
```

```
dhow@compy:~/repos/Opti$ paste smalltruth.txt smalltest  | awk &apos;{ print ($1!
=0) ? ($1-$2)/$1: ($1-$2)/0.0000001; }&apos; | awk &apos;{ print
sqrt($1*$1); }&apos; | sort -k1 -gur  | head
2.17557e-11
1.69499e-11
6.0238e-12
5.01562e-12
3.12075e-12
2.12032e-12
7.85469e-13
7.56412e-13
6.08076e-13
4.93334e-13
```

Slightly worse but still ~ 0.000000002% error.

Now restore full resolution through command line parameter:

```
dhow@compy:~/repos/Opti$ make release

gcc -c  -DNDEBUG -O3 -I/usr/lib/x86_64-linux-gnu/glib-2.0/include
-I/usr/include/glib-2.0 -o release/main.o main.c
gcc  -DNDEBUG -O3 -o release/Optimization  release/main.o -lm -lglib-2.0
dhow@compy:~/repos/Opti$ time release/Optimization 0.01

real    0m14.792s
user    0m14.722s
sys     0m0.024s
dhow@compy:~/repos/Opti$ time release/Optimization 0.01

real    0m14.719s
user    0m14.719s
sys     0m0.000s
```

Faster by 7.6x

Rebuild with higher optimization flag:

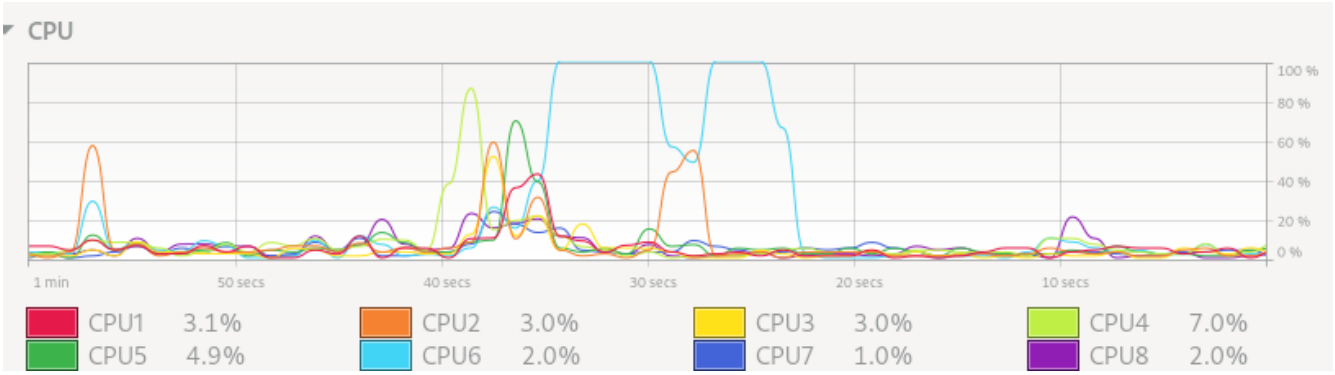
```
dhow@compy:~/repos/Opti$ gcc -c  -DNDEBUG -Ofast -I/usr/lib/x86_64-linux-gnu/glib-
2.0/include -I/usr/include/glib-2.0 -o release/main.o main.c

dhow@compy:~/repos/Opti$ gcc  -DNDEBUG -Ofast -o release/Optimization
release/main.o -lm -lglib-2.0
dhow@compy:~/repos/Opti$ time release/Optimization 0.01

real    0m12.168s
user    0m12.167s
sys     0m0.000s
dhow@compy:~/repos/Opti$ time release/Optimization 0.01

real    0m12.667s
user    0m12.657s
sys     0m0.000s
```

About 9x

Rechecking accuracy:

```
dhow@compy:~/repos/Opti$ gcc -c  -DNDEBUG -DTEST  -Ofast -I/usr/lib/x86_64-linux-
gnu/glib-2.0/include -I/usr/include/glib-2.0 -o test/main.o main.c

dhow@compy:~/repos/Opti$ gcc  -DNDEBUG -DTEST  -Ofast -o test/Optimization
test/main.o -lm -lglib-2.0
dhow@compy:~/repos/Opti$ test/Optimization > smalltest
dhow@compy:~/repos/Opti$                                  (reverse-i-
search)`awk&apos;: paste smalltruth.tx2)/0.0000001; }&apos; | awk &apos;{ print
sqrt($1*$1); }&apos; | sort -k1 -gur  | head
2.17557e-11
1.69499e-11
6.0238e-12
5.01562e-12
3.12075e-12
2.12032e-12
7.85469e-13
7.56412e-13
7.36774e-13
6.08076e-13
```

In all these trials on net no more than one cpu is used:



We can use state structure to recast algorithm to a parallel one with a thread pool:
(threadpool branch, commit b0c279ccc3caf9)

```
dhow@compy:~/repos/Opti$ make release
gcc -c  -DNDEBUG -O3 -I/usr/lib/x86_64-linux-gnu/glib-2.0/include
-I/usr/include/glib-2.0 -o release/main.o main.c
gcc -c  -DNDEBUG -O3 -I/usr/lib/x86_64-linux-gnu/glib-2.0/include
-I/usr/include/glib-2.0 -o release/thpool.o thpool.c
gcc  -DNDEBUG -O3 -o release/Optimization  release/main.o release/thpool.o -lm -
lglib-2.0
dhow@compy:~/repos/Opti$ time release/Optimization 0.01

real    0m3.474s
user    0m25.750s
sys     0m0.024s
dhow@compy:~/repos/Opti$ time release/Optimization 0.01

real    0m3.438s
user    0m25.867s
```

Using -Ofast

```
dhow@compy:~/repos/Opti$ gcc  -DNDEBUG -O3 -o release/Optimization  release/main.o
release/thpool.o -lm -lglib-2.0

dhow@compy:~/repos/Opti$ gcc  -DNDEBUG -Ofast -o release/Optimization
release/main.o release/thpool.o -lm -lglib-2.0
dhow@compy:~/repos/Opti$ time release/Optimization 0.01

real    0m2.986s
user    0m22.218s
sys     0m0.012s
dhow@compy:~/repos/Opti$ time release/Optimization 0.01

real    0m2.941s
user    0m22.108s
sys     0m0.000s
dhow@compy:~/repos/Opti$
```

With thread pool all CPUs are soaked for full 3 seconds needed to run junction on full res array.