Lecture 11: Pipelining, Hyperthreading, and Compiler Optimisation

Big Picture

- Remember in the ”brief history” lecture that there had been several early attempts to write high level languages
- They were resisted because the average human assembly programmer could write faster code than that produced by a high level language.
- Even though high level languages were much easier to read, it wasn’t until FORTRAN came along and could produce code as fast as that the average human programmer could write that high level languages took off.
- Compiler developers have had 60 years to improve their code generation.
- “We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil.” - Sir Tony Hoare (Inventor of QuickSort)
- This means before you optimize your code make sure you’re optimizing something that matters (i.e. find the bottlenecks and fix those).
- Assuming you are using an efficient algorithm (In the Big-O sense) it is rarely worth your time to try and optimize low level routines – the compiler will usually do a better job.

Big Picture

- OK given that the compiler will do a better job than most programmers in speeding up your code what are we going to talk about for the next 48 minutes?

- 1) You often have to tell the compiler to optimize your code.
- 2) You have to tell the compiler what you want to optimize (code size, compilation time, or execution speed)
- 3) The compiler doesn't always get it right. You have to test to make sure the optimisations you asked for worked.
- 4) Writing your code in a certain way can make the compilers job easier. Write code that is easy to optimize not optimized code
- 5) Identify bottlenecks can help you target compiler optimisations.
- 6) "Don't give up your performance accidently" – compilers can't compensate for carelessness.
- 7) Even if the compiler does it for you, it's still good for you to know what it's doing.

Pipelining

CPU instruction execution follows the Fetch, Decode, Execute, Write cycle. This is the code execution pipeline.

- **Fetch**: Get an instruction from memory.
- **Decode**: Figure out what the instruction is supposed to do (ADD, MULTIPLY, ETC) and potentially starts fetching operands from memory.
- **Execute**: CPU executes the decoded instruction on the operands.
- **Write**: Write the result of the instruction execution somewhere.

Pipelining

CPU instruction execution follows the Fetch, Decode, Execute, Write cycle.

- **Fetch**: Get an instruction from memory.
- **Decode**: Figure out what the instruction is supposed to do (ADD, MULTIPLY, ETC) and potentially starts fetching operands from memory.
- **Execute**: CPU executes the decoded instruction on the operands.
- **Write**: Write the result of the instruction execution somewhere.

How long do these things take? In CPU cycles (it's complex and the values below are generalities, some CPUs are designed to do multiplication in 1 cycle – but they have to make other sacrifices to do it.)

- **Fetch**: If the instruction is in L1 cache (fastest) then 3 CPU cycles, slower cache 12 cycles, if DDR RAM then more than 100 cycles.
- **Decode**: 1-2 cycles
- **Execute**: ADD 1-3, SUBTRACT 1-3, MULTIPLICATION 3-12, DIVISION 16-80 cycles.
- **Write**: Write the result of the instruction execution somewhere. Depends where.

Pipelining

So different operations take different amounts of time (DIVIDE could be 40x slower than ADD).

And some things take a REALLY long time. Fetching data from RAM for example (100 cycles). What about fetching from swap*?

When the CPU is waiting 100 cycles for data to arrive from RAM, those CPU cycles are waisted.

*Varys a lot but an average ballpark figure is 1,000,000 CPU cycles.

Pipelining

- When the CPU is waiting 100 cycles for data to arrive from RAM those CPU cycles are waisted.
- So instead, the CPU tries to fill those idle cycles with productive work.

- **Out of order execution.** Just because the programmer said to perform operations in a certain order doesn't mean they really have to be done in that order. Maybe there is no dependency between the operations and later operations can be performed while the CPU is idle.
- CDC6400 built in 1964 was the first machine to do this.
- The CPU looks at the sequence of operations waiting to run and picks the next instruction that's "ready".
- Meaning all the prerequisite operations are complete.

Hyperthreading

- When the CPU is waiting 100 cycles for data to arrive from RAM those CPU cycles are waisted.
- So instead, the CPU tries to fill those idle cycles with productive work.

- Out-of-order execution allows the CPU to fill idle cycles with other instructions from the same thread. (A thread is a program counter (PC) iterating through the instructions in a process.) But this isn't perfect and stalls or bubbles happen where no operation in the thread is ready.
- The Kernel can swap processes in and out of so they can use the CPU, but as we saw in the Kernel lecture the context switch is very expensive.
- CPUs support switching whole threads without a context switch. They remember the PC and register values for a different program and start executing that when there is a pipeline stall.
- This is called hyperthreading. It looks to the operating system like there is a whole additional CPU ready to execute a process. But really it is using the same CPU hardware.

First chip to implement hyperthreading (2002)

Sun Microsystems has the patent (1994)

Hyperthreading

- When the CPU is waiting 100 cycles for data to arrive from RAM those CPU cycles are waisted.
- So instead, the CPU tries to fill those idle cycles with productive work.

- Hyperthreading can improve CPU throughput by 0-50% depending on the program.
- But that means it uses the CPU more efficiently – not that the programs running on the CPU will necessarily run faster. After all they now have to share hardware.
- The amount of improvement for 1 socket is something like 30%, 2 sockets 15%, and for 4 or more sockets there are no rules of thumb.
- The more threads there are the more contention there is for accessing the cache and RAM and other resources, which can slow everything down.

First chip to implement hyperthreading (2002)

Sun Microsystems has the patent (1994)

Hyperthreading

```
[matthew@moonshine ~]$ lscpu
Architecture:            x86_64
  CPU op-mode(s):        32-bit, 64-bit
  Address sizes:         46 bits physical, 48 bits virtual
  Byte Order:            Little Endian
CPU(s):                  32
  On-line CPU(s) list:   0-31
Vendor ID:               GenuineIntel
  Model name:            Intel(R) Xeon(R) CPU E5-2670 0 @
2.60GHz
    CPU family:          6
    Model:               45
    Thread(s) per core:  2
    Core(s) per socket:  8
    Socket(s):           2
    Stepping:            7
    CPU max MHz:         3300.0000
    CPU min MHz:         1200.0000
    BogoMIPS:            5199.97
```

First chip to implement hyperthreading (2002)

Sun Microsystems has the patent (1994)

Identify whether hyperthreading is enabled. This is something you can turn on and off in the BIOS, so you may or may not have it enabled.

Hyperthreading

Login to Wheeler, Hopper, or Xena.

Is hyperthreading (HT) enabled?

In HPC Clusters Hyperthreading is often disabled because users don't like to think their code is sharing the CPU with someone else's program.

Even if it makes the CPU more efficient it might not speed up their particular program.

There is also the idea that HPC software will already be so efficient that there is no point the HT and it just false advertising to make it look like there are 2 (virtual) processors.

First chip to implement hyperthreading (2002)

Sun Microsystems has the patent (1994)

Compiler Optimisation: Strength Reduction

- Motivation: Not all CPU operations cost the same.
- Solution: The compiler rewrites your code to use the cheaper (called weaker) operation.

So, for example,

`4*x` could be replaced with with `(x+x)+(x+x)`

(3-12 cycles) versus (1-3 cycles x 3 = 3-9 cycles.)

Hoisting

Motivation: Don't recompute values more than once
Solution: "hoist" loop invariants out of the loop

Eg.

```
int x = 0, y = 1, z = 2;
for (int i = 0; i < 100; i++)
        a[i] = i*x + y/z;
```

```
int x = 0, y = 1, z = 2;
int a = y/z;
for (int i = 0; i < 100; i++)
        a[i] = i*x + a;
```

The expression y/z is loop invariant, i.e. it doesn't change from one iteration to the next. So hoist it out.

Precalculate constants

- Motivation: Constant values are much easier to optimize because it simplifies the code and makes it predictable.
- Solution: replace calculations with results right away

```
int x = 0, y = 1, z = 2;
int a = y/z;
for (int i = 0; i < 100; i++)
        a[i] = i*x + a;
```

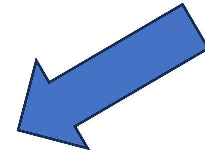
```
int x = 0;
int a = 1/2;
for (int i = 0; i < 100; i++)
        a[i] = i*x + a;
```

Precalculate constants

- Motivation: Constant values are much easier to optimize because it simplifies the code and makes it predictable.
- Solution: replace calculations with results right away

```
int x = 0, y = 1, z = 2;
int a = y/z;
for (int i = 0; i < 100; i++)
        a[i] = i*x + a;
```

```
int x = 0;
int a = 1/2;
for (int i = 0; i < 100; i++)
        a[i] = i*x + a;
```



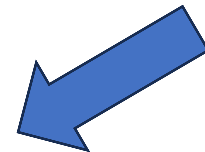
```
int x = 0;
int a = 0;
for (int i = 0; i < 100; i++)
        a[i] = i*x + 0;
```

Precalculate constants

- Motivation: Constant values are much easier to optimize because it simplifies the code and makes it predictable.
- Solution: replace calculations with results right away

```
int x = 0, y = 1, z = 2;
int a = y/z;
for (int i = 0; i < 100; i++)
        a[i] = i*x + a;
```

```
int x = 0;
int a = 1/2;
for (int i = 0; i < 100; i++)
        a[i] = i*x + a;
```



```
int x = 0;
int a = 0;
for (int i = 0; i < 100; i++)
        a[i] = i*x + 0;
```

```
int x = 0;
for (int i = 0; i < 100; i++)
        a[i] = i*x;
```

Loop Unrolling

- Motivation. CPU Pipelining works well when the CPU can predict what is coming next. Everytime there is a logic branch, that gets harder because a choice is being made.
- Solution: Reduce the number of choices in the code.

```
cin >> y;
int array[4*y] = {0}
for (i = 0; i < 2*y; i++)
        array[i] = i;
```

```
cin >> y;
int array[4*y] = {0}
for (i = 0; i < 2*y; i+=2) {
        array[i] = i;
        array[i+1] = i+1; }
```

The number of choices is reduced from 2y to y. We also do half as many i < 2y? checks.

Loop Splitting

- Motivation: Pipelining does well when there are fewer dependencies between parts of the code, so it can do out-of-order execution.
- Solution: Split loops to remove dependencies.

```
int i = 10;
for (int j = 0; j < 10; j++)
{
        if ( i < 5)
                y[j] = x[i];
        else
                y[j] = x[j];
        i = j;
}
```

```
int i = 10;
for (int j = 0; j < 5; j++)
{
        y[j] = x[i];
        i = j;
}

for (int j = 5; j < 10; j++)
        y[j] = x[j];
```

The second loop doesn't depend on i anymore and so can be executed out-of-order.

Loop Splitting (peeling)

- Motivation: Pipelining does well when there are fewer dependencies between parts of the code, so it can do out-of-order execution.
- Solution: Split loops to remove dependencies.

```
int i = 5;
for (int j = 0; j < 5; j++)
{
        y[j] = x[i];
        i = j;
}
```

```
y[0] = x[5]
for (int j = 0; j < 5; j++)
{
        y[i-1] = x[i];
}
```

In this case we removed the dependency on i inside the loop entirely.

Inlining

- Motivation: Functions make code easier to read and understand, but calling a function is expensive (has to be put on the call stack, etc).
- Solution: Replace frequently used functions with a copy of the code in the function.

```
int add(int a, b) { return a + b; }

int x = 2;
for (int i = 0; i < 10e9; i++){
        array[i] = add(i, x);
}
```

```
int x = 2;
for (int i = 0; i < 10e9; i++)
        array[i] = i+x;
```

I'll ask you to recall this one
the end of class.

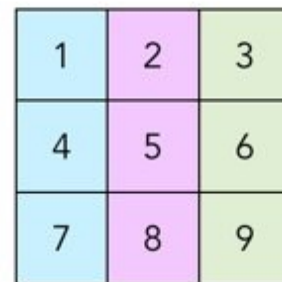
Cache Awareness (loop indexing order)

```
int array2d[3][3] = {{1,2,3},{4,5,6},{7,8.9}}
integer :: array(3,3) = reshape([(i, i=1,9)], [3,3])
```

- Motivation: When accessing a 2D array, it is actually layed out as a single 1D array in memory. When looping over the 2D array, accessing memory that is near the memory you accessed in the last iteration is much faster than memory that is far away. (Nearby memory will likely have been put in the cache when the first element was read.)
- Solution: you have to know how your compiler lays out 2D arrays in memory. Does it use row-major or column major layouts. C, C++, and Python are row major. FORTRAN, MATLAB, and Julia are column major.

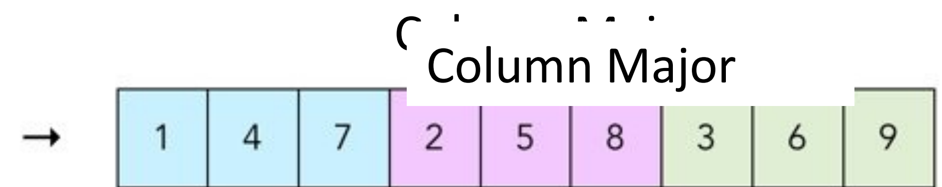
→

Row Major



→

Column Major



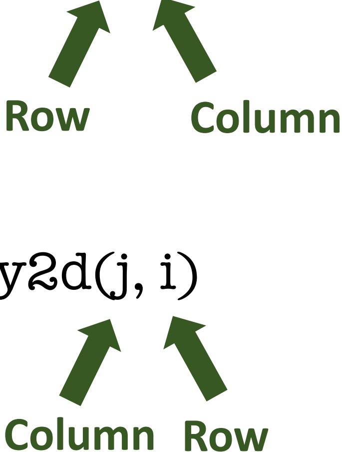
In RAM

Conceptually

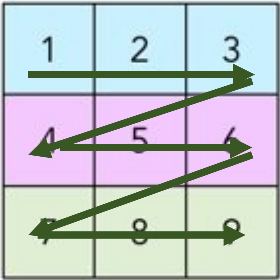
Cache Awareness (loop indexing order)

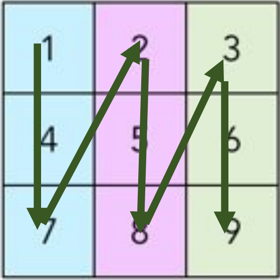
```
int array2d[3][3] = {{1,2,3},{4,5,6},{7,8.9}}
integer :: array(3,3) = reshape([(i, i=1,9)], [3,3])
```

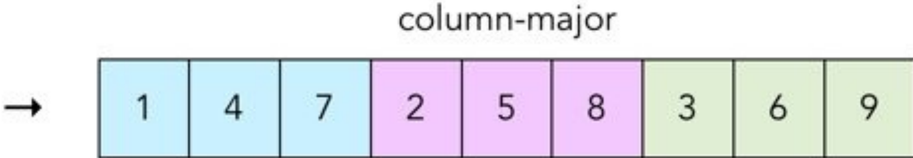
```
for (int i = 0; i < 3; i++)
  for (int j = 0; j < 3; j++)
    printf(array2d[i][j]);
```




```
do i = 1, 3
  do j = 1, 3
    print *, array2d(j, i)
  end do
end do
```







Conceptually

In RAM

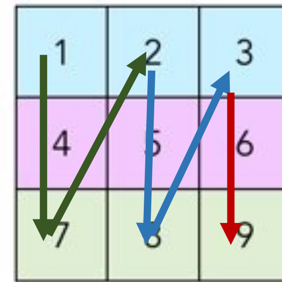
The double loops access the closest memory locations on consecutive iterations. Fast.

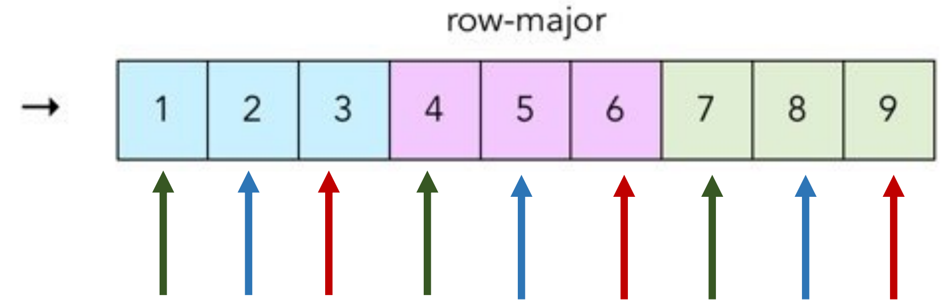
Cache Awareness (loop indexing order)

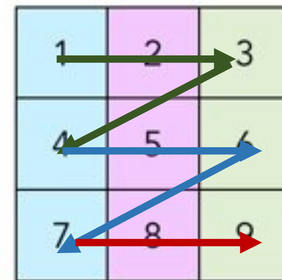
```
int array2d[3][3] = {{1,2,3},{4,5,6},{7,8.9}}
integer :: array(3,3) = reshape([(i, i=1,9)], [3,3])
```

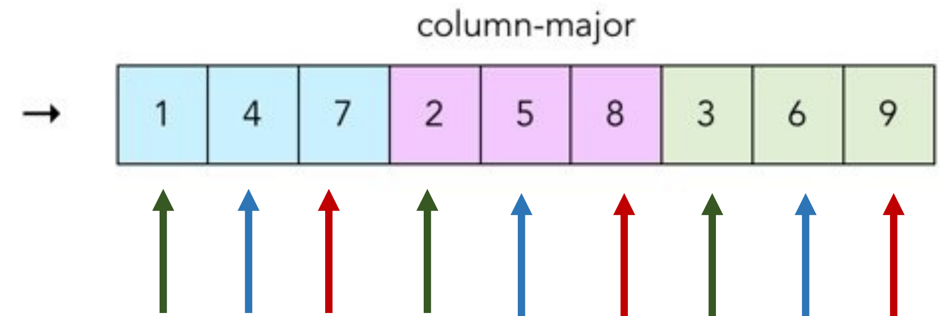
```
for (int i = 0; i < 3; i++)
  for (int j = 0; j < 3; j++)
    printf(array2d[j][i]);
```

```
do i = 1, 3
  do j = 1, 3
    print *, array2d(i, j)
  end do
end do
```









Conceptually

In RAM

The double loops access the memory locations far from each other on consecutive iterations. Slow.

```
[matthew@moonshine ~]$ git clone https://github.com/gmfricke/cache_example.git
Cloning into 'cache_example'...
remote: Enumerating objects: 6, done.
remote: Counting objects: 100% (6/6), done.
remote: Compressing objects: 100% (5/5), done.
remote: Total 6 (delta 0), reused 3 (delta 0), pack-reused 0
Receiving objects: 100% (6/6), done.
[matthew@moonshine ~]$ cd cache_example/
```

```
float averageMatRowMajor(int** mat, int n){
  int i, j, total=0;
  for (i = 0; i < n; i++){
    for (j = 0; j < n; j++){
      total+= mat[i][j];
    }
  }
  return (float)total/(n*n);
}
```

```
float averageMatColumnMajor(int** mat, int n){
  int i, j, total=0;
  for (i = 0; i < n; i++){
    for (j = 0; j < n; j++){
      total+= mat[j][i];
    }
  }
  return (float)total/(n*n);
}
```

```
[matthew@moonshine cache_example]$ gcc cachex.c -o gnu_cachex
[matthew@moonshine cache_example]$ ./gnu_cachex 10000
Row major average is: 7.56; time is 228084773 nanoseconds
Column major average is: 7.56; time is 1259461051 nanoseconds

Row major indexing was 7.557427 times faster than column major indexing.
[matthew@moonshine cache_example]$
```

This is a C program. The reverse will be true for a FORTRAN or Julia program.

Loop Interchange

- Motivation: The order in which array elements are accessed matters.
- Solution: Rearrange loops to make sure the indexing matches row or column major arrays.

```
int size=10000;
int a[size][size];
  for (int j=0; y < size; i++)
    for (int i=0; x<size; j++)
      a[i][j]=i*j
```

```
int size=10000;
int a[size][size];
  for (int i=0; y < size; i++)
    for (int j=0; x<size; j++)
      a[i][j]=i*j
```

GCC Optimization Options

- Those compiler optimisations are some of the more common ones.
- The GNU C compiler has more than 200 optimizations like these.
- The optimisations are usually grouped into speed, space, and debugging categories:

1. -O0 no optimization
2. -O or –O1 the compiler reduces program execution time and program size, at the expense of taking longer to compile.
3. -O2 enables more optimisations, at the expense of compilation time.
4. -O3 everything in O2 but adds more aggressive optimisations such as loop unrolling that will increase the size of the compiled program.
5. -Ofast everything in O3 but violates some C standards to produce faster code.
6. -Os enables all O2 optimisations except those that often increase program size.
7. -Oz aggressively optimizes to reduce program size.
8. -Og optimized for debugging

GNU Compiler List Optimisations

```
$ gcc --help=optimizers
$ gcc -Q --help=optimizers

# To see the difference between optimizations we can use diff
# Diff is a useful linux command that compares two files and shows
the differences.
$ diff <(gcc -O0 -Q --help=optimizers) <(gcc -O1 -Q --help=optimizers)
$ diff <(gcc -Q --help=optimizers) <(gcc -O2 -Q --help=optimizers)
 (In the last two commands we use "<()" process substitution to
give the output of the two gcc commands a temporary file, since
diff wants to compare two files)
```

Clone Sample Code (matrix multiply and quicksort)

```
[matthew@moonshine ~]$ git clone https://github.com/gmfricke/gnu_compiler_optimisations.git
Cloning into 'gnu_compiler_optimisations'...
remote: Enumerating objects: 9, done.
remote: Counting objects: 100% (9/9), done.
remote: Compressing objects: 100% (8/8), done.
remote: Total 9 (delta 1), reused 9 (delta 1), pack-reused 0
Receiving objects: 100% (9/9), done.
Resolving deltas: 100% (1/1), done.
```

Compare Optimisation Levels

- Take 5 minutes, cd into each of the subdirectories O0, 01, O2, and O3 and compile the simple_matmul.c and quicksort.c code with the optimization level that matches the directory name.
- For example:

```
$ cd gnu_compiler_optimisations/
$ ls
O0   O1   O2   O3   prof
$ cd O0
```

Compare Optimisation Levels

- Take 5 minutes, cd into each of the subdirectories O0, 01, O2, and O3 and compile the simple_matmul.c and quicksort.c code with the optimization level that matches the directory name, and run the programs
- For example:

```
$ cd O0
$ gcc -O0 simple_matmul.c -o gnu_matmul
$ ./gnu_matmul 1500 1000 1000 500
 1500x1000 * 1000x500 took 4.388221 seconds
```

Compare Optimisation Levels

- Take 5 minutes, cd into each of the subdirectories O0, 01, O2, and O3 and compile the simple_matmul.c and quicksort.c code with the optimization level that matches the directory name, and run the programs
- For example:

```
$ gcc -O0 quicksort.c -o gnu_quicksort
$ ./gnu_quicksort 100000000
Sorting 1e+08 elements took 28.694275 seconds
```

That’s 8 zeros in the argument. It is the size of the array to sort.

Compiler Optimisations

- Did you see differences in the execution times?
- Were there always improvements in execution speed as you increased the optimization level?
- Did the changes in speed differ for the matrix multiplication program and quicksort?

Optimisation is all about finding the Performance Bottlenecks

- Profiling allows you to learn where your program spent its time and which functions called which other functions while it was executing. This information can show you which pieces of your program are slower than you expected, and might be candidates for rewriting to make your program execute faster. It can also tell you which functions are being called more or less often than you expected. This may help you spot bugs that had otherwise been unnoticed.

Procedure

- You must compile and link your program with profiling enabled.
- You must execute your program to generate a profile data file.
- You must run a profilers gprof to analyze the profile data.

Profiling with GNU Tools

```
$ cd ../prof
$ gcc --prof quicksort.c -o gnu_quicksort
$ ./gnu_quicksort 100000000
Sorting 1e+08 elements took 45.831943 seconds
```

Profiling with GNU Tools

```
$ cd ../prof
$ gcc --prof quicksort.c -o gnu_quicksort
$ ./gnu_quicksort 100000000
Sorting 1e+08 elements took 45.831943 seconds
$ ls
gmon.out  gnu_quicksort  quicksort.c  simple_matmul.c
```

Profiling with GNU Tools – You can see which functions use the most time.

```
$ gprof gnu_quicksort gmon.out


So about 14% of the time the program is in the swap
function.
[4]      14.3    3.06    0.00 1765232924    swap [4]
```

Profiling with GNU Tools
Let’s see how compiling with O2 changes things

```
$ gcc -O2 --prof quicksort.c -o gnu_quicksort
$ ./gnu_quicksort 100000000
Sorting 1e+08 elements took 11.085591 seconds
$ gprof gnu_quicksort gmon.out
```

How much is the swap function used now?

Notice the swap function has been optimized away.

Function calls are expensive, they make code easier to understand, but the compiled code doesn’t need to be easy to read. The swap function is very simple and called a lot, so it was removed by the compiler.

Anyone remember which compiler optimization removes functions?