

# Your optimizing compiler is not smart enough.

To hell with multiple recursions. Is there a gap in compiler theory?

Federico Bruzzone,<sup>1</sup> PhD Student

Milan, Italy – 24 November 2025

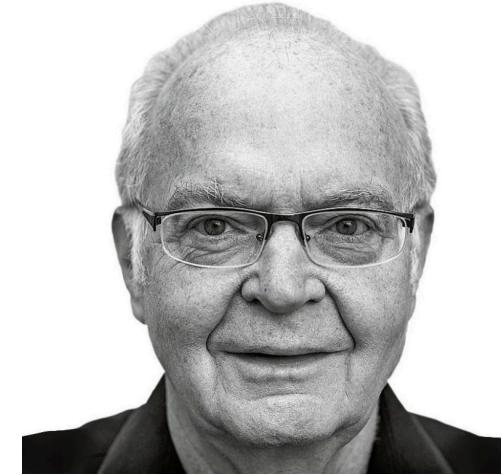
---

<sup>1</sup>ADAPT Lab – Università degli Studi di Milano,  
Website: [federicobruzzone.github.io](https://federicobruzzone.github.io),  
Github: [github.com/FedericoBruzzone](https://github.com/FedericoBruzzone),  
Email: [federico.bruzzone@unimi.it](mailto:federico.bruzzone@unimi.it)

# Premature Optimizations

Donald E. Knuth warned in 1974 about the dangers of **premature optimization** in programming [1]:

*We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%.*



In the absence of either empirically measured or theoretically justified performance issues, programmers should **avoid** making optimizations based **solely** on assumptions about potential performance gains.

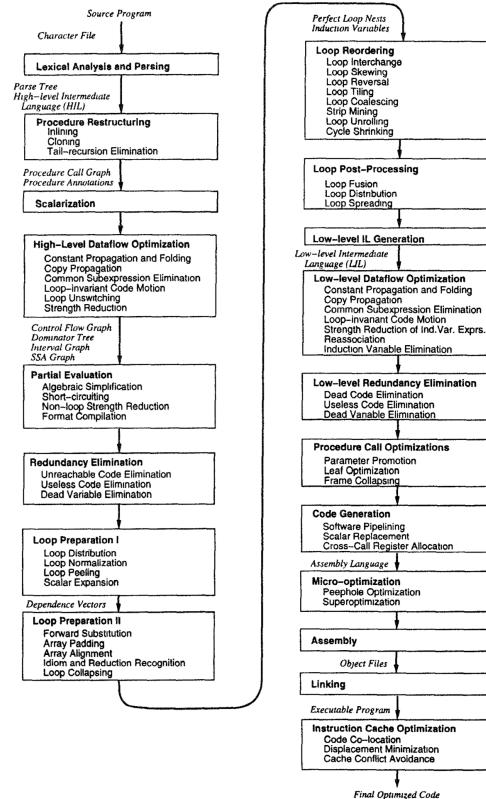
# Optimizing Compilers as Musical Compositions

Compilers are frequently perceived as intricate musical compositions—like the unfinished J. S. Bach's *Art of Fugue*—where mathematical precision and logical interplay guide each part.

Every module enters in perfect timing, weaving together a structure that only the keenest ears can fully grasp.

# Optimizing Compiler Infrastructure

TODO



From Bacon *et al.* [2]

# Peephole Optimizations

(widely studied since the 1960s [3], [4])

```
; x = x * 2
```

asm

```
LOAD R1, 0      ; load from 0
MUL R1, 2      ; multiply R1 by 2
STORE R1, 0     ; store R1 back to 0
```

# Peephole Optimizations

(widely studied since the 1960s [3], [4])

```
; x = x * 2                                asm
LOAD R1, 0        ; load from 0
MUL R1, 2        ; multiply R1 by 2
STORE R1, 0       ; store R1 back to 0
```

The optimized version replaces the multiplication by 2 with a **more efficient** binary shift operation.



```
LOAD R1, 0                                asm
SHL R1, 1        ; shift left by 1
STORE R1, 0
```

# Peephole Optimizations

(widely studied since the 1960s [3], [4])

```
; x = x * 2  
LOAD R1, 0      ; load from 0  
MUL R1, 2      ; multiply R1 by 2  
STORE R1, 0     ; store R1 back to 0
```

asm

```
; x = x + 0  
LOAD R1, 0  
ADD R1, 0      ; add 0 to R1  
STORE R1, 0
```

asm

The optimized version replaces the multiplication by 2 with a **more efficient** binary shift operation.



```
LOAD R1, 0  
SHL R1, 1      ; shift left by 1  
STORE R1, 0
```

asm

# Peephole Optimizations

(widely studied since the 1960s [3], [4])

```
; x = x * 2  
LOAD R1, 0      ; load from 0  
MUL R1, 2      ; multiply R1 by 2  
STORE R1, 0     ; store R1 back to 0
```

asm

The optimized version replaces the multiplication by 2 with a **more efficient** binary shift operation.



```
LOAD R1, 0  
SHL R1, 1      ; shift left by 1  
STORE R1, 0
```

asm

```
; x = x + 0  
LOAD R1, 0  
ADD R1, 0      ; add 0 to R1  
STORE R1, 0
```

asm

The optimized version removes the **unnecessary** addition operation.



```
LOAD R1, 0  
STORE R1, 0
```

asm

# Loop Nest Optimizations – Loop Tiling (cf. [5], [6])

```
for (int i=0; i<n; ++i) { C++  
    for (int j=0; j<m; ++j) {  
        c[i][j] = a[i] * b[j];  
    }  
}
```

The vector **b** **may not** fit into a line of CPU cache, causing multiple cache misses during the inner loop.

It implies multiple **fetches** from the main memory, which is **slow**.

# Loop Nest Optimizations – Loop Tiling (cf. [5], [6])

```
for (int i=0; i<n; ++i) {  
    for (int j=0; j<m; ++j) {  
        c[i][j] = a[i] * b[j];  
    }  
}
```

The vector **b** **may not** fit into a line of CPU cache, causing multiple cache misses during the inner loop.

It implies multiple **fetches** from the main memory, which is **slow**.

```
int TS = 16; // Tile Size  
  
for (int jj=0; jj<m; jj+=TS) {  
    for (int i=0; i<n; ++i) {  
        for (int j=jj; j<MIN(jj + TS, m); ++j) {  
            c[i][j] = a[i] * b[j];  
        }  
    }  
}
```

The inner loop works on a **tile** of **b** that fits into the cache.

# Loop Nest Optimizations – Loop Tiling (cf. [5], [6])

```
for (int i=0; i<n; ++i) {  
    for (int j=0; j<m; ++j) {  
        c[i][j] = a[i] * b[j];  
    }  
}
```

The vector **b** **may not** fit into a line of CPU cache, causing multiple cache misses during the inner loop.

It implies multiple **fetches** from the main memory, which is **slow**.

```
int TS = 16; // Tile Size  
  
for (int jj=0; jj<m; jj+=TS) {  
    for (int i=0; i<n; ++i) {  
        for (int j=jj; j<MIN(jj + TS, m); ++j) {  
            c[i][j] = a[i] * b[j];  
        }  
    }  
}
```

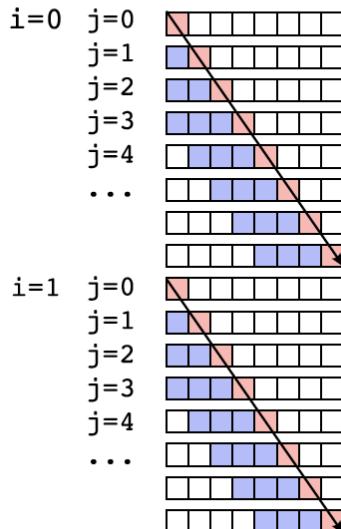
The inner loop works on a **tile** of **b** that fits into the cache.

But, the values for the array **a** will be read  $m / TS$  times!

# Loop Tiling Visualization

**Original:**

```
for (i=0; i<m; i++)
    for (j=0; j<n; j++)
        ...=....*b[j];
```



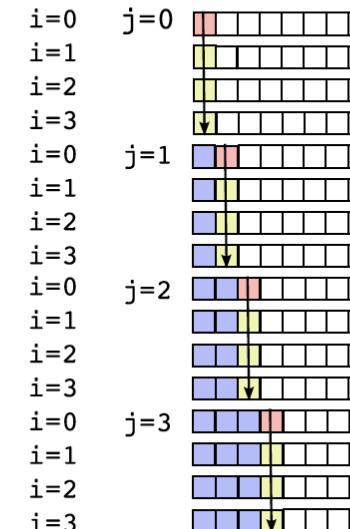
█ - cached, LRU eviction policy  
█ - cache miss (read from memory, slow)  
█ - cache hit (read from cache, fast)

Cache size: 4  
 TILE=4  
 (must be tuned to cache size)

Cache hit rate without tiling: 0%  
 Cache hit rate with tiling: 50%

**Tiled:**

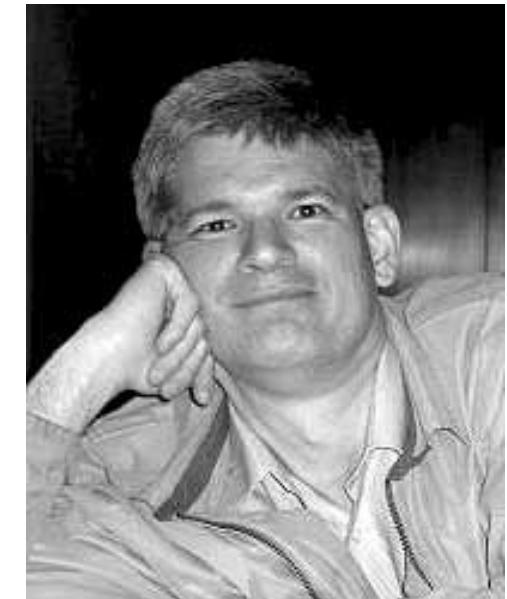
```
for (ii=0; ii<m; ii+=TILE)
    for (j=0; j<n; j++)
        for (i=ii; i<ii+TILE; i++)
            ...=....*b[j];
```



# Tail Call/Recursion Optimization (cf. [7], [8], [9])

Guy L. Steele, Jr. in 1977 observed that **tail-recursive procedure calls** can be optimized to avoid growing the call stack [10]:

*In general, procedure calls may be usefully thought of as GOTO statements which also pass parameters, and can be uniformly coded as [machine code] JUMP instructions.*



# Tail Recursive Factorial Function

```
int fact(int n) {  
    if (n == 0) {  
        return 1;  
    }  
    return n * fact(n - 1);  
}
```

C++

LLVM

```
define i32 @fact(int)(i32 %n) {  
entry:  
    %cmp3 = icmp eq i32 %n, 0  
    br i1 %cmp3, label %return, label %if.else.preheader7 ; label %if.else.preheader  
; if.else.preheader: vector.ph: vector.body: middle.block:  
if.else.preheader7:  
    %n.tr5.ph = phi i32 [ %n, %if.else.preheader ], [ %0, %middle.block ]  
    %acc.tr4.ph = phi i32 [ 1, %if.else.preheader ], [ %4, %middle.block ]  
    br label %if.else  
if.else:  
    %n.tr5 = phi i32 [ %sub, %if.else ], [ %n.tr5.ph, %if.else.preheader7 ]  
    %acc.tr4 = phi i32 [ %mul, %if.else ], [ %acc.tr4.ph, %if.else.preheader7 ]  
    %sub = add nsw i32 %n.tr5, -1  
    %mul = mul nsw i32 %n.tr5, %acc.tr4  
    %cmp = icmp eq i32 %sub, 0  
    br i1 %cmp, label %return, label %if.else  
return:  
    %acc.tr.lcssa = phi i32 [ 1, %entry ], [ %4, %middle.block ], [ %mul, %if.else ]  
    ret i32 %acc.tr.lcssa
```

# Bibliography

- [1] D. E. Knuth, “Structured Programming with go to Statements,” *ACM Comput. Surv.*, vol. 6, no. 4, pp. 261–301, Dec. 1974.
- [2] D. F. Bacon, S. L. Graham, and O. J. Sharp, “Compiler transformations for high-performance computing,” *ACM Computing Surveys (CSUR)*, vol. 26, no. 4, pp. 345–420, 1994.
- [3] W. M. McKeeman, “Peephole optimization,” *Commun. ACM*, vol. 8, no. 7, pp. 443–444, July 1965.

- [4] A. S. Tanenbaum, H. van Staveren, and J. W. Stevenson, “Using Peephole Optimization on Intermediate Code,” *ACM Trans. Program. Lang. Syst.*, vol. 4, no. 1, pp. 21–36, Jan. 1982.
- [5] M. Wolfe, “More iteration space tiling,” in *Proceedings of the 1989 ACM/IEEE Conference on Supercomputing*, in Supercomputing '89. Reno, Nevada, USA: Association for Computing Machinery, 1989, pp. 655–664.
- [6] M. E. Wolf and M. S. Lam, “A data locality optimizing algorithm,” *SIGPLAN Not.*, vol. 26, no. 6, pp. 30–44, May 1991.

- [7] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*. Reading, Massachusetts: Addison Wesley, 1986.
- [8] S. Muchnick, *Advanced compiler design implementation*. Morgan kaufmann, 1997.
- [9] K. D. Cooper and L. Torczon, *Engineering a Compiler*, no. . Morgan Kaufmann, 2022.
- [10] G. L. Steele, “Debunking the “expensive procedure call” myth or, procedure call implementations considered harmful or, LAMBDA: The Ultimate GOTO,” in *Proceedings of the 1977 Annual Conference*,

in ACM '77. Seattle, Washington: Association for Computing Machinery, 1977, pp. 153–162.