Your Optimizing Compiler is not Optimizing Enough. To hell with Multiple Recursions!

Federico Bruzzone,[1] PhD Student

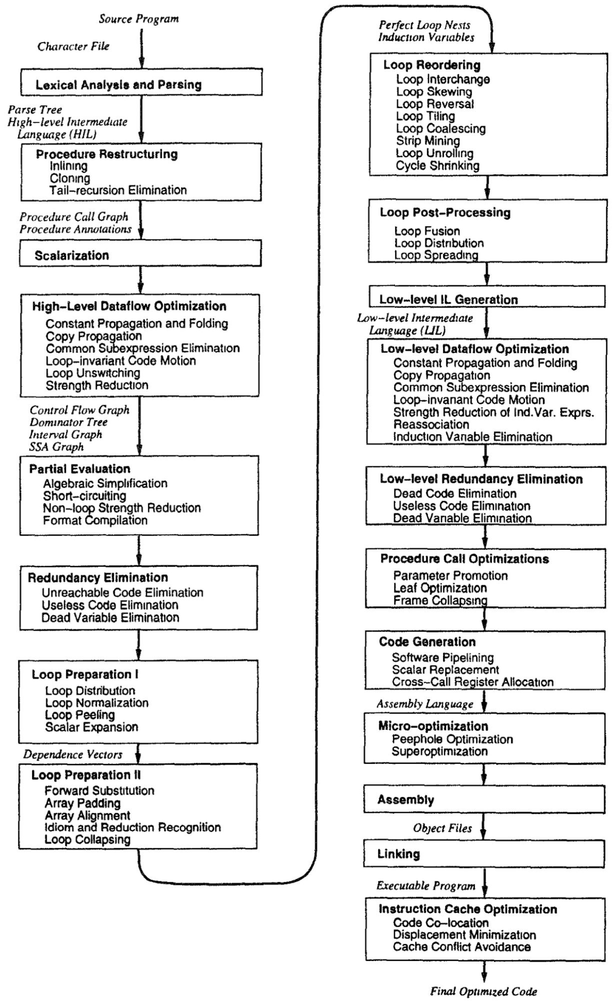
Milan, Italy – 25 November 2025

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

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.

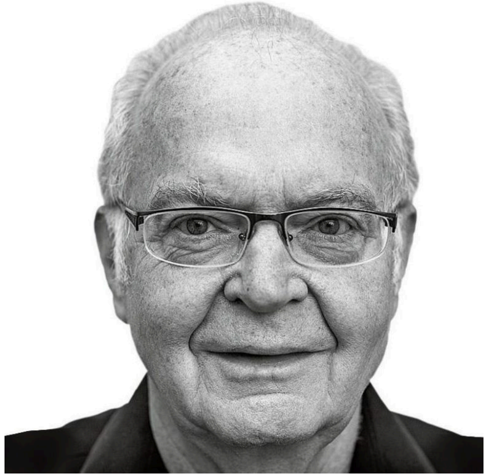


Bacon *et al.*, CSUR 1994 [1]

Premature Optimizations

Donald E. Knuth warned in 1974 about the dangers of **premature optimization** in programming [2]:

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

Compilers use information collected during analysis passes to guide transformations [3], [4].

Compiler optimizations are such transformations (say *meaning-preserving mappings* [5]) applied to the input code to improve certain aspects—such as performance, resource utilization, and power consumption—without altering its observable behavior.

In accordance with the literature [6], [7], such compilers are referred to as **optimizing compilers**.

Optimizing Compilers

Compilers use information collected during analysis passes to guide transformations [3], [4].

Compiler optimizations are such transformations (say *meaning-preserving mappings* [5]) applied to the input code to improve certain aspects—such as performance, resource utilization, and power consumption—without altering its observable behavior.

In accordance with the literature [6], [7], such compilers are referred to as **optimizing compilers**.

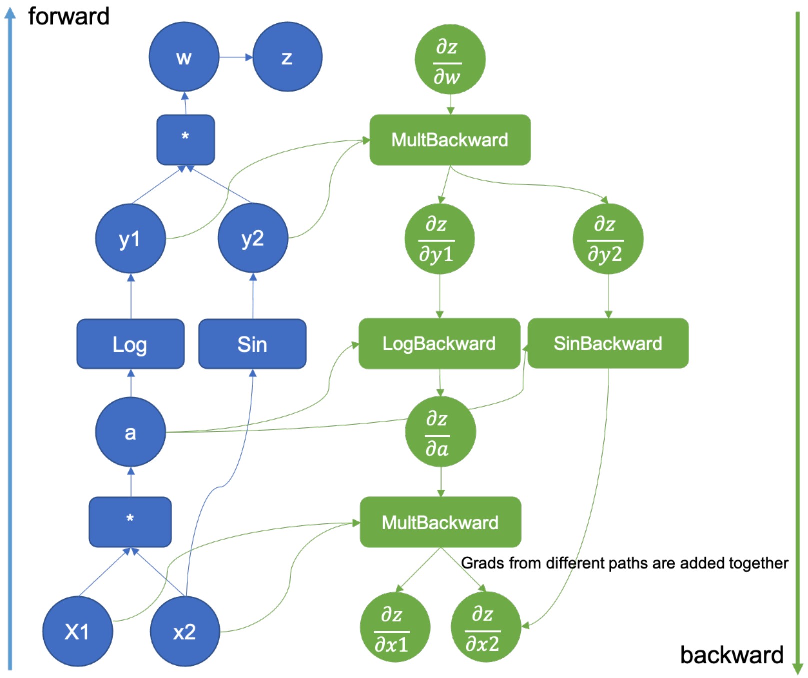
Optimizing Compilers

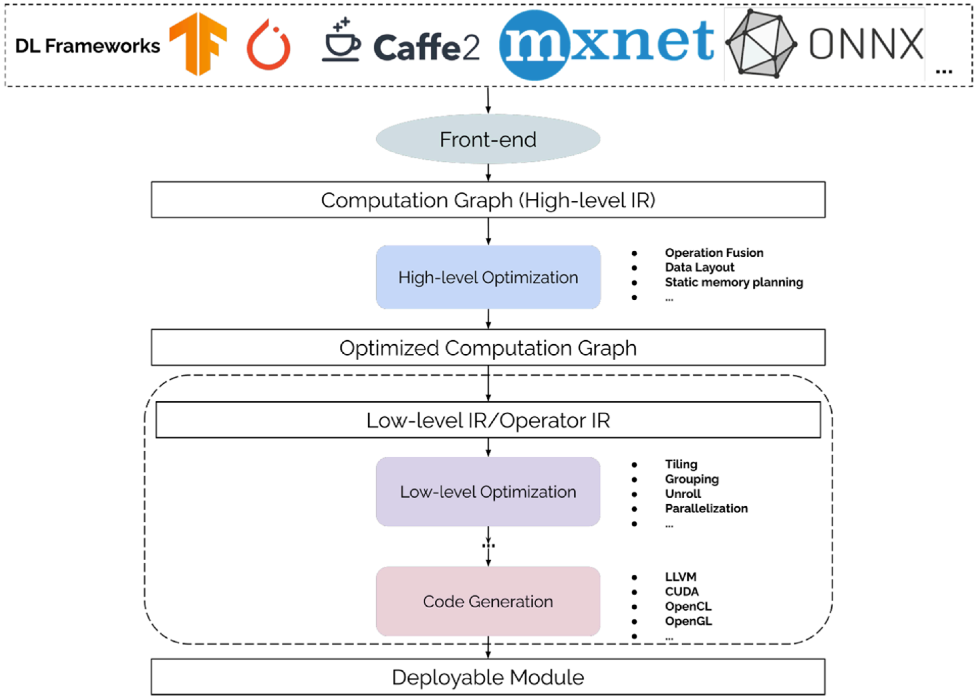
Compilers use information collected during analysis passes to guide transformations [3], [4].

Compiler optimizations are such transformations (say *meaning-preserving mappings* [5]) applied to the input code to improve certain aspects—such as performance, resource utilization, and power consumption—without altering its observable behavior.

In accordance with the literature [6], [7], such compilers are referred to as **optimizing compilers**.

Machine Learning Framework are Just Optimizing Compilers[2]





[2] Li *et al.*, CSUR 2020, [8] compiled a comprehensive survey on deep learning compilers.

Peephole Optimizations in x86-64 (cf. [9], [10])

```asm
; x = x * 2  x: i32
mov     eax, dword ptr [rbp - 4]
imul    eax, 2
mov     dword ptr [rbp - 4], eax
```

Peephole Optimizations in x86-64 (cf. [9], [10])

```asm
; x = x * 2  x: i32
mov      eax, dword ptr [rbp - 4]
imul     eax, 2
mov      dword ptr [rbp - 4], eax
```

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

```asm
; x = x << 1
mov     eax, dword ptr [rbp - 4]
shl     eax
mov     dword ptr [rbp - 4], eax
```

Peephole Optimizations in x86-64 (cf. [9], [10])

```asm
; x = x * 2  x: i32
mov     eax, dword ptr [rbp - 4]
imul    eax, 2
mov     dword ptr [rbp - 4], eax
```

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

```asm
; x = x << 1
mov    eax, dword ptr [rbp - 4]
shl    eax
mov    dword ptr [rbp - 4], eax
```

```asm
; x = x + 0
mov     eax, dword ptr [rbp - 4]
add     eax, 0
mov     dword ptr [rbp - 4], eax
```

Peephole Optimizations in x86-64 (cf. [9], [10])

```asm
; x = x * 2  x: i32
mov     eax, dword ptr [rbp - 4]
imul    eax, 2
mov     dword ptr [rbp - 4], eax
```

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

```asm
; x = x << 1
mov    eax, dword ptr [rbp - 4]
shl    eax
mov    dword ptr [rbp - 4], eax
```

```asm
; x = x + 0
mov     eax, dword ptr [rbp - 4]
add     eax, 0
mov     dword ptr [rbp - 4], eax
```

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

```asm
mov     eax, dword ptr [rbp - 4]
mov     dword ptr [rbp - 4], eax
```

Peephole Optimizations in x86-64 (cf. [9], [10])

```asm
; x = x * 2  x: i32
mov     eax, dword ptr [rbp - 4]
imul    eax, 2
mov     dword ptr [rbp - 4], eax
```

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

```asm
; x = x << 1
mov    eax, dword ptr [rbp - 4]
shl    eax
mov    dword ptr [rbp - 4], eax
```

```asm
; x = x + 0
mov     eax, dword ptr [rbp - 4]
add     eax, 0
mov     dword ptr [rbp - 4], eax
```

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

```asm
mov     eax, dword ptr [rbp - 4]
mov     dword ptr [rbp - 4], eax
```

The `mov` instructions are redundant and can be **pruned** as well!

Loop Nest Optimizations – Loop Tiling (cf. [11], [12])

C++

```cpp
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**.

Loop Nest Optimizations — Loop Tiling (cf. [11], [12])

```cpp
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**.

```cpp
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. [11], [12])

```cpp
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.

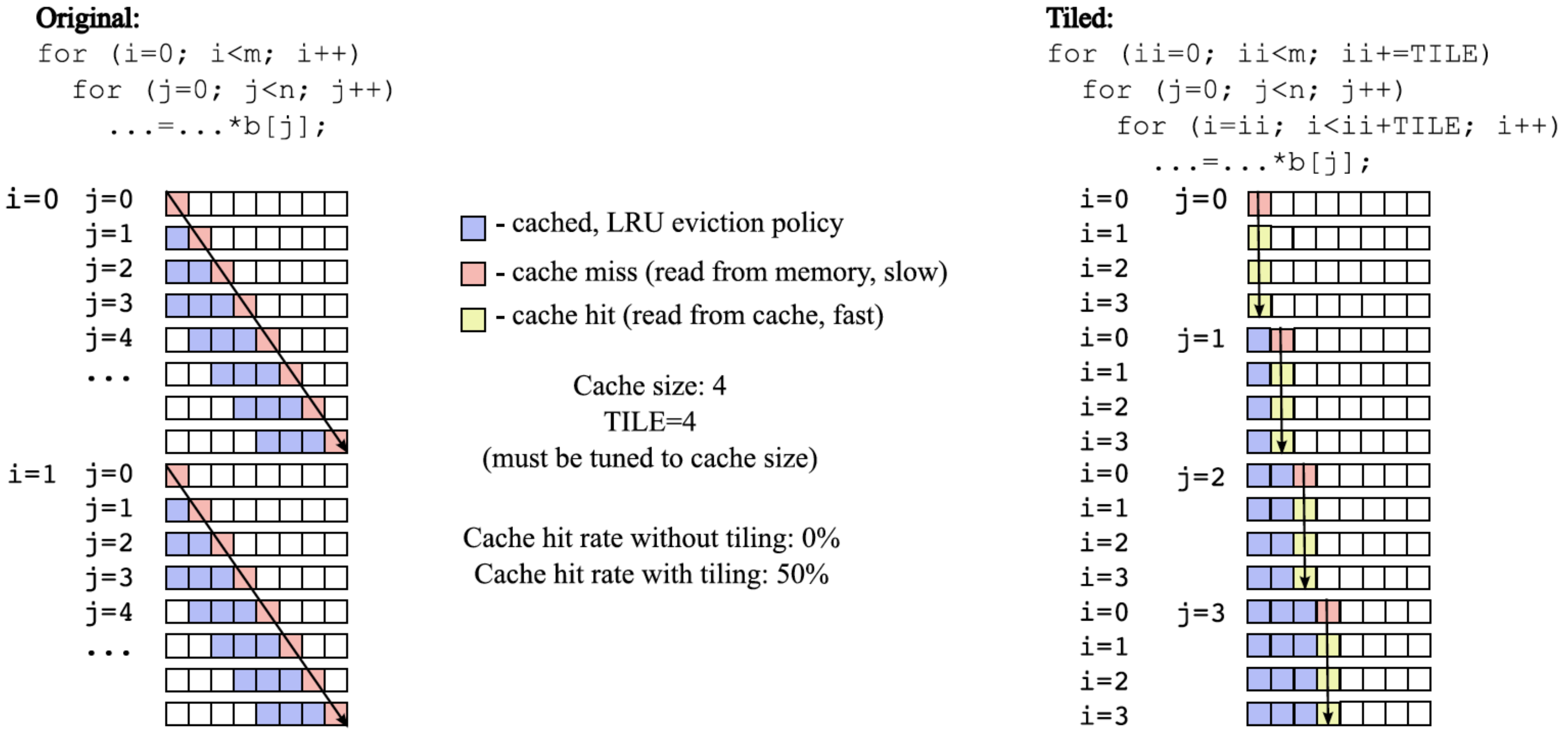
It implies multiple **fetches** from the main memory, which is **slow**.

```cpp
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!

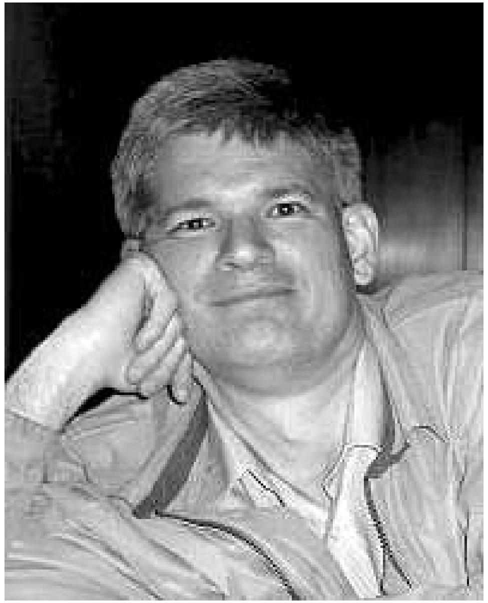
Loop Tiling Visualization



Tail Call/Recursion Optimization (cf. [3], [13], [14])

Guy L. Steele, Jr. in 1977 observed that **tail-recursive procedure calls** can be optimized to avoid growing the call stack [15]:

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.



From Recursive to Iterative Functions (cf. [16])

$$\boldsymbol{f}(x) = \begin{cases} b(x_0) \text{ if } x = x_0 \\ a(x, \boldsymbol{f}(d(x))) \text{ otherwise} \end{cases}$$

$s.t.$ a, b, and so on may denote any pieces of code.

From Recursive to Iterative Functions (cf. [16])

$$\boldsymbol{f}(x) = \begin{cases} b(x_0) \text{ if } x = x_0 \\ a(x, \boldsymbol{f}(d(x))) \text{ otherwise} \end{cases}$$

$s.t.$ a, b, and so on may denote any pieces of code.

To transform recursive function $\boldsymbol{f}$ into iterative form, we need to:

1. Identifies an increment $\oplus$ to the argument of $\boldsymbol{f}$, i.e., $x' = x \oplus y$ such that $x = \mathit{prev}(x')$, where prev is based on the arguments of the recursive call. In this case, $\mathit{prev}(x) = d(x)$ and, if d^{-1} exists, $x \oplus y = d^{-1}(x)$, can be plugged in for y.
2. Derives an incremental program $\boldsymbol{f}'(x, r)$ that computes $\boldsymbol{f}(x)$ using an accumulator r of $\boldsymbol{f}(\mathit{prev}(x))$.
3. Forms an iterative version that initializes using the base case of $\boldsymbol{f}$ and iteratively applies $\boldsymbol{f}'$ until reaching the desired argument.

From Recursive to Iterative Functions (cf. [16])

$$\boldsymbol{f}(x) = \begin{cases} b(x_0) \text{ if } x = x_0 \\ a(x, \boldsymbol{f}(d(x))) \text{ otherwise} \end{cases}$$

$s.t.$ a, b, and so on may denote any pieces of code.

To transform recursive function $\boldsymbol{f}$ into iterative form, we need to:

1. Identifies an increment $\oplus$ to the argument of $\boldsymbol{f}$, i.e., $x' = x \oplus y$ such that $x = prev(x')$, where $prev$ is based on the arguments of the recursive call. In this case, $prev(x) = d(x)$ and, if d^{-1} exists, $x \oplus y = d^{-1}(x)$, can be plugged in for y.
2. Derives an incremental program $\boldsymbol{f}'(x, r)$ that computes $\boldsymbol{f}(x)$ using an accumulator r of $\boldsymbol{f}(prev(x))$.
3. Forms an iterative version that initializes using the base case of $\boldsymbol{f}$ and iteratively applies $\boldsymbol{f}'$ until reaching the desired argument.

$$
\begin{array}{l}
\boldsymbol{f}(x) = \{ \\
\quad x_1 = x_0; r = b(x_0); \\
\quad \text{while } (x_1 \neq x)\{ \\
\qquad x_1 = d^{-1}(x_1); \\
\qquad r = a(x_1, r); \\
\quad \} \\
\quad \text{return } r; \\
\}
\end{array}
$$

From Recursive to Iterative Functions (cf. [16])

$$\boldsymbol{f}(x) = \begin{cases} b(x_0) \text{ if } x = x_0 \\ a(x, \boldsymbol{f}(d(x))) \text{ otherwise} \end{cases}$$

$s.t.$ a, b, and so on may denote any pieces of code.

To transform recursive function $\boldsymbol{f}$ into iterative form, we need to:

1. Identifies an increment $\oplus$ to the argument of $\boldsymbol{f}$, i.e., $x' = x \oplus y$ such that $x = prev(x')$, where $prev$ is based on the arguments of the recursive call. In this case, $prev(x) = d(x)$ and, if d^{-1} exists, $x \oplus y = d^{-1}(x)$, can be plugged in for y.
2. Derives an incremental program $\boldsymbol{f}'(x, r)$ that computes $\boldsymbol{f}(x)$ using an accumulator r of $\boldsymbol{f}(prev(x))$.
3. Forms an iterative version that initializes using the base case of $\boldsymbol{f}$ and iteratively applies $\boldsymbol{f}'$ until reaching the desired argument.

$$
\begin{aligned}
&\boldsymbol{f}(x) = \{ \\
&\quad x_1 = x_0; r = b(x_0); \\
&\quad \text{while } (x_1 \neq x)\{ \\
&\qquad x_1 = d^{-1}(x_1); \\
&\qquad r = a(x_1, r); \\
&\quad \} \\
&\quad \text{return } r; \\
&\}
\end{aligned}
$$

Note that, when a is in the form $a(a_1(x), y)$ and a is associative, we do not need d^{-1} and x_1.

Tail-recursive Factorial Function

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

The replacement of $n * ((n-1) * (n-2))$ by $(n * (n-1)) * (n-2)$ is valid due to the **associativity** of multiplication.

$$
\begin{aligned}
&\boldsymbol{f}(x) = \{ \\
&\qquad r = b(x_0); \\
&\qquad \text{while } (x \neq x_0)\{ \\
&\qquad\qquad r = a(r, a_1(x)); \\
&\qquad\qquad x = d(x); \\
&\qquad \} \\
&\qquad \text{return } r; \\
&\}
\end{aligned}
$$

Tail-recursive Factorial Function

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

The replacement of $n * ((n-1) * (n-2))$ by $(n * (n-1)) * (n-2)$ is valid due to the **associativity** of multiplication.

$$
\begin{aligned}
&\boldsymbol{f}(x) = \{\\
&\qquad r = b(x_0);\\
&\qquad \text{while } (x \neq x_0)\{\\
&\qquad\qquad r = a(r, a_1(x));\\
&\qquad\qquad x = d(x);\\
&\qquad \}\\
&\qquad \text{return } r;\\
&\}
\end{aligned}
$$

Note that, (i) when dealing with IEEE754 numbers, multiplication is **not** strictly associative, and (ii) the latter *might be* slower due to multiply bigger numbers.

Tail-recursive Factorial Function

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

The replacement of $n * ((n-1) * (n-2))$ by $(n * (n-1)) * (n-2)$ is valid due to the **associativity** of multiplication.

$$
\begin{aligned}
&\boldsymbol{f}(x) = \{ \\
&\qquad r = b(x_0); \\
&\qquad \text{while } (x \neq x_0)\{ \\
&\qquad\qquad r = a(r, a_1(x)); \\
&\qquad\qquad x = d(x); \\
&\qquad \} \\
&\qquad \text{return } r; \\
&\}
\end{aligned}
$$

Note that, (i) when dealing with IEEE754 numbers, multiplication is **not** strictly associative, and (ii) the latter *might be* slower due to multiply bigger numbers.

```llvm
define i32 @fact(int)(i32 %n)  {
entry:
  %cmp3 = icmp eq i32 %n, 0
  br i1 %cmp3, label %return, label %if.else  ; label %if.else.preheader
; if.else.preheader -> vector.ph -> vector.body -> middle.block -> if.else.preheader7
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. 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.

[2] D. E. Knuth, “Structured Programming with go to Statements,” *ACM Comput. Surv.*, vol. 6, no. 4, pp. 261–301, Dec. 1974.

[3] K. D. Cooper and L. Torczon, *Engineering a Compiler*, no. . Morgan Kaufmann, 2022.

[4] K. Kennedy and J. R. Allen, *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc., 2001.

[5] R. Paige, "Future directions in program transformations," *SIGPLAN Not.*, vol. 32, no. 1, pp. 94–98, Jan. 1997.

[6] F. E. Allen, "Program Optimization," *Annual Review of Automatic Programming*, vol. 5. Pergamon Press, pp. 239–307, 1966.

[7] W. A. Wulf, "The design of an optimizing compiler," 1973.

[8] M. Li *et al.*, “The deep learning compiler: A comprehensive survey,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 3, pp. 708–727, 2020.

[9] W. M. McKeeman, “Peephole optimization,” *Commun. ACM*, vol. 8, no. 7, pp. 443–444, July 1965.

[10] 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.

[11] 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.

[12] M. E. Wolf and M. S. Lam, “A data locality optimizing algorithm,” *SIGPLAN Not.*, vol. 26, no. 6, pp. 30–44, May 1991.

[13] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*,. Reading, Massachusetts: Addison Wesley, 1986.

[14] S. Muchnick, *Advanced compiler design implementation*. Morgan kaufmann, 1997.

[15] 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.

[16] Y. A. Liu and S. D. Stoller, “From recursion to iteration: what are the optimizations?,” in *Proceedings of the 2000 ACM SIGPLAN workshop on Partial evaluation and semantics-based program manipulation*, 1999, pp. 73–82.