

Your Optimizing Compiler is Not Optimizing Enough. To Hell With Multiple Recursions!

Federico Bruzzone,¹ PhD Candidate

Milan, Italy – 4 December 2025



¹ADAPT Lab – Università degli Studi di Milano,

Website: federicobruzzone.github.io,

Github: github.com/FedericoBruzzone,

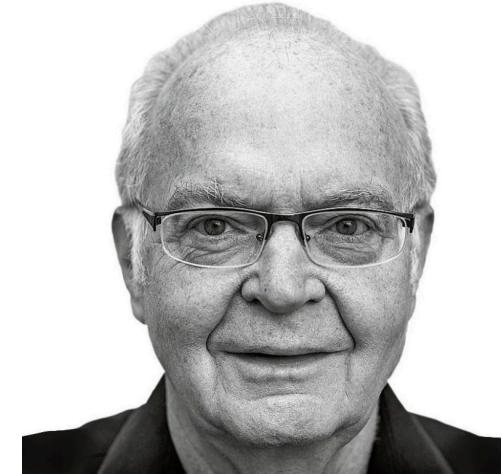
Email: federico.bruzzone@unimi.it

Slides: federicobruzzone.github.io/activities/presentations/your-optimizing-compiler-is-not-optimizing-enough.pdf

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

Compilers use information collected during analysis passes to guide transformations [2], [3].

Compiler optimizations² 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**.

²A chronologically sorted list of papers on compiler optimization, from the works of 1952 through the techniques of 1994, is available at [4].

Optimizing Compilers

Compilers use information collected during analysis passes to guide transformations [2], [3].

Compiler optimizations³ 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**.

³A chronologically sorted list of papers on compiler optimization, from the works of 1952 through the techniques of 1994, is available at [4].

Optimizing Compilers

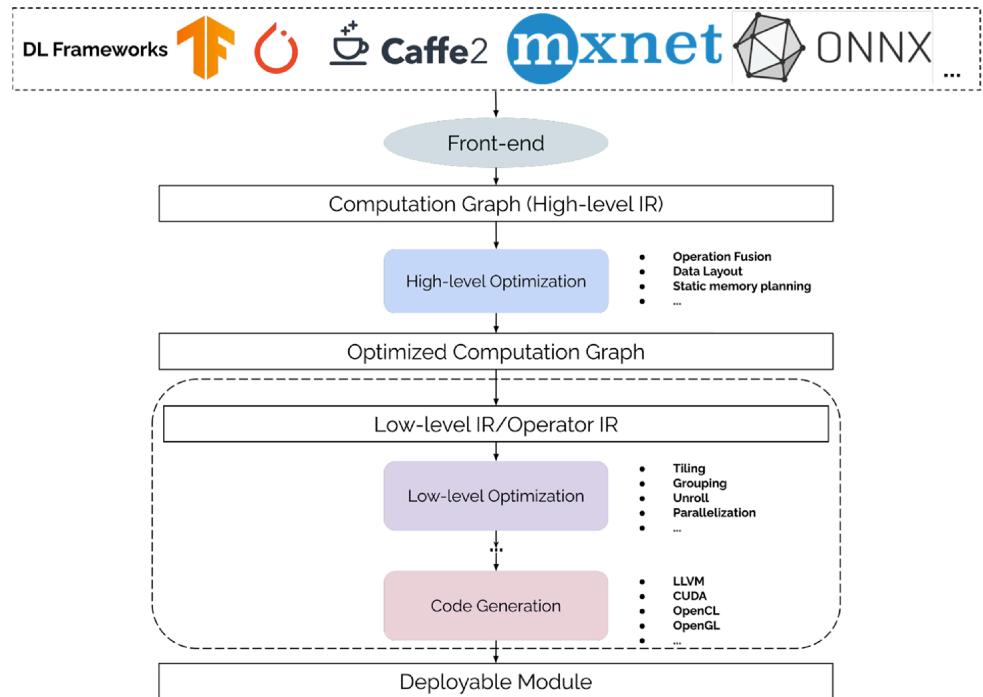
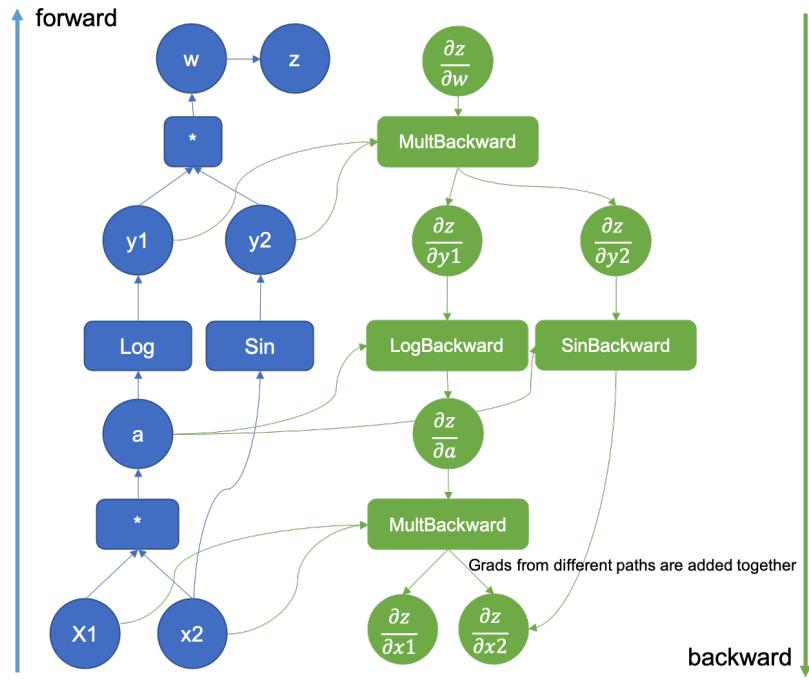
Compilers use information collected during analysis passes to guide transformations [2], [3].

Compiler optimizations⁴ 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**.

⁴A chronologically sorted list of papers on compiler optimization, from the works of 1952 through the techniques of 1994, is available at [4].

Machine Learning Framework are Just Optimizing Compilers⁵



⁵TensorFlow XLA, NVIDIA CUDA Compiler (NVCC), MLIR, and TVM all use LLVM [8]. Li *et al.*, [9] compiled a survey on ML compilers.

Peephole Optimizations in x86-64 (cf. [10], [11])

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

Peephole Optimizations in x86-64 (cf. [10], [11])

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

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

```
1 ; x = x << 1, s.t. x: i32          asm
2 mov     eax, dword ptr [rbp - 4]
3 shl     eax
4 mov     dword ptr [rbp - 4], eax
```

Peephole Optimizations in x86-64 (cf. [10], [11])

```
1 ; x = x * 2, s.t. x: i32
2 mov     eax, dword ptr [rbp - 4]
3 imul    eax, 2
4 mov     dword ptr [rbp - 4], eax
```

asm

```
1 ; x = x + 0, s.t. x: i32
```

```
2 mov     eax, dword ptr [rbp - 4]
3 add     eax, 0
4 mov     dword ptr [rbp - 4], eax
```

asm

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

```
1 ; x = x << 1, s.t. x: i32
2 mov     eax, dword ptr [rbp - 4]
3 shl     eax
4 mov     dword ptr [rbp - 4], eax
```

asm

Peephole Optimizations in x86-64 (cf. [10], [11])

```
1 ; x = x * 2, s.t. x: i32
2 mov     eax, dword ptr [rbp - 4]
3 imul    eax, 2
4 mov     dword ptr [rbp - 4], eax
```

asm

```
1 ; x = x + 0, s.t. x: i32
```

```
2 mov     eax, dword ptr [rbp - 4]
3 add     eax, 0
4 mov     dword ptr [rbp - 4], eax
```

asm

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

```
1 ; x = x << 1, s.t. x: i32
2 mov     eax, dword ptr [rbp - 4]
3 shl     eax
4 mov     dword ptr [rbp - 4], eax
```

asm

```
1 mov     eax, dword ptr [rbp - 4]
2 mov     dword ptr [rbp - 4], eax
```

asm

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

Peephole Optimizations in x86-64 (cf. [10], [11])

```
1 ; x = x * 2, s.t. x: i32
2 mov     eax, dword ptr [rbp - 4]
3 imul    eax, 2
4 mov     dword ptr [rbp - 4], eax
```

asm

```
1 ; x = x + 0, s.t. x: i32
```

```
2 mov     eax, dword ptr [rbp - 4]
3 add     eax, 0
4 mov     dword ptr [rbp - 4], eax
```

asm

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

```
1 ; x = x << 1, s.t. x: i32
2 mov     eax, dword ptr [rbp - 4]
3 shl     eax
4 mov     dword ptr [rbp - 4], eax
```

asm

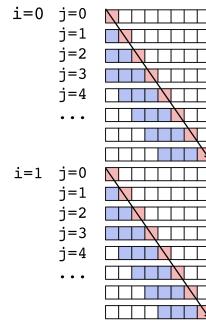
```
1 mov     eax, dword ptr [rbp - 4]
2 mov     dword ptr [rbp - 4], eax
```

asm

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

Loop Nest Optimizations – Loop Tiling (cf. [12], [13])

```
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.

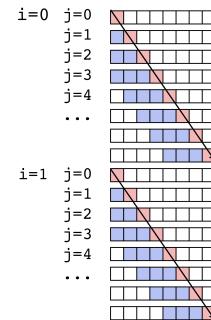
It implies multiple **fetches** from the main memory, which is **slow**.

Loop Nest Optimizations – Loop Tiling (cf. [12], [13])

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

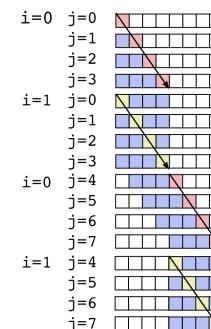
The vector **b** **may not** fit into a line of CPU cache, causing multiple cache misses during the inner loop.

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



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

```
for (int jj = 0; jj < m; jj += TILE_SIZE) C++
    for (int i = 0; i < n; ++i)
        for (int j = jj; j < MIN(jj + TILE_SIZE, m); ++j)
            c[i][j] = a[i] * b[j];
```



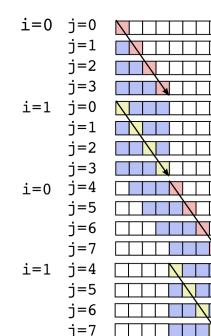
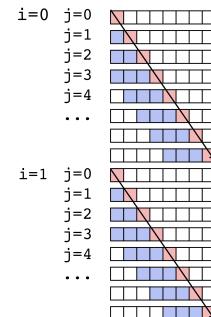
[A. Vladimirov, Session 10](#)

Loop Nest Optimizations – Loop Tiling (cf. [12], [13])

```
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**.



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

```
for (int jj = 0; jj < m; jj += TILE_SIZE) C++
    for (int i = 0; i < n; ++i)
        for (int j = jj; j < MIN(jj + TILE_SIZE, m); ++j)
            c[i][j] = a[i] * b[j];
```

Careful readers may notice that, in this version, the values for the array **a** will be read $m / \text{TILE_SIZE}$!

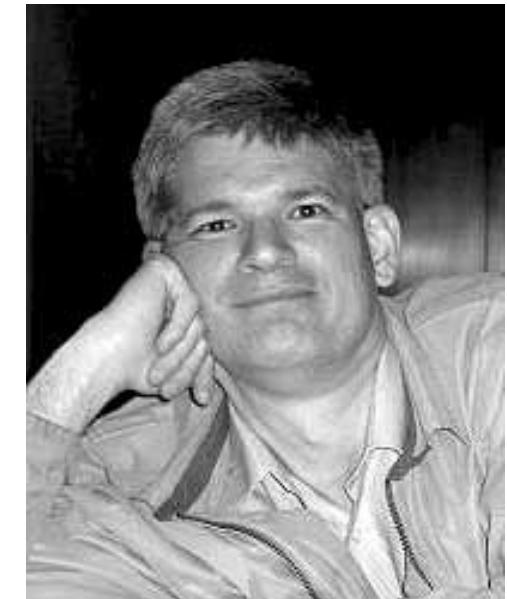
```
for (int ii = 0; ii < n; ii += TILE_SIZE_I) C++
    for (int jj = 0; jj < m; jj += TILE_SIZE_J)
        for (int i = ii; i < MIN(n, ii + TILE_SIZE_I); i++)
            for (int j = jj; j < MIN(m, jj + TILE_SIZE_J); j++)
                c[i][j] = a[i] * b[j];
```

[A. Vladimirov, Session 10](#)

Tail Call/Recursion Optimization (cf. [2], [14], [15])

Guy L. Steele, Jr. in 1977 observed that **tail-recursive procedure calls** can be optimized to avoid growing the call stack [16]:

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 Recursion to Iteration (cf. [17])

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

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

From Recursion to Iteration (cf. [17])

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

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

To transform recursive function f into iterative form, we need to:

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

```
f(x) = {
    x1 = x0; r = b(x0);
    while (x1 ≠ x){
        x1 = d⁻¹(x1);
        r = a(x1, r);
    }
    return r;
}
```

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

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

C++

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

$$f(x) = \{$$
$$r = b(x_0);$$
$$\text{while } (x \neq x_0) \{$$
$$r = a(r, a_1(x));$$
$$x = d(x);$$
$$\}$$
$$\text{return } r;$$
$$\}$$

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

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

C++

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

$$f(x) = \{$$
$$r = b(x_0);$$
$$\text{while } (x \neq x_0) \{$$
$$r = a(r, a_1(x));$$
$$x = d(x);$$
$$\}$$
$$\text{return } r;$$
$$\}$$

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

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

C++

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

```
f(x) = {
    r = b(x₀);
    while (x ≠ x₀){
        r = a(r, a₁(x));
        x = d(x);
    }
    return r;
}
```

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.

"clang -O3 -S -emit-llvm fact.c -o -" produces something like:

```
define i32 @fact(i32 %n) {
entry:
%cmp3 = icmp eq i32 %n, 0
br i1 %cmp3, label %return, label %if.else
if.else:
%n.tr5 = phi i32 [ %sub, %if.else ], [ %n, %entry ]
%acc.tr4 = phi i32 [ %mul, %if.else ], [ 1, %entry ]
%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 ], [ %mul, %if.else ]
ret i32 %acc.tr.lcssa
}
```

The basic blocks related to loop vectorization (for performing SIMD operations) have been omitted for clarity

What About Fibonacci? To Hell With Multiple Recursions!

```
int fib(int n) {
    if (n <= 1) {
        return n;
    }
    return fib(n - 1) + fib(n - 2);
}
```

C++

The LLVM Optimized Version (but human readable)

```
int fib(int n) {
    if (n < 2) {
        return n;
    }
    int acc = 0;
loop: /* while (1) { */
    int call = fib(n - 1);
    acc = call + acc;
    if (n < 4) goto ret; /* return acc + (n - 2); */
    n = n - 2;
    goto loop; /* } */
ret:
    return acc + (n - 2);
}
```

C++

Note that, the following LLVM IR is a fixed-point representation of the `fib` function; observable by the output of “`opt -passes="default<O3>" -S fib-03.ll -o -`”

```
define i32 @fib(i32 %n) {
entry:
    %cmp6 = icmp slt i32 %n, 2
    br i1 %cmp6, label %return, label %if.end
if.end:
    %n.tr8 = phi i32 [ %sub1, %if.end ], [ %n, %entry ]
    %accumulator.tr7 = phi i32 [ %add, %if.end ], [ 0, %entry ]
    %sub = add nsw i32 %n.tr8, -1
    %call = tail call i32 @fib(i32 %sub)
    %sub1 = add nsw i32 %n.tr8, -2
    %add = add nsw i32 %call, %accumulator.tr7
    %cmp = icmp samesign ult i32 %n.tr8, 4
    br i1 %cmp, label %return, label %if.end
return:
    %accumulator.tr.lcssa = phi i32 [ 0, %entry ], [ %add, %if.end ]
    %n.tr.lcssa = phi i32 [ %n, %entry ], [ %sub1, %if.end ]
    %accumulator.ret.tr = add nsw i32 %n.tr.lcssa, %accumulator.tr.lcssa
    ret i32 %accumulator.ret.tr
}
```

llvm

The Y. Annie Liu's Incrementalization



In 1990, Liu *et al.* have done extensive research on **Incrementalization** [17], [18], [19], [20].

Even in presence of multiple recursions, in [17, Sect. 7], they proposed a **systematic** approach (*static analysis* and *semantic-preserving transformations*) to derive an incremental program following the three steps outlined earlier (cf. slide “*From Recursion to Iteration*”).

But the Step 2. builds upon the principles of [18] and [19] – which, typically rely on user-provided knowledge or a theorem prover to derive the incremental program.

Conclusions

The papers are a little bit old-fashioned, and the key aspect of deriving the incremental program in presence of multiple recursions is **opaque**.

Last week I wrote an email to Liu asking for clarification, and two days ago I received a kind reply directing me to [21] (2024)—which confirms my understanding of the situation.

We are trying to understand whether it is really possible to rely exclusively on static analysis steps to automatically perform the transformation in a “general” context.

Thank You!

Bibliography

- [1] D. E. Knuth, “Structured Programming with go to Statements,” *ACM Comput. Surv.*, vol. 6, no. 4, pp. 261–301, Dec. 1974.
- [2] K. D. Cooper and L. Torczon, *Engineering a Compiler*, no. . Morgan Kaufmann, 2022.
- [3] K. Kennedy and J. R. Allen, *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc., 2001.
- [4] F. Bruzzone, “Papers on Compiler Optimizations: Analysis and Transformations (1952-1994).” <https://github.com/FedericoBruzzone/papers-on-compiler-optimizations>, 2025.
- [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] C. Lattner and V. Adve, “LLVM: A compilation framework for lifelong program analysis & transformation,” in *International symposium on code generation and optimization, 2004. CGO 2004.*, 2004, pp. 75–86.
- [9] 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.
- [10] W. M. McKeeman, “Peephole optimization,” *Commun. ACM*, vol. 8, no. 7, pp. 443–444, July 1965.
- [11] 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.
- [12] 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.
- [13] M. E. Wolf and M. S. Lam, “A data locality optimizing algorithm,” *SIGPLAN Not.*, vol. 26, no. 6, pp. 30–44, May 1991.

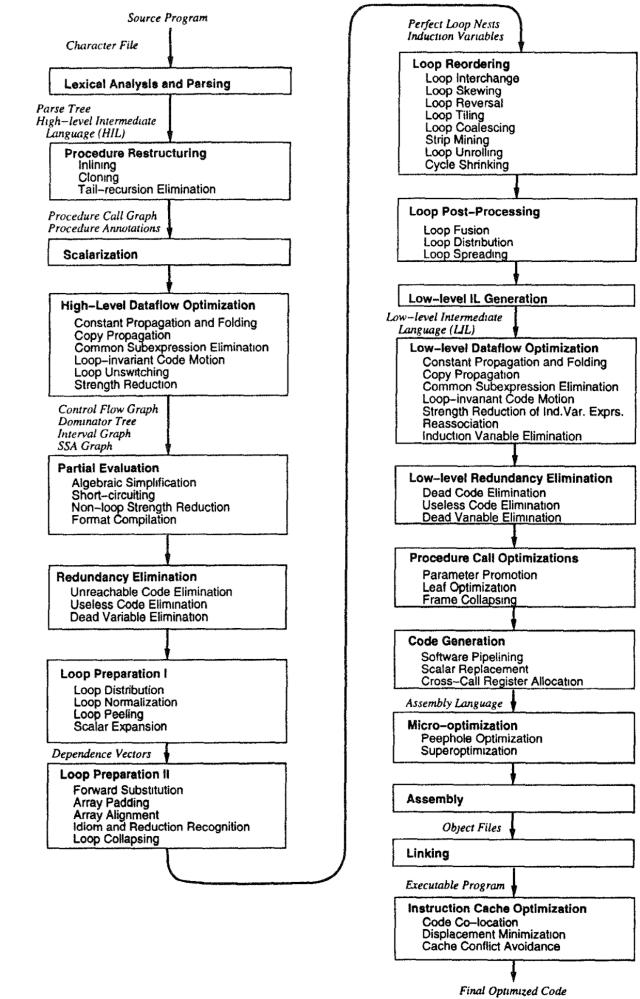
- [14] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*. Reading, Massachusetts: Addison Wesley, 1986.
- [15] S. Muchnick, *Advanced compiler design implementation*. Morgan kaufmann, 1997.
- [16] G. L. Steele, “Debunking the “expensive procedure call” myth or, procedure 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.
- [17] 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.
- [18] Y. A. Liu, S. D. Stoller, and T. Teitelbaum, “Static caching for incremental computation,” *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 20, no. 3, pp. 546–585, 1998.
- [19] Y. A. Liu and T. Teitelbaum, “Systematic derivation of incremental programs,” *Science of Computer Programming*, vol. 24, no. 1, pp. 1–39, 1995.

- [20] Y. A. Liu, “CACHET: An interactive, incremental-attribution-based program transformation system for deriving incremental programs,” in *Proceedings 1995 10th Knowledge-Based Software Engineering Conference*, 1995, pp. 19–26.
- [21] Y. A. Liu, “Incremental Computation: What Is the Essence? (Invited Contribution),” in *Proceedings of the 2024 ACM SIGPLAN International Workshop on Partial Evaluation and Program Manipulation*, in PEPM 2024. London, UK: Association for Computing Machinery, 2024, pp. 39–52.
- [22] 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.

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 [22]