Specific techniques[[edit](https://en.wikipedia.org/w/index.php?title=Optimizing_compiler&action=edit&section=4)]

**Loop optimizations**[[edit](https://en.wikipedia.org/w/index.php?title=Optimizing_compiler&action=edit&section=5)]

*Main article:*[*Loop optimization*](https://en.wikipedia.org/wiki/Loop_optimization)

Some optimization techniques primarily designed to operate on loops include

[**Induction variable analysis**](https://en.wikipedia.org/wiki/Induction_variable_analysis)

Roughly, if a variable in a loop is a simple linear function of the index variable, such as j := 4\*i + 1, it can be updated appropriately each time the loop variable is changed. This is a [strength reduction](https://en.wikipedia.org/wiki/Strength_reduction), and also may allow the index variable's definitions to become [dead code](https://en.wikipedia.org/wiki/Dead_code). This information is also useful for [bounds-checking elimination](https://en.wikipedia.org/wiki/Bounds-checking_elimination) and [dependence analysis](https://en.wikipedia.org/wiki/Dependence_analysis), among other things.

[**Loop fission**](https://en.wikipedia.org/wiki/Loop_fission)**or loop distribution**

Loop fission attempts to break a loop into multiple loops over the same index range but each taking only a part of the loop's body. This can improve [locality of reference](https://en.wikipedia.org/wiki/Locality_of_reference), both of the data being accessed in the loop and the code in the loop's body.

[**Loop fusion**](https://en.wikipedia.org/wiki/Loop_fusion)**or loop combining or loop ramming or loop jamming**

Another technique which attempts to reduce loop overhead. When two adjacent loops would iterate the same number of times (whether or not that number is known at compile time), their bodies can be combined as long as they make no reference to each other's data.

[**Loop inversion**](https://en.wikipedia.org/wiki/Loop_inversion)

This technique changes a standard *while* loop into a *do/while* (also known as *repeat/until*) loop wrapped in an *if* conditional, reducing the number of jumps by two, for cases when the loop is executed. Doing so duplicates the condition check (increasing the size of the code) but is more efficient because jumps usually cause a [pipeline stall](https://en.wikipedia.org/wiki/Pipeline_stall). Additionally, if the initial condition is known at compile-time and is known to be [side-effect](https://en.wikipedia.org/wiki/Side_effect_(computer_science))-free, the *if* guard can be skipped.

[**Loop interchange**](https://en.wikipedia.org/wiki/Loop_interchange)

These optimizations exchange inner loops with outer loops. When the loop variables index into an array, such a transformation can improve [locality of reference](https://en.wikipedia.org/wiki/Locality_of_reference), depending on the array's layout.

[**Loop-invariant code motion**](https://en.wikipedia.org/wiki/Loop-invariant_code_motion)

If a quantity is computed inside a loop during every iteration, and its value is the same for each iteration, it can vastly improve efficiency to hoist it outside the loop and compute its value just once before the loop begins. This is particularly important with the address-calculation expressions generated by loops over arrays. For correct implementation, this technique must be used with [loop inversion](https://en.wikipedia.org/wiki/Loop_inversion), because not all code is safe to be hoisted outside the loop.

[**Loop nest optimization**](https://en.wikipedia.org/wiki/Loop_nest_optimization)

Some pervasive algorithms such as matrix multiplication have very poor cache behavior and excessive memory accesses. Loop nest optimization increases the number of cache hits by performing the operation over small blocks and by using a loop interchange.

[**Loop reversal**](https://en.wikipedia.org/w/index.php?title=Loop_reversal&action=edit&redlink=1)

Loop reversal reverses the order in which values are assigned to the index variable. This is a subtle optimization which can help eliminate [dependencies](https://en.wikipedia.org/wiki/Dependence_analysis) and thus enable other optimizations. Furthermore, on some architectures, loop reversal contributes to smaller code, as when the loop index is being decremented, the condition that needs to be met in order for the running program to exit the loop is a comparison with zero. This is often a special, parameter-less instruction, unlike a comparison with a number, which needs the number to compare to. Therefore, the amount of bytes needed to store the parameter is saved by using the loop reversal. Additionally, if the comparison number exceeds the size of word of the platform, in standard loop order, multiple instructions would need to be executed in order to evaluate the comparison, which is not the case with loop reversal.

[**Loop unrolling**](https://en.wikipedia.org/wiki/Loop_unrolling)

Unrolling duplicates the body of the loop multiple times, in order to decrease the number of times the loop condition is tested and the number of jumps, which hurt performance by impairing the instruction pipeline. A "fewer jumps" optimization. Completely unrolling a loop eliminates all overhead, but requires that the number of iterations be known at compile time.

[**Loop splitting**](https://en.wikipedia.org/wiki/Loop_splitting)

Loop splitting attempts to simplify a loop or eliminate [dependencies](https://en.wikipedia.org/wiki/Dependence_analysis) by breaking it into multiple loops which have the same bodies but iterate over different contiguous portions of the index range. A useful special case is [*loop peeling*](https://en.wikipedia.org/wiki/Loop_peeling), which can simplify a loop with a problematic first iteration by performing that iteration separately before entering the loop.

[**Loop unswitching**](https://en.wikipedia.org/wiki/Loop_unswitching)

Unswitching moves a conditional from inside a loop to outside the loop by duplicating the loop's body inside each of the if and else clauses of the conditional.

[**Software pipelining**](https://en.wikipedia.org/wiki/Software_pipelining)

The loop is restructured in such a way that work done in an iteration is split into several parts and done over several iterations. In a tight loop this technique hides the latency between loading and using values.

[**Automatic parallelization**](https://en.wikipedia.org/wiki/Automatic_parallelization)

A loop is converted into multi-threaded or vectorized (or even both) code in order to utilize multiple processors simultaneously in a shared-memory multiprocessor (SMP) machine, including multi-core machines.

**Data-flow optimizations**[[edit](https://en.wikipedia.org/w/index.php?title=Optimizing_compiler&action=edit&section=6)]

[Data-flow](https://en.wikipedia.org/wiki/Dataflow) optimizations, based on [data-flow analysis](https://en.wikipedia.org/wiki/Data-flow_analysis), primarily depend on how certain properties of data are propagated by control edges in the [control flow graph](https://en.wikipedia.org/wiki/Control_flow_graph). Some of these include:

[**Common subexpression elimination**](https://en.wikipedia.org/wiki/Common_subexpression_elimination)

In the expression (a + b) - (a + b)/4, "common subexpression" refers to the duplicated (a + b). Compilers implementing this technique realize that (a + b) will not change, and so only calculate its value once.

[**Constant folding and propagation**](https://en.wikipedia.org/wiki/Constant_folding)

replacing expressions consisting of constants (e.g., 3 + 5) with their final value (8) at compile time, rather than doing the calculation in run-time. Used in most modern languages.

[**Induction variable recognition and elimination**](https://en.wikipedia.org/wiki/Induction_variable_recognition_and_elimination)

see discussion above about *induction variable analysis*.

[**Alias classification and pointer analysis**](https://en.wikipedia.org/wiki/Strict_aliasing)

in the presence of [pointers](https://en.wikipedia.org/wiki/Pointer_(computer_programming)), it is difficult to make any optimizations at all, since potentially any variable can have been changed when a memory location is assigned to. By specifying which pointers can alias which variables, unrelated pointers can be ignored.

[**Dead store**](https://en.wikipedia.org/wiki/Dead_store)**elimination**

removal of assignments to variables that are not subsequently read, either because the lifetime of the variable ends or because of a subsequent assignment that will overwrite the first value.

**SSA-based optimizations**[[edit](https://en.wikipedia.org/w/index.php?title=Optimizing_compiler&action=edit&section=7)]

These optimizations are intended to be done after transforming the program into a special form called Static Single Assignment (see [SSA](https://en.wikipedia.org/wiki/SSA_(compilers)) form), in which every variable is assigned in only one place. Although some function without SSA, they are most effective with SSA. Many optimizations listed in other sections also benefit with no special changes, such as register allocation.

[**Global value numbering**](https://en.wikipedia.org/wiki/Global_value_numbering)

GVN eliminates redundancy by constructing a [value graph](https://en.wikipedia.org/w/index.php?title=Value_graph_(compilers)&action=edit&redlink=1) of the program, and then determining which values are computed by equivalent expressions. GVN is able to identify some redundancy that [common subexpression elimination](https://en.wikipedia.org/wiki/Common_subexpression_elimination) cannot, and vice versa.

[**Sparse conditional constant propagation**](https://en.wikipedia.org/wiki/Sparse_conditional_constant_propagation)

Combines constant propagation, [constant folding](https://en.wikipedia.org/wiki/Constant_folding), and [dead code elimination](https://en.wikipedia.org/wiki/Dead_code_elimination), and improves upon what is possible by running them separately.[[4]](https://en.wikipedia.org/wiki/Optimizing_compiler#cite_note-4)[[5]](https://en.wikipedia.org/wiki/Optimizing_compiler#cite_note-5) This optimization symbolically executes the program, simultaneously propagating constant values and eliminating portions of the [control flow graph](https://en.wikipedia.org/wiki/Control_flow_graph) that this makes unreachable.

**Code generator optimizations**[[edit](https://en.wikipedia.org/w/index.php?title=Optimizing_compiler&action=edit&section=8)]

[**Register allocation**](https://en.wikipedia.org/wiki/Register_allocation)

The most frequently used variables should be kept in processor registers for fastest access. To find which variables to put in registers an interference-graph is created. Each variable is a vertex and when two variables are used at the same time (have an intersecting liverange) they have an edge between them. This graph is colored using for example [Chaitin's algorithm](https://en.wikipedia.org/wiki/Chaitin%27s_algorithm" \o "Chaitin's algorithm) using the same number of colors as there are registers. If the coloring fails one variable is "spilled" to memory and the coloring is retried.

[**Instruction selection**](https://en.wikipedia.org/wiki/Instruction_selection)

Most architectures, particularly [CISC](https://en.wikipedia.org/wiki/Complex_instruction_set_computer) architectures and those with many [addressing modes](https://en.wikipedia.org/wiki/Addressing_mode), offer several different ways of performing a particular operation, using entirely different sequences of instructions. The job of the instruction selector is to do a good job overall of choosing which instructions to implement which operators in the low-level [intermediate representation](https://en.wikipedia.org/wiki/Intermediate_representation) with. For example, on many processors in the [68000 family](https://en.wikipedia.org/wiki/68000_family) and on the x86 architecture, complex addressing modes can be used in statements like "lea 25(a1,d5\*4), a0", allowing a single instruction to perform a significant amount of arithmetic with less storage.

[**Instruction scheduling**](https://en.wikipedia.org/wiki/Instruction_scheduling)

Instruction scheduling is an important optimization for modern [pipelined](https://en.wikipedia.org/wiki/Instruction_pipeline) processors, which avoids stalls or bubbles in the pipeline by clustering instructions with no dependencies together, while being careful to preserve the original semantics.

[**Rematerialization**](https://en.wikipedia.org/wiki/Rematerialization)

Rematerialization recalculates a value instead of loading it from memory, preventing a memory access. This is performed in tandem with register allocation to avoid spills.

**Code factoring**

If several sequences of code are identical, or can be parameterized or reordered to be identical, they can be replaced with calls to a shared subroutine. This can often share code for subroutine set-up and sometimes tail-recursion.[[6]](https://en.wikipedia.org/wiki/Optimizing_compiler#cite_note-keil-6)

**Trampolines**

Many[[*citation needed*](https://en.wikipedia.org/wiki/Wikipedia:Citation_needed)] CPUs have smaller subroutine call instructions to access low memory. A compiler can save space by using these small calls in the main body of code. Jump instructions in low memory can access the routines at any address. This multiplies space savings from code factoring.[[6]](https://en.wikipedia.org/wiki/Optimizing_compiler#cite_note-keil-6)

**Reordering computations**

Based on [integer linear programming](https://en.wikipedia.org/wiki/Integer_linear_programming), restructuring compilers enhance data locality and expose more parallelism by reordering computations. Space-optimizing compilers may reorder code to lengthen sequences that can be factored into subroutines.

**Functional language optimizations**

Although many of these also apply to non-functional languages, they either originate in, are most easily implemented in, or are particularly critical in [functional languages](https://en.wikipedia.org/wiki/Functional_language) such as [Lisp](https://en.wikipedia.org/wiki/Lisp_programming_language) and [ML](https://en.wikipedia.org/wiki/ML_programming_language).

**Removing**[**recursion**](https://en.wikipedia.org/wiki/Recursion)

Recursion is often expensive, as a function call consumes stack space and involves some overhead related to parameter passing and flushing the instruction cache. [Tail recursive](https://en.wikipedia.org/wiki/Tail_recursion) algorithms can be converted to [iteration](https://en.wikipedia.org/wiki/Iteration), which does not have call overhead and uses a constant amount of stack space, through a process called tail recursion elimination or [tail call optimization](https://en.wikipedia.org/wiki/Tail_call_optimization). Some functional languages (e.g., [Scheme](https://en.wikipedia.org/wiki/Scheme_(programming_language)#Proper_tail_recursion) and [Erlang](https://en.wikipedia.org/wiki/Erlang_(programming_language)" \o "Erlang (programming language))) mandate that tail calls be optimized by a conforming implementation, due to their prevalence in these languages.

[**Deforestation**](https://en.wikipedia.org/wiki/Deforestation_(computer_science))**(**[**data structure**](https://en.wikipedia.org/wiki/Data_structure)**fusion)**

Because of the high level nature by which data structures are specified in functional languages such as Haskell, it is possible to combine several recursive functions which produce and consume some temporary data structure so that the data is passed directly without wasting time constructing the data structure.

**Other optimizations**

[**Bounds-checking elimination**](https://en.wikipedia.org/wiki/Bounds-checking_elimination)

Many languages, for example [Java](https://en.wikipedia.org/wiki/Java_(programming_language)), enforce bounds checking of all array accesses. This is a severe performance bottleneck on certain applications such as scientific code. Bounds-checking elimination allows the compiler to safely remove bounds checking in many situations where it can determine that the index must fall within valid bounds, for example if it is a simple loop variable.

**Branch offset optimization (machine dependent)**

Choose the shortest branch displacement that reaches target

**Code-block reordering**

Code-block reordering alters the order of the [basic blocks](https://en.wikipedia.org/wiki/Basic_block) in a program in order to reduce conditional branches and improve [locality of reference](https://en.wikipedia.org/wiki/Locality_of_reference).

[**Dead code elimination**](https://en.wikipedia.org/wiki/Dead_code_elimination)

Removes instructions that will not affect the behaviour of the program, for example definitions which have no uses, called [dead code](https://en.wikipedia.org/wiki/Dead_code). This reduces code size and eliminates unnecessary computation.

**Factoring out of invariants**

If an expression is carried out both when a condition is met and is not met, it can be written just once outside of the conditional statement. Similarly, if certain types of expressions (e.g., the assignment of a constant into a variable) appear inside a loop, they can be moved out of it because their effect will be the same no matter if they're executed many times or just once. Also known as total redundancy elimination. A more powerful optimization is [partial redundancy elimination](https://en.wikipedia.org/wiki/Partial_redundancy_elimination) (PRE).

[**Inline expansion**](https://en.wikipedia.org/wiki/Inline_expansion)**or**[**macro**](https://en.wikipedia.org/wiki/Macro_(computer_science))**expansion**

When some code invokes a [procedure](https://en.wikipedia.org/wiki/Subroutine), it is possible to directly insert the body of the procedure inside the calling code rather than transferring control to it. This saves the overhead related to procedure calls, as well as providing great opportunity for many different parameter-specific optimizations, but comes at the cost of space; the procedure body is duplicated each time the procedure is called inline. Generally, inlining is useful in performance-critical code that makes a large number of calls to small procedures. A "fewer jumps" optimization. The [statements](https://en.wikipedia.org/wiki/Statement_(computer_science)) of [imperative](https://en.wikipedia.org/wiki/Imperative_programming) programming languages are also an example of such an optimization. Although statements could be implemented with [function calls](https://en.wikipedia.org/wiki/Subroutine) they are almost always implemented with code inlining.

[**Jump threading**](https://en.wikipedia.org/wiki/Jump_threading)

In this pass, consecutive conditional jumps predicated entirely or partially on the same condition are merged.

E.g., **if** (c) { foo; } **if** (c) { bar; } to **if** (c) { foo; bar; },

and **if** (c) { foo; } **if** (!c) { bar; } to **if** (c) { foo; } **else** { bar; }.

**Macro compression**

A space optimization that recognizes common sequences of code, creates subprograms ("code macros") that contain the common code, and replaces the occurrences of the common code sequences with calls to the corresponding subprograms.[[3]](https://en.wikipedia.org/wiki/Optimizing_compiler#cite_note-MCO-3) This is most effectively done as a machine code optimization, when all the code is present. The technique was first used to conserve space in an interpretive byte stream used in an implementation of [Macro Spitbol](https://en.wikipedia.org/wiki/SPITBOL_compiler) on microcomputers.[[7]](https://en.wikipedia.org/wiki/Optimizing_compiler#cite_note-MicroSpitbol-7) The problem of determining an optimal set of macros that minimizes the space required by a given code segment is known to be [NP-complete](https://en.wikipedia.org/wiki/NP-complete),[[3]](https://en.wikipedia.org/wiki/Optimizing_compiler#cite_note-MCO-3) but efficient heuristics attain near-optimal results.[[8]](https://en.wikipedia.org/wiki/Optimizing_compiler#cite_note-8)

**Reduction of cache collisions**

(e.g., by disrupting alignment within a page)

**Stack height reduction**

Rearrange expression tree to minimize resources needed for expression evaluation.

**Test reordering**

If we have two tests that are the condition for something, we can first deal with the simpler tests (e.g. comparing a variable to something) and only then with the complex tests (e.g., those that require a function call). This technique complements [lazy evaluation](https://en.wikipedia.org/wiki/Lazy_evaluation), but can be used only when the tests are not dependent on one another. [Short-circuiting](https://en.wikipedia.org/wiki/Minimal_evaluation)semantics can make this difficult.