

# Compiler optimization

From Wikipedia, the free encyclopedia

**Compiler optimization** is the process of tuning the output of a compiler to minimize or maximize some attributes of an executable computer program. The most common requirement is to minimize the time taken to execute a program; a less common one is to minimize the amount of memory occupied. The growth of portable computers has created a market for minimizing the power consumed by a program. Compiler optimization is generally implemented using a sequence of *optimizing transformations*, algorithms which take a program and transform it to produce a semantically equivalent output program that uses less resources.

It has been shown that some code optimization problems are NP-complete, or even undecidable. In practice, factors such as the programmer's willingness to wait for the compiler to complete its task place upper limits on the optimizations that a compiler implementor might provide. (Optimization is generally a very CPU- and memory-intensive process.) In the past, computer memory limitations were also a major factor in limiting which optimizations could be performed. Because of all these factors, optimization rarely produces "optimal" output in any sense, and in fact an "optimization" may impede performance in some cases; rather, they are heuristic methods for improving resource usage in typical programs.

## Contents

- 1 Types of optimizations
- 2 Factors affecting optimization
- 3 Common themes
- 4 Specific techniques
  - 4.1 Loop optimizations
  - 4.2 Data-flow optimizations
  - 4.3 SSA-based optimizations
  - 4.4 Code generator optimizations
  - 4.5 Functional language optimizations
  - 4.6 Other optimizations
  - 4.7 Interprocedural optimizations
- 5 Problems with optimization
- 6 List of compiler optimizations
- 7 List of static code analyses
- 8 See also
- 9 References
- 10 External links

## Types of optimizations

Techniques used in optimization can be broken up among various *scopes* which can affect anything from a single statement to the entire program. Generally speaking, locally scoped techniques are easier to implement than global ones but result in smaller gains. Some examples of scopes include:

Peephole optimizations

Usually performed late in the compilation process after machine code has been generated. This form of optimization examines a few adjacent instructions (like "looking through a peephole" at the code) to see whether they can be replaced by a single instruction or a shorter sequence of instructions. For instance, a multiplication of a value by 2 might be more efficiently executed by left-shifting the value or by adding the value to itself. (This example is also an instance of strength reduction.)

#### Local optimizations

These only consider information local to a function definition. This reduces the amount of analysis that needs to be performed (saving time and reducing storage requirements) but means that worst case assumptions have to be made when function calls occur or global variables are accessed (because little information about them is available).

#### Loop optimizations

These act on the statements which make up a loop, such as a *for* loop (e.g., loop-invariant code motion). Loop optimizations can have a significant impact because many programs spend a large percentage of their time inside loops.

#### Interprocedural or whole-program optimization

These analyze all of a program's source code. The greater quantity of information extracted means that optimizations can be more effective compared to when they only have access to local information (i.e., within a single function). This kind of optimization can also allow new techniques to be performed. For instance function inlining, where a call to a function is replaced by a copy of the function body.

#### Machine code optimization

These analyze the executable task image of the program after all of a executable machine code has been linked. Some of the techniques that can be applied in a more limited scope, such as macro compression (which saves space by collapsing common sequences of instructions), are more effective when the entire executable task image is available for analysis.<sup>[1]</sup>

In addition to scoped optimizations there are two further general categories of optimization:

#### Programming language-independent vs language-dependent

Most high-level languages share common programming constructs and abstractions: decision (if, switch, case), looping (for, while, repeat.. until, do.. while), and encapsulation (structures, objects). Thus similar optimization techniques can be used across languages. However, certain language features make some kinds of optimizations difficult. For instance, the existence of pointers in C and C++ makes it difficult to optimize array accesses (see Alias analysis). However, languages such as PL/1 (that also supports pointers) nevertheless have available sophisticated optimizing compilers to achieve better performance in various other ways. Conversely, some language features make certain optimizations easier. For example, in some languages functions are not permitted to have side effects. Therefore, if a program makes several calls to the same function with the same arguments, the compiler can immediately infer that the function's result need be computed only once.

#### Machine independent vs machine dependent

Many optimizations that operate on abstract programming concepts (loops, objects, structures) are independent of the machine targeted by the compiler, but many of the most effective optimizations are those that best exploit special features of the target platform.

The following is an instance of a local machine dependent optimization. To set a register to 0, the obvious way is to use the constant '0' in an instruction that sets a register value to a constant. A less obvious way is to XOR a register with itself. It is up to the compiler to know which instruction variant to use. On many RISC

machines, both instructions would be equally appropriate, since they would both be the same length and take the same time. On many other microprocessors such as the Intel x86 family, it turns out that the XOR variant is shorter and probably faster, as there will be no need to decode an immediate operand, nor use the internal "immediate operand register". (A potential problem with this is that XOR may introduce a data dependency on the previous value of the register, causing a pipeline stall. However, processors often have XOR of a register with itself as a special case that doesn't cause stalls.)

## Factors affecting optimization

### The machine itself

Many of the choices about which optimizations can and should be done depend on the characteristics of the target machine. It is sometimes possible to parameterize some of these machine dependent factors, so that a single piece of compiler code can be used to optimize different machines just by altering the machine description parameters. GCC is a compiler which exemplifies this approach.

### The architecture of the target CPU

Number of CPU registers: To a certain extent, the more registers, the easier it is to optimize for performance. Local variables can be allocated in the registers and not on the stack.

Temporary/intermediate results can be left in registers without writing to and reading back from memory.

- RISC vs CISC: CISC instruction sets often have variable instruction lengths, often have a larger number of possible instructions that can be used, and each instruction could take differing amounts of time. RISC instruction sets attempt to limit the variability in each of these: instruction sets are usually constant length, with few exceptions, there are usually fewer combinations of registers and memory operations, and the instruction issue rate (the number of instructions completed per time period, usually an integer multiple of the clock cycle) is usually constant in cases where memory latency is not a factor. There may be several ways of carrying out a certain task, with CISC usually offering more alternatives than RISC. Compilers have to know the relative costs among the various instructions and choose the best instruction sequence (see instruction selection).
- Pipelines: A pipeline is essentially a CPU broken up into an assembly line. It allows use of parts of the CPU for different instructions by breaking up the execution of instructions into various stages: instruction decode, address decode, memory fetch, register fetch, compute, register store, etc. One instruction could be in the register store stage, while another could be in the register fetch stage. Pipeline conflicts occur when an instruction in one stage of the pipeline depends on the result of another instruction ahead of it in the pipeline but not yet completed. Pipeline conflicts can lead to pipeline stalls: where the CPU wastes cycles waiting for a conflict to resolve.

Compilers can *schedule*, or reorder, instructions so that pipeline stalls occur less frequently.

- Number of functional units: Some CPUs have several ALUs and FPUs. This allows them to execute multiple instructions simultaneously. There may be restrictions on which instructions can pair with which other instructions ("pairing" is the simultaneous execution of two or more instructions), and which functional unit can execute which instruction. They also have issues similar to pipeline conflicts.

Here again, instructions have to be scheduled so that the various functional units are fully fed with instructions to execute.

### The architecture of the machine

- Cache size (256 kiB–12 MiB) and type (direct mapped, 2-/4-/8-/16-way associative, fully associative): Techniques such as inline expansion and loop unrolling may increase the size of the generated code and reduce code locality. The program may slow down drastically if a highly utilized section of code (like inner loops in various algorithms) suddenly cannot fit in the cache. Also, caches which are not fully associative have higher chances of cache collisions even in an unfilled cache.
- Cache/Memory transfer rates: These give the compiler an indication of the penalty for cache misses. This is used mainly in specialized applications.

## Intended use of the generated code

### Debugging

While a programmer is writing an application, he will recompile and test often, and so compilation must be fast. This is one reason most optimizations are deliberately avoided during the test/debugging phase. Also, program code is usually "stepped through" (see Program animation) using a symbolic debugger, and optimizing transformations, particularly those that reorder code, can make it difficult to relate the output code with the line numbers in the original source code. This can confuse both the debugging tools and the programmers using them.

### General purpose use

Prepackaged software is very often expected to be executed on a variety of machines and CPUs that may share the same instruction set, but have different timing, cache or memory characteristics. So, the code may not be tuned to any particular CPU, or may be tuned to work best on the most popular CPU and yet still work acceptably well on other CPUs.

### Special-purpose use

If the software is compiled to be used on one or a few very similar machines, with known characteristics, then the compiler can heavily tune the generated code to those specific machines (if such options are available). Important special cases include code designed for parallel and vector processors, for which special parallelizing compilers are employed.

### Embedded systems

These are a common case of special-purpose use. Embedded software can be tightly tuned to an exact CPU and memory size. Also, system cost or reliability may be more important than the code's speed. So, for example, compilers for embedded software usually offer options that reduce code size at the expense of speed, because memory is the main cost of an embedded computer. The code's timing may need to be predictable, rather than as fast as possible, so code caching might be disabled, along with compiler optimizations that require it.

## Common themes

To a large extent, compiler optimization techniques have the following themes, which sometimes conflict.

### Optimize the common case

The common case may have unique properties that allow a *fast path* at the expense of a *slow path*. If the fast path is taken most often, the result is better over-all performance.

### Avoid redundancy

Reuse results that are already computed and store them for use later, instead of recomputing them.

### Less code

Remove unnecessary computations and intermediate values. Less work for the CPU, cache, and memory usually results in faster execution. Alternatively, in embedded systems, less code brings a lower product

cost.

Fewer jumps by using *straight line code*, also called *branch-free code*

Less complicated code. Jumps (conditional or unconditional branches) interfere with the prefetching of instructions, thus slowing down code. Using inlining or loop unrolling can reduce branching, at the cost of increasing binary file size by the length of the repeated code. This tends to merge several basic blocks into one.

Locality

Code and data that are accessed closely together in time should be placed close together in memory to increase spatial locality of reference.

Exploit the memory hierarchy

Accesses to memory are increasingly more expensive for each level of the memory hierarchy, so place the most commonly used items in registers first, then caches, then main memory, before going to disk.

Parallelize

Reorder operations to allow multiple computations to happen in parallel, either at the instruction, memory, or thread level.

More precise information is better

The more precise the information the compiler has, the better it can employ any or all of these optimization techniques.

Runtime metrics can help

Information gathered during a test run can be used in profile-guided optimization. Information gathered at runtime (ideally with minimal overhead) can be used by a JIT compiler to dynamically improve optimization.

Strength reduction

Replace complex or difficult or expensive operations with simpler ones. For example, replacing division by a constant with multiplication by its reciprocal, or using induction variable analysis to replace multiplication by a loop index with addition.

## Specific techniques

### Loop optimizations

*Main article: Loop optimization*

Some optimization techniques primarily designed to operate on loops include:

Induction variable analysis

Roughly, if a variable in a loop is a simple 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, and also may allow the index variable's definitions to become dead code. This information is also useful for bounds-checking elimination and dependence analysis, among other things.

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, both of the data being accessed in the loop

and the code in the loop's body.

#### Loop fusion or loop combining

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

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. Additionally, if the initial condition is known at compile-time and is known to be side-effect-free, the *if* guard can be skipped.

#### 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, depending on the array's layout.

#### 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, because not all code is safe to be hoisted outside the loop.

#### 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

Loop reversal reverses the order in which values are assigned to the index variable. This is a subtle optimization which can help eliminate dependencies and thus enable other optimizations.

#### 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

Loop splitting attempts to simplify a loop or eliminate dependencies 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*, which can simplify a loop with a problematic first iteration by performing that iteration separately before entering the loop.

#### 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

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

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

Data flow optimizations, based on Data-flow analysis, primarily depend on how certain properties of data are propagated by control edges in the control flow graph. Some of these include:

#### 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  $(a+b)$  won't change, and as such, only calculate its value once.

#### Constant folding and propagation

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

see discussion above about *induction variable analysis*.

#### Alias classification and pointer analysis

in the presence of pointers, 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 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

These optimizations are intended to be done after transforming the program into a special form called static single assignment (see SSA 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

GVN eliminates redundancy by constructing a value graph of the program, and then determining which values are computed by equivalent expressions. GVN is able to identify some redundancy that common subexpression elimination cannot, and vice versa.

#### Sparse conditional constant propagation

Effectively equivalent to iteratively performing constant propagation, constant folding, and dead code elimination until there is no change, but is much more efficient. This optimization symbolically executes the program, simultaneously propagating constant values and eliminating portions of the control flow graph that this makes unreachable.

## Code generator optimizations

### 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 live range) they have an edge between them. This graph is colored using for example 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

Most architectures, particularly CISC architectures and those with many addressing modes, 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 with. For example, on many processors in the 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

Instruction scheduling is an important optimization for modern pipelined 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

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.<sup>[2]</sup>

### Trampolines

Many 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.<sup>[2]</sup>

### Reordering computations

Based on 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 such as Lisp and ML.

### Removing 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 algorithms can be converted to iteration, which does not have call overhead and uses a constant amount of stack space, through a process



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. Some functional languages (e.g., Scheme and Erlang) mandate that tail calls be optimized by a conforming implementation, due to their prevalence in these languages.

#### Deforestation (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

*Please help separate and categorize these further and create detailed pages for them, especially the more complex ones, or link to one where one exists.*

#### Bounds-checking elimination

Many languages, for example Java, 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 independent)

Choose the shortest branch displacement that reaches target

#### Code-block reordering

Code-block reordering alters the order of the basic blocks in a program in order to reduce conditional branches and improve locality of reference.

#### Dead code elimination

Removes instructions that will not affect the behaviour of the program, for example definitions which have no uses, called 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 (PRE).

#### Inline expansion or macro expansion

When some code invokes a procedure, 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.

#### 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.<sup>[1]</sup> 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 on microcomputers.<sup>[3]</sup> 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,<sup>[1]</sup> but efficient heuristics attain near-optimal results.<sup>[4]</sup>

## 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, but can be used only when the tests are not dependent on one another. Short-circuiting semantics can make this difficult.

# Interprocedural optimizations

Interprocedural optimization works on the entire program, across procedure and file boundaries. It works tightly with intraprocedural counterparts, carried out with the cooperation of a local part and global part. Typical interprocedural optimizations are: procedure inlining, interprocedural dead code elimination, interprocedural constant propagation, and procedure reordering. As usual, the compiler needs to perform interprocedural analysis before its actual optimizations. Interprocedural analyses include alias analysis, array access analysis, and the construction of a call graph.

Interprocedural optimization is common in modern commercial compilers from SGI, Intel, Microsoft, and Sun Microsystems. For a long time the open source GCC was criticized<sup>[citation needed]</sup> for a lack of powerful interprocedural analysis and optimizations, though this is now improving.<sup>[citation needed]</sup> Another good open source compiler with full analysis and optimization infrastructure is Open64, which is used by many organizations for research and for commercial purposes.

Due to the extra time and space required by interprocedural analysis, most compilers do not perform it by default. Users must use compiler options explicitly to tell the compiler to enable interprocedural analysis and other expensive optimizations.

# Problems with optimization

Early in the history of compilers, compiler optimizations were not as good as hand-written ones. As compiler technologies have improved, good compilers can often generate better code than human programmers, and good post pass optimizers can improve highly hand-optimized code even further. For RISC CPU architectures, and even more so for VLIW hardware, compiler optimization is the key for obtaining efficient code, because RISC instruction sets are so compact that it is hard for a human to manually schedule or combine small instructions to get efficient results. Indeed, these architectures were designed to rely on

compiler writers for adequate performance.

However, optimizing compilers are by no means perfect. There is no way that a compiler can guarantee that, for all program source code, the fastest (or smallest) possible equivalent compiled program is output; such a compiler is fundamentally impossible because it would solve the halting problem.

This may be proven by considering a call to a function, `foo()`. This function returns nothing and does not have side effects (no I/O, does not modify global variables and "live" data structures, etc.). The fastest possible equivalent program would be simply to eliminate the function call. However, if the function `foo()` in fact does *not* return, then the program with the call to `foo()` would be different from the program without the call; the optimizing compiler will then have to determine this by solving the halting problem.

Additionally, there are a number of other more practical issues with optimizing compiler technology:

- Optimizing compilers focus on relatively shallow constant-factor performance improvements and do not typically improve the algorithmic complexity of a solution. For example, a compiler will not change an implementation of bubble sort to use mergesort instead.
- Compilers usually have to support a variety of conflicting objectives, such as cost of implementation, compilation speed and quality of generated code.
- A compiler typically only deals with a part of a program at a time, often the code contained within a single file or module; the result is that it is unable to consider contextual information that can only be obtained by processing the other files.
- The overhead of compiler optimization: Any extra work takes time; whole-program optimization is time consuming for large programs.
- The often complex interaction between optimization phases makes it difficult to find an optimal sequence in which to execute the different optimization phases.

Work to improve optimization technology continues. One approach is the use of so-called post-pass optimizers (some commercial versions of which date back to mainframe software of the late 1970s<sup>[5]</sup>). These tools take the executable output by an "optimizing" compiler and optimize it even further. Post pass optimizers usually work on the assembly language or machine code level (contrast with compilers that optimize intermediate representations of programs). The performance of post pass compilers are limited by the fact that much of the information available in the original source code is not always available to them.

As processor performance continues to improve at a rapid pace, while memory bandwidth improves more slowly, optimizations that reduce memory bandwidth (even at the cost of making the processor execute relatively more instructions) will become more useful. Examples of this, already mentioned above, include loop nest optimization and rematerialization.

## List of compiler optimizations

- Automatic programming
- automatic parallelization
- Constant folding
- Algebraic simplifications:
- Value numbering
- Copy propagation
- Constant propagation
- Sparse conditional constant propagation

- Common subexpression elimination (CSE)
- Partial redundancy elimination
- Dead code elimination
- Induction variable elimination, strength reduction
- Loop optimizations
  - Loop invariant code motion
  - Loop unrolling
- Software pipelining
- Inlining
- Code generator
  - Register allocation: local and global
  - Instruction scheduling
  - Branch predication
  - Tail merging and cross jumping
  - Machine idioms and instruction combining
- Vectorization
- Phase ordering
- Profile-guided optimization
- Macro compression

## List of static code analyses

- Alias analysis
- Pointer analysis
- Shape analysis
- Escape analysis
- Array access analysis
- Dependence analysis
- Control flow analysis
- Data flow analysis
  - Use-define chain analysis
  - Live variable analysis
  - Available expression analysis

## See also

- Algorithmic efficiency
- Full employment theorem
- Just-in-time compilation (JIT)

## References

1. <sup>^</sup> <sup>*a*</sup> <sup>*b*</sup> <sup>*c*</sup> Clinton F. Goss (June 1986). "Machine Code Optimization - Improving Executable Object Code" (<http://www.ClintGoss.com/mco/>) . pp. 112. <http://www.ClintGoss.com/mco/>. Retrieved 24-Mar-2011.
2. <sup>^</sup> <sup>*a*</sup> <sup>*b*</sup> Cx51 Compiler Manual, version 09.2001, p155, Keil Software Inc.
3. <sup>^</sup> Robert B. K. Dewar; Martin Charles Golumbic; Clinton F. Goss (October 1979). *MICRO SPITBOL*. Computer Science Department Technical Report. **No. 11**. Courant Institute of Mathematical Sciences.
4. <sup>^</sup> Martin Charles Golumbic; Robert B. K. Dewar; Clinton F. Goss (1980). "Macro Substitutions in MICRO

11. "Machine Compiler Optimizations", Robert D. M. Bell, Chapter 11, 1988, "Machine Optimizations in Theory and Practice", SPITBOL - a Combinatorial Analysis". *Proc. 11th Southeastern Conference on Combinatorics, Graph Theory and Computing, Congressus Numerantium, Utilitas Math., Winnipeg, Canada* **29**: 485–495.
5. ^ <http://portal.acm.org/citation.cfm?id=358728.358732>

## External links

- Optimization manuals (<http://www.agner.org/optimize/#manuals>) by Agner Fog - documentation about x86 processor architecture and low-level code optimization
- Citations from CiteSeer (<http://citeseer.ist.psu.edu/Programming/CompilerOptimization/>)

Retrieved from "[http://en.wikipedia.org/wiki/Compiler\\_optimization](http://en.wikipedia.org/wiki/Compiler_optimization)"

Categories: Compiler optimizations | Programming language implementation

---

- This page was last modified on 28 August 2011 at 07:13.
  - Text is available under the Creative Commons Attribution-ShareAlike License; additional terms may apply. See Terms of use for details.
- Wikipedia® is a registered trademark of the Wikimedia Foundation, Inc., a non-profit organization.