WIKIPEDIA

Program optimization

In computer science, **program optimization** or **software optimization** is the process of modifying a software system to make some aspect of it work more efficiently or use fewer resources.[1] In general, a computer program may be optimized so that it executes more rapidly, or to make it capable of operating with less memory storage or other resources, or draw less power.

Contents

General

Levels of optimization

- Design level
- Algorithms and data structures
- Source code level
- Build level
- Compile level
- Assembly level
- Run time
- Platform dependent and independent optimizations

Strength reduction

Trade-offs

Bottlenecks

When to optimize

Macros

Automated and manual optimization

Time taken for optimization

References

External links

General

Although the word "optimization" shares the same root as "optimal", it is rare for the process of optimization to produce a truly optimal system. A system can generally be made optimal not in absolute terms, but only with respect to a given quality metric, which may be in contrast with other possible metrics. As a result, the optimized system will typically only be optimal in one application or for one audience. One might reduce the amount of time that a program takes to perform some task at the price of making it consume more memory. In an application where memory space is at a premium, one might deliberately choose a slower algorithm in order to use less memory. Often there is no "one size fits all" design which works well in all cases, so engineers make trade-offs to optimize the attributes of greatest interest. Additionally, the effort required to make a piece of software completely optimal — incapable of any further improvement — is almost always more than is reasonable for the benefits that would be accrued; so the process of optimization may be halted before a completely optimal solution has been reached. Fortunately, it is often the case that the greatest improvements come early in the process.

Even for a given quality metric (such as execution speed), most methods of optimization only improve the result; they have no pretense of producing optimal output. Superoptimization is the process of finding truly optimal output.

Levels of optimization

Optimization can occur at a number of levels. Typically the higher levels have greater impact, and are harder to change later on in a project, requiring significant changes or a complete rewrite if they need to be changed. Thus optimization can typically proceed via refinement from higher to lower, with initial gains being larger and achieved with less work, and later gains being smaller and requiring more work. However, in some cases overall performance depends on performance of very low-level portions of a program, and small changes at a late stage or early consideration of low-level details can have outsized impact. Typically some consideration is given to efficiency throughout a project – though this varies significantly – but major optimization is often considered a refinement to be done late, if ever. On longer-running projects there are typically cycles of optimization, where improving one area reveals limitations in another, and these are typically curtailed when performance is acceptable or gains become too small or costly.

As performance is part of the specification of a program – a program that is unusably slow is not fit for purpose: a video game with 60 Hz (frames-per-second) is acceptable, but 6 frames-per-second is unacceptably choppy – performance is a consideration from the start, to ensure that the system is able to deliver sufficient performance, and early prototypes need to have roughly acceptable performance for there to be confidence that the final system will (with optimization) achieve acceptable performance. This is sometimes omitted in the belief that optimization can always be done later, resulting in prototype systems that are far too slow – often by an order of magnitude or more – and systems that ultimately are failures because they architecturally cannot achieve their performance goals, such as the Intel 432 (1981); or ones that take years of work to achieve acceptable performance, such as Java (1995), which only achieved acceptable performance with HotSpot (1999). The degree to which performance changes between prototype and production system, and how amenable it is to optimization, can be a significant source of uncertainty and risk.

Design level

At the highest level, the design may be optimized to make best use of the available resources, given goals, constraints, and expected use/load. The architectural design of a system overwhelmingly affects its performance. For example, a system that is network latency-bound (where network latency is the main constraint on overall performance) would be optimized to minimize network trips, ideally making a single request (or no requests, as in a push protocol) rather than multiple roundtrips. Choice of design depends on the goals: when designing a compiler, if fast compilation is the key priority, a one-pass compiler is faster than a multi-pass compiler (assuming same work), but if speed of output code is the goal, a slower multi-pass compiler fulfills the goal better, even though it takes longer itself. Choice of platform and programming language occur at this level, and changing them frequently requires a complete rewrite, though a modular system may allow rewrite of only some component – for example, a Python program may rewrite performance-critical sections in C. In a distributed system, choice of architecture (client-server, peer-to-peer, etc.) occurs at the design level, and may be difficult to change, particularly if all components cannot be replaced in sync (e.g., old clients).

Algorithms and data structures

Given an overall design, a good choice of efficient algorithms and data structures, and efficient implementation of these algorithms and data structures comes next. After design, the choice of algorithms and data structures affects efficiency more than any other aspect of the program. Generally data structures are more difficult to change than algorithms, as a data structure assumption and its performance assumptions are used throughout the program, though this can be minimized by the use of abstract data types in function definitions, and keeping the concrete data structure definitions restricted to a few places.

For algorithms, this primarily consists of ensuring that algorithms are constant O(1), logarithmic O(log *n*), linear O(*n*), or in some cases log-linear O(*n* log *n*) in the input (both in space and time). Algorithms with quadratic complexity O(n^2) fail to scale, and even linear algorithms cause problems if repeatedly called, and are typically replaced with constant or logarithmic if possible.

Beyond asymptotic order of growth, the constant factors matter: an asymptotically slower algorithm may be faster or smaller (because simpler) than an asymptotically faster algorithm when they are both faced with small input, which may be the case that occurs in reality. Often a hybrid algorithm will provide the best performance, due to this tradeoff changing with size.

A general technique to improve performance is to avoid work. A good example is the use of a fast path for common cases, improving performance by avoiding unnecessary work. For example, using a simple text layout algorithm for Latin text, only switching to a complex layout algorithm for complex scripts, such as Devanagari. Another important technique is caching, particularly memoization, which avoids redundant computations. Because of the importance of caching, there are often many levels of caching in a system, which can cause problems from memory use, and correctness issues from stale caches.

Source code level

Beyond general algorithms and their implementation on an abstract machine, concrete source code level choices can make a significant difference. For example, on early C compilers, `while(1)` was slower than `for(;;)` for an unconditional loop, because `while(1)` evaluated 1 and then had a conditional jump which tested if it was true, while `for (;;)` had an unconditional jump . Some optimizations (such as this one) can nowadays be performed by optimizing compilers. This depends on the source language, the target machine language, and the compiler, and can be both difficult to understand or predict and changes over time; this is a key place where understanding of compilers and machine code can improve performance. Loop-invariant code motion and return value optimization are examples of optimizations that reduce the need for auxiliary variables and can even result in faster performance by avoiding round-about optimizations.

Build level

Between the source and compile level, directives and build flags can be used to tune performance options in the source code and compiler respectively, such as using preprocessor defines to disable unneeded software features, optimizing for specific processor models or hardware capabilities, or predicting branching, for instance. Source-based software distribution systems such as BSD's Ports and Gentoo's Portage can take advantage of this form of optimization.

Compile level

Use of an optimizing compiler tends to ensure that the executable program is optimized at least as much as the compiler can predict.

Assembly level

At the lowest level, writing code using an assembly language, designed for a particular hardware platform can produce the most efficient and compact code if the programmer takes advantage of the full repertoire of machine instructions. Many operating systems used on embedded systems have been traditionally written in assembler code for this reason. Programs (other than very small programs) are seldom written from start to finish in assembly due to the time and cost involved. Most are compiled down from a high level language to assembly and hand optimized from there. When efficiency and size are less important large parts may be written in a high-level language.

With more modern optimizing compilers and the greater complexity of recent CPUs, it is harder to write more efficient code than what the compiler generates, and few projects need this "ultimate" optimization step.

Much code written today is intended to run on as many machines as possible. As a consequence, programmers and compilers don't always take advantage of the more efficient instructions provided by newer CPUs or quirks of older models. Additionally, assembly code tuned for a particular processor without using such instructions might still be suboptimal on a different processor, expecting a different tuning of the code.

Typically today rather than writing in assembly language, programmers will use a disassembler to analyze the output of a compiler and change the high-level source code so that it can be compiled more efficiently, or understand why it is inefficient.

Run time

Just-in-time compilers can produce customized machine code based on run-time data, at the cost of compilation overhead. This technique dates to the earliest regular expression engines, and has become widespread with Java HotSpot and V8 for JavaScript. In some cases adaptive optimization may be able to perform run time optimization exceeding the capability of static compilers by dynamically adjusting parameters according to the actual input or other factors.

Profile-guided optimization is an ahead-of-time (AOT) compilation optimization technique based on run time profiles, and is similar to a static "average case" analog of the dynamic technique of adaptive optimization.

Self-modifying code can alter itself in response to run time conditions in order to optimize code; this was more common in assembly language programs.

Some CPU designs can perform some optimizations at run time. Some examples include Out-of-order execution, Speculative execution, Instruction pipelines, and Branch predictors. Compilers can help the program take advantage of these CPU features, for example through instruction scheduling.

Platform dependent and independent optimizations

Code optimization can be also broadly categorized as platform-dependent and platform-independent techniques. While the latter ones are effective on most or all platforms, platform-dependent techniques use specific properties of one platform, or rely on parameters depending on the single platform or even on the single processor. Writing or producing different versions of the same code for different processors might therefore be needed. For instance, in the case of compile-level optimization, platform-independent techniques are generic techniques (such as loop unrolling, reduction in function calls, memory efficient routines, reduction in conditions, etc.), that impact most CPU architectures in a similar way. A great example of platform-independent optimization has been shown with inner for loop, where it was observed that a loop with an inner for loop performs more computations per unit time than a loop without it or one with an inner while loop.[2]

Generally, these serve to reduce the total instruction path length required to complete the program and/or reduce total memory usage during the process. On the other hand, platform-dependent techniques involve instruction scheduling, instruction-level parallelism, data-level parallelism, cache optimization techniques (i.e., parameters that differ among various platforms) and the optimal instruction scheduling might be different even on different processors of the same architecture.

Strength reduction

Computational tasks can be performed in several different ways with varying efficiency. A more efficient version with equivalent functionality is known as a strength reduction. For example, consider the following C code snippet whose intention is to obtain the sum of all integers from 1 to *N*:

```
int i, sum = 0;
for (i = 1; i <= N; ++i) {
  sum += i;
}
printf("sum: %d\n", sum);
```

This code can (assuming no arithmetic overflow) be rewritten using a mathematical formula like:

```
int sum = N * (1 + N) / 2;
printf("sum: %d\n", sum);
```

The optimization, sometimes performed automatically by an optimizing compiler, is to select a method (algorithm) that is more computationally efficient, while retaining the same functionality. See algorithmic efficiency for a discussion of some of these techniques. However, a significant improvement in performance can often be achieved by removing extraneous functionality.

Optimization is not always an obvious or intuitive process. In the example above, the "optimized" version might actually be slower than the original version if *N* were sufficiently small and the particular hardware happens to be much faster at performing addition and looping operations than multiplication and division.

Trade-offs

In some cases, however, optimization relies on using more elaborate algorithms, making use of "special cases" and special "tricks" and performing complex trade-offs. A "fully optimized" program might be more difficult to comprehend and hence may contain more faults than unoptimized versions. Beyond eliminating obvious antipatterns, some code level optimizations decrease maintainability.

Optimization will generally focus on improving just one or two aspects of performance: execution time, memory usage, disk space, bandwidth, power consumption or some other resource. This will usually require a trade-off — where one factor is optimized at the expense of others. For example, increasing the size of cache improves run time performance, but also increases the memory consumption. Other common trade-offs include code clarity and conciseness.

There are instances where the programmer performing the optimization must decide to make the software better for some operations but at the cost of making other operations less efficient. These trade-offs may sometimes be of a non-technical nature — such as when a competitor has published a benchmark result that must be beaten in order to improve commercial success but comes perhaps with the burden of making normal usage of the software less efficient. Such changes are sometimes jokingly referred to as *pessimizations*.

Bottlenecks

Optimization may include finding a bottleneck in a system – a component that is the limiting factor on performance. In terms of code, this will often be a hot spot – a critical part of the code that is the primary consumer of the needed resource – though it can be another factor, such as I/O latency or network bandwidth.

In computer science, resource consumption often follows a form of power law distribution, and the Pareto principle can be applied to resource optimization by observing that 80% of the resources are typically used by 20% of the operations.[3] In software engineering, it is often a better approximation that 90% of the execution time of a computer program is spent executing 10% of the code (known as the 90/10 law in this context).

More complex algorithms and data structures perform well with many items, while simple algorithms are more suitable for small amounts of data — the setup, initialization time, and constant factors of the more complex algorithm can outweigh the benefit, and thus a hybrid algorithm or adaptive algorithm may be faster than any single algorithm. A performance profiler can be used to narrow down decisions about which functionality fits which conditions.[4]

In some cases, adding more memory can help to make a program run faster. For example, a filtering program will commonly read each line and filter and output that line immediately. This only uses enough memory for one line, but performance is typically poor, due to the latency of each disk read. Caching the result is similarly effective, though also requiring larger memory use.

When to optimize

Optimization can reduce readability and add code that is used only to improve the performance. This may complicate programs or systems, making them harder to maintain and debug. As a result, optimization or performance tuning is often performed at the end of the development stage.

Donald Knuth made the following two statements on optimization:

> "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%"[5]

(He also attributed the quote to Tony Hoare several years later,[6] although this might have been an error as Hoare disclaims having coined the phrase.[7])

> "In established engineering disciplines a 12% improvement, easily obtained, is never considered marginal and I believe the same viewpoint should prevail in software engineering"[5]

"Premature optimization" is a phrase used to describe a situation where a programmer lets performance considerations affect the design of a piece of code. This can result in a design that is not as clean as it could have been or code that is incorrect, because the code is complicated by the optimization and the programmer is distracted by optimizing.

When deciding whether to optimize a specific part of the program, Amdahl's Law should always be considered: the impact on the overall program depends very much on how much time is actually spent in that specific part, which is not always clear from looking at the code without a performance analysis.

A better approach is therefore to design first, code from the design and then profile/benchmark the resulting code to see which parts should be optimized. A simple and elegant design is often easier to optimize at this stage, and profiling may reveal unexpected performance problems that would not have been addressed by premature optimization.

In practice, it is often necessary to keep performance goals in mind when first designing software, but the programmer balances the goals of design and optimization.

Modern compilers and operating systems are so efficient that the intended performance increases often fail to materialize. As an example, caching data at the application level that is again cached at the operating system level does not yield improvements in execution. Even so, it is a rare case when the programmer will remove failed optimizations from production code. It is also true that advances in hardware will more often than not obviate any potential improvements, yet the obscuring code will persist into the future long after its purpose has been negated.

Macros

Optimization during code development using macros takes on different forms in different languages.

In some procedural languages, such as C and C++, macros are implemented using token substitution. Nowadays, inline functions can be used as a type safe alternative in many cases. In both cases, the inlined function body can then undergo further compile-time optimizations by the compiler, including constant folding, which may move some computations to compile time.

In many functional programming languages macros are implemented using parse-time substitution of parse trees/abstract syntax trees, which it is claimed makes them safer to use. Since in many cases interpretation is used, that is one way to ensure that such computations are only performed at parse-time, and sometimes the only way.

Lisp originated this style of macro, and such macros are often called "Lisp-like macros." A similar effect can be achieved by using template metaprogramming in C++.

In both cases, work is moved to compile-time. The difference between C macros on one side, and Lisp-like macros and C++ template metaprogramming on the other side, is that the latter tools allow performing arbitrary computations at compile-time/parse-time, while expansion of C macros does not perform any computation, and relies on the optimizer ability to perform it. Additionally, C macros do not directly support recursion or iteration, so are not Turing complete.

As with any optimization, however, it is often difficult to predict where such tools will have the most impact before a project is complete.

Automated and manual optimization

See also Category:Compiler optimizations

Optimization can be automated by compilers or performed by programmers. Gains are usually limited for local optimization, and larger for global optimizations. Usually, the most powerful optimization is to find a superior algorithm.

Optimizing a whole system is usually undertaken by programmers because it is too complex for automated optimizers. In this situation, programmers or system administrators explicitly change code so that the overall system performs better. Although it can produce better efficiency, it is far more expensive than automated optimizations. Since many parameters influence the program performance, the program optimization space is large. Meta-heuristics and machine learning are used to address the complexity of program optimization.[8]

Use a profiler (or performance analyzer) to find the sections of the program that are taking the most resources — the *bottleneck*. Programmers sometimes believe they have a clear idea of where the bottleneck is, but intuition is frequently wrong. Optimizing an unimportant piece of code will typically do little to help the overall performance.

When the bottleneck is localized, optimization usually starts with a rethinking of the algorithm used in the program. More often than not, a particular algorithm can be specifically tailored to a particular problem, yielding better performance than a generic algorithm. For example, the task of sorting a huge list of items is usually done with a quicksort routine, which is one of the most efficient generic algorithms. But if some characteristic of the items is exploitable (for example, they are already arranged in some particular order), a different method can be used, or even a custom-made sort routine.

After the programmer is reasonably sure that the best algorithm is selected, code optimization can start. Loops can be unrolled (for lower loop overhead, although this can often lead to *lower* speed if it overloads the CPU cache), data types as small as possible can be used, integer arithmetic can be used instead of floating-point, and so on. (See algorithmic efficiency article for these and other techniques.)

Performance bottlenecks can be due to language limitations rather than algorithms or data structures used in the program. Sometimes, a critical part of the program can be re-written in a different programming language that gives more direct access to the underlying machine. For example, it is common for very high-level languages like Python to have modules written in C for greater speed. Programs already written in C can have modules written in assembly. Programs written in D can use the inline assembler.

Rewriting sections "pays off" in these circumstances because of a general "rule of thumb" known as the 90/10 law, which states that 90% of the time is spent in 10% of the code, and only 10% of the time in the remaining 90% of the code. So, putting intellectual effort into optimizing just a small part of the program can have a huge effect on the overall speed — if the correct part(s) can be located.

Manual optimization sometimes has the side effect of undermining readability. Thus code optimizations should be carefully documented (preferably using in-line comments), and their effect on future development evaluated.

The program that performs an automated optimization is called an **optimizer**. Most optimizers are embedded in compilers and operate during compilation. Optimizers can often tailor the generated code to specific processors.

Today, automated optimizations are almost exclusively limited to compiler optimization. However, because compiler optimizations are usually limited to a fixed set of rather general optimizations, there is considerable demand for optimizers which can accept descriptions of problem and language-specific optimizations, allowing an engineer to specify custom optimizations. Tools that accept descriptions of optimizations are called program transformation systems and are beginning to be applied to real software systems such as C++.

Some high-level languages (Eiffel, Esterel) optimize their programs by using an intermediate language.

Grid computing or distributed computing aims to optimize the whole system, by moving tasks from computers with high usage to computers with idle time.

Time taken for optimization

Sometimes, the time taken to undertake optimization therein itself may be an issue.

Optimizing existing code usually does not add new features, and worse, it might add new bugs in previously working code (as any change might). Because manually optimized code might sometimes have less "readability" than unoptimized code, optimization might impact maintainability of it as well. Optimization comes at a price and it is important to be sure that the investment is worthwhile.

An automatic optimizer (or optimizing compiler, a program that performs code optimization) may itself have to be optimized, either to further improve the efficiency of its target programs or else speed up its own operation. A compilation performed with optimization "turned on" usually takes longer, although this is usually only a problem when programs are quite large.

In particular, for just-in-time compilers the performance of the run time compile component, executing together with its target code, is the key to improving overall execution speed.

References

- Jon Bentley: *Writing Efficient Programs*, ISBN 0-13-970251-2.
- Donald Knuth: *The Art of Computer Programming*

1. Robert Sedgewick, *Algorithms*, 1984, p. 84
2. Inner loop program construct: A faster way for program execution (https://doi.org/10.1515/comp-2018-0004)
3. Wescott, Bob (2013). *The Every Computer Performance Book, Chapter 3: Useful laws*. CreateSpace. ISBN 978-1482657753.
4. "Performance Profiling with a Focus" (http://www.developforperformance.com/PerformanceProfilingWithAFocus.html#FittingTheSituation). Retrieved 15 August 2017.
5. Knuth, Donald (December 1974). "Structured Programming with go to Statements". *ACM Computing Surveys*. **6** (4): 268. CiteSeerX 10.1.1.103.6084 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.103.6084). doi:10.1145/356635.356640 (https://doi.org/10.1145%2F356635.356640).
6. *The Errors of Tex*, in *Software—Practice & Experience*, Volume 19, Issue 7 (July 1989), pp. 607–685, reprinted in his book Literate Programming (p. 276)
7. Tony Hoare, a 2004 email (http://hans.gerwitz.com/2004/08/12/premature-optimization-is-the-root-of-all-evil.html)
8. Memeti, Suejb; Pllana, Sabri; Binotto, Alécio; Kołodziej, Joanna; Brandic, Ivona (26 April 2018). "Using meta-heuristics and machine learning for software optimization of parallel computing systems: a systematic literature review". *Computing*. Springer Vienna. **101** (8): 893–936. arXiv:1801.09444 (https://arxiv.org/abs/1801.09444). Bibcode:2018arXiv180109444M (https://ui.adsabs.harvard.edu/abs/2018arXiv180109444M). doi:10.1007/s00607-018-0614-9 (https://doi.org/10.1007%2Fs00607-018-0614-9).

External links

- How To Write Fast Numerical Code: A Small Introduction (http://www.ece.cmu.edu/~franzf/papers/gttse07.pdf)
- "What Every Programmer Should Know About Memory" (http://people.redhat.com/drepper/cpumemory.pdf) by Ulrich Drepper — explains the structure of modern memory subsystems and suggests how to utilize them efficiently
- "Linux Multicore Performance Analysis and Optimization in a Nutshell" (http://icl.cs.utk.edu/~mucci/latest/pubs/Notur2009-new.pdf), presentation slides by Philip Mucci
- Programming Optimization (http://www.azillionmonkeys.com/qed/optimize.html) by Paul Hsieh

- Writing efficient programs ("Bentley's Rules") (http://www.new-npac.org/projects/cdroms/cewes-1999-06-vol1/nhse/hpccsurvey/orgs/sgi/bentley.html) by Jon Bentley
- "Performance Anti-Patterns" (http://queue.acm.org/detail.cfm?id=1117403) by Bart Smaalders

---