The advent of the zero-cost exception model, which adopted a different set of trade-offs, eliminated literally *all* runtime overhead on the hot path at the expense of larger programs and significantly increased the relative latency on the cold path — e.g, due to likely cache misses and perhaps even page faults. This newer "zero-cost" model made allowing support for exception handling practical for a wider range of applications.

The zero-cost exception model In the increasingly ubiquitous **zero-cost** model, employed by most modern compilers, the runtime cost of supporting exceptions on the normal (i.e., exception-free) path is *effectively* zero. That is to say, compared with not supporting exceptions at all, there are no additional machine instructions inserted into the hot path as result of the model itself: all of the additional object code is implemented in the form of tables and cold-path code.[37]

We say *effectively* here because there exist both theoretical and practical cases where — if the optimizer somehow knows that a particular function cannot throw — one or more machine instructions having nothing to do with the exception model itself might be able to be reordered if not entirely elided from the active instruction stream on the hot path. These forms of potential collateral optimization are characterized below.

Note that the term zero cost refers to the elimination of bookkeeping on the hot path. Support for exceptions is never truly *free*, as there must always be code generated to handle exceptions and unwind the stack, including calling destructors for local variables.

Theoretical opportunities for performance improvement

There are at least two distinct categories of runtime-performance optimization that a compiler could theoretically employ when a called function, `g`, is known by the compiler not to throw: (1) when instructions in the generated code are known to be independent, the compiler has maximum flexibility to choose and order machine instructions (**instruction selection** and **code motion**) so as to to minimize latency and take optimal advantage of the parallelism and pipelining available in modern CPUs; and (2) when a sequence of instructions is free from unpredictable branches, the optimizer can discover and elide "dead code" (**code elision**), i.e., remove instructions whose effects are never used.

As a hypothetical illustration of this first type of optimization, consider the familiar example where we have a function, `f`, having one or more local variables of type `S`, this time having a visible and substantial **default constructor** body along with a potentially opaque non-trivial destructor. Furthermore, we have a subroutine, `g`, whose implementation is opaque, currently **`noexcept(false)`** (by default), but happens to not throw. Note that, without visibility into the body of the constructor of `S`, the compiler cannot know that there is no interaction between `S` and `g`. Also note that, had the body of `g` been available, there would be no need to explicitly state that `g` does not throw (see *Annoyances — Algorithmic optimization is conflated with reducing object-code size* on page 1143):

[37] For a more detailed yet approachable introduction to the zero-cost exception model, see **mortoray13**.