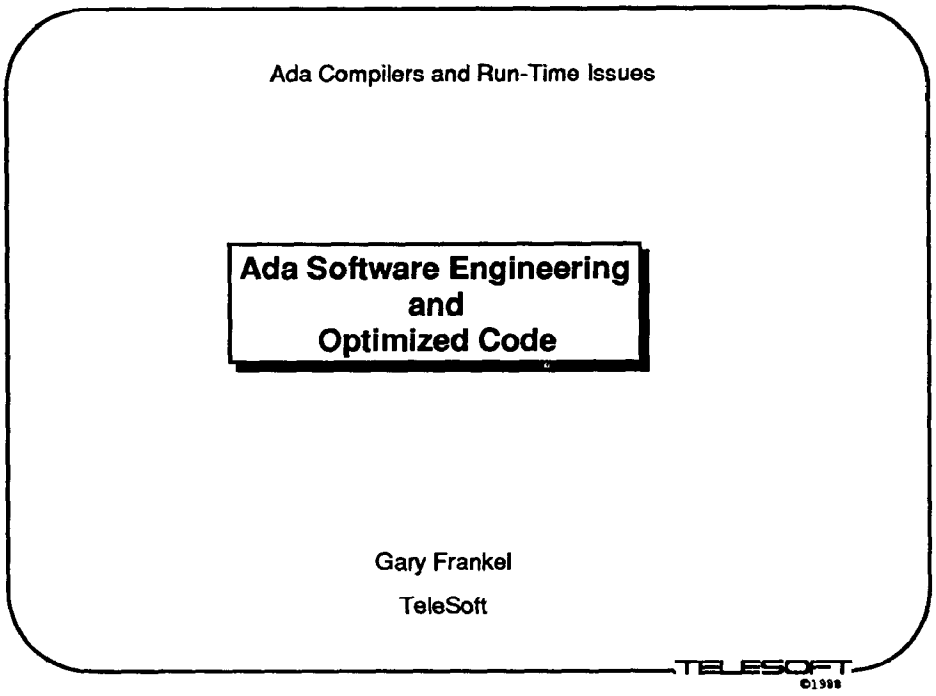
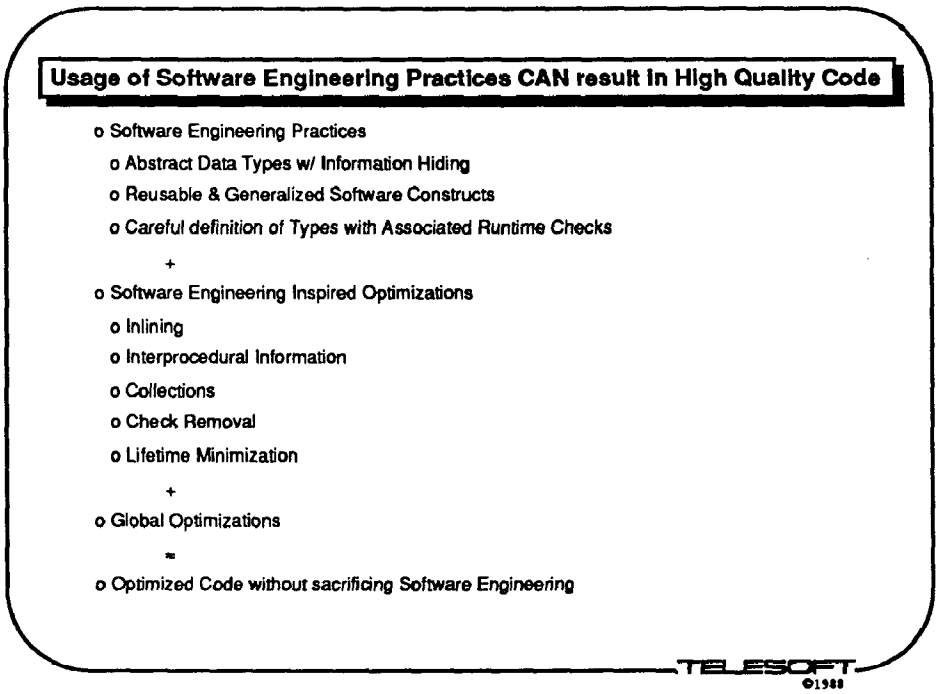
Ada Software Engineering and Optimized Code



Mr. Frankel received a B.S. from California Institute of Technology in 1974; graduating with honors. He then spent 3 years designing and implementing real-time simulators and trainers. For the last 9 years he has been engaged in systems programming including 7 years of design and development work on the TeleSoft Ada compiler and related tools.

Ada Software Engineering and Optimized Code



Ada promotes the usage of a number of software engineering practices. Historically, users believed that constructing software using these approaches resulted in poorer code and that careful low-level coding was the only way to get high quality results.

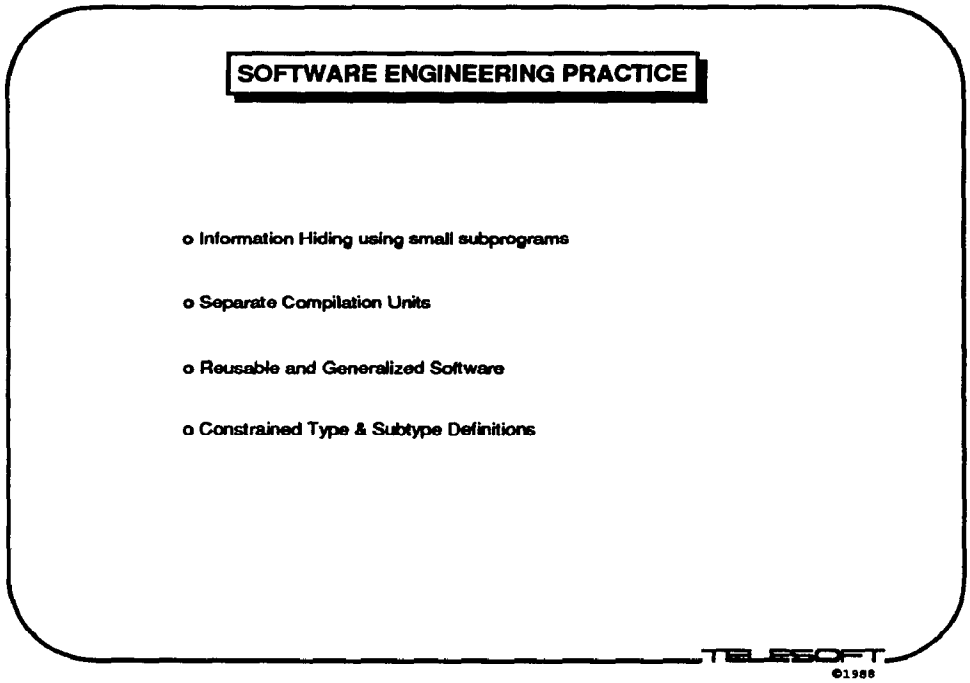
The maturation of Ada global optimizers can now provide implementations of classical global optimizations that significantly improve the output code quality. However, an Ada Optimizing Compiler must still take into account the high level software engineering practices to ensure that users can get the full power of other important optimizations.

A number of optimizations in the TeleGen2 Ada Compilation System have been specially aimed at supporting users who practice good engineering skills. These users should be encouraged, not penalized by the compiler. These optimizations allow the user to produce generalized reusable software with information hiding without being penalized. The optimizations also support the user in making extensive usage of Ada type and subtype constraints. That information is both used to remove runtime checks and to actually produce better code.

These software engineering oriented optimizations must be combined with a powerful implementation of classical global and local optimizations. Those optimizations provide the high code quality while the engineering oriented optimizations remove impediments that would block the other powerful improvements.

The combination results in a compiler which is capable of producing highly optimal code while encouraging good software engineering.

Ada Software Engineering and Optimized Code



Ada promotes the use of a number of software engineering practices. These practices include :

* Construction of abstract data types with small subprograms to provide information hiding. This approach makes it critical for the compiler to be able to minimize or remove the effect of the subprogram when the structure is only utilized for information hiding.

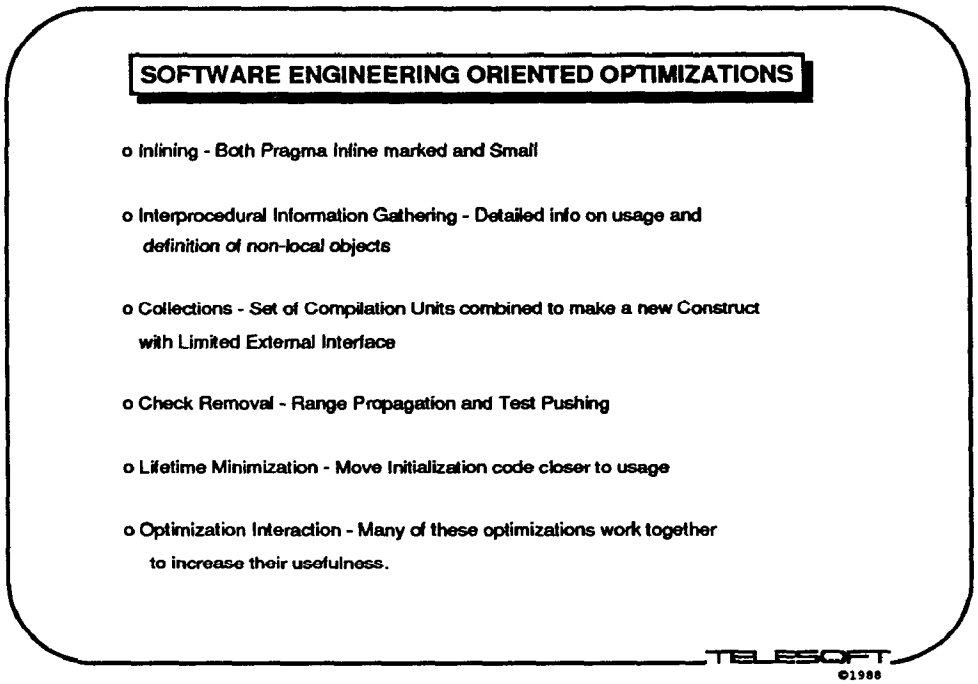
* Construction of programs in separated compilation units. This technique is clearly necessary to minimize the compilation effect of changes and modifications in program development. Yet, this separation normally works against compilers by making considerable information unavailable.

* Production of reusable general software constructs. These program units frequently handle generalized cases or multiple possible variations. A specific usage will only utilize a partial capability and the compiler needs to remove the unnecessary code.

* Extensive usage of constraints and Ada's language provided runtime checks to catch errors. Programs that provide proper assertions about the valid values of objects clearly are more robust and better defined. However, most current users either do a sloppy job of definition or turn off all checks when really using the code. The compiler should make it possible to leave the checks on and still get good quality code.

* Explicit declarations with appropriate initialization. The Ada declarations are in specific source locations in program units and this frequently places the object initialization far from its usage. Without compiler support, that can be detrimental to code quality by artificially extending the lifetime of that object.

Ada Software Engineering and Optimized Code



* Inlining - This includes not only user directed inlining, but also inlining of subprograms that are sufficiently small or called only once. Inlining is done early in the optimization process to expose the inlined code to other optimizations.

* Interprocedural Information Gathering - With the modern practice of extensive usage of subprograms and the Ada encouragement of this practice, it is critical that the compiler not have to make overly pessimistic assumptions about the effects of that call. This requires gathering information on the uses and definitions of non-local objects by subprograms so that the caller can take into account those effects.

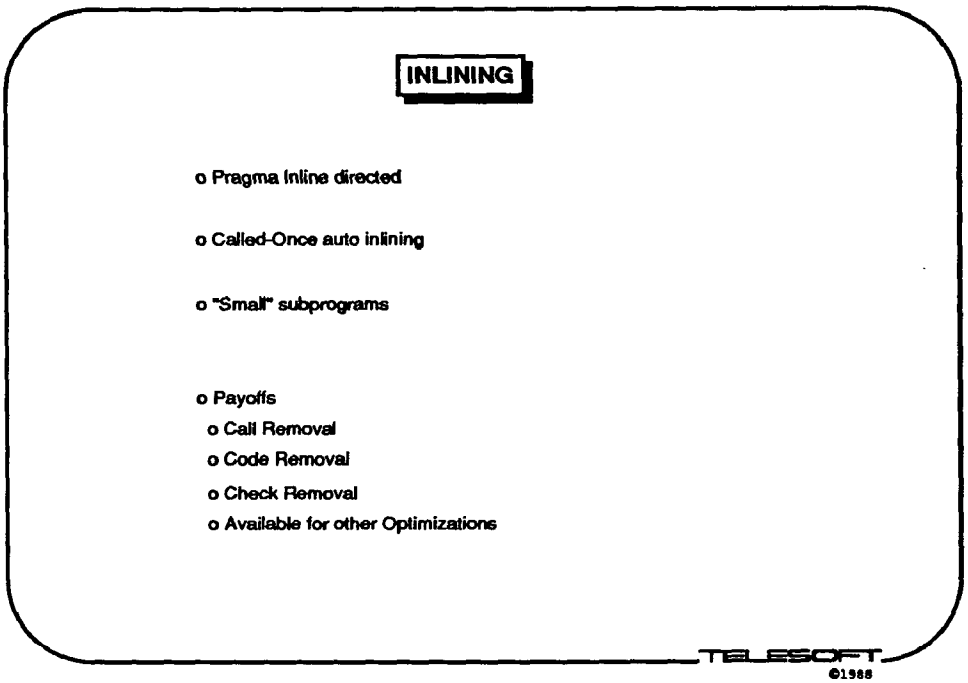
* Collections - The compiler allows the user to group a set of compilation units together and define a collection. That collection has a defined interface. The user could define an equivalent top-down construct, but frequently the bottom up build of that construct uses existing components and is easier to define. This allows the compiler to utilize the same optimizations on a larger code structure with information not available otherwise.

* Check Removal - Checks in Ada are excellent tools for debugging code and asserting the proper behavior of that code. The compiler can utilize range declarations, explicit assignments, tests, etc to analyze the possible range of objects and expressions. This allows the compiler to remove checks that are not strictly necessary to maintain program correctness.

* Lifetime Minimization - Ada declarations with initialization (especially default initialization) are frequently far removed from the usage of those declared objects. This extends the lifetime of those objects unnecessarily and worsens register utilization. Therefore, these initializations are moved as close as possible to the usage.

* Optimization Interaction - Optimizations work together in important ways. Inlining opens up code for check removal and other optimizations including removal of dead code (due to generalized constructs). Collections allow more extensive inlining and significantly improved interprocedural information. Check removal unblocks optimizations which would be prevented by those checks.

Ada Software Engineering and Optimized Code



* Inlining is the critical element of optimizations aimed at software engineering. This allows users to utilize subprograms for information hiding and to write generalized software.

* Users can direct the inlining based on their knowledge of the subprogram. The user may know that the subprogram will be small and is just there for information hiding or it may be a generalized software component where a particular instance of it will utilize only a single case.

* The compiler also inlines subprograms that are called only once. In that case, the code only needs to be present once and the call overhead is automatically removed. This situation could occur due to user structuring for readability or due to collections or due to maintenance changes.

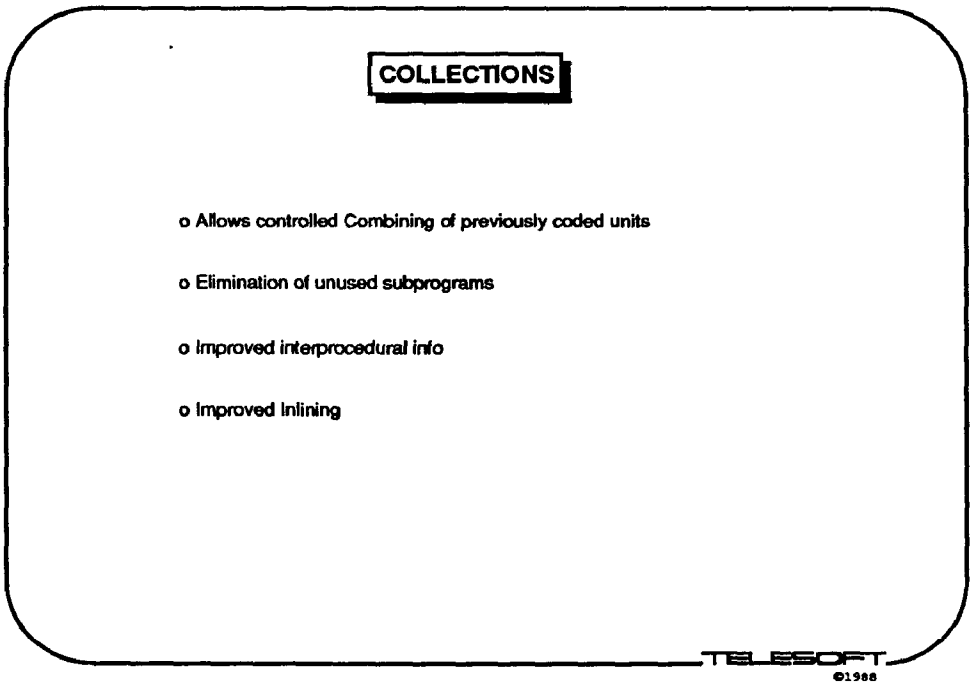
* The compiler also recognizes subprograms that are sufficiently small that the call itself consumes as much space as the actual body. This happens frequently when subprograms are used for information hiding.

* Inlining has a number of benefits. Of course, the first payoff is simply the removal of the call overhead. This overhead includes costs involved in parameter passing. This payoff occurs even without other optimizations.

* Other and often more important payoffs occur due to opening the subprogram body to other optimizations. Considerable information on ranges and values of parameters can now be utilized. This can result in removal of entire basic blocks and branches - especially for a general purpose routine.

* Check removal is also made easier in the open inlined body due to the improved range information available on the specific parameters. Information on the flow-thru effects of the subprogram body are also improved relative to the normal interprocedural information gathering.

Ada Software Engineering and Optimized Code



* A collection is a set of separate compilation units (library units and subunits) that the user wants to combine for improved optimization. The collection has a specified interface, which is normally a subset of the interfaces of the library units in the collection. This capability is usually utilized when a set of units combines to provide a specific function and the development has stabilized sufficiently to not need frequent changes to those units.

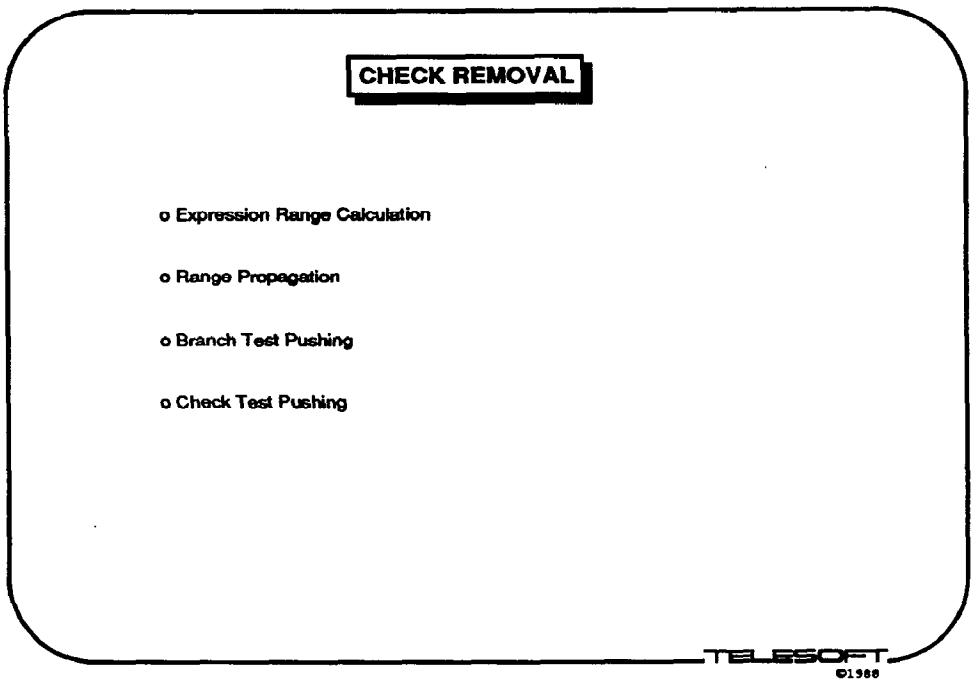
  The primary purpose of collections is to allow good development practices with a number of separately compiled units and still get the payoff of global optimization utilizing all of the available information.

* The optimizing compiler will eliminate any unused subprograms from the collection. These may be subprograms which are used in some contexts, but not in this particular collection. An example would be large parts of Text_IO when only the strings or integer portions are used.

* The available interprocedural information is significantly improved. Without collections, the compiler must make worse case assumptions about any external units. For units whose interface is not visible outside the collection, this is no longer true. The compiler can utilize information on items accessed by those subprograms and also can provide data on what parameters are actually used to call the subprogram.

* In a collection, the optimizing compiler can now inline routines that were previously external and that are sufficiently small or called only once. In other words, the primary benefit of collections is to enable other optimizations which were blocked by the separate compilation.

Ada Software Engineering and Optimized Code



* Ada checks are a powerful tool in helping debug programs. The checks form implicit assertions on the programmers part of how the values of the program should behave. If the user has explicitly checked a particular assertion, then the compiler should remove any checks which further test that same assertion. Similarly, the compiler should remove checks where the possible value set of the object or expression tested cannot violate the assertion. Check removal is accomplished by careful analysis of the range of the value being checked. If that value passes the check assertion, then the check is removed. Check removal is critical both for basic code improvement and for unblocking other optimizations.

* The first step in check removal is the calculation of the possible range of an expression (particularly numeric expressions). The more refined the user subtypes are, the more accurate the expression range will become. The expression range can be used to remove checks and to further refine the range of any object which the expression value is assigned to. The propagated range of objects also improves the expression range calculation (ex. an object with range 1..100 but whose only reaching value is an initialization of 5).

* Range propagation is accomplished by tracking the possible values of an object in linear code and merging possible incoming values in branch situations. Obviously, good subtype constraints on objects initially helps the compiler do a better job of check removal. It can also help actually generate better code in some cases. Consistent usage of a subtype also makes it easier to remove checks. For example, some benchmarks are very poor in this area and actually make it impossible to remove the checks where a simple proper subtype on a formal parameter would solve the problem.

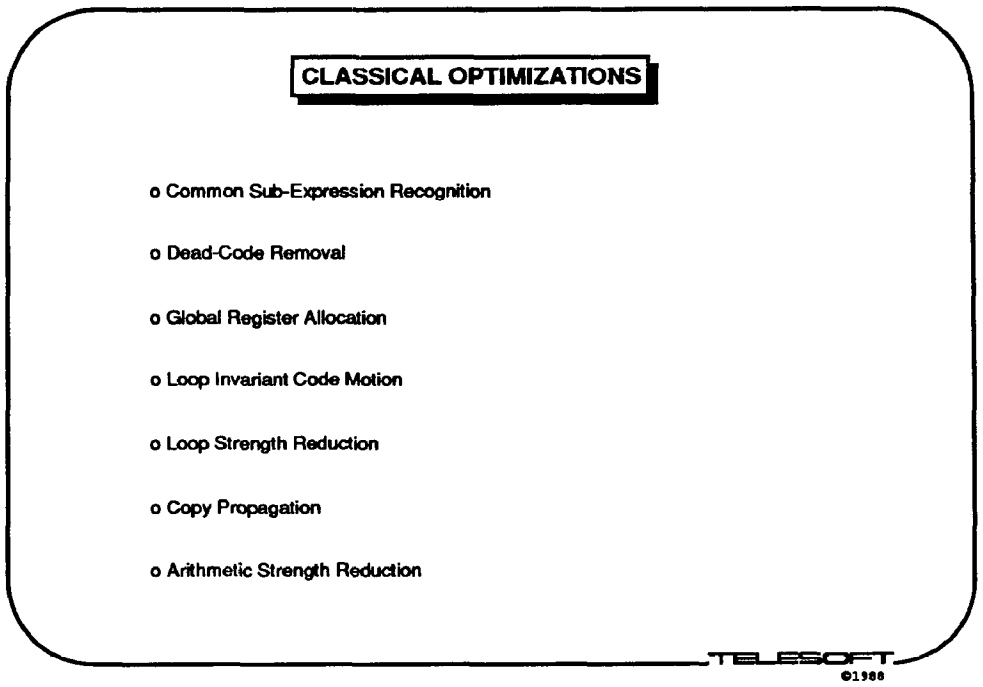
* Branch test pushing occurs when an object is tested at some branch point (ex. an if test).

```
if x < 5 then
  z := x;
else
  y := x - 3;
end if;
```

On the true branch, x has a value bounded with a max of 5, while the min is 6 on the false branch. This contributes to check removal on the branches and expression range calculations. The user has often put in this test as a specific assertion and that allows the compiler to remove any checks that test that same characteristic.

* Check test pushing occurs when a check has already been made on a specific value or object for a particular assertion. Further checks on that assertion are removed. The prime example is a null access check. Once a single test has verified that the access value is not null, no other checks are necessary that test for null.

Ada Software Engineering and Optimized Code



* CSE Recognition - The compiler recognizes common sub expressions that compute the same value and replaces subsequent evaluations with usage of the first calculation of the expression. This replacement is controlled by a target dependent cost since CSEs may be better recalculated depending on the machine and the context.

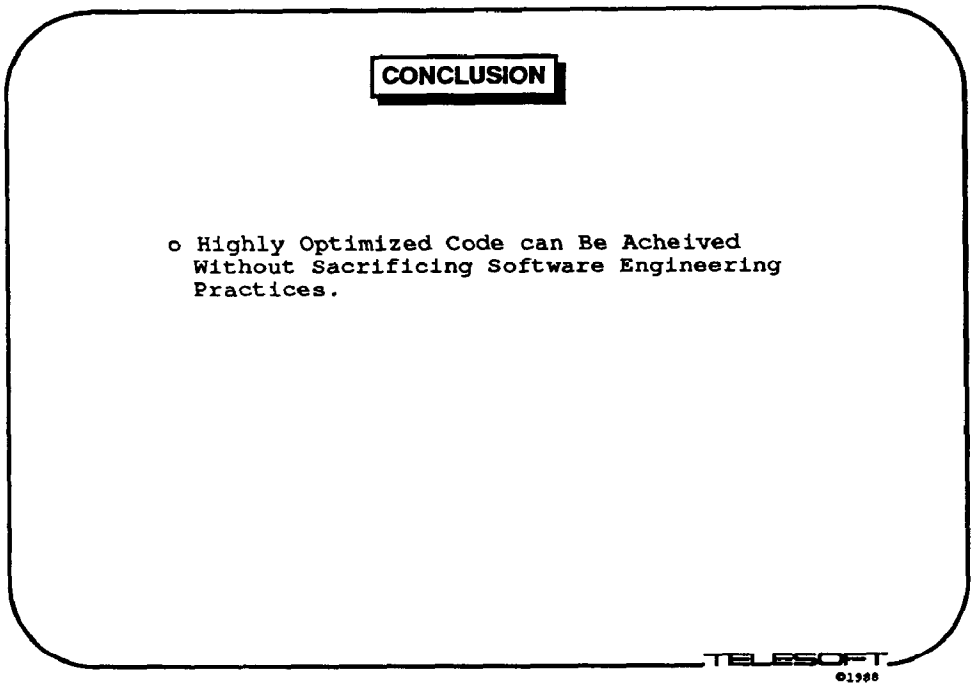
* Dead-Code Removal - Entire basic blocks can be dead code in a given program. This can occur due to range propagation eliminating a code branch or the inlining of a generalized software unit where a set of cases is not used in that particular context. Dead statements can also occur when the target of an assignment is never utilized. This is a frequent occurrence in long-term maintenance of code.

* Global Register Allocation - The quality of the register allocation is a dominant factor in making all other optimizations payoff. Global allocation is performed for the entire subprogram using a priority based graph coloring approach.

* Loop Invariant Code Motion - Loops are the key component in speed improvements in most code. The compiler moves any expressions and statements in a loop that are the same on all passes of the loop. In nested loops, these are moved as far out in the loop structure as possible.

* Loop Strength Reduction - Arithmetic operations inside loops (including addressing arithmetic) that depend on the loop induction variables can frequently be improved. This improvement "reduces" a more expensive operation such as a multiply to an add. The common examples of this optimization are in array indexing inside loops, but other cases also occur and can be handled uniformly.

Ada Software Engineering and Optimized Code



An optimizing compiler that does a quality job on classical optimizations and has optimizations that strongly support Ada software engineering practices can and does produce highly optimized code. That combination allows the user to produce well-designed code and achieve the required efficiency.